C++内存管理教学题：编写深拷贝容器

目标

基于我们给的代码框架，编写一个容器MyContainer，用该容器维护一个堆内存上的数组

该内存容器是一个典型的RAII容器，通过这个练习学习如何使用RAII来安全管理资源

框架代码概要

我们提供了一个代码框架

- 私有成员变量
  - `int *_data`：内容为堆内存上的某个地址
  - `int size`：内容为数组的长度
- 类的静态成员变量
  - `int _count` ：用于记录当前共创建了多少个MyContainer实例
- 下列函数各有不同的行为和职责，但都需要维护`_count` 来记录当前堆上共创建了多少实例
  - 构造函数行为：参数为int类型的变量size，在堆上构建一个int类型的长度为size的数组
  - 析构函数行为：销毁分配的堆内存
  - 拷贝构造函数行为：对堆上的数据进行深拷贝（拷贝数组中的每个成员）
  - 拷贝赋值函数行为：对堆上的数据进行深拷贝（拷贝数组中的每个成员）

测试代码

- 包括main函数和对应的test_*()测试用例
- 测试内容为当前创建了多少个MyContainer实例

练习要求

- **完成TODO标注的函数**
  - 注意维护好`_count`变量
  - 注意处理自赋值的情况
  - 注意处理好静态变量的声明和定义
- 提交要求
  - **请不要修改main函数和测试代码！**可能会影响后台用例的判定
  - 不要投机取巧！
    - 助教会人工检查运行行为异常的代码提交，并将本次练习记录为0分
- 测试样例

```
void test(){
    MyContainer m(5);
    std::cout << m.count() << std::endl;
    MyContainer m2(m);
    std::cout << m2.count() << std::endl;
    MyContainer m3 = m2;
    std::cout << m3.count() << std::endl;
}
//正确输出结果
1
2
3
```

练习之外（不作为练习，仅供扩展学习）

- 理解代码中的一些细节
    - 构造函数为什么会使用explicit关键字进行标注
        - 如果不使用explicit，对于`MyContainer m = 10`，编译器会进行隐式类型转换，此时程序的行为可能不符合我们预期
        - 有的时候利用explicit的特性可以帮助我们简化代码，但可能会对可读性造成影响
    - 成员变量定义时为什么加上`{}`
        - 这是一个好习惯，可以防止一些因未初始化问题导致的难以分析的bug
- 可以尝试在构造函数、拷贝构造、拷贝赋值中插入打印语句，查看下列代码的输出

```
MyContainer get(){
  MyContainer m {1};
  return m;
}
int main(){
  MyContainer m = get();
  return 0;
}
```

    - 可以先猜测一下共输出多少语句，再运行程序
        - 拷贝了1次？2次？3次？
        - 我在x86-64 gcc 7.5上用-O0优化的输出结果中，并没有任何拷贝的发生，只有一次构造和一次析构
    - 如果实际输出的结果比你预想的要少，可以查看以下链接进一步了解

- https://stackoverflow.com/questions/12953127/what-are-copy-elision-and-return-value-optimization

附录：代码框架

```
#include <iostream>

class MyContainer {
public:
    MyContainer(int size) : _size(size) {
        // TODO
    }

    ~MyContainer() {
        // TODO
    }

    MyContainer(const MyContainer &Other) {
        // TODO
    }

    MyContainer &operator=(const MyContainer &Other) {
        // TODO
        return *this;
    }

    int size() const {
        return _size;
    }

    int* data() const {
        return _data;
    }

    static int count() {
        return _count;
    }

    static int _count;

private:
    // C++11 引入的 initializer_list
    int *_data{nullptr};
    int _size{0};
};

int MyContainer::_count = 0;
```

```
void test1(){
    MyContainer m(5);
    std::cout << m.count() << std::endl;

    MyContainer m2(m);
    std::cout << m2.count() << std::endl;

    MyContainer m3 = m2;
    std::cout << m3.count() << std::endl;
}

void test2(){
    MyContainer m1(5);
    std::cout << m1.count() << std::endl;

    MyContainer m2 = m1;
    std::cout << m2.count() << std::endl;
    std::cout << (m2.data() == m1.data()) << std::endl;
}

void test3(){
    MyContainer m1(3);
    std::cout << m1.count() << std::endl;

    MyContainer m2 = m1;
    std::cout << m2.count() << std::endl;
    std::cout << (m2.data() == m1.data()) << std::endl;

    m1 = m2;
    std::cout << m1.count() << std::endl;
    std::cout << (m2.data() == m1.data()) << std::endl;

    m2 = m1;
    std::cout << m2.count() << std::endl;
    std::cout << (m2.data() == m1.data()) << std::endl;

    int * prev_ptr = m1.data();
    m1 = m1;
    std::cout << m1.count() << std::endl;
    std::cout << (m1.data() == prev_ptr) << std::endl;

}

void test4(){
    MyContainer m1(3);
    std::cout << m1.count() << std::endl;

    {
```

```
        MyContainer m2 = m1;
        std::cout << m2.count() << std::endl;
        std::cout << (m2.data() == m1.data()) << std::endl;

        m1 = m2;
        std::cout << m1.count() << std::endl;
        std::cout << (m2.data() == m1.data()) << std::endl;

        m2 = m1;
        std::cout << m2.count() << std::endl;
        std::cout << (m2.data() == m1.data()) << std::endl;

    }

    std::cout << m1.count() << std::endl;
}


int main(){
    test1();
    test2();
    test3();
    test4();
    return 0;
}
```