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`int`类型的数组

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，你需要复制框架代码中的`MyContainer.h`和`Main.cpp`，并补全`MyContainer.cpp`文件中的内容。下面是`MyContainer`类的介绍：

- 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`MyContainer(int size)`：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`~MyContainer()`：销毁分配的堆内存
  - 拷贝构造函数行为`MyContainer(const MyContainer &Other)`：
    - 创建类对象，在为其分配内存空间的同时，利用参数初始化成员变量
  - 赋值重载函数行为`MyContainer &operator=(const MyContainer &Other)`：
    - 重写赋值操作符"=", 对堆上的数据进行**深拷贝**
    - 与拷贝构造类似，通过传入的对象引用对成员变量进行赋值，但不属于构造函数

练习要求

- **完成TODO标注的函数**
  - 注意维护好`_count`变量
  - 注意处理赋值函数自赋值的情况
  - 注意处理好静态变量的声明和定义
  - 注意拷贝构造函数和赋值重载函数需要对数据进行深拷贝，而不是复制指针，否则在调用析构函数时可能会导致重复的内存释放
- 提交要求
  - **请不要修改Main.cpp**（好消息：`Main.cpp`锁定功能已经上线，你在自测或提交测试时Main.cpp会自动替换成框架代码）
  - 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。**

代码框架

MyContainer.h

```
#ifndef MYCONTAINER_H
#define MYCONTAINER_H
```

```
#include <iostream>

class MyContainer {
public:
    MyContainer(int size);
    ~MyContainer();
    MyContainer(const MyContainer &Other);
    MyContainer &operator=(const MyContainer &Other);

    int size() const;
    int* data() const;
    static int count();

private:
    int *_data{nullptr};
    int _size{0};
    static int _count;
};

#endif // MYCONTAINER_H
```

MyContainer.cpp

```
#include "MyContainer.h"

int MyContainer::_count = 0;

MyContainer::MyContainer(int size) : _size(size) {
    // TODO: finish me
}

MyContainer::~MyContainer() {
    // TODO: finish me
}

MyContainer::MyContainer(const MyContainer &Other) : _size(Other._size) {
    // TODO: finish me
}

MyContainer& MyContainer::operator=(const MyContainer &Other) {
    // TODO: finish me
}

int MyContainer::size() const {
    return _size;
}

int* MyContainer::data() const {
    return _data;
}

int MyContainer::count() {
    return _count;
}
```

测试文件

Main.cpp

```
#include "MyContainer.h"
#include <cassert>
#include <functional>
#include <iostream>

// === TEST_CASES ===
void TEST_1();
void TEST_2();
void TEST_3();
void TEST_4();
void TEST_5();

#define REGISTER_TEST_CASE(name) {#name, name}

int main() {
  std::unordered_map<std::string, std::function<void()>>
      test_functions_by_name = {
          REGISTER_TEST_CASE(TEST_1),
          REGISTER_TEST_CASE(TEST_2),
          REGISTER_TEST_CASE(TEST_3),
          REGISTER_TEST_CASE(TEST_4),
          REGISTER_TEST_CASE(TEST_5),
      };

  std::string test_case_name;
  std::cin >> test_case_name;
  auto it = test_functions_by_name.find(test_case_name);
  assert(it != test_functions_by_name.end());
  auto fn = it->second;
  fn();
  return 0;
}

void TEST_1() {
  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 TEST_2() {
  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 TEST_3() {
  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 TEST_4() {
  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;
}

void TEST_5() {
  int x, y;
  std::cin >> x >> y;
  MyContainer m1(x);
  {
    std::cout << m1.count() << std::endl;
    MyContainer m2(y);
    std::cout << m1.count() << std::endl;
    std::cout << m1.size() << " " << m2.size() << std::endl;

    m2 = m1;
    std::cout << m1.count() << std::endl;
    std::cout << m1.size() << " " << m2.size() << std::endl;
    int *prev_ptr = m2.data();

    MyContainer m3(x + y);
    std::cout << m2.count() << std::endl;
```

```
    m2 = m2;
    std::cout << m2.count() << std::endl;
    std::cout << m1.size() << " " << m2.size() << std::endl;
    std::cout << (m2.data() == prev_ptr) << std::endl;
    std::cout << (m2.data() == m1.data()) << std::endl;
    m1 = m3;
  }
  std::cout << m1.count() << std::endl;
  std::cout << m1.size() << std::endl;
}
```

- 在标注输入中输入`TEST_1`进行测试，只支持测试`TEST_1`到`TEST_4`

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