C++内存管理教学：编写引用计数的共享内存容器

目标

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

这个堆内存上的对象可被一个或多个SharedContainer所共享，当没有SharedContainer持有这个对象的话，则销毁这个对象

框架代码概要

我们提供了一个代码框架

- class Content
  - `int id`：用于指示当前内存块的序号
  - `char data[1024]`：无具体意义，只是用来表示Content是一个较大的内存块
  - 调用构造和析构函数时会输出相关信息，包括内存块的id
- class SharedContainer
  - `Content *_data`：表示该容器维护的Context对象实例
  - `_ref_count`：
    - 表示当前Context对象实例被多少SharedContainer实例所共享
    - 数据类型未定，需要自己设计
  - 构造函数：将参数的`mem_id`作为Content对象的id，创建新的内存块和引用计数器
  - 析构函数：调整引用计数器，若当前Content对象没有被共享，则删除对象实例
  - 拷贝构造：调整引用计数器，共享Content对象
  - 拷贝赋值：同拷贝构造

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SharedContainer拥有的Context对象被多少SharedContainer实例共享
  - SharedContainer创建和销毁的输出结果
- 测试样例

```
void test(){
    SharedContainer m1(1);
    SharedContainer m2 = m1;
    SharedContainer m3(m2);
    std::cout << m1.get_count() << std::endl;
```

```
}
//正确输出结果
create 1
3
destroy 1
```

练习要求

- **完成TODO标注的函数**
  - 注意设计并维护好_ref_count变量，本题不考虑多线程的情况
  - 注意处理自赋值的情况
- 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

练习之外（不作为练习，仅供扩展学习）

- 思考如何扩展本练习中的共享内存容器，以支持对任意类型内存的共享
  - 请参考shared_ptr的基本原理，可能需要一些模板编程的知识
- 有了shared_ptr，我们是不是可以只需要创建资源，剩下的都交给shared_ptr管理
  - shared_ptr可能产生循环引用而导致的内存泄漏
  - shared_ptr的额外性能开销
  - 标准库的shared_ptr不是线程安全的
- shared_ptr既然能清理不被使用的内存，那么垃圾收集又是什么？
  - 前者回收资源是eager的；后者回收资源是lazy的
  - 前者有循环引用问题；后者没有
  - ...

附录：代码框架

```
#include <iostream>
#include <string>

class Content {
public:
    explicit Content(int id) : id(id) {
        std::cout << "create " << std::to_string(id) << std::endl;
    }

    ~Content() {
        std::cout << "destroy " << std::to_string(id) << std::endl;
    }
```

```
private:
    int id{-1};
    char data[1024]{};
};

class SharedContainer {
public:
    //TODO
    explicit SharedContainer(int mem_id);
    //TODO
    ~SharedContainer();
    //TODO
    SharedContainer(const SharedContainer &other);
    //TODO
    SharedContainer& operator=(const SharedContainer &other);
    //TODO
    int get_count() const;

    SharedContainer(const SharedContainer &&) = delete;
    SharedContainer &operator=(const SharedContainer &&) = delete;

private:
    Content *_data{nullptr};
    //TODO: design your own reference counting mechanism
};

void test1(){
    SharedContainer m1(1);
    SharedContainer m2 = m1;
    SharedContainer m3(m2);
    std::cout << m1.get_count() << std::endl;
    std::cout << m2.get_count() << std::endl;
    std::cout << m3.get_count() << std::endl;
}

void test2(){
    SharedContainer m1(1);
    SharedContainer m2 = m1;
    m1 = m1;
    {
        SharedContainer m3 = m1;
        std::cout << m1.get_count() << std::endl;
    }
    std::cout << m1.get_count() << std::endl;
    std::cout << m2.get_count() << std::endl;
}

void test3(){
```

```
    SharedContainer m1(1);
    SharedContainer m2(2);
    m1 = m2;
    std::cout << m1.get_count() << std::endl;
    std::cout << m2.get_count() << std::endl;
    {
        SharedContainer m3(3);
        m1 = m3;
        std::cout << m1.get_count() << std::endl;
        std::cout << m2.get_count() << std::endl;
        std::cout << m3.get_count() << std::endl;
    }
    std::cout << m1.get_count() << std::endl;
    std::cout << m2.get_count() << std::endl;

}

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