题目：

公司 A 准备开发这样一种设计软件，用户可以向一个平面上添加多个矩形，针对每个矩形，用户可以赋予不同的颜色，如下图所示。在用户操作的时候，软件会后台记录日志。公司需要通过日志分析知道用户编辑完成之后，平面上每个矩形的颜色是什么，请你写一个程序帮他们**逐步实现**这个功能。

程序输入的基本格式为

输入内容	解释

N
Add normal P1
…
Normal
	//表示接下来会有 N 条操作日志**(N<100)**
	//一条操作命令
	//省略 N-1 命令
	//指定输出的格式**（此条命令不包含在 N 条操作日志中）**

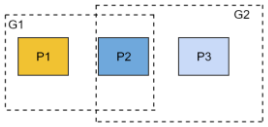
基本需求：用户会依次向平面添加普通的矩形。每个矩形拥有一个编号来标志身份，同时拥有一个颜色属性来标记自身的颜色，颜色用 RGB 的方式表示。在矩形被加入到平面时，会被指定唯一的 polygonid（由 P+数字组成，P1），矩形默认为黑色，对应的 RGB 表示为（0,0,0）。用户会向平面中指定的矩形填充颜色，如将编号 Pi 的矩形的颜色修改为（255,0,0）。请你在用户修改完成之后，按照矩形的**编号中的数字部分从小到大的顺序**输出矩形的**编号和颜色**。

衍进需求 1：用户会朝平面添加 3 种矩形，第一种是**普通矩形**，没有特别之处，第二种是**反色矩形**，如果用户为其填充颜色（X,Y,Z），那么矩形颜色会被修改成（255-X,255-Y,255-Z），第三种是**单红色矩形**，如果用户为其填充颜色（X,Y,Z），那么矩形的颜色会被修改为（X,0,0）。(反色矩形和单红色矩形初始都为黑色)

衍进需求 2：可以选择**按照多表形灰度值从小到大的顺序**输出矩形的编号和颜色，如果灰度值相同，那么按照编号中数字部分从小到大的顺序输出编号和颜色。由 RGB 计算灰度值的公式如下。

$$Gray = R*0.299 + G*0.587 + B*0.114$$

衍进需求 3：用户觉得一个个修改颜色太过麻烦，用户会选择现将一部分矩形编辑成一个组，并且会为组指定一个 groupid（由 G+数字组成，如 G1），用户可以直接指定矩形的编号来赋予颜色，或者指定 groupid 来给所有属于这个 Group 的矩形来赋予颜色。(同一个矩形可能会属于多个组，如下图)



日志中操作命令说明：

操作命令格式	
Add normal/single/reverse ploygonid	添加一个普通/单色/反色矩形，将其编号为 ploygonid
Group N index1 index2 ... indexN groupid	**N 代表后面编号的个数**，将编号为 index1 到 indexN 的矩形编为一个组，小组的编号为 groupid
Set index red green blue	将编号为 index 的对象颜色修改为（red,green,blue）
Normal/Gray	按照矩形编号数字部分/灰度值顺序输出

输入输出样例:

输入内容	解释

3
Add normal P2
Set P2 255 0 0
Add normal P1
Normal
	//接下来有 3 条操作日志
	//添加一个普通矩形，编号为 P2
	//将 P2 的颜色修改为(255,0,0)
	//添加一个普通矩形，编号为 P1
	//按照编号数字部分大小顺序输出

输出	
P1 0 0 0 P2 255 0 0	

输入内容	解释

7
Add single P2
Set P2 255 0 0
Add normal P1
Group 2 P2 P1 G1
Add reverse P3
Group 2 P1 P3 G2
Set G2 0 255 0
Gray
	//接下来有 7 条操作日志
	//添加一个单色矩形，编号为 P2
	//将 P2 的颜色修改为(255,0,0)
	//添加一个普通矩形，编号为 P1
	//将 P2 与 P1 归为一个组，组的编号为 G1
	//添加反色矩形，编号为 P3
	//将 P1 和 P3 编为一组，组的编号为 G2
	//将 G2 中所有矩形的颜色修改为（0,255,0）
	//按照灰度值大小顺序输出

输出	
P2 255 0 0 P3 255 0 255 P1 0 255 0	

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;
}
```

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>

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;
}
```

内存管理

> 主要考察析构函数

由于 libc 原生的内存管理 API 的种种问题（例如，内存泄露和 double-free），某些数据库管理系统基于这些 API 实现了新的动态内存管理机制。具体地，这些实现中引入了一个称为“memory context”的概念（后文简称为“MC”），动态内存管理不再直接使用 `malloc` / `free` 等函数，而是调用某个 MC 对象提供的 API 进行。这些 MC 对象被组织为树状结构，如图 1 所示。这样，使用者不再需要为之前分配的每个内存块调用 `free` 等函数，而是等到相应的 MC 对象被销毁时统一地归还其下的动态内存。同时，由于树状结构的存在，在销毁较为高层的 MC 对象时，处于低层次的 MC 对象也同样会被销毁。这种机制极大降低了管理动态内存的心智负担，更好地避免了内存泄露和 double-free 等问题。

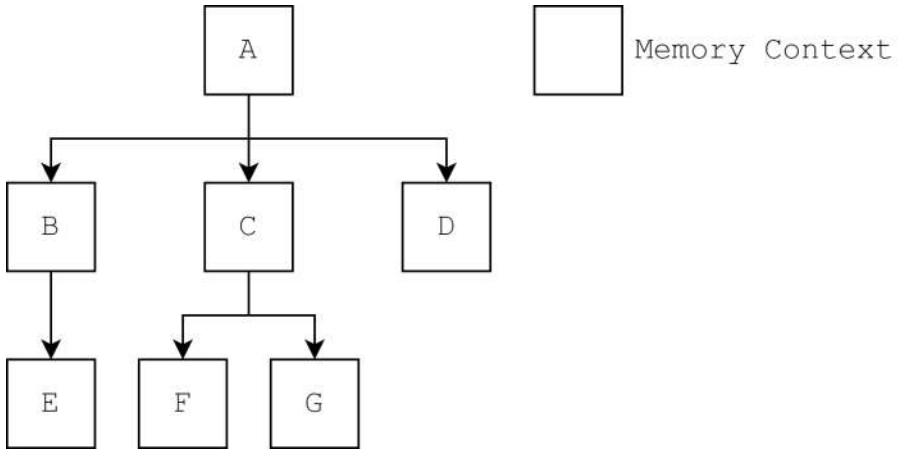


图 1：MC 的树状结构

我们提供了骨架代码，请填充其实现。需要实现如下的功能：

- 维护 MC 对象之间的父子关系
- 给定一个 ID，分配一块内存并将二者关联起来
- 在析构函数中进行上述的销毁工作

输入描述

本题不需要处理输入。

输出描述

销毁某个 ID 为 `X` 的内存块时，输出

```
Chunk X freed.
```

对于一个 MC 下的内存块，按照**分配顺序的逆序**（即，后进先出）的顺序进行销毁。

如果一个 MC 有多个子 MC，那么按照创建这些子 MC 的顺序进行销毁。

示例

示例 1

MC 结构和操作序列

图 2：示例 1 的 MC 结构

操作序列为：

1. A.alloc("1")
2. A.alloc("2")
3. A.alloc("3")

输出

```
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

示例 2

MC 结构和操作序列

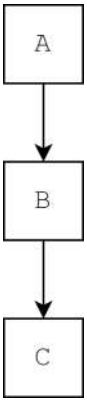


图 3：示例 2 的 MC 结构

操作序列为:

1. a.alloc("1")
2. a.alloc("2")
3. a.alloc("3")
4. b.alloc("1/1")
5. b.alloc("1/2")
6. b.alloc("1/3")
7. c.alloc("1/1/1")
8. c.alloc("1/1/2")
9. c.alloc("1/1/3")

输出

```
Chunk 1/1/3 freed.
Chunk 1/1/2 freed.
Chunk 1/1/1 freed.
Chunk 1/3 freed.
Chunk 1/2 freed.
Chunk 1/1 freed.
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

示例 3

MC 结构和操作序列

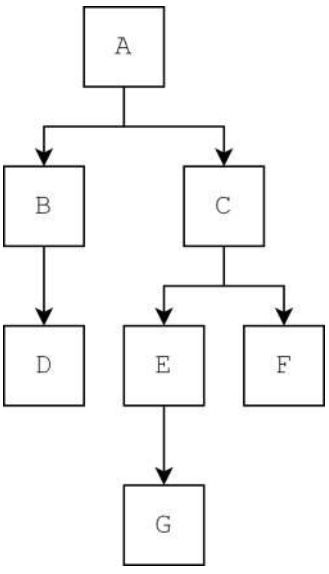


图 4：示例 3 的 MC 结构

操作序列：

1. a.alloc("1")
2. a.alloc("2")
3. a.alloc("3")
4. b.alloc("1/1")
5. c.alloc("1/2")
6. d.alloc("1/1/1")
7. d.alloc("1/1/2")
8. g.alloc("1/2/1/1")

输出

```
Chunk 1/1/2 freed.
Chunk 1/1/1 freed.
Chunk 1/1 freed.
Chunk 1/2/1/1 freed.
Chunk 1/2 freed.
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

面向对象：

1.定义棋盘类和棋子类

2.棋盘为 n*n 的方阵，棋子分为 X 和 O 两种

3.两名棋手各执一种棋子，轮流下棋，规定执 O 棋子的先手

4.获胜条件为：当一方在横向、纵向或者斜向（4 种方向中的任意一种即可）有 m 个棋子连城一排，即为胜利（有出现平局的可能）

5.输入为每次下棋的坐标，输出为结果（O Success、X Success 和 Dogfall3 种，每次的输入必在这 3 种结果中）

6.平局一定是下满了棋盘，而不会要求判断在哪一步时就已经注定平局

7.样例：

输入：

3 3　　前一个 3 表示棋盘是 3x3，后一个 3 表示 3子连成一线才算获胜

0 0　　表示 O 落子坐标为（0，0）

0 1　　表示 X 落子坐标为（0，1）

1 0

0 2

2 0

输出：

O Success　　O 后有一个空格，最后没有换行符