Solution3

191220008 陈南疃

概念题

1、什么时候需要定义析构函数?

一般情况下，类中不需要自定义析构函数，但如果对象创建后，自己又额外申请了资源（如：额外申请了内存空间），则可以自定义析构函数来归还它们。

2、什么时候会调用拷贝构造函数？使用默认的拷贝构造函数有什么需要特别注意的情况?

在创建一个对象时，若用另一个同类型的对象对其初始化，将会调用对象类中的拷贝构造函数。

具体有三种如下情况：

①创建对象时显式指出。

②把对象作为值参数传给函数时。

③把对象作为函数的返回值时。

一般情况下，编译程序提供的隐式拷贝构造函数的行为足以满足要求，类中不需要自定义拷贝构造函数。但在一些特殊情况下，必须要自定义拷贝构造函数，否则，将会产生设计者未意识到的严重的程序错误。

它带来的问题是：

①如果对一个对象（s1或s2）操作之后修改了这块空间的内容，则另一个对象将会受到影响。如果不是设计者特意所为，这将是一个隐藏的错误。

②当对象s1和s2消亡时，将会分别去调用它们的析构函数，这会使得同一块内存区域将被归还两次，从而导致程序运行错误。

③当对象s1和s2中有一个消亡，另一个还没消亡时，则会出现使用已被归还的空间问题！

解决上面问题的办法是在类中显式定义一个拷贝构造函数。

3、请说明C++中 const 和 static 关键词的作用。

const:

1、const修饰普通类型的变量

2、const 修饰指针变量

3、const参数传递和函数返回值

4、const修饰类相关

(1)const修饰成员变量

const修饰类的成员变量，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。

(2)const修饰成员函数

const修饰类的成员函数，则该成员函数不能修改类中任何非const成员函数。一般写在函数的最后来修饰。

a. const成员函数不允许修改它所在对象的任何一个数据成员。

b. const成员函数能够访问对象的const成员，而其他成员函数不可以。

(3)const修饰类对象/对象指针/对象引用

a. const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。

b. const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。

static:

1、修饰全局变量时，表明一个全局变量只对定义在同一文件中的函数可见。

2、修饰局部变量时，表明该变量的值不会因为函数终止而丢失。

3、修饰函数时，表明该函数只在同一文件中调用。

c++独有：

4、修饰类的数据成员，表明对该类所有对象这个数据成员都只有一个实例。即该实例归所有对象共有。

5、用static修饰不访问非静态数据成员的类成员函数。这意味着一个静态成员函数只能访问它的参数、类的静态数据成员和全局变量。

4、简述C++友元的特性以及其利弊。

特性：指定某些与一个类密切相关的、又不适合作为该类成员的程序实体直接访问该类的非public成员，这些程序实体称为该类的友元。友元不是一个类的成员，具有不对称性，不具有传递性。

利：提高在类的外部对类的数据成员的访问效率。

弊：破坏数据封装和数据隐藏，破坏了类的封装性。

编程题

1、小明编写了一段程序，实现了一个商品类Merchandise，每个商品有其名字name，并希望通过静态成员MerchandiseCnt记录创建的对象数，但实现的程序中存在较多问题，请你帮他指出错误并改正。

```
#include <iostream>
#include <cstring>
using namespace std;

class Merchandise
{
    static int MerchandiseCnt;
    char *name;
public:
    Merchandise(const char *_name);
    Merchandise(const Merchandise& m)// 增加自定义拷贝构造函数
    ~Merchandise();
    char *get_name() const;
    void set_name(const char *_name) const;
};

Merchandise::Merchandise(const char *_name)
{
    name = new char[strlen(_name) + 1];
    strcpy(name, _name);
    MerchandiseCnt++;
}

Merchandise::Merchandise(const Merchandise& m)// 增加自定义拷贝构造函数
{
    delete []name;
    name = new char[strlen(m.name) + 1];
    strcpy(name, m.name);
    MerchandiseCnt++;
}

Merchandise::~Merchandise()
{
    delete []name;// 增加[]
    name = nullptr;
}

char *Merchandise::get_name() const
{
    return name;
}

void Merchandise::set_name(const char *_name)// 删去const
{
    delete []name;// 增加[]
    name = new char[strlen(_name) + 1];
    strcpy(name, _name);
}

int main()
{
    {
        Merchandise::MerchandiseCnt = 0;// 给MerchandiseCnr初始化
```

```
        Merchandise m1("phone");
        Merchandise m2(m1);
    }
    return 0;
}
```

2、定义一个元素类型为float、元素个数不受限制的集合类FloatSet，要求如下：

```
class FloatSet
{
    float *numbers;
    // 可根据需要添加其他成员变量
public:
    FloatSet();
    FloatSet(const FloatSet& s);
    ~FloatSet();
    bool is_empty() const; //判断是否为空集
    int size() const; //获取元素个数
    bool is_element(float e) const; //判断e是否属于集合
    bool is_subset(const FloatSet& s) const; //判断集合是否包含于s
    bool is_equal(const FloatSet& s) const; //判断集合是否相等
    bool insert(float e); //将元素e加入集合，成功返回true，否则返回false(e已属于集合)
    bool remove(float e); //将e从集合中删除，成功返回true，否则返回false(e不属于集合)
    void display() const; //打印集合所有元素
    FloatSet union2(const FloatSet &s) const; //计算集合和s的并集
    FloatSet intersection2(const FloatSet &s) const; //计算集合和s的交集
    FloatSet difference2(const Floatet& s) const; //计算集合和s的差
};
```

代码如下：

```
#include <iostream>
#include <cmath>
#define unit_capacity 5
using namespace std;

class FloatSet
{
    float* numbers;
    int count; //元素个数
    int capacity; //数组大小
    // 可根据需要添加其他成员变量
public:
    FloatSet();
    FloatSet(const FloatSet& s);
    ~FloatSet();
    bool is_empty() const; //判断是否为空集
    int size() const; //获取元素个数
    bool is_element(float e) const; //判断e是否属于集合
    bool is_subset(const FloatSet& s) const; //判断集合是否包含于s
    bool is_equal(const FloatSet& s) const; //判断集合是否相等
    bool insert(float e); //将元素e加入集合，成功返回true，否则返回false(e已属于集合)
```

```
    bool remove(float e); //将e从集合中删除，成功返回true，否则返回false(e不属于集合)
    void display() const; //打印集合所有元素
    FloatSet union2(const FloatSet& s) const; //计算集合和s的并集
    FloatSet intersection2(const FloatSet& s) const; //计算集合和s的交集
    FloatSet difference2(const FloatSet& s) const; //计算集合和s的差
};

FloatSet::FloatSet()
{
    count = 0;
    capacity = unit_capacity;
    numbers = new float[capacity];
}

FloatSet::FloatSet(const FloatSet& s)
{
    count = s.count;
    capacity = s.capacity;
    numbers = new float[s.capacity];
    for (int i = 0; i < s.count; i++)
        numbers[i] = s.numbers[i];
}

FloatSet::~FloatSet()
{
    count = 0;
    capacity = unit_capacity;
    delete[]numbers;
    numbers = NULL;
}

bool FloatSet::is_empty() const //判断是否为空集
{
    return count == 0;
}

int FloatSet::size() const //获取元素个数
{
    return count;
}

bool FloatSet::is_element(float e) const //判断e是否属于集合
{
    for (int i = 0; i < count; i++)
    {
        if (fabs(numbers[i] - e) < 0.00001)
            return true;
    }
    return false;
}

bool FloatSet::is_subset(const FloatSet& s) const //判断集合是否包含于s
{
    for (int i = 0; i < count; i++)
    {
        if (!s.is_element(numbers[i]))
            return false;
    }
```

```
    return true;
}

bool FloatSet::is_equal(const FloatSet& s) const //判断集合是否相等
{
    return is_subset(s) && s.is_subset(*this);
}

bool FloatSet::insert(float e) //将元素e加入集合，成功返回true，否则返回false(e已属于集合)
{
    if (is_element(e))
        return false;
    if (count == capacity)
    {
        float* temp = new float[capacity + unit_capacity];
        for (int i = 0; i < count; i++)
            temp[i] = numbers[i];
        delete[]numbers;
        numbers = temp;
        capacity += unit_capacity;
    }
    numbers[count] = e;
    count++;
    return true;
}

bool FloatSet::remove(float e) //将e从集合中删除，成功返回true，否则返回false(e不属于集合)
{
    for (int i = 0; i < count; i++)
    {
        if (numbers[i] == e)
        {
            for (int j = i; j < count - 1; j++)
                numbers[j] = numbers[j + 1];
            count--;
            return true;
        }
    }
    return false;
}

void FloatSet::display() const //打印集合所有元素
{
    for (int i = 0; i < count; i++)
        cout << numbers[i] << " ";
    cout << endl;
}

FloatSet FloatSet::union2(const FloatSet& s) const //计算集合和s的并集
{
    FloatSet x;
    for (int i = 0; i < count; i++)
        x.insert(numbers[i]);
    for (int j = 0; j < s.count; j++)
        x.insert(s.numbers[j]);
    return x;
```

```
}

FloatSet FloatSet::intersection2(const FloatSet& s) const //计算集合和s的交集
{
    FloatSet x;
    for (int i = 0; i < count; i++)
    {
        if (s.is_element(numbers[i]))
            x.insert(numbers[i]);
    }
    return x;
}

FloatSet FloatSet::difference2(const FloatSet& s) const //计算集合和s的差
{
    FloatSet x;
    for (int i = 0; i < count; i++)
    {
        if (!s.is_element(numbers[i]))
            x.insert(numbers[i]);
    }
    return x;
}
```