Assignment 3

一. 概念题

1.1 什么时候需要定义析构函数?

解：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），可以自定义析构函数来归还它们.

1.2 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?

解：当创建一个对象时，如果用一个同类型的对象对它进行初始化，则会调用拷贝构造函数，有三种情况：创建对象时显式指出、把对象作为值参数传给函数、把对象作为函数的返回值时；当有指针类的数据成员时，使用默认拷贝构造函数会使成员指针指向同一片内存区域，如果对一个对象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，对象消亡时，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，从而导致程序运行异常.

1.3 请说明C++中 const 和 static 关键词的作用.

解：const的主要作用在于保持某些值不变，const成员函数不能修改数据成员的值；static关键字的作用: 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，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，修饰函数时，表明该函数只在同一文件中调用，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，用static修饰不访问非静态数据成员的类成员函数，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.

1.4 简述C++友元的特性以及其利弊.

解：友元提供了一种普通函数或者类成员函数访问另一个类中的私有或保护成员的机制，是数据保护和数据访问效率之间的一种折衷方案，友元关系具有不对称性，不具有传递性；利：实现类之间的数据共享，减少系统开销，提高效率，使得其它的类成员函数可以直接访问该类的私有变量；弊：友元函数破坏了封装机制.

二. 编程题

2.1 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();
    Merchandise::Merchandise(const Merchandise &m); // 1. 自定义拷贝构造函数
    char *get_name() const;
    void set_name(const char *_name); // 2. 去掉const
};

int Merchandise::MerchandiseCnt = 0; // 3. 静态成员变量

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

// 4. 自定义拷贝构造函数
Merchandise::Merchandise(const Merchandise &m)
{
    name = new char[strlen(m.name) + 1];
    strcpy(name, m.name);
    MerchandiseCnt++;
}

Merchandise::~Merchandise()
{
    delete[] name; // 5. delete[]
    name = nullptr;
}

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

void Merchandise::set_name(const char *_name) // 6. 去掉const
{
    delete[] name; // 7. delete[]
    name = new char[strlen(_name) + 1];
    strcpy(name, _name);
}

int main()
{
    {
        Merchandise m1("phone");
        Merchandise m2(m1);
    }

    return 0;
}
```

2.2 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：

```
#include <iostream>

class FloatSet
{
    float *numbers;
    int size;
    int capacity;
    static const int INIT_CAPACITY = 16;
    void resize(int new_cap);

public:
    FloatSet();
    FloatSet(const FloatSet &s);
    ~FloatSet();
    bool is_empty() const;                              //判断是否为空集
    int get_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()
{
    numbers = new float[INIT_CAPACITY];
    size = 0;
    capacity = INIT_CAPACITY;
}

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

FloatSet::~FloatSet()
{
    delete[] numbers;
    numbers = nullptr;
}

void FloatSet::resize(int new_cap)
```

```
{
    float *old = numbers;
    numbers = new float[new_cap];
    for (int i = 0; i < size; ++i)
    {
        numbers[i] = old[i];
    }
    capacity = new_cap;
    delete[] old;
}

bool FloatSet::is_empty() const
{
    return size == 0;
}

int FloatSet::get_size() const
{
    return size;
}

bool FloatSet::is_element(float e) const
{
    for (int i = 0; i < size; ++i)
    {
        if (numbers[i] == e)
        {
            return true;
        }
    }
    return false;
}

bool FloatSet::is_subset(const FloatSet &s) const
{
    for (int i = 0; i < size; ++i)
    {
        if (!s.is_element(numbers[i]))
        {
            return false;
        }
    }
    return true;
}

bool FloatSet::is_equal(const FloatSet &s) const
{
    return is_subset(s) && s.is_subset(*this);
}

bool FloatSet::insert(float e)
{
    if (is_element(e))
    {
        return false;
    }
    numbers[size++] = e;
    if (size >= capacity)
```

```
    {
        resize(capacity * 2);
    }
    return true;
}

bool FloatSet::remove(float e)
{
    int pos;
    for (pos = 0; pos < size && numbers[pos] != e; ++pos)
    {
    }
    if (pos == size)
    {
        return false;
    }
    for (; pos < size - 1; ++pos)
    {
        numbers[pos] = numbers[pos + 1];
    }
    --size;
    if (capacity > INIT_CAPACITY && size <= capacity / 2)
    {
        resize(capacity / 2);
    }
    return true;
}

void FloatSet::display() const
{
    for (int i = 0; i < size; ++i)
    {
        std::cout << numbers[i] << ' ';
    }
    std::cout << std::endl;
}

FloatSet FloatSet::union2(const FloatSet &s) const
{
    FloatSet fs(s);
    for (int i = 0; i < size; ++i)
    {
        fs.insert(numbers[i]);
    }
    return fs;
}

FloatSet FloatSet::intersection2(const FloatSet &s) const
{
    FloatSet fs;
    for (int i = 0; i < size; ++i)
    {
        if (s.is_element(numbers[i]))
        {
            fs.insert(numbers[i]);
        }
    }
    return fs;
```

```
}

FloatSet FloatSet::difference2(const FloatSet &s) const
{
    FloatSet fs;
    for (int i = 0; i < size; ++i)
    {
        if (!s.is_element(numbers[i]))
        {
            fs.insert(numbers[i]);
        }
    }
    return fs;
}
```