Solution8

191220008 陈南疃

概念题

1、简述多继承的含义；在多继承中，什么情况下会出现二义性？C++是怎样消除二义性的？（请举一个简单例子配合说明）

多继承是指派生类可以有一个以上的直接基类。多继承的派生类定义格式为：

```
class <派生类名>: [<继承方式>] <基类名1>, [<继承方式>] <基类名2>, …

{ <成员说明表>

};
```

多继承带来的两个主要问题：

- 名冲突问题
- 重复继承问题

解决名冲突的办法是：基类名受限

```
class C: public A, public B
{   ......
public:
    void func()
    {   A::f(); //OK，调用A的f。
        B::f(); //OK，调用B的f。
    }
};
......
C c;
c.A::f(); //OK，调用A的f。
c.B::f(); //OK，调用B的f。
```

解决重复继承的办法是：把重复继承的类定义为虚基类

```
class A
{  int x;
    ......
};
class B: virtual public A {...};
class C: virtual public A {...};
class D: public B, public C {...};
D d;
```

2、继承和组合相较彼此有什么优缺点？你觉得它们各自适用于什么样的场景？

组合	继承
优点：不破坏封装，整体类与局部类之间松耦合，彼此相对独立	缺点：破坏封装，子类与父类之间紧密耦合，子类依赖于父类的实现，子类缺乏独立性
优点：具有较好的可扩展性	缺点：支持扩展，但是往往以增加系统结构的复杂度为代价
优点：支持动态组合。在运行时，整体对象可以选择不同类型的局部对象	缺点：不支持动态继承。在运行时，子类无法选择不同的父类
优点：整体类可以对局部类进行包装，封装局部类的接口，提供新的接口	缺点：子类不能改变父类的接口，但子类可以覆盖父类的接口
缺点：整体类不能自动获得和局部类同样的接口	优点：子类能自动继承父类的接口
缺点：创建整体类的对象时，需要创建所有局部类的对象	优点：创建子类的对象时，无须创建父类的对象

应用场景

继承：一般与特殊关系（is-a-kind-of）

组合：整体与部分的关系（is-a-part-of）

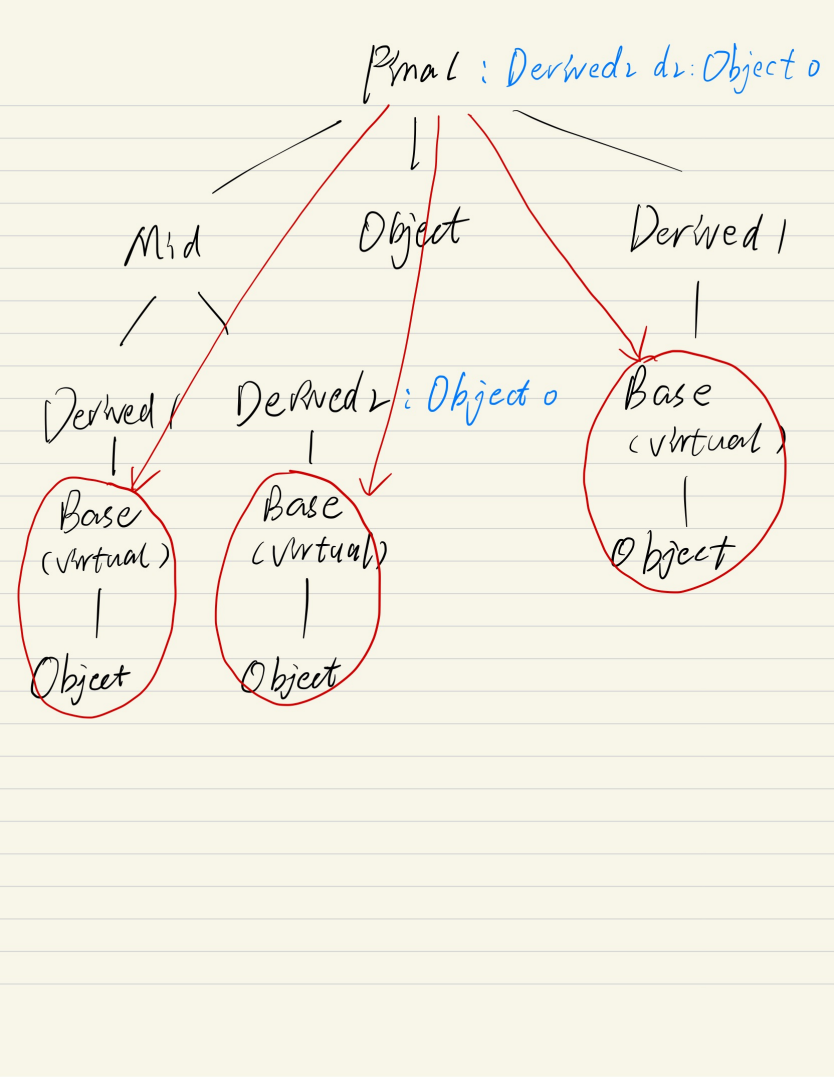
编程题

1、在以下调用中，给出类的构造和析构顺序并解释原因。

```
class Object{};
class Base: public Object{};
class Derived1: virtual public Base{};
class Derived2: virtual public Base{
private:
    Object o;
};
class Mid : public Derived1, public Derived2{};
```

```
class Final: public Mid, public Object, public Derived1{
private:
    Derived2 d2;
};
int main(){
    {
        Final f;
    }
    return 0;
}
```

类的关系：



构造顺序：

```
construct Object
construct Base
construct Derived1
construct Object
construct Derived2
construct Mid
construct Object
construct Derived1
construct Object
construct Base
construct Object
construct Derived2
construct Final
```

分析：

多重继承的情况下，严格按照派生类定义时从左到右的顺序来调用构造函数，析构函数与之相反。

但是如果基类中有虚基类的话则构造函数的调用顺序如下：

- 虚基类的构造函数在非虚基类的构造函数之前调用。
- 若同一层次中包含多个虚基类，这些虚基类的构造函数按照他们的说明顺序调用。
- 若虚基类由非虚基类派生而来，则任然先调用基类构造函数，再调用派生诶，在调用派生类的构造函数。

因此，一开始会先调用Final的间接虚基类Base的基类Object的构造函数，再调用虚基类Base的构造函数。

```
construct Object
construct Base
```

然后，按照先调用基类构造函数再调用派生类构造函数的规则，按照定义顺序，依次调用：调用Derived1的构造函数，调用Derived2的成员对象Object o的构造函数，调用Derived2的构造函数，调用Mid的构造函数，调用Object的构造函数，调用Derived1的构造函数。

```
construct Derived1
construct Object
construct Derived2
construct Mid
construct Object
construct Derived1
```

然后，Final中有成员对象Derived2 d2，调用Derived2的直接虚基类Base的基类Object的构造函数，再调用虚基类Base的构造函数，再调用Derived2的成员对象Object o的构造函数，再调用Derivrd2的构造函数。

```
construct Object
construct Base
construct Object
construct Derived2
```

最后，调用Final的构造函数。

```
construct Final
```

析构顺序：

```
destruct Final
destruct Derived2
destruct Object
destruct Base
destruct Object
destruct Derived1
destruct Object
destruct Mid
destruct Derived2
destruct Object
destruct Derived1
destruct Base
destruct Object
```

分析：

析构顺序与构造顺序完全相反

2、仿照课堂上的例子，使用通用指针实现归并排序算法，可以对double数组进行排序。

```
int double_compare(const void* p1, const void* p2)
{
    if (*(double*)p1 < *(double*)p2)
        return -1;
    else if (*(double*)p1 > * (double*)p2)
        return 1;
    else
        return 0;
}

/*
通用归并排序算法（从小到大）
参数：
    base：需要排序的数据内存首地址
    count：数据元素个数
    element_size：一个数据元素所占内存大小
    cmp：比较两个元素的函数
*/
void merge_sort(void* base, unsigned int count, unsigned int element_size, int
(*cmp)(const void*, const void*))
{
    if (count <= 1)
        return;
    merge_sort(base, count / 2, element_size, cmp);
    merge_sort((void*)((char*)base + count / 2 * element_size), count - count /
2, element_size, cmp);
    int i = 0, j = 0, k = 0;
    char* temp = new char[count * element_size];
```

```
    while (i < count / 2 && j < count - count / 2)
    {
        char* p1 = (char*)base + i * element_size;
        char* p2 = (char*)base + count / 2 * element_size + j * element_size;
        if (cmp(p1, p2) < 0)
        {
            for (int t = 0; t < element_size; t++)
                temp[k * element_size + t] = p1[t];
            i++;
        }
        else
        {
            for (int t = 0; t < element_size; t++)
                temp[k * element_size + t] = p2[t];
            j++;
        }
        k++;
    }
    while (i < count / 2)
    {
        char* p1 = (char*)base + i * element_size;
        for (int t = 0; t < element_size; t++)
            temp[k * element_size + t] = p1[t];
        i++;
        k++;
    }
    while (j < count - count / 2)
    {
        char* p2 = (char*)base + count / 2 * element_size + j * element_size;
        for (int t = 0; t < element_size; t++)
            temp[k * element_size + t] = p2[t];
        j++;
        k++;
    }
    for (int m = 0; m < count; m++)
    {
        char* p1 = (char*)base + m * element_size;
        char* p2 = (char*)temp + m * element_size;
        for (int n = 0; n < element_size; n++)
            p1[n] = p2[n];
    }
}
```