



C++期末不挂科

CH5 - 面向对象3 - 更多知识点

本章内容

- 深拷贝与浅拷贝
- 运算符重载
- 类的模块化编程
- 类的静态成员
- 构造函数参数列表
- 构造/析构顺序
- *多继承与菱形继承
- *友元函数
- *虚函数表

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

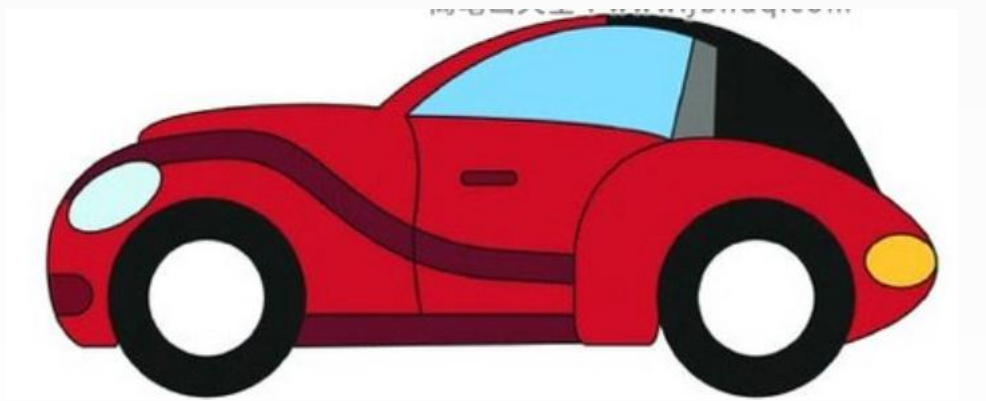
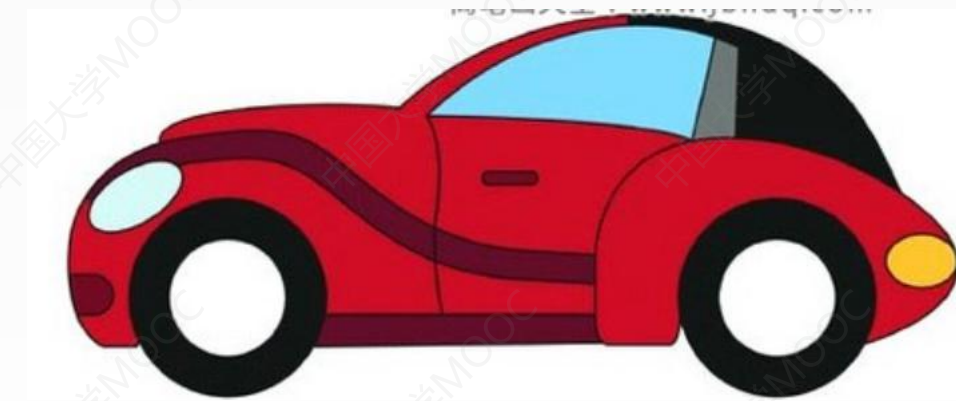
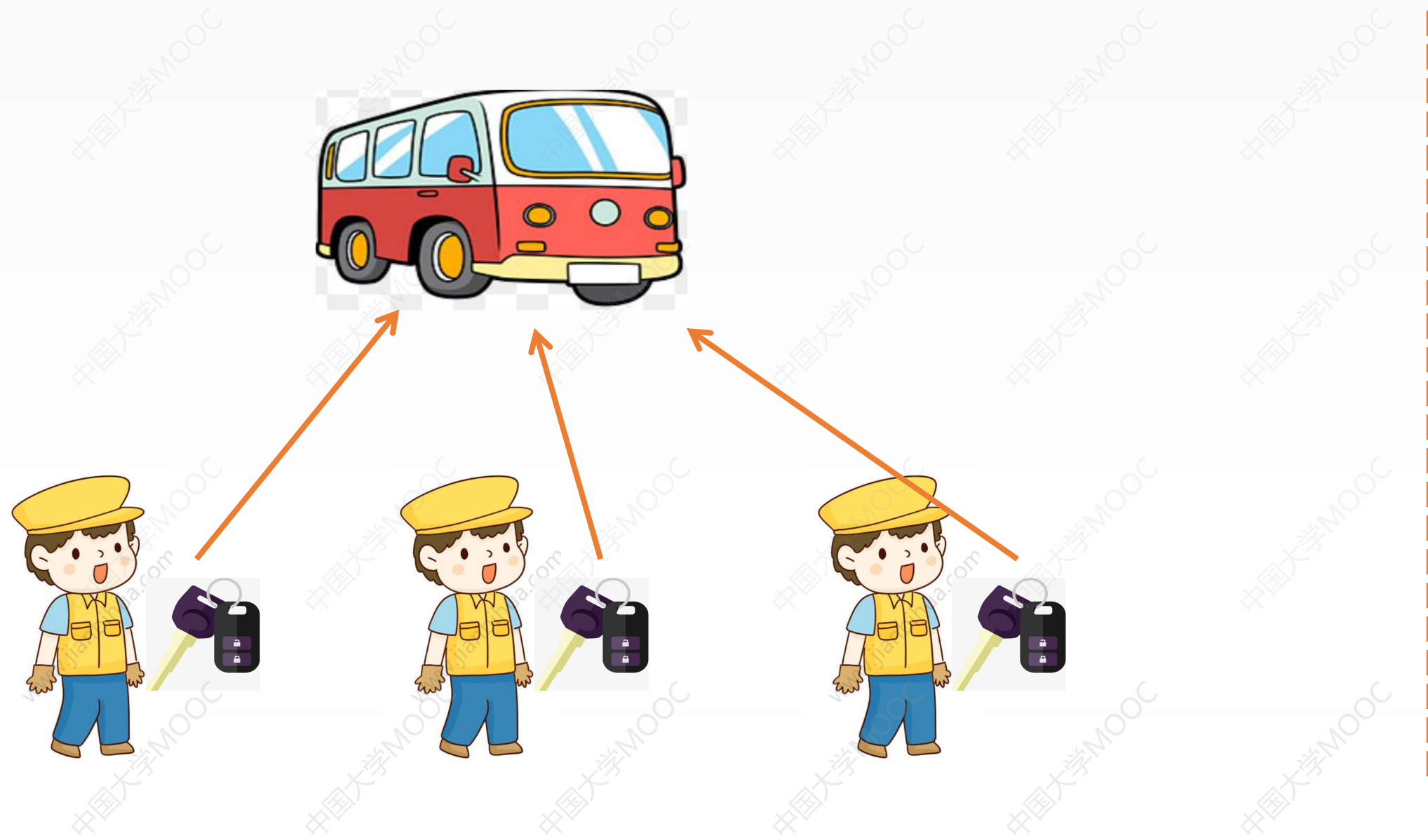
*多继承与菱形继承

*友元函数

*虚函数表

深拷贝与浅拷贝

- 如何拷贝一个对象？逐一拷贝成员变量？
- 如果对象持有**间接资源**呢(动态内存、文件句柄、网络连接等...)
- **浅拷贝**：只简单复制成员变量的值
- **深拷贝**：对于外部资源也复制一份



拷贝构造函数

- 拷贝构造函数：拷贝已存在同类对象来构造新对象，因此参数是该类的**引用**

```
class A {  
public:  
    int n;  
    char* ptr;  
    A(int n) {  
        this->n = n;  
        ptr = (char*)malloc(100);  
    }  
    A(A& other) {  
        this->n = other.n;  
        this->ptr = other.ptr;  
    }  
};
```

→ A的拷贝构造函数

→ 浅拷贝

```
A(A& other) {  
    this->n = other.n;  
    this->ptr = (char*)malloc(100);  
    memcpy(this->ptr, other.ptr, 100);  
}
```

若改成这样，就是**深拷贝**
将间接资源也复制了一遍

```
A a1(10);  
A a2(a1);  
A a3 = a1; // 等价于A a3(a1);
```

默认拷贝构造函数

- 如果没有显式定义拷贝构造函数，则类会默认拥有一个**浅拷贝**构造函数
- 因此，如果类内拥有间接资源，记得按需自定义一个深拷贝构造函数

```
class A {  
public:  
    int n;  
    char* ptr;  
    A(int n) {  
        this->n = n;  
        ptr = (char*)malloc(100);  
    }  
};
```

自动生成

```
A(A& other) {  
    this->n = other.n;  
    this->ptr = other.ptr;  
}
```

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表

运算符重载

- **运算符可理解为函数**：如 `cout << x` 等价于调用`cout`对象的成员函数`<<`，且将`x`作为`<<`函数的参数
- 那有了函数为什么还要有**运算符重载**机制呢？
- 就问你 `printf("%d" , n)` 好看还是 `cout << n` 好看
- 思考：有一个类`Vector`表示直角坐标系下的向量，如何按照数学定义实现两个向量对象的加法？

```
class Vector {  
public:  
    int a;  
    int b;  
    Vector(int a, int b) {  
        this->a = a;  
        this->b = b;  
    }  
};
```

定义函数

```
Vector add_two_vec(Vector& v1, Vector& v2) {  
    Vector result(v1.a + v2.a, v1.b + v2.b);  
    return result;  
}
```

然后调用函数即可

```
Vector v3 = add_two_vec(v1, v2);
```

这样有什么问题？

不直观！ 我就要 `v3 = v1 + v2`

运算符重载

- 为自定义的类重载一个运算符函数，其第一个操作数是对象本身，其他操作数是该函数参数
- 为Vector类重载 '+' 号，使 '+' 作为Vector类的一个成员函数，参数是另一个Vector对象的引用

```
class Vector {  
public:  
    int a;  
    int b;  
    Vector(int a, int b) {  
        this->a = a;  
        this->b = b;  
    }  
    Vector operator +(Vector& other) {  
        return Vector(this->a + other.a, this->b + other.b);  
    }  
};
```

这样就可以

本质在调用v1对象的operator+函数
将v2引用传参给other

v1.+(v2); *//不能这么写, 只是意会*

Vector v3 = v1 + v2;

重载赋值号(等号)

- 在定义对象时直接使用赋值号相当于调用**拷贝构造函数**，如：A a1 = a2; 相当于 A a1(a2);
- 对已定义对象赋值相当于调用重载的 operator = 函数，如无显式定义，则默认进行对象的**浅拷贝**
- 在类中有间接资源时，记得按需自定义重载 operator = 以实现深拷贝

```
class A {  
public:  
    int n;  
    char* ptr;  
    A(int n) { // 省略... }  
    A& operator = (A& other) {  
        this->n = other.n;  
        this->ptr = other.ptr;  
        return *this;  
    }  
};
```

```
A& operator = (A& other) {    深拷贝版本  
    this->n = other.n;  
    memcpy(this->ptr, other.ptr, 100);  
}
```

这样可以支持**连续赋值**:

```
A a1(1), a2(2), a3(3), a4(4);  
a1 = a2 = a3 = a4;
```


为什么有了指针还要引用

- 引用与指针的作用类似：使得函数可以就地修改参数，避免参数的拷贝
- 且引用的底层实现也利用了指针
- 因此你应该有此疑问非常久了：**为什么有了指针还要有引用**
- 现在学习了运算符重载，你应该可以回答这个问题了：

引用使得使用**运算符重载**时可以不必要带着 * 符号
既清晰美观，又避免了歧义

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表

类的模块化编程

- 在C语言中，为了代码的封装，我们会将函数的声明与定义分开
- 函数声明在 .h 头文件中，定义在 .c 源代码文件中，将 .c 编译为二进制文件后同 .h 一起交付给使用方
- 这样使用方编写代码时只看见声明，不会知道具体实现方式
- C++中编写类也可以使用相同的模块化方式，将声明放在 .h 文件中，实现放在 .cpp 文件中

MyClass.h文件:

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
private:
    int n;
public:
    MyClass(int n);
    int get_n();
    void set_n(int n);
};

#endif
```

防止重复
包含同一
个头文件

MyClass.cpp文件:

```
#include "MyClass.h"

MyClass::MyClass(int n) {
    this->n = n;
}

int MyClass::get_n() {
    return n;
}

void MyClass::set_n(int n) {
    this->n = n;
}
```

main.cpp文件:

```
#include <iostream>
#include "MyClass.h"

int main() {
    MyClass a(5);
    std::cout << a.get_n();
}
```

用名字空间来表明这是
类MyClass的成员函数定义

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表

静态 static 的概念

- 回顾：函数中的static变量存在于全局/静态区，生命周期伴随整个程序，不随着函数结束而死亡

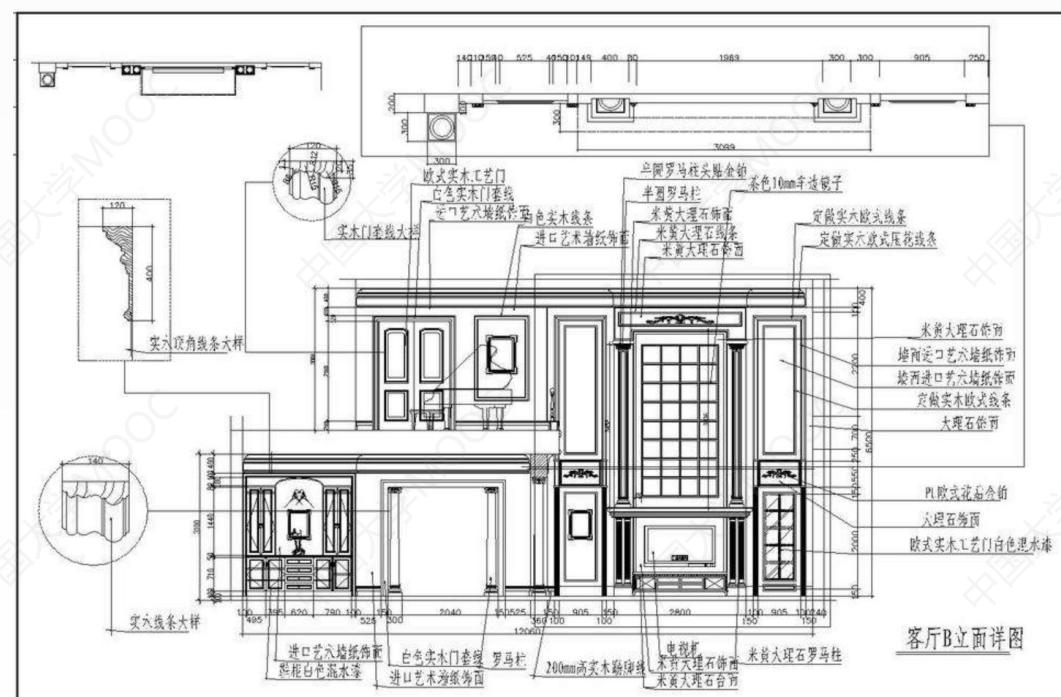
```
void func() {  
    static int n = 1;  
    cout << n++ << "\n";  
}  
// 省略...  
func();  
func();  
func();
```

A terminal window with a black background and yellow text. It displays the output of the program: 1, 2, 3, each on a new line.

```
1  
2  
3
```

类的静态成员

- 隶属于类，不属于任何一个对象，生命周期贯穿整个程序
- 类比：静态成员变量是写在设计图纸上的数据，与楼无关
- 类内可直接访问，外部则需要通过 **类名::变量名** 访问
- 静态成员变量必须在类声明**外部单独初始化**！
- 格式：**数据类型 类名::变量名 = 初始值;**



设计图纸

起楼



真实存在的楼

一个例子搞懂类的static成员：对象计数

- 类利用其static成员变量来记录它的实例化对象的**实时数目**

```
class A {  
public:  
    static int count;  
    A() {  
        count++;  
    }  
    ~A() {  
        count--;  
    }  
};  
  
int A::count = 0;
```

```
A a1;  
cout << A::count << "\n";  
A a2;  
cout << A::count << "\n";  
{  
    A a3;  
    cout << A::count << "\n";  
}  
cout << A::count << "\n";
```



1
2
3
2

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表

构造函数的调用顺序：父 -> 子

- 假设 $A \leftarrow B \leftarrow C$
- 则实例化一个C类的对象需要调用几个构造函数？仅调用C类的构造函数吗？
- 显然不够。继承的含义是获取，需要先存在，才能获取
- 实例化一个C类对象需要**依次调用**A、B、C的构造函数

```
class A {  
public:  
    A() {  
        cout << "A构造\n";  
    }  
};
```

```
class B : public A {  
public:  
    B() {  
        cout << "B构造\n";  
    }  
};
```

```
class C : public B {  
public:  
    C() {  
        cout << "C构造\n";  
    }  
};
```

C c;



A构造
B构造
C构造

析构函数的调用顺序：子 -> 父


- 析构函数与构造函数的调用次序刚好相反，先调用子类的析构函数

```
class A {  
public:  
    A() {  
        cout << "A构造\n";  
    }  
    ~A() {  
        cout << "A析构\n";  
    }  
};
```

```
class B : public A {  
public:  
    B() {  
        cout << "B构造\n";  
    }  
    ~B() {  
        cout << "B析构\n";  
    }  
};
```

```
class C : public B {  
public:  
    C() {  
        cout << "C构造\n";  
    }  
    ~C() {  
        cout << "C析构\n";  
    }  
};
```

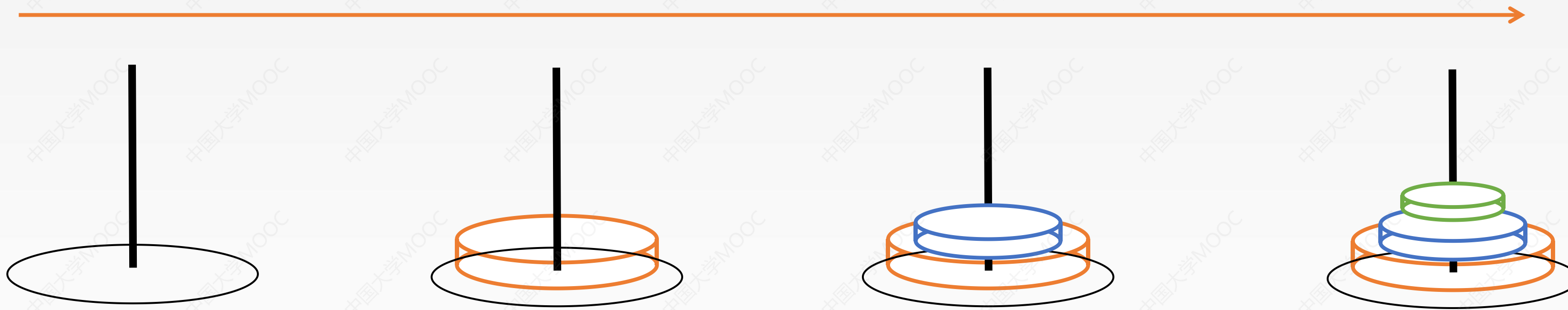
```
int main() {  
    C c;  
}
```



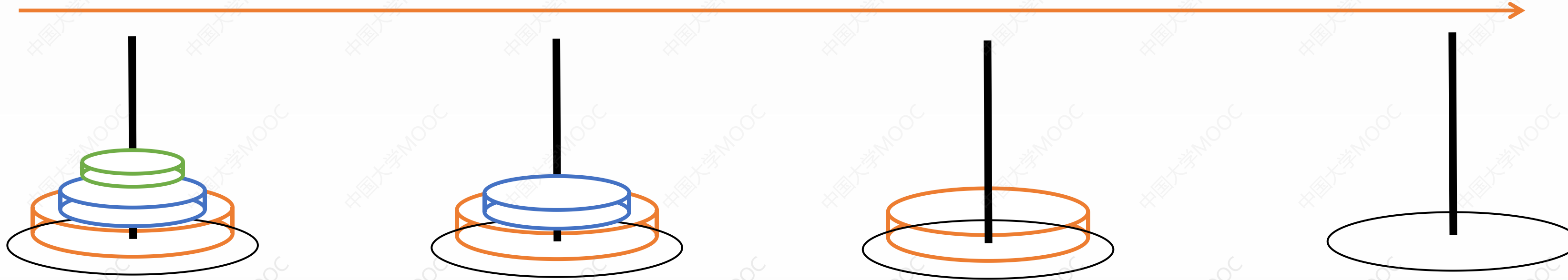
```
A构造  
B构造  
C构造  
C析构  
B析构  
A析构
```

直观理解

- 构造过程



- 析构过程



深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表

构造函数参数列表

- 思考：构造函数一般干什么？
- 初始化变量、分配资源...
- 试想这种情况：**某些成员变量没有无参构造函数**
- 此时无法在构造函数中初始化，因为这样实际上是先默认构造，再赋值

```
class A {  
public:  
    int n;  
    A(int n) {  
        this->n = n;  
    }  
};
```

```
class B {  
public:  
    A a;  
    B(int n) {  
        a = A(n);  
    }  
};
```

```
class B {  
public:  
    A a;  
    B(int n) : a(n) { }  
};
```




构造函数参数列表

- 构造函数参数列表其实就是一系列**构造函数的调用**，使它们在本对象构造前构造好
- **没有默认构造函数**的类成员数据或父类必须放在参数列表里构造

```
class A {  
public:  
    float fA;  
    A(float fA) {  
        this->fA = fA;  
    }  
};
```

```
class B {  
public:  
    int n1;  
    int n2;  
    A a;  
    B(int n1, int n2, float fA) : n1(n1), n2(n2), a(fA) {}  
};
```



这样也可，n1和n2可以先定义再赋值

```
B(int n1, int n2, float fA) : a(fA) {  
    this->n1 = n1;  
    this->n2 = n2;  
}
```

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

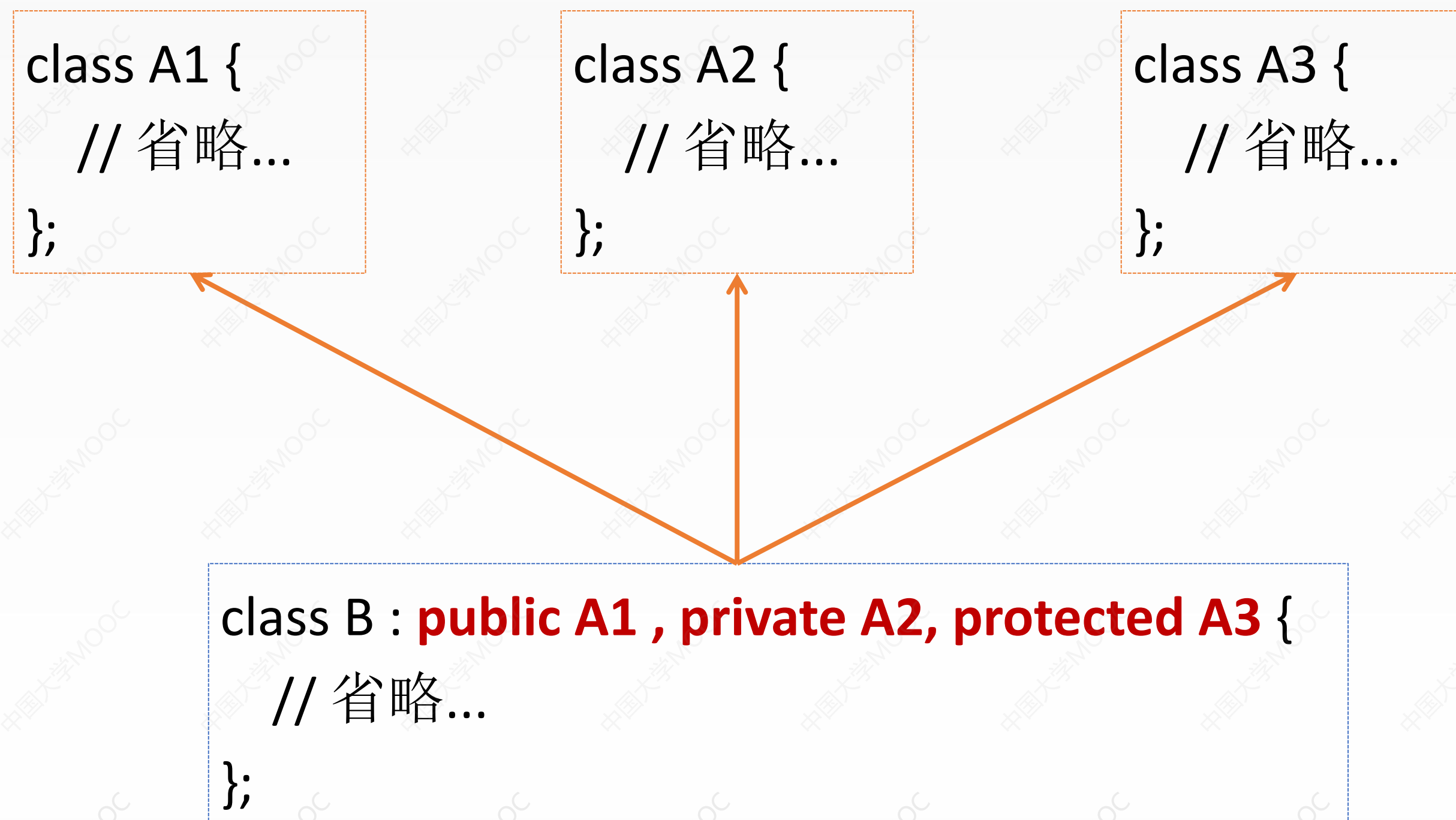
***多继承与菱形继承**

***友元函数**

***虚函数表**

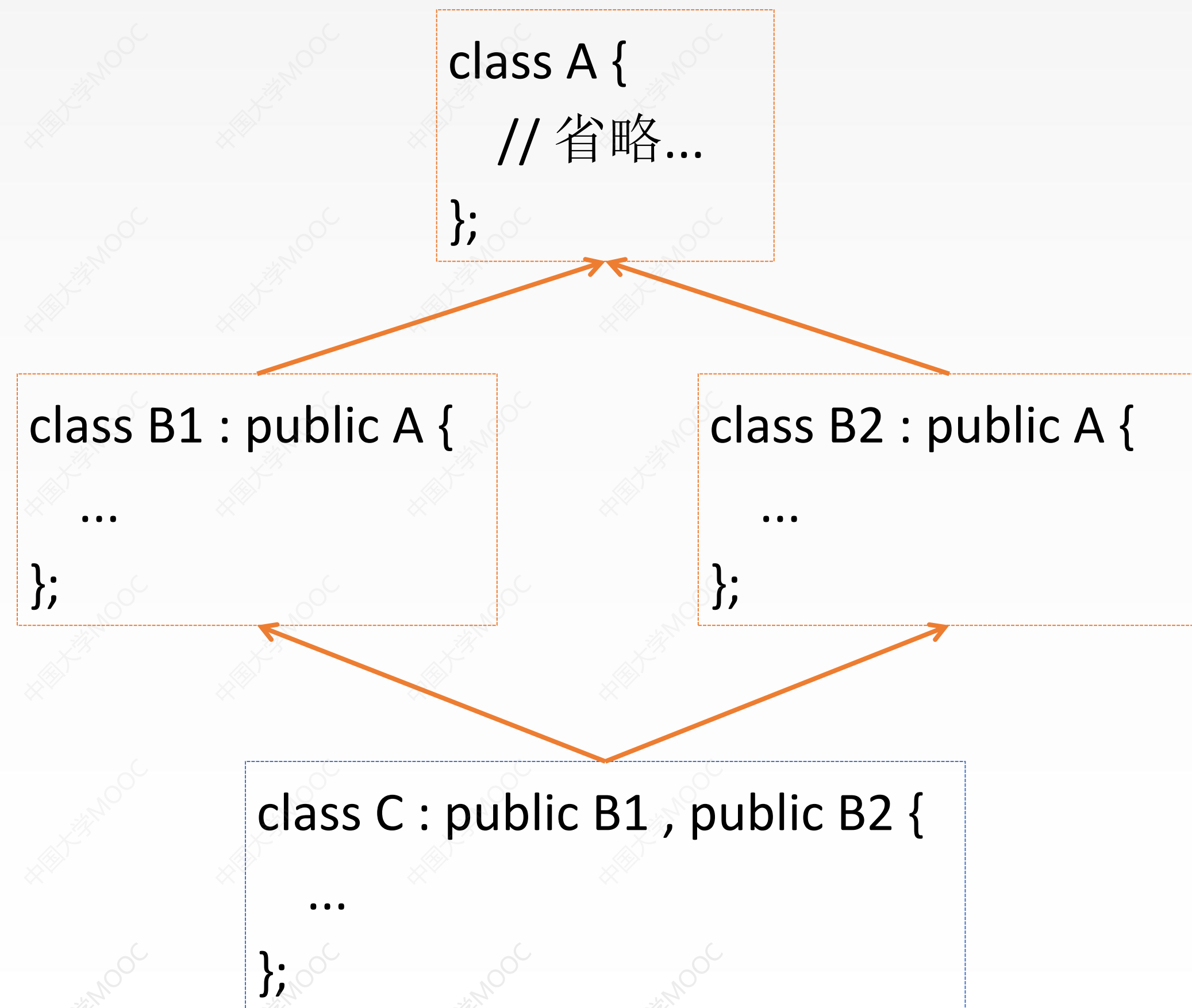
多继承

- C++ **允许多继承**：一个子类可以继承自多个父类
- 相当于分别按照指定的继承方式获得这些父类的成员



菱形继承

- 在多继承时有可能出现此情况：多个父类还有它们的共同父类
- 会出现**父类成员多个副本**问题
- 解决方案：**虚继承**



深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表

友元函数

- 好东西当然要和朋友(friend)分享，不要私藏(private)起来或者保护(protected)起来
- 友元函数虽然需要通过**friend**关键字在类内进行**声明**，但并**不是类的成员函数**！
- 类的友元函数**可以直接访问**该类的private和protected成员

```
class A {  
private:  
    int n;  
public:  
    friend void print_n(A &a);  
    A(int n) :n(n) {}  
};
```

```
void print_n(A& a) {  
    cout << a.n << "\n";  
}
```

print_n 并不是类A的成员函数！

深拷贝与浅拷贝

运算符重载

类的模块化编程

类的静态成员

构造/析构顺序

构造函数参数列表

*多继承与菱形继承

*友元函数

*虚函数表



中国大学MOOC
搜索：C++不挂科

虚函数表

指C++底层是怎么支持的

- 运行时多态好神奇，怎么实现的？
- 每个对象存有一张**虚函数表**，记录了其所有虚函数指针
- 当子类重写父类的虚函数时，用子类重写的版本**覆盖**父类原有的虚函数指针
- 因此通过指针调用时会使用最新重写的那个版本

Father虚函数表

函数名	函数地址
func1	123
func2	456

override

Son虚函数表

函数名	函数地址
func1	123
func2	999
func3	789

ptr访问的虚函数表

```
Son son;  
Father *ptr = (Father*) &son;  
ptr -> func2();
```