

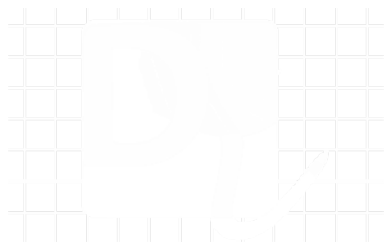
高级语言C++程序设计

Lecture 10 类的继承与多态

南开大学 计算机学院

2022

类的继承与派生



Recap:类的访问权限

```
class A {  
    private:  
        int x;  
    public:  
        int z;  
    public:  
        void test() {  
            x = 1; // ok  
            z = 1; // ok  
        }  
};
```

test是类的成员函数
，可以访问类的私有成员x和z

Recap:类的访问权限

```
class A {  
    private:  
        int x;  
    public:  
        int z;  
    public:  
        void test() {  
            x = 1; // ok  
            z = 1; // ok  
            A a;  
            a.x = 1; // ok?  
            a.z = 1; // ok?  
        }  
};
```

a.x和test不属于一个类对象,但两个类对象都是A类型, 类A是自己的友元

Recap:类的访问权限

```
class A {  
    private:  
        int x;  
    public:  
        int z;  
    public:  
        void test() {  
            x = 1; //ok  
            z = 1; //ok  
            A a;  
            a.x = 1; //ok  
            a.z = 1; //ok  
        }  
};
```

```
void f() {  
    A a;  
    a.x = 1; //error  
    a.z = 1;  
}
```

f是类外部函数，不能访问类的私有成员

Recap:类的访问权限

```
class A {  
    private:  
        int x;  
    public:  
        int z;  
    public:  
        void test() {  
            x = 1; //ok  
            z = 1; //ok  
            A a;  
            a.x = 1; //ok  
            a.z = 1; //ok  
        }  
};
```

```
void f() {  
    A a;  
    a.x = 1; //error  
    a.z = 1; //ok  
}
```

f是类外部函数，可以访问类的公有成员

Recap:类的访问权限

```
class A {  
    private:  
        int x;  
        protected:  
            int y;  
    public:  
        int z;  
    public:  
        void test() {  
            x = 1; //ok  
            z = 1; //ok  
            y = 1; //ok  
        }  
};
```

protected是保护类型
，在没有类继承时与
private权限相同

```
void f() {  
    A a;  
    a.x = 1; //error  
    a.y = 1; //error  
    a.z = 1; //ok  
}
```

类的继承与派生

事物之间的Is-A关系

- ❑ employee（雇员）
 - 姓名、年龄、工资
- ❑ engineer（工程师）
 - 姓名、年龄、工资、专业、学位
- ❑ manager（经理）
 - 姓名、年龄、工资、行政级别

如果为每个事物分别定义一个类会导致冗余

类的继承与派生

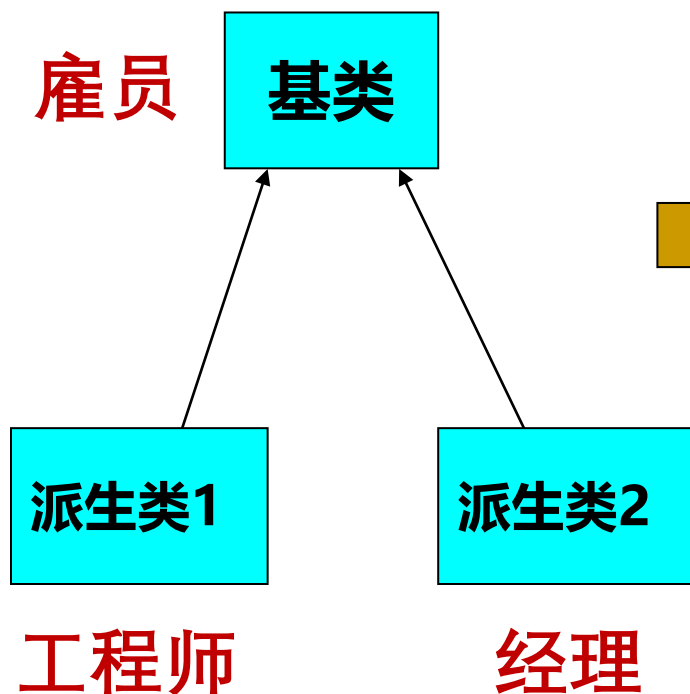
继承和派生：为解决Is-A关系设计

- ❑ 面向对象程序设计代码复用、消除冗余最重要的手段
 - ❑ 允许在保持原有类特性的基础上进行扩展(继承)，创建新的类
 - ❑ 继承产生新的类，称为派生类或子类，被继承的类称为基类或父类
-

类的继承与派生

单继承

- 派生类只有一个直接基类

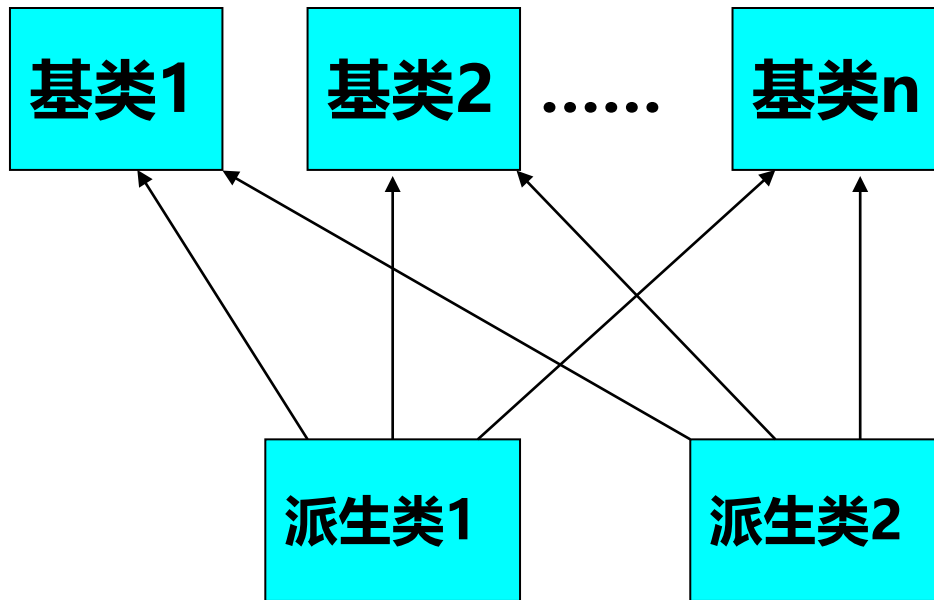


一个基类可以直接派生出多个派生类

类的继承与派生

多重继承

- 一个派生类同时有多个基类，派生类同时得到多个已有类的特征



类的继承与派生

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

类B继承了类A，类B有6个成员变量，包括继承A的3个成员x,y,z和自己新加的成员l,m,n

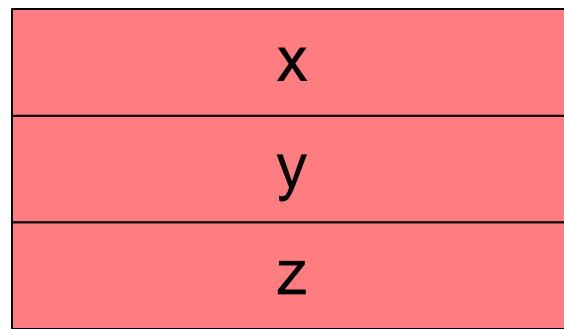
类的继承与派生

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



类对象的内存布局



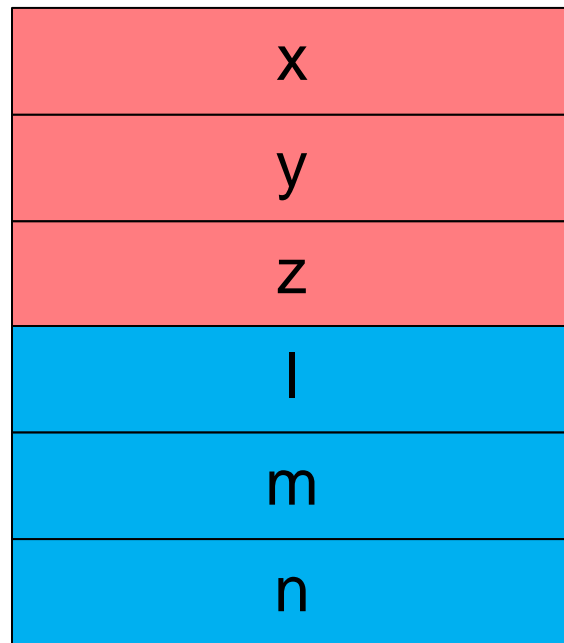
类A的每个类对象包含3个成员，x, y, z

类的继承与派生

派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

类对象的内存布局

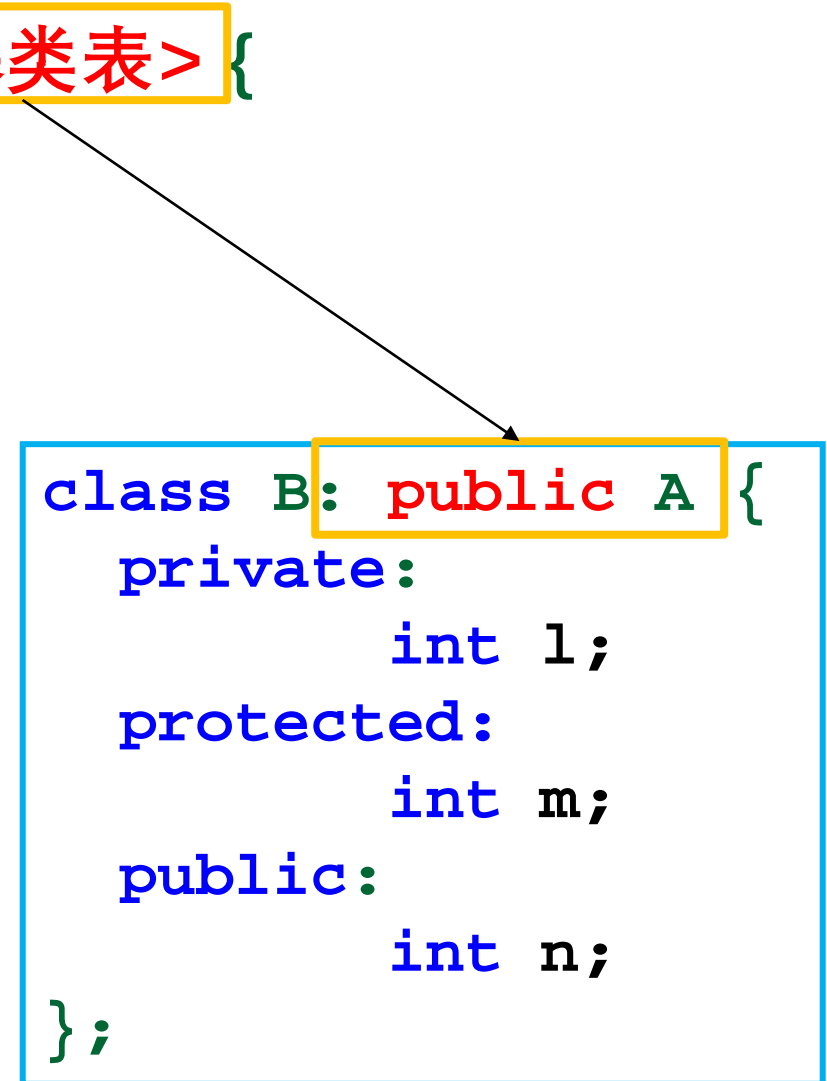


类B的每个类对象包含6个成员，x, y, z, l, m, n

派生类的定义

```
class <派生类类型名> : <基类表> {  
    private:  
        <各私有成员说明>;  
    public:  
        <各公有成员说明>;  
    protected:  
        <各保护成员说明>;  
};
```

相比类定义只是
多了一个基类表

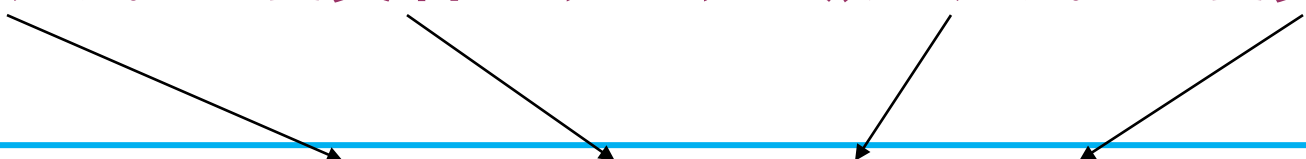


```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

派生类的定义

<基类表>的一般格式为：

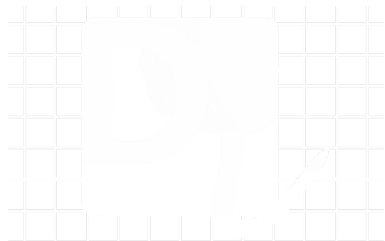
<派生方式> <基类名1>, ... , <派生方式> <基类名n>



```
class B: public A, private C {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

派生方式可以是 private
、 public 或 protected

衍生类的访问权限



派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
    void test() {  
        x = 1; // ok?  
        y = 1; // ok?  
        z = 1; // ok?  
    }  
};
```

问题1: 类B是否可以访问从类A继承过来的成员?

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
    void test() {  
        x = 1; // error  
        y = 1; // ok?  
        z = 1; // ok?  
    }  
};
```

派生类不能访问基类的private成员

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
    void test() {  
        x = 1; // error  
        y = 1; // ok  
        z = 1; // ok?  
    }  
};
```

派生类可以访问基类的protected成员

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
    void test() {  
        x = 1; // error  
        y = 1; // ok  
        z = 1; // ok  
    }  
};
```

派生类可以访问基类的public成员

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
    void test() {  
        y = 1; // ok  
        A a;  
        a.y = 1; // error  
    }  
};
```

派生类的成员函数只能访问自己继承过来的基类保护成员，不能访问其他基类对象的保护成员

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

问题2： 在外界看来，类B的6个成员变量分别是什么属性？

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

派生类新添加的成员属性取决于他们定义时的
权限操作符

```
void test() {  
    B b;  
    b.l = 1; // error  
}
```

l是private类型

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

派生类新添加的成员属性取决于他们定义时的
权限操作符

```
void test() {  
    B b;  
    b.m = 1; // error  
}
```

m是protected类型

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

派生类新添加的成员属性取决于他们定义时的
权限操作符

```
void test() {  
    B b;  
    b.n = 1; // ok  
}
```

n是public类型

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



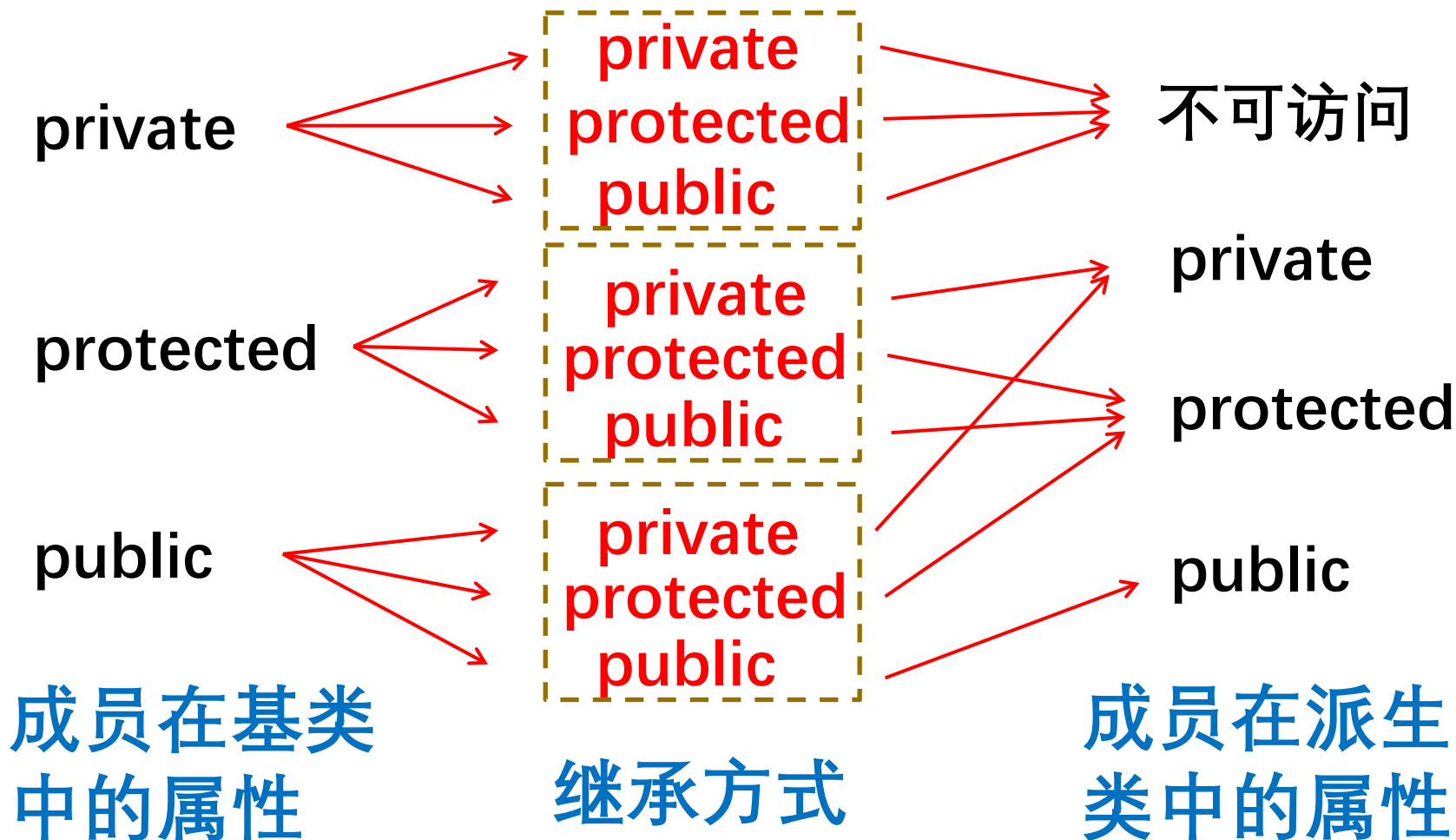
派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

派生类继承来的成员的属性取决于**基类中的权限操作符和派生方式**

派生类的访问权限

派生类继承来的成员的属性



派生类的访问权限

■ public派生方式

- 基类的私有成员不可在派生类中被存取，公有成员和保护成员在派生类中仍然是公有成员和保护成员

■ protected派生方式

- 基类的私有成员不可在派生类中被存取，公有成员和保护成员在派生类中都变为保护成员

■ private派生方式

- 基类的私有成员不可在派生类中被存取，公有成员和保护成员在派生类中都变为私有成员
-

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

private + public继承方式
结果为不可访问

```
void test() {  
    B b;  
    b.x = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

protected + public继承
方式结果为protected

```
void test() {  
    B b;  
    b.y = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: public A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

public + public继承方式
结果为public

```
void test() {  
    B b;  
    b.z = 1; // ok  
}
```


派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B:protected A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

private + protected继承
方式结果为不可访问

```
void test() {  
    B b;  
    b.x = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B:protected A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

protected + protected继承方式结果为protected

```
void test() {  
    B b;  
    b.y = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B:protected A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

public + protected继承
方式结果为protected

```
void test() {  
    B b;  
    b.z = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: private A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

private + private继承方式结果为不可访问

```
void test() {  
    B b;  
    b.x = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: private A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

protected + private继承
方式结果为private

```
void test() {  
    B b;  
    b.y = 1; // error  
}
```

派生类的访问权限

基类

```
class A {  
    private:  
        int x;  
    protected:  
        int y;  
    public:  
        int z;  
};
```



派生类

```
class B: private A {  
    private:  
        int l;  
    protected:  
        int m;  
    public:  
        int n;  
};
```

public + private继承方式
结果为private

```
void test() {  
    B b;  
    b.z = 1; // error  
}
```

例子-1

```
class employee{//employee类将作为基类
protected:
    char * name;
    short age;
    float salary;
public:
    void print (){
        cout<<"employee's print ";
    }
};
```

例子-1

```
class manager: public employee { // 派生类
    int level;
public:
    void print() {
        employee::print(); // 调用基类print
        cout << "manager's print " << endl;
    }
};
```

例子-1

```
void main() { // 主函数
    employee emp1;
    manager man1, man2;

    emp1.print();
    man1.print();
    man2.employee::print(); // 调用基类的print
}
```

允许派生类中的print与基类的print重名：对子类而言，不加类名限时默认为是处理子类成员，而要访问父类重名成员时，则要通过类名限定

例子-2

```
class baseCla { // 定义基类
    int privData;
protected:
    int protData;
public:
    int publData;
};
```

例子-2

```
class publDrvCla : public baseCla {
public:
    void usebaseClaData() {
        publData=11;    //OK!
        protData=12;    //OK!
        privData=13;    //ERROR!
    }
};

class claD21 : public publDrvCla {
public:
    void usebaseClaData() {
        publData=111;    //OK!
        protData=121;    //OK!
        privData=131;    //ERROR!
    }
};
```

例子-2

```
class protDrvCla : protected baseCla {
public:
    void usebaseClaData() {
        publData=21;      //OK!
        protData=22;      //OK!
        privData=23;      //ERROR!
    }
};

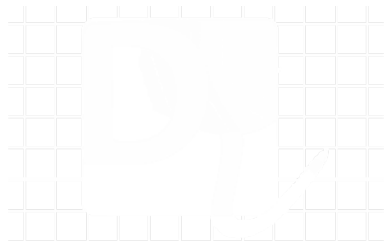
class claD22 : public protDrvCla {
public:
    void usebaseClaData() {
        publData=211;      //OK!
        protData=221;      //OK!
        privData=231;      //ERROR!
    }
};
```

例子-2

```
class privDrvCla : private baseCla {
public:
    void usebaseClaData() {
        publData=31;           //OK!
        protData=32;           //OK!
        privData=33;           //ERROR!
    }
};

class claD23 : public privDrvCla {
public:
    void usebaseClaData() {
        publData=311;          //ERROR!
        protData=321;          //ERROR!
        privData=331;          //ERROR!
    }
};
```

派生类的构造函数



派生类的构造函数

派生类的构造函数的一般格式如下：

```
<派生类名>(<参数总表>):<初始化符表>
{
    <构造函数体>
}
```

初始化符表完成对基类和对象成员的初始化

```
<基类名>(<基类参数表>), ... , <对象成员名>(<
    对象成员参数表>), ...
```

基类名与对象成员名的次序无关紧要

派生类的构造函数

```
class Box
```

```
{
```

```
protected:
```

```
    double length {1.0};
```

```
    double width {1.0};
```

```
    double height {1.0};
```

```
public:
```

```
    Box(double lv, double wv, double hv)
```

```
        : length(lv), width(wv), height(hv) {
```

```
        cout<<"Box(double,double,double) called";
```

```
    } //3个参数的构造函数
```


派生类的构造函数

```
explicit Box(double side):Box(side, side,
side) {
    cout<<"Box(double) called.";
} //1个参数的构造函数
```

```
Box() {cout<<"Box() called.";} //无参数构造函数
```

```
double volume() const
{
    return length*width*height;
} //计算box的体积
};
```

派生类的构造函数

```
class Carton : public Box
{
private:
```

```
    string material {"Cardboard"}; //派生类新对象成员
```

public: 初始化表，包括基类初始化，对象成员初始化

```
    Carton(double lv, double wv, double hv, string
mat) : Box(lv, wv, hv), material(mat) {
        cout << "Carton(double,double,double,string)
called.";
    } //派生类构造函数，包含4个参数，3个用来初始化基类成员，
        1个初始化对象成员
```

派生类的构造函数

```
explicit Carton(string mat) : material(mat) {  
    cout << "Carton(string) called.";  
}
```

//派生类构造函数，只包含1个参数，用来初始化对象成员；
同时会调用基类的无参构造函数！

```
Carton(double side, string mat):Box(side),  
material(mat) {  
    cout << "Carton(double,string) called.";  
}
```

//派生类构造函数，包含2个参数，一个用来初始化基类成员，
一个用来初始化对象成员

```
Carton() { cout << "Carton() called."; }
```

//派生类无参构造函数，会调用基类的无参构造函数！

```
};
```

派生类的构造函数

```
int main() {  
    Carton carton1; //调用无参构造函数  
    cout << endl;  
    //调用1个参数的构造函数  
    Carton carton2 ("Thin cardboard");  
    cout << endl;  
    //调用4个参数的构造函数  
    Carton carton3 (4.0, 5.0, 6.0, "Plastic");  
    cout << endl;  
    //调用2个参数的构造函数  
    Carton carton4 (2.0, "paper");  
    cout << endl;  
}
```

派生类的构造函数

Box() called.

Carton() called.

程序的运行结果

Box() called.

Carton(string) called.

Box(double, double, double) called.

Carton(double,double,double,string) called.

Box(double, double, double) called.

Box(double) called.

Carton(double,string) called.

派生类的构造函数

派生类构造函数与基类构造函数的联系

- 在派生类构造函数中，只要基类不是使用无参的默认构造函数都要显式给出基类名和参数表
 - 如果基类没有定义构造函数，则派生类也可以不定义，全部采用系统给定的默认构造函数
 - 如果基类定义了带有形参表的构造函数时，派生类就应当定义构造函数
-

派生类的构造函数

派生类构造函数执行的一般次序

- 调用各基类的构造函数，调用顺序为派生继承时的**基类声明顺序**
 - 若派生类含有对象成员的话，调用各对象成员的构造函数，调用顺序按照派生类中**对象成员的声明顺序**
 - 执行派生类构造函数的函数体
-

派生类的构造函数调用顺序

```
class CB{
    int b;
public:
    CB(int n){    b=n;
                cout<<"CB::b="<<b<<endl; };
};
class CC{
    int c;
public:
    CC(int n1,int n2){ c=n1;
                       cout<<"CC::c="<<c<<endl; };
};
```


派生类的构造函数调用顺序

```
class CD:public CB,public CC {
    int d;
    CC obcc;
    CB obcb;
public:
    CD(int n1,int n2,int n3,int n4):CC(n3,n4),
    CB(n2), obcb(100+n2),obcc(100+n3,100+n4){
        d=n1;        cout<<"CD::d="<<d<<endl; };
};

void main(void){
    CD CObj(2,4,6,8);
}
```

输出结果:

CB::b=4

CC::c=6

CC::c=106

CB::b=104

CD::d=2

派生类构造函数的进一步讨论

如果把中Carton类的构造函数

```
Carton(double lv, double wv, double hv, string  
mat)
```

改为:

```
Carton(double lv, double wv, double hv, string  
mat):length{lv}, width{wv}, height{hv},  
material{mat} {  
    cout<<"Carton(double,double,double,string)  
called.";  
} Error: "length"不是类"Carton"的非静态成员或基类
```

派生类初始化表中不能对继承过来的成员直接初始化

派生类构造函数的进一步讨论

如果把中Carton类的构造函数

```
Carton(double lv, double wv, double hv, string  
mat)
```

改为:

```
Carton(double lv, double wv, double hv, string  
mat):material{mat} {  
    length = lv; width = wv; height = hv;  
    cout<<"Carton(double,double,double,string)  
called."  
}  
OK, 但其实调用了基类无参构造函数
```

派生类初始化表如果不显示调用基类构造函数, 就会调用基类的默认无参构造函数

派生类的拷贝构造函数

- 格式和适用场景与普通类的拷贝构造函数相同
 - 可以在成员初始化符表位置调用基类的拷贝构造函数，“拷贝”派生类中的基类部分
 - 如果不显式地调用基类的拷贝构造函数，将自动调用基类的**无参构造函数**为派生类创建基类部分
-

派生类的拷贝构造函数

为基类Box和派生类Carton分别添加拷贝构造函数

```
Box(const Box& box) :length{box.length},  
width{box.width}, height{box.height} {  
    cout<<"Box copy constructor";  
} //基类Box的拷贝构造函数
```

```
Carton(const Carton& carton)  
: material {carton.material} {  
    cout<<"Carton copy constructor";  
} //派生类的拷贝构造函数，没有显示地调用基类拷贝构造函数，  
    将会调用基类无参构造函数！
```

派生类的拷贝构造函数

```
int main() {  
    //调用4个参数的构造函数  
    Carton carton (4.0, 5.0, 6.0, "Plastic");  
  
    //调用拷贝构造函数  
    Carton cartonCopy = carton;  
  
    //输出体积: length*width*height  
    cout << carton.volum();  
  
    //输出体积: length*width*height  
    cout << cartonCopy.volum();  
}
```

派生类的拷贝构造函数

程序运行结果

`Box(double, double, double) called.`

`Carton(double,double,double,string) called.`

`Box() called.`

`Carton copy constructor`

`Volume of carton is 120`

`Volume of cartonCopy is 1`

在调用派生类Carton的时候，先调用了基类Box的无参构造函数

基类构造函数调用时，使用成员变量的默认值(length,width,height默认都是1)，因此cartonCopy的volume()值为1

派生类对象的“深”拷贝

定义派生类拷贝构造函数时，显式地调用基类的拷贝构造函数，将派生类的基类部分“深拷贝”给相应的派生类对象

```
Box(const Box& box) :length{box.length},  
width{box.width}, height{box.height} {  
    cout<<"Box copy constructor";  
} //基类Box的拷贝构造函数
```

```
Carton(const Carton& carton)  
: Box(carton), material {carton.material}  
{  
    cout<<"Carton copy constructor";  
} //派生类的拷贝构造函数显示地调用基类拷贝构造函数
```


派生类对象的“深”拷贝

```
int main() {  
    //调用4个参数的构造函数  
    Carton carton (4.0, 5.0, 6.0, "Plastic");  
  
    //调用拷贝构造函数  
    Carton cartonCopy = carton;  
  
    //输出体积: length*width*height  
    cout << carton.volum();  
  
    //输出体积: length*width*height  
    cout << cartonCopy.volum();  
}
```

派生类对象的“深”拷贝

程序运行结果

```
Box(double, double, double) called.  
Carton(double,double,double,string) called.
```

```
Box copy constructor  
Carton copy constructor
```

```
Volume of carton is 120
```

```
Volume of cartonCopy is 120
```

在调用派生类Carton的时候，调用了基类Box的拷贝构造函数

基类拷贝构造函数对成员变量进行了深拷贝，因此cartonCopy的volume()值为120

派生类构造函数的“继承”

- ❑ 派生类可以使用`using`关键字，显式地“继承”基类的构造函数（无参构造函数除外）
- ❑ 继承来的基类构造函数可以当作派生类的构造函数使用，初始化派生类对象的基类部分

```
class Carton : public Box
{
    using Box::Box; //继承了Box所有的构造函数（无参构造函数除外）

    private:
        std::string material {"Cardboard"};
    public:
        Carton(double lv, double wv, double hv, std::string
mat) : Box {lv, wv, hv}, material {mat}
        {cout<<"Carton(double,double,double,string) called.";}
};
```

派生类构造函数的“继承”

```
int main()
```

```
{
```

```
    Carton cart;
```

//**error!** Carton本身没有无参构造函数，Box的无参构造函数也没有继承过来

```
    Carton cube{4.0};
```

//**OK!** Carton本身没有单参数构造函数，但是继承了Box的单参构造函数

```
    cout<<cube.volume()<<endl;
```

//通过Box的单个参数的构造函数，length, width, height都被初始化为4

派生类构造函数的“继承”

```
Carton carton {1.0, 2.0, 3.0};
```

//OK! Carton本身没有3个参数的构造函数，但是继承了Box的3个参数的构造函数

```
cout<< carton.volume()<<endl<<endl;
```

//通过Box的3个参数的构造函数，length, width, height被初始化为1, 2, 3

```
Carton candyCarton (50.0, 30.0, 20.0, "Thin cardboard");
```

//OK! 调用Carton自己的4个参数的构造函数

```
cout<<candyCarton.volume()<<endl;
```

// length, width, height被初始化为50, 30, 20

```
}
```

派生类构造函数的“继承”

程序运行结果

```
Box(double, double, double) called.
```

```
Box(double) called.
```

```
64
```

```
Box(double, double, double) called.
```

```
6
```

```
Box(double, double, double) called.
```

```
Carton(double,double,double,string)  
called.
```

```
30000
```

派生类的析构函数

- 析构函数的功能是做善后工作
 - 只要在函数体内把派生类新增的一般成员处理好就可以了，而对新增的成员对象和基类的善后工作，系统会自己调用对象成员和基类的析构函数来完成
 - 析构函数各部分执行次序与构造函数相反
 - 首先对派生类新增一般成员析构，然后对新增对象成员析构，最后对基类成员析构
-

派生类的析构函数

```
class CB{
    int b;
public:
    CB(int n){    b=n;
                cout<<"CB::b="<<b<<endl; };
    ~CB(){cout<<"CB destructing"<<endl;};
};

class CC{
    int c;
public:
    CC(int n1,int n2){ c=n1;
                    cout<<"CC::c="<<c<<endl; };
    ~CC(){cout<<"CC destructing"<<endl;};
};
```

派生类的析构函数

```
class CD:public CB,public CC{
    int d;
public:
    CD(int n1,int n2,int n3,int n4)
        :CC(n3,n4),CB(n2){ // 先CB,后CC,按照继承顺序
        d=n1;          cout<<"CD::d="<<d<<endl;
    }
    ~CD(){cout<<"CD destructing"<<endl;}
};

void main(void){
    CD CObj(2,4,6,8);
}
```

派生类的析构函数

运行结果为：

CB::b=4

CC::c=6

CD::d=2

CD destructing

CC destructing

CB destructing

派生类的析构函数

【思考】 将派生类CD改写为如下形式后，请给出输出结果

```
class CD:public CB,public CC {
    int d;
    CC obcc; //CC类型的对象成员
    CB obcb; //CB类型的对象成员
public:
    CD(int n1,int n2,int n3,int n4)
    :CC(n3,n4), CB(n2), obcb(100+n2),
    obcc(100+n3,100+n4) {
        d=n1;          cout<<"CD::d="<<d<<endl; };
    ~CD(){cout<<"CDobj is destructing"<<endl;};
}; //先基类CB、CC, 再对象成员CC、CB, 最后派生类CD
```

派生类的析构函数

输出结果:

CB::b=4

CC::c=6

CC::c=106

CB::b=104

CD::d=2

CDobj is destructing

CBobj is destructing

CCobj is destructing

CCobj is destructing

CBobj is destructing

友元的继承

- 如果基类有友元类或友元函数，则其派生类不因继承关系也有此友元类或友元函数
- 如果基类是某类的友元类，则这种友元关系将被继承
 - 被派生类继承过来的成员，如果原来是某类的友元，那么它作为派生类的成员仍然是某类的友元

静态成员的继承

- 如果基类的**静态成员**是公有的或是保护的，则它们被其派生类继承为派生类的静态成员
 - 这些成员通常用“<类名>::<成员名>”方式访问
 - 这些成员无论有多少个对象被创建，都只有一个拷贝，它为基类和派生类的所有对象所共享

赋值兼容性问题

- 派生类对象间的赋值操作依据下面的原则：
 - 如果派生类有自己的赋值运算符的重载定义，按重载后的运算符含义处理
 - 派生类未定义自己的赋值操作，而基类定义了赋值操作，则系统自动定义派生类赋值操作（按位拷贝），其中基类成员的赋值按基类的赋值操作进行
 - 二者都未定义专门的赋值操作，系统自动定义缺省赋值操作（按位进行拷贝）
-

赋值兼容性问题

■ 基类对象和派生类对象之间允许有下述的赋值关系：

由“大”到“小”可以，反方向不行

□ 基类对象 = 派生类对象

- 只赋“共性成员”部分

- 反方向赋值“派生类对象 = 基类对象”不被允许

□ 指向基类对象的指针 = 派生类对象的地址

- 访问非基类成员部分时，要经过指针类型的强制转换

- 下述赋值不允许：指向派生类类型的指针 = 基类对象的地址

□ 基类的引用 = 派生类对象

- 通过引用只可以访问基类成员部分

- 下述赋值不允许：派生类的引用 = 基类对象

例子

```
#include <iostream>
using namespace std;
class base{ //基类base
    int a;
public:
    base (int sa) {a = sa;}
    int geta(){return a;}
};
class derived:public base { //派生类derived
    int b;
public:
    derived(int sa, int sb):base(sa) {b=sb;}
    int getb(){return b;}
};
```

例子

```
void main ()  
    base bs1(123); // base 类对象bs1  
    cout<<"bs1.geta()="<<bs1.geta()<<endl;  
  
    derived der(246,468); // derived 类对象der  
  
    bs1=der; //OK! "基类对象 = 派生类对象"  
    cout<<"bs1.geta()="<<bs1.geta()<<endl;  
  
    der=bs1; //ERROR! "派生类对象 = 基类对象"
```

例子

```
base *pb = &der;
//“指向基类型的指针 = 派生类对象的地址”
cout<<"pb->geta( )="<<pb->geta( )<<endl;
//访问基类成员部分
cout<<pb->getb( )<<endl;
//ERROR! 直接访问非基类成员部分
cout<<"((derived *)pb)-
>getb( )="<<((derived *)pb)->getb( )<<endl;
//访问非基类成员部分时，要经过指针类型的强制转换
derived *pd = &bs1;
//ERROR! “指向派生类类型的指针=基类对象的地址”
}
```

例子

程序执行结果：

`bs1.geta()=123`

`bs1.geta()=246`

`pb->geta()=246`

`((derived *)pb)->getb()=468`

二义性问题

单继承时父类与子类间重名

- 单继承时父类与子类间成员重名时：对子类而言，不加类名限时默认为是处理子类成员，而要访问父类重名成员时，则要通过类名限定

```
class CB {  
public:  
    int a;  
    CB(int x){a=x;}  
    void showa(){  
        cout<<"Class CB -- a="<<a<<endl;  
    }  
};
```

二义性问题

```
class CD:public CB {
public:
    int a;    //与基类a同名
    CD(int x, int y):CB(x){a=y;}
    void showa(){    //与基类showa同名
        cout<<"Class CD -- a="<<a<<endl;
    }
    void print2a() {
        cout<<"a="<<a<<endl;    //子类a
        cout<<"CB::a="<<CB::a<<endl;    //父类a
    }
};
```

二义性问题

```
void main() {  
    CB CObj(12);  
    CObj.showa();  
    CD CObj(48, 999);  
    CObj.showa(); //子类的showa  
    CObj.CB::showa(); //父类的showa  
    cout<<"CObj.a="<<CObj.a<<endl;  
    cout<<"CObj.CB::a="<<CObj.CB::a<<endl;  
}
```

程序结果:

```
Class CB -- a=12  
Class CD -- a=999  
Class CB -- a=48  
CObj.a=999  
CObj.CB::a=48
```

二义性问题

多继承情况下二基类间重名

- 多继承情况下二基类间成员重名时：对子类而言，不加类名限时默认为是处理子类成员，而要访问父类重名成员时，则要通过类名限定

```
class CB1 {  
    public:  
    int a;  
    CB1(int x){a=x;}  
    void showa(){  
        cout<<"Class CB1 ==> a="<<a<<endl;}  
};
```


二义性问题

```
class CB2 {  
    public:  
        int a;  
        CB2(int x){a=x;}  
        void showa(){  
            cout<<"Class CB2 ==> a="<<a<<endl;  
        }  
};  
class CD:public CB1, public CB2 {  
    public:  
        int a; //与二基类数据成员a同名  
        CD(int x, int y, int z): CB1(x), CB2(y)  
        {a=z;}  
};
```

二义性问题

```
void showa() { //与二基类成员函数showa同名
    cout<<"Class CD ==> a="<<a<<endl;
}
void print3a() {
    //显示出派生类的a及其二父类的重名成员a
    cout<<"a="<<a<<endl;
    cout<<"CB1::a="<<CB1::a<<endl;
    cout<<"CB2::a="<<CB2::a<<endl;
}
};
```

二义性问题

```
void main() {  
    CB1 CB1obj(11);  
    CB1obj.showa();  
    CD CObj(101, 202, 909);  
    CObj.showa(); //子类showa  
    CObj.CB1::showa(); //父类showa  
    cout<<"CObj.a="<<CObj.a<<endl;  
    cout<<"CObj.CB2::a="<<CObj.CB2::a<<endl;  
}
```

程序结果:

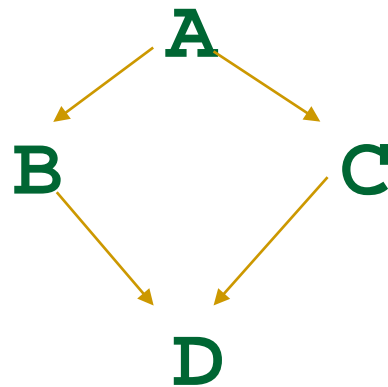
```
Class CB1 ==> a=11  
Class CD ==> a=909  
Class CB1 ==> a=101  
CObj.a=909  
CObj.CB2::a=202
```

二义性问题

多级混合继承包含两个基类实例

- 多级混合继承情况下，若类D从两条不同“路径”同时对类A进行了一般性继承，则类D的对象中会同时包含着两个类A的实例，要通过类名限定来指定访问两个类A实例中的哪一个

```
class A
class B : public A
class C : public A
class D : public B, public C
```



二义性问题

```
class A {  
public:  
    int a;  
    A(int x){a=x;}  
    void showall(){cout<<"a="<<a<<endl;}  
};  
  
class B:public A {  
public:  
    int b;  
    B(int x):A(x-1){b=x;}  
};  
  
class C:public A {  
public:  
    int c;  
    C(int x):A(x-1){c=x;}  
};
```

二义性问题

```
class D:public B,public C {  
public:  
    int d;  
    D(int x, int y, int z):B(x+1),C(y+2){d=z;}  
    void showall() {
```

//在类D定义范围内，要通过类名限定来指定访问
两个类A实例中的哪一个

```
        cout<<"C::a="<<C::a<<endl;
```

```
        cout<<"B::a="<<B::a<<endl;
```

```
        cout<<"b,c,d="<<b<<" , "<<c<<" , "<<d<<endl;
```

//b、c、d不重名，具有唯一性

```
    }  
};
```

二义性问题

```
void main() {  
    D Dobj(101, 202, 909);  
    Dobj.showall();  
  
    //访问类D的从C继承而来的a  
    cout<<"Dobj.C::a="<<Dobj.C::a<<endl;  
  
    //访问类D的从B继承而来的a  
    cout<<"Dobj.B::a="<<Dobj.B::a<<endl;  
}
```

程序结果：

C::a=203

B::a=101

b,c,d=102, 204, 909

Dobj.C::a=203

Dobj.B::a=101

虚基类

- 多级混合继承情况下，若类D从两条不同“路径”同时对类A进行了**虚拟继承**的话，则类D的对象中只包含着类A的一个实例，被虚拟继承的基类A被称为**虚基类**

虚基类的说明：在定义派生类时增加关键字virtual

```
class A
class B : virtual public A
class C : virtual public A
class D : public B, public C
```


虚基类

```
class A {  
public:  
    int a;  
    void showa() {cout<<"a="<<a<<endl;}  
};  
class B: virtual public A //对类A进行了虚拟继承  
{  
public:  
    int b;  
};
```

虚基类

```
class C: virtual public A //对类A进行了虚拟继承
{
public:
    int c;
};

class D : public B, public C
//派生类D的二基类B、C具有共同的基类A，但采用了虚
//继承，从而使类D的对象中只包含着类A的1个实例
{
public:
    int d;
};
```

虚基类

```
void main() {  
    D Dobj;    //说明D类对象  
  
    Dobj.a=11; //因为虚拟继承，D里面只包含一个a，不会出现二义性!  
  
    Dobj.b=22;  
  
    Dobj.showa(); //因为虚拟继承，D里面只包含一个showa(), 不会出现二义性!  
  
    cout<<"Dobj.b="<<Dobj.b<<endl;  
}
```

程序结果: a=11 Dobj.b=22

虚继承-深入理解

普通继承--类的内存布局:

Class A (4字节)

a;

Class B (8字节)

A::a;

b;

Class C (8字节)

A::a;

c;

```
class A {  
    public:  
        int a;  
};  
class B:public A {  
    public:  
        int b;  
};  
class C:public A {  
    public:  
        int c;  
};
```

虚继承-深入理解

普通继承--类的内存布局:

Class D (20字节)

B::A::a;

B::b;

C::A::a;

C::c;

d;

```
class D: public B, public C {  
public:  
    int d;  
};
```

虚继承-深入理解

虚继承--类的内存布局:

Class A (4 字节)

a;

Class B (12字节)

vbptr;

虚基类表指针

b;

A::a;

虚基类A

```
class A {  
public:  
    int a;  
};  
class B:virtual public A {  
public:  
    int b;  
};
```

每个虚继承的子类都有一个虚基类表指针vbptr，占用一个指针的大小，和该指针指向的一个**虚基类表**（额外的空间，不占用实例对象的空间）

虚基类	偏移地址
A	8

虚继承-深入理解

虚继承--类的内存布局:

Class A (4 字节)

a;

Class B (12字节)

vbptr;

虚基类表指针

b;

A::a;

虚基类A

```
class A {  
public:  
    int a;  
};  
class B:virtual public A {  
public:  
    int b;  
};
```

虚基类表中的偏移地址: 记录虚基类 A 与本类的偏移地址 (距离类B开始位置的字节数), 类B可以通过偏移地址找到虚基类A

虚基类	偏移地址
A	8

虚继承-深入理解

虚继承--类的内存布局:

Class A (4 字节)

a;

Class C (12字节)

vbptr;

虚基类表指针

c;

A::a;

虚基类A

```
class A {  
public:  
    int a;  
};  
class C:virtual public A {  
public:  
    int c;  
};
```

类C也是类A的子类，也有自己的虚基类指针和虚基类表

虚基类

偏移地址

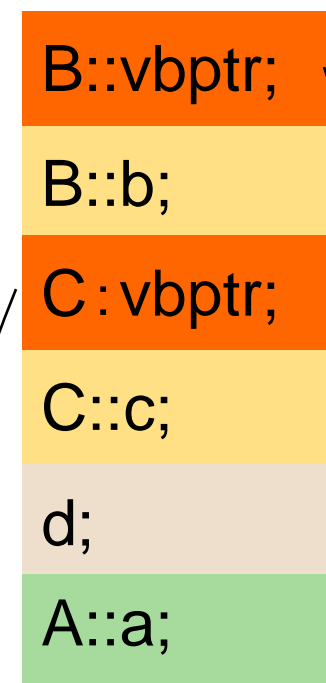
A

8

虚继承-深入理解

虚继承--类的内存布局:

Class D (24字节)



```
class D: public B, public C {  
    public:  
        int d;  
};
```

虚基类	偏移地址
A	20

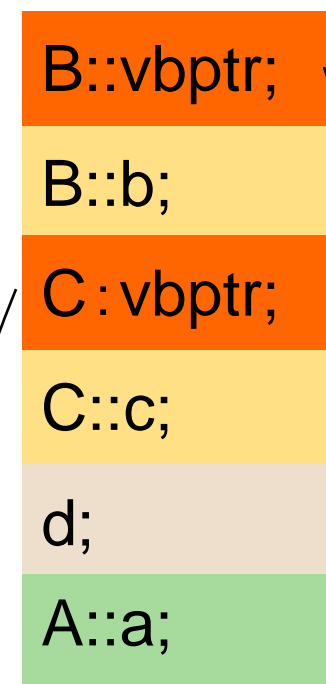
虚基类	偏移地址
A	12

当继承的子类被当作父类继承时，虚基类表指针也会被继承；例如 D 中包含从 B 和 C 继承过来的虚基类表指针

虚继承-深入理解

虚继承--类的内存布局:

Class D (24字节)



```
class D: public B, public C {  
public:  
    int d;  
};
```

虚基类	偏移地址
A	20

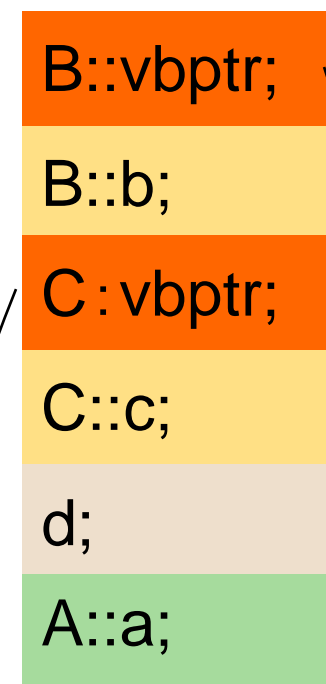
虚基类	偏移地址
A	12

类 D 多重继承类 B 和类 C, 虚基类 A 依旧会在子类中存在拷贝, 但仅仅只存在一份

虚继承-深入理解

虚继承--类的内存布局:

Class D (24字节)



```
class D: public B, public C {  
    public:  
        int d;  
};
```

虚基类	偏移地址
A	20

在D类中, B::a 和 C::a指的是同一个a, 因为通过虚基类表访问到的都是A::a

虚基类	偏移地址
A	12

函数重载(overloading)

- 允许多个不同函数使用同一个函数名，但要求这些同名函数具有不同的参数表
 - 参数表中的参数个数不同
 - 参数表中对应的参数类型不同
 - 参数表中不同类型参数的次序不同

```
int abs(int n){  
    return (n<0 ? -n : n);  
}
```

```
float abs(float n){  
    return (n<0 ? -n : n);  
}
```

函数重写/超载(overriding)

- 仅在基类与其派生类的范围内实现
- 允许多个不同函数使用**完全相同**的函数名、函数参数表以及函数返回类型

```
class graphelem {  
protected:  
    int color;  
public:  
    graphelem(int col) {  
        color=col;  
    }  
    void draw() {cout<<"graphelement";  
};
```

函数重写

```
class line:public graphelem {
    public:
    void draw(){ cout<<"draw line"; };
};
class circle:public graphelem{
    public:
    void draw(){ cout<<"draw circle" };
};
class triangle:public graphelem{
    public:
    void draw(){ cout<<"draw triangle" };
};
```

以上3个类都对draw()函数进行了重写

函数重写

```
void main() {  
    line ln1;  
    circle cir1;  
    triangle tri1;  
  
    ln1.draw(); //line::draw()  
    ln1.graphelem::draw(); //graphelem::draw()  
    cir1.draw(); //circle::draw()  
    cir1.graphelem::draw(); //graphelem::draw()  
    tri1.draw(); //triangle::draw()  
    tri1.graphelem::draw(); //graphelem::draw()  
}
```

每个类都包含两个draw()函数，通过::来进行区分

函数重写与静态绑定

```
void main() {  
    line ln1;  
    line *p1 = &ln1;  
  
    ln1->draw(); //line::draw()  
    ln1->graphem::draw(); //graphem::draw()  
  
    graphem *pg = &ln1; //基类指针指向子类对象  
    pg->draw(); //graphem::draw()  
}
```

当基类指针指向子类对象时，通过基类指针只能访问子类里面的基类部分，因此，pg->draw() 指的是基类的draw()，这是因为编译器采用**静态绑定**！

函数重写与静态绑定

```
void main() {  
    line ln1;  
    line *p1 = &ln1;  
  
    ln1->draw(); //line::draw()  
    ln1->graphelem::draw(); //graphelem::draw()  
  
    graphelem *pg = &ln1; //基类指针指向子类对象  
    pg->draw(); //graphelem::draw()  
}
```

编译器认为pg是基类 graphelem 的指针，就去查找基类 graphelem 的定义，发现确实存在draw()函数，于是将函数调用的地址绑定为基类的draw()函数的地址，此过程叫**静态绑定**

静态绑定引起的问题

```
void call_draws(graphelem *pg){
    pg->draw(); //graphelem::draw()
}

void main() {
    line ln1;
    circle cir1;
    triangle tri1;

    call_draws(&ln1);

    call_draws(&cir1);

    call_draws(&tri1);
}
```

call_draw函数的形参类型是基类指针，无论实参传入基类或是子类的类对象，pg->draw()函数都会调用基类的draw()函数，这并不是我们想要的结果！

期望结果：传入line的对象时，可以调用line::draw()；传入circle的对象时，可以调用circle::draw()；即表现出多态性

虚函数

虚函数可解决静态绑定的问题，实现**动态绑定**

- ❑ 定义基类(或其派生类)时，若将其中的某一函数成员的属性说明为**virtual**，则称该函数为**虚函数**
- ❑ 若基类中某函数被说明为虚函数，则意味着其派生类中有该函数的**重写函数**
- ❑ 在基类中定义虚函数，其派生类的**重写函数默认为虚函数**，可省略virtual关键字

虚函数

示例

```
class graphelem {  
protected:  
    int color;  
public:  
    graphelem(int col) {  
        color=col;  
    }  
    virtual void draw() {cout<<"graphelem";  
        //将draw函数定义为虚函数  
    };  
};
```

虚函数

```
class line:public graphelem {
    public:
    void draw(){
        cout<<"draw line"
    }; //重写虚函数draw, 负责画出"line"
};

class circle:public graphelem{
    public:
    void draw(){
        cout<<"draw circle"
    }; //重写虚函数draw, 负责画出"circle"
};
```

虚函数

```
class triangle:public graphelem{
    public:
    void draw(){
        cout<<"draw triangle"
    }; //重写虚函数draw, 负责画出"triangle"
};
```

动态绑定

```
void call_draws(graphelem *pg){
    pg->draw(); //根据实参动态绑定
}

void main() {
    line ln1;
    circle cir1;
    triangle tri1;

    call_draws(&ln1);

    call_draws(&cir1);

    call_draws(&tri1);
}
```

当实参传入&ln1时，pg作为基类指针指向子类的对象ln1，pg->draw()调用过程会发生**动态绑定**，调用line::draw()，而不是基类graphelem::draw()，这种现象称为**多态性**，传入&cri1和&tri1的结果同理

动态绑定-深入理解

```
class graphelem {  
protected:  
    int color;  
public:  
    virtual void draw() {cout<<"graphelem";}  
};
```

每个包含虚函数的类都拥有一个虚表(虚函数表)，虚表是一个指针数组，每个元素对应一个**虚函数的函数指针**(就是函数的调用地址)



动态绑定-深入理解

```
class graphelem {  
protected:  
    int color;  
public:  
    virtual void draw() {cout<<"graphelem";}  
};
```

graphelem类只有1个虚函数，因此虚表只包含1个元素，即指向draw()这个虚函数的指针



动态绑定-深入理解

```
class graphelem {  
protected:  
    int color;  
public:  
    virtual void draw() {cout<<"graphelem";}  
};
```

虚表内的元素(即虚函数指针)的赋值发生在编译器的编译阶段，即在代码的编译阶段，虚表就构造出来了



动态绑定-深入理解

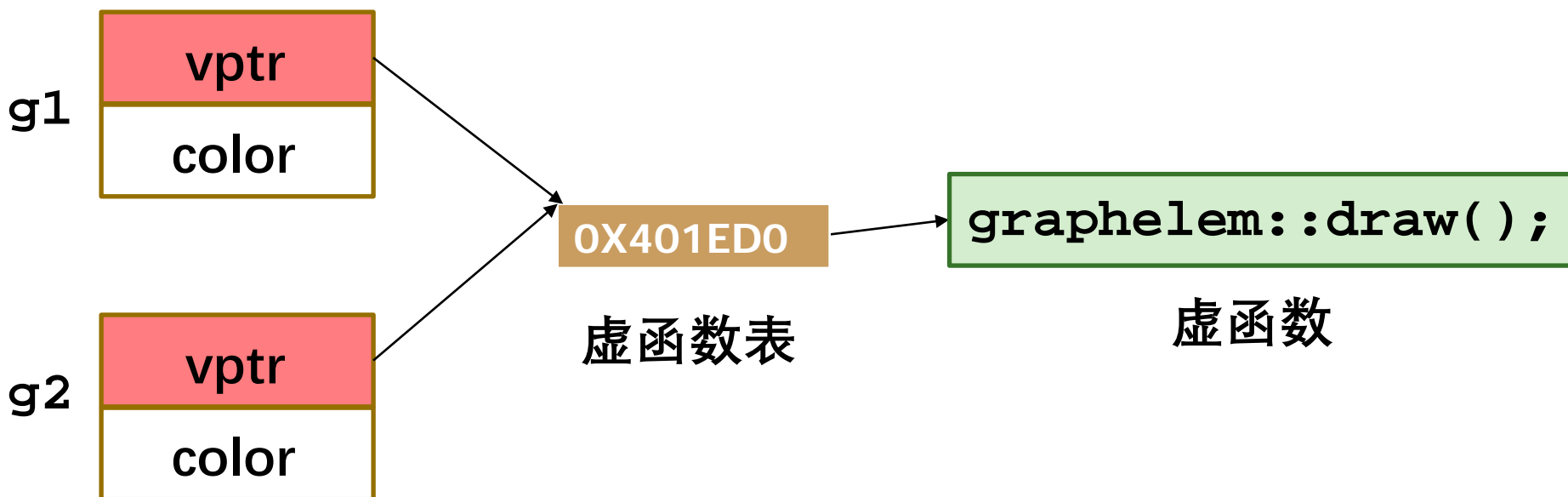
```
class graphelem {  
protected:  
    int color;  
public:  
    virtual void draw() {cout<<"graphelem";}  
};
```

虚表是属于类的，而不是属于某个具体的对象，一个类只需要一个虚表即可，同一个类的所有对象都使用同一个虚表



动态绑定-深入理解

```
graphelem g1, g2; //两个类对象
```



包含虚表的类的每个类对象都拥有一个虚表指针vptr，类对象在创建时便拥有了这个指针，这个指针的值会自动被设置为指向**该类的虚表**

动态绑定-深入理解

```
graphelem g1, *p1 = g1, &rp = g1;  
p1->draw(); //动态绑定, 通过指针  
rp.draw(); //动态绑定, 通过引用
```



当通过**指针或引用**访问虚函数时，首先读取指针所指的类对象的vptr，通过vptr找到虚表，再从虚表中读取虚函数的指针，通过函数指针调用函数，这个过程叫做**动态绑定**（动态绑定发生在程序执行阶段，非编译阶段）

例如，通过指针p1访问g1，读取g1的vptr，通过vptr找到虚表，读取虚表中的函数指针找到函数调用地址，调用函数

动态绑定-深入理解

```
graphelem g1, *p1 = g1, &rp = g1;  
p1->draw(); //动态绑定, 通过指针  
g1.draw(); //静态绑定
```



注意：通过类对象(非指针或非引用)访问虚函数时，不会发生动态绑定，因为编译器可以通过对象的类型在编译阶段就可以确定调用的函数(即静态绑定)

动态绑定-深入理解

```
class line:public graphelem {  
    public:  
    void draw(){  
        cout<<"draw line"  
    }; //虚函数draw, 负责画出"line"  
};
```

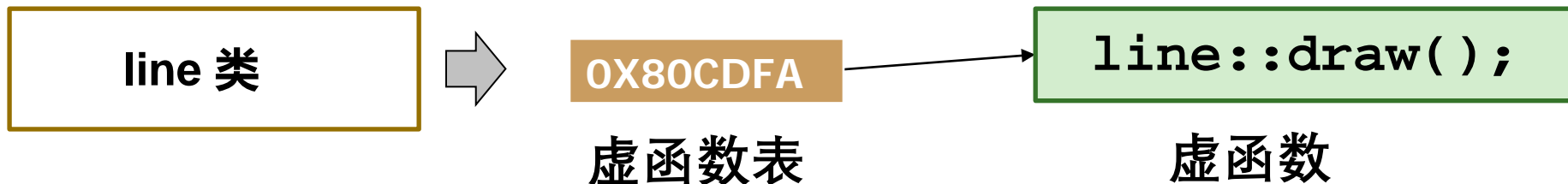
一个派生类的基类如果包含虚函数，那个这个派生类也拥有自己的虚表，**虚表的构建过程如下**：(1)先把父类虚表里的虚函数放入虚表；(2)将自己新定义的虚函数放入虚表；(3)如果父类虚表里的虚函数在本类里被重写了，那么删掉这个虚函数，只保留重写后的

动态绑定-深入理解

```
class line:public graphelem {  
    public:  
    void draw() {  
        cout<<"draw line"  
    }; //虚函数draw, 负责画出"line"  
};
```

虽然draw()没有加virtual, 但因为父类里面draw()是虚函数, 那么子类中的draw()自动变为虚函数

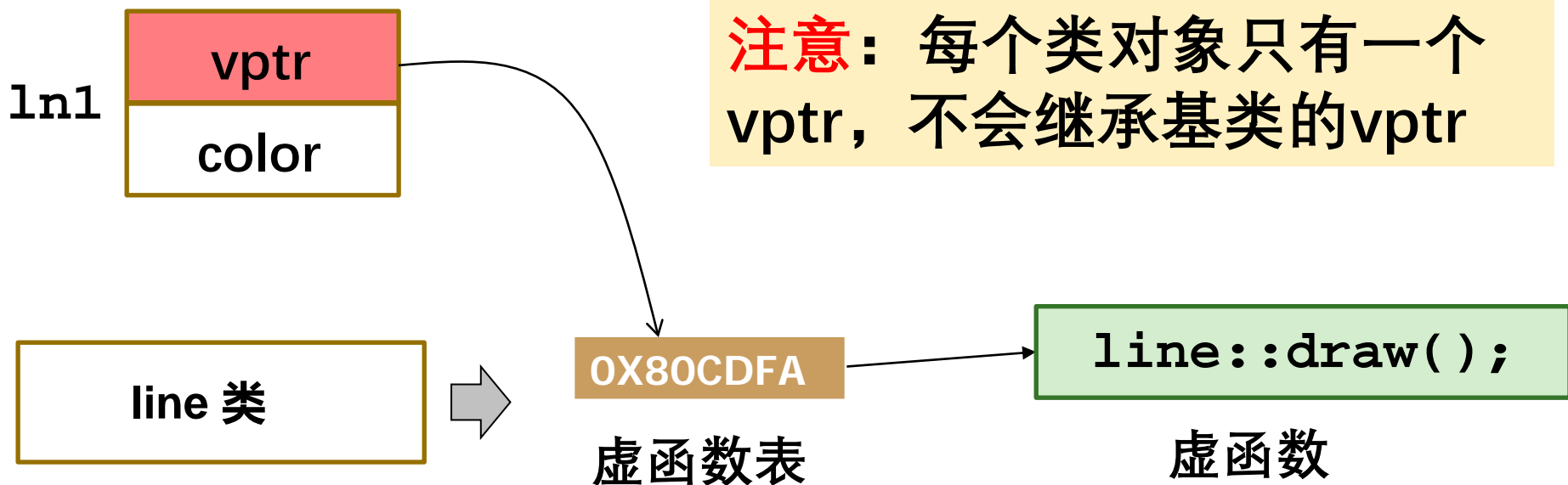
派生类line的虚表的构建过程如下: (1)先把父类虚表里的虚函数graphelem::draw()放入虚表; (2)将自己新定义的虚函数draw()放入虚表; (3)因为graphelem::draw()被自己定义的draw()重写了, 只保留重写后的draw()



动态绑定-深入理解

```
graphelem *pg;  
line ln1;  
pg = &ln1;  
pg->draw();
```

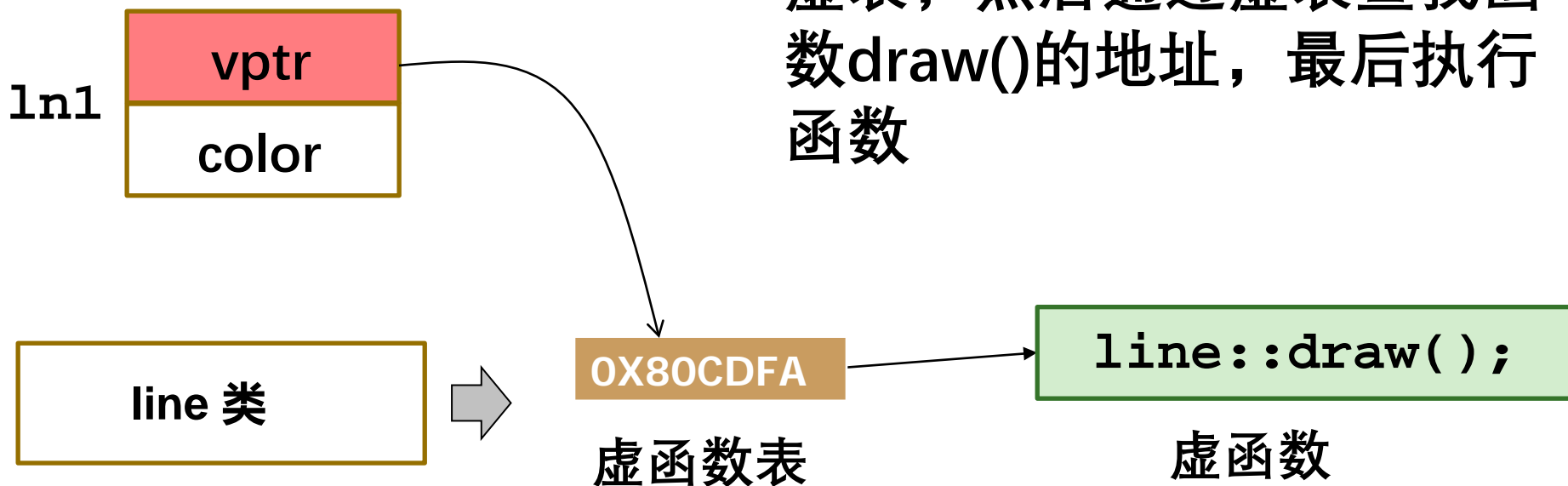
(1) 子类对象ln1生成时，ln1里面的vp_ptr被赋值，指向line类的虚表



动态绑定-深入理解

```
graphelem *pg;  
line ln1;  
pg = &ln1;  
pg->draw();
```

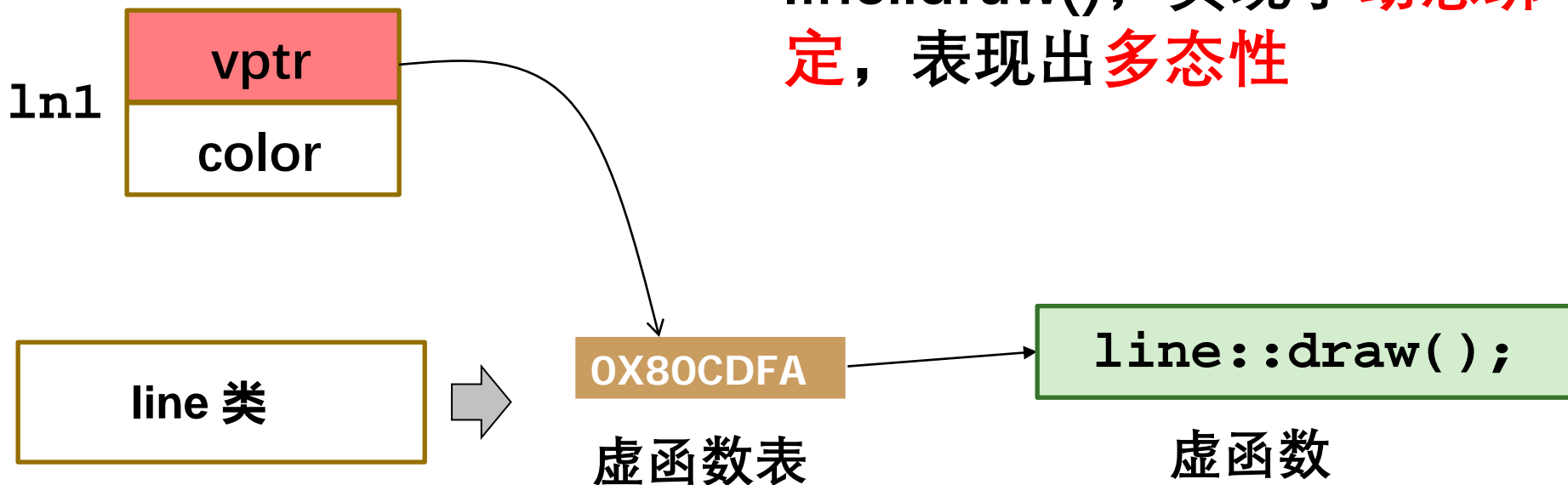
(2) 通过基类指针pg调用函数draw()时，先找到指针所指向的类对象(ln1)里面的vptr，通过vptr找到对应的虚表，然后通过虚表查找函数draw()的地址，最后执行函数



动态绑定-深入理解

```
graphelem *pg;  
line ln1;  
pg = &ln1;  
pg->draw();
```

(3) 因为pg指向的是一个子类对象，vptr指向的是子类line的虚表，因此，调用的draw()函数是子类重写后的line::draw()，实现了**动态绑定**，表现出**多态性**



动态绑定发生的条件

- 必须把函数定义为类的虚函数
- 必须通过指针或引用调用虚函数

```
graphelem gr1;  
line ln1;  
gr1.draw(); //非指针或引用调用函数, 静态绑定  
ln1.draw(); //非指针或引用调用函数, 静态绑定  
  
graphelem *pg1 = &gr1;  
graphelem *pg2 = &ln1;  
pg1->draw(); //动态绑定  
pg2->draw(); //动态绑定  
line *p11 = &ln1;  
line & r11 = ln1;  
p11->draw(); //动态绑定  
r11.draw(); //动态绑定
```

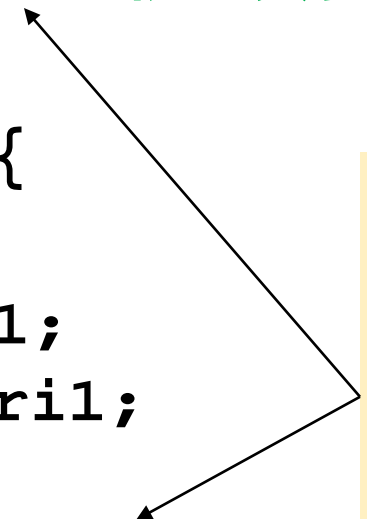
多态性发生的条件

- 必须把函数定义为类的虚函数
 - 类之间应满足子类型关系，通常表现为一个类从另一个类公有派生而来
 - 必须先使用**基类指针(或引用)指向子类的对象**，然后使用基类指针或者引用调用虚函数
-

多态性发生的条件

```
void call_draws(graphelem *pg){  
    pg->draw(); //根据实参动态绑定  
}
```

```
void main() {  
    line ln1;  
    circle cir1;  
    triangle tri1;  
  
    call_draws(&ln1);  
  
    call_draws(&cir1);  
  
    call_draws(&tri1);  
}
```

- 
- (1) draw是虚函数
 - (2) graphelem和line之间存在继承关系
 - (3) pg作为基类指针，当实参传入&ln1 时，pg指向子类的对象ln1

满足多态发生的3个条件

动态绑定总结

- 在程序运行时动态地进行，根据当时的情况来确定调用哪个同名函数（静态绑定是在编译阶段完成）
 - 当涉及到多态性和虚函数时应该使用动态绑定（非虚函数都是静态绑定）
 - 动态绑定的优点是灵活性强，但效率低（静态绑定灵活性差，但效率高）
-

动态绑定-更复杂的例子

```
class A {  
public:  
    virtual void vfunc1(){cout<<"A::vfunc1"<<endl;}  
    virtual void vfunc2(){cout<<"A::vfunc2"<<endl;}  
    void func1(){cout<<"A::func1"<<endl;}  
    void func2(){cout<<"A::func2"<<endl;}  
private:  
    int m_data1, m_data2;  
};
```

A有4个函数，但只有两个是虚函数，因此虚表中包含两项

vfunc1	虚函数
vfunc2	虚函数
func1	非虚函数
func2	非虚函数

动态绑定-更复杂的例子

```
class A {  
public:  
    virtual void vfunc1(){cout<<"A::vfunc1"<<endl;}  
    virtual void vfunc2(){cout<<"A::vfunc2"<<endl;}  
    void func1(){cout<<"A::func1"<<endl;}  
    void func2(){cout<<"A::func2"<<endl;}  
private:  
    int m_data1, m_data2;  
};
```



动态绑定-更复杂的例子

```
class B: public A {  
public:  
    virtual void vfunc1(){cout<<"B::vfunc1"<<endl;}  
    void func1(){cout<<"B::func1"<<endl;}  
private:  
    int m_data3;  
};
```

B有6个函数，从A继承过来4个，自己新定义2个

A::vfunc1	虚函数
A::vfunc2	虚函数
A::func1	非虚函数
A::func2	非虚函数
vfunc1	虚函数，对A::vfunc1的重写
func1	非虚函数

动态绑定-更复杂的例子

```
class B: public A {  
public:  
    virtual void vfunc1(){cout<<"B::vfunc1"<<endl;}  
    void func1(){cout<<"B::func1"<<endl;}  
private:  
    int m_data3;  
};
```

B有6个函数，从A继承过来4个，自己新定义2个，其中虚函数3个，虚函数中A::vfunc1()被重写，不会出现在虚表中



动态绑定-更复杂的例子

```
class C: public B {  
public:  
    virtual void vfunc2(){cout<<"C::vfunc2"<<endl;}  
    void func2(){cout<<"C::func2"<<endl;}  
private:  
    int m_data1, m_data4;  
};
```

C有8个函数，从B继承过来6个，自己新定义2个

B::A::vfunc1	虚函数
B::A::vfunc2	虚函数
B::A::func1	非虚函数
B::A::func2	非虚函数
B::vfunc1	虚函数，对A::vfunc1的重写
B::func1	非虚函数
vfunc2	虚函数，对B::vfunc2的重写
func2	非虚函数

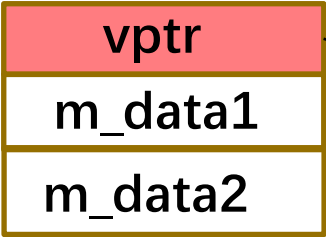
动态绑定-更复杂的例子

```
class C: public B {  
public:  
    virtual void vfunc2(){cout<<"C::vfunc2"<<endl;}  
    void func2(){cout<<"C::func2"<<endl;}  
private:  
    int m_data1, m_data4;  
};
```

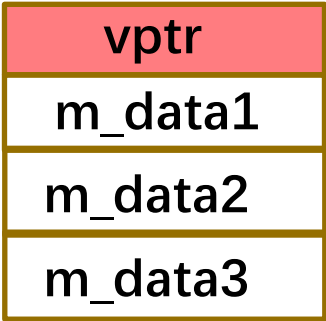
C有8个函数，从B继承过来6个，自己新定义2个；其中虚函数4个，虚函数中A::vfunc1和B::vfunc2被重写，不会出现在虚表中



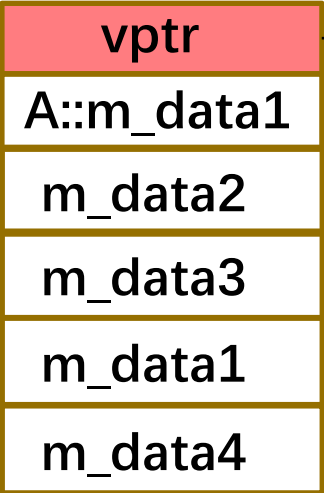
class A 的对象



class B 的对象



class C 的对象



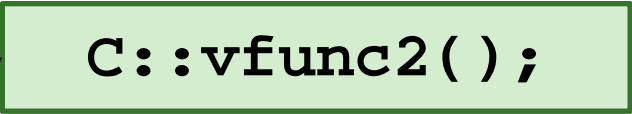
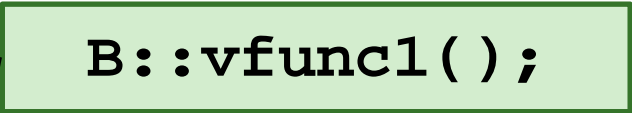
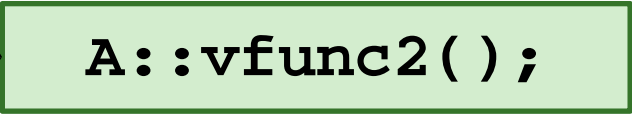
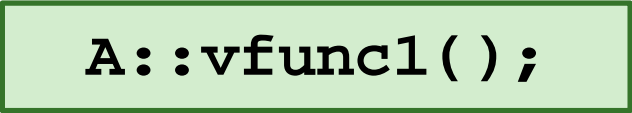
A的虚函数表



B的虚函数表



C的虚函数表



虚函数

动态绑定-更复杂的例子

```
int main() {  
    A a;  
    a.func1(); //静态绑定  
    a.func2(); //静态绑定  
    a.vfunc1(); //静态绑定  
    a.vfunc2(); //静态绑定
```

A有4个函数，2个普通函数，
2个虚函数

```
    B b;  
    b.A::func1(); //A::func1(); 静态绑定  
    b.A::func2(); //A::func2(); 静态绑定  
    b.A::vfunc1(); //A::vfunc1(); 静态绑定  
    b.A::vfunc2(); //A::vfunc2(); 静态绑定  
    b.vfunc1(); //B::vfunc1(); 静态绑定  
    b.func1(); //B::func1(); 静态绑定
```

B有6个函数，
从A继承了4个
，自己定义了
2个

动态绑定-更复杂的例子

```
C c;  
c.B::A::func1();  
c.B::A::vfunc1();  
c.B::A::func2();  
c.B::A::vfunc2();  
c.B::vfunc1();  
c.B::func1();  
c.func2();  
c.vfunc2();  
return 0;  
}
```

C有8个函数，从B继承了6个，自己定义了2个

全部是静态绑定

结果：

A::func1
A::func2
A::vfunc1
A::vfunc2
A::func1
A::func2
A::vfunc1
A::vfunc2
B::vfunc1
B::func1
A::func1
A::vfunc1
A::func2
A::vfunc2
B::vfunc1
B::func1
C::func2
C::vfunc2

在没有发生动态绑定时，虚函数和普通函数没有任何区别

动态绑定-更复杂的例子

```
int main() {
```

```
    A a;
```

```
    B b;
```

```
    A *pA = &b;
```

```
    pA->vfunc1();
```

pA指向类B的对象，B的虚表中有vfunc1()，因此，发生动态绑定，调用B::vfunc1()

```
    pA = &a;
```

```
    pA->vfunc1();
```

pA指向类A的对象，A的虚表中有vfunc1()，因此，发生动态绑定，调用A::vfunc1()

```
    a.vfunc1(); //静态绑定
```

```
    return 0;
```

```
}
```

结果:

B::vfunc1

A::vfunc1

```

int main() {
    A a;
    B b;
    A *pA = &b;
    pA->vfunc1();

    pA = &a;
    pA->vfunc1();

    a.vfunc1();
    return 0;
}

```

```

call    0x41c2f0 <__main>
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d898 <A::A()>
mov     %esp, %eax
mov     %eax, %ecx
call    0x42d8f0 <B::B()>
mov     %esp, %eax

```

```

mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax //函数指针

```

```

lea     0x10(%esp), %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax //函数指针

```

```

lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d838 <A::vfunc1()>
mov     $0x0, %eax
leave
ret

```

//函数名

```
A a;
```

```
B b;
```

```
A *pA = &b;
```

```
pA->vfunc1();
```

将虚函数表指针的地址放入寄存器eax

```
call    0x41c2f0 <__main>
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d898 <A::A()>
mov     %esp, %eax
mov     %eax, %ecx
call    0x42d8f0 <B::B()>
mov     %esp, %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax //函数指针
lea     0x10(%esp), %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d838 <A::vfunc1()>
mov     $0x0, %eax
leave
ret
```

```
A a;
```

```
B b;
```

```
A *pA = &b;
```

```
pA->vfunc1();
```

取出虚函数表指针，放入寄存器eax中

```
call    0x41c2f0 <__main>
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d898 <A::A()>
mov     %esp, %eax
mov     %eax, %ecx
call    0x42d8f0 <B::B()>
mov     %esp, %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax //函数指针
lea     0x10(%esp), %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d838 <A::vfunc1()>
mov     $0x0, %eax
leave
ret
```

```
A a;
```

```
B b;
```

```
A *pA = &b;
```

```
pA->vfunc1();
```

按照虚函数表指针，取出虚函数表中虚函数的地址，放入eax中

```
call    0x41c2f0 <__main>
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d898 <A::A()>
mov     %esp, %eax
mov     %eax, %ecx
call    0x42d8f0 <B::B()>
mov     %esp, %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax //函数指针
lea     0x10(%esp), %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d838 <A::vfunc1()>
mov     $0x0, %eax
leave
ret
```

```
A a;
```

```
B b;
```

```
A *pA = &b;
```

```
pA->vfunc1();
```

从eax中取出虚函数指针，调用函数

```
call    0x41c2f0 <__main>
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d898 <A::A()>
mov     %esp, %eax
mov     %eax, %ecx
call    0x42d8f0 <B::B()>
mov     %esp, %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax //函数指针
lea     0x10(%esp), %eax
mov     %eax, 0x1c(%esp)
mov     0x1c(%esp), %eax
mov     (%eax), %eax
mov     (%eax), %eax
mov     0x1c(%esp), %edx
mov     %edx, %ecx
call    *%eax
lea     0x10(%esp), %eax
mov     %eax, %ecx
call    0x42d838 <A::vfunc1()>
mov     $0x0, %eax
leave
ret
```

纯虚函数

- 如果不准备在基类的虚函数中做任何事情，则可使用如下的格式将该虚函数说明成**纯虚函数**：

virtual <函数原型> = 0;

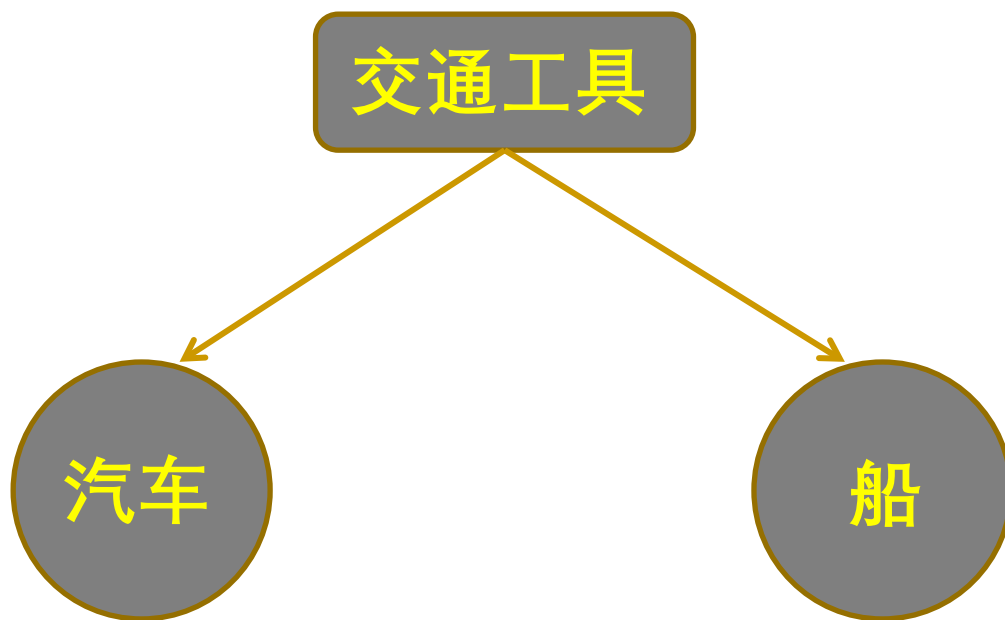
- 纯虚函数**不能被直接调用**，它只为其派生类的各虚函数规定了一个一致的“原型规格”，该虚函数的实现将在它的派生类中给出

抽象基类

- 含有纯虚函数的基类称为**抽象基类**
 - 不可使用抽象基类来说明并创建它自己的对象，只有在创建其派生类对象时，才有抽象基类自身的实例伴随而生
 - 如果一个抽象基类的派生类中没有定义基类中的纯虚函数、而只是继承了基类之纯虚函数的话，则这个派生类还是一个抽象基类
-

例子

虚函数、动态联编、纯虚函数与抽象基类综合示例



例子

```
class Vehicle //交通工具, 基类
{
    public:
        Vehicle(int w) {
            weight = w;
        }
        virtual void ShowMe() {
            cout<<"我是交通工具！重量为"<<weight<<"吨";
        }
        protected:
            int weight;
};
```

例子

```
class Car: public Vehicle//汽车，派生类
{
    public:
        Car(int w,int a):Vehicle(w){
            aird = a;
        }
        virtual void ShowMe() {
            cout<<"我是汽车！排气量为" <<aird <<"CC";
        }
    protected:
        int aird;
};
```

例子

```
class Boat: public Vehicle//船, 派生类
{
    public:
        Boat(int w,float t):Vehicle(w) {
            tonnage = t;
        }
        virtual void ShowMe() {
            cout<<"我是船！排水量为"<<tonnage<<"吨";
        }
    protected:
        float tonnage;
};
```

例子

```
int main(){
    Vehicle *pv = new Vehicle(10);
    pv ->ShowMe();
    Car c(15,200);
    Boat b(20,1.25f);
    pv = &c;
    pv->ShowMe(); //基类指针，实现动态绑定
    Vehicle v (10); //创建一个基类对象
    v = b;           //将派生类对象赋值给基类对象
    v.ShowMe(); //基类对象访问虚函数，未实现动态绑定
    pv = &b;
    pv -> ShowMe(); //基类指针，实现动态绑定
    return 0;
}
```

例子

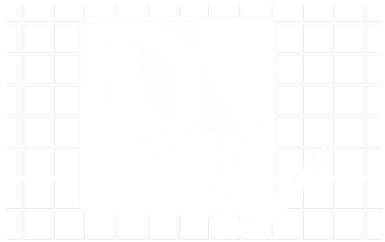
将ShowMe()函数改为纯虚函数

```
class Vehicle //交通工具, 基类
{
    public:
        Vehicle(int w) {
            weight = w;
        }
        virtual void ShowMe() = 0;
        //纯虚函数, Vehicle类成为抽象基类
        //不能够创建Vehicle类的对象
    protected:
        int weight;
};
```

例子

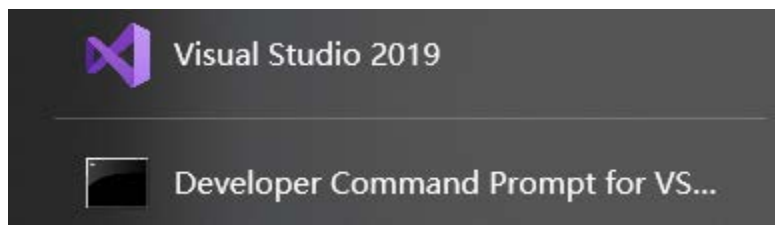
```
int main(){
    Vehicle *pv = new Vehicle(10); //ERROR
    pv ->ShowMe(); //ERROR, 纯虚函数不能直接调用
    Car c(15,200);
    Boat b(20,1.25f);
    Vehicle * pv = &c; //用派生类对象进行初始化
    pv->ShowMe(); //基类指针，实现动态绑定
    pv = &b;
    pv -> ShowMe(); //基类指针，实现动态绑定
    return 0;
}
```

END



如何利用VS2019获得类的内存布局

1. 打开Developer Command Prompt for VS 2019工具(VS2019自带，在Windows->程序列表可以找到)



```
Developer Command Prompt for VS 2019
*****
** Visual Studio 2019 Developer Command Prompt v16.10.1
** Copyright (c) 2021 Microsoft Corporation
*****
D:\Visual Studio\Visual Studio2019Community>_
```

<https://blog.csdn.net/whhcs>

如何利用VS2019获得类的内存布局

2. 通过命令行，跳转到程序(main.cpp)所在的文件夹下，例如，程序在D:\Visual Studio\practice\test下面：

```
D:\Visual Studio\Visual Studio2019Community>cd D:\Visual Studio\practice\test
```

```
D:\Visual Studio\practice\test>dir
```

驱动器 D 中的卷是 Data

卷的序列号是 FAFE-6A74

D:\Visual Studio\practice\test 的目录

2021/07/09	12:08	<DIR>	.
2021/07/09	12:08	<DIR>	..
2021/04/22	15:18		229 CPPclass.cpp
2021/07/09	12:08	<DIR>	Debug
2021/04/22	15:18		262 Point.h
2021/07/09	12:08		362 test.cpp
2021/04/16	09:06		1,428 test.sln
2021/05/07	01:56		7,243 test.vcxproj
2021/05/07	01:56		966 test.vcxproj.filters
2021/04/16	09:06		168 test.vcxproj.user
2021/04/20	21:39	<DIR>	x64

7 个文件 10,658 字节
4 个目录 68,113,137,664 可用字节

<https://blog.csdn.net/whhcs>

如何利用VS2019获得类的内存布局

3. 输入 `cl /d1 reportSingleClassLayout`类名 文件名

例如：`cl /d1 reportSingleClassLayoutBoss test.cpp`，其中Boss是类名，test.cpp是文件名

```
class Boss      size(28):
+---
0      +--- (base class Monster)
0      {vfptr}
4      M_rank
8      M_exp
12     M_hp
16     M_damage
20     M_money
+---
24     count
+---

Boss::$vftable@:
    &Boss_meta
    0
0      &Boss::display
1      &Boss::lose
```

The diagram illustrates the memory layout of the `Boss` class. Red arrows point from Chinese labels to specific parts of the code output:

- 字节** (Byte) points to `size(28):`.
- 基类** (Base Class) points to `(base class Monster)`.
- 虚函数指针** (Virtual Function Pointer) points to `{vfptr}`.
- 基类中的成员变量** (Member Variables in Base Class) points to the variables `M_rank`, `M_exp`, `M_hp`, `M_damage`, and `M_money`.
- 自身的成员变量** (Own Member Variables) points to `count`.
- 虚函数列表** (Virtual Function List) points to the `Boss::$vftable@:` section.