

C++ Programming

Chapter 11 Inheritance and Derivation

Part 2/2

Zheng Guibin
(郑贵滨)

目录

Inheritance and Derivation

继承与派生

CONTENT

- 继承概述
- 基类和派生类
- 派生类的构造与析构
- 多重继承



11. 继承与派生

1 继承概述

2 基类和派生类

3 派生类的构造与析构

4 多重继承



哈爾濱工業大學

Harbin Institute of Technology

11.3 派生类的构造与析构

基类的构造/析构函数不被继承

- 派生类中需要声明自己的构造/析构函数。
- 定义派生类构造函数时，只需要对类中新增成员进行初始化，对继承来的基类成员的初始化，自动调用基类构造函数完成。
- 派生类的构造函数需要给基类的构造函数传递参数
- C++提供一种机制，在创建派生类对象时调用基类的构造函数来初始化基类数据。
- 执行派生类的析构函数时，基类的析构函数也将被调用

11.3 派生类的构造与析构

◆ 构造函数执行顺序：

基类 → 派生类中对象成员 → 派生类

先祖先，再客人，后自己

◆ 析构函数执行顺序：

- 与构造函数相反

◆ 派生类的构造函数定义

派生类名::派生类名(形参表):基类名1(参数), 基类名2(参数), ...基类名n(参数), 本类对象成员和基本类型成员初始化列表

{

// 其他初始化

};



11.3 派生类的构造与析构

◆ 复制构造函数

- 若建立派生类对象时没有编写复制构造函数，编译器会生成一个隐含的复制构造函数，该函数先调用基类的复制构造函数，再为派生类新增的成员对象执行拷贝。
- 若编写派生类的复制构造函数，则需要为基类相应的复制构造函数传递参数。

例如：

C::C(const C &c1): B(c1) {...}



调用构造函数顺序测试，构造函数无参数

```
#include <iostream.h>

class Base
{
public :
    Base () { cout << "Base is created.\n" ; }
    ~Base () { cout << "Base is deleted!\n" ; }
};

class Derived : public Base
{
public :
    Derived() { cout << "Derived is created.\n" ; }
    ~Derived() { cout << "Derived is deleted!\n" ; }
};

void main ()
{   Derived d ;   cout<<"..."<<endl; }
```

11.3 派生类的构造与析构(带参数)

```
#include <iostream.h> // 带参数构造函数调用顺序
```

```
class parent_class  
{
```

```
    int private1, private2;
```

```
public :
```

```
    parent_class ( int p1, int p2 ):private1 ( p1 ), private2 ( p2 ) {}
```

```
    int inc1 () { return ++ private1; }
```

```
    int inc2 () { return ++ private2; }
```

```
    void display ()
```

```
    { cout << "private1=" << private1  
        << ", private2=" << private2 << endl ; }
```

```
};
```

基类有一个
参数化的构造函数

parent_class(p1, p2) , obj4(p3 , p4)

改为obj4(p3 , p4) , parent_class(p1 , p2)

构造顺序不变

// 带参数构造函数调用顺序

类成员

class derived_class : private parent_class

{ int private3 ;

派生类有参数构造函数

parent_class obj4 ;

public:

derived_class (int p1 , int p2 , int p3 , int p4 , int p5):

parent_class(p1 , p2), obj4(p3 , p4), private3 (p5){}

int inc1(){ return parent_class :: inc1(); }

int inc3(){ return ++ private3 ; }

void display()

调用基类构造函数

{ parent_class :: display();

构造对象成员

obj4.display ();

cout << "private3=" << private3 << endl ; }

} ;

派生类的构造与析构

- 1) 派生类构造函数的定义中可省略对基类构造函数的调用，其条件是在基类中必须有**默认的构造函数**或者根本没有定义构造函数。
- 2) 当基类的构造函数使用一个或多个参数时，则派生类**必须**定义构造函数，提供将参数传递给基类构造函数途径。有时派生类构造函数体可能为空，仅起到参数传递作用。

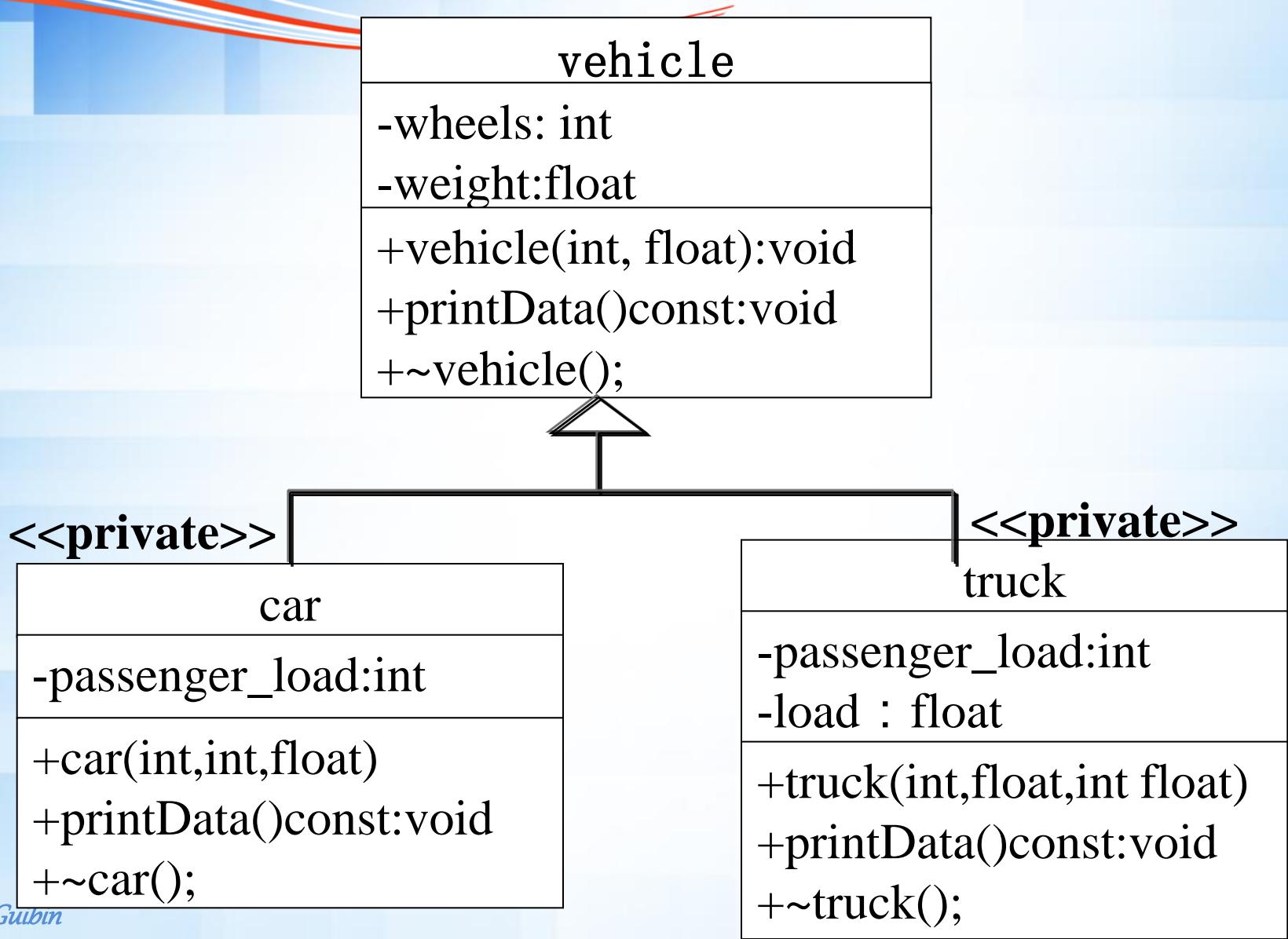


11.3 派生类的构造与析构(带参数)

//带参数构造函数调用顺序

```
class parent_class class
{...} ;
derived_class : private parent_class
{ ... } ;

void main ()
{
    derived_class d1 ( 18, 18 , 1 , 2 , -5 ) ;
    d1 . inc1 () ;
    d1 . display () ;
}
```



练习

- ◆ 1. 派生类构造函数执行的次序是怎样的？析构函数呢？
- ◆ 2. 派生类的构造函数的成员初始化列表中，不能包含（ ）。
 - A 基类的构造函数
 - B 派生类中对象成员的初始化
 - C 基类的对象成员初始化
 - D 派生类中一般数据成员的初始化



◆ 试写出下述程序的执行结果

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

class A
{
private:
    int a,b;
public:
    A(int i,int j){a=i;b=j;}
    void move(int x,int y){a+=x;b+=y;}
    void show(){cout<<"("<<a<<","<<b<<")"<<endl;}
};

class B:public A
{
private :
    int x,y;
public:
    B(int i,int j,int k,int l):A(i,j),x(k),y(l){}
    void show(){cout<<x<<","<<y<<endl;}
    void fun(){move(3,5);}
    void f1(){A::show();}
};

void main()
{
    A e(1,2);
    e.show();
    B d(3,4,5,6);
    d.fun();
    d.A::show();
    d.B::show();
    d.f1();
}
```

```
void main()
{
    A e(1,2);
    e.show();
    B d(3,4,5,6);
    d.fun();
    d.A::show();
    d.B::show();
    d.f1();
}
```

11. 继承与派生

1 继承概述

2 基类和派生类

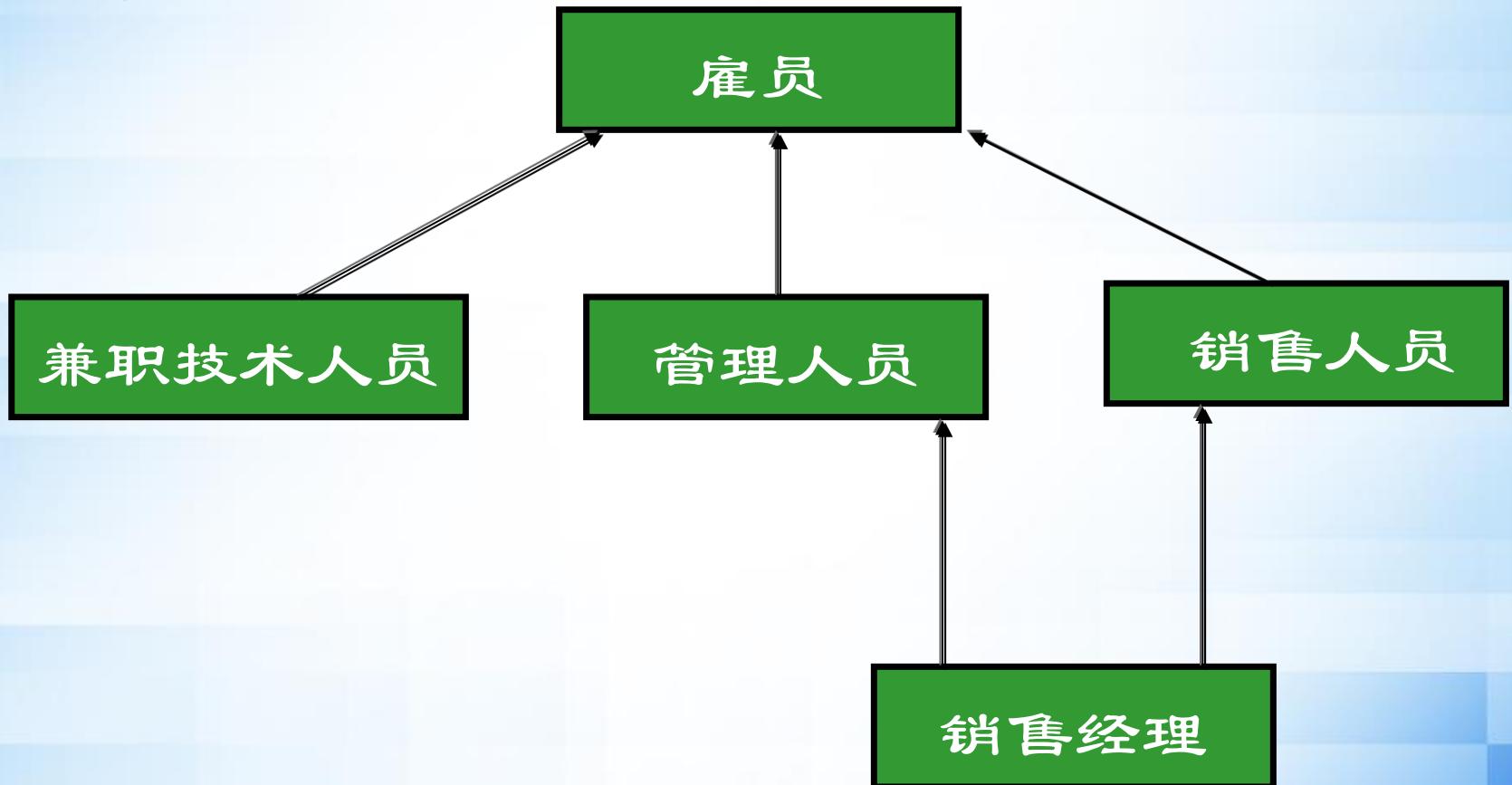
3 派生类的构造与析构

4 多重继承



11.4 多重继承

举例



11.4 多重继承

- 一个类有多个直接基类的继承关系称为多重继承
- 多重继承声明语法

class 派生类名 : 访问控制 基类名₁, ..., 访问控制 基类名_n

{

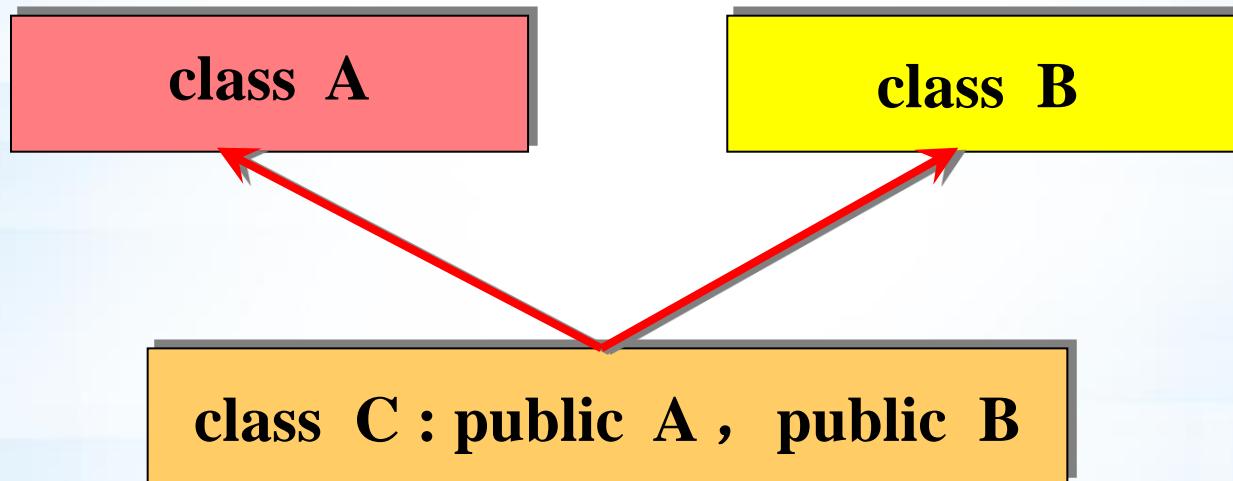
数据成员和成员函数声明

};



11.4 多重继承

类 C 可以同时继承类 A 和类 B 的成员，并添加自己的成员



11.4 多重继承

```
class Base1
{ public:
    Base1(int x) { value = x; }
    int getData() const { return value; }
protected:
    int value;
};
```

```
class Base2
{ public:
    Base2(char c) { letter=c; }
    char getData() const { return letter; }
protected:
    char letter;
};
```

11.4 多重继承

```
class Derived : public Base1, public Base2
```

```
{
```

```
public :
```

```
    Derived ( int, char, double ) ;
```

```
    double getReal() const ;
```

```
private :
```

```
    double real ;
```

```
};
```

```
void main()
```

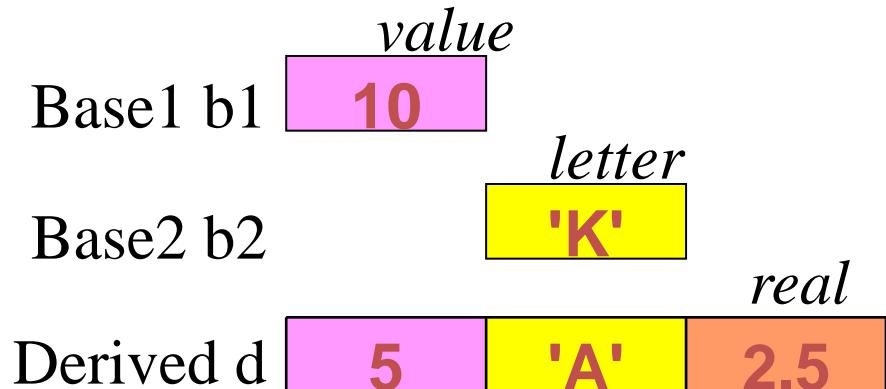
```
{
```

```
    Base1 b1 ( 10 ) ;
```

```
    Base2 b2 ( 'k' ) ;
```

```
    Derived d ( 5, 'A', 2.5 ) ;
```

```
}
```



11.4 多重继承的派生类构造

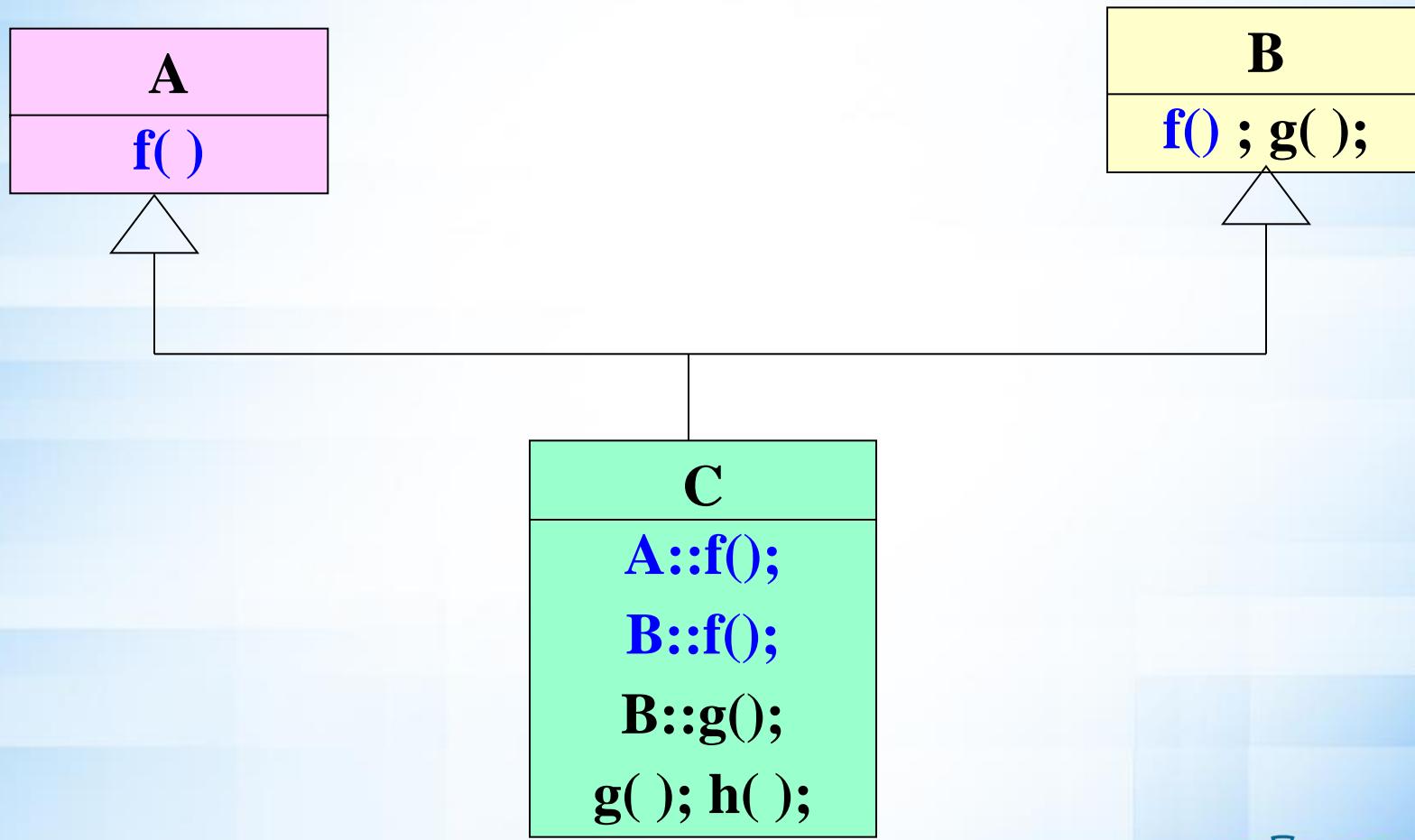
- ◆ 多个基类的派生类构造函数用初始化列表调用基类构造函数，执行顺序与单继承构造函数情况类似。
- ◆ 多个直接基类构造函数执行顺序取决于定义派生类时指定的各个继承基类的顺序。按基类在被继承时所声明的次序从左到右依次调用。

11.4 多重继承的派生类访问

- ◆ 一个派生类对象拥有多个直接或间接基类的成员。
- ◆ 不同名成员访问不会出现二义性。
- ◆ 如果不同的基类有同名成员，派生类对象访问时应该加以识别。
- ◆ 由于多重继承情况下，可能造成对基类中某个成员的访问出现了不惟一的情况，则称为对基类成员访问的二义性问题。



多重继承图示



11.4 多重继承——二义性和支配原则

```
class A
{
public:
    void f();
};
```

```
class B
{ public:
    void f();
    void g();
};
```

```
class C: public A,public B
{   public:
    void g();
    void h();
};
```

若有C obj;
则对函数 f()的访问obj.f()是二义的



11.4 多重继承——二义性和支配原则

1. 同名成员的二义性

- ◆ 不同基类中有同名函数，使用基类名可避免这种二义：

```
obj.A::f();  
obj.B::f();
```

```
void C :: f ()  
{   A :: f (); // B :: f (); }
```

- ◆ 基类与派生类同名函数

```
obj.g();      //隐含用C的g( )  
obj.B::g(); //用B的g( )
```

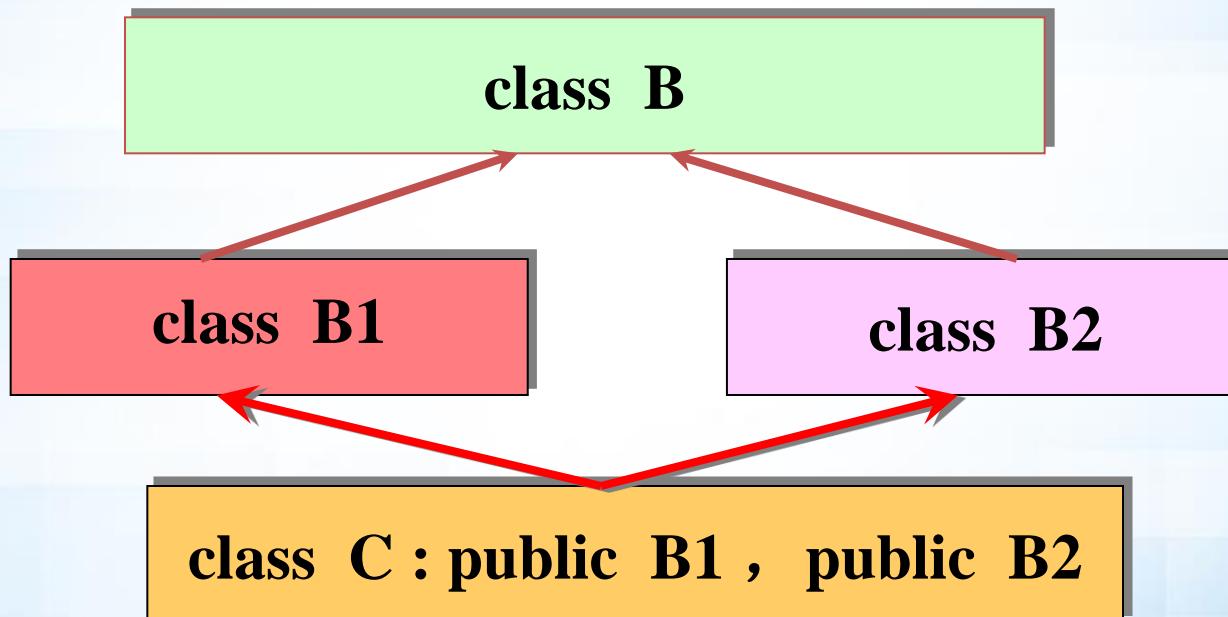
- ◆ 这种用基类名来控制成员访问的规则称为**支配原则**。



11.4 多重继承——二义性和支配原则

2. 同一基类被多次继承产生的二义性

一个类不能从同一类直接继承二次或更多次。



11.4 多重继承——二义性和支配原则

```
class B { public : int b ; }

class B1 : public B {int b1 ; }

class B2 : public B {int b2 ; }

class C : public B1 , public B2
```

```
{ public : int f () ;

private : int d ; }
```

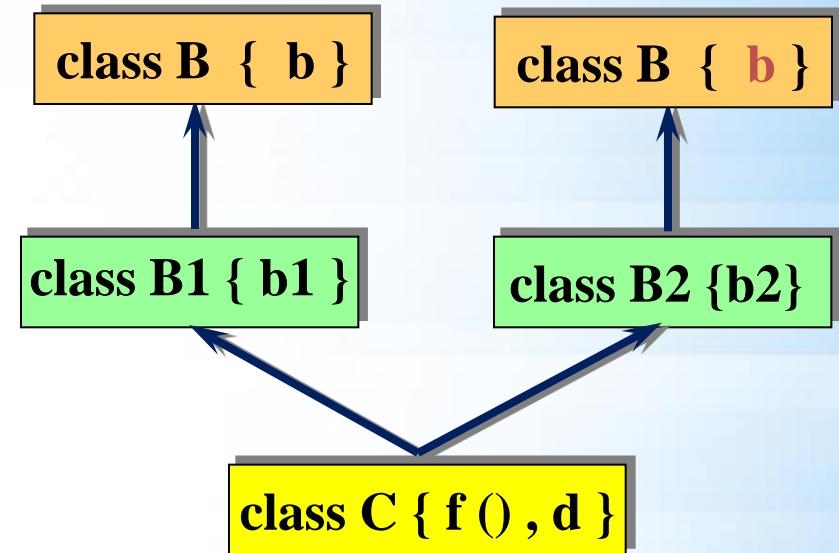
C obj ;

obj . b; // error

obj. B::b ; // error, 从哪里继承的?

obj. B1::b ; // OK

obj. b1 ;



obj.B1 :: b

obj.B2 :: b

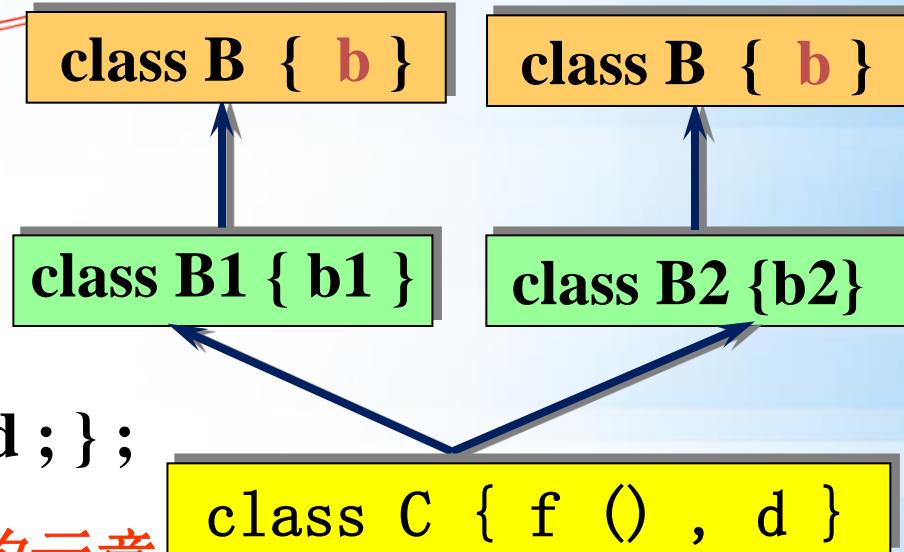
obj. b1 ;

// OK

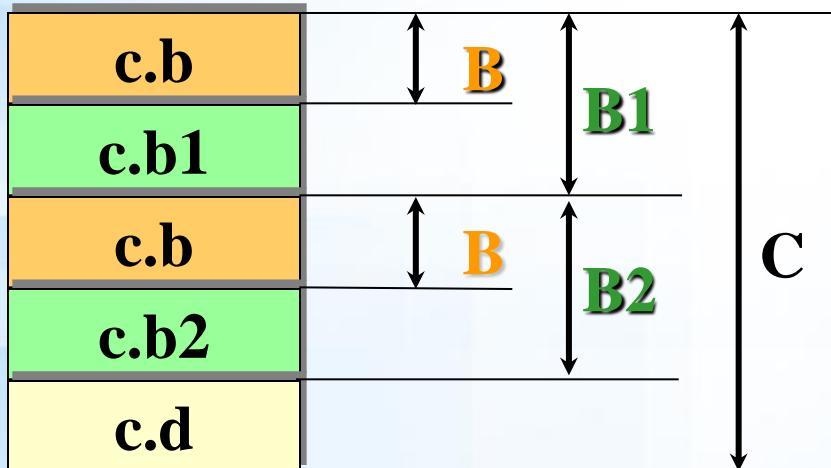
11.4 多重继承——二义性和支配原则

例如：

```
class B { public : int b ; };
class B1 : public B {int b1 ; };
class B2 : public B {int b2 ; };
class C : public B1 , public B2
{ public : int f () ; private : int d ; };
```



多重派生类 C 的对象的存储结构示意



建立 C 类的对象时，B 的构造函数将被调用两次：
分别由 B1 调用和 B2 调用，
以初始化 C 类的对象中所包含的两个 B 类的子对象



11.4 多重继承——虚基类

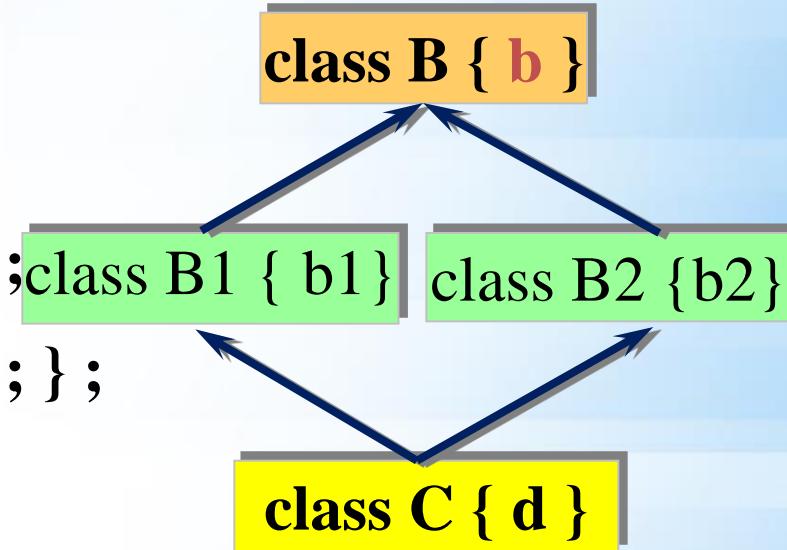
二义性的解决——虚基类

- ◆ 如果在多条继承路经上有一个公共的基类，那么在继承路经的某处汇合点，这个公共基类就会在派生类的对象中产生多个基类子对象
- ◆ 要使这个公共基类在派生类中只产生一个子对象，必须将这个基类声明为**虚基类**。
- ◆ 虚基类声明使用关键字 **virtual**



11.4 多重继承——虚基类

```
class B { public : int b ; } ;
class B1 : virtual public B {int b1 ; };
class B2 : virtual public B {int b2 ; } ;
class C : public B1 , public B2
{ private : float d ; } ;
cc . b // C cc
```



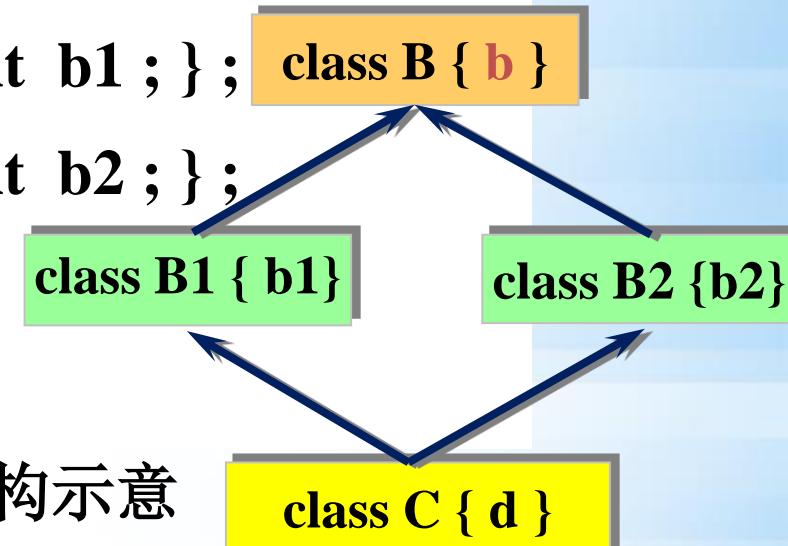
由于类 C 的对象中只有一个 B 类子对象，名字 b 被约束到该子对象上，所以，当以不同路径使用名字 b 访问 B 类的子对象时，所访问的都是

那个唯一的基类子对象。即 cc . B1 :: b 和 cc . B2 :: b 引用是同一个基类 B 的子对象

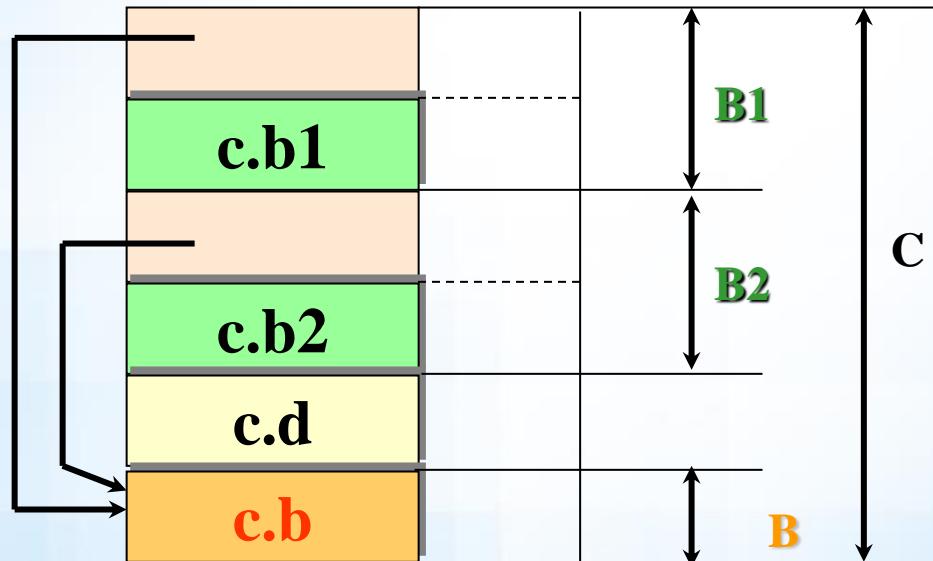
11.4 多重继承 南其米

```
class B { public : int b ; } ;

class B1 : virtual public B { private : int b1 ; } ;
class B2 : virtual public B { private : int b2 ; } ;
class C : public B1 , public B2
{ private : float d ; } ;
```

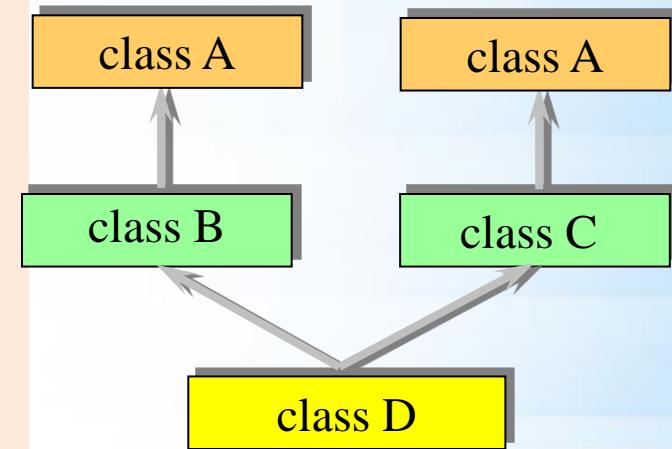


带有虚基类的派生类 C 的对象的存储结构示意



例：虚继承的测试

```
#include <iostream.h >
class A
{ public :
    A () { cout << "class A" << endl ; }
} ;
class B : public A
{ public :
    B () {cout << "class B" << endl ; }
} ;
class C : public A
{ public :
    C () {cout << "class C" << endl ; }
} ;
class D : public B , public C
{ public :
    D () {cout << "class D" << endl ; }
} ;
void main ()
{ D dd ; }
```



两次调用
A的构造函数

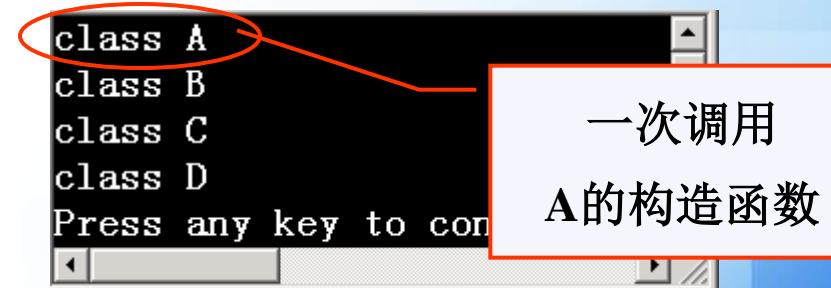
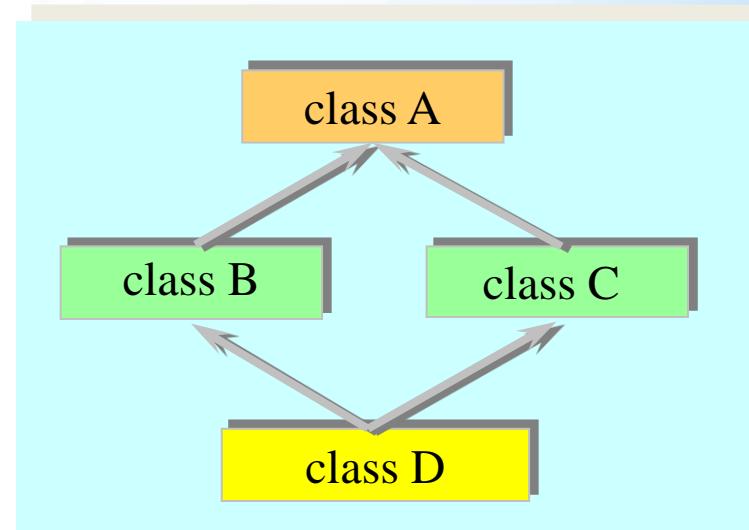
```

class A
class B
class A
class C
class D
Press any key to continue
  
```

//例: 虚继承的测试

```
#include <iostream.h>
class A
{ public :
    A () { cout << "class A" << endl ; }
} ;
class B : virtual public A
{ public :
    B () {cout << "class B" << endl ; }
} ;
class C : virtual public A
{ public :
    C () {cout << "class C" << endl ; }
} ;
class D : public B , public C
{ public :
    D () {cout << "class D" << endl ; }
} ;
void main ()
{ D dd ; }
```

类



11.4 多重继承——虚基类

虚基类的构造函数

- 由于派生类的对象中只有一个虚基类对象。为保证虚基类对象只被初始化一次，这个虚基类构造函数必须只被调用一次。
- 规定将在建立对象时所指定的类称为**最直接派生类**。
- 虚基类对象是由**最直接派生类**的构造函数通过调用虚基类的构造函数进行初始化的。
- 从虚基类直接或间接继承的派生类中的构造函数的成员初始化列表中都要列出这个虚基类构造函数的调用。但是，只有用于建立对象的那个派生类的构造函数调用虚基类的构造函数，而该派生类的基类中所列出的对这个虚基类的构造函数调用在执行中被忽略。



11.5 深度探索

- ◆ 11.5.1 组合与继承
- ◆ 11.5.2 派生类对象的内存布局
- ◆ 11.5.3 基类向派生的转换及其安全性问题

11.5.1 组合与继承

- ◆ **组合与继承：**通过已有类来构造新类的两种基本方式
- ◆ **组合：**B类中存在一个A类型的内嵌对象
 - 有一个（has-a）关系：表明每个B类型对象“有一个”A类型对象
 - A类型对象与B类型对象是部分与整体关系
 - B类型的接口不会直接作为A类型的接口

```

class Engine {//发动机类
public:
    void work();          //发动机运转
    .....
};

class Wheel {//轮子类
public:
    void roll();          //轮子转动
    .....
};

class Automobile {//汽车类
public:
    void move();          //汽车移动
private:
    Engine engine; //汽车引擎
    Wheel wheels[4];//4个车轮
    .....
};

```

- 意义
 - ✓ 一辆汽车有一个发动机
 - ✓ 一辆汽车有四个轮子
- 接口
 - ✓ 作为整体的汽车不再具备发动机的运转功能，和轮子的转动功能，但通过将这些功能的整合，具有了自己的功能——移动

11.5.1 组合与继承

◆ 公有继承的意义

公有继承：A类是B类的公有基类

- 是一个 (is-a) 关系：表明每个B类型对象“是一个”A类型对象
- A类型对象与B类型对象是一般与特殊关系
 - 回顾类的兼容性原则：在需要基类对象的任何地方，都可以使用公有派生类的对象来替代
- B类型对象包括A类型的全部接口



11.5.1 组合与继承——“is-a” 举例

```
class Truck: public Automobile{
    //卡车
public:
    void load(...);    //装货
    void dump(...);   //卸货
private:
    .....
};
```

```
class Pumper: public Automobile {
    //消防车
public:
    void water();      //喷水
private:
    .....
};
```

- 意义
 - 卡车是汽车
 - 消防车是汽车
- 接口
 - 卡车和消防车具有汽车的通用功能（移动）
 - 它们还各自具有自己的功能（卡车：装货、卸货；消防车：喷水）

11.5.2 派生类对象的内存布局

◆ 派生类对象的内存布局

- 因编译器而异
- 内存布局应使类型兼容规则便于实现
 - 一个基类指针，无论其指向基类对象，还是派生类对象，通过它来访问一个基类中定义的数据成员，都可以用相同的步骤

◆ 不同情况下的内存布局

- 单继承：基类数据在前，派生类新增数据在后
- 多继承：各基类数据按顺序在前，派生类新增数据在后
- 虚继承：需要增加指针，间接访虚基类数据

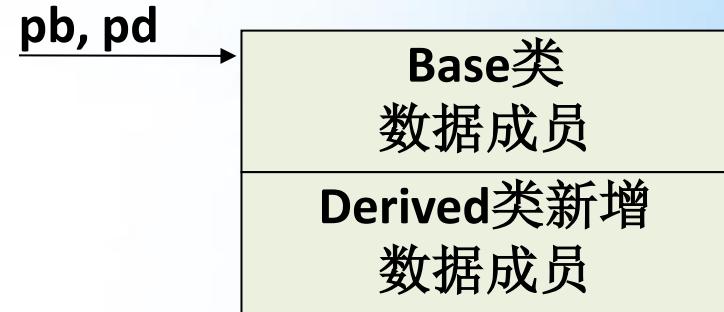


11.5. 2 派生类对象的内存布局

◆ 单继承

```
class Base { ... };
class Derived: public Base { ... };
```

```
Derived *pd = new Derived();
//Derived *pd = new Derived;
Base *pb = pd;
```



Derived对象

Derived类型指针pd转换为Base类型指针时，地址不需要改变

11.5. 2 派生类对象的内存布局

◆ 多继承

```
class Base1 { ... };
```

```
class Base2 { ... };
```

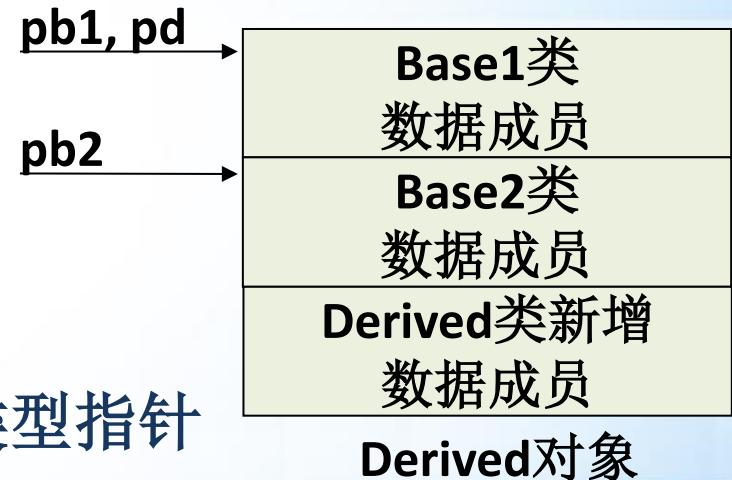
```
class Derived: public Base1, public Base2 { ... };
```

```
Derived *pd = new Derived();
```

```
Base1 *pb1 = pd;
```

```
Base2 *pb2 = pd;
```

Derived类型指针pd转换为Base2类型指针时，原地址需要增加一个偏移量



11.5. 2 派生类对象的内存布局

◆ 虚拟继承

```
class Base0 { ... };
```

```
class Base1: virtual public Base0 { ... };
```

```
class Base2: virtual public Base0 { ... };
```

```
class Derived: public Base1, public Base2 { ... };
```

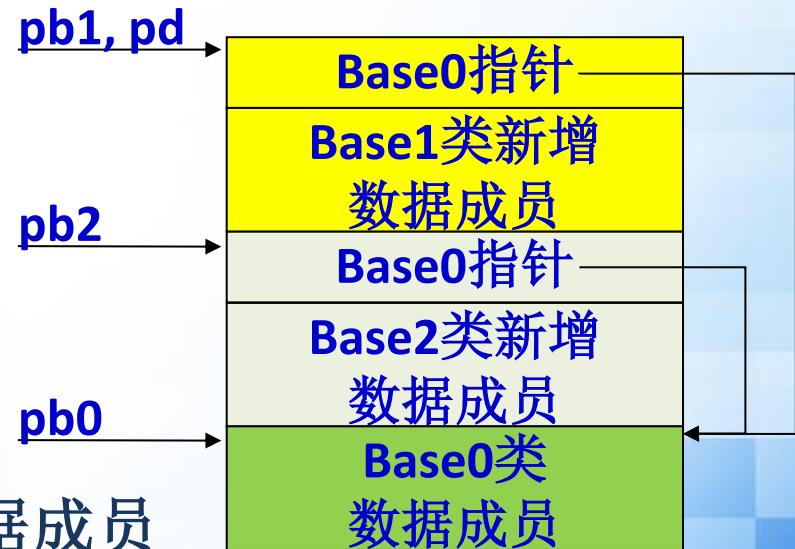
```
Derived *pd = new Derived();
```

```
Base1 *pb1 = pd;
```

```
Base2 *pb2 = pd;
```

```
Base0 *pb0 = pb1;
```

通过指针间接访问虚基类的数据成员



Derived对象



哈爾濱工業大學

Harbin Institute of Technology

11.5.3 基类向派生的转换及其安全性问题

基类向派生类的转换

- 基类指针可以转换为派生类指针
- 基类引用可以转换为派生类引用
- 需要用static_cast显式转换

例：

```
Base *pb = new Derived();
Derived *pd = static_cast<Derived *>(pd);
Derived d;
Base &rb = d;
Derived &rb = static_cast<Derived &>(rb);
```

11.5.3 类型转换时的注意事项(1)

◆ 基类对象一般无法被显式转换为派生类对象

- 对象到对象的转换，需要调用构造函数创建新的对象
- 派生类的复制构造函数无法接受基类对象作为参数

◆ 执行基类向派生类的转换时，一定要确保被转换的指针和引用所指向或引用的对象符合转换的目的类型：

- 对于`Derived *pd = static_cast<Derived *>(pb);`
- 一定要保证`pb`所指向的对象具有`Derived`类型，或者是`Derived`类型的派生类。



11.5.3 类型转换时的注意事项(2)

- ◆ 如果A类型是B类型的虚拟基类， A类型指针无法通过 **static_cast** 隐含转换为B类型的指针
 - 可以结合虚继承情况下的对象内存布局，思考为什么不允许这种转换
- ◆ **void指针参加的转换，可能导致不可预期的后果：**
 - 例：（Base2是Derived的第二个公共基类）

```
Derived *pd = new Derived();
void *pv = pd;      //将Derived指针转换为void指针
Base2 *pb = static_cast<Base2 *>(pv);
```

 - 转换后pb与pd有相同的地址，而正常的转换下应有一个偏移量
 - 结论：有void指针参与的转换，兼容性规则不适用
- ◆ 更安全更灵活的基类向派生类转换方式——**dynamic_cast**

