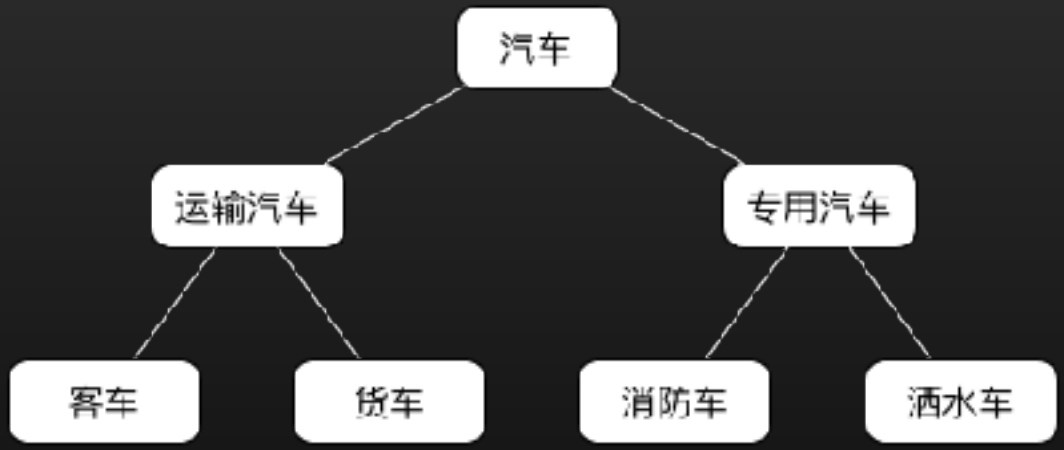
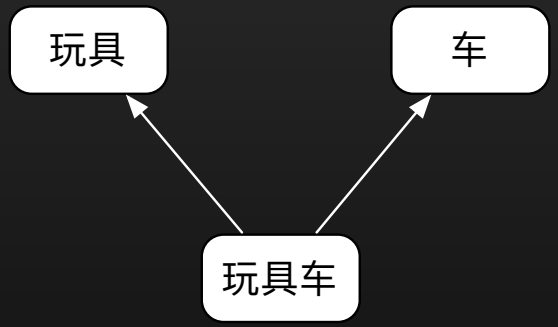
C++面向对象程序设计

第四讲：继承与派生

计算机科学学院@中南民族大学

实体的层次分类方法





类的继承与派生

代码重用
已有程序的改造

- 所谓继承就是从先辈处得到属性和行为特征；
- 从已有类产生新类的过程就是类的派生。

- 已有的（被继承的）类称为基类（或父类）；
- 产生的新类称为派生类或子类。

- 派生类同类可以作为基类派生出新的类；
- 直接参与派生出某类的基类称为直接基类，基类的基类甚至更高层的基类称为间接基类。

派生类的定义

```
class 派生类名：继承方式   基类名
{
    成员声明;
}
```

例如：

```
class Derived: public Base1, private Base2
{
public:
       Derived ();
       ~Derived ();
};
```

继承方式

- 一个派生类，可以同时有多个基类，这种情况称为多继承
- 一个派生类只有一个直接基类的情况，称为单继承。

- 派生类的成员
  - 直接从基类继承来的成员
    - 重写从基类继承来的成员
  - 新增加的数据和函数成员

派生类生成过程

- 吸收基类成员
  - 吸收基类成员之后，派生类实际上就包含了它的全部基类中除构造和析构函数之外的所有非私有成员。
- 改造基类成员
  - 如果派生类声明了一个和某基类成员同名的新成员（如果是成员函数，则参数表也要相同，参数不同的情况属于重载），派生的新成员就覆盖了外层同名成员
- 添加新的成员
  - 派生类新成员的加入是继承与派生机制的核心，是保证派生类在功能上有所发展

访问控制

- 三种继承方式
  - 公有继承
  - 私有继承
  - 保护继承
- 不同继承方式的影响主要体现在
  - 派生类成员对基类成员的访问权限
  - 通过派生类对象对基类成员的访问权限

公有继承(public)

- 基类的 公有 和 保护 成员的访问属性在派生类中保持不变，但基类的 私有 成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的 公有 和 保护 成员，但不能直接访问基类的 私有 成员。
- 通过派生类的对象只能访问基类的 公有 成员。

例1 公有继承举例

```
//Point.h
#ifndef _POINT_H
#define _POINT_H
class Point {//基类Point类的定义
public://公有函数成员
   void initPoint(float x = 0, float y = 0)
   { this->x = x; this->y = y;}
   void move(float offX, float offY)
   { x += offX; y += offY; }
   float getX() const { return x; }
   float getY() const { return y; }
private://私有数据成员
   float x, y;
};
#endif //_POINT_H
```

例1（续）

```
//Rectangle.h
#ifndef _RECTANGLE_H
#define _RECTANGLE_H
#include "Point.h"
class Rectangle: public Point {      //派生类定义部分
public:      //新增公有函数成员
      void initRectangle(float x, float y, float w, float h) {
            initPoint(x, y);        //调用基类公有成员函数
            this->w = w;
            this->h = h;
      }
      float getH() const { return h; }
      float getW() const { return w; }
private: //新增私有数据成员
      float w, h;
};
#endif //_RECTANGLE_H
```

例1（续）

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
        Rectangle rect;  //定义Rectangle类的对象
        //设置矩形的数据
        rect.initRectangle(2, 3, 20, 10);
        rect.move(3,2);  //移动矩形位置
        cout << "The data of rect(x,y,w,h): " << endl;
        //输出矩形的特征参数
        cout << rect.getX() <<", "
             << rect.getY() << ", "
             << rect.getW() << ", "
             << rect.getH() << endl;
        return 0;
}
```

私有继承(private)

- 基类的public和protected成员都以private身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员。

例2 私有继承举例

```
//Point.h
#ifndef _POINT_H
#define _POINT_H
class point {      //基类point类的定义
public:            //公有函数成员
        void initpoint(float x = 0, float y = 0)
              { this->x = x; this->y = y;}
        void move(float offx, float offy)
             { x += offx; y += offy; }
        float getx() const { return x; }
        float gety() const { return y; }
private:           //私有数据成员
        float x, y;
};
#endif //_POINT_H
```

例2 （续）

```
//Rectangle.h
#ifndef _RECTANGLE_H
#define _RECTANGLE_H
#include "Point.h"
class Rectangle: private Point {    //派生类定义部分
public: //新增公有函数成员
        void initRectangle(float x, float y, float w, float h) {
                initPoint(x, y); //调用基类公有成员函数
                this->w = w;
                this->h = h;
        }
        void move(float offX, float offY) {
                Point::move(offX, offY);
        }
        float getX() const { return Point::getX(); }
        float getY() const { return Point::getY(); }
        float getH() const { return h; }
        float getW() const { return w; }
private:        //新增私有数据成员
        float w, h;
};
#endif //_RECTANGLE_H
```

例2 (续)

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
        Rectangle rect;  //定义Rectangle类的对象
        rect.initRectangle(2, 3, 20, 10);  //设置矩形的数据
        rect.move(3,2);  //移动矩形位置
        cout << "The data of rect(x,y,w,h): " << endl;
        cout << rect.getX() <<", " //输出矩形的特征参数
                << rect.getY() << ", "
                << rect.getW() << ", "
                << rect.getH() << endl;
        return 0;
}
```

保护继承(protected)

- 基类的public和protected成员都以protected身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员

protected 成员的特点与作用

- 对建立其所在类对象的模块来说，它与 PRIVATE 成员的性质相同。
- 对于其派生类来说，它与 PUBLIC 成员的性质相同。
- 既实现了数据隐藏，又方便继承，实现代码重用。

例3： protected 成员举例

```
class A {
protected:
        int x;
};
```

```
int main()
{
        A a;
        a.x = 5;    //错误
}
```

例3 (续)

```
class A {
protected:
        int x;
};
```

```
class B: public A {
public:
        void function();
};
void B:function() {
        x = 5;   //正确
}
```

类型兼容规则

- 一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：
  - 派生类的对象可以隐含转换为基类对象。
  - 派生类的对象可以初始化基类的引用。
  - 派生类的指针可以隐含转换为基类的指针。
- 通过基类对象名、指针只能使用从基类继承的成员

例4 类型兼容规则举例

```
#INCLUDE <IOSTREAM>
USING NAMESPACE STD;
CLASS BASE1 { //基类BASE1定义
PUBLIC:
  VOID DISPLAY() CONST {
          COUT << "BASE1::DISPLAY()" << ENDL;
  }
};
CLASS BASE2: PUBLIC BASE1 { //公有派生类BASE2定义
PUBLIC:
  VOID DISPLAY() CONST {
          COUT << "BASE2::DISPLAY()" << ENDL;
  }
};
CLASS DERIVED: PUBLIC BASE2 { //公有派生类DERIVED定义
PUBLIC:
  VOID DISPLAY() CONST {
          COUT << "DERIVED::DISPLAY()" << ENDL;
  }
};
```

例4 (续)

```
VOID FUN(BASE1 *PTR) {    //参数为指向基类对象的指针
  PTR->DISPLAY();                    //"对象指针->成员名"
}
INT MAIN() {      //主函数
  BASE1 BASE1;    //声明BASE1类对象
  BASE2 BASE2;    //声明BASE2类对象
  DERIVED DERIVED;          //声明DERIVED类对象

  FUN(&BASE1);    //用BASE1对象的指针调用FUN函数
  FUN(&BASE2);    //用BASE2对象的指针调用FUN函数
  FUN(&DERIVED);      //用DERIVED对象的指针调用FUN函数

  RETURN 0;
}
```

运行结果:

```
Base1::display()
Base1::display()
Base1::display()
```

基类与派生类的对应关系

- 单继承
  - 派生类只从一个基类派生。
- 多继承
  - 派生类从多个基类派生。
- 多重派生
  - 由一个基类派生出多个不同的派生类。
- 多层派生
  - 派生类又作为基类，继续派生新的类。

多继承时派生类的声明

CLASS 派生类名：继承方式1 基类名1，
继承方式2 基类名2，...
{
　　成员声明；
}
注意：每一个“继承方式”，只用于限制对紧随其后之基类的继承。

多继承举例

```
CLASS A {
PUBLIC:
  VOID SETA(INT);
  VOID SHOWA() CONST;
PRIVATE:
  INT A;
};
CLASS B {
PUBLIC:
  VOID SETB(INT);
  VOID SHOWB() CONST;
private:
  int b;
};
class C : public A, private
    B {
public:
  void setC(int, int, int);
  void showC() const;
private const:
  int c;
};
```

多继承举例 (续)

```
VOID  A::SETA(INT X) {
  A=X;
}
VOID B::SETB(INT X) {
  B=X;
}
VOID C::SETC(INT X, INT Y, INT Z) {
  //派生类成员直接访问基类的
  //公有成员
  SETA(X);
  SETB(Y);
  C = Z;
}
//其他函数实现略
```

```
int main() {
  C obj;
  obj.setA(5);
  obj.showA();
  obj.setC(6,7,9);
  obj.showC();
// obj.setB(6);  错误
// obj.showB(); 错误
  return 0;
}
```

继承时的构造函数

- 基类的构造函数不被继承，派生类中需要声明自己的构造函数。
- 定义构造函数时，只需要对本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

单一继承时的构造函数

派生类名::派生类名(基类所需的形参，本类成员所需的形参):
基类名(参数表)

{

　　本类成员初始化赋值语句；

};

单一继承时的构造函数举例

```
#include<iostream>
using namecpace std;
class B {
public:
        B();
        B(int i);
        ~B();
        void print() const;
private:
        int b;
};
```

单一继承时的构造函数举例 (续)

```
B::B() {
        b=0;
        cout << "B's default constructor called." << endl;
}
B::B(int i) {
        b=i;
        cout << "B's constructor called." << endl;
}
B::~B() {
        cout << "B's destructor called." << endl;
}
void B::print() const {
        cout << b << endl;
}
```

单一继承时的构造函数举例 (续)

```
class C: public B {
public:
        C();
        C(int i, int j);
        ~C();
        void print() const;
private:
        int c;
};
C::C() {
        c = 0;
        cout << "C's default constructor called." << endl;
}
C::C(int i,int j): B(i) {
        c = j;
        cout << "C's constructor called." << endl;
}
```

单一继承时的构造函数举例 (续)

```
C::~C() {
        cout << "C's destructor called." << endl;
}
void C::print() const {
        B::print();
        cout << c << endl;
}

int main() {
        C obj(5, 6);
        obj.print();
        return 0;
}
```

多继承时的构造函数

派生类名::派生类名(参数表):基类名1(基类1初始化参数表), 基类名2(基类2初始化参数表), ...基类名N(基类N初始化参数表)

{

    本类成员初始化赋值语句;

};

派生类与基类的构造函数

- 当基类中声明有缺省构造函数或未声明构造函数时，派生类构造函数可以不向基类构造函数传递参数，也可以不声明，构造派生类的对象时，基类的缺省构造函数将被调用。
- 当需要执行基类中带形参的构造函数来初始化基类数据时，派生类构造函数应在初始化列表中为基类构造函数提供参数。

多继承且有内嵌对象时的构造函数

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)，新增成员对象的初始化

{

　　本类成员初始化赋值语句；

};

构造函数的执行顺序

1. 调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。
2. 对成员对象进行初始化，初始化顺序按照它们在类中声明的顺序。
3. 执行派生类的构造函数体中的内容。

例5 派生类构造函数举例

例5 (续)

运行结果:

```
constructing Base2 2
constructing Base1 1
constructing Base3 *
constructing Base1 3
constructing Base2 4
constructing Base3 *
```

拷贝构造函数

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

  例如:

  C::C(CONST C &C1): B(C1) {…}

析构函数

- 析构函数也不被继承，派生类自行声明
- 声明方法与一般（无继承关系时）类的析构函数相同。
- 不需要显式地调用基类的析构函数，系统会自动隐式调用。
- 析构函数的调用次序与构造函数相反。

例6派生类析构函数举例

例6(续)

例6 (续)

运行结果:

```
CONSTRUCTING BASE2 2
CONSTRUCTING BASE1 1
CONSTRUCTING BASE3 *
CONSTRUCTING BASE1 3
CONSTRUCTING BASE2 4
CONSTRUCTING BASE3 *
DESTRUCTING BASE3
DESTRUCTING BASE2
DESTRUCTING BASE1
DESTRUCTING BASE3
DESTRUCTING BASE1
DESTRUCTING BASE2
```

同名隐藏规则

当派生类与基类中有相同成员时：

- 若未强行指名，则通过派生类对象使用的是派生类中的同名成员。
- 如要通过派生类对象访问基类中被隐藏的同名成员，应使用基类名限定。

例7 多继承同名隐藏举例

例7 (续)

二义性问题

- 在多继承时，基类与派生类之间，或基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用虚函数（参见第8章）或同名隐藏规则来解决。
- 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。

二义性问题举例(1)

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

如果定义：C c1;
则 c1.f() 具有二义性
而 c1.g() 无二义性（同名隐藏）

二义性的解决方法

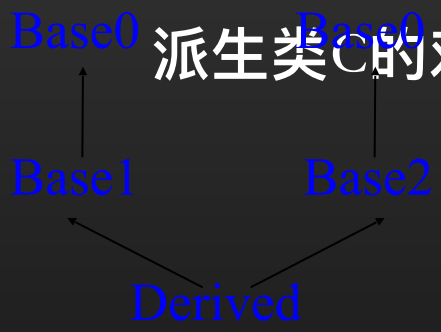
- 解决方法一：用类名来限定
  C1.A::F()　或　C1.B::F()
- 解决方法二：同名隐藏
  在C 中声明一个同名成员函数F()，F()再根据需要调用
  A::F()　或　B::F()

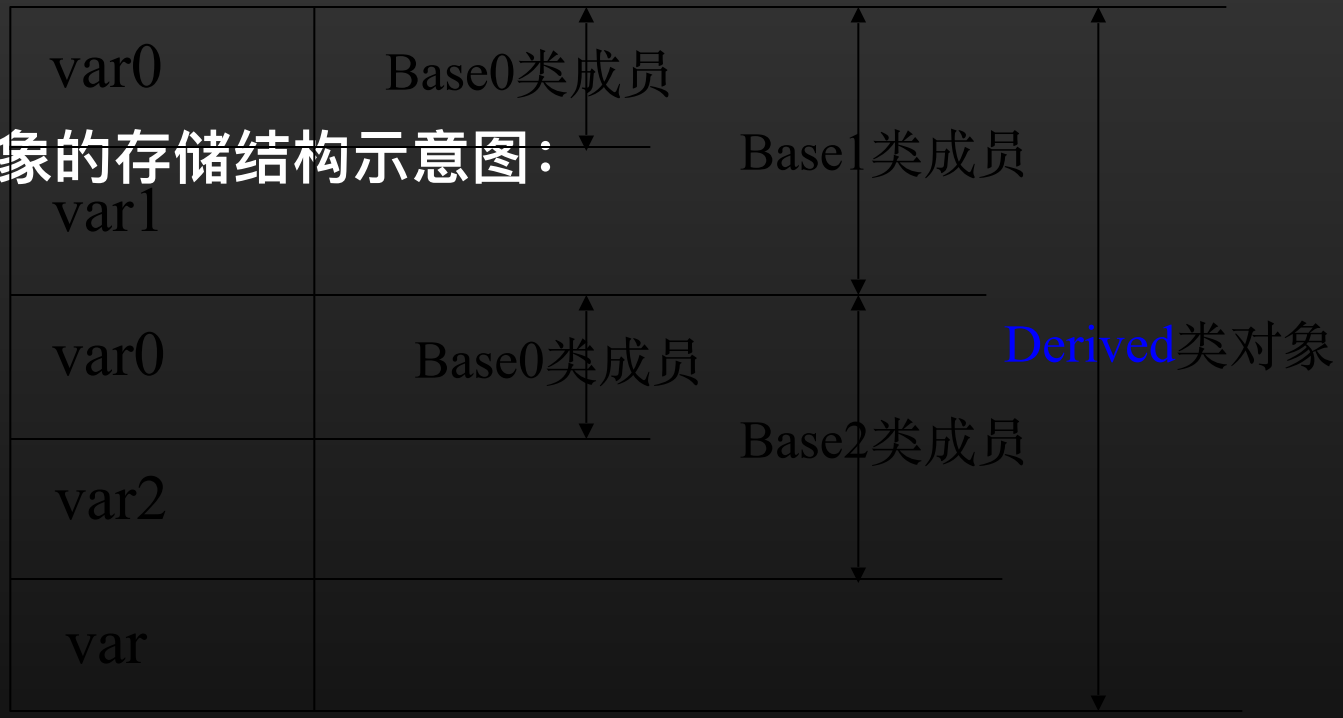
例8 多继承同名隐藏举例(2)

```
#include <iostream>
using namespace std;
class Base0 {//定义基类Base0
public:
  int var0;
  void fun0() { cout << "Member of Base0" << endl; }
};
class Base1: public Base0 {//定义派生类Base1
public://新增外部接口
  int var1;
};
class Base2: public Base0 {//定义派生类Base2
public://新增外部接口
  int var2;
};
class Derived: public Base1, public Base2 {//定义派生类Derived
public://新增外部接口
  int var;
  void fun() { cout << "Member of Derived" << endl; }
};

int main() {//程序主函数
 Derived d;//定义Derived类对象d
 d.Base1::var0 = 2;//使用直接基类
 d.Base1::fun0();
 d.Base2::var0 = 3;//使用直接基类
 d.Base2::fun0();
```

派生类C的对象的存储结构示意图：





有二义性：

```
Derived d;
d.var0
d.Base0::var0
```

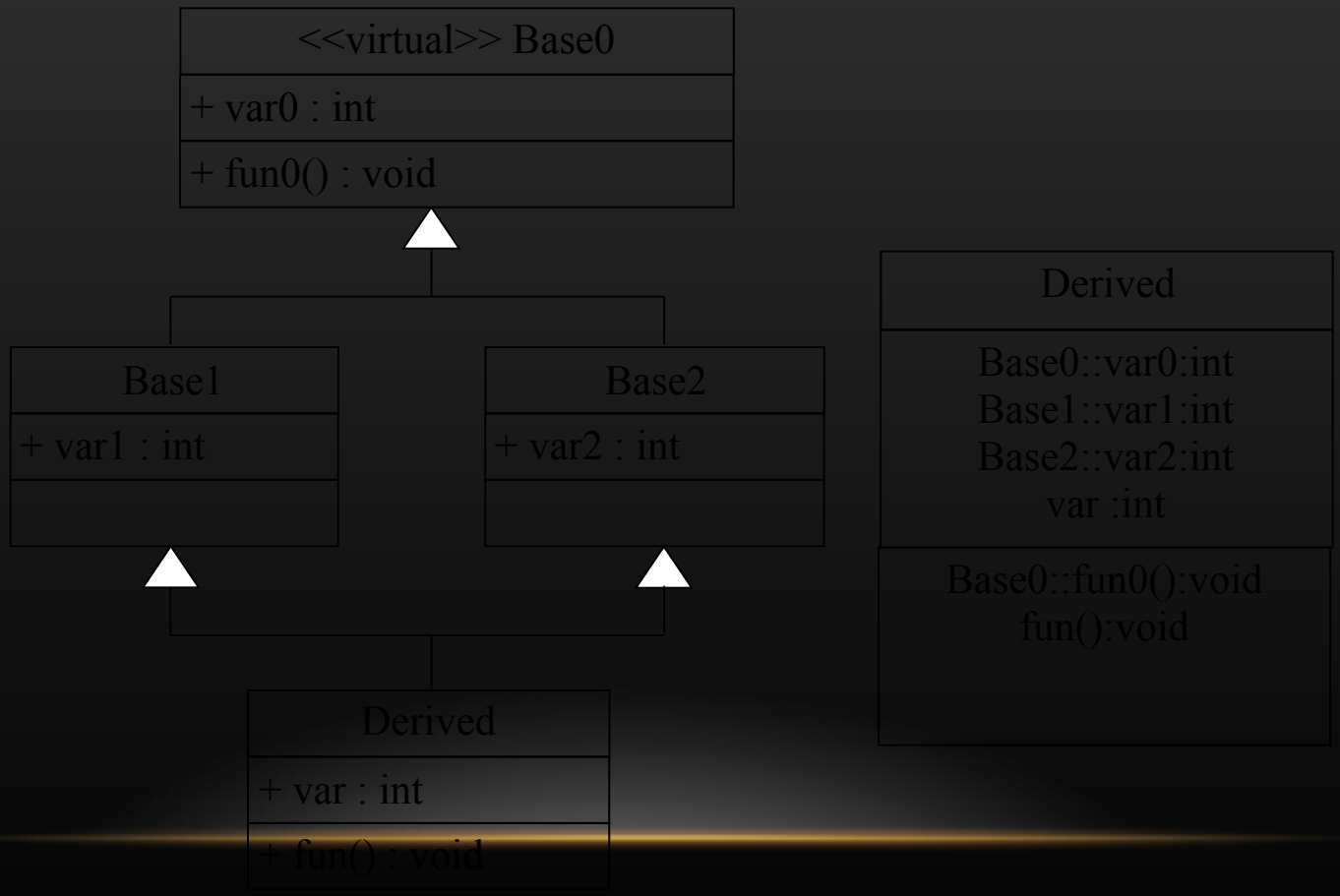
无二义性：

```
d.Base1::var0
d.Base2::var0
```

虚基类

- 虚基类的引入
  - 用于有共同基类的场合
- 声明
  - 以VIRTUAL修饰说明基类
    例：CLASS B1:VIRTUAL PUBLIC B
- 作用
  - 主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题.
  - 为最远的派生类提供唯一的基类成员，而不重复产生多次拷贝
- 注意：
  - 在第一级继承时就要将共同基类设计为虚基类。

例9虚基类举例


<<virtual>> Base0
+ var0 : int
+ fun0() : void
Base1
+ var1 : int
Base2
+ var2 : int
Derived
+ var : int
+ fun() : void
Derived
Base0::var0:int
Base1::var1:int
Base2::var2:int
var :int
Base0::fun0():void
fun():void

例9 (续)

例9 (续)

虚基类及其派生类构造函数

- 建立对象时所指定的类称为最（远）派生类。
- 虚基类的成员是由最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。
- 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。
- 在建立对象时，只有最派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略。

有虚基类时的构造函数举例

有虚基类时的构造函数举例 (续)

深度探索：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

“HAS-A”举例

- 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

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

“IS-A”举例

- 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）

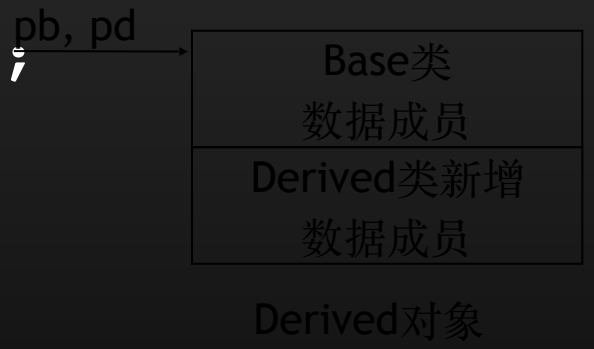
派生类对象的内存布局

- 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

单继承情形

```
CLASS BASE { … };
CLASS DERIVED: PUBLIC BASE { … };

DERIVED *PD = NEW DERIVED();
BASE *PB = PD;
```

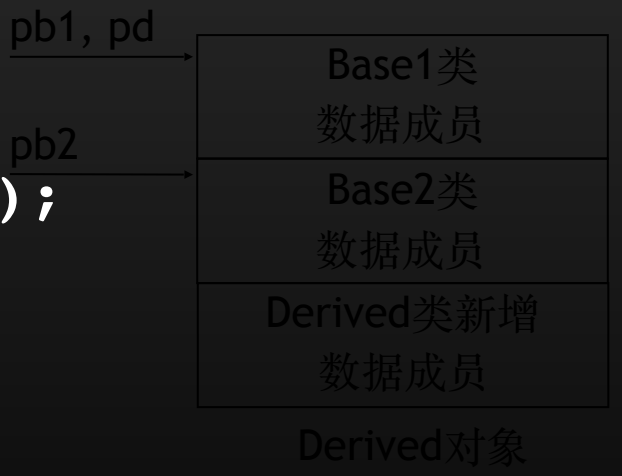


Derived类型指针pd转换为Base类型指针时，地址不需要改变

多继承情形

```
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

虚拟继承情形

```
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

小结

- 主要内容
  - 类的继承、类成员的访问控制、单继承与多继承、派生类的构造和析构函数、类成员的标识与访问
- 达到的目标
  - 理解类的继承关系，学会使用继承关系实现代码的重用。