计算机程序设计语言（VC++）

# 第 7 章

# 继承与多态性

张晓如，华伟

《C++程序设计基础教程》
人民邮电出版社，2018.05

# 本章内容

# 7.1 继承与派生

面向对象程序设计的基本特性

- 封装性：通过类定义封装数据及其操作，隐藏属性；
- 继承性：通过扩展代码模块，实现代码复用；
- 多态性：通过虚函数提供公共接口，实现接口复用。

## 7.1.1 派生类

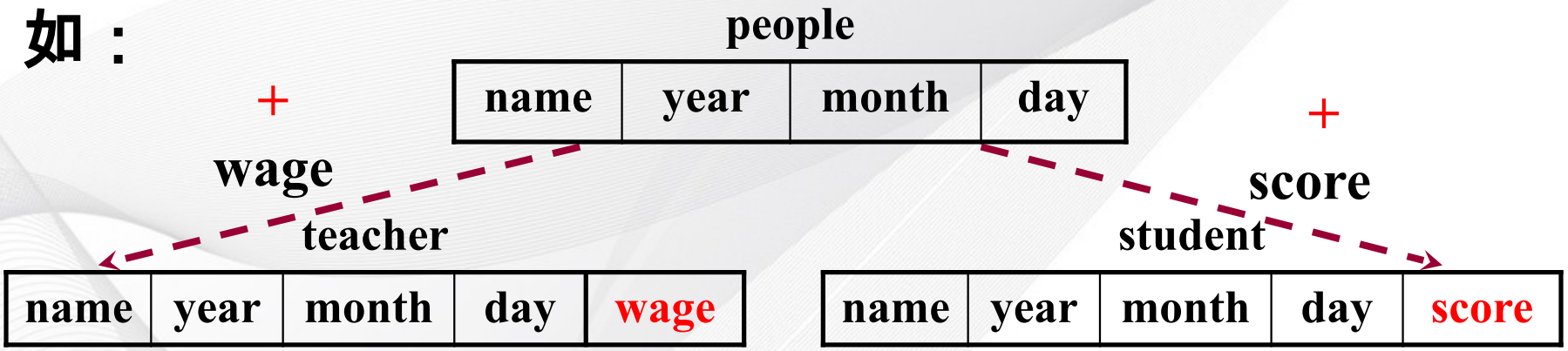
### 1. 派生的概念

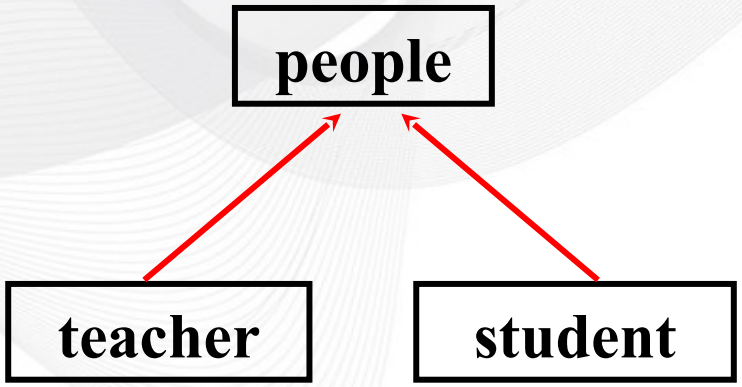
在已有类的基础上产生新类，即让一个类获得其他类属性的方法。

- 单继承；
- 多基类继承；
- 多级继承。

# 7.1 继承与派生

如：




基类（父类）

派生类（子类）

# 7.1 继承和派生

## 2．派生类的定义

定义派生 类的一般格式：

class 派生 类名：派生方式基 类名 {

新增 成员列表

};

- 派生类类名与基类之间用“ ：”分隔；
- 派生方式的关键字与成员的访问权限相同
  - public ：公有派生；
  - private ：私有派生，默认派生方式；
  - protected ：保护派生方式的派生分别。
- 新增成员的定义方法与基类中成员的定义方法相同。

# 7.1 继承和派生

**【例 7-1 】定义类 people ，包含数据成员姓名、出生日期；以类 people 为基类，定义派生类 teacher ，包含数据成员姓名、出生日期、工资和工作部门。**

程序设计

- 先定义基类 people ；
- 再定义派生类 teacher ，列出基类中没有的成员，即新增成员
  - 工资： wage ；
  - 工作部门： department 。
- 派生方式 ？

# 7.1 继承和派生

【源程序代码】

```
class people{
    char name[10];                          // 姓名
    int year,month,day;                     // 出生日期
};
class teacher: public people{               // 派生方式 public ？
    float wage;                             // 工资
public:
    char department[20];                    // 工作部门
};
```

定义 teacher 类时，说明了访问权限 department 为公有、 wage 为私有，而 name 、 year 、 month 和 day 的权限是什么？

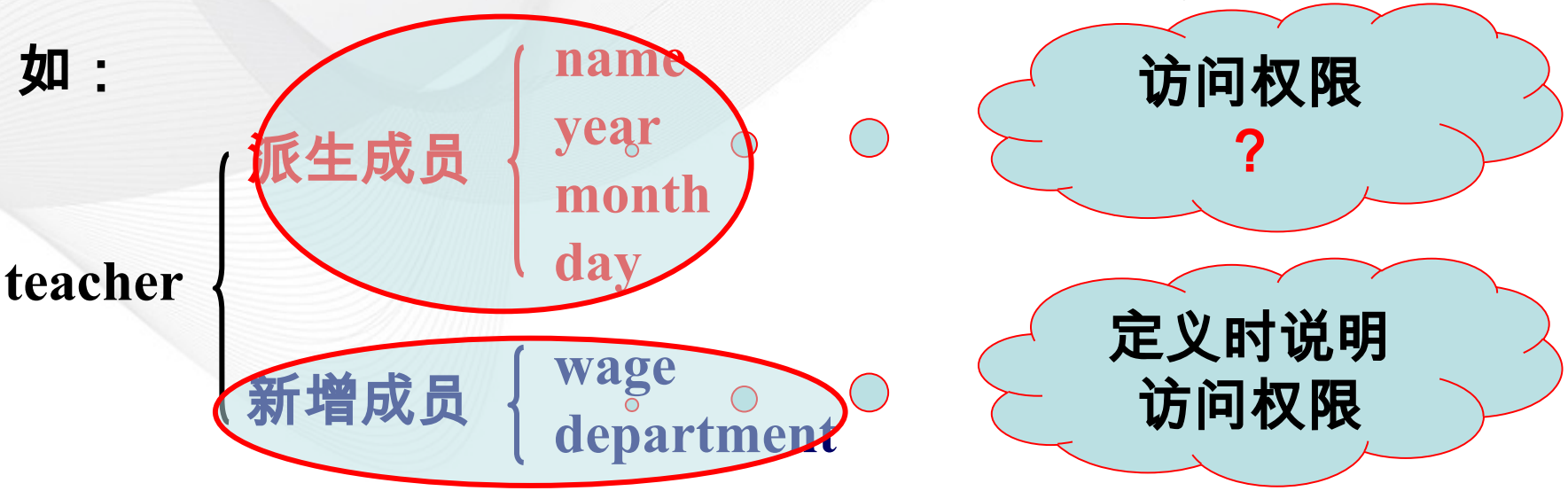
# 7.1 继承和派生

## 7.1.2 派生成员及其访问权限

### 1. 派生类中的成员

- 新增成员；
- 派生成员：从基类继承来的成员
  - 基类中的所有数据成员（**虚基类除外**）；
  - 除**构造函数**和**析构函数**以外的其他成员函数。

如：



# 7.1 继承和派生

## 7.1.2 派生成员及其访问权限

### 2. 派生成员的访问权限

派生成员的访问权限由其在基类中的**原有属性**和**派生方式**两个因素共同决定。

- **公有**派生：派生成员的访问权限维持其在基类中的原有属性**不变**；
- **私有**派生：基类中的所有成员，派生后均变为**私有**成员；
- **保护**派生
  - 基类中原有的**公有**和**保护**成员，派生后变为**保护**成员；
  - 原有的**私有**成员派生后仍为**私有**成员。

# 7.1 继承和派生

公有派生时派生成员的访问权限

| 基类原有成员 | 派生成员权限 | 内部访问方式 | 外部访问方式 |
|---|---|---|---|
| 公有成员 | 公有成员 | 直接访问 | 直接访问 |
| 保护成员 | 保护成员 | 直接访问 | 间接访问 |
| 私有成员 | 私有成员 | 间接访问 | 间接访问 |

私有派生时派生成员的访问权限

| 基类原有成员 | 派生成员权限 | 内部访问方式 | 外部访问方式 |
|---|---|---|---|
| 公有成员 | 私有成员 | 直接访问 | 间接访问 |
| 保护成员 | 私有成员 | 直接访问 | 间接访问 |
| 私有成员 | 私有成员 | 间接访问 | 间接访问 |

# 7.1 继承和派生

保护派生时派生成员的访问权限

| 基类原有成员 | 派生成员权限 | 内部访问方式 | 外部访问方式 |
|---|---|---|---|
| 公有成员 | 保护成员 | 直接访问 | 间接访问 |
| 保护成员 | 保护成员 | 直接访问 | 间接访问 |
| 私有成员 | 私有成员 | 间接访问 | 间接访问 |

- 访问方式
  - 直接访问：直接使用成员；
  - 间接访问：通过公有成员函数间接使用成员。

# 7.1 继承和派生

保护派生时派生成员的访问权限

| 基类原有成员 | 派生成员权限 | 内部访问方式 | 外部访问方式 |
|---|---|---|---|
| 公有成员 | 保护成员 | 直接访问 | 间接访问 |
| 保护成员 | 保护成员 | 直接访问 | 间接访问 |
| 私有成员 | 私有成员 | 间接访问 | 间接访问 |

- 派生类**内部访问**派生成员：由派生成员在基类中的**原有属性**决定，直接访问非私有成员，间接访问私有成员；
- 派生类**外部访问**派生成员：由**派生后的属性**决定，直接访问派生后仍为公有的成员，间接访问派生后为非公有的成员。

**【例 7-2 】公有派生时派生成员的访问示例。**

# 7.1 继承和派生

【源程序代码】

```
class Base{
    int y;
protected:
    int z;
public:
    int x;
    Base() { x=1; y=2; z=3; }
    int gety() { return y; }
    int getz() { return z; }
};
```

```
class Derived:public Base{          // 派生类内部
    public:
    void print(){
        cout<<x<<'\t'<<gety()<<'\t'<<z<<'\n';
    }
    };
```

间接访问原私有成员

直接访问原公有和保护成员

```
int main(){                         // 派生类外部
    Derived test;
    test.print();
    cout<<test.x<<'\t'<<test.gety()
        <<'\t'<<test.getz()<<'\n';
    return 0;
    }
```

直接访问派生后公有成员

间接访问派生后非公有成员

程序运行结果

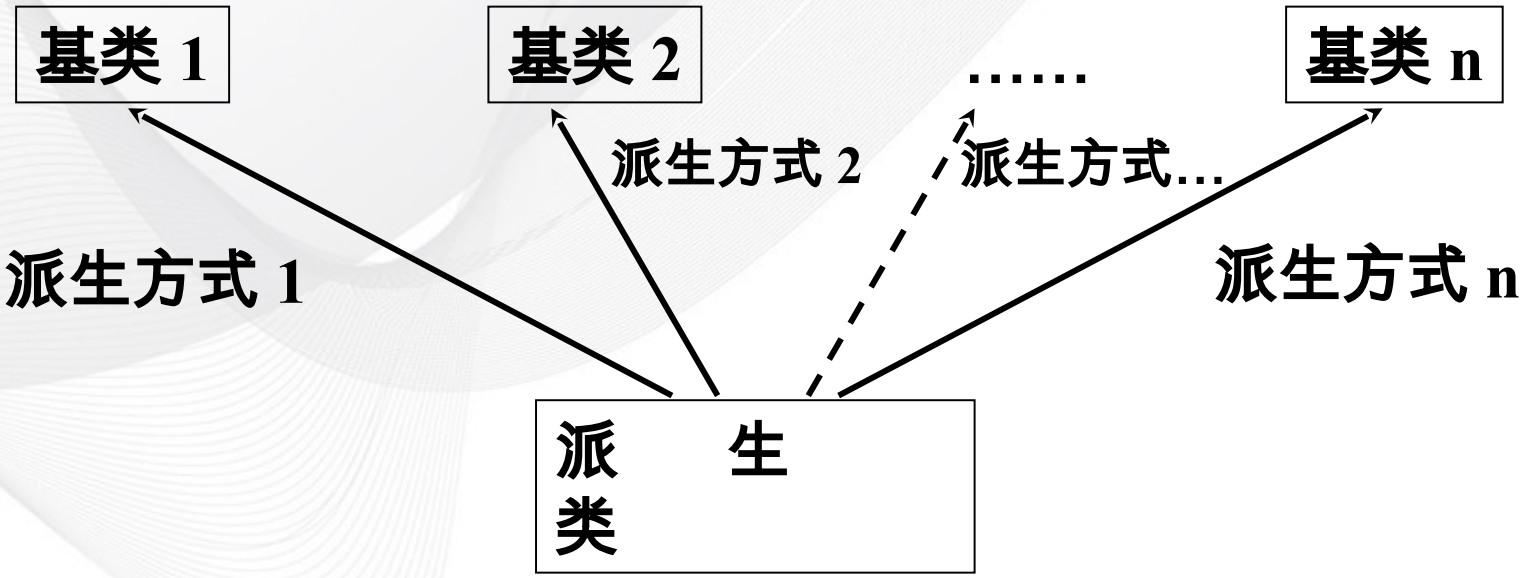
若把派生方式改为保护派生或私有派生呢？

# 7.1 继承和派生

## 7.1.3 多继承

### 1. 多基类继承

- 派生类具有两个或两个以上基类的继承方式；



# 7.1 继承和派生

## 7.1.3 多继承

- 多基类继承是单继承的**简单扩展**
  - 派生类与每个基类之间仍是单继承的关系；
  - 派生类包含各个基类的成员，以及新增成员；
  - 派生成员访问权限由各自的原有属性和派生方式决定。

- 定义格式

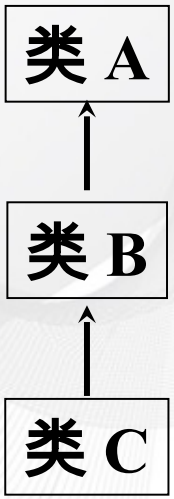
```
class 派生类名：派生方式 1 基类名 1, 派生方式 2 基类名 2, …, 派生方式 n 基类名 n{
    新增成员列表
};
```

# 7.1 继承和派生

## 7.1.3 多继承

### 2. 多级继承

- 以一个派生类为基类，产生另一个新的派生类的继承方式。




- 类 B 既是类 A 的派生类，又是类 C 的基类。
- 类 C 的基类 B 是类 A 的派生类，所以类 C 是一个多级继承。

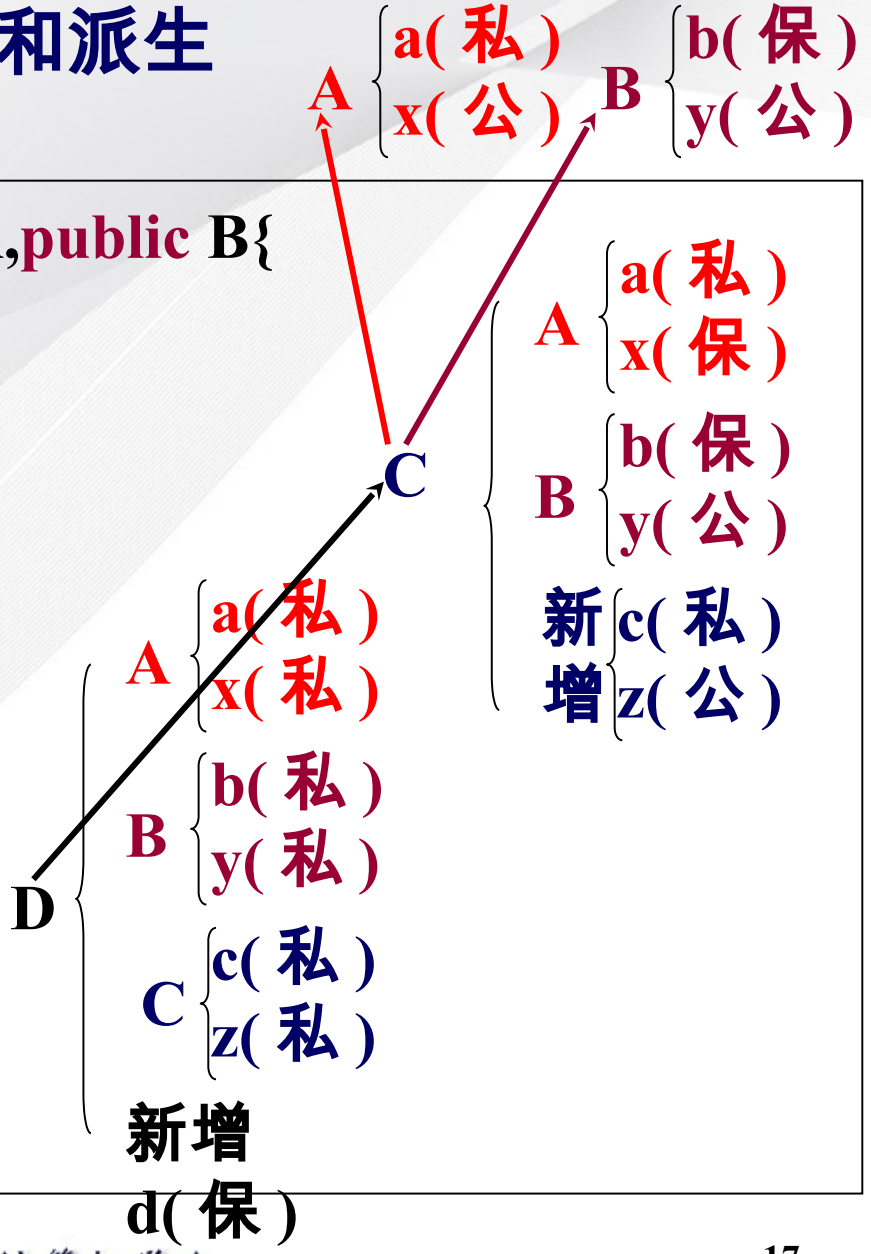
【例 7-3】写出下列类定义中类 C 和类 D 的所有数据成员，以及各数据成员的访问权限。

# 7.1 继承和派生

【源程序代码】

```
class A{
    int a;
public:
    int x;
};
class B{
protected:
    int b;
public:
    int y;
};
```

```
class C:protected A,public B{
    int c;
public:
    int z;
};
class D:private C{
protected:
    int d;
};
```



# 7.1 继承和派生

## 7.1.4 赋值兼容性

- 通常情况下，只有同类型的对象才能赋值；
- 公有派生时，可将派生类的数据赋值给基类数据——赋值兼容性。
  - 将派生类对象赋值给基类的对象；
  - 用派生类对象初始化基类对象的引用；
  - 将派生类对象地址赋值给基类指针，即基类指针指向派生类对象。

【例 7-4】用赋值兼容性实现派生类向基类赋值的示例。

# 7.1 继承和派生

【源程序代码】

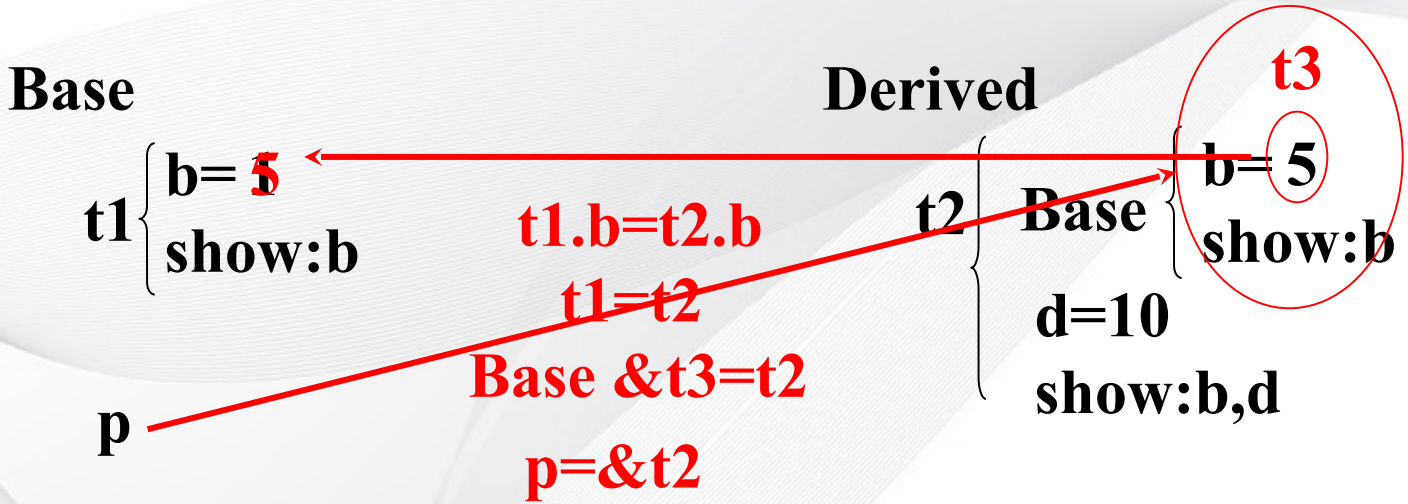
```
class Base{
public:
    int b;
    void show()
    { cout<<b<<'\n'; }
};
class Derived:public Base{
public:
    int d;
    void show()
    { cout<<b<<'\t'<<d<<'\n'; }
};
```

```
int main( ){
    Base t1={1},*p;
    Derived t2;
    t2.b=5,t2.d=10;
    t1=t2;     // 对象之间的赋值
    t1.show();
    Base &t3=t2; // 初始化引用
    t3.show();
    p=&t2;      // 指针指向对象
    p->show();
    return 0;
}
```

程序运行结果

```
5
5
5
```

# 7.1 继承和派生




**赋值兼容性注意：**

- 单向赋值，即只能从派生类向基类赋值，反之错误；
- 只有公有派生时才成立，私有和保护派生不能赋值；
- 基类指针指向派生类对象时，通常只能访问派生成员，而不能访问新增成员，除非该新增成员是虚函数。

# 7.2 派生类的构造函数与析构函数

- 在构造函数头部通过调用基类构造函数初始化派生成员；
- 在函数体中初始化新增成员。

## 7.2.1 单继承时派生类的构造函数

派生类名(形参列表):基类名(实参列表) {
    新增成员初始化列表
}

- 派生类名即派生类构造函数的函数名，其后的参数是形参，包含参数类型和名称；
- “基类名(实参列表)”是基类构造函数的调用形式，其参数是实参，参数只有名称没有类型；
- 新增成员初始化方法与基本类成员初始化方法相同。

# 7.2 派生类的构造函数与析构函数

- 派生类的构造函数在类中说明、类外定义
  - 类中说明

  派生类名 ( 形参列表 ) ;
  - 类外定义

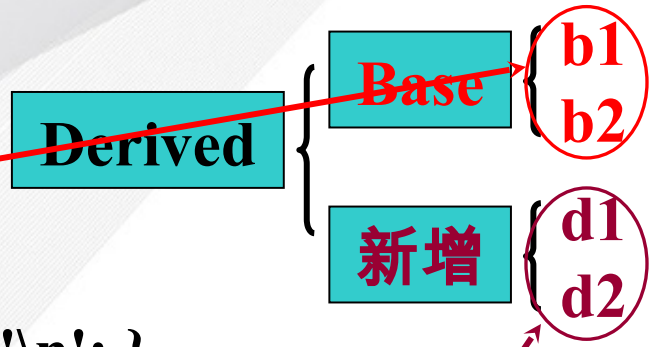
  派生类名 :: 派生类名 ( 形参列表 ): 基类名 ( 实参列表 ){
  新增成员初始化
  }
  - 说明时，参数可以只有类型没有名称；
  - 基类构造函数的调用只能在定义时列出。

【例 7-5】定义派生类，初始化数据成员。

# 7.2 派生类的构造函数与析构函数

【源程序代码】

```
class Base{ int b1,b2;
public:
    Base(int x,int y) { b1=x; b2=y; }
    void show()
    { cout<<"b1="<<b1<<",b2="<<b2<<'\n'; }
};
class Derived:public Base{ int d1,d2;
public:
    Derived (int a,int b,int c,int d):Base(a,b) { d1=c; d2=d; }
    void print(){ cout<< " 派生成员 :"; show();
        cout<< " 新增成员 :";
        cout<<"d1="<<d1<<",d2="<<d2<<'\n';
    }
        cout<<"b1="<<b1<<",b2="<<b2<<'\n';?
};
```



# 7.2 派生类的构造函数与析构函数

【源程序代码】

```
int main(){
    Derived test(1,2,3,4);
    test.print();
    return 0;
}
```

程序运行结果

```
派生成员：
  b1=1 ， b2=2
新增成员：
    d1=3 ， d2=4
```

- 派生类的构造函数通常必须包括基类构造函数的调用。
- 当基类有默认的构造函数时，派生类构造函数的头部可省略基类构造函数的调用。
  - 此时，并不是不调用基类的构造函数，而是调用基类默认的构造函数。
  - 若基类没有默认的构造函数，则编译报错。

# 7.2 派生类的构造函数与析构函数

【源程序代码】

```
class A{
public:
    A(int x=0) {
        cout<<x<<'\n';
    }
};

class B{
public:
    B(int x) {
        cout<<x<<'\n';
    }
};
```

```
class C:public A{
    public:
    C(int x) :A()
    {   cout<<x<<'\n';
    }
};

class D:public B{
public:
    D(int x):B()     // 语法错误
    {   cout<<x<<'\n';
    }
};
```

# 7.2 派生类的构造函数与析构函数

## 7.2.2 多继承时的派生类的构造函数

### 1. 多基类派生类构造函数

派生类构造函数头部逐一列出各基类构造函数调用。

派生类名 ( 形参列表 ): 基类名 1( 实参列表 1), 基类名 2( 实参列表 2), ..., 基类名 n( 实参列表 n){

    新增成员初始化

}

- 类中说明

派生类名 ( 形参列表 ) ;

- 类外定义

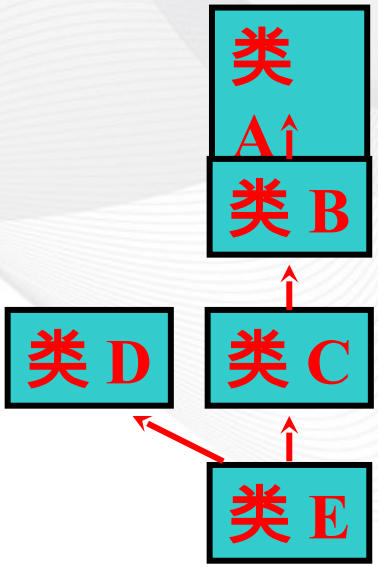
派生类名 :: 派生类名 ( 形参列表 ): 基类名 1( 实参列表 1), 基类名 2( 实参列表 2), ..., 基类名 n( 实参列表 n){ 新增成员初始化 }

# 7.2 派生类的构造函数与析构函数

## 7.2.2 多继承时的派生类的构造函数

### 2. 多级派生类构造函数

- 若每级的派生类都只有一个基类，则各级派生类构造函数的定义都与单继承时构造函数的定义方法相同；
- 若其中某级有多个基类，则该级派生类构造函数的定义采用多基类派生类构造函数定义的方法。




```
B( 形参 ):A( 实参 ){ 函数体 }

C( 形参 ):B( 实参 ){ 函数体 }

E( 形参 ):D ( 实参 ),C( 实参 ){ 函数体 }
```

# 7.2 派生类的构造函数与析构函数

## 7.2.3 派生类对象

- 生成派生类对象时必须调用构造函数初始化数据成员
  - 先调用基类构造函数初始化派生数据成员；
  - 再执行派生类构造函数函数体，初始化新增数据成员。
- 若派生类是多级派生类，则要向上逐级调用基类的构造函数；
- 若派生类是多基类派生类，则要按照继承顺序逐一调用各基类的构造函数。

**【例 7-6】分析下列多基类继承和多级继承时，派生类对象的产生过程，写出程序运行结果。**

# 7.2 派生类的构造函数与析构函数

【源程序代码】

```
class A{
public:A(){ cout<<" 调用类 A 构造函数 \n";
    }
};
class B{
public:B(){ cout<<" 调用类 B 构造函数 \n";
    }
};
class C:public B,public A{    // 多基类继承
public:C(){ cout<<" 调用类 C 构造函数 \
    n"; }
};
class D:public C{                    // 多级继承
public:D(){ cout<<" 调用类 D 构造函数 \n";
    }
```

public A,public B

结果？

程序运行结果

```
调用类 B 构造函数
调用类 A 构造函数
调用类 C 构造函数
调用类 B 构造函数
调用类 A 构造函
```

# 7.2 派生类的构造函数与析构函数

## 7.2.4 派生类析构函数

- 释放派生类的对象，包括新增成员和派生成员的空间；
  - 先执行派生类析构函数的函数体，释放新增成员；
  - 再调用其基类的析构函数，释放派生成员。
- 释放派生类对象（调用析构函数）与建立派生类对象（调用构造函数）的顺序相反。

如例 7-6 中：

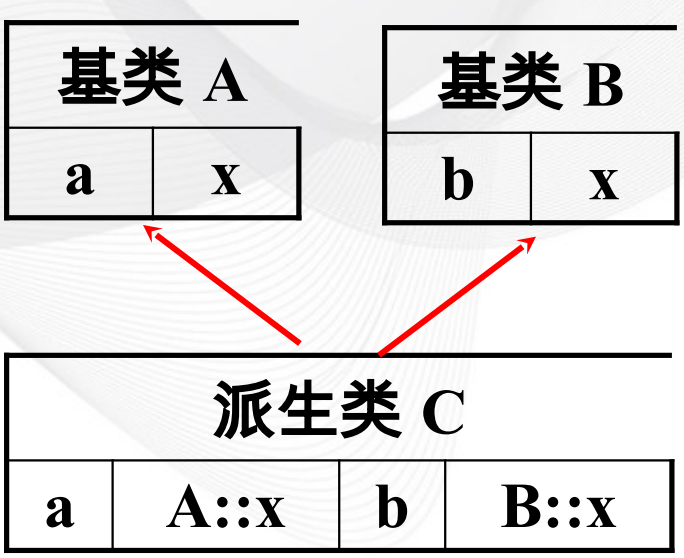
- 先建立对象 t1，后建立对象 t2；则先释放 t2，后释放 t1。
- 建立 t2 时，构造函数调用顺序：类 B→ 类 A→ 类 C→ 类 D；释放 t2 时，析构函数调用顺序：类 D→ 类 C→ 类 A→ 类 B。
- 建立 t1 时，构造函数调用顺序：类 B→ 类 A→ 类 C；释放 t1 时，析构函数调用顺序：类 C→ 类 A→ 类 B。

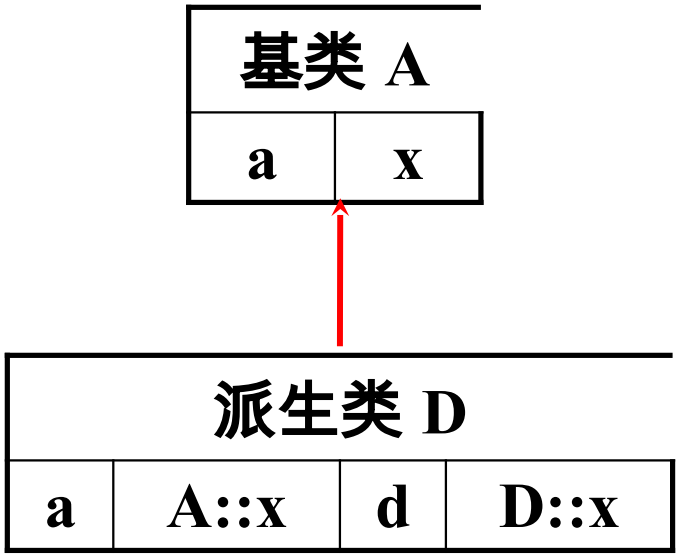
# 7.3 冲突及解决方法

## 7.3.1 冲突

### 1. 冲突的概念

- 派生类中，同时存在来自不同类的名称相同的成员
  - 来自不同基类的名称相同的派生成员；
  - 从基类继承的派生成员与派生类中的新增成员。



# 7.3 冲突及解决方法

2. 冲突**解决方法**：用“类名 ::” 来区分不同作用域 ( 类 ) 的同名成员。

## 7.3.2 支配规则

- 若**新增成员**与**派生成员**同名，在没有使用类名和作用域运算符进行限定时，**默认**使用的是**新增成员**。
  - 没有出现同名冲突时，既可以用类名和作用域运算符使用成员，也可以直接使用成员，习惯于直接使用成员；
  - 派生成员之间冲突时，必须用“类名 ::” 使用成员；
  - 派生成员与新增冲突时，用“类名 ::” 使用派生成员；可以用“类名 ::” 使用新增成员，也可以直接使用新增成员。

**【例 7-7 】根据冲突情况与支配规则，分析下列程序的运行结果。**

# 7.3 冲突及解决方法

【源程序代码】

```
class A{ protected:int a,x;
public:A(){ a=1; x=2; }
};
class B{ protected:int b,x;
public:B(){ b=3; x=4; }
};
class C:public A,public B{
    int c,x;
public:C(){ c=5; x=6; }
    void show() {
        cout<<A::a<<"\t"<<b<<"\t"<<c<<'\n';
        cout<<A::x<<"\t"<<B::x<<"\t"<<x<<'\n';
    }
};
```

test

A::a=1
A::x=2
B::b=3
B::x=4
C::c=5
C::x=6

```
int main(){
    C test;
    test.show();
    return 0;
    }
```

程序运行结果

| | | |
|---|---|---|
| 1 | 3 | 5 |
| 2 | 4 | 6 |

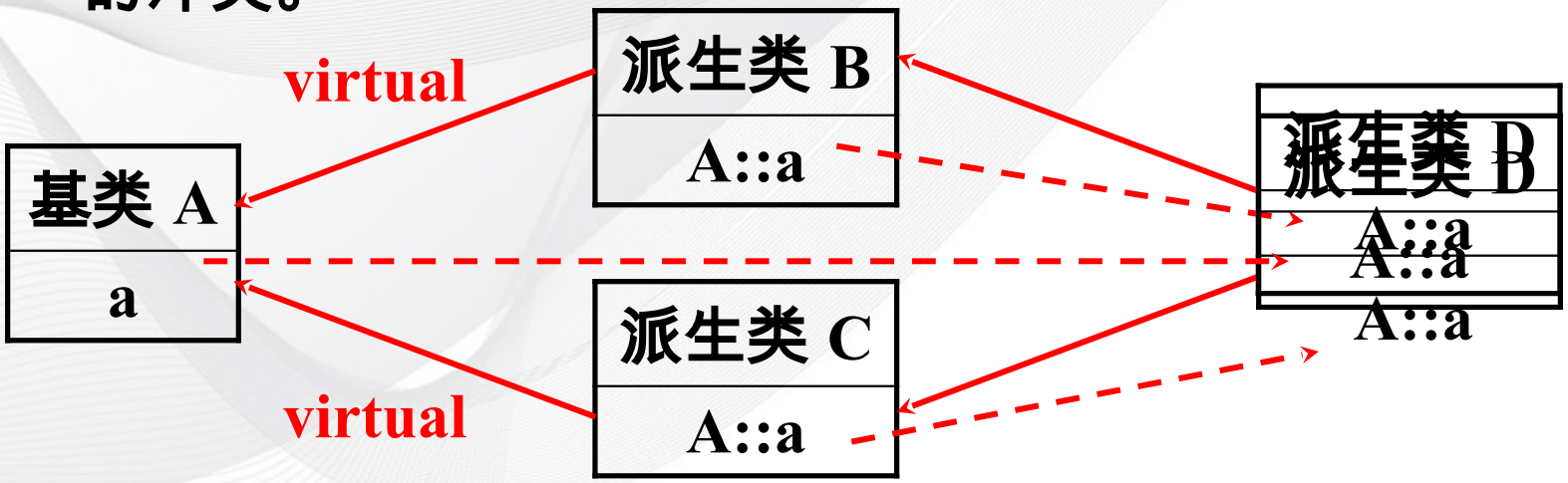
a ?

C::x ?

# 7.3 冲突及解决方法

## 7.3.3 虚基类

- 同一个基类经过多级继承后会出现用“类名：：”无法解决的冲突。




### 1．虚基类概念与定义

- 若将共同基类设置为**虚基类**，则从不同的路径继承过来的虚基类的**派生成员**成员在派生类中只出现**一次**。

# 7.3 冲突及解决方法

- 说明虚基类的方法是，定义派生类时，在基类名称的前面加上关键字 virtual 。

```
class 派生类名： virtual 派生方式 基类名 {
    新增成员列表
}
```

或

```
class 派生类名：派生方式 virtual 基类名 {
    新增成员列表
}
```

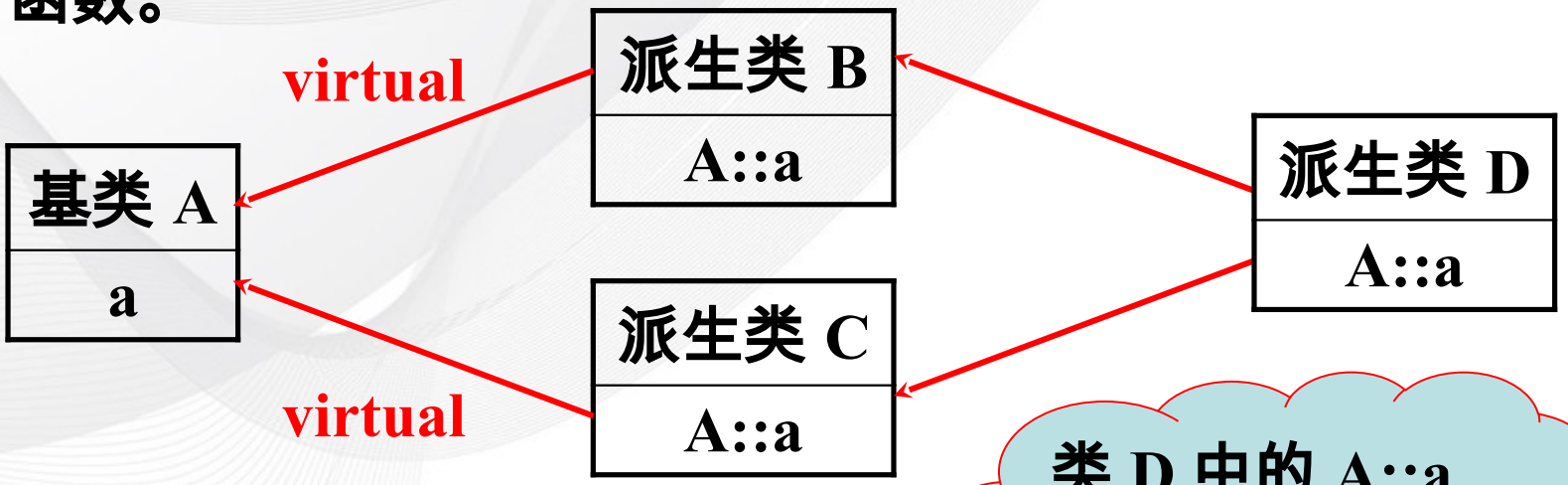
  - 关键字 virtual 可以放在派生方式的前面，也可以放在派生方式的后面。

# 7.3 冲突及解决方法

## 2. 虚拟继承的构造函数

- 从虚基类直接或间接继承的派生类构造函数头部，都必须列出虚基类构造函数调用，除非虚基类有默认的构造函数。

如




B( 形参 ):A( 实参 ){ 函数体 }

C( 形参 ):A( 实参 ){ 函数体 }

D( 形参 ): B( 实参 ), C( 实参 ), A( 实参 ){ 函数体 }

# 7.3 冲突及解决方法

## 【例 7-8】分析虚基类的定义及其派生类对象的产生，写出程序的运行结果。

【源程序代码】

```
class A{ protected: int a;
public: A(int x){ a=x; cout<<" 调用类 A 构造函数 \n";}
};
class B:public virtual A{
protected: int b;
public:
    B(int y,int z):A(z)
    { b=y; cout<<" 调用类 B 构造函数 \n"; }
    void print() { cout<<a<<'\t'<<b<<endl; }
};
```

B t1(1,2)

A::a 2
B::b 1

类 B 包含 A::a ，但 B 中的 A::a 不会出现在类 B 的派生类中！

# 7.3 冲突及解决方法

【源程序代码】

```
class C:virtual public A{ protected: int c;
public: C(int x,int y):A(y){ c=x; cout<<" 调用类 C 构造函数 \
    n"; }
};
class D:public C,public B{
    int d;
public:
    D(int m,int n,int k):B(m+10,n+10),C(m+20,n+20),A(m+n+k)
    {   d=m; cout<<" 调用类 D 构造函数 \n";  }
    void show() {
        cout<<a<<"\t"<<b<<"\t"<<c<<"\t"<<d<<"\n";
    }
};
```

不调用

基类继承顺序

基类列表（无顺序）

- 基类继承顺序决定基类构造函数的调用顺序！
- 基类列表只决定派生成员的值，与调用顺序无关！

A::a 6
B::b 11
C::c 21
D::d 1

# 7.3 冲突及解决方法

【源程序代码】

```
int main(){
    B t1(1,2);
    t1.print();
    D t2(1,2,3);
    t2.show();
    return 0;
}
```

程序运行结果

```
调用类 A 构造函数
调用类 B 构造函数
2    1
调用类 A 构造函数
调用类 C 构造函数
调用类 B 构造函数
调用类 D 构造函数
6    11    21    1
```

- 构造函数调用顺序
  - 首先调用虚基类的构造函数；
  - 然后按继承顺序调用基类的构造函数；
  - 最后执行自身函数体。

# 7.4 虚函数与多态性

## 7.4.1 多态性的基本概念

- 多态是指一个类实例的**相同函数**在不同情形下有不同的**表现形式**，**不同对象**接收到相同指令时，可以产生**不同的行为**。
- **调用函数**就是执行与函数名相应的某段存储空间的代码，函数名与存储空间首地址的匹配过程称为**地址绑定**。
  - **编译**（**静态**）多态性：编译时进行地址绑定；
  - **运行**（**动态**）多态性：运行时根据具体对象绑定地址。

### 1. 编译多态性

编译时，根据函数参数确定调用地址，如函数重载等。

**【例 7-9】静态多态性示例。**

# 7.4 虚函数与多态性

【源程序代码】

```
class A{
public: void f(){ cout<<"classA::f()\n"; }
};
class B:public A{
public: void f(){ cout<<"classB::f()\n"; }
};
int main( ){
    A t1,*p; B t2;
    t1.f();
    t2.f();
    p=&t1; p->f();
    p=&t2; p->f();
    return 0;
}
```

t1 ： A::f( )

t2 { A::f( ) B::f( )

程序运行结果

```
classA : :
  f()
classB : :
  f()
classA : :
  f()
classA : :
  f()
```

- 根据支配规则，默认调用的是新增成员； t2.A::f( ) ;?
- 基类指针指向派生类对象时，通常调用的是派生成员，虚函数除外！

# 7.4 虚函数与多态性

## 2. 运行多态性

- 当基类指针指向派生类对象的普通函数时，在编译时被绑定到从基类继承来的派生成员上；
- 要想绑定到派生类的新增成员上实现运行的多态性，必须把函数定义为虚函数；
- 虚函数允许在派生类中被重写，即重新定义函数体，也称为函数覆盖；
- 虚函数在编译时不绑定调用地址，而是在程序运行时，根据具体的对象绑定所调用函数的入口地址。

# 7.4 虚函数与多态性

## 7.4.2 虚函数实现动态多态性

### 1. 虚函数的定义

- 虚函数是在类中被声明为 virtual 的非静态成员函数。

```
virtual 函数类型 函数名 ( 形参列表 ){
    函数体
}
```

- 虚函数也可以在类中说明，在类外定义
  - 类中说明

```
virtual 函数类型 函数名称 ( 形参列表 ) ;
```

  - 类外定义

```
函数类型类名 :: 函数名 ( 形参列表 ){
    函数体
}
```

# 7.4 虚函数与多态性

- 虚函数在类中定义或说明时，必须在函数类型或函数名的前面加关键字 virtual ；
- 类外定义虚函数时不能用 virtual 说明；
- 虚函数具有遗传性，即派生类中与基类虚函数具有相同的函数类型、名称和参数的新增函数，即使不用 virtual 说明，仍然是虚函数；
- 虚函数具有不确定性
  - 不能将构造函数定义为虚函数；
  - 可以将析构函数定义为虚函数。
- 虚函数是实现动态多态性的基础和必要条件，但仅有虚函数还不能实现动态多态性。

# 7.4 虚函数与多态性

## 2. 动态多态性的实现

- 实现动态多态性必须同时满足下列条件
  - 要有具有**继承关系**的类，并将要实现动态多态性的函数定义为**虚函数**；
  - 在派生类中必须**重写虚函数**，即重新定义虚函数的函数体，且与基类对应的虚函数**同类型、名称和参数**；
  - 必须通过**基类的指针**或**基类对象的引用**调用虚函数。

**基类指针变量名->虚函数名(实参表)**

或

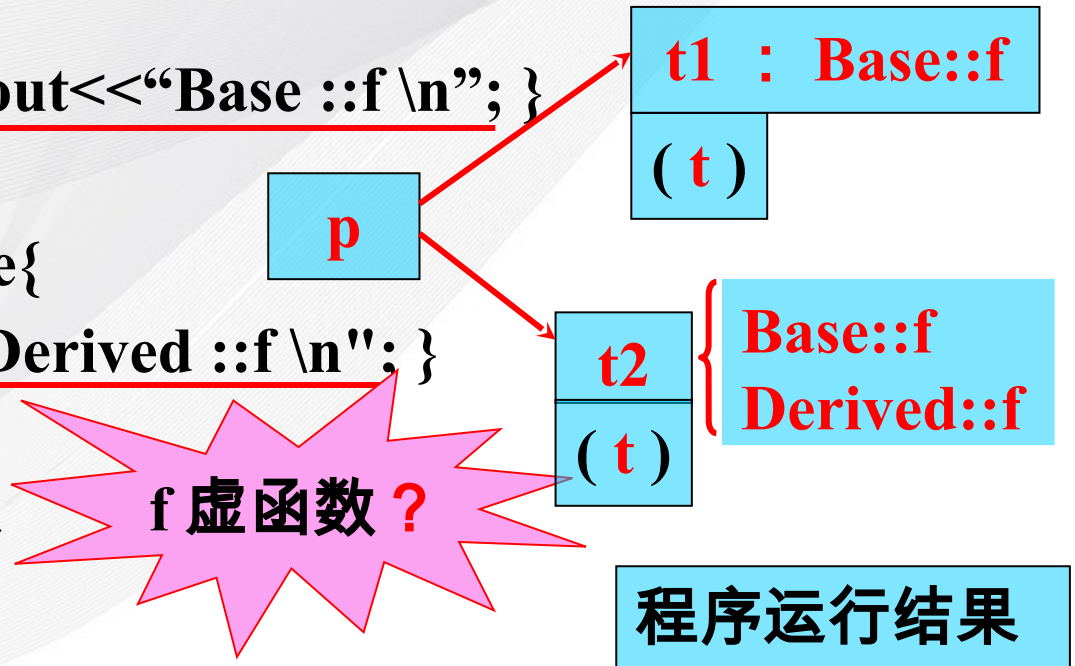
**基类对象引用名.虚函数名(实参表)**

**【例 7-10】通过基类指针和引用，用虚函数实现动态多态性示例。**

# 7.4 虚函数与多态性

【源程序代码】

```
class Base{
public:virtual void f(){ cout<<“Base ::f \n”; }
};
class Derived:public Base{
public:void f(){ cout<<"Derived ::f \n"; }
};
void fa(Base *p){ p->f();}
void fb(Base &t){ t.f();}
int main( ){   Base t1;     Derived t2;
     fa(&t1);  fa(&t2);  fb(t1);  fb(t2);
     return 0;
}
```



# 7.4 虚函数与多态性

## 7.4.3 纯虚函数与抽象类

### 1. 纯虚函数

- 纯虚函数是只声明、未定义的虚函数，无函数体。

virtual 函数类型函数名 ( 形参列表 )=0 ；

或

函数类型 virtual 函数名 ( 形参列表 )=0 ；

### 2. 抽象类

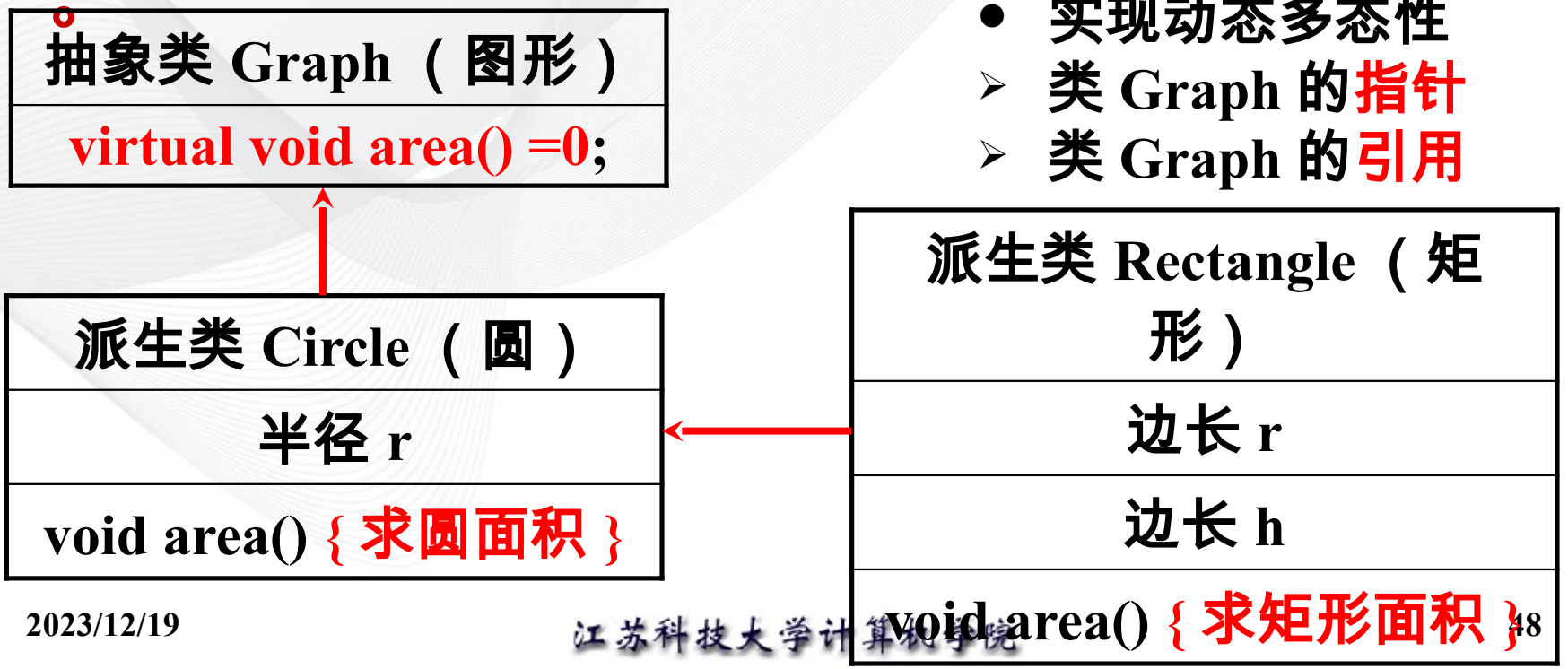
- 含纯虚函数的类是一个不完整的类，不能创建对象；
- 含纯虚函数的类称为**抽象类**
  - 抽象类只能作为派生类的基类；
  - 可以定义抽象类的指针和对象引用，并指向或引用派生类的对象。

# 7.4 虚函数与多态性

## 3. 纯虚函数实现动态多态性

- 只要在派生类中重写基类纯虚函数的函数体；
- 便可用派生类实现动态多态性。

**【例 7-11】设计程序，用纯虚函数实现动态多态性**




- 实现动态多态性
  - 类 Graph 的指针
  - 类 Graph 的引用

# 7.4 虚函数与多态性

【源程序代码】

```
class Graph{                                    // 定义图形类（抽象类）
public:
    virtual void area() =0;                     //  纯虚函数
};
class Circle:public Graph{                      // 定义圆类
protected:
    double r;
public:
    Circle (double x) { r=x; }
    void area() {                               //  重写纯虚函数
        cout<<" 半径为 "<<r;
        cout<<" 的圆面积为 "<<3.14*r*r<<endl;
    }
};
```

# 7.4 虚函数与多态性

【源程序代码】

```
class Rectangle:public Circle{                          // 定义矩形类
    double h;
public:
    Rectangle(double x,double y): Circle (x){ h=y; }
    void area() { cout<<" 边长为 "<<r<<" 和 "<<h; //  重写虚函
      数
        cout<<" 的矩形面积为 "<<r*h<<endl;
    }
};
int main(){
    Graph *p;
    Circle c(10);          p=&c;  p->area();
    Rectangle r(4,5);  p=&r;  p->area();
    return 0;
}
```

```
void f ( Graph &t ) {   t.area(); }
f ( c );
f ( r );
```

程序运行结果：

```
半径为 10   的圆面积为 314
边长为 4   和 5   的矩形面积为
20
```

如何用基类的引用实现？

# 7.5 程序举例

**【例 7-12 】分析含对象成员的派生类构造函数和析构函数的调用过程，并写出程序运行结果。**

**【源程序代码】**

```
class A{ protected: int a;
public:
    A(int x) { a=x; cout<<" 调用类 A  构造函数 \n"; }
    void show() { cout<<a<<'\n'; }
    ~A() { cout<<" 释放成员 a\n";}
};
class B{ protected: int b;
public:
    B(int x) { b=x; cout<<" 调用类 B  构造函数 \n"; }
    ~B() { cout<<" 释放成员 b\n";}
};
```

# 7.5 程序举例

【源程序代码】

```
class C:public B{  int c;                    // 定义子类
    A obj;                  // 对象成员，类 A 的对象
public:
    C(int x,int y,int z):      obj(y) , B(z)   {
        c=x; cout<<" 调用类 C  构造函数 \n";
    }
    void print() {
        cout<<" 对象成员 :"; obj.show();
        cout<<" 派生成员 :"<<b<<'\n';
        cout<<" 普通成员 :"<<c<<'\n';
    }
    ~C() { cout<<" 释放成员 c\n";}
};
int main(){   C test(10,20,30);   test.print();   return 0; }
```

初始化派生成员

初始化对象成员

cout<<obj.a <<'\n';?

```
调用类 B 构造函数
调用类 A 构造函数
对象成员 : 10
派生成员 : 20
普通成员 : 30
释放成员 c
释放成员 a
释放成员 b
```

# 7.5 程序举例

- 复杂派生类包含从基类继承的派生成员、新增的对象成员和新增的普通成员；
- 派生成员和对象成员的初始化，在派生类构造函数头部完成：
  - 派生成员通过基类名调用基类的构造函数实现；
  - 对象成员通过对象名调用对象所属类的构造函数实现。
- 普通成员通常在派生类构造函数的函数体内实现；
- 生成复杂类对象构造函数的调用顺序：
  (1) 调用虚基类构造函数；
  (2) 按继承顺序调用基类构造函数；
  (3) 按对象说明顺序调用对象的构造函数；
  (4) 执行派生类构造函数的函数体。
- 析构函数调用顺序：与构造函数的调用顺序相反。

基类和对象构造函数调用顺序与派生类构造函数头部列表顺序无关！

# 7.5 程序举例

## 【例 7-13】分析下列程序，写出运行结果。

【源程序代码】

```
class Base{
public:
    Base(char *s="string") { cout<<s<<endl; }
};
class A:virtual public Base{                    // 虚基类
public:
    A(char *s1,char *s2):Base(s1) { cout<<s2<<endl; }
};
class B:public virtual Base{                    // 虚基类
public:
    B(char *s1,char *s2):Base(s1) { cout<<s2<<endl; }
};
```

# 7.5 程序举例

【源程序代码】

```
class AB:public A,public B{                          // 多基类继承
public:
    AB(char *s1,char *s2,char *s3,char *s4):B(s1,s2),A(s3,s4)
    { cout<<s2<<endl; }
};
int main(){
    AB test("stringA","stringB","stringC","stringD");
    return 0;
}
```

虚基类有默认的构造函数，可不列出其调用

构造函数调用顺序：Base→A→B→AB

程序运行结果
```
string
stringD
stringB
stringB
```

非虚基类时，构造函数调用顺序：Base→A→Base→B→AB

程序运行结果
```
stringC
stringD
stringA
stringB
stringB
```

# 7.5 程序举例

**【例 7-14】根据赋值兼容性与支配规则写出下列程序的运行结果。**

**【源程序代码】**

```
class MyclassA{
public:
    int val;
    MyclassA(int x) { val=x; }
};
class MyclassB:public MyclassA{
public:
    int val;
    MyclassB(int x):MyclassA(2*x)
    {  val=x;  }
};
```

# 7.5 程序举例

【源程序代码】

```
class MyclassC:public MyclassB{
public:
    int val;
    MyclassC(int x):MyclassB(2*x)
    {  val=x;  }
};
int main(){
    MyclassC test(3),*pc=&test;
    MyclassB *pb=&test; MyclassA *pa=&test;
    cout<<pa->val<<'\n';
    cout<<pb->val<<'\n';
    cout<<pc->val<<'\n';
    return 0;
}
```

test

MyclassA::val=12
MyclassB:: val=6
MyclassC::val=3

程序运行结果
12
6
3

基类指针指向派生类对象时，默认访问的是从该基类继承到派生类中的派生成员，派生类指针默认访问的是派生类的新增成员。

# 7.5 程序举例

**【例 7-15】根据虚函数与动态多态性，写出下列程序的运行结果。**

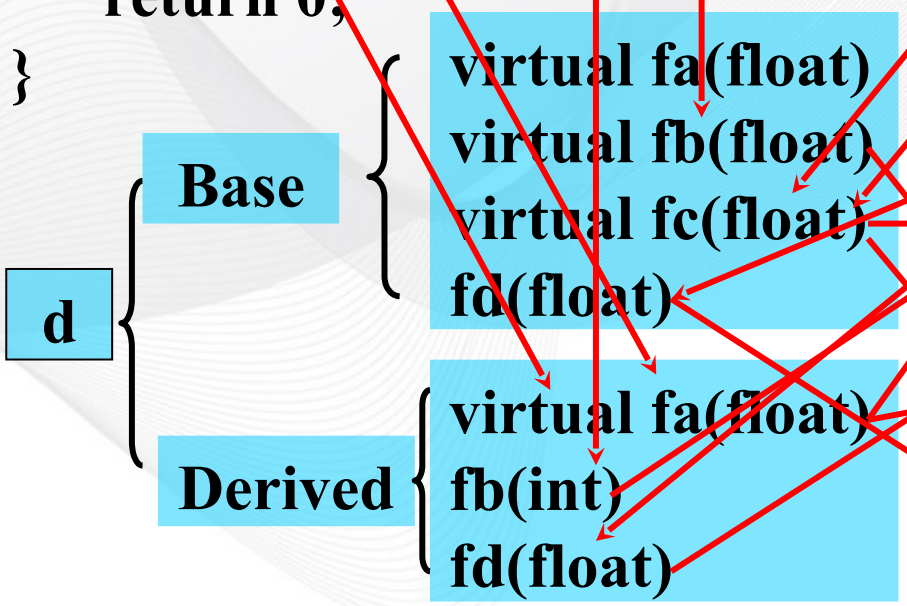
**【源程序代码】**

```
class Base{
public:
    virtual void fa(float x) { cout<<"Base::fa\t"<<x<<endl; }
    virtual void fb(float x) { cout<<"Base::fb\t"<<x<<endl; }
    void virtual fc(float x) { cout<<"Base::fc\t"<<x<<endl; }
    void fd(float x) { cout<<"Base::fd\t"<<x<<endl; }
};
class Derived : public Base{
public:
    void fa(float x) { cout<<"Derived::fa\t"<<x<<endl; }
    void fb(int x) { cout<<"Derived::fb\t"<<x<<endl; }
```

# 7.5 程序举例

【源程序代码】

```
    void fd(float x) { cout<<"Derived::fd\t"<<x<<endl; }
};
int main(void){  Derived d,*pd=&d; Base *pb=&d;
    pd->fa(1.23f); pd->fb(1.23f); pd->fc(1.23f); pd->fd(1.23f);
    pb->fa(1.23f); pb->fb(1.23f); pb->fc(1.23f); pb->fd(1.23f);
    return 0;
}
```




程序运行结果

```
Derived::fa
  1.23
Derived::fb 1
Base::fc 1.23
Derived::fd
  1.23
Derived::fa
  1.23
Base::fb 1.23
Base::fc 1.23
```

# 7.5 程序举例

- 派生类指针指向派生类对象时
  - 默认引用的是派生类的新增成员（支配规则）；
  - 若无新增成员，则引用基类继承来的派生成员。
- 基类指针指向派生类对象时
  - 若调用的是虚函数则引用派生类的新增成员（运行多态性）；
  - 若调用的不是虚函数，则引用基类继承来的派生成员。

# 7.6 习题

1．教师月工资的计算公式为：基本工资 + 课时补贴。教授的基本工资为 5000 元，补贴为 50 元 / 课时；讲师的基本工资为 3000 元，补贴为 20 元 / 课时。设计一个程序求教授和讲师的月工资。具体要求如下。

（1）定义教师类 Teacher 作为基类，数据成员包含姓名、月工资和月授课时数，以及构造函数（初始化姓名和月授课时数）、输出数据成员的函数。

（2）定义类 Teacher 的公有派生类 Professor 表示教授，公有派生类 Lecturer 表示讲师，并分别计算其月工资。

（3）在主函数中对定义的类进行测试。

# 7.6 习题

2．设计一个程序求两点间的距离，具体要求如下。

（1）定义表示平面直角坐标中点的类 Point 作为基类，包含数据成员横坐标和纵坐标，初始化坐标的构造函数，以坐标形式输出一个点的输出函数。

（2）定义类 Point 的公有派生类 Distance ，新增 Point 类的对象 p ，与从 Point 继承的数据成员构成两个点，以及表示两点间距离的数据成员；求两点间距离的成员函数，输出两个点的函数。

（3）在主函数中对定义的类进行测试。

# 7.6 习题

3．设计一个程序求正方形和长方形的周长，具体要求如下。

（1）定义正方形类 Square 作为基类，包含数据成员边长和周长，以及构造函数、求正方形周长的虚函数、输出函数。

（2）定义类 Square 的公有派生类 Rectangular ，新增边长，与派生成员共同作为长方形边长，以及求长方形周长和输出数据成员的函数。

（3）在主函数中对定义的类进行测试，用基类的指针实现动态多态性。

# 7.6 习题

4．设计一个程序输出汽车信息，具体要求如下。

（1）定义汽车类 Auto 作为抽象类，包含车牌号、车轮数等数据成员，以及构造函数、输出车辆信息的纯虚函数。

（2）定义类 Auto 的公有派生类 Car 表示小客车，新增核载人数，重新定义输出函数。

（3）定义类 Auto 的公有派生类 Truck 表示货车，新增核载吨位，重新定义输出函数。

（4）定义用基类对象引用实现动态多态性的外部函数 fun 。

（5）在主函数中调用 fun 函数，完成测试。