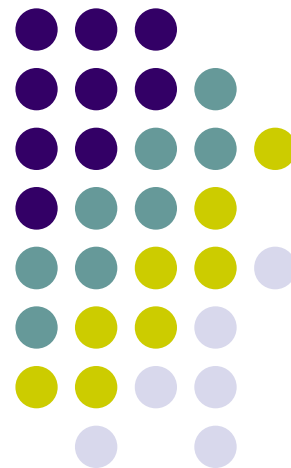


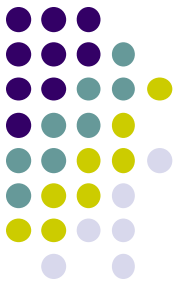
继承与派生

吴清锋

2021年春

目标：
更好更快地设计类





前言

- 基础：学习了类和对象,了解了面向对象程序设计的两个重要特征:**数据抽象与封装**,能设计**基于对象的程序**.
- 要较好地进行**面向对象程序设计**,还必须了解：继承和多态.
- 继承是面向对象程序设计最重要的特征.

那究竟为何要学习继承呢？



从现实世界视角：

- 客观事物之间都有一定的联系，表现在同类事物间的共性和特性。如：车、小汽车、卡车等概念之间既有共性也有特性。

从编程视角：

- 通过继承，可以通过重用并扩展**已有的**类定义，**创建**新类，适应新的需求。
- 体现软件的复用性（不需要重头再来过）！使得开发工作能够站在“巨人”的肩膀上，提高程序开发效率。

提纲

- 继承与派生的基本概念
- 继承方式（私有继承和保护继承）
- 完善派生类
- 公有继承下的赋值兼容规则
- 派生类的构造和析构
- 多继承
- 二义性

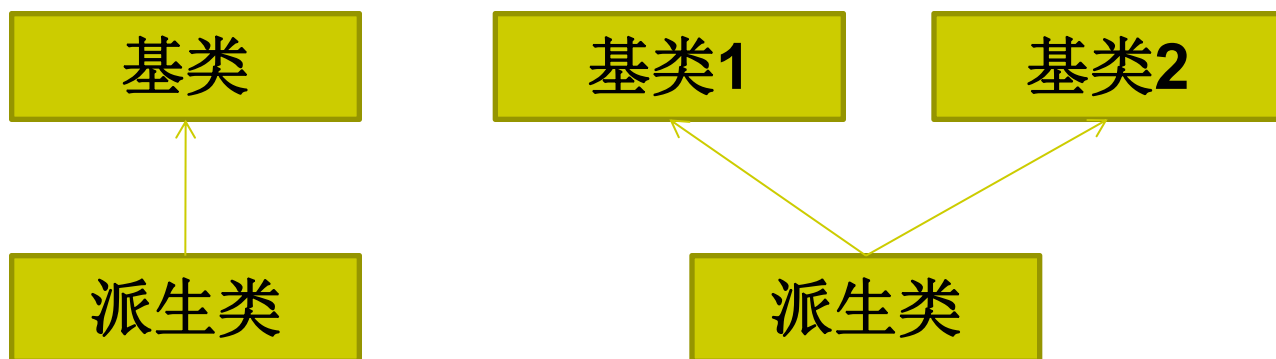
什么是继承？

- 一个编程中的问题
 - 背景：已经有一个类，现在还需要定义第二个类，发现其内容基本相同或有一部分相同
 - 期望，能否利用原来声明的类为基础，加上新内容，减少重复的工作
- 继承作为C++语言中类机制的一部分，使类与类之间可以建立一种上下级关系，可以通过提供来自另一个类的操作和数据成员来创建新类，程序员只需在新类中定义已有类中没有的成分即可建立新类；

若干基本概念

一个事情
两个角度

- 继承：新的类从已有类那里得到已有的特性
- 派生：从已存在的类产生一个新的类；



- 基类和派生类是相对而言的，
 - 基类（父类）：在继承关系中,被继承的类（或已存在的类）；
 - 派生类（子类）：通过继承关系定义出来的新类（新建立的类）；
- 基类和派生类的关系：派生类通过增加信息将抽象的基类变为某种有用的类型。派生类是基类定义的延续。

派生类的声明方式

本质就是声明一个新的类

- 一般形式为：

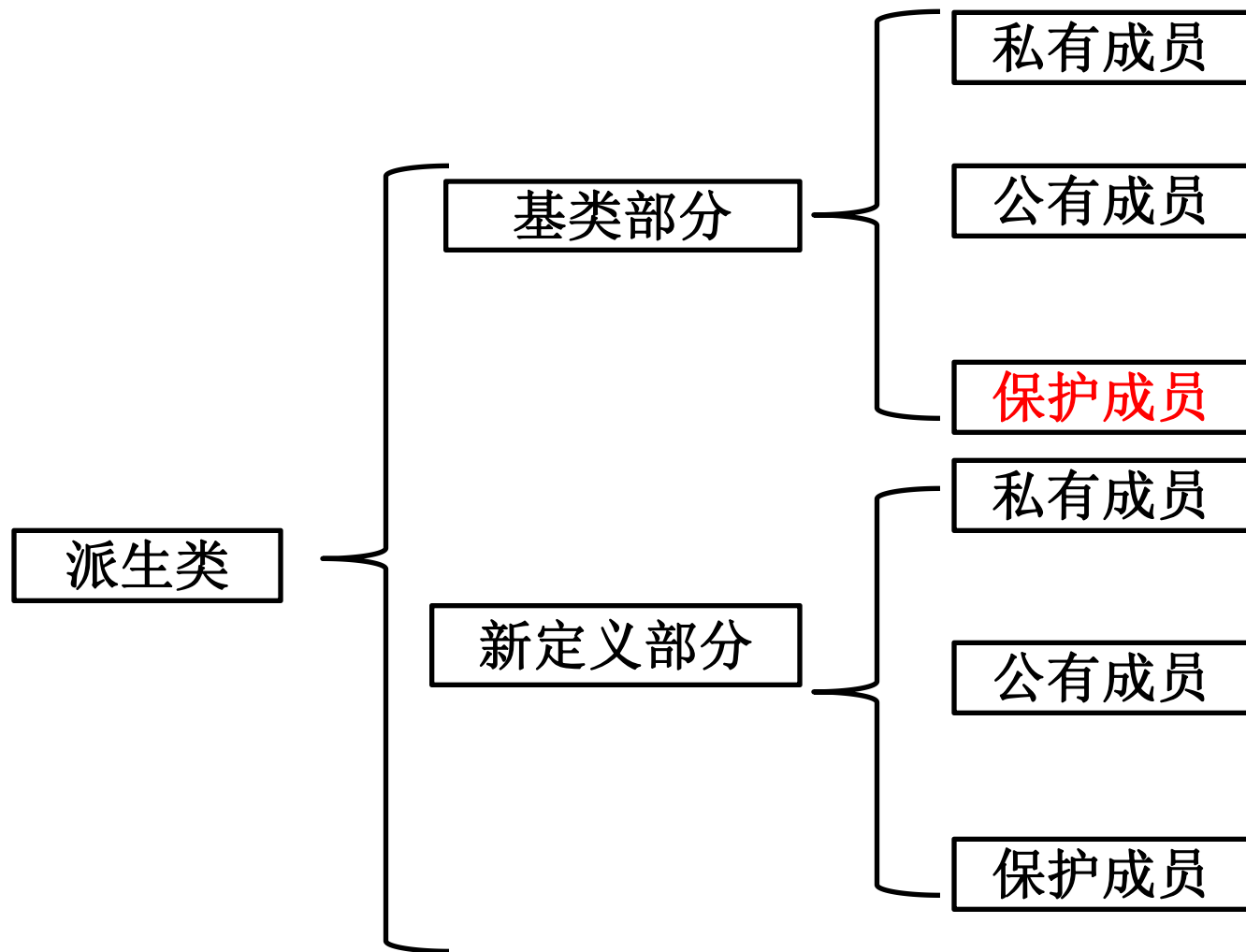
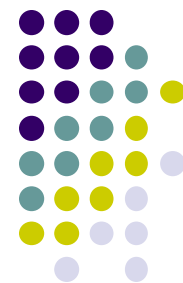
```
class 派生类名: [继承方式] 基类名
{
    派生类新增加的成员
};
```

有三种：公有**public**、私有**private**和保护**protected**

- 其中：

- 继承方式：即派生类的访问控制方式，用于控制基类中声明的成员在多大的范围内能被派生类的用户访问。
- 派生类新增加的成员：是指除了从基类继承来的所有成员之外，新增加的数据成员和成员函数。

定义背后：了解派生类构成



派生类自动继承了基类的成员，但不继承基类的构造函数和析构函数。

定义背后：基类中的protected

- protected成员：缓解继承与数据封装的矛盾
 - 与私有成员一样，在基类中，不能被使用类程序员进行公共访问，但可以被类内部的成员函数访问；
 - 如果使用的类是派生类成员，则可以被访问（即，能够被派生类的成员函数访问）；

```

1. class A {
2.     protected:
3.         int x,y;
4.     public:
5.         void f();
6. };
//A为基类
    
```

```

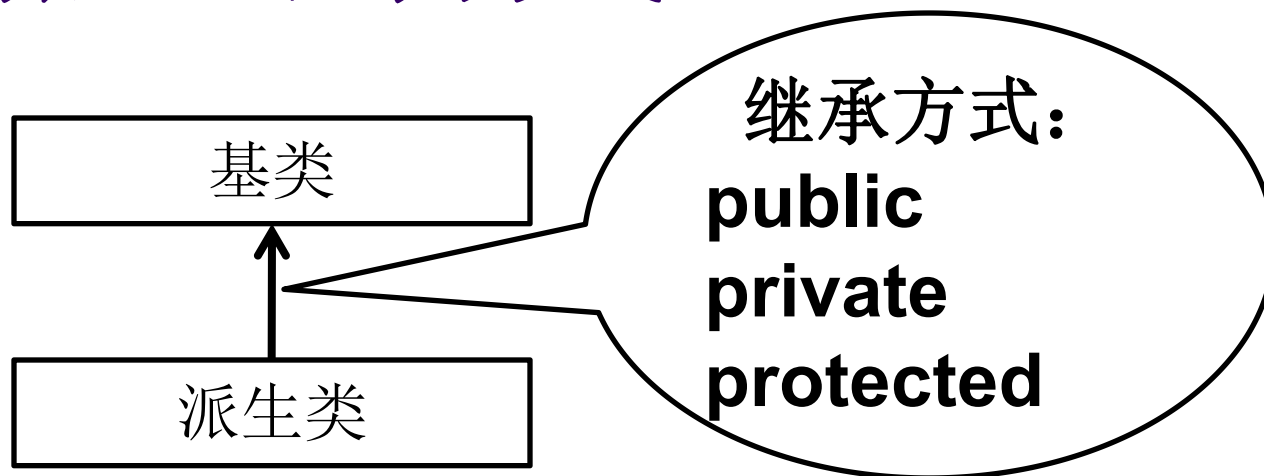
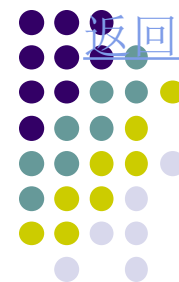
1. class B:public A {
2.     public:
3.         void h() {
4.             ..x...y..
5.             //OK
6.             f();    //OK
7.         }
    
```

//B为派生类

```

1. void g() {
2.     A a;
3.     ..a.x..
4.     //Error
5.     ..a.y..
6.     //Error
7.     a.f(); //OK
8. } //访问属性
    
```

定义背后：继承方式



- 继承方式.如果省略，则默认为private方式；
- 不同的继承方式，导致具有不同访问属性的基类成员在派生类中具有新的访问属性

访问控制：在代码中需要注意

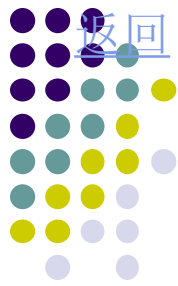
- 不同继承方式的影响主要体现在：

- 派生类成员函数对**基类成员**的访问权限

基类中的成员函数可以访问基类中的任一成员的，那么派生类中新增加的成员函数是否可以同样地访问基类中的私有成员？

- 通过派生类对象对**基类成员**的访问权限

在派生类外，能否通过派生类的对象名访问从基类继承的成员？



公用继承方式

- 采用公用继承方式，则
 - 基类的公用成员和保护成员在派生类中仍然保持其公用成员和保护成员的属性；
 - 基类的私有成员在派生类中**并没有**成为派生类的私有成员（派生类中不可见），它仍然是基类的私有成员，只有基类的成员函数可以引用它，而不能被派生类的成员函数引用；
 - 通过派生类的对象只能访问基类的public成员；
- 若非如此，则将破坏基类的封装性；

公有继承的例子

```
#include<iostream.h>
class A{
private:
    int x;
public:
    void Setx (int i) { x = i ;}
    void Showx () { cout<<x<<endl ;}
};

class B:public A{
    //究竟继承了什么该如何用
private:
    int y ;
public:
    void Sety (int i) { y = i ;}
    void Showy ()
    { Showx () ;
      cout<<y<<endl ;}
};
```

公有

能否描述**B**类中究竟有啥？
派生类**B**中

继承下来的基类的私有成员**x**
继承下来的公有的**Setx\Showx**
新定义的私有成员**y**
新定义的公有成员函数

- 有，但是不等于能访问
- 焦点：基类成员

```
void main ()
```

```
{
    B b ; //派生类对象
    b.Setx (10) ; //继承的基类公有
    b.Sety (20) ;
    b.Showy() ;
}
执行该程序，输出结果如下：
10
20
```



B 类外

有，但是不等于能访问

私有继承方式

- 采用私有继承方式，则
 - 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；
- 例子
- 编程的借鉴：对于无需再往下继承的类的功能可以采用私有继承方式将其隐蔽起来！

私有继承情况下，派生类对基类成员访问的例子

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

```
class A{ // 定义基类
```

```
    int x ;    // 基类私有成员
```

```
public:
```

```
    void setx(int i) { x = i ; } // 基类公有成员函数
```

```
    void showx( ) { cout<<x<<endl ; } // 基类公有成员函数
```

```
};
```

```
class B:private A{
```

// 定义一个派生类,继承方式为私有继承

```
    int y ;
```

```
public:
```

```
    void setxy( int m,int n)
```

```
    { setx(m);
```

//派生类成员函数访问基类公有成员

//函数是合法的

```
        y = n ;
```

```
    }
```

```
    void showxy( )
```

```
    { showx( ) ; cout<<y<<endl ; }
```

```
};
```

```
void main( ){
```

```
    A a;    //基类的对象，未涉及派生
```

```
    a.setx(5);
```

```
    a.showx();
```

```
    B b;    // 定义派生类对象b
```

```
    b.setxy(10,20);
```

```
    b.showx();
```

```
    //访问派生类的私有成员，非法！
```

```
    b.showxy()
```

```
}
```


保护继承方式

- 采用保护继承方式，则
 - 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；

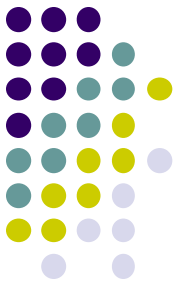
不同继承方式的基类特性和派生类特性



继承方式	基类特性	在派生类展现的特性	派生类成员函数	派生类对象 (main函数)
公有继承	public	public	可以	可以
	protected	protected	可以	不可以
	private	不可访问	不可以	不可以
私有继承	public	private	可以	不可以
	protected	private	可以	不可以
	private	不可访问	不可以	不可以
保护继承	public	protected	可以	不可以
	protected	protected	可以	不可以
	private	不可访问	不可以	不可以

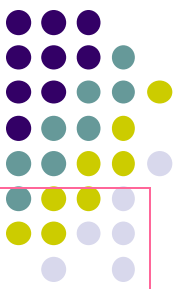
公有继承时：派生类的对象可以访问基类中的公有成员，派生类的成员函数可以访问基类中的公有成员和保护成员。

私有继承时：基类的所有成员不能被派生类的对象访问，而派生类的成员函数可以访问基类中的公有成员和保护成员。



如何把派生类设计的更好？

- 派生一个新类的过程：
 - 1、继承（吸收）基类成员
 - 2、改造基类成员
 - A 通过派生类定义时的**继承方式**来控制，改变基类成员在派生类的**访问控制属性**；
 - 恢复或调整访问控制
 - B 定义同名成员覆盖（屏蔽）基类成员；
 - 继承成员的重命名或重定义
 - 3、添加派生类新成员



第一个方式：恢复访问控制方式（比较少用）

- 前提：在派生类中该成员是可见的，即非隐藏的
- 若希望基类某些成员的访问控制方式在`private`或`protected`继承方式下，在派生类中的身份不要改变。即保持在基类中声明的访问控制方式，这可通过恢复访问控制方式的声明来实现。

- 声明格式如下：

访问控制方式：

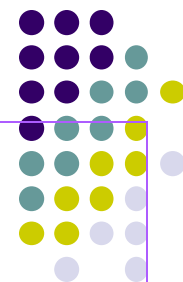
基类名::基类成员名；

- 其中，访问控制方式是`protected`、`public`之一
- 基类成员名是要恢复访问控制权限的基类成员名，对数据成员和函数成员都一样，只写成员名即
- 这种声明是在派生类中针对基类的某个成员进行的。

恢复或调整访问控制方式

```
#include<iostream.h>
class A{
public:
    void f(int i) {cout<<i<<endl ;}
    void g( ) { cout<<"g"<<endl ;}
};
class B: private A{          // 以私有继承方式派生新类B
public:
    void h( ) { cout<<"h"<<endl ;}
    A::f;
// 恢复从基类继承的成员函数f ( ) 的访问控制权限
};
void main( )
{
    B b;
    b.f(6);
//在类B的外部通过派生类对象b访问继承的成员函数f( )
    b.h( ) ;
}
```

说明：派生类B中对继承的成员函数f()进行了恢复访问控制方式的声明，使得该函数在派生类B中仍为公有成员，若没有这种声明，则按继承方式private，在main()函数中是不能访问f()函数的。





第二个方式：继承成员的重命名和重定义

- 在C++的类继承时，首先是将基类的成员全盘接收（除了构造函数和析构函数之外的基类所有成员）。
- 在程序中有时需要对基类成员进行改造或调整。在派生类中对继承成员可以重命名或重定义。
 - 重命名：若私有或保护继承下，对于继承下的成员函数，可定义一个新的、同名的、公有的函数；
 - 重定义：函数名不变，函数体发生改变；
- 当派生类与基类中有相同成员时：
若未明确指明，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覆盖的同名成员，应使用基类名限定。

继承成员的重定义

```
#include<iostream.h>
class Point{           // 定义基类Point
    int x , y ;
public:
    Point( int xx , int yy ) {x = xx ; y = yy ;}
    void move(int m , int n) { x+=m ; y+=n ;}
    int area() { return 0 ;}
    int Getx() { return x ;}
    int Gety() { return y ;}
};

class Rectangle: public Point{
    int w , h ;
public:
    Rectangle ( int a , int b , int c , int d )
    { w = c ; h = d ;}
    void shift ( int i , int j)
    { Point::move(i , j) ;}
    int area() { return w*h ;}
    int Getw() { return w ;} //派生类
    int Geth() { return h ;}
};
```

Point类描述了一个点的位置。由于矩形可以由一个左上角的点以及宽和高构成，因此通过继承**Point**类，增加一些新的数据成员和操作，可以方便定义矩形类。

```
void main ( ){
    Rectangle rect(2,3,20,10);
    rect.shift(3,2);
    cout<<"The data of rect(x,y,w,h):"<<endl ;
    cout<<rect.Getx( )<<","<<rect.Gety( )<<"," ;
    cout<<rect.Getw ( )<<","<<rect.Geth( )<<endl;
    cout<<"The area of that rect is :"
```

执行该程序，输出结果如下：

The data of rect(x,y,w,h):
5,5,20,10

The area of that rect is :200

重命名使派生类可以用不同的名字调用基类中定义的函数。而**重定义**使派生类与基类的同名函数有不同的执行版本。

在调用派生类的公有成员函数时，首先在派生类新定义的成员中寻求匹配，若派生类新定义成员中没有请求的函数，则从直接基类开始在上层基类中寻求匹配。直到找到该函数为止。

公有继承下的赋值兼容规则



- C++中，以public方式继承的派生类可以看成基类的子类型；
- 所谓赋值兼容规则指的是不同类型的对象间允许相互赋值的规定；
 - 面向对象程序设计语言中，在公有派生的情况下，**允许将派生类的对象赋值给基类的对象**，但反过来却不行，即不允许将基类的对象赋值给派生类的对象。这是因为一个派生类对象的存储空间总是大于它的基类对象的存储空间。若将基类对象赋值给派生类对象，这个派生类对象中将会出现一些未赋值的不确定成员。



允许将派生类的对象赋值给基类的对象，有以下三种具体作法：

1.直接将派生类对象赋值给基类对象，例如：

Base objB;

Derived objD; //假设Derived已定义为Base的派生类

ObjB=objD; //合法

ObjD=objB; //非法

2.定义派生类对象的基类引用，例如：

Base &b=objD

3.用指向基类对象的指针指向它的派生类对象，例如：

Base *pb=&objD;

为何讨论这个？

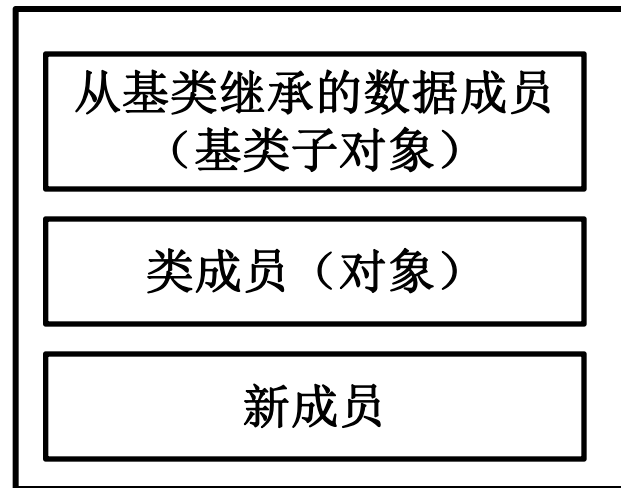


- 赋值可以是函数参数传递的基础；
- “公有继承下的赋值兼容规则”是后续多态的基础

派生类的构造函数 【由类转为对象】



- 派生类对象由基类中继承的数据成员和派生类新定义的数据成员共同构成。



- 在派生类对象中由基类继承的数据成员和成员函数所构成的封装体称为基类子对象。
- 由于构造函数不能被继承，派生类的构造函数除了对派生类新增数据成员进行初始化外，还需要承担为基类子对象初始化的任务，即为（调用的）基类的构造函数提供参数。
- 另外，对于含有其它类对象成员的派生类，还要负责这些对象成员的初始化提供参数，以完成派生类对象的整个初始化工作。

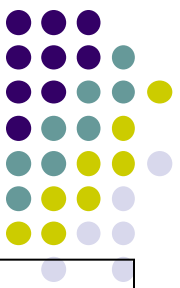


是否一定写派生类的构造函数？

- 基类中若未声明构造函数（此时有：C++派送的默认构造函数），派生类中可以不声明；
- 基类中有无参构造函数或参数带默认值时，派生类既可以不定义构造函数，也可以根据需要定义自己的构造函数，构造函数可以带参数也可以省略（取决于是否要传递信息）

上述，基类构造函数的调用是自动完成的，是隐式调用

- 如果基类仅定义有带参数的构造函数，则派生类必须**显式的定义**其构造函数，并在声明时指定基类的某一构造函数和参数表，把参数传递给基类构造函数。



派生类构造函数的定义格式

派生类构造函数的定义格式如下（采用成员初始化列表形式）

派生类名::派生类名（〈总参数表〉）：

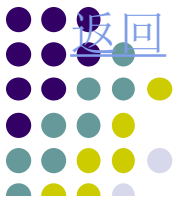
对象成员1（〈参数表1〉），...对象成员n（〈参数表n〉），
基类构造函数（〈参数表n+1〉）

调用基类的构造函数

```
{  
    ...;    //派生类新增数据成员的初始化  
}
```

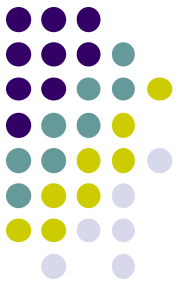
派生类构造函数执行的一般顺序是：

- （1）基类构造函数，
- （2）派生类对象成员类的构造函数（如果有的话），
- （3）派生类构造函数体中的内容。



只能使用构造函数的初始化列表

- 很多时候，初始化列表可以**转化为**通过在函数体内对数据成员赋值来实现。
- 但是，在某些情况下，只能使用初始化列表：
 - 某个类中的类成员，没有默认构造函数；
 - 反思：若没有为类成员提供初始化式，则编译器会隐式地使用类成员的默认构造函数。而该类成员若恰好没有默认构造函数，那么编译器尝试使用默认构造函数就会失败；
 - 某个类中有**const**成员；
 - 某个类中有引用类型的成员；
 - **如果类存在继承关系，派生类必须在其初始化列表中调用基类的构造函数**



派生类构造函数实例

```
#include<iostream.h>
class A{           //定义基类
private:
    int a ;
public:
    A(int x) { a = x ;cout<<"A's constructor called."<<endl ; }
    void show( ) { cout<<a<<endl;}
};

class B{           //定义另一个类，将作为类成员
private:
    int b ;
public:
    B(int x) { b = x ;cout<<"B's constructor called."<<endl ; }
    int get( ) { return b;}
};
```

派生类构造函数实例

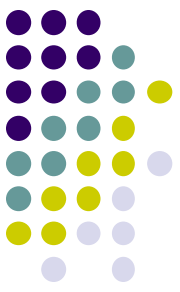
```
class C : public A{                                //定义派生类
private:
    int c;
    B obj_b ;

public:
    C( int x ,int y , int z):A(x),obj_b(y) // 派
    {
        c = z ;
        cout<<"C's constructor called."<<endl ;
    }
    void show( ) {
        A::show( );
        cout<<obj_b.get()<<","<<c<<endl ;
    }
};
```

```
void main(){
    C c1(1,2,5), c2(3,4,7);
    c1.show( ) ;
    c2.show( ) ;
}
```

程序输出如下:

```
A's constructor called.
B's constructor called.
C's constructor called.
A's constructor called.
B's constructor called.
C's constructor called.
1
2,5
3
4,7
```

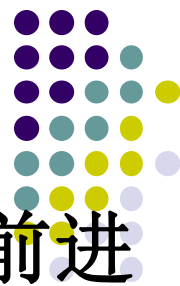



什么时候写派生类的构造函数？

核心点：是否需要将参数传递给基类构造函数

- 派生类构造函数提供了将参数传递给基类构造函数的途径，以保证在基类进行初始化时能够获得必要的参数。因此，如果基类的构造函数定义了一个或多个参数时，派生类必须定义构造函数。
- 如果新增成员中包括成员对象，成员对象的处理情况与基类相同。
- 如果基类中定义了默认构造函数或没有定义任何一个构造函数（由编译器自动生成默认构造函数）时，在派生类构造函数的定义中可以省略对基类构造函数的调用。

派生类析构函数




- 派生类析构函数的功能是在该类对象释放之前进行一些必要的清理工作。
- 由于析构函数也不能被继承，派生类需要定义自己的析构函数。派生类析构函数的定义与一般类的析构函数的定义完全相同。只要在函数体中负责把派生类新增的非对象成员的清理工作做好就可以了，系统会自己调用基类和对象成员的析构函数来对基类子对象和对象成员进行清理。
- 释放派生类对象时，析构函数的执行顺序是：
先执行派生类的析构函数，再执行对象成员类的析构函数（如果派生类有对象成员的话），最后执行基类的析构函数，其顺序与执行构造函数时的顺序正好相反。

派生类析构造函数实例

```
#include<iostream.h>
class X{
    int x1,x2 ;
public:
    X (int i,int j) {x1 = i ; x2 = j ;}
    void print ( ) { cout<<x1<<","<<x2<<endl ; }
    ~X( ) {cout<<"X's destructor called."<<endl; }
};

class Y:public X{
    int y ;           // 派生类Y新增数据成员
public:
    Y( int i , int j , int k) : X(i , j)      { y = k ;}
    //派生类构造函数
    void print( ) {X::print ( ) ;cout<<y<<endl ;}
    ~Y( ) { cout<<"Y's destructor called."<<endl ; }
};
```



```
void main()
{
    Y y1(5,6,7) ,
      y2(2,3,4) ;
    y1.print() ;
    y2.print ( ) ;
}
```

程序输出结果如下：

5, 6

7

2, 3

4

Y's destructor called.

X's destructor called.

Y's destructor called.

X's destructor called.

多继承与多层派生

- 从派生类来看:

- 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

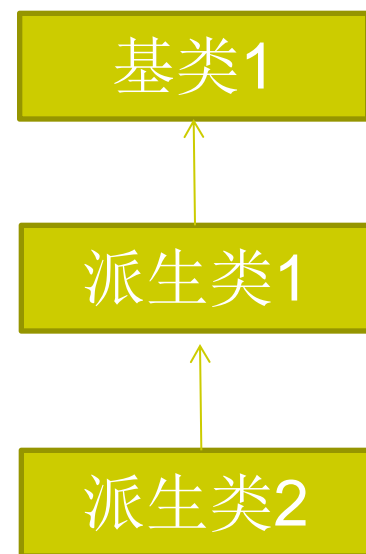
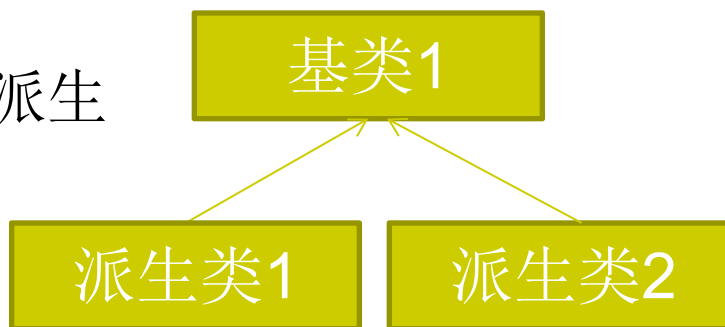
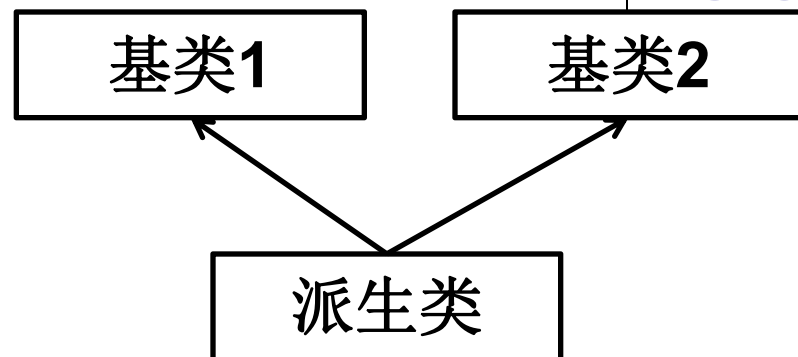
- 从基类来看:

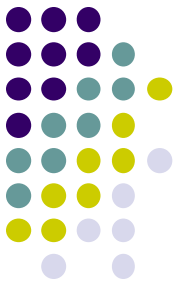
- 多重派生

- 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



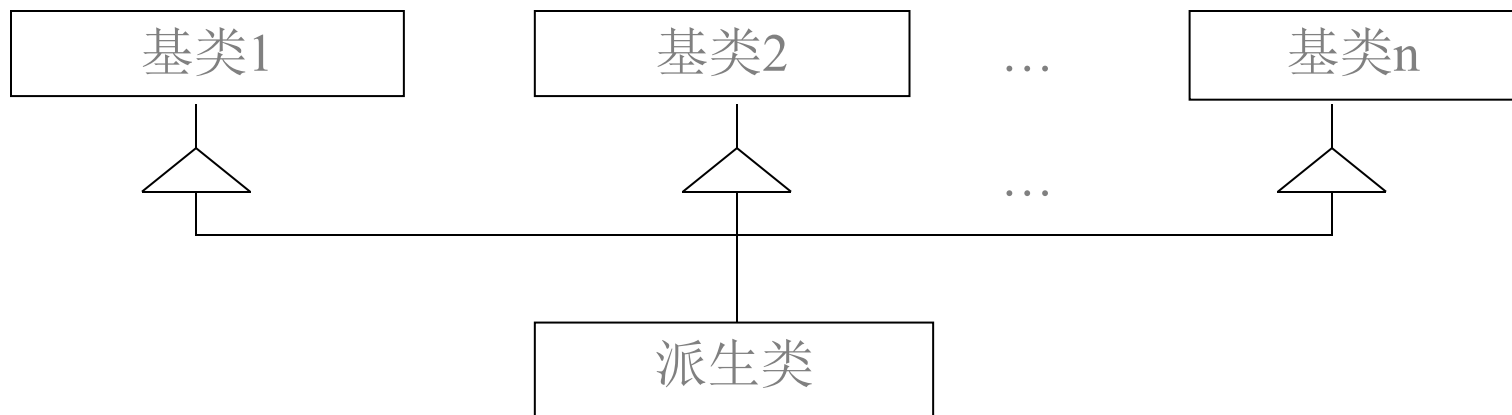
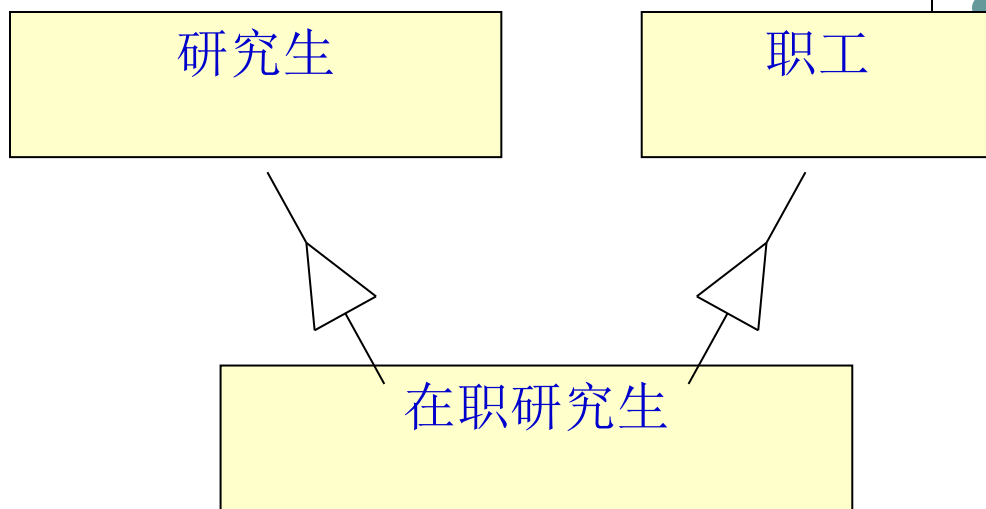


多继承时派生类的声明

```
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。

多继承例子



多继承的派生类与多个基类的关系

多继承派生类的构造函数



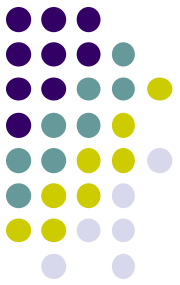
在多继承的情况下，派生类的构造函数格式为：

```
派生类名::派生类构造函数名（〈总参数表〉）： 基类名1  
（〈参数表1〉）， ...,基类名n（〈参数表n〉）， ...,  
子对象1（〈参数表n+1〉）， ...  
{  
    ...; //派生类构造函数函数体  
}
```

多继承派生类构造函数执行顺序是：

- (1) 所有基类的构造函数；多个基类构造函数的执行顺序取决于定义派生类时所指定的顺序，与派生类构造函数中所定义的成员初始化列表的参数顺序无关。
- (2) 子对象（如果有的话）类的构造函数(调用顺序按照它们在类中声明的顺序)；
- (3) 派生类本身构造函数的函数代码。

多继承派生类的构造函数



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

```
class A1{                                // 定义基类A1
```

```
    int a1;
```

```
public:
```

```
    A1(int i){a1=i;cout<<"constructor A1."<<a1<<endl;}
```

```
    void print( ){cout<<a1<<endl;}
```

```
};
```

```
class A2{                                // 定义基类A2
```

```
    int a2;
```

```
public:
```

```
    A2(int i){a2=i;cout<<"constructor A2."<<a2<<endl;}
```

```
    void print(){cout<<a2<<endl;}
```

```
};
```

```
class A3{                                // 定义类A3
```

```
    int a3;
```

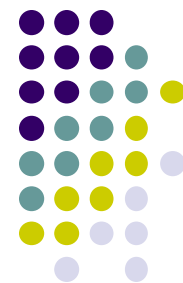
```
public:
```

```
    A3(int i){a3=i;cout<<"constructor A3."<<a3<<endl;}
```

```
    int geta3( ){return a3;}
```

```
};
```


多继承派生类的构造函数



```
class B:public A2 , public A1{ // 定义派生类 B，基类为A1和A2
    int b;
    A3 obj_a3; //对象成员
public:
    B(int i,int j,int k,int l): A1(i),A2(j), obj_a3 (k)//派生类构造函数
    {
        b=l;
        cout<<"constructor B."<<b<<endl;
    }
    void print_b()
    {
        A1::print();
        A2::print();
        cout<<obj_a3. geta3()<<","<<b<<endl;
    }
};

void main( )
{
    B bb(1,2,3,4);
    bb.print_b();
}
```

执行该程序，输出结果如下：

```
constructor A2.2
constructor A1.1
constructor A3.3
constructor B.4
1
2
3,4
```



说明：

1、程序中派生类B的构造函数有四部分组成，分别完成基类A1、A2，派生类子对象a3，B新增数据成员b的初始化工作。该函数的总参数表中有4个参数，分别是基类A1、A2，派生类子对象obj_a3以及派生类B构造函数的参数。

2、在构造函数的成员初始化列表中，两个基类顺序是A1在前，A2在后，而定义派生类B时两个基类顺序是A2在前，A1在后。从输出结果中可以看出：先执行A2的构造函数，后执行A1的构造函数。因此，执行基类构造函数的顺序取决于定义派生类时指定的基类的顺序，而派生类构造函数的成员初始化列表中各项顺序可以任意排列。

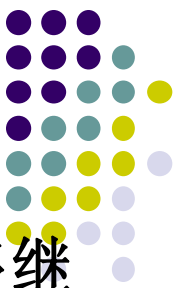


3、作用域运算符::用于解决命名冲突问题。在派生类B中的print_b()函数体中，使用A1::print(); 和A2::print(); 分别指明调用哪一个类中的print()函数

4、派生类对从基类继承的某个函数重新定义后，当在派生类中调用基类的同名函数时会出现二义性，例如，对上述print_b函数以下面形式调用时：

```
void print_b(){  
    print();    // 对print（）的调用有二义性  
    cout<<obj_a3.geta3()<<","<<b<<endl;  
}
```

则系统分不清print()是哪一个基类的。对这种二义性问题应该用作用域运算符::来解决。



总结：多继承中的二义性问题

在派生类中对基类成员的访问应该是唯一的。但是，在多继承情况下，**由于多个基类中包含同名成员**，这可能造成对基类中某个成员的访问出现不唯一情况。这时就称对基类成员的访问产生了二义性。

```
class Base1 {  
    public:  
        void fun();  
};
```

```
class Base2 {  
    public:  
        void fun();  
};
```

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

```
int main() {  
    Derived obj;  
    obj.fun(); //无法确定访问的是哪个基类中的函数  
    return 0;  
}
```

措施1：通过域运算符明确指出访问的哪个基类的函数 **obj.Base1::fun();** 即依赖对象

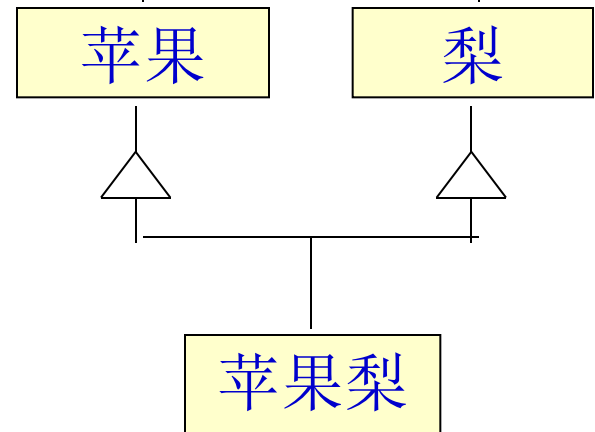
措施2：在类中定义同名成员（覆盖原则）

```
void fun() {  
    Base1::fun(); }
```



二义性问题之二：重复继承

多继承时，一个派生类可能有多基类，若这些基类又有一个共同的基类，这就产生了重复继承问题。在通过重复继承而来的派生类对象访问共同的祖先基类的成员时，可能会产生二义性。



苹果和梨均从类水果继承了数据成员和成员函数，因此在苹果和梨中同时存在着同名的数据成员和成员函数；



重复继承的现象

```
class A
{ protected:
    int a;

public:
    A(int k){a=k;}
};

class B1:public A
{ protected:
    int b1;

public:
    B1(int p, int q):A(q)
    { b1=p;}
};

class B2:public A
{ protected:
    int b2;

public:
    B2(int p, int q):A(q)
    { b2=p;}
};
```

```
class C:public B1,public B2
{
    int c;

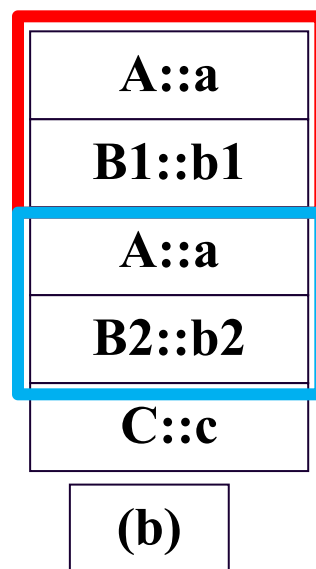
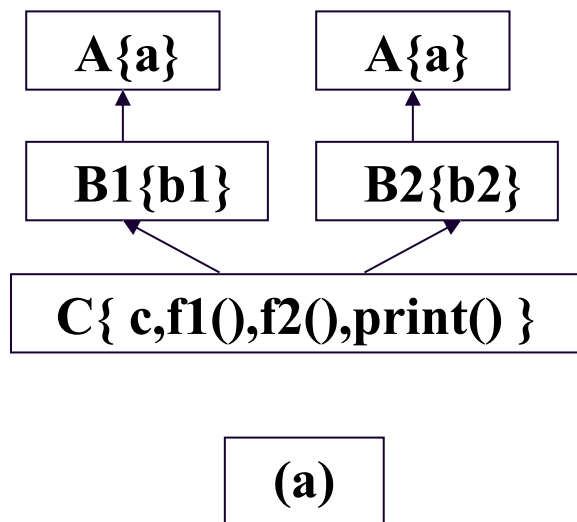
public:
    C(int p, int q, int r, int s):B1(p,q),B2(r,s){ }
    //int f( ){c=a+b1+b2; } 二义性语句，无法区别是究竟是基类B1或是B2中继承下来的成员；
    int f1( ){return c=B1::a+b1+b2;}
    int f2( ){return c=B2::a+b1+b2;}
    void print(){cout<<"Obj data value may equal "
        cout<<f1()<<" or "<<f2()<<endl; }
};

void main()
{
    C obj_c(1,2,3,4);
    obj_c.print();
}
```

程序输出：

Obj data value may equal 6 or 8

重复继承及其二义性



重复继承关系及其内存映象

在上例中若定义：C c1；则对继承的数据成员a的访问 c1.a；或 c1.A::a；都会出现二义性。系统将没有足够的信息区分是访问 B 1中的a，还是 B 2中的a。派生类 C 的对象c 1 中保存了间接基类 A 的成员a的两份拷贝。

为避免出现这种二义性，可以用作用域运算符 :: 加以限定。例如：

c1.B1::a；

c1.B2::a；

要想从根本上消除这种二义性，应使这个公共基类在间接派生类中只产生一个该公共基类的子对象。这就需要将这个公共基类定义为虚基类。

虚基类：有效解决重复继承二义性



- 虚基类与普通基类的区别只有在发生重复继承时才能表现出来：被重复继承的基类被说明为虚基类后，其成员在派生类中只产生一个唯一的副本，这就从根本上解决了二义性问题。
- 虚基类的说明是在派生类的定义中进行
- 虚基类说明的语法格式如下：

```
class 派生类名: virtual 继承方式 基类名
{
    ...;
}
```

其中，**virtual** 是说明虚基类的关键字。虚基类的说明是在定义派生类时，写在派生类名的后面。

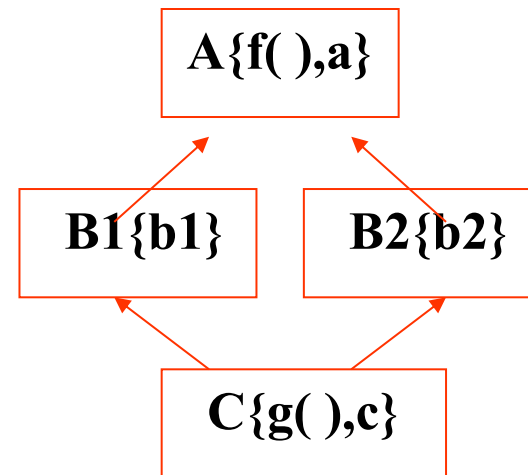
虚基类的说明

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

class B1: virtual public A
{
protected:
    int b1;
};

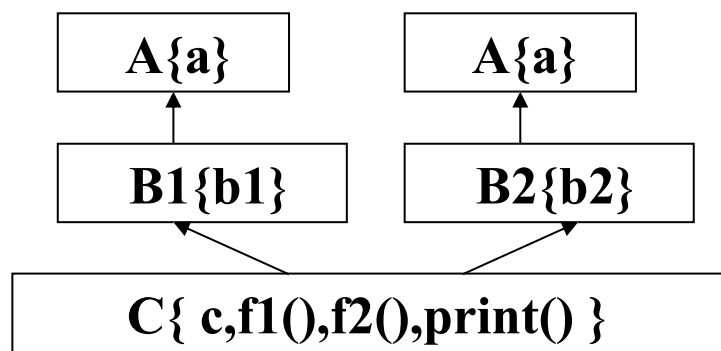
class B2: virtual public A
{
protected:
    int b2;
};
```

```
class c: public B1, public B2
{
private:
    int c;
public:
    C();
    int g( );
}
```

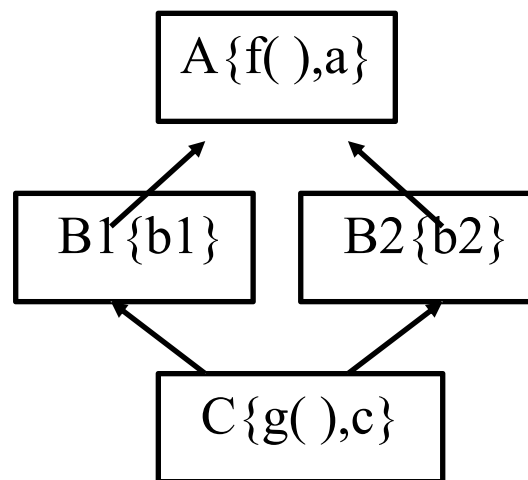


含有虚基类时的类层次关系图

虚基类的功能



重复继承关系



含有虚基类时的类层次关系图

虚基类的构造函数与最后派生类



- 创建派生类对象时，需调用派生类构造函数进行对象的初始化工作。那么，为了初始化其中的基类子对象，派生类构造函数要负责调用基类构造函数。
- 对于虚基类而言，由于**派生类的对象中只有一个虚基类子对象**，为确保虚基类子对象只被初始化一次，该虚基类构造函数必须只被调用一次。

由于继承结构的层次可能很深，规定将在定义对象时所指定的类称为**最后派生类**。**C++规定，虚基类子对象是由最后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**。如果一个派生类有一个直接或间接的虚基类，那么派生类的构造函数的成员初始化列表中必须列出虚基类构造函数的调用，如果没有列出，则表示使用该虚基类的缺省构造函数来初始化派生类对象中的虚基类子对象。

虚基类构造函数

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

```
class A{
```

```
protected:
```

```
    int a;
```

```
public:
```

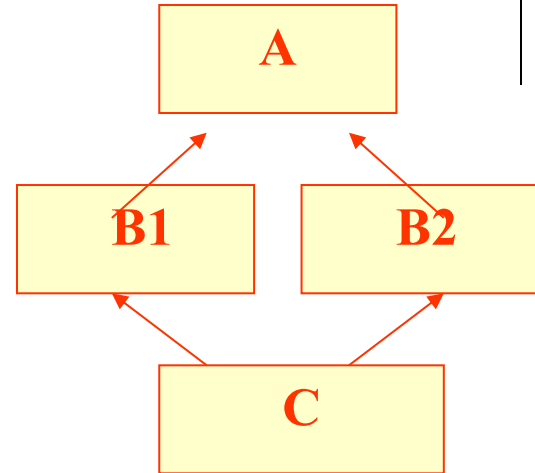
```
    A( int x)
```

```
    { a = x ;
```

```
      cout<<"A's constructor called and a= "<<a<<endl; }
```

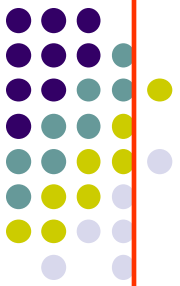
```
    ~A( ){};
```

```
};
```

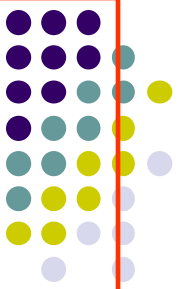


```
class B1:virtual public A{    //说明类A是派生类B1的虚基类
protected:
    int b1;
public:
    B1(int x,int y):A(x)
    { b1 = y ;
        cout<<"B1's constructor called and b1= "<<b1<<endl ;
    }
};

class B2:virtual public A{    //说明类A是派生类B2的虚基类
protected:
    int b2;
public:
    B2( int x,int y):A(x)
    { b2 = y ;
        cout<<"B2's constructor called and b2= "<<b2<<endl;
    }
};
```



```
class C:public B1, public B2{// 由基类B1和B2派生新类C
private:
    int c;
public:
    C( int x , int y ,int z , int w ):B1(x,y), B2(x,z), A(x)
    {
        c = w ;
        cout<<"C's constructor called and c= "<<c<<endl;
    }
};
```



```
void main( )
{
    C *pc=new C(10,20,30,40);
    delete pc;
}
```

执行该程序，输出结果如下：

A's constructor called and a= 10

B1's constructor called and b1= 20

B2's constructor called and b2= 30

C's constructor called and c= 40

说明:

1. 程序中，定义了类A、类B1、类B2和类C，其中类A是类B1和类B2的直接虚基类，类C是由类B1和类B2多继承派生的，类A是类C的间接虚基类。

2. 在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。

3. 在主函数中建立类C的对象时，只有类C的构造函数的成员初始化列表中列出的虚基类A的构造函数被调用，而类C的基类B1和B2中对类A构造函数的调用将被忽略，这样便保证了对虚基类的子对象只初始化一次。

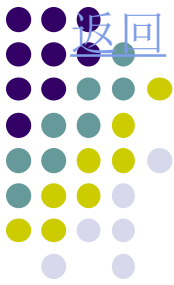
4. 在一个成员初始化列表中出现对虚基类和非虚基类构造函数的调用，则虚基类的构造函数**先于非虚基类构造函数的执行**

继承与组合

- 相关定义：
 - 类的组合：在一个类中以另一个类的对象作为数据成员；
- 继承与组合中的对象的结构（图10-4）
- 性质异同点：
 - 同：是软件重用的重要方式；
 - 异：
 - 继承体现了“是”的关系；组合体现了“有”的关系；
 - 函数调用中参数的传递问题；
 - 对于组合，成员对象的**private**数据必须通过成员对象的操作区间接访问（P335第41行），而继承的访问属性之前讨论过了；

多继承（1）

- 相关定义：
 - 单继承：一个类是从一个基类派生而来的；
 - 多重继承：一个类可以从多个基类派生，即派生类从两个或多个基类中继承所需的属性，这样的继承结构称为多重继承；
- 例子：沙发床，同时继承了沙发和床的特征；
- 声明多重继承的方法
 - 若已声明了类A、类B和类C,则声明多重继承的派生类D
`class D: public A, private B, protected C`
 { 类D新增加的成员 }
 - 说明：D是多重继承的派生类，它以公用继承方式继承A类，以私有继承方式继承B类，以保护继承方式继承C类；D按不同的继承方式的规则继承A，B，C的属性，确定各基类的成员在派生类中的访问权限。
- 例子（P347，f1006.cpp）



基类分解

- 实际也是多重继承引起的二义性问题;
- 问题描述:
 - 图10-6,多重继承中的基类又都从同一个基类派生;
 - 从某个角度来看, **Bed**和**Sofa**均从类**Furniture**继承了数据成员和成员函数, 因此在**Bed**和**Sofa**中同时存在着同名的数据成员和成员函数;
 - 如何访问**Bed** (**Sofa**) 类中从基类继承下来的成员?
 - 图示分析
- 错误的解决方法:
 - `ss.setweight(20);`或`ss.Furniture::setweight();`, 因为无法区别是究竟是类**A**或是类**B**中从基类继承下来的成员;
- 解决方法:
 - 应当通过类**Furniture**的直接派生类名来指明要访问的是哪个派生类的基类成员, 即`ss.Bed::setWeight(20);`

虚拟继承

- 上述问题中的一个现象：

- 在一个类中保留间接共同基类的多份同名成员，它在某些情景下可以在不同的数据成员中分别存放不同的数据，但是：保留多份数据成员的拷贝，不仅占用较多的存储空间，还增加了访问的困难，况且实际还不需要有多份拷贝；

- C++提供虚基类的方法，使得在继承间接共同基类时只保留一份成员，即合并了子对象：

- 图示

- 形式

```
class A {...}; //声明基类A
```

```
class B: virtual public A {...}; //公用派生类，且A是B虚基类
```

```
class C: virtual public A {...};
```

注：虚基类并不是在声明基类时声明的，而是在声明派生类时，指定继承方式时声明的；

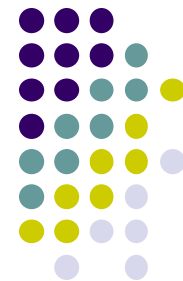
对象构造

- 回顾：在P351，f1008.cpp中关注构造函数
 - 原则：在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化；
 - 执行顺序（P352）
 - 任何虚拟基类的构造函数按照它们被继承的顺序构造；
 - 任何非虚拟基类的构造函数按照它们被继承的顺序构造；
 - 任何成员对象的构造函数按照它们的声明的顺序构造；
 - 类自己的构造函数
- 注意：**若有多个基类构造函数的执行顺序取决于定义派生类时所指定的各个基类的顺序，而与派生类构造函数的成员初始列表中给定的基类顺序无关。

C++多文件应用程序的实现

- C++程序允许由多个文件组成，多文件程序的实现要创建项目文件。
- 相关步骤如下：
 - 编辑程序中的多个文件（该过程与单文件程序一致！）
 - 创建项目文件（Win32 Console Application）；
 - 将多个文件添加到项目中；
 - 编译连接项目文件
 - 运行项目文件

注意，程序文件和项目文件的创建顺序并不限制！



交通工具

飞机

汽车

火车

小汽车

大卡车

大客车

工具车

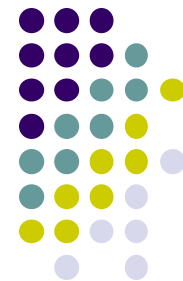
轿车

小面包车

奥迪

本田

千里马



动物

猴子

猫

鸟

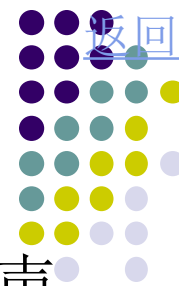
狮子

虎

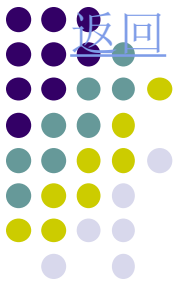
猎豹

[返回](#)

继承的实质和作用



- 继承的实质
 - 继承就是让子类继承父类的属性和操作，子类可以声明新的属性和操作，还可以剔除那些不适合的父类操作。
 - 在新的应用中，父类代码已经存在，无须修改。所要完成的是派生子类，并在子类中增加和修改。
- 继承的作用
 - 传统的程序设计，程序人员需要为每一种应用项目单独地进行一次程序开发，由于每一种应用有不同目的和要求，程序的结构和编码是不同的，人们无法使用已有的软件资源,显然，这种方法的重复工作量很大。究其原因，就是因为过去的程序设计方法和计算机语言缺乏软件重用的机制；
 - 面向对象技术强调软件可重用性；C++语言提供了继承机制，解决软件重用问题。



派生类的生成过程

- 派生新类经历了三个步骤：

- 吸收基类成员

派生类**继承**了基类的全部数据成员以及除了构造函数、析构函数之外的全部函数成员。

- 改造基类成员

对继承到派生类中的基类成员的**改造**包括两个方面：

- 基类成员的**访问方式**问题，这由派生类定义时的访问方式来控制；
 - 对基类数据成员或成员函数的**覆盖**，也就是在派生类中定义了与基类中同名的数据成员或函数成员，由于作用域不同，发生同名覆盖，基类中的成员被替换成派生类中的同名成员；

- 添加新成员

根据需要在派生类中添加新的数据成员和成员函数，以此实现必要的新功能；

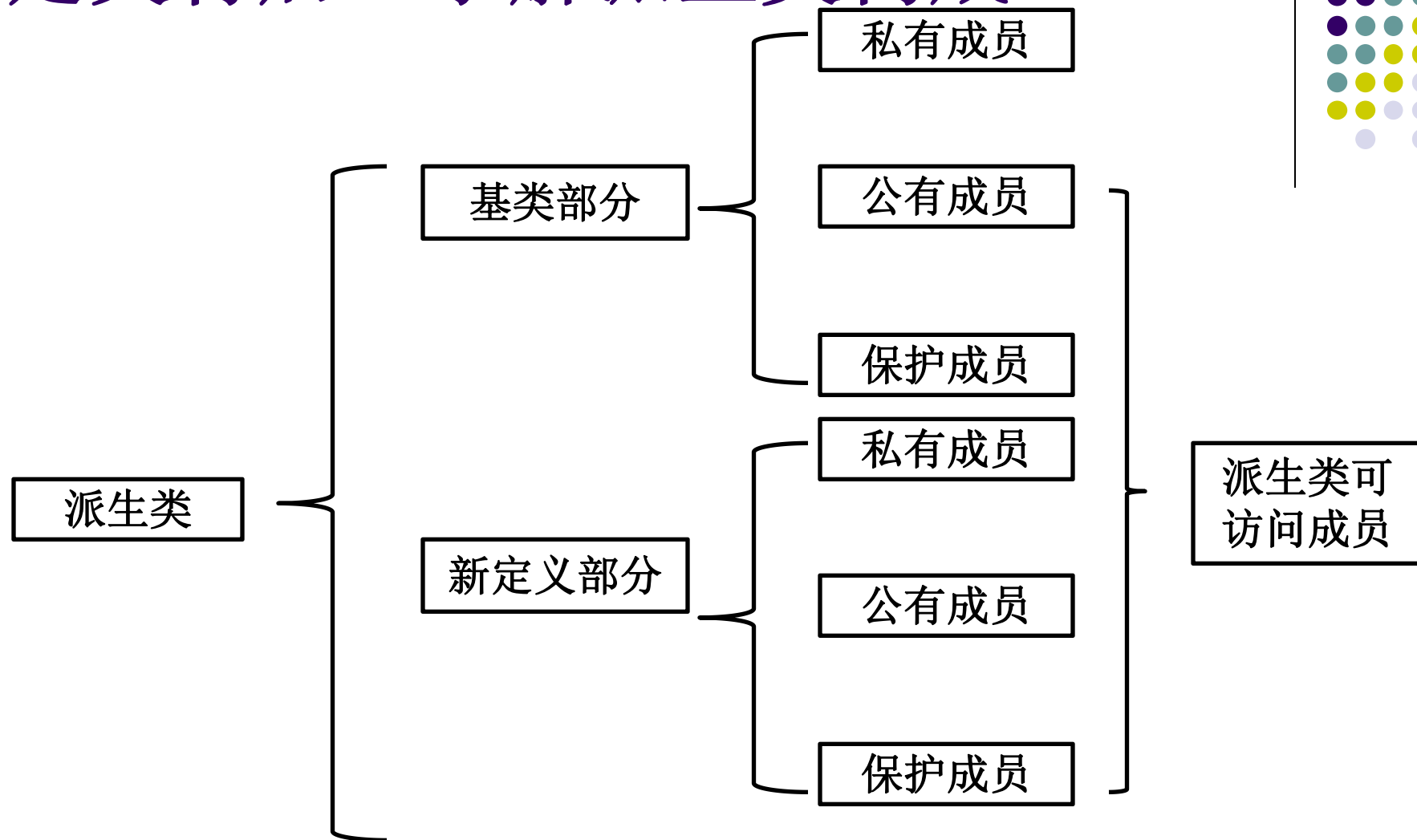
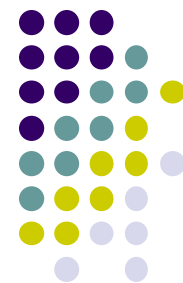
派生类对基类成员的访问能力

基类成员	公有继承 public	私有继承 private	保护继承 protected
私有成员	X	X	X
公有成员	√	√ （私有）	√ （保护）
保护成员	√	√ （私有）	√

两个视角：

- 类对象使用者视角（main函数中）：
- 派生类中访问视角：
 - 基类的成员在派生类中的访问属性的问题，不仅要考虑基类成员所声明的访问属性，还要考虑派生类所声明的对基类的继承方式，根据这两个因素共同决定基类成员在派生类中的访问属性；

定义背后：了解派生类构成



派生类自动继承了基类的成员，但不继承基类的构造函数和析构函数。