C++程序设计(II)

浙江工业大学计算机学院

第10讲 继承机制

1. 继承的基本概念
2. 继承与构造、析构
3. 多重继承和重复继承
4. 继承和组合

第10讲 继承机制-继承的基本概念

C++的继承机制

- IS-A关系：日常生活中，我们常用is-A的关系来组织与表达知识，从而将知识组织成一种有层次、可分类的结构。如，红富士是一种苹果，而苹果又是一种水果；鸭梨是一种梨，梨也是一种水果。
- 面向对象的程序设计方法率先将is-A关系引入到程序设计领域，用于描述类与类之间的关系。我们不必每次都从头定义一个新的类，而是将这个新的类作为一个或者若干个现有类的扩充或特殊化。即利用现有的类来定义新的类。
- C++引入继承机制后，定义一个新的类只需定义那些与其它类不同的特征，那些与其它类相同的通用特征可以通过继承得到。

第10讲 继承机制-继承的基本概念

❖C++的继承机制

- 如果类B继承类A，则在类B中除了自己定义的成员以外，还自动包括了类A中定义的数据成员和成员函数。

  ▲类A称为类B的 基类(base class) 、父类(father class)或者超类(super class)。

  ▲类B称为类A的派生类(derived class)或子类(son class)。

  ▲类B中自动继承下来的成员称为继承成员(derived member)。

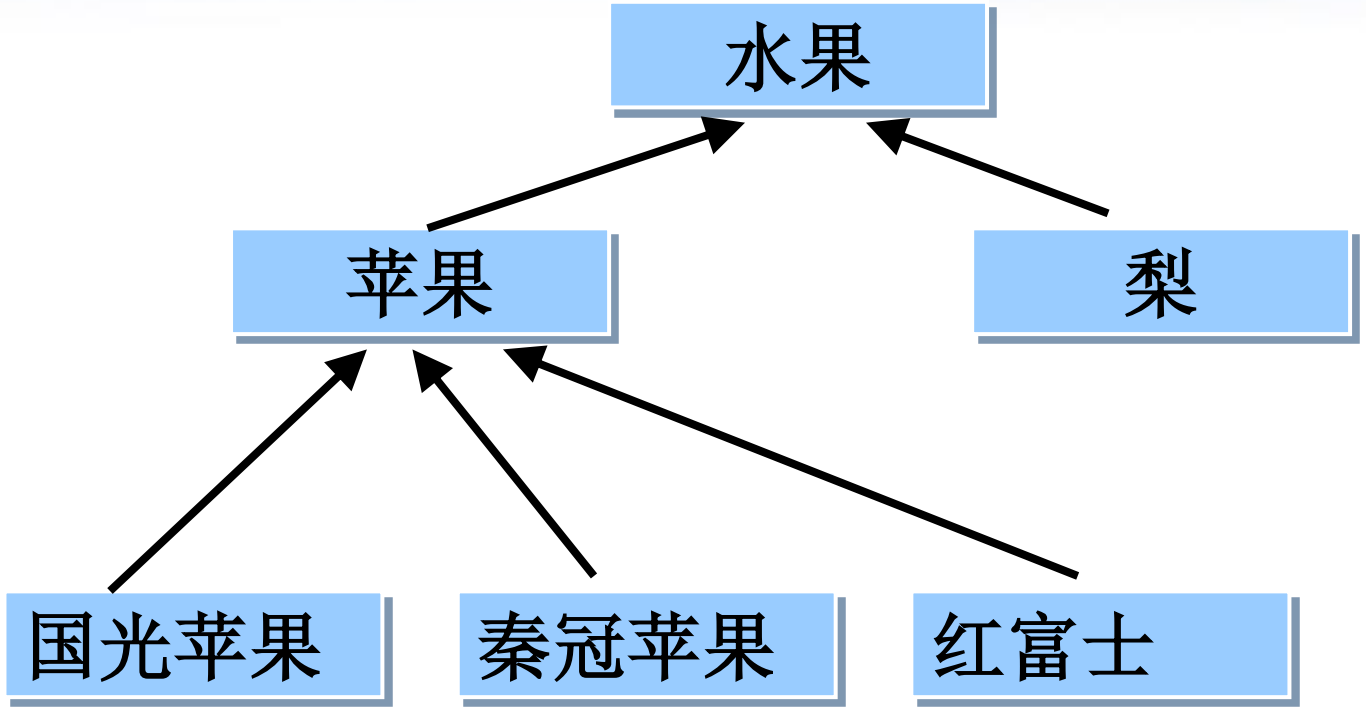
  ▲一个类的祖先类包括该类的父类及父类的祖先类。

  ▲一个类的后代类包括该类的子类以及子类的后代类。

第10讲 继承机制-继承的基本概念

C++的继承机制

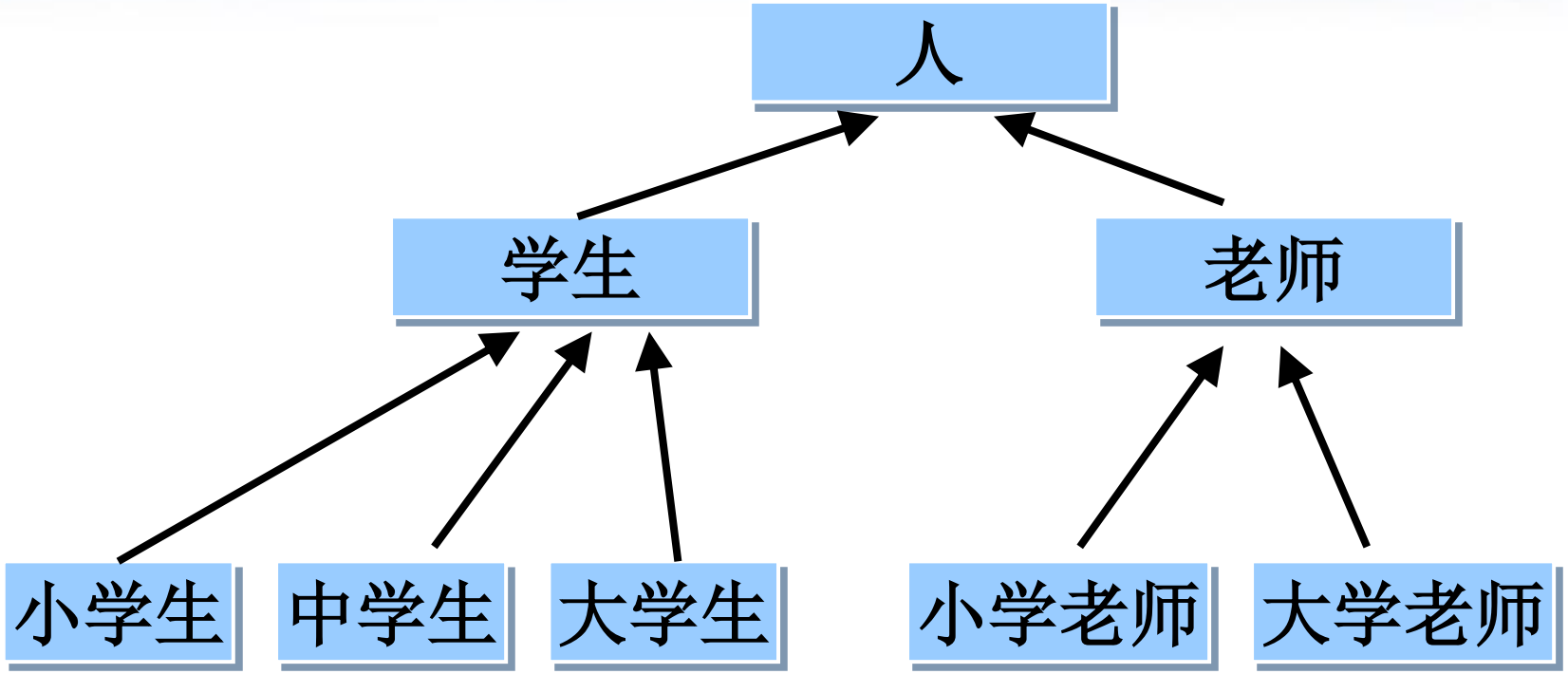
- 继承模型的例子1：



第10讲 继承机制-继承的基本概念

C++的继承机制

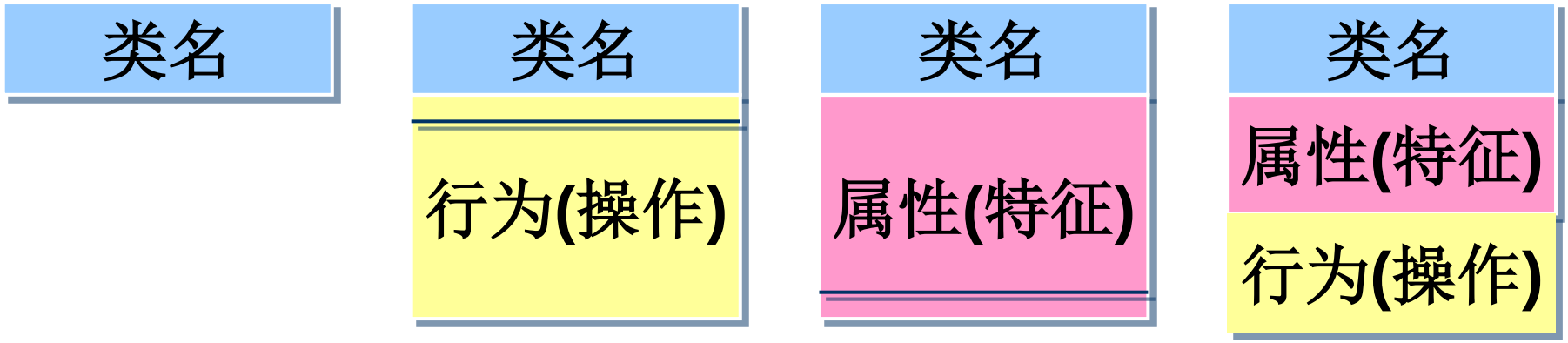
- 继承模型的例子2：



第10讲 继承机制-继承的基本概念

C++的继承机制

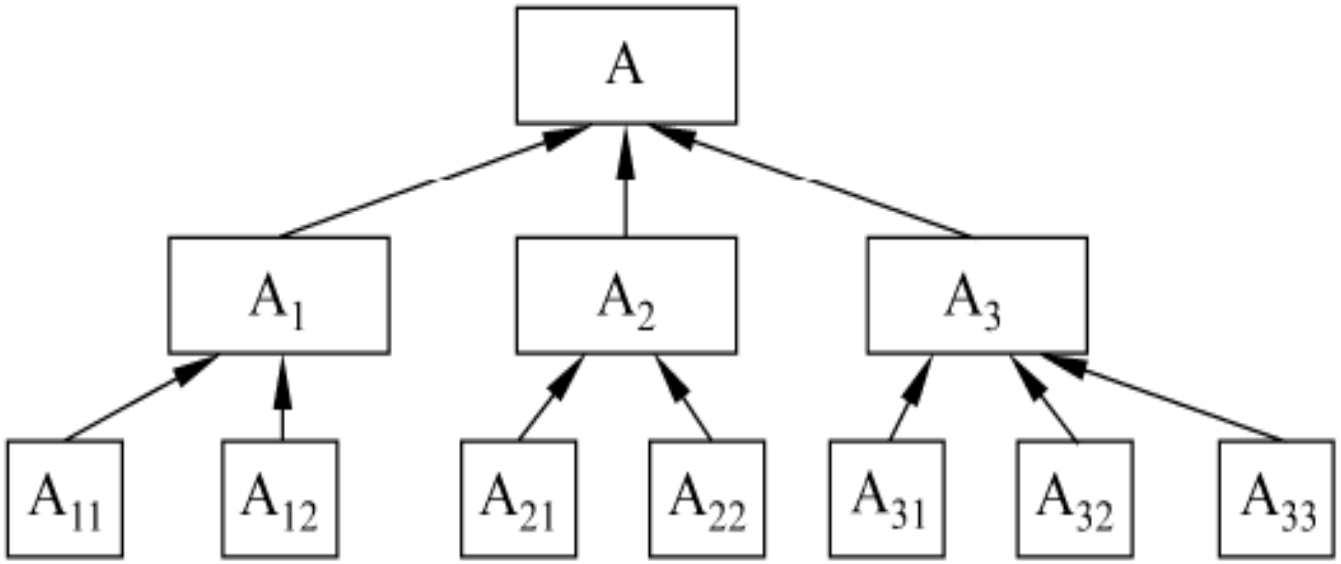
- 继承模型使用的图形表示：



第10讲 继承机制-继承的基本概念

❖C++的继承机制

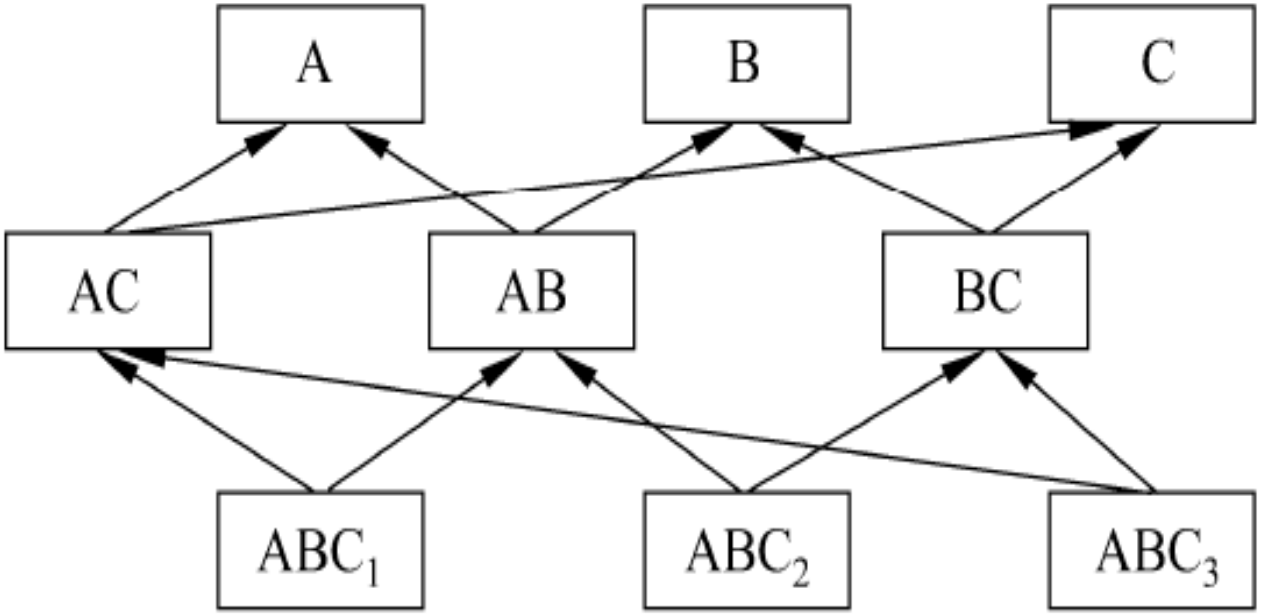
- 单继承(Single Inheritance)：一个派生类只从一个基类派生，这种继承关系所形成的层次是一个树形结构。



第10讲 继承机制-继承的基本概念

C++的继承机制

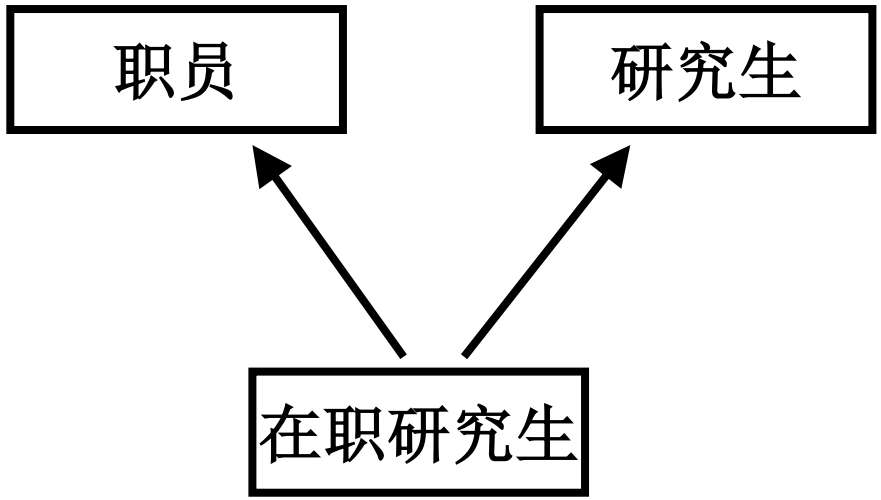
- 多重继承(Multiple Inheritance)：一个派生类拥有两个或更多的基类。



第10讲 继承机制-继承的基本概念

❖C++的继承机制

- 多重继承(Multiple Inheritance) 例



第10讲 继承机制-继承的基本概念

❖ 继承的作用

- 类的构造机制：通过扩充、组合现有的类来构造新的类。扩充是指形成现有类的特例—派生类；组合是指抽取出若干现有类的共性形成新的抽象层次—基类。
- 类型的构造机制：如果类B继承类A，则所有要求对象为A类型的地方也可以接受B类型对象。
- 软件重用：面向对象技术是迄今为止提高软件可重用性的最有效的途径。其特色的继承机制为软件重用提供支持。
- 提高程序的可靠性：派生类通常都基于那些设计完善并经过严格测试的基类。从而使程序设计工作建立在一个可靠的基础之上。

第10讲 继承机制-继承的基本概念

- 继承的语法

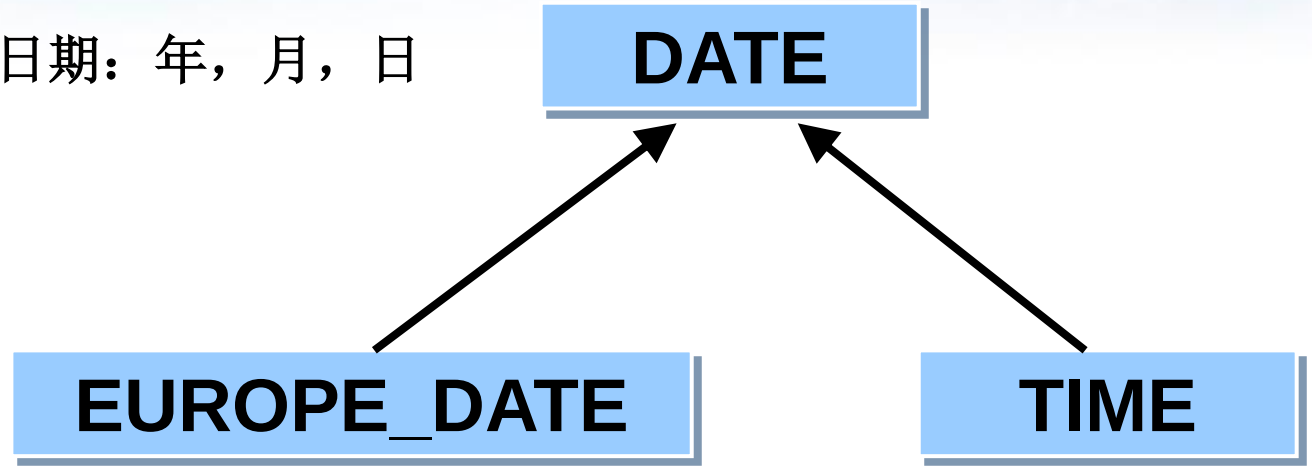
```
class 派生类名：继承访问控制 基类名
{  //派生类新增加的成员
  public:
    //公有成员列表
  protected:
    //受保护成员说明列表
  private:
    //私有成员说明列表
};
```

①public
②protected
③private(缺省时)

第10讲 继承机制-继承的基本概念

❖继承的语法-例

日期：年，月，日



以dd-mm-yy格式显示的日期

带有日期信息的时间

年，月，日 时，分，秒

第10讲 继承机制-继承的基本概念

❖ 继承的语法-例

```
//日期类date.h
class DATE{
public:
  DATE(int yy=0,int mm=0,int dd=0); //构造函数
  void set_date(int yy,int mm,int dd);//设置日期
  void get_date(int& yy,int& mm,int& dd);
  //取日期到三个实际的参数中
  void print_date();
  //以ANSI格式打印(yy.mm.dd)
protected:
   int year,month,day; //年，月，日
};
```

第10讲 继承机制-继承的基本概念

❖ 继承的语法-例

```
//欧洲日期类europe_date.h
class EUROPE_DATE:public DATE{
public:
   void print_date();
   //以欧洲格式(dd-mm-yy)打印日期
};
```

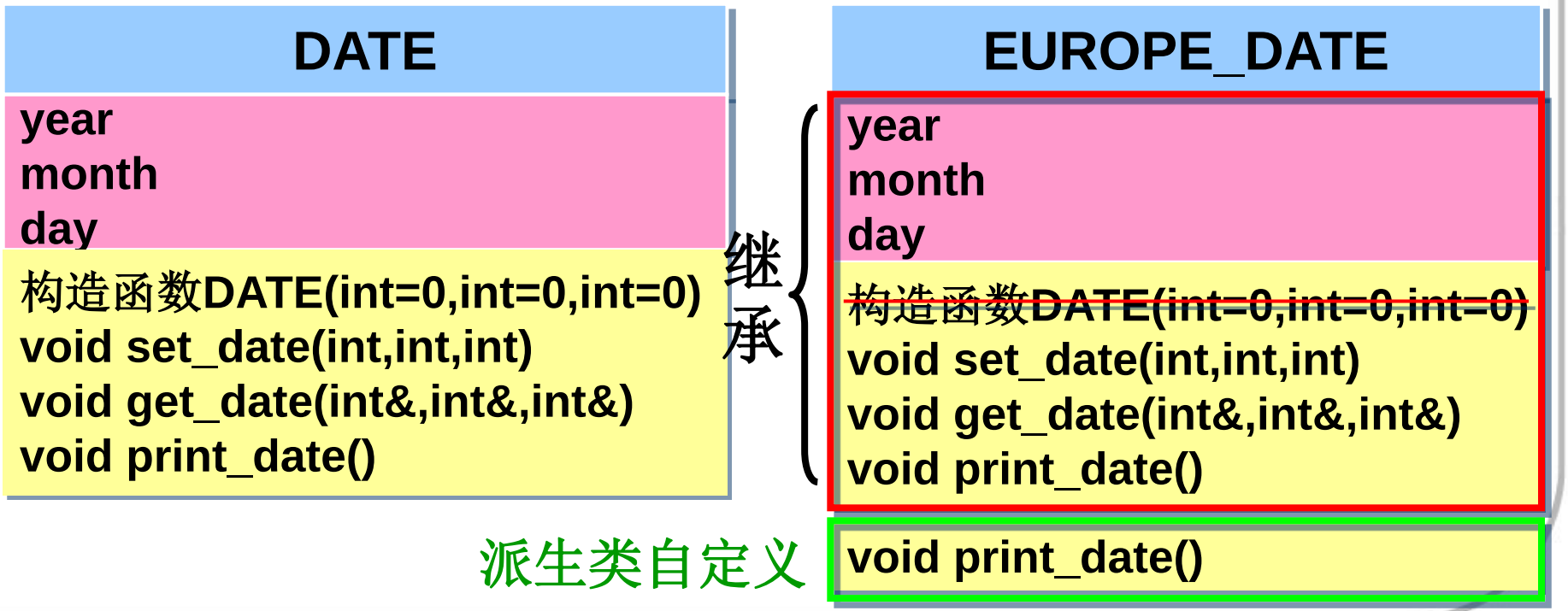
第10讲 继承机制-继承的基本概念

❖ 继承的语法-例

```
class TIME:public DATE{
public:
   //TIME(int hh,int mm,int ss); //构造函数
   void set_time(int hh,int mm,int ss); //设置时间
   void get_time(int&hh,int& mm,int& ss);
    //取时间到三个实际的参数中
   void print_time();
   //以hh:mm:ss格式打印时间
protected:
   //时，分，秒
   int hour,minute,second;
};
```

第10讲 继承机制-继承的基本概念

- 派生类的构成
  - 1.从基类继承过来的成员
  - 2. 声明派生类时增加的成员
  - 每一部分均分别包括数据成员和成员函数。



第10讲 继承机制-继承的基本概念

❖派生类的构成

DATE

year
month
day

构造函数DATE(int=0,int=0,int=0)
void set_date(int,int,int)
void get_date(int&,int&,int&)
void print_date()

继承

TIME

year
month
day

~~构造函数DATE(int=0,int=0,int=0)~~
void set_date(int,int,int)
void get_date(int&,int&,int&)
void print_date()

hour
minute
second

void set_time(int,int,int)
void get_time(int&,int&,int&)
void print_time()

派生类自定义

第10讲 继承机制-继承的基本概念

❖ 派生类的构成

实际上，不是把基类的成员和派生类自己增加的成员简单地加在一起就成为派生类。构造一个派生类包括以下3部分工作：

- (1)从基类接收成员：派生类把基类全部的成员(不包括构造函数和析构函数)接收过来。不能选择接收其中一部分成员，而舍弃另一部分成员。需要慎重选择基类，使冗余量最小。事实上，有些类是专门作为基类而设计的，在设计时充分考虑到派生类的要求。
- (2)调整从基类接收的成员：接收基类成员是程序人员不能选择的，但是程序人员可以对这些成员作某些调整。
- (3)在声明派生类时增加的成员：体现了派生类对基类功能的扩展。要根据需要仔细考虑应当增加哪些成员，精心设计。此外，在声明派生类时，一般还应当自己定义派生类的构造函数和析构函数，因为构造函数和析构函数是不能从基类继承的。

第10讲 继承机制-继承的基本概念

❖ 继承的成员访问控制规则

- 类成员的访问控制规则

成员访问控制	类自身	派生类	其他类/ 程序代码
public	可访问	可访问	可访问
protected	可访问	可访问	不可访问
private	可访问	不可访问	不可访问

第10讲 继承机制-继承的基本概念

❖ 继承的成员访问控制规则

- 类成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

1.类自身可访问类内所有成员

```
void A::set_A(int x){
   public_x=x;
   protected_x=x;
   private_x=x;
}
```

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 类成员的访问控制规则-举例

```
class B :public A{
public/protected/private:
   void set_B(int x);
};
```

2.派生类只可访问基类共有的成员 和 受保护成员
不可访问基类的私有成员

```
void B::set_B(int x){
   public_x=x;
   protected_x=x;
   private_x=x; //X
}
```

第10讲 继承机制-继承的基本概念

❖ 继承的成员访问控制规则

- 类成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

3. 类外只可访问类中的公有成员；不可访问受保护的，或私有成员

```
void main()
{   A obj;
cout<<obj.public_x<<endl;

cout<<obj.protected_x<<endl;
cout<<obj.private_x<<endl;
}
```

X

第10讲 继承机制-继承的基本概念

❖ 继承的成员访问控制规则

- 继承成员的访问控制规则

基类成员访问控制	继承访问控制	在派生类中的访问控制
public	public	public
protected		protected
private		不可访问
public	protected	protected
protected		protected
private		不可访问
public	private	private
protected		private
private		不可访问

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 继承成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

```
class B :public A{
public/protected/private:
   void set_B(int x);
};
```

1.公有继承：派生类继承基类的成员(私有成员除外)，所有成员维持原来的访问控制。

```
void B::set_B(int x){
```

```
void main()
{   B obj;
cout<<obj.public_x<<endl;
cout<<obj.protected_x<<endl;
cout<<obj.private_x<<endl;
}
```

X

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 继承成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

```
class B :protected A{
public/protected/private:
   void set_B(int x);
};
```

2.保护继承：派生类继承基类的成员(私有成员除外)，所有成员统一为受保护访问控制。

```
void B::set_B(int x){
```

```
void main()
{   B obj;
cout<<obj.public_x<<endl;
cout<<obj.protected_x<<endl;
cout<<obj.private_x<<endl;
}
```

X

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 继承成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

```
class C :public B{
public/protected/private:
   void set_C(int x);
};
```

2.保护继承：派生类继承基类的成员(私有成员除外)，所有成员统一为受保护访问控制。

```
void C::set_C(int x){
```

```
void main()
{   C obj;
cout<<obj.public_x<<endl;
cout<<obj.protected_x<<endl;
cout<<obj.private_x<<endl;
}
```

X

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 继承成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

```
class B :private A{
public/protected/private:
   void set_B(int x);
};
```

3. 私有继承：派生类继承基类的成员(私有成员除外)，所有成员统一为私有访问控制。

```
void B::set_B(int x){
```

```
void main()
{   B obj;
cout<<obj.public_x<<endl;
cout<<obj.protected_x<<endl;
cout<<obj.private_x<<endl;
}
```

X

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 继承成员的访问控制规则-举例

```
class A{
public:
   void set_A(int x);
   int public_x;
protected:
   int protected_x;
private:
   int private_x;
};
```

```
class C :public B{
public/protected/private:
   void set_C(int x);
};
```

3. 私有继承：派生类继承基类的成员(私有成员除外)，所有成员统一为私有访问控制。

```
void C::set_C(int x){
```

```
void main()
{   C obj;
cout<<obj.public_x<<endl;
cout<<obj.protected_x<<endl;
cout<<obj.private_x<<endl;
}
```

X

第10讲 继承机制-继承的基本概念

❖ 继承的成员访问控制规则

- 访问控制规则的总结

(1)基类的成员函数访问基类成员。

什么都可以访问。

(2)派生类的成员函数访问基类的成员。

结合继承访问控制；私有成员不可访问(不能继承)。

公有继承：

公有成员-可访问（在派生类中依然为公有成员）

受保护成员-可访问（在派生类中依然为受保护成员）

受保护继承：

公有成员-可访问（在派生类中变为受保护成员）

受保护成员-可访问（在派生类中依然为受保护成员）

私有继承：

公有成员-可访问（在派生类中变为私有成员）

受保护成员-可访问（在派生类中依然为私有成员）

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 访问控制规则的总结

(3)在基类/派生类外访问基类的成员。

只可访问公有成员。

(4)基类的成员函数访问派生类的成员。

什么都不能访问。

(5)派生类的成员函数访问派生类自己增加的成员。

什么都可以访问。

(6)在派生类外访问派生类的成员。

按继承后的访问控制访问。只能访问继承后仍然为公有访问控制的成员。

第10讲 继承机制-继承的基本概念

❖继承的成员访问控制规则

```
class A //基类
{
public:
int i;
protected:
void f2( );
int j;
private:
int k;
};
```

```
class B: public A
{
public:
void f3( );
protected:
void f4( );
private:
int m;
};
```

```
class C: protected B
{
public:
void f5( );
private:
int n;
};
```

访问控制	i	f2	j	k	f3	f4	m	f5	n
A	公有	保护	保护	私有					
B	公有	保护	保护	不可访问	公有	保护	私有		
C	保护	保护	保护	不可访问	保护	保护	不可访问	公有	私有

第10讲 继承机制-继承的基本概念

- 继承的成员访问控制规则
  - 如果在多级派生时都采用公用继承方式，那么直到最后一级派生类都能访问基类的公用成员和保护成员。
  - 如果采用私有继承方式，经过若干次派生之后，基类的所有的成员已经变成不可访问的了。
  - 如果采用保护继承方式，在派生类外是无法访问派生类中的任何成员的。而且经过多次派生后，人们很难清楚地记住哪些成员可以访问，哪些成员不能访问，很容易出错。
  - 因此，在实际中，常用的是公用继承。

第10讲 继承机制-继承的基本概念

❖派生类对象的存储组织

- 每一个对象实例都占有一块存储空间，其中仅存放该对象实例的非静态数据成员，而静态数据成员以及成员函数则整个类才存放一份。
- 例：

```
class A {
public:
    int pub;
    static int pubs;
protected:
    float pro;
    static int pros;
private:
    char pri;
    static int pris;
};
```

```
int mian()
{ A  a1, a2;
  ……
```

类A的静态数据

a1

a2

第10讲 继承机制-继承的基本概念

❖派生类对象的存储组织

- 派生类的对象实例：派生类中定义的非静态数据成员+从基类中继承下来的非静态数据成员。因而在派生类中存放的这些从基类继承下来的数据存储又称派生类的子对象。
- 例：

```
class B: public A
{
public:
……
protected:
……
private:
……
};
```

```
int mian()
{ B  b1, b2;
  ……
```

类B的静态数据

……

b1

pub:4字节

pro:4字节

B类定义的非静态数据

b2

pub:4字节

pro:4字节

B类定义的非静态数据

课堂练习

- 派生类的对象不可以直接访问的基类成员为 ___。
- A）公有继承的基类公有成员
- B）公有继承的基类保护成员
- C）公有继承的基类私有成员
- D）保护继承的基类公有成员

程序5-1：CPoint类及其派

```
//基类Point类的声明
class Point
{private:
     float X,Y;
  public:
     void InitP(float xx=0, float
      {  X=xx;
         Y=yy;
      }
     void Move(float xOff, float y
      {  X+=xOff;
         Y+=yOff;
      }
     float GetX() {return X;}
     float GetY() {return Y;}
};
```

```
#include<iostream.h>
#include<math.h>
int main()
{  CRect rect;
   rect.InitR(2,3,20,10);
        //通过派生类对象访问基类公有成员
   rect.Move(3,2);
   cout <<rect.GetX()<<','
        <<rect.GetY()<<','
        <<rect.GetH()<<','
        <<rect.GetW()<<endl;
   return 0;
}
```

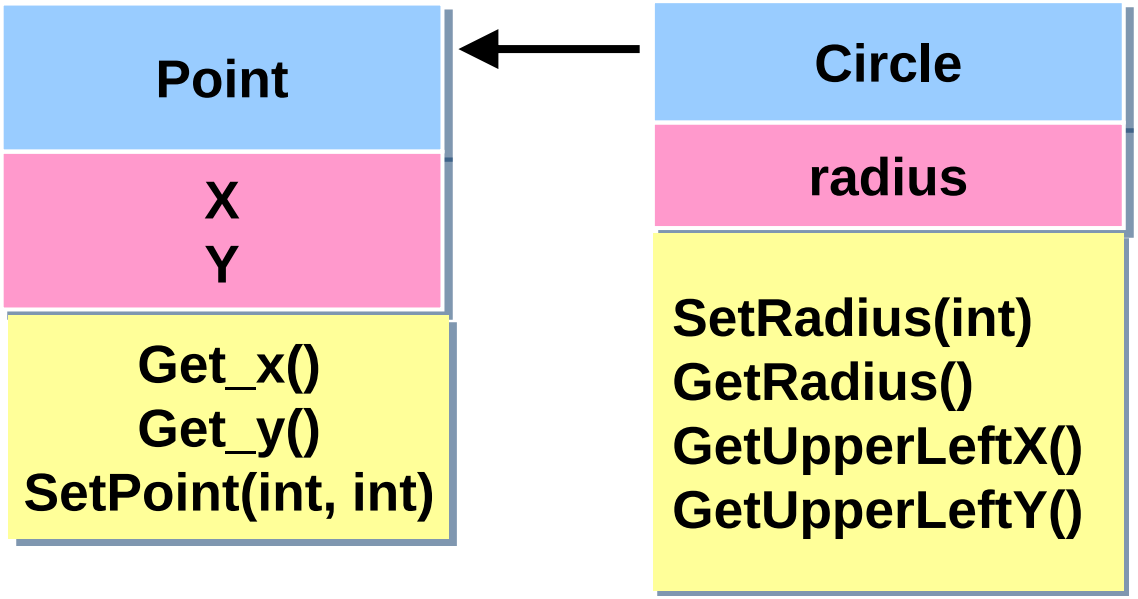
```
//派生类声明
class CRect: public Point
{ private:          //新增私有数据成员
     float W,H;
  public:          //新增公有成员函数
    void InitR(float x, float y, float w, float h)
    {  InitP(x,y); //调用基类公有成员函数
       W=w;H=h;
     }
     float GetH() {return H;}
     float GetW() {return W;}
};
```

运行结果：

```
5, 5, 10, 20
Press any key to continue
```

课堂练习

- 设计如下一个继承类，给定圆心的坐标值和圆的半径（如圆心（100,150），半径100），计算其外接正方形左上角的X,Y坐标并显示。



```
#include <iostream>
Class Point {                          //定义基类，表示point
Private:                  //私有成员x，y，在派生类中不可见
    int x;
    int y;
Public:
    void SetPoint(int a, int b) {x=a; y =b;} //设置坐标
    int getX() {return x;}
    int getY() {return y;}
}

Class Circle: public Point {     //定义派生类，表示圆
Private:
    int radius;                          //圆的半径
Public:
    void SetRadius (int r) {radius = r;}
    int getRadius() {return radius;}
    int GetUpperLeftX() {return getX()-radius;}
    GetUpperLeftY() {return getY()+radius;}
}
```

```
void main()
{
Circle c;
c.SetPoint (100, 150);//公有派生类的对象可以直接访问
   基类的公有成员
c.SetRadius (100);
Cout<<“UpperLeft X=”<<c.GetUpperLeftX()<<“,
   UpperLeft Y=”<<c. GetUpperLeftY()<<endl;
}
```

若将继承方式改为protected呢？

第10讲 继承机制

❖ 继承与构造、析构

- **构造函数可缺省：**用户在声明类时可以不定义构造函数，系统会自动设置一个默认的构造函数，在定义类对象时会自动调用这个默认的构造函数。
- **缺省的构造函数：**实际上是一个空函数，不执行任何操作。如果需要对类中的数据成员初始化，应自己定义构造函数。
- **构造函数的作用：**对数据成员初始化。派生类的构造函数需要同时考虑派生类所增加的数据成员和基类的数据成员初始化。从而在执行派生类的构造函数时，使派生类的数据成员和基类的数据成员同时都被初始化。
- **C++的解决思路：**在执行派生类的构造函数时，调用基类的构造函数。

第10讲 继承机制

- 继承与构造、析构
  - 1. 简单派生类的构造函数
  - 简单的派生类：只有一个基类，而且只有一级派生（只有直接派生类，没有间接派生类），在派生类的数据成员中不包含基类的对象（即子对象）。任何派生类都包含基类的成员。

```
#include <string>
using namespace std;
//基类Student
 class Student {
public:
    Student(int,string,char); //构造
    ~Student( ); //析构
protected: //保护部分
    int num; //学号
    string name; //姓名
    char sex ; //性别     };
```

```
#include “Student.h”
//构造
Student ::Student(int n,
                  string  nam,char s)
{ num=n;
  name=nam;
  sex=s; }

//析构
Student ::~Student( )
{ }
```

第10讲 继承机制

- 继承与构造、析构
  - 1. 简单派生类的构造函数

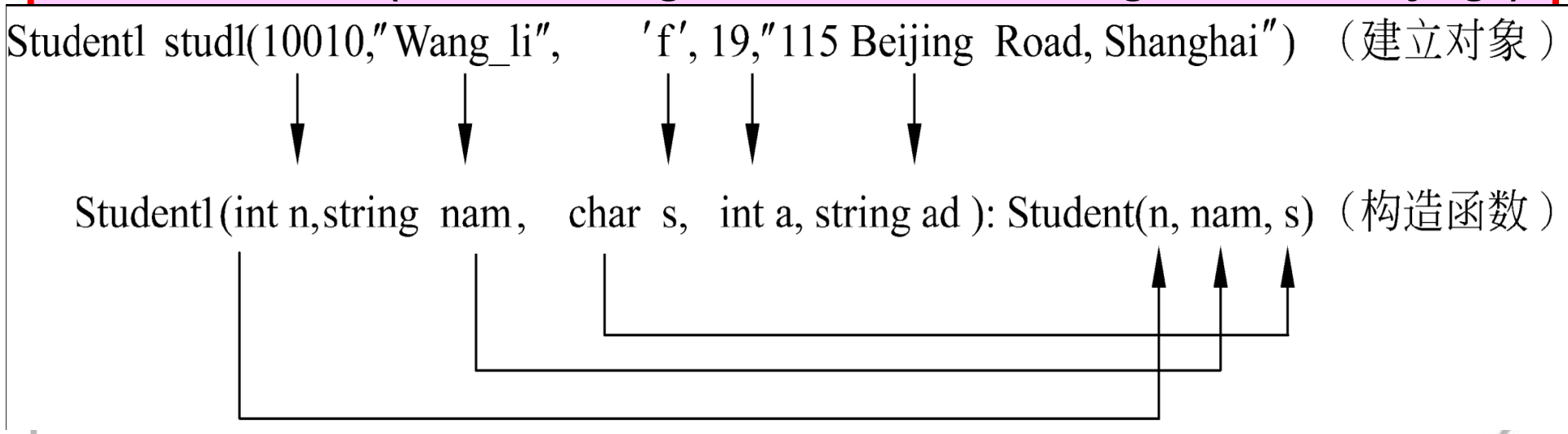
```
#include “Student.h”
//派生类Student1
 class Student1: public Student{
public:
   //派生类构造
   Student1(int,string,char,int,string);
   void show( );//显示
   ~Student1( ); //派生类析构
private: //派生类私有部分
   int age; //年龄
   string addr; //地址
};
```

```
#include “Student1.h”
//构造
Student1::Student1(int n,
string nam,char s,int a,string ad)
                    :Student(n,nam,s)
{//函数体中只对新增的数据成员初
 //始化
    age=a; addr=ad;     }
//显示
void Student1::show( )
{cout<<"num:"<<num<<endl;
cout<<"name:"<<name<<endl;
cout<<"sex:"<<sex<<endl;
cout<<"age:"<<age<<endl;
cout<<"address:"<<addr<<endl;
cout<<endl;                 }
//析构
Student1::~Student1( ){ }
```

第10讲 继承机制

- 继承与构造、析构
  - 1. 简单派生类的构造函数

```
//测试主函数
#include “Student1.h”
int main( )
{ Student1 stud1(10010,”Wang-li”,’f’,19,”115 Beijing Road,Shanghai”);
  Student1 stud2(10011,”Zhang-fu”,’m’,21,”213 Shanghai Road,Beijing”);
```



2点说明：

①(总参数列表)为派生类构造函数定义的形参，需要参数类型说明；

②(参数表)为调用基类构造函数传递的实参，不需要说明参数类型；

```
派生类构造函数名(总参数列表) :基类构造函数名(参数列表)
{
        派生类中新增数据成员初始化语句
}
```

★调用基类构造函数时的实参：从派生类构造函数的总参数表中得到的；也可以不从派生类构造函数的总参数表中传递过来，而直接使用常量或全局变量。

例如，派生类构造函数首行可以写成以下形式：

```
Student1(string nam，char s，int a，string ad)
                                    : Student(10010，nam，s)
```

第10讲 继承机制

❖ 继承与构造、析构

- 1. 简单派生类的构造函数
- 可以对age和addr也用初始化表处理，将构造函数改写为:

```
Student1(int n, string nam,char s,int a, string ad)
                    : Student(n,nam,s),age(a),addr(ad){  }
```

这样函数体为空，更显得简单和方便。

- 构造顺序：在建立一个对象时顺序是:①派生类构造函数先调用基类构造函数；②再执行派生类构造函数本身(即派生类构造函数的函数体)。

例，先初始化num，name，sex，然后再初始化age和addr。

- 析构相反：在派生类对象释放时，先执行派生类析构函数~Student1( )，再执行基类析构函数~Student( )。

第10讲 继承机制

❖ 继承与构造、析构

- **2. 有子对象的派生类的构造函数**
- 子对象：类的数据成员中包含类的对象。即在类中内嵌对象。

```
//基类Student
#include <string>
using namespace std;
 class Student {
public:
    Student(int,string); //构造
    void display(); //显示
    ~Student( ); //析构
protected: //保护部分
    int num; //学号
    string name; //姓名
};
```

```
#include “Student.h”
//构造
Student ::Student(int n,string  nam)
{ num=n;
  name=nam;
}
//显示
Student::display()
{cout<<“num:”<<num<<endl;
  cout<<“name:”<<name<<endl;
}
//析构
Student ::~Student( )
{ }
```

第10讲 继承机制

- 继承与构造、析构
  - 2. 有子对象的派生类的构

```
#include <string>
using namespace std;
//派生类Student1
 class Student1: public Student{
public:
   //派生类构造
Student1(int,string,int,string,int,string);
   void show( );//显示
   void show_monitor();//显示班长
~Student1( ); //派生类析构
private: //派生类私有部分
   Student Monitor; //班长
   int age; //年龄
   string addr; //地址
};
```

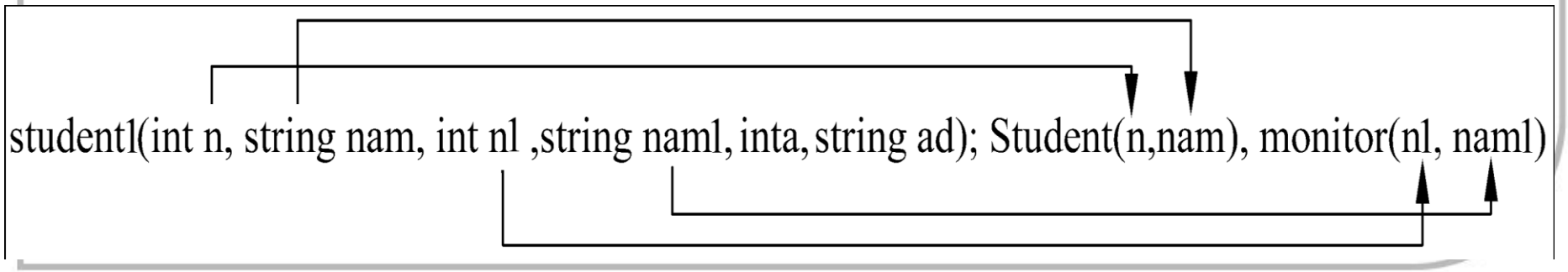
```
#include “Student1.h”
#include <iostream>
using namespace std;
//构造
Student1::Student1(int n,
string nam,int n1,string nam1,int
a,string
ad) :Student(n,nam),Monitor(n1,nam1)
{//函数体中只对新增的数据成员初
 //始化
    age=a; addr=ad;     }
//显示
void Student1::show( )
{display();
cout<<"age:"<<age<<endl;
cout<<"address:"<<addr<<endl;
}
//显示班长
void Student1::show_monitor()
{ monitor.display();     }
//析构
Student1::~Student1( ){ }
```

第10讲 继承机制

❖ 继承与构造、析构

▪ 2. 有子对象的派生类的构造函数

```
//测试主函数
#include “Student1.h”
int main( )
{ Student1 stud1(10010, ”Wang-li”,10001,”Li-sun”,19, ”115 Beijing
                                                    Road,Shanghai”);
  stud1.show( ); //输出第一个学生的数据
  stud2.show_monitor( ); //输出这个学生的班长（子对象）信息
  return 0;
}
```



第10讲 继承机制

❖ 继承与构造、析构

▪ 2. 有子对象的派生类的构造函数

```
派生类构造函数名(总参数列表) :基类构造函数名(参数列表)
                                ,子对象名(参数列表)
{
        派生类中新增数据成员初始化语句
}
```

★总参数列表中的参数：应当包括基类构造函数和子对象的参数列表中的参数。基类构造函数和子对象的次序可以是任意的。例，**Student1Student1(int n, string nam,int n1, string nam, int a, string ad) : monitor(n1,nam1), Student(n,nam)**

第10讲 继承机制

❖ 继承与构造、析构

- **2. 有子对象的派生类的构造函数**
- 构造顺序：①调用基类构造函数，对基类数据成员初始化；②调用子对象构造函数，对子对象数据成员初始化；③再执行派生类构造函数本身，对派生类数据成员初始化。

- 编译系统根据参数名(而不是根据参数的顺序)来确立传递关系的。习惯上，先写基类构造函数。若有多个子对象，派生类构造函数的写法依此类推。

第10讲 继承机制

- 继承与构造、析构
  - 3. 多层派生时的构造函数
  - 多层派生：一个类不仅可以派生出一个派生类，派生类还可以继续派生，形成派生的层次结构。

```
//基类Student
 class Student {
public:
    Student(int,string); //构造
    void display(); //显示
    ~Student( ); //析构
protected: //保护部分
    int num; //学号
    string name; //姓名
};
```

```
//构造
Student ::Student(int n,string  nam)
{ num=n;
  name=nam;  }
//显示
Student::display()
{cout<<“num:”<<num<<endl;
  cout<<“name:“<<name<<endl;    }
//析构
Student ::~Student( )
{ }
```

第10讲 继承机制

- 继承与构造、析构
  - 3. 多层派生时的构造函

```
//派生类Student1
 class Student1: public Student{
public:
  //派生类构造
Student1(int,string,int);
  void show( );//显示
  void show_monitor();//显示班长
~Student1( ); //派生类析构
private: //派生类私有部分
  int age; //年龄
};
```

```
//构造
Student1::Student1(int n,
string nam,int a) :Student(n,nam)
{//函数体中只对新增的数据成员初
 //始化
   age=a;  }
//显示
void Student1::show( )
{display();
cout<<"age:"<<age<<endl;
}
//显示班长
void Student1::show_monitor()
{ monitor.display();     }
//析构
Student1::~Student1( ){ }
```

第10讲 继承机制

- 继承与构造、析构
  - 3. 多层派生时的构造函数

```
//派生类Student2
 class Student2: public Student1{
public:
  //派生类构造
Student2(int,string,int,int);
  void show_all( );//显示
~Student2( ); //派生类析构
private: //派生类私有部分
  int score;
};
```

```
//构造
Student2::Student2(int n,
string nam,int a,int s) :
                    Student1(n,nam,a)
{//函数体中只对新增的数据成员初
 //始化
   score=s;    }
//显示
void Student2::show_all( )
{show();
cout<<"score:"<<score<<endl;
}
//析构
Student2::~Student2( ){ }
```

第10讲 继承机制

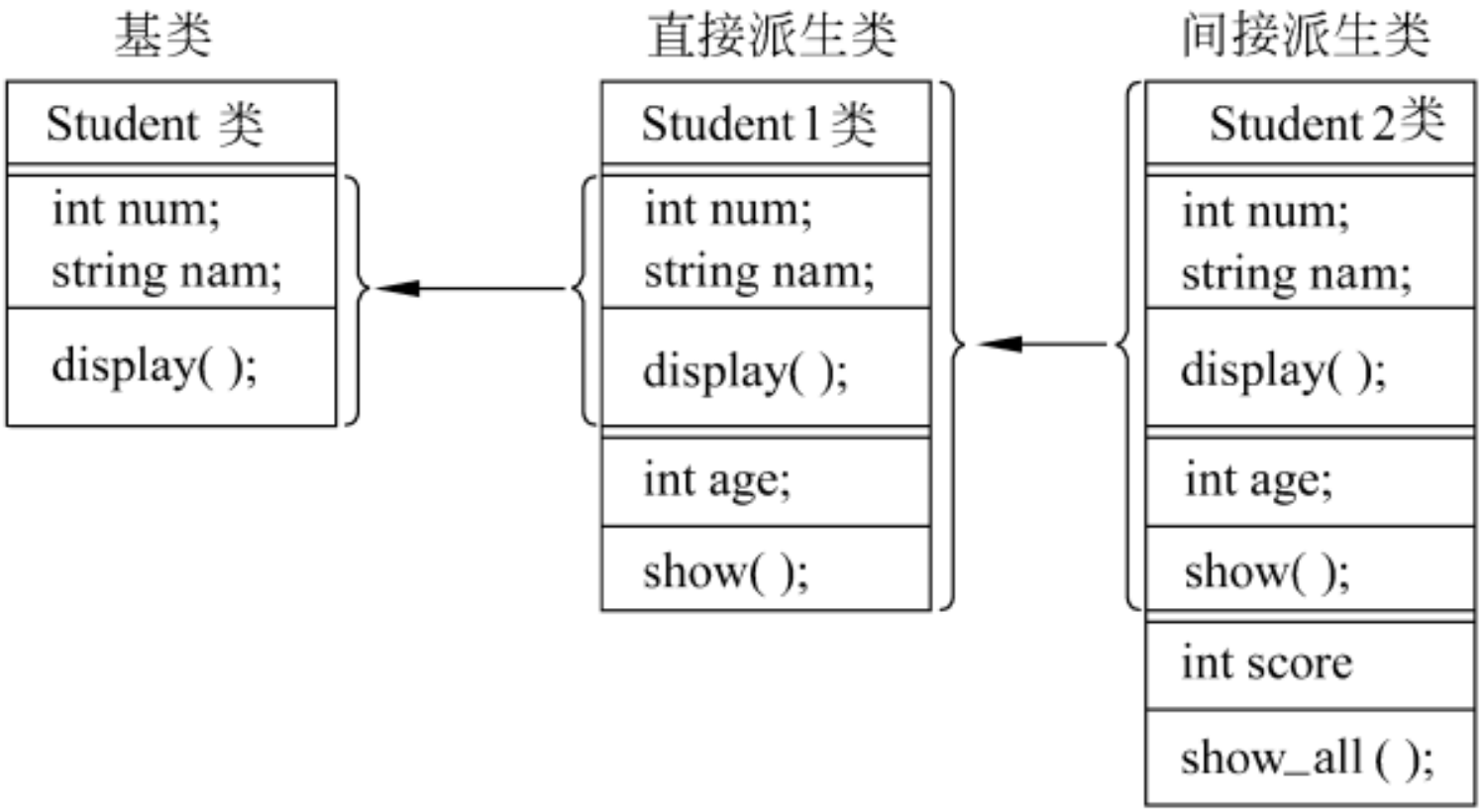
- 继承与构造、析构
  - 3. 多层派生时的构造函数

```
//测试主函数
#include “Student.h”
#include “Student1.h”
#include “student2.h”
int main( )
{ Student2 stud(10010,"Li",17,89);
  stud.show_all( ); //输出学生的全部数据
   return 0;
}
```

第10讲 继承机制

❖ 继承与构造、析构

▪ 3. 多层派生时的构造函数



第10讲 继承机制

❖ 继承与构造、析构

▪ 3. 多层派生时的构造函数

基类的构造函数:

Student(int n, string nam)

派生类**Student1**的构造函数:

Student1(int n, string nam,int a):Student(n,nam)

派生类**Student2**的构造函数:

Student2(int n, string nam,int a,ints):Student1(n,nam,a)

在声明**Student2**类对象时：调用**Student2**构造函数；
在执行**Student2**构造函数时，先调用**Student1**构造函数；
在执行**Student1**构造函数时，先调用基类**Student**构造函数。

初始化的顺序是:
①先初始化基类的数据成员**num**和**name**。
②再初始化**Student1**的数据成员**age**。
③最后再初始化**Student2**的数据成员**score**。

第10讲 继承机制

❖ 继承与构造、析构

▪ 4. 派生类构造函数的特殊形式

(1) 当不需要对派生类新增的成员进行任何初始化操作时：

派生类构造函数的函数体可以为空，即构造函数是空函数。

例，派生类Student1构造函数可以改写为

```
Student1(int n,string nam,int n1,string nam1)
                    :Student(n,nam),monitor(n1,nam1)
{    }
```

此派生类构造函数的作用：只是为了将参数传递给基类构造函数和子对象，并在执行派生类构造函数时调用基类构造函数和子对象构造函数。在实际工作中常见这种用法。

第10讲 继承机制

❖ 继承与构造、析构

▪ 4. 派生类构造函数的特殊形式

(2)基类中没有定义构造函数，或定义了没有参数的构造函数：在定义派生类构造函数时可不写基类构造函数。因为此时派生类构造函数没有向基类构造函数传递参数的任务。

调用派生类构造函数时系统会自动首先调用基类的默认构造函数。

第10讲 继承机制

❖ 继承与构造、析构

▪ 4. 派生类构造函数的特殊形式

(3)基类和子对象类型中都没有定义带参数的构造函数：

当不需对派生类自己的数据成员初始化，则可以不必显式地定义派生类构造函数。因为此时派生类构造函数既没有向基类构造函数和子对象构造函数传递参数的任务，也没有对派生类数据成员初始化的任务。

在建立派生类对象时，系统会自动调用系统提供的派生类的默认构造函数，并在执行派生类默认构造函数的过程中，调用基类的默认构造函数和子对象类型默认构造函数。

第10讲 继承机制

❖ 继承与构造、析构

▪ 4. 派生类构造函数的特殊形式

(4)基类或子对象类型中定义了带参数的构造函数：就必须显式地定义派生类构造函数，并在派生类构造函数中写出基类或子对象类型的构造函数及其参数表。

(5)基类中既定义无参的构造函数，又定义了有参的构造函数(构造函数重载)：在定义派生类构造函数时，既可以包含基类构造函数及其参数，也可以不包含基类构造函数。

在调用派生类构造函数时，根据构造函数的内容决定调用基类的有参的构造函数还是无参的构造函数。编程可以根据派生类的需要决定采用哪一种方式。

第10讲 继承机制

❖ 继承与构造、析构

- 派生类的析构函数： 在派生时，派生类是不能继承基类的析构函数的，也需要通过派生类的析构函数去调用基类的析构函数。
- 在派生类中可以根据需要定义自己的析构函数，用来对派生类中所增加的成员进行清理工作。基类的清理工作仍然由基类的析构函数负责。
- 在执行派生类的析构函数时，系统会自动调用基类的析构函数和子对象的析构函数，对基类和子对象进行清理。
- 调用的顺序与构造函数正好相反：执行派生类自己的析构函数，对派生类新增加的成员进行清理➔调用子对象的析构函数，对子对象进行清理➔最后调用基类的析构函数，对基类进行清理。

对于派生类构造函数的初始化列表，其不能包含（ ）

A）基类的构造函数;

B）派生类中内嵌对象的初始化;

C）基类中内嵌对象的初始化;

D）派生类中一般数据成员的初始化.

课堂练习

```
#include <iostream>
Class Point {                          //定义基类Point
Protected:
    int x,y;

Public:
    Point (int a=0, int b=0){
    x=a; y=b;
    cout<<“Point constructor: ”<<x<<“, ”<<y<<endl;
}

    ~Point() {cout<<“Point destructor: ”<<x<<
    <<“, ”<<y<<endl;}
}
```

```
Class Circle: public Point {     //定义Point的派生类
Protected:
    int radius;

Public:
    Circle (int a=0, int b=0, int r=0): Point (a, b){
    radius =r;
    cout<<“Circle constructor: ”<<
                x<<“, ”<<y<<“, ”<<radius<<endl;
}

    ~Circle() {cout<<“Circle destructor: ”<<
                        x<<“, ”<<y<<“, ”<<radius<<endl;

}
```

```
Class Cylinder: public Circle {//定义Circle的派生类
Protected:
    int height;

Public:
    Cylinder (int a=0, int b=0, int r=0, int h=0):
    Circle (a, b, r){
    height =h;
    cout<<“Cylinder constructor: ”<<

        x<<“, ”<<y<<“, ”<<radius<<“, ”<<height<<endl;
}

    ~ Cylinder() {cout<<“Cylinder destructor: ”<<

        x<<“, ”<<y<<“, ”<<radius<<“, ”<<height<<endl;

}
```

```
Void main ()
{
   Cylinder cylinder(400, 300, 200, 100);
}
```

程序运行结果：

Point constructor: 400, 300
Circle constructor: 400, 300, 200
Cylinder constructor: 400, 300, 200, 100
Cylinder destructor: 400, 300, 200, 100
Circle destructor: 400, 300, 200
Point destructor: 400, 300

```
//-----------------------------
#include<iostream>
using namespace std;
//-----------------------------
class A{
  int x;
public:
  A(int xx):x(xx){ cout<<"class A "<<x<<'\n'; }
}; //-------------------------
class B{
A a1;
public:
  B(int x):a1(x){ cout<<"class B\n"; }
}; //-------------------------
```

```
class C : public B{
A a2;
public:
  C(int x):B(x),a2(x){ cout<<"class C\n”; }
}; //--------------------------
class D : public C{
public:
 D(int x):C(x){ cout<<"class D\n"; }
}; //--------------------------
int main(){
  D d(10);
} //--------------------------
```

c	l	a	s	s		A		1	0
c	l	a	s	s		B			
c	l	a	s	s		A		1	0
c	l	a	s	s		C			
c	l	a	s	s		D			

练习

- 为上节课的类**Point**和**Circle**写出构造函数和析构函数（初始值：圆心（100,150），半径100）

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承(multiple inheritance)：一个派生类有两个或多个基类，派生类从两个或多个基类中继承所需的属性。C++允许一个派生类同时继承多个基类。这种行为称为多重继承。
- 例，已经定义类A，类B，类C，则

```
class D:public A,private B,protected C
{
   //类D新增加的成员
};
```

★ D是多重继承的派生类，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，确定了各基类的成员在派生类中的访问权限。

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承派生类的构造函数：与单继承时的构造函数形式基本相同，只是在初始表中包含多个基类构造函数。

派生类构造函数名(总参数列表):基类1构造函数(参数列表),基类2构造函数(参数列表),基类3构造函数(参数列表)

{

// 派生类中新增数成员据成员初始化语句

}

★各基类的排列顺序任意。派生类构造函数的执行顺序同样为:先调用基类的构造函数，再执行派生类构造函数的函数体。调用基类构造函数的顺序是按照声明派生类时基类出现的顺序。

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承例：教师类Teacher,学生类Student,研究生类Graduate（以多重继承方式得到）

```
//基类Teacher
 class Teacher {
public:
  //构造
  Teacher(string,int,string);
   void display( );//显示
protected: //保护部分
  string name;//姓名
  int age;//年龄
  string title;//职称
};
```

```
//构造
Teacher::Teacher(string nam,int a,
                                    string t)
{ name=nam;
   age=a;
   title=t;
}
//显示
void Teacher::display( )
{cout<<"name:"<<name<<endl;
cout<<"age:"<<age<<endl;
cout<<"title:"<<title<<endl;
  }
```

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承例：教师类Teacher,学生类Student,研究生类Graduate（以多重继承方式得到）

```
//基类Student
 class Student {
public:
  //构造
  Student(string,int,string);
   void display1( );//显示
protected: //保护部分
  string name1;//姓名
  int age1;//年龄
  int  score ;//成绩
};
```

```
//构造
Student::Student(string nam,int a, int s)
{ name1=nam;
   age1=a;
   score=s;
}
//显示
void Student::display1( )
{cout<<"name:"<<name1<<endl;
cout<<"age:"<<age1<<endl;
cout<<"score:"<<score<<endl;
  }
```

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承例：教师类Teacher,学生类Student,研究生类Graduate（以多重继承方式得到）

```
//派生类Graduat
 class Graduate
public:
  //构造
 Graduate(string
  void show( );//
private: //私有部
  float wage;//工
};
```

```
//构造
Graduate::Graduate(string nam,int a,string t,int s,float
w): Teacher(nam,a,t),Student(nam,a,s)
{ wage=w;
}
//显示
void Graduate::show( )
{  display();//或者display1();
   cout<<"score:"<<score<<endl;
   cout<<"wage:"<<wage<<endl;
 }
//也可以全部重写输出
```

第10讲 继承机制

- 多重继承和重复继承
  - 多重继承例：教师类Teacher,学生类Student,研究生类Graduate（以多重继承方式得到）

```
//测试主函数
#include “Teacher.h”
#include “Student.h”
#include “Graduate.h”
int main( )
{  Graduate grad1("Wang-li",24,"assistant",89.5,1234.5);
   grad1.show( ); //输出研究生的全部数据
   return 0;
}
```

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承例：教师类Teacher,学生类Student,研究生类Graduate（以多重继承方式得到）
- 设计存在的问题：

Teacher: name,age,title

Student: name1,age1,score

Graduate:name,age,title,name1,age1,score,wage

- 一个对象会有两套数据，不是一个自然的表达。
- 解决方法：将Student中的数据改为name,age,score。此时，Graduate中数据为Teacher::name,Teacher::age,title,Student::name,Student::age,score,wage。name,age需要限定作用域来避免有歧义的访问。

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承例：教师类Teacher,学生类Student,研究生类Graduate（以多重继承方式得到）

- 通过这个程序还可以发现一个问题：在多重继承时，从不同的基类中会继承一些重复的数据。如果有多个基类，问题会更突出。在设计派生类时要细致考虑其数据成员，尽量减少数据冗余。

第10讲 继承机制

❖ 多重继承和重复继承

- 多重继承的二义性问题：多重继承增加了程序的复杂度，使程序的编写和维护变困难，容易出错。最常见的问题就是继承的成员同名而产生的二义性问题。

```
//基类Student
 class Student {
public:
  //构造
  Student(string,int,string);
   void display( );//显示
protected: //保护部分
  string name;//姓名
  int age;//年龄
  int  score ;//成绩
};
```

```
//构造
Student::Student(string nam,int a, int s)
{ name=nam;
   age=a;
   score=s;
}
//显示
void Student::display( )
{cout<<"name:"<<name<<endl;
cout<<"age:"<<age<<endl;
cout<<"score:"<<score<<endl;
  }
```

第10讲 继承机制

❖ 多重继承和重复继承

- **(1)基类有多个同名成员：**

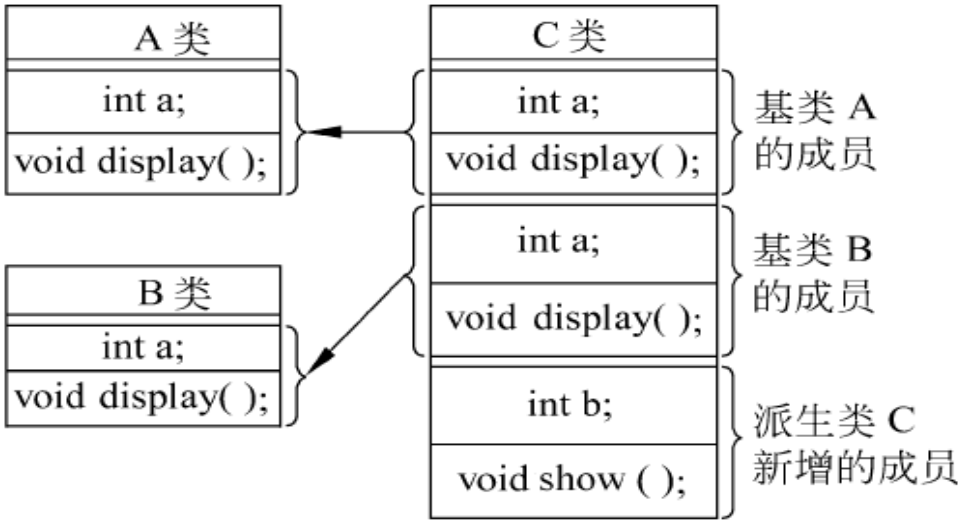
Teacher：数据成员：name,age

成员函数：void display();

Student：数据成员：name,age

成员函数：void display();

或者：



第10讲 继承机制

- 多重继承和重复继承
  - (1)基类有多个同名成员：

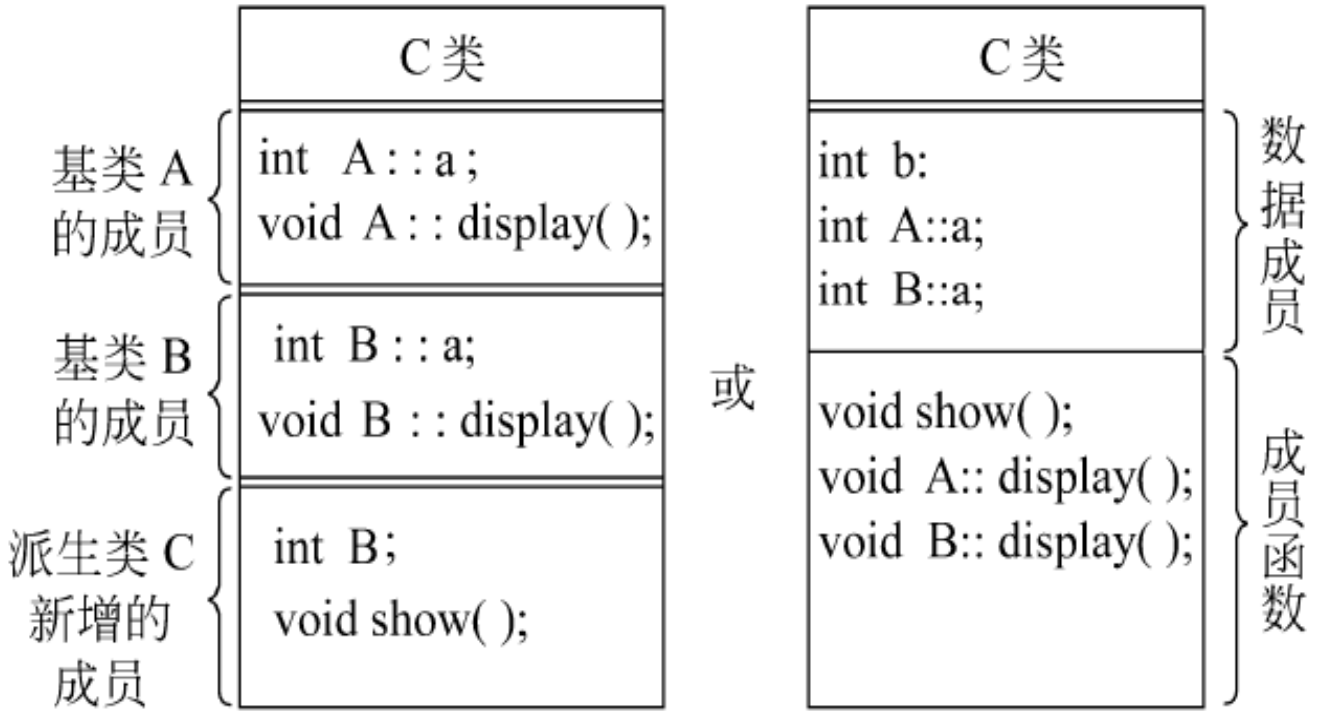
```
int main()
{
C c1;
c1.a=3;//X,二义性错误
c1.display();//X，二义性错误
c1.A::a=3;
c1.A::display();
c1.B::a=3;
c1.B::display();
} //在类中使用可以不用对象名
```

```
class A {
public:
  int a;
  void display( );
};
class B {
public:
  int a;
  void display( );
};
class C :public A,public B{
public :
  int b;
  void show();
};
```

第10讲 继承机制

❖ 多重继承和重复继承

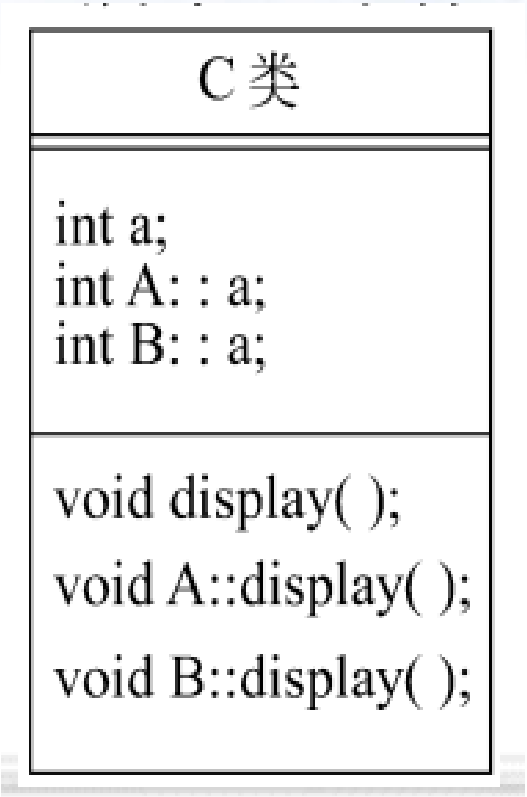
- (1)基类有多个同名成员：



第10讲 继承机制

- 多重继承和重复继承
  - (2)基类和派生类都有同名成员：

```
class A {
public:
  int a;
  void display( );   };
class B {
public:
  int a;
  void display( );
};
class C :public A,public B{
public :
  int a;
  void display();
};
```



第10讲 继承机制

❖ 多重继承和重复继承

▪ (2)基类和派生类都有同名成员：

```
int main()
{
C c1;
c1.a=3;//V，使用C类数据
c1.display();//V，使用C类数据
c1.A::a=3;
c1.A::display();
c1.B::a=3;
c1.B::display();
} //在类中使用可以不用对象名
```

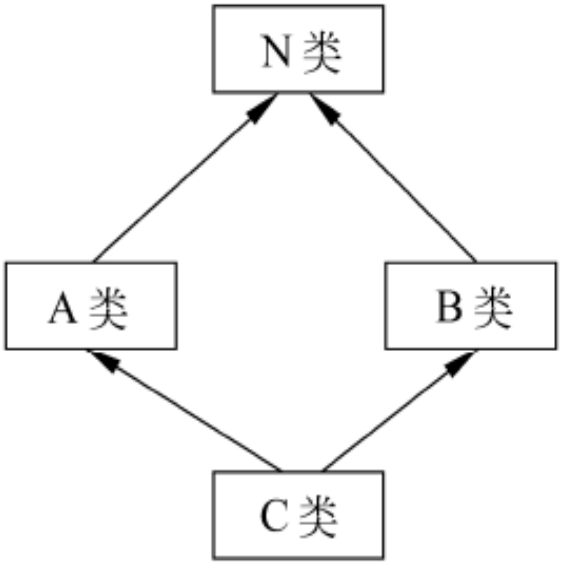
★派生类新增加的同名成员覆盖了基类中的同名成员（基类的同名成员在派生类中被屏蔽）。通过对象名访问同名的成员，则访问的是派生类的成员。

请注意:不同的成员函数，只有在函数名和参数个数相同、类型相匹配的情况下才发生同名覆盖，如果只有函数名相同而参数不同，不会发生同名覆盖，而属于函数重载。

第10讲 继承机制

❖ 多重继承和重复继承

- (3)派生类继承自从同一个基类派生的类



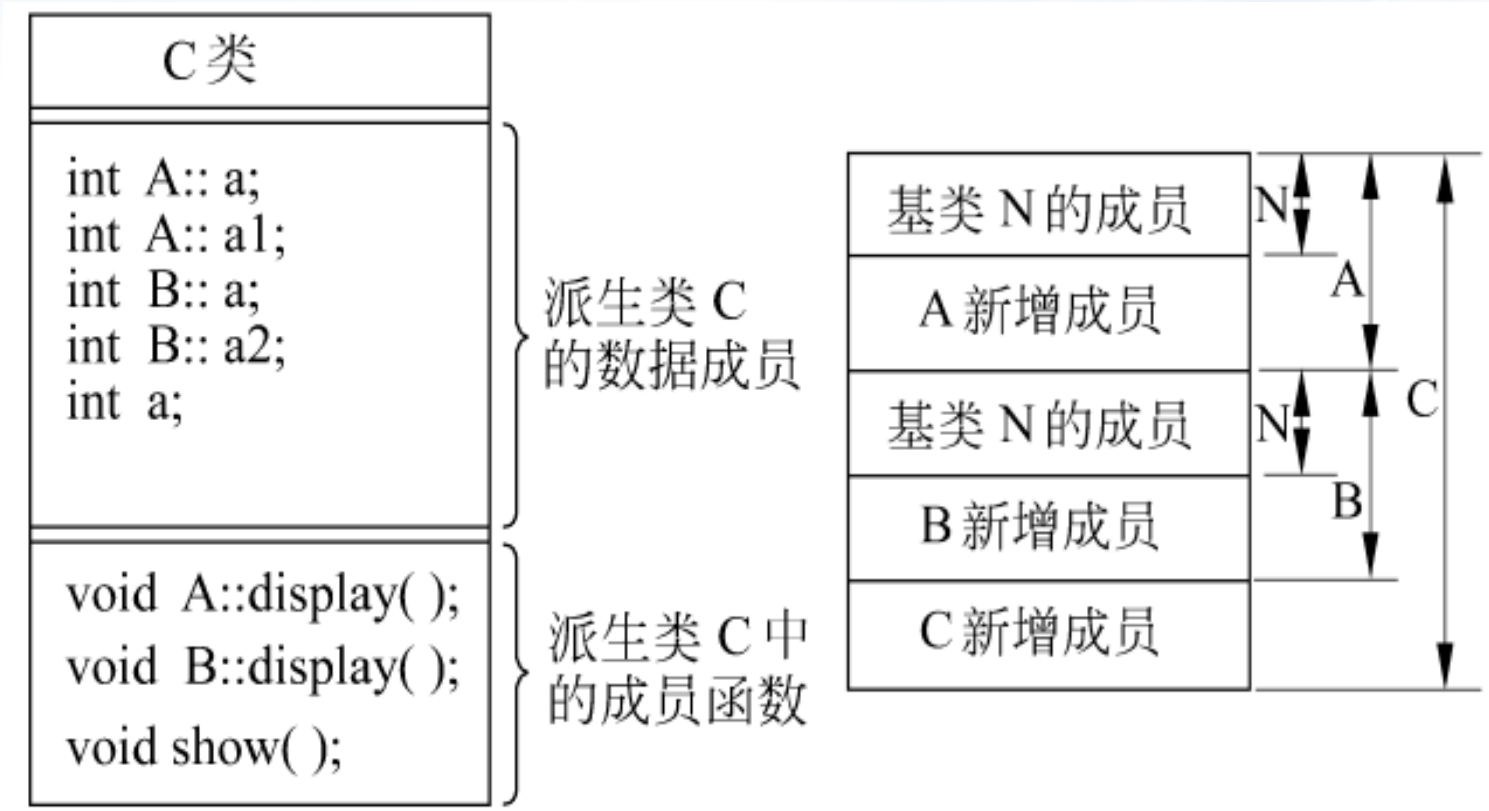

```
class N{
public:
int a;
void display(){
cout<<"A::a="<<a<<endl;}
};
class A:public N{
public:
int a1;
};
class B:public N{
public:
int a2;
};
```

```
class C :public A,public B
{public :
int a3;
void show( ){
cout<<"a3="<<a3<<endl;}
};
```

第10讲 继承机制

❖ 多重继承和重复继承

- **(3)派生类继承自从同一个基类派生的类**



第10讲 继承机制

❖ 多重继承和重复继承

- **(3)派生类继承自从同一个基类派生的类**
- 如何才能访问类A中从基类N继承下来的成员呢？

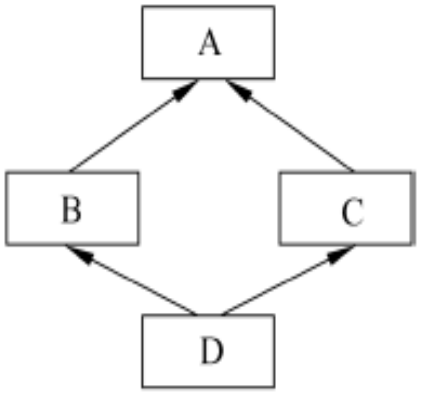
```
C  c1;
① c1.a=3;
   c1.display( );
② c1.N::a=3;
   c1.N::display();
③ c1.A::a=3;
   c1.A::display();
```

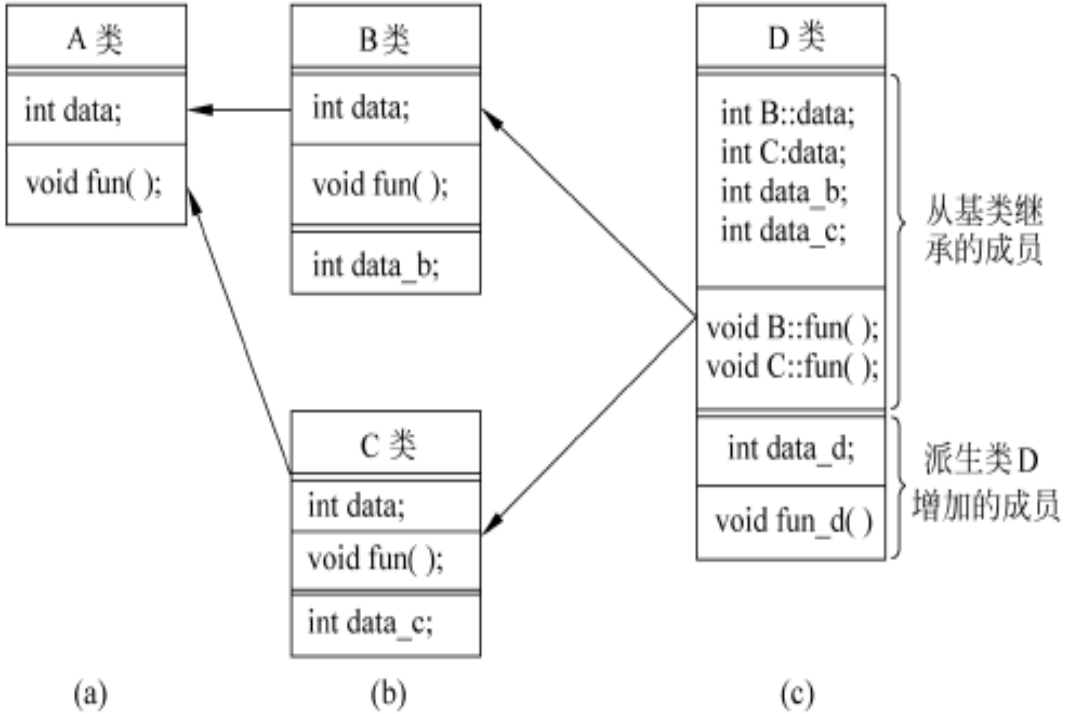
- ① ②不对。因为无法区别是类A中从基类N继承下来的成员，还是类B中从基类N继承下来的成员。

  ③对。通过类N的直接派生类名来指出要访问的是类N的哪一个派生类中的基类成员。

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 在一个类中保留间接共同基类的多份同名成员，这种现象是人们不希望出现的。C++提供虚基类(virtual base class)的方法，使得在继承间接共同基类时只保留一份成员。





第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 将A声明为虚基类。语法：

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

注意：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。因为一个基类可以在生成一个派生类时作为虚基类，而在生成另一个派生类时不作为虚基类。

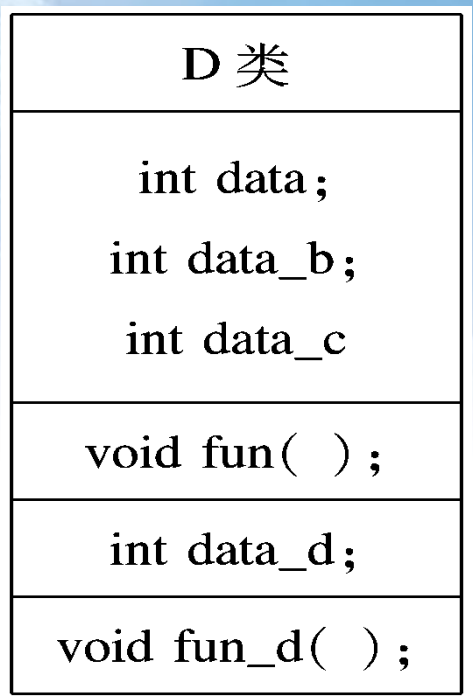
在派生类B和C做了上面的声明后，派生类D中的成员如右图：

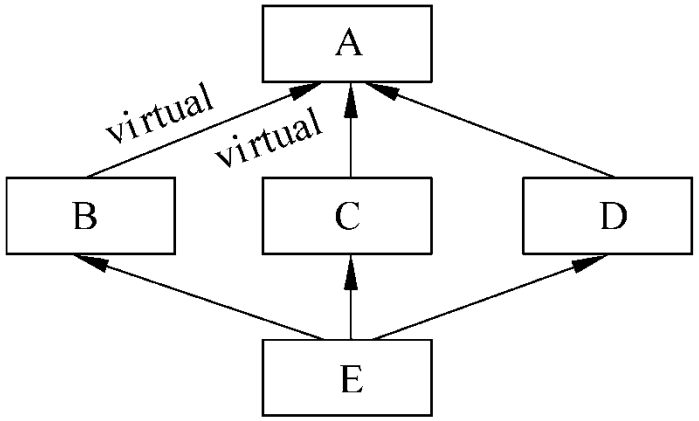
基类A成员只保留一次继承。

！！！注意：

为保证虚基类在派生类中只继承一次，应当在该基类所有直接派生类中都声明为虚基类。否则仍会出现对基类的多次继承。

如右图，类D中没有声明类A为虚基类，则类E会保留对基类A成员的2次继承。





第10讲 继承机制

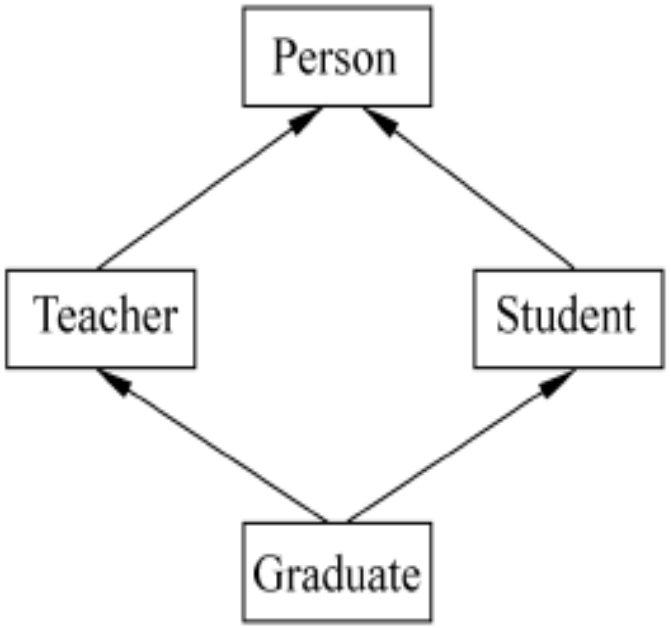
多重继承和重复继承（虚基类）

- (3)派生类继承自从同一个基类派生的类
- 虚基类的初始化：如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则在其所有派生类(包括直接派生或间接派生的派生类)中，通过构造函数的初始化表对虚基类进行初始化。

- 注意：规定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。
- C++编译系统只执行最后的派生类对虚基类的构造函数的调用，而忽略虚基类的其他派生类(如类B和类C)对虚基类的构造函数的调用，这就保证了虚基类的数据成员不会被多次初始化。

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 虚基类的应用例




```
//声明公共基类Person
class Person{
public:
Person(string nam,char s,int a)//构造函数
 {   name=nam;
      sex=s;
      age=a;
}
protected: //保护成员
    string name;
    char sex;
    int age;
};
```

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 虚基类的应用例

```
//声明Person的直接派生类Teacher
class Teacher :virtual public Person {
//声明Person为公用继承的虚基类
public:
   Teacher(string nam,char s,int a, string t): Person(nam,s,a)
   //构造函数
   {     title=t;     }

protected: //保护成员
   string title; //职称
};
```

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 虚基类的应用例

```
//声明Person的直接派生类Student
class Student :virtual public Person {
 //声明Person为公用继承的虚基类
public:
    Student(string nam,char s,inta,float sco) //构造函数
                          :Person(nam,s,a),score(sco){ } //初始化表
protected: //保护成员
    float score; //成绩
};
```

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 虚基类的应用例

```
//声明多重继承的派生类Graduate
class Graduate:public Teacher,public Student {
//Teacher和Student为直接基类
public:
    //构造函数
    Graduate(string nam,char s,int a, string t,float sco,float w)
:Person(nam,s,a),Teacher(nam,s,a,t)
  ,Student(nam,s,a,sco),wage(w) //初始化表
   {     }

//接下页
```

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 虚基类的应用例

```
//接上页

void show( ) //输出研究生的有关数据
{cout<<"name:"<<name<<endl;
cout<<"age:"<<age<<endl;
cout<<"sex:"<<sex<<endl;
cout<<"score:"<<score<<endl;
cout<<"title:"<<title<<endl;
cout<<"wages:"<<wage<<endl;
}
private:
   float wage; //工资
};
```

第10讲 继承机制

- 多重继承和重复继承（虚基类）
  - (3)派生类继承自从同一个基类派生的类
  - 虚基类的应用例

```
//主函数
int main( )
{Graduate grad1("Wang-li",'f',24,"assistant",89.5,1234.5);
grad1.show( );
return 0;
}
```

★使用多重继承时要十分小心，经常会出现二义性问题。

许多专业人员认为:不要提倡在程序中使用多重继承，只有在比较简单和不易出现二义性的情况或实在必要时才使用多重继承；能用单一继承解决的问题就不要使用多重继承。也是由于这个原因，有些面向对象的程序设计语言(如Java)并不支持多重继承。

第10讲 继承机制

❖ 对继承成员的调整

（1）恢复访问控制方式：在一些应用环境里，我们希望从基类继承的大多数公有成员都变为私有的或受保护的，仅其中一小部分人保持为公有成员。使用私有或受保护派生类可以屏蔽从基类继承下来的公有成员，但这样又导致所有公有成员均被屏蔽。使用访问声明将被屏蔽的公有成员恢复到原来的访问控制。形式，

基类名::成员名;

注意：访问仅能将继承成员恢复到原来的访问控制方式。如果原来在基类中是公有的，被继承访问控制protected或private屏蔽之后，只能用访问声明恢复为公有的，而不能改变为受保护或私有的。无法使用访问声明来调整访问控制方式。

第10讲 继承机制

❖ 对继承成员的调整

（1）恢复访问控制方式：

```
#include <iostream.h>
class BASE{
public:
  void set_i(int x)
  { i=x;        }
  int get_i()
  { return i;  }
protected:
  int j;
};
```

```
class DERIVED:private BASE{
public:
  BASE::set_i;
  void set_j(int x){
    j=x;
  }
  int get_ij()
  { return i+j;
  }
protected:
  int j;
};
```

```
int main()
{
    DERIVED obj;
    obj.set_i(5);
    obj.set_j(7);
   cout<<obj.get_ij()<<"\n";
    return 0;
}
```

第10讲 继承机制

❖ 对继承成员的调整

（2）继承成员的重定义：在语义上重新修改继承成员的函数实现。如果在派生类中定义了一个函数原型与继承成员一模一样的成员函数，则该函数实现的函数体式对继承成员函数的重定义。

如果对一个派生类的对象是用这个函数名进行函数调用，那么使用的将是派生类中新定义的成员函数，而不是原有的继承成员函数。

编译程序发现调用对象的一个成员函数时，首先在派生类中查找是否有该成员函数的定义，有则调用派生类函数，否则才继续查找基类对该成员函数的定义，直到查完所有的祖先类。

派生类中的名字支配了基类中的名字。

第10讲 继承机制

❖ 对继承成员的调整

（2）继承成员的重定义：

```
#include <iostream.h>
class BASE{
public:
  void set(int x)
  { i=x;        }
  int get_i()
  { return i;  }
protected:
  int i;
};
```

```
class DERIVED:public BASE{
public:
  void set(int x){
    j=x;
  }
  int get_ij()
  { return i+j;
  }
protected:
  int j;
};
```

```
int main()
{
    DERIVED obj;
    obj.set(7);
    obj.BASE::set(5);
   cout<<obj.get_ij()<<"\n";
    return 0;
}
```

★被重定义后继续使用基类中被屏蔽的成员，使用域运算符::

第10讲 继承机制

❖ 对继承成员的调整

（3）继承成员的重命名:在派生类中不改变继承成员的语义，仅仅为该成员起一个新的名字。重命名主要用于解决两个问题：一、解决名字冲突引起的问题，多重继承时，一个类的多个基类可能包含相同的名字；二、允许派生类可以根据新的应用环境选择更合适的，更容易理解的命名。

方法：C++没有提供直接的重命名机制，常用的方法是在派生类中增加一个以新名字命名的成员函数，该成员函数的实现仅仅是调用旧名字的函数。然后根据需要决定是否要屏蔽旧名字的继承成员函数。

第10讲 继承机制

❖ 对继承成员的调整

（4）屏蔽继承成员：C++无法在派生类中删除从基类继承下来的成员以节省存储空间，即派生类无法访问一些继承下来的成员，C++编译程序也为这些成员分配了存储空间。

C++屏蔽继承成员的方法：使用访问控制方式protected和private。派生类中成员的访问控制方式是由基类成员访问控制方式结合继承访问控制方式两者共同决定的。

如果要在一个公有派生类中屏蔽从基类继承下来的公有成员函数，通常在派生类的protected或private之后定义一个完全相同的函数原型，然而函数体为空，这样使原有的继承成员函数被派生类新定义的成员函数所支配，但新定义的成员函数变得不可用了。

第10讲 继承机制

❖ 对继承成员的调整

（4）屏蔽继承成员：

```
#include <iostream.h>
class BASE{
public:
  void set_i(int x)
  {     i=x;   }
  int get_i()
  {     return i;}
protected:
  int i;
};
```

```
class DERIVED:public BASE{
public:
  void set_ij(int x,int y){
    i=x;
    j=y;     }
  int get_ij()
  { return i+j;    }
protected:
  int j;
private:
  void set_i(int x)
{   }
};
```

```
int main()
{
    DERIVED obj;
    //obj.set_i(5);
    obj.set_ij(5,7);
   cout<<obj.get_ij()<<"\n";
    return 0;
}
```

第10讲 继承机制

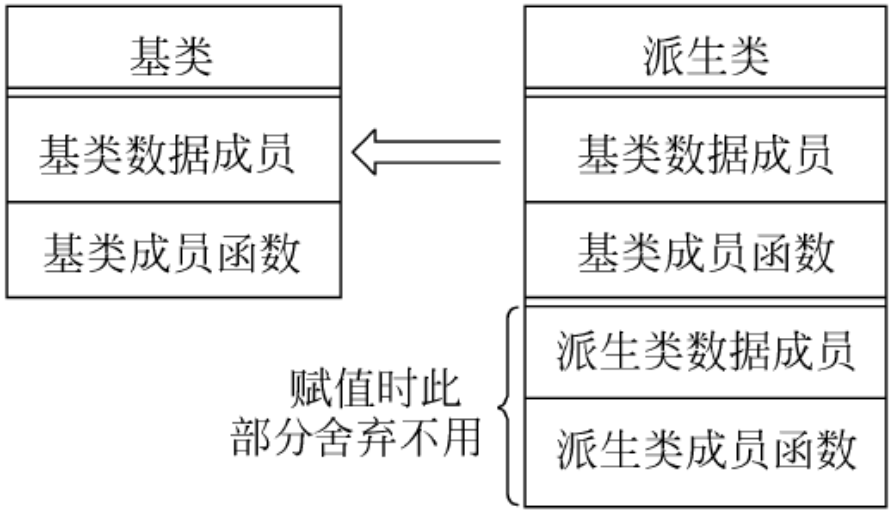
❖ 基类和派生类的转换

- 基类与派生类对象之间有赋值兼容关系，由于派生类中包含从基类继承的成员，因此可以将派生类的值赋给基类对象，在用到基类对象的时候可以用其子类对象代替。具体表现在以下几个方面:

(1)派生类对象可以向基类对象赋值: 可以用子类(即公用派生类)对象对其基类对象赋值。

```
A a1; //基类A对象a1
B b1; //类A的公有派生
a1=b1; //用派生类B对
```

在赋值时舍弃派生类自
只是对数据成员赋值，对成



第10讲 继承机制

- 基类和派生类的转换

(1)派生类对象可以向基类对象赋值

注意:赋值后不能通过基类对象a1去访问派生类对象b1的成员，因为b1的成员与a1的成员是不同的。

假设age是派生类B中增加的公用数据成员，分析下面的用法:

a1.age=23;//错误，a1不包含派生类增加的成员

b1.age=21; //正确，b1包含派生类增加的成员

- 子类型关系是单向的。B是A的子类型，不能说A是B的子类型。只能用子类对象对其基类对象赋值，而不能用基类对象对其子类对象赋值。同理，同一基类的不同派生类对象之间也不能赋值。

第10讲 继承机制

- 基类和派生类的转换

(2)派生类对象可以替代基类对象向基类对象的引用进行赋值或初始化。

```
A a1; //基类A的对象a1
B b1; //公有派生类B的对象b1
A& r=a1; //基类A的引用变量r，并用a1对其初始化
```

或者：

```
A& r=b1;//用派生类B对象b1对其初始化
```

或者：

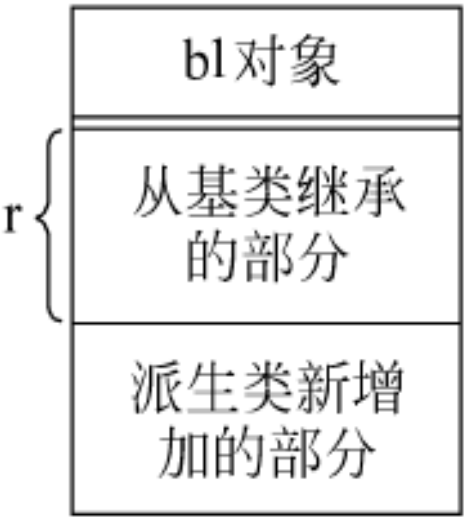
```
A a1;
B b1;
A& r=a1;
r=b1;//用派生类B对象b1对a1的引用变量r赋值
```

第10讲 继承机制

- 基类和派生类的转换

(2)派生类对象可以替代基类对象向基类对象的引用进行赋值或初始化。

注意：此时r并不是b1的别名，也不与b1共享同一段存储单元。它只是b1中基类部分的别名，r与b1中基类部分共享同一段存储单元，r与b1具有相同的起始地址。



第10讲 继承机制

- 基类和派生类的转换

(3)如果函数的参数是基类对象或基类对象的引用，相应的实参可以用子类对象。

```
void fun(A& r)//形参是类A的对象的引用变量
{ //输出该引用变量的数据成员
   cout<<r.num<<endl;    }
```

可能的使用方式：

```
A a1;
B b1;
fun(a1);
fun(b1);
```

★ 在调用fun函数时可以用派生类B的对象b1作实参:输出类B的对象b1的基类数据成员num的值。在fun函数中只能输出派生类中基类成员的值。

第10讲 继承机制

- 基类和派生类的转换

(4)派生类对象的地址可以赋给指向基类对象的指针变量

（指向基类对象的指针变量也可以指向派生类对象）

```
class Student
{
public:
  Student(int, string,float); //构造
  void display( ); //输出
protected:
  int num;  //学号
  string name; //姓名
  float score; //成绩
};
```

```
class Graduate:public Student {
public:
  Graduate( int, string ,float,float); //构造
  void display( ); //输出
private:
  float pay; //工资
};
```

第10讲 继承机制

- 基类和派生类的转换

(4)派生类对象的地址可以赋给指向基类对象的指针变量（指向基类对象的指针变量也可以指向派生类对象）

```
Student::Student(intn, string nam,float s) //定义构造函数
{
  num=n;
  name=nam;
  score=s;
}
void Student::display( ) //定义输出函数
{
  cout<<"num:"<<num<<endl;
  cout<<"name:"<<name<<endl;
  cout<<"score:"<<score<<endl;
}
```

第10讲 继承机制

- 基类和派生类的转换

(4)派生类对象的地址可以赋给指向基类对象的指针变量

（指向基类对象的指针变量也可以指向派生类对象）

```
//定义构造函数
Graduate::Graduate(int n, string nam,float s,float p):
Student(n,nam,s),pay(p)
{          }

//定义输出函数
void Graduate::display()
{
   Student::display(); //调用Student类的display函数
   cout<<"pay:"<<pay<<endl;
}
```

第10讲 继承机制

- 基类和派生类的转换

(4)派生类对象的地址可以赋给指向基类对象的指针变量

（指向基类对象的指针变量也可以指向派生类对象）

```
int main()
{Student stud1(1001,"Li",87.5); //Student类德对象
 Graduate grad1(2001,"Wang",98.5,563.5); //Graduate类的对象
Student*pt=&stud1;//定义指向Student类对象的指针，指向stud1
pt->display( );
pt=&grad1; //指针指向grad1
pt->display( );
}
```

1）仔细研究输出结果？考虑原因。

2）若要通过指针使用派生类的display(),如何做到？

第10讲 继承机制

❖ 基类和派生类的转换

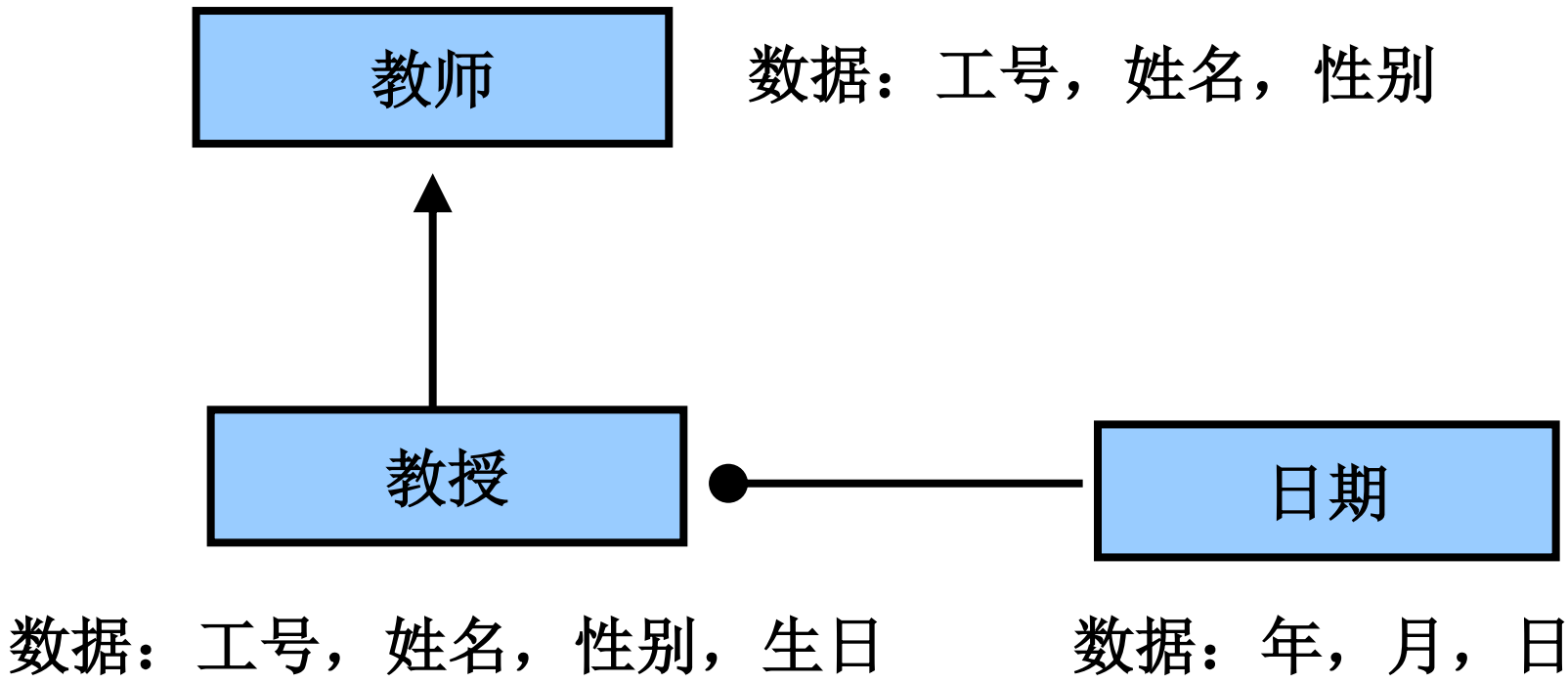
(4)派生类对象的地址可以赋给指向基类对象的指针变量

（指向基类对象的指针变量也可以指向派生类对象）

- 用指向基类对象的指针变量指向子类对象是合法的、安全的，不会出现编译上的错误。
- 但在应用上却不能完全满足人们的希望，人们有时希望通过使用基类指针就能够灵活自如的使用基类和子类对象的成员。➔虚函数和多态性。

第10讲 继承机制

- 继承和组合(两种不同的重用策略)
  - 组合(composition)：在一个类中以另一个类的对象作为数据成员的。



第10讲 继承机制

- 继承和组合(两种不同的重用策略)

```
class Teacher {  //教师类
public:
……
private:
  int num;
  string name;
  char sex;
};
```

```
class Date {  //日期类
public:
……
private:
  int year;
  int month;
  int day;
};
```

```
class Professor:public Teacher {   //教授类
public:
……
private:
  Date birthday;
};
```

思考：为什么人们如此看重继承？他们不是将已有的类加以修改来满足应用要求，而是尽可能地通过继承机制创建一批新的类？

继承在软件开发中的重要意义：

☆要求保留原有的基类不被改变——继承建立新类，即继承了基类的所有特性，又不会破坏基类。

☆用户往往得不到基类的源代码——无法对基类进行修改以适应程序的需要。

☆类库中的基类不允许修改———一个基类可能已与多个用户程序建立了某种关系。

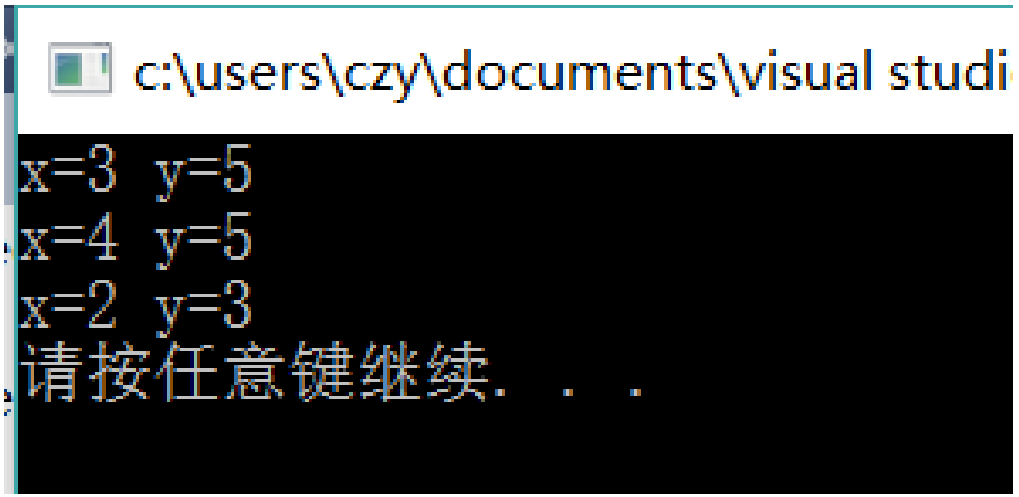
☆许多基类是专门作为基类设计的，并没有独立功能——这些基类只是一个框架，或者说是抽象类。

☆需要设计类的层次结构——不断地从抽象到具体，逐步实现。

课堂练习

```
#include <iostream.h>
  class Sample
  {
    protected:
    int x;
    public:
    Sample() { x=0; }
    Sample(int val) { x=val; }
    void operator++() { x++; }
  };
```

```
class Derived:public Sample
{
   int y;
   public:
   Derived():Sample(){ y=0; }
   Derived(int val1,int val2):Sample(val1){ y=val2; }
   void operator--(){ x--;y--;}
   void disp()
   {
     cout<<"x="<< x <<" y=" << y << endl;
   }
};
void main ()
{
   Derived d(3,5);
   d.disp();
   d++;
   d.disp ();
   d--;
   d--;
   d.disp();
}
```



```
#include <iostream.h>
#include<iostream.h>
class Base
 {    int i;
   public:
      Base(int n){i=n;cout <<"Constucting base class" << i<<endl;}
     ~Base(){cout <<"Destructing base class" << i<<endl;}
      void showi(){cout << i<< ",";}
      int Geti(){return i;}
   };
 class Derived:public Base
{    int j;
     Base aa;
  public:
      Derived(int n,int m,int p):Base(m),aa(p){
      cout << "Constructing derived class" <<endl;
      j=n;
      }
      ~Derived(){cout <<"Destructing derived clas
      void show(){Base::showi();
         cout << j<<"," << aa.Geti() << endl;}
    };
 void main()
 { Derived obj(8,13,24);
 obj.show();
 }
```

```
Constucting base class13
Constucting base class24
Constructing derived class
13,8,24
Destructing derived class
Destructing base class24
Destructing base class13
Press any key to continue
```

写出程序的运行结果！

C++程序设计（II）

浙江工业大学计算机学院