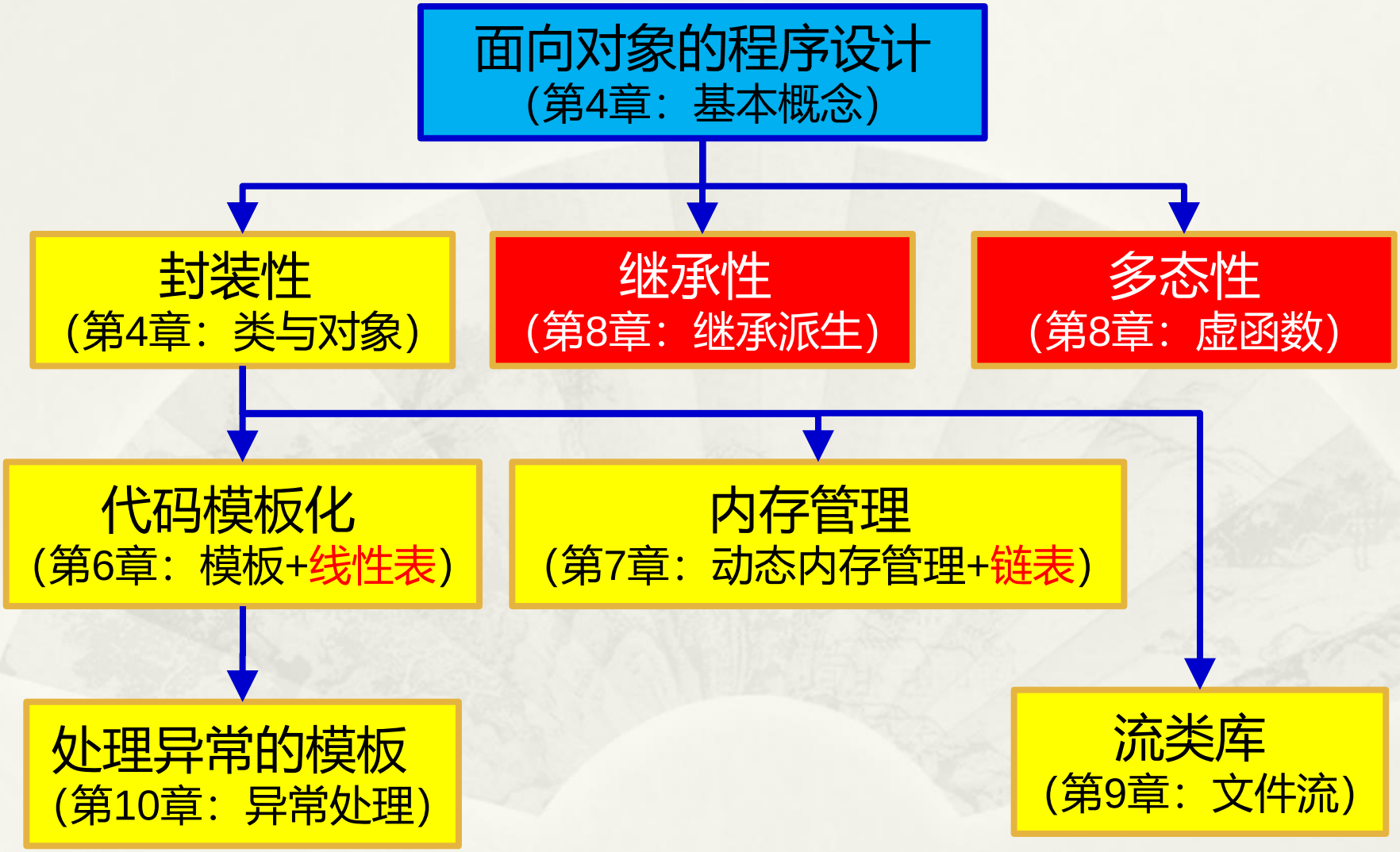
程序设计与算法语言II

第八章 继承与多态

曹鹏
Email: caopeng@seu.edu.cn
Tel: 13851945861

面向对象的程序设计


面向对象的程序设计
(第4章：基本概念)
封装性
(第4章：类与对象)
继承性
(第8章：继承派生)
多态性
(第8章：虚函数)
代码模板化
(第6章：模板+线性表)
内存管理
(第7章：动态内存管理+链表)
处理异常的模板
(第10章：异常处理)
流类库
(第9章：文件流)

本章提纲

- 继承性（8.1,8.2,8.5节）
  - 基本概念（8.1.1节）
  - 派生方式：公有、私有、保护（8.1.2节）
  - 构造函数与析构函数（8.2节）
  - 多重继承（*8.3节）
  - 虚基类（*8.4节）
  - 应用讨论（8.5节）
- 多态性（8.6节）
  - 虚函数（8.6.1节）
  - 纯虚函数（8.6.2节）

面向对象的程序设计

- 封装性
  - 类→对象
  - 类：成员函数；成员变量
  - 类成员访问权限：公有；私有；继承
  - 成员函数一般公有；成员变量一般私有
  - 例外：静态成员；友元
- 继承性
- 多态性

面向对象的程序设计

- 封装性
- 继承性
  - 学生是人的一种
  - 人具有的属性（姓名，年龄）和操作（修改/读取姓名，修改/年龄），学生也具备
  - 学生还需要支持新的属性（学号）和操作（读取学号）
  - 在人的基础上，定义学生，可以“复用”原来的成员，并增加新的成员，称为继承
- 多态性

继承与派生

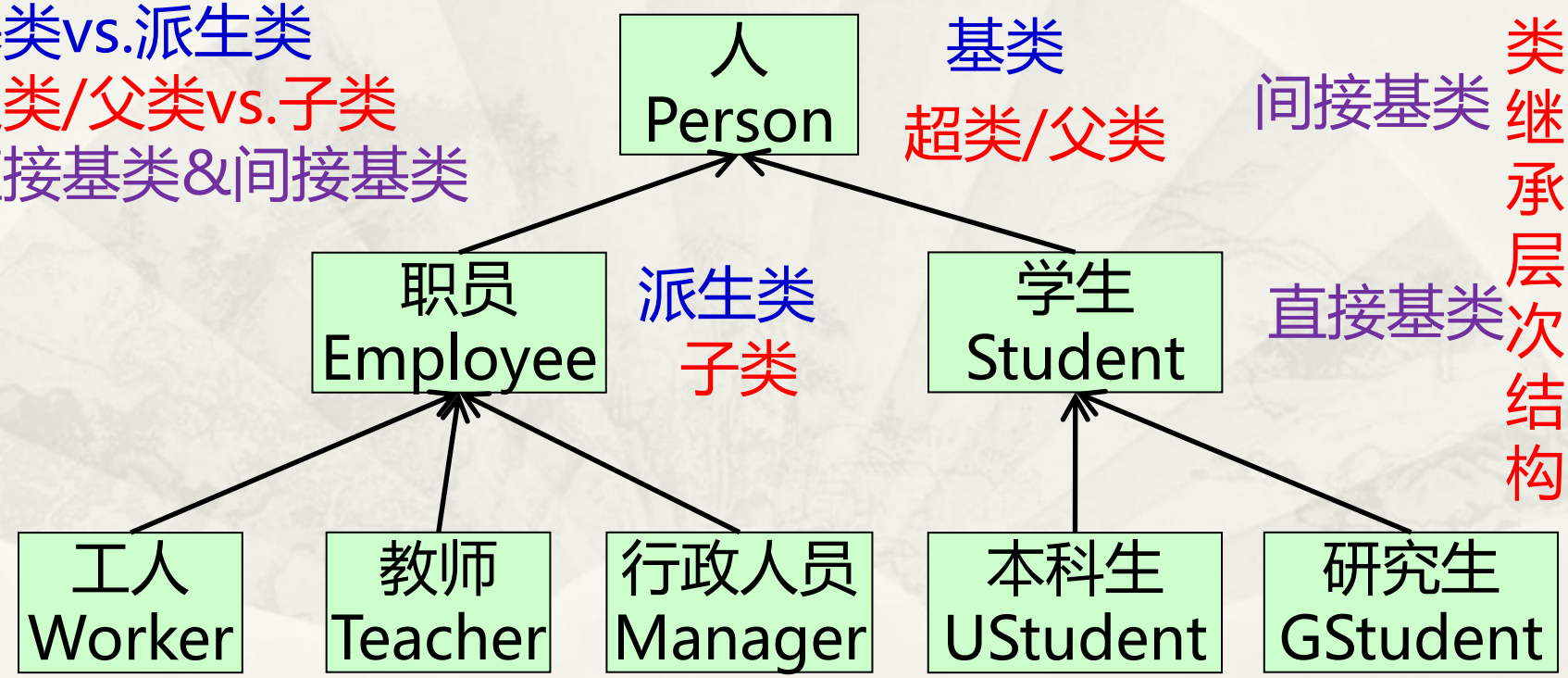
- 人
  - 姓名
  - 年龄
  - 读取姓名
  - 读取年龄

- 学生
  - 姓名
  - 年龄
  - 学号
  - 读取姓名
  - 读取年龄
  - 读取学号

继承与派生

- 派生：以一个(或多个)已经存在的类为基础，定义新的类
- 目的：代码复用

- 基类vs.派生类
- 超类/父类vs.子类
- 直接基类&间接基类



单继承 vs. 多重继承

- 单继承：只有一个直接基类（人→学生）
- 多重继承：有多个直接基类（学生+老师→助教）

派生过程

吸收基类成员，除了构造/析构函数

当派生类中希望沿用基类成员名，但赋予新的含义或功能时，改造基类成员

如何改造：定义同名成员，屏蔽基类成员

同名成员函数，称为函数覆盖(override)

函数重载(overload)

扩展新成员：派生类自己的新成员

重写构造/析构函数

派生类访问限定

- 类访问限定
  - 公有(public)/私有(private)/保护(protected)
  - 默认私有
  - 类成员函数（内部访问）：可以访问所有
  - 类对象（外部访问）：只能访问公有
- 派生类访问限定
  - 派生类成员函数（内部访问）能不能访问基类的公有(public)/私有(private)/保护(protected)?
  - 派生类对象（外部访问）呢?

派生类访问限定

- 类访问限定
  - 公有(public)/私有(private)/保护(protected)
  - 类成员函数（内部访问）：可以访问所有
  - 类对象（外部访问）：只能访问公有
- 派生类访问限定
  - 公有(public)/私有(private)/保护(protected)
  - 派生类成员函数（内部访问）
  - 派生类对象（外部访问）

		派生类对基类成员的访问限定		
		public	private	protected
基类成员访问限定	public			
	private			
	protected			

派生类成员函数/对象能否访问基类成员？

3*3*2=18

派生类访问限定

派生类对基类的访问限定	基类成员的访问限定	派生类的成员函数访问基类成员（内部访问）	派生类的类对象访问基类成员（外部访问）
public	public	Y (public)	
	private	N	
	protected	Y (protected)	
private	public	Y (private)	
	private	N	
	protected	Y (private)	
protected	public	Y (protected)	
	private	N	
	protected	Y (protected)	

派生类访问限定

派生类对基类的访问限定	基类成员的访问限定	派生类的成员函数访问基类成员（内部访问）	派生类的类对象访问基类成员（外部访问）
public	public	Y (public)	Y
	private	N	N
	protected	Y (protected)	N
private	public	Y (private)	N
	private	N	N
	protected	Y (private)	N
protected	public	Y (protected)	N
	private	N	N
	protected	Y (protected)	N

只要基类成员不是私有就能访问

两个都是公有才能访问

基类成员公有 + 公有派生

派生类的定义（单继承）

```
class 派生类名 : 访问限定符 基类名 {
private:
    成员表1;
public:
    成员表2;
protected:
    成员表3;
};
```

- ◆直接基类
- ◆可以同名覆盖基类成员，或定义新的成员。可以是对象成员。

派生类的定义（多重继承）

```
class 派生类名：访问限定符 基类名1，访问限定符 基类名2，… {
private:
        成员表1;
public:
        成员表2;
protected:
        成员表3;
};
```

- ◆直接基类
- ◆分别定义每个基类的访问限定方式

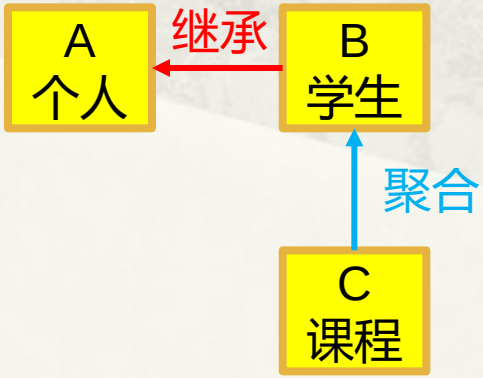
继承与聚合

◆定义基类A

```
class A{
.....
}
```

◆定义派生类B

```
class B:public A{
  C m_c;
  ......
}
```



◆继承（来自基类）

- ◆ A类作为B类的基类
- ◆ B属于(is a kind of)A，A是类别
- ◆ B只能/需从基类A继承1次

```
class A{public:int K;...};
class B:public A,public A{...};  //错误
```

◆聚合（来自对象成员）

- ◆ C类作为B类的对象成员
- ◆ B拥有(has a)C，C是属性
- ◆ B可以拥有2个成员对象

```
class B{
    C m_c1, m_c2;
 Public:
    ...};
```

派生类重写构造函数

- 为什么要重写构造函数
  - 初始化基类成员（未覆盖的）：调用基类构造函数完成
  - 初始化基类成员（覆盖的）：派生类构造函数完成
  - 初始化派生类成员（内置数据类型）：派生类构造函数完成
  - 初始化派生类成员（类对象）：调用成员对象构造函数完成

- 派生类构造函数定义格式

```
派生类名::派生类名（参数总表）
:基类名1（参数名表1），
 基类名2（参数名表2），
 ….,
 成员对象1（成员对象参数名表1），
 成员对象2（成员对象参数名表2），
 …
{
    ……..
}
```

派生类重写构造函数

```
//基类A定义
class A{
  char m_a;
public:
   //基类构造函数声明
   A(char a);
}
//基类构造函数定义
A::A(char a){
    m_a=a;
}
```

参数总表

参数名表

```
//派生类B定义
class B:public A{
  int m_b;
  C m_c;
public:
  //派生类构造函数声明
  B(char a, int b, C c);
}
//派生类构造函数定义
B::B(char a, int b, C c)
:A(a),m_c(c){
  m_b=b;
}
```

基类A
单继承
公有派生

对象成员m_c

基类成员形参　派生类成员形参

派生类重写构造函数

- 构造函数的执行顺序
  - 先依次执行各个直接基类的构造函数
    - 按派生类定义的先后顺序（不是按派生类构造函数中的先后顺序）
    - 如果直接基类本身也是派生类，则再先调用其直接基类的构造函数）
  - 再依次执行成员对象的构造函数
    - 按派生类定义的先后顺序（不是按派生类构造函数中的先后顺序）
  - 最后执行派生类定义的构造函数

派生类重写构造函数

```
//基类A
class A{
  char m_a;
public:
   A();
   A(char a);
}

//派生类B2
class B2:public A{
  int m_b;
public:
  B2(int b){m_b=b;}
//没有调用基类A构造函数，表示调用基类A无参数构造函数
}
```

```
//派生类B1
class B1:public A{
  int m_b;
public:
  B1(char a,int b):A(a) {m_b=b;}
//调用基类A带参数构造函数
}

//定义派生类B3
class B3:public A{
  int m_b;
//没有定义派生类B3构造函数
表示只会调用默认构造函数，此时调用基类A无参数构造函数


}
```

派生类重写析构函数

```
class B{
   …
public:
~B();  //声明
}
//定义
B::~B() {…};
```

- 析构顺序（与构造顺序相反）
  - 先执行派生类的析构函数
  - 再依次执行成员对象的析构函数（按构造时定义先后）
  - 最后依次执行直接基类的析构函数（按构造时定义先后）

派生类构造/析构函数

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

class A{
    char ch;
public:
    A(char n):ch(n){cout<<ch;};
};
class B:public A{
    char ch;
public:
    B(char n):A(n+1), ch(n)
    {cout<<ch;};
};
void main(){
    B b('a');
}
```

输出结果 D

A. aa
B. bb
C. ab
D. ba

【例8.1】派生类定义与使用

```
#include<iostream>
#include<string>
using namespace std;
enum Tsex{mid,man,woman};

struct course{
    string coursename;
    int grade;
};

class Person{
    string IdPerson;            //身份证号,18位数字
    string Name;                //姓名
    Tsex Sex;                 //性别
    int Birthday;               //生日,格式1986年8月18日写作19860818
    string HomeAddress;         //家庭地址
```

【例8.1】派生类定义与使用

```
public:
    Person(string, string,Tsex,int, string);
    Person();
    ~Person();
    void SetName(string);
    string GetName(){return Name;}
    void SetSex(Tsex sex){Sex=sex;}
    Tsex GetSex(){return Sex;}
    void SetId(string id){IdPerson=id;}
    string GetId(){return IdPerson;}
    void SetBirth(int birthday){Birthday=birthday;}
    int GetBirth(){return Birthday;}
    void SetHomeAdd(string );
    string GetHomeAdd(){return HomeAddress;}
    void PrintPersonInfo();
};
```

【例8.1】派生类定义与使用

```
Person::Person(string id, string name,Tsex sex,int birthday, string
    homeadd){
    IdPerson=id;
    Name=name;
    Sex=sex;
    Birthday=birthday;
    HomeAddress=homeadd;
}
Person::Person(){
    IdPerson="#";Name="#";Sex=mid;
    Birthday=0;HomeAddress="#";
}
Person::~Person(){}
void Person::SetName(string name){
    Name=name;                    //拷入新姓名
}
```

【例8.1】派生类定义与使用

```
void Person::SetHomeAdd(string homeadd){
    HomeAddress=homeadd;
}
void Person::PrintPersonInfo(){
    int i;
    cout<<"身份证号:"<<IdPerson<<'\n'<<"姓名:"<<Name<<'\n'<<"
    性别:";
    if(Sex==man)cout<<"男"<<'\n';
    else if(Sex==woman)cout<<"女"<<'\n';
    else cout<<" "<<'\n';
    cout<<"出生年月日:";
    i=Birthday;
    cout<<i/10000<<"年";
    i=i%10000;
    cout<<i/100<<"月"<<i%100<<"日"<<'\n'<<"家庭住
    址:"<<HomeAddress<<'\n';
}
```

【例8.1】派生类定义与使用

```
class Student:public Person{                    //定义派生的学生类
    string NoStudent;                           //学号
    course cs[30];                              //30门课程与成绩
public:
    Student(string id, string name,Tsex sex,int birthday, string
    homeadd, string nostud);
    //注意派生类构造函数声明方式
    Student();
    ~Student();
    int SetCourse(string,int);
    int GetCourse(string);
    void PrintStudentInfo();
};
```

【例8.1】派生类定义与使用

```
Student::Student(string id, string name,Tsex sex,int birthday, string
    homeadd, string nostud):Person(id,name,sex,birthday,homeadd){
    int i ;
    NoStudent=nostud;
    for(i=0;i<30;i++){ //课程与成绩清空,将来由键盘输入
        cs[i].coursename="#";
        cs[i].grade=0;
    }
}
Student::Student(){ //基类默认的无参数构造函数不必显式给出
    int i;
    NoStudent="#";
    for(i=0;i<30;i++){ //课程与成绩清空,将来由键盘输入
        cs[i].coursename="#";
        cs[i].grade=0;
    }
}
Student::~Student(){}
```

【例8.1】派生类定义与使用

```
int Student::SetCourse(string coursename,int grade){ //设置课程
    int i;
    bool b=false;                          //标识新输入的课程,还是更新成绩
    for(i=0;i<30;i++){
        if(cs[i].coursename=="#"){      //判断表是否进入未使用部分
            cs[i].coursename=coursename;
            cs[i].grade=grade;
            b=false;
            break;
        }
        else if(cs[i].coursename==coursename){   //是否已有该课程记录
            cs[i].grade=grade;
            b=true;
            break;
        }
    }
    if(i==30) return 0;       //成绩表满返回0
    if(b) return 1;           //修改成绩返回1
    else return 2;            //登记成绩返回2
}
```

【例8.1】派生类定义与使用

```
int Student::GetCourse(string coursename){
    int i;
    for(i=0;i<30;i++)
        if(cs[i].coursename==coursename) return cs[i].grade;
    return -1;
}//找到返回成绩,未找到返回-1

void Student::PrintStudentInfo(){
    int i;
    cout<<"学号:"<<NoStudent<<'\n';
    PrintPersonInfo();
    for(i=0;i<30;i++)//打印各科成绩
        if(cs[i].coursename!="#")
            cout<<cs[i].coursename<<'\t'<<cs[i].grade<<'\n';
        else break;
    cout<<"--------完-------- "<<endl;
}
```

【例8.1】派生类定义与使用

```
int main(void){
    char temp[30];
    int i,k;
    Person per1("320102820818161","沈俊",man,19820818,"南京四牌楼2
    号");
    Person per2;
    per2.SetName("朱明");
    per2.SetSex(woman);
    per2.SetBirth(19780528);
    per2.SetId("320102780528162");
    per2.SetHomeAdd("南京市成贤街9号");
    per1.PrintPersonInfo();
    per2.PrintPersonInfo();
    Student stu1("320102811226161","朱海鹏",man,19811226,"南京市黄
    浦路1号","06000123");
    cout<<"请输入各科成绩:"<<'\n'; //完整的程序应输入学号,查找,再操作
```

【例8.1】派生类定义与使用

```
while(1){                          //输入各科成绩,输入"end"停止
    cin>>temp;                         //输入格式:物理  80
    if(!strcmp(temp,"end")) break;
    cin>>k;
    i=stu1.SetCourse(temp,k);
    if(i==0) cout<<"成绩列表已满!"<<'\n';
    else if(i==1) cout<<"修改成绩"<<'\n';
    else cout<<"登记成绩"<<'\n';
}
stu1.PrintStudentInfo();
while(1){
    cout<<"查询成绩"<<'\n'<<"请输入科目:"<<'\n';
    cin>>temp;
    if(!strcmp(temp,"end")) break;
    k=stu1.GetCourse(temp);
    if(k==-1)cout<<"未查到"<<'\n';
    else cout<<k<<'\n';
}
return 0;
}
```

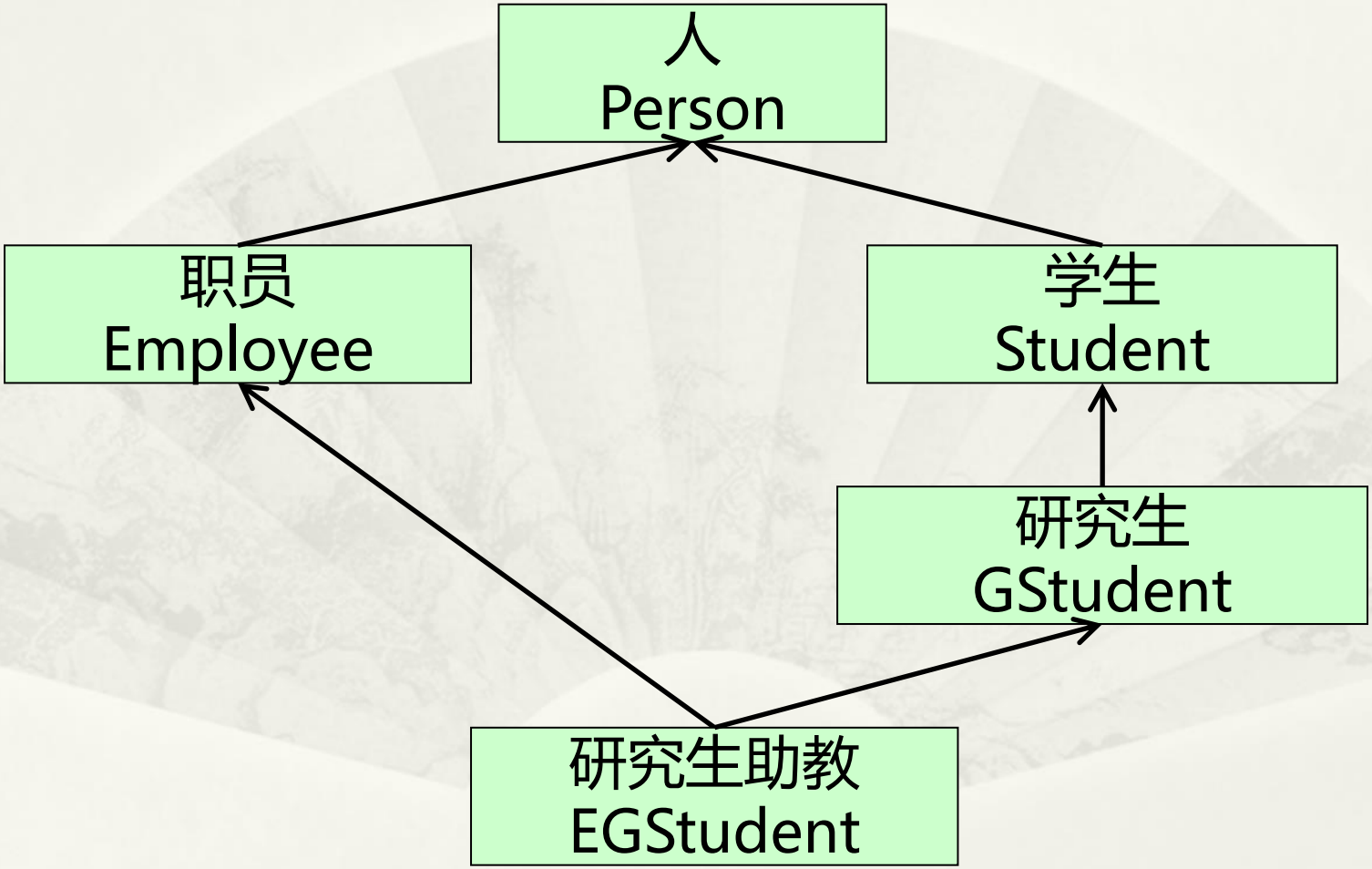
【例8.1】派生类定义与使用

身份证号:320102820818161
姓名:沈俊
性别:男
出生年月日:1982年8月18日
家庭住址:南京四牌楼2号
身份证号:320102780528162
姓名:朱明
性别:女
出生年月日:1978年5月28日
家庭住址:南京市成贤街9号
请输入各科成绩:
语文 90
登记成绩
语文 85
修改成绩
物理 95
登记成绩
英文 93
登记成绩
end

学号:06000123
身份证号:320102811226161
姓名:朱海鹏
性别:男
出生年月日:1981年12月26日
家庭住址:南京市黄浦路1号
语文　85
物理　95
英文　93
--------完--------
查询成绩
请输入科目:
语文
85
查询成绩
请输入科目:
end

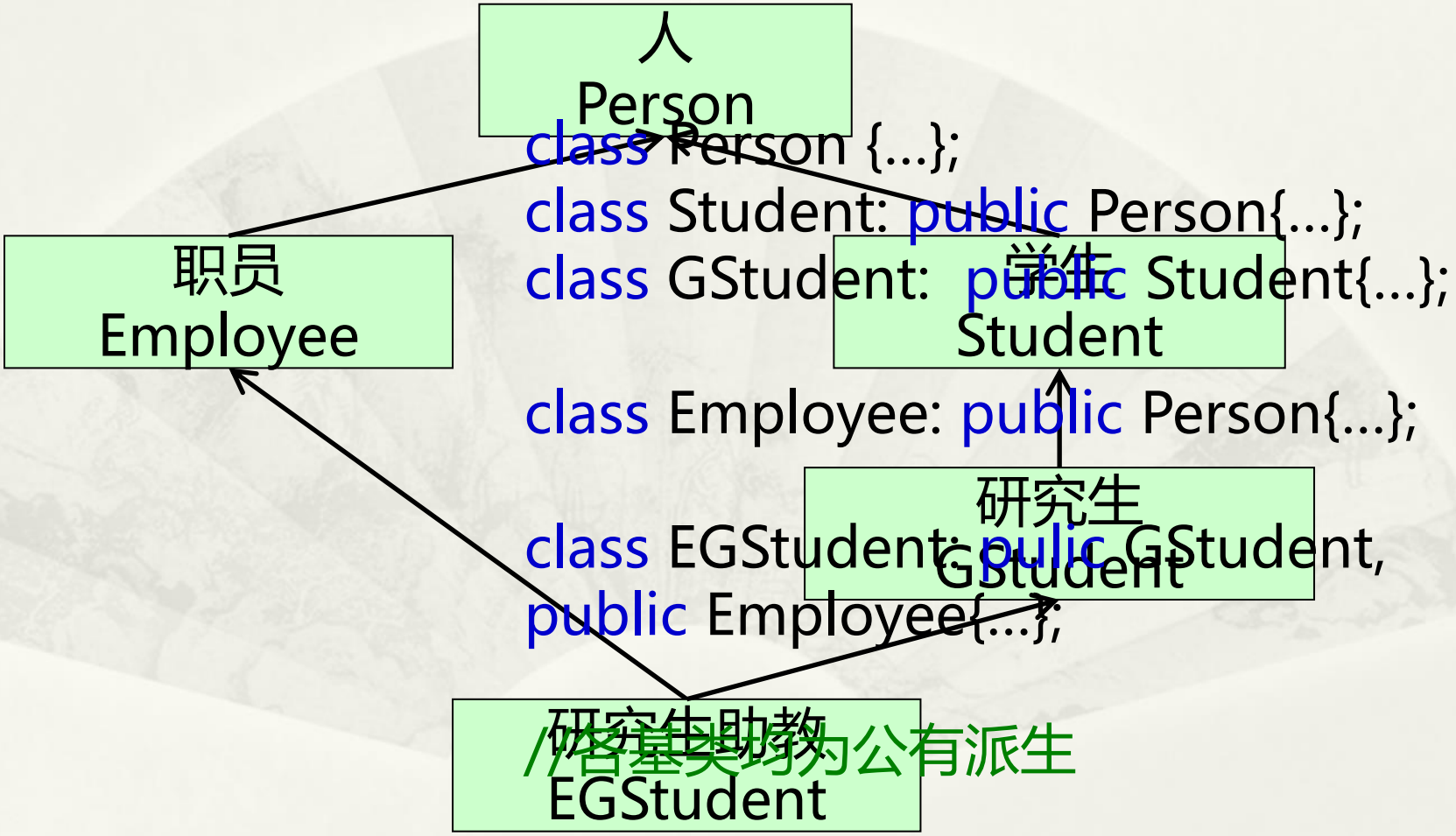
多重继承

◆多个基类共同派生出新的派生类



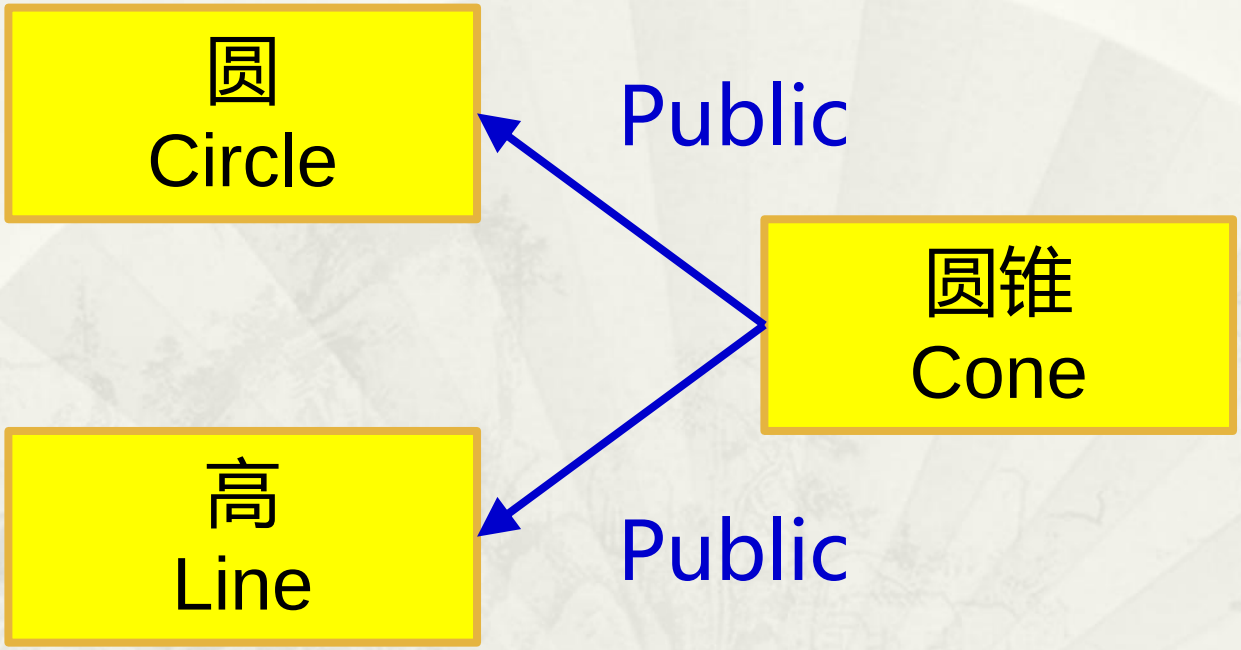
多重继承

◆多个基类共同派生出新的派生类



【例8.2】多重继承

◆由圆(circle)和高(Line)多重继承派生出圆锥(Cone)



【例8.2】多重继承

```
class Circle{
protected:  //Circle类对象不能访问，但公有派生类的成员函数可以访问
    float x,y,r;                    //(x,y)为圆心,r为半径
public:
    Circle(float a=0,float b=0,float R=0){x=a;y=b;r=R;}
    void Setcoordinate(float a,float b){x=a;y=b;}
    void Getcoordinate(float &a,float &b){a=x;b=y;}
    void SetR(float R){r=R;}
    float GetR(){return r;}
    float GetAreaCircle(){return float(r*r*3.14159);}
    float GetCircumference(){return float(2*r*3.14159);}
};
```

【例8.2】多重继承

```
class Line{
protected: //Circle类对象不能访问，但公有派生类的成员函数可以访问
    float High;
public:
    Line(float a=0){High=a;}
    void SetHigh(float a){High=a;}
    float GetHigh(){return High;}
};
```

【例8.2】多重继承派生出圆锥

```
class Cone:public Circle,public Line{
public:
    Cone(float a,float b,float R,float d):Circle(a,b,R),Line(d){}
    float GetCV(){  //计算体积
        return float(GetAreaCircle()*High/3);}
    float GetCA(){  //计算表面积
        return float(GetAreaCircle()+r*3.14159*sqrt(r*r+High*High));
    }//公有派生类中能直接访问直接基类的保护成员
};
```

【例8.2】多重继承派生出圆锥

```
class Cone:public Circle,public Line{
public:
    Cone(float a,float b,float R,float d):Circle(a,b,R),Line(d){}
    float GetCV(){  //计算体积
        return float(GetAreaCircle()*High/3);}
    float GetCA(){  //计算表面积
        return float(GetAreaCircle()+r*3.14159*sqrt(r*r+High*High));
    }//公有派生类中能直接访问直接基类的保护成员
};
```

【例8.2】多重继承派生出圆锥

```
int main(){
    Cone c1(5,8,3,4);
    float a,b;
    cout<<"圆锥体积:"<<c1.GetCV()<<'\n';
    cout<<"圆锥表面积:"<<c1.GetCA()<<'\n';
    cout<<"圆锥底面积:"<<c1.GetAreaCircle()<<'\n';
    cout<<"圆锥底周长:"<<c1.GetCircumference()<<'\n';
    cout<<"圆锥底半径:"<<c1.GetR()<<'\n';
    c1.Getcoordinate(a,b);
    cout<<"圆锥底圆心坐标:("<<a<<','<<b<<")\n";
    cout<<"圆锥高:"<<c1.GetHigh()<<'\n';
    return 0;
}
```

派生类Cone成员函数

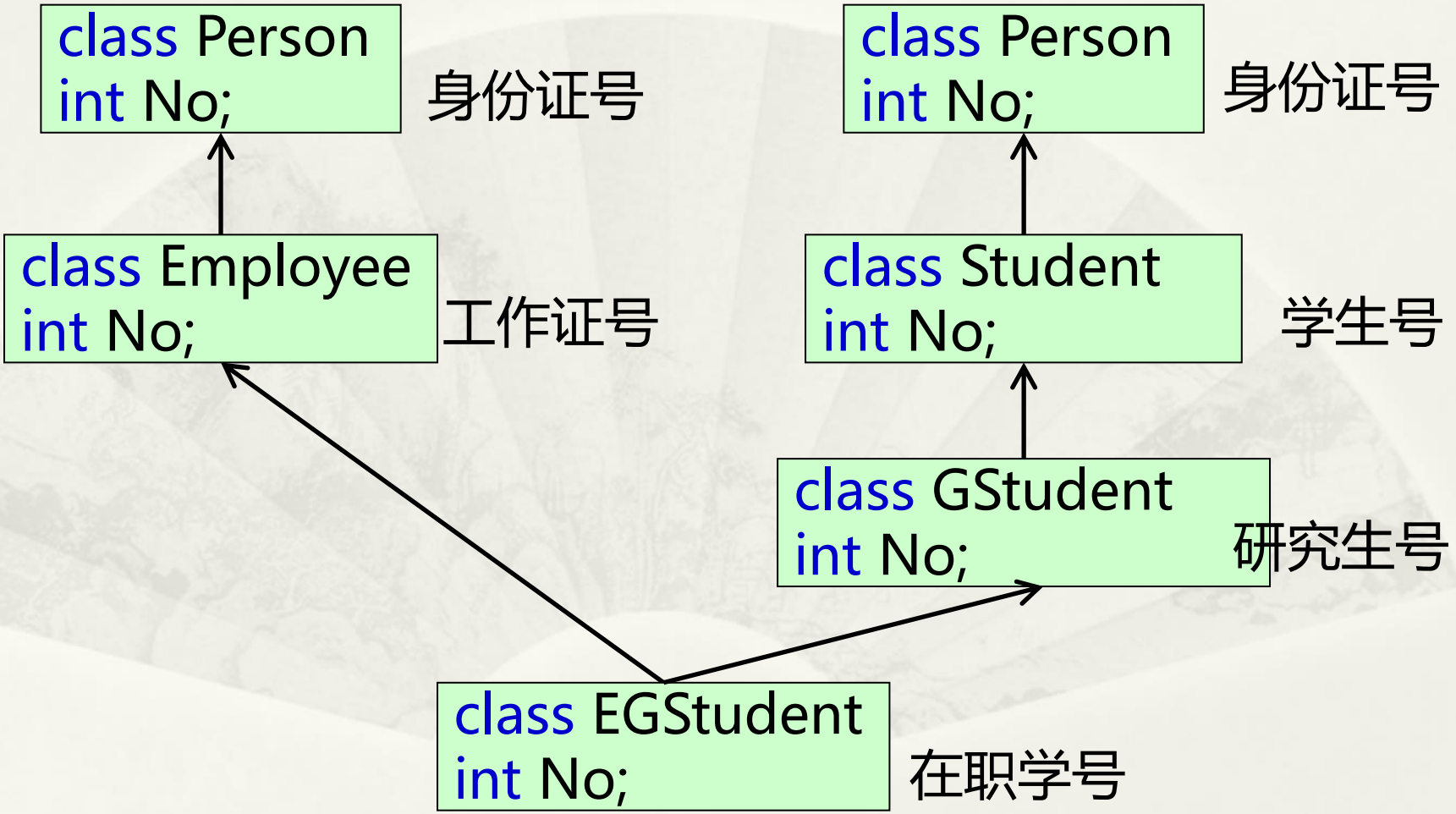
基类Circle成员函数

基类Line成员函数

```
圆锥体积:37.6991
圆锥表面积:75.3982
圆锥底面积:28.2743
圆锥底周长:18.8495
圆锥底半径:3
圆锥底圆心坐标:(5,8)
圆锥高:4
```

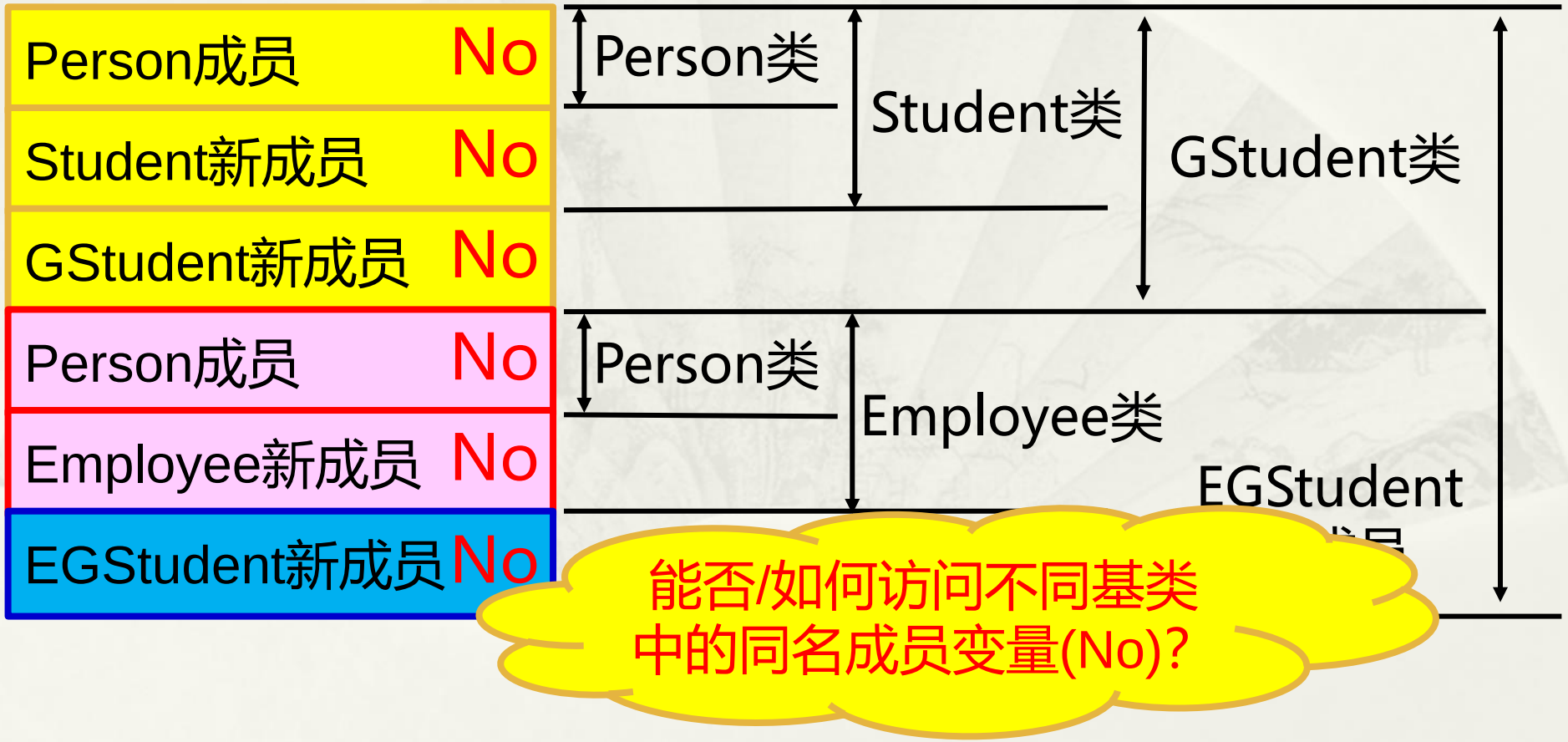
多重继承

◆成员变量（如No）的二义性问题



多重继承

◆类对象成员存储结构



多重继承

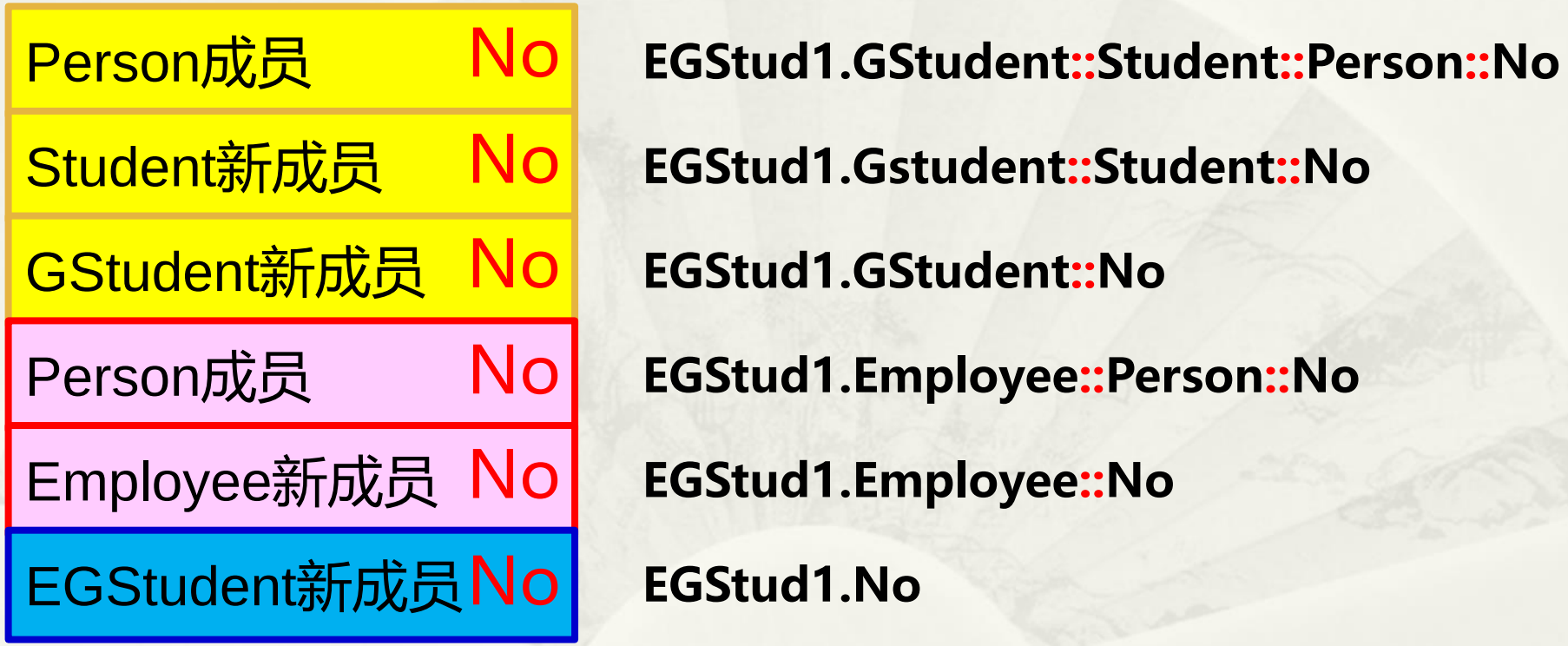
◆各（直接/间接）基类中成员变量的访问方式

Person成员	No
Student新成员	No
GStudent新成员	No
Person成员	No
Employee新成员	No
EGStudent新成员	No

多重继承

◆各（直接/间接）基类中成员变量的访问方式

EGStudent EGStud1; //各基类均为公有派生, No为公有成员变量



:: 作用域辨析符

多重继承

◆也适用于各（直接/间接）基类中成员函数访问

EGStudent EGStud1; //各基类均为公有派生, getNo为公有成员函数

EGStud1.GStudent::Student::Person::getNo()

EGStud1.GStudent::Student::getNo()

EGStud1.GStudent::getNo()

EGStud1.Employee::Person::getNo()

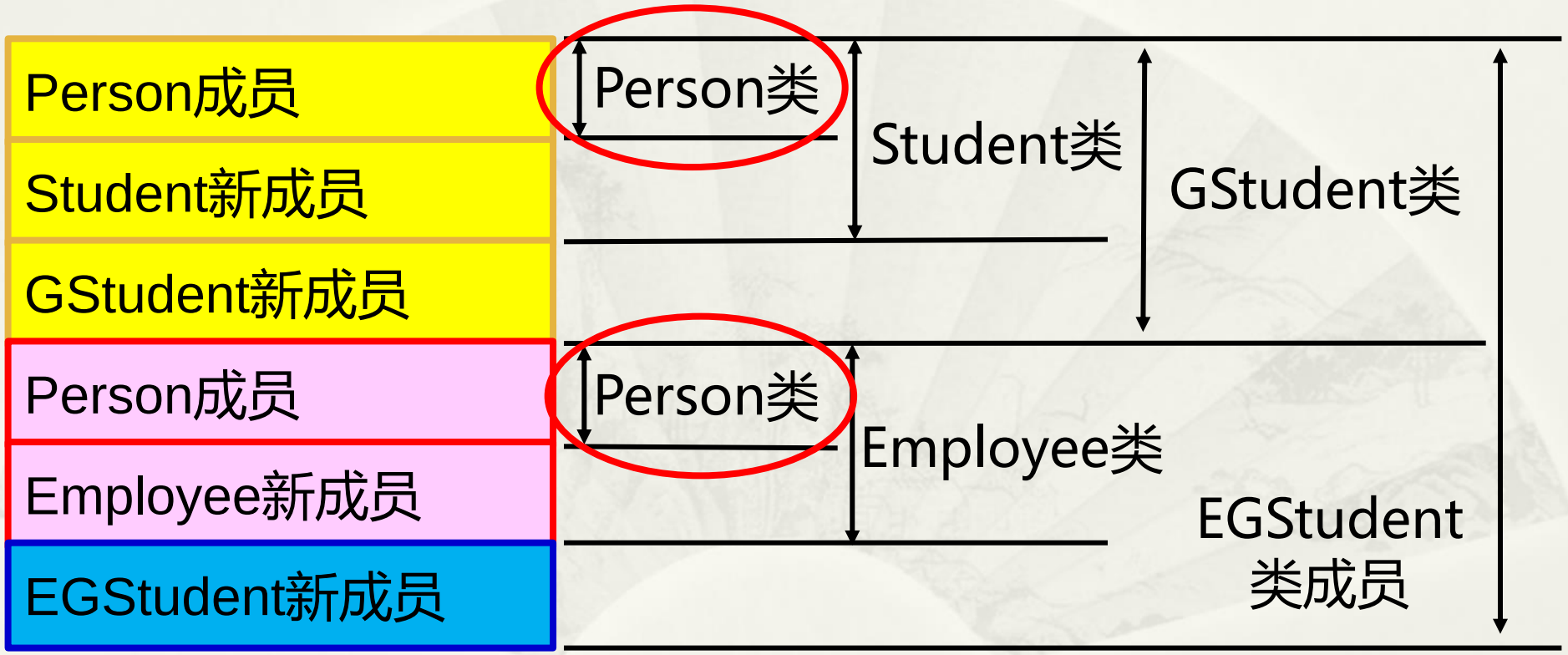
EGStud1.Employee::getNo()

EGStud1.getNo()

:: 作用域辨析符

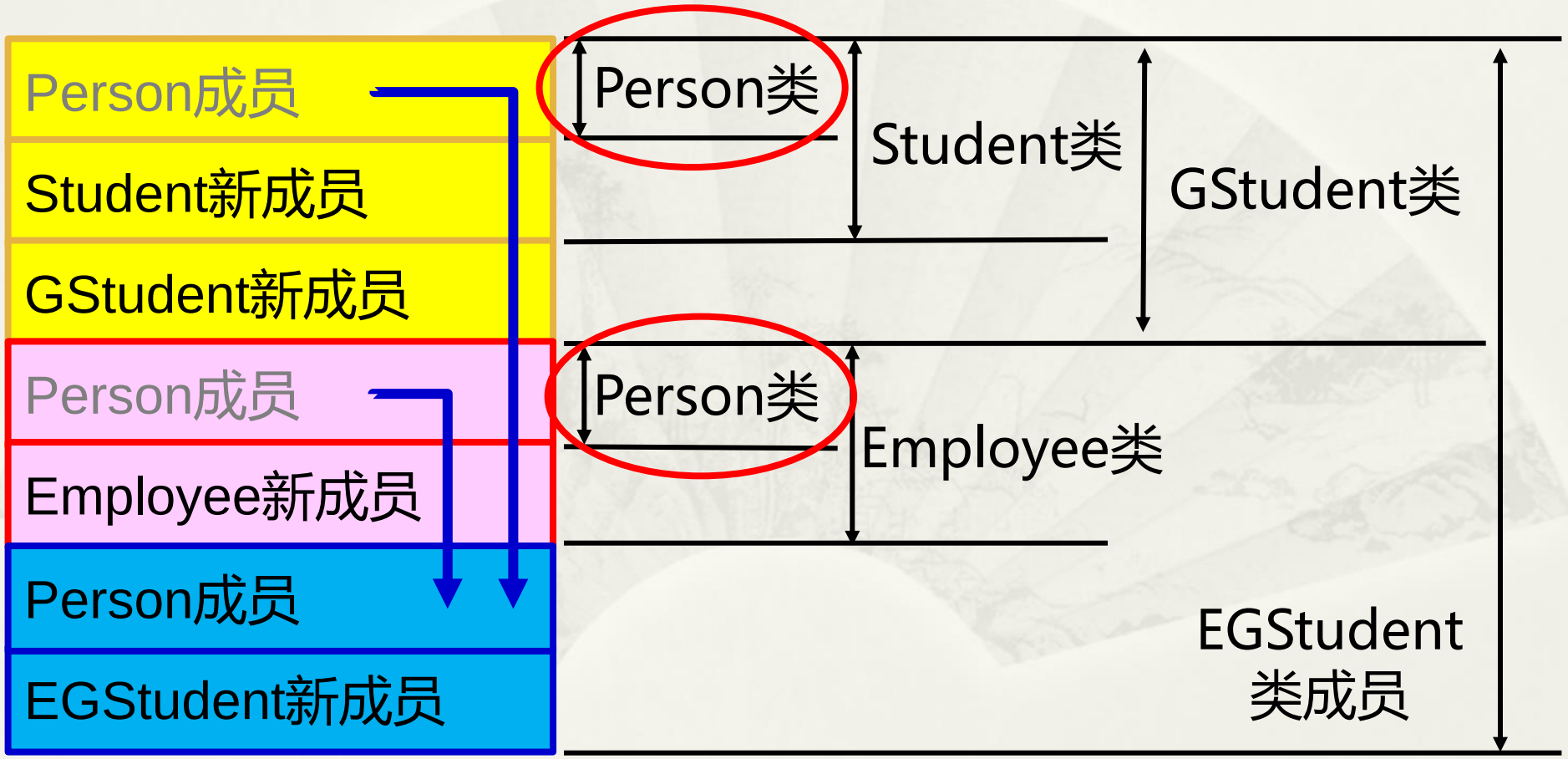
虚基类

◆问题由来：不应该出现两个Person类对象成员



虚基类

◆解决方法：将共同基类Person类定义为虚基类，使得只存储一份Person成员（虚拟继承）



虚基类

◆定义方式

```
//普通继承方式
class Student: public Person{...};

//虚拟继承方式
class Student: virtual public Person{...};
class Student: public virtual Person{...};
```

虚基类

◆定义方式

```
class Person {…};
class Student: virtual public Person{…};
class GStudent: public Student{…};

class Employee: virtual public Person{…};

class EGStudent: public GStudent, public
Employee{…};
```

虚基类

◆构造函数定义

```
EGStudent::EGStudent(int NoStu, int NoGStu, int
NoEmpl, int No)
:GStudent(No, NoStu, NoGStu), //直接基类
Employee(No, NoEmpl), //直接基类
Person(No)  //间接基类
{…}
```

◆GStudent, Employee：直接基类（多重继承）

◆Person：间接基类，作为虚基类，必须显式给出构造函数

虚基类

◆构造函数调用顺序

```
EGStudent::EGStudent(int NoStu, int NoGStu, int
NoEmpl, int No)
:GStudent(NoStu, NoGStu),
Employee(NoEmpl),
Person(No)
{…}
```

1. 虚基类(Person)
2. 直接基类(GStudent, Employee)
3. 自身(EGStudent)

◆析构函数与之相反

【例8.3】虚基类的多重继承

```
#include<iostream>
using namespace std;
class Object{
public:
    Object(){cout<<"constructor Object\n";}
    ~Object(){cout<<"deconstructor Object\n";}
};
```

【例8.3】虚基类的多重继承

```
class Bclass1{
public:
    Bclass1(){cout<<"constructor Bclass1\n";}
    ~Bclass1(){cout<<"deconstructor Bclass1\n";}
};
class Bclass2{
public:
    Bclass2(){cout<<"constructor Bclass2\n";}
    ~Bclass2(){cout<<"deconstructor Bclass2\n";}
};
class Bclass3{
public:
    Bclass3(){cout<<"constructor Bclass3\n";}
    ~Bclass3(){cout<<"deconstructor Bclass3\n";}
};
```

【例8.3】虚基类的多重继承

```
class Dclass:public Bclass1,virtual Bclass3,virtual Bclass2{ //默认私有
    Object object;
public:
    Dclass():object(),Bclass2(),Bclass3(),Bclass1()
    {cout<<"派生类建立!\n";}
    ~Dclass(){cout<<"派生类析构!\n";}
};
int main(){
    Dclass dd;
    cout<<"主
    return 0;
}
```

```
Constructor Bclass3   //第一个虚拟基类,与派生类析构函数排列无关
Constructor Bclass2   //第二个虚拟基类
Constructor Bclass1   //非虚拟基类
Constructor Object    //对象成员
派生类建立!
主程序运行!
派生类析构!
deconstructor Object //析构次序相反
deconstructor Bclass1
deconstructor Bclass2
deconstructor Bclass3
```

由实现复制构造函数说起……

```
//基类复制构造函数
A::A(A &a)
{
  m1=a.m1;
  m2=a.m2;
}

//派生类复制构造函数
B::B(B &b) :A(?)
{
  ……
}
```

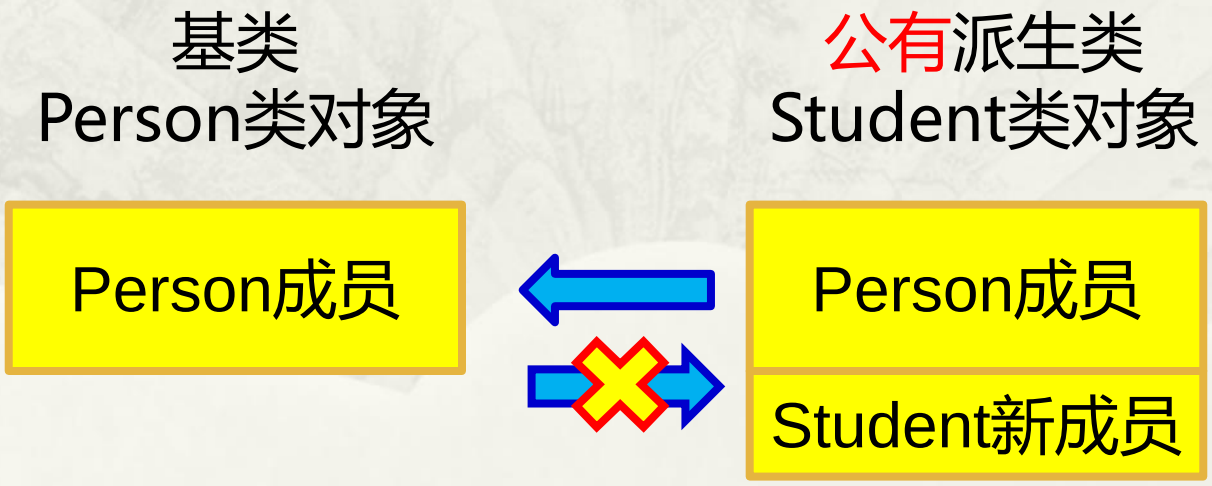
就是b!
为什么?

基类构造函数的实参是哪个?

- 似乎不是b。因为b类型是B，应该找类型是A的参数
- 但除了b，似乎又别无选择

公有派生与赋值兼容规则

- ◆公有派生类的对象
  - ◆属性/操作：派生类吸收基类所有成员（除构造/析构函数）
  - ◆访问限定：派生类对象访问（外部访问）基类成员，与基类对象一样
- ➔赋值兼容规则
  - ◆公有派生类对象可以代替基类对象使用



赋值兼容规则(1)

◆公有派生类的对象可以代替基类对象使用

- ◆ 派生类的对象可以赋值给基类对象
- ◆ 反之不行，即不能将基类对象赋值给派生类对象

```
Person p; //基类
Student s; //派生类
p=s;    //派生类对象赋值给基类对象，正确，赋值兼容
s=p;    //错误，因为Student派生类中的新成员无法根
        据基类的类对象进行初始化
```

赋值兼容规则(2)

◆公有派生类的对象可以代替基类对象使用

- ◆ 派生类对象的地址可以赋值给基类的指针，通过该基类指针访问基类成员，但不能访问派生类新成员
- ◆ 反之不行，即不能将基类对象地址赋值给派生类指针，通过该派生类指针访问

```
Student s;  //派生类
Person *p=&s; //基类指针，引用派生类对象地址
p→SetName(“朱明”); //基类成员函数SetName
 //正确，通过基类指针访问由基类继承的成员
p→SetCourse(“英语”,90)；//派生类成员函数SetCourse
 //错误，无法通过基类指针访问派生类新成员
```

赋值兼容规则(3)

- 公有派生类的对象可以代替基类对象使用
  - 派生类对象可用于初始化基类对象的引用，
  - 反之不行，即不能用基类对象初始化派生类对象的引用

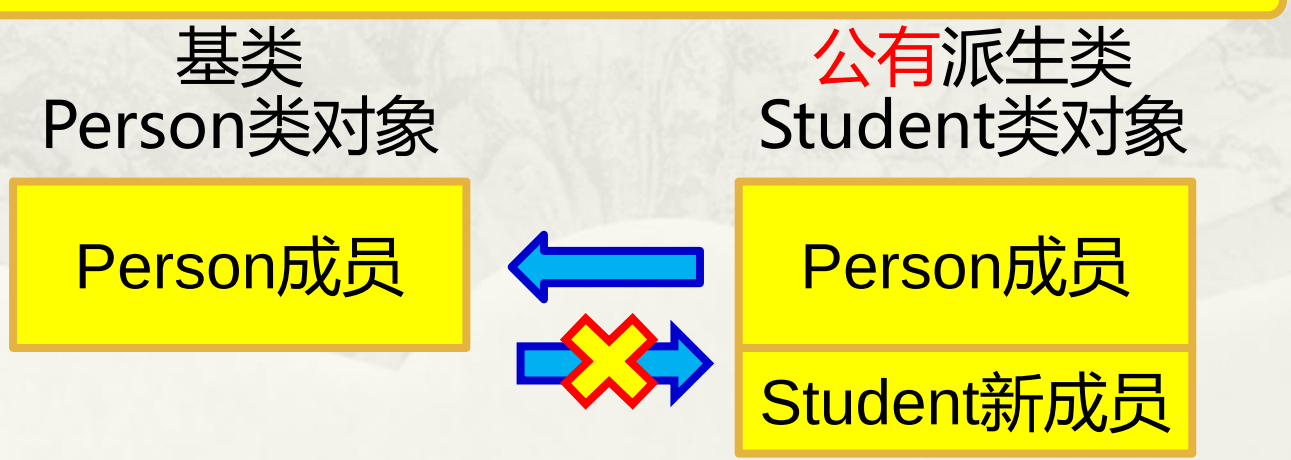
```
Person p;
Student s;  //派生类
void Myfunc(Person &a);
Myfunc(p);  //正确，引用调用
Myfunc(s);  //正确，赋值兼容，用派生类对象s初始化基类
            Person的引用
```

赋值兼容规则

◆公有派生类的对象可以代替基类对象使用

1. 派生类的对象可以赋值给基类对象，反之不行
2. 派生类的地址可以赋值给基类的指针，通过该指针访问基类成员，但不能访问派生类新成员，反之不行
3. 派生类对象可用于初始化基类对象的引用，反之不行

可以将派生类对象当作基类使用，反之不行



赋值兼容规则下的成员函数定义

- 复制构造函数
- 赋值操作符重载函数

赋值兼容规则下的复制构造函数

```
//基类复制构造函数
A::A(A &a){
  m1=a.m1;
  m2=a.m2;
}

//派生类复制构造函数
B::B(B &b):A(b){    调用基类复制构造函数
                    时，b是实参
  ......
}
```

定义派生类复制构造函数时，b是形参

为什么b能成为实参？

赋值兼容

赋值兼容规则下的赋值操作符重载函数

```
//基类赋值操作符重载函数
A & A::operator=(A &a){
  m1=a.m1;
  m2=a.m2;
  return *this;
}
```

定义派生类赋值操作符重载函数时，b是形参

```
//派生类赋值操作符重载函数
B & B::operator=(B &b){
  this->A::operator=(b);
  ......
  return *this;
}
```

调用基类赋值操作符重载函数时，b是实参

为什么b能成为实参？

赋值兼容

派生类 vs. 类模板

- 类模板
  - 独立性和通用性
  - 通过自动化设计提高编程效率，同时不影响运行效率
- 派生类
  - 封装性、扩展性和通用性
  - 通过继承的层次结构提高编程效率，但影响运行效率

多态性

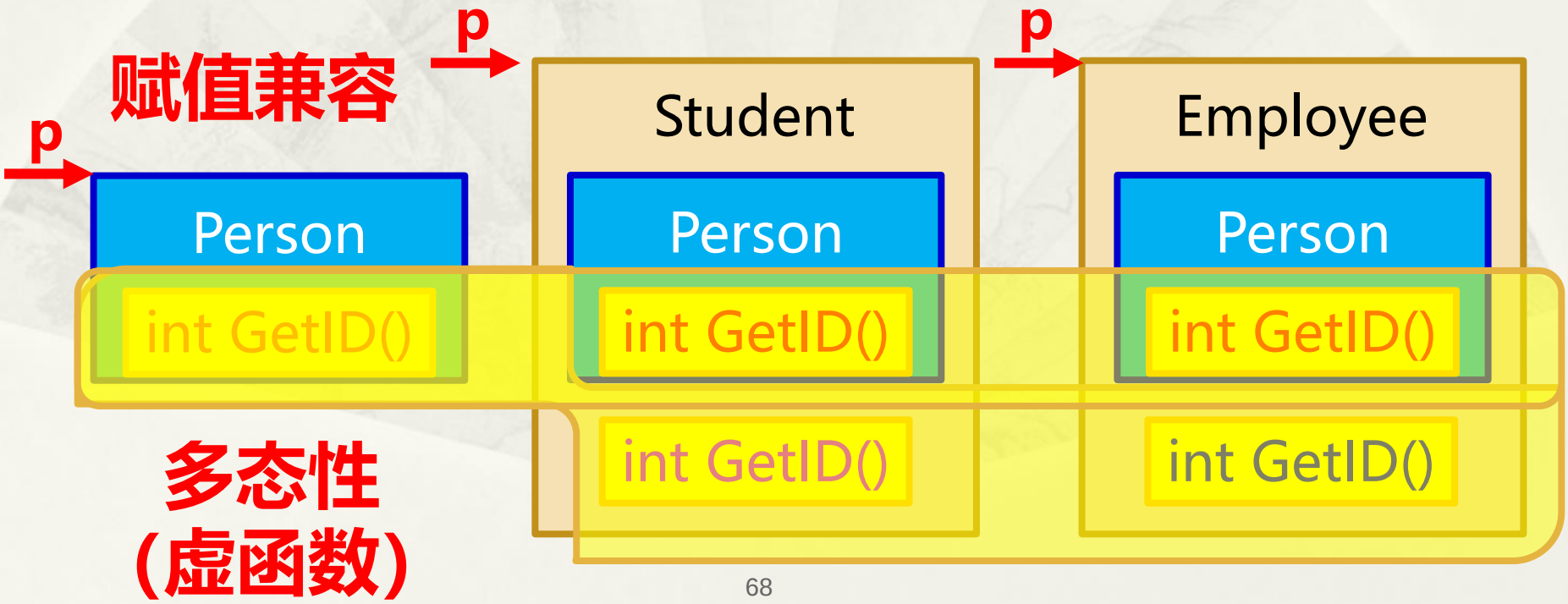
◆什么叫多态

◆接口相同（函数名相同，“开”）
◆功能不同（函数体不同，具体“开”的方式不同）

◆编译时的多态性（静态）
通过重载函数实现
◆运行时的多态性（动态）
通过虚函数(virtual function)实现

多态性

```
class Person {...};
class Student: Public Person {...};
class Employee: Public Person {...};
Person per, *p;     p=&per;     cout<<p->GetID();
Student stu;        p=&stu;     cout<<p->GetID();
Employee emp;       p=&emp;     cout<<p->GetID();
```

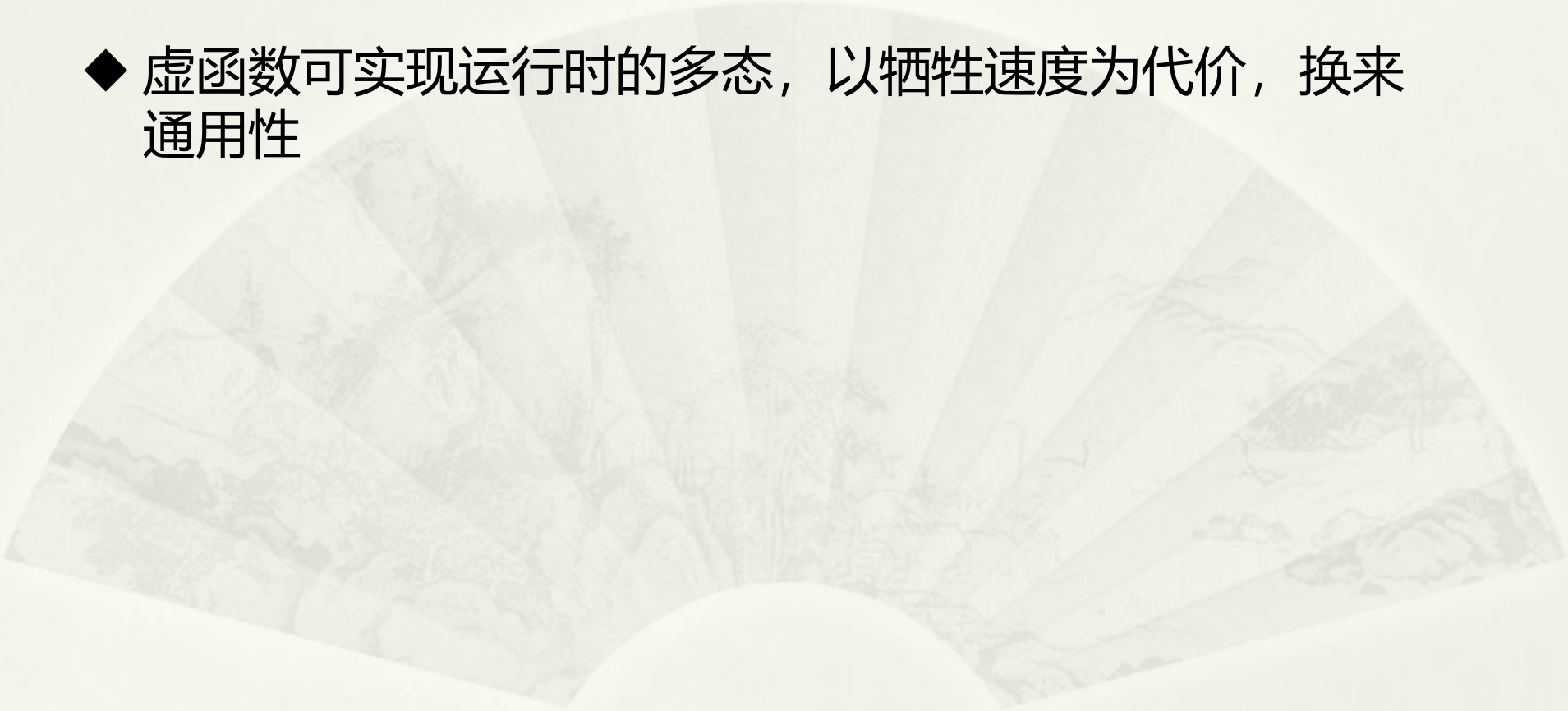


虚函数

- 派生类中可定义基类的覆盖(override)函数，
- 虚函数：一类特殊的覆盖函数
- 与普通被覆盖的函数相比
  - 相同点：都是函数名相同，参数相同，返回值相同
  - 不同点：虚函数可实现运行时的多态，普通的同名覆盖则不能
- 虚函数的例外：特殊情况下返回值可以不同
  - 基类中虚函数返回值是基类指针时，派生类中对应虚函数返回值可以是派生类的指针
- 虚函数的前提
  - 基类→派生类
  - 类的成员函数

虚函数

- 如果基类的某个成员函数是虚函数，则在其派生类，派生类的派生类……，该函数始终是虚函数
- 虚函数可实现运行时的多态，以牺牲速度为代价，换来通用性



虚函数的定义

virtual 返回类型 函数名（参数表）{...};

- 关键字virtual指明该成员函数为虚函数
- 只需要在基类中，虚函数定义/声明时加virtual
- 不用加virtual
  - 派生类的声明和定义时
  - 基类的虚函数在类外定义时

虚函数的使用

条件		调用基类or派生类定义的f()函数	
基类A 是否定义f()	公有派生类B 是否定义f()	派生类对象 b.f() 或派生类指针 pb->f()	基类指针访问 派生类对象a->f()
是	未定义f()	基类	基类
	定义同名函数f()	派生类	基类（赋值兼容）
是，虚函数	未定义f()	基类	基类
	定义同名函数f()	派生类	派生类（多态）

虚函数的使用

```
class A{
public:
    virtual void show(){cout<<"A";};
};

class B:public A{
public:
    void show(){cout<<"B";};
};
```

```
int main()
{
    A a; B b;
    A *pa; B *pb;
    pa=&a; pa->show();
    pa=&b; pa->show();
    pb=&b; pb->show();
    return 0;
}
```

1)　　2)

1) A B B
2) A A B
3) B A B
4) A B A

【例8.6】虚函数计算本科生和研究生学分

```
class Student{
    string coursename;              //课程名
    int classhour;                  //学时
    int credit;                     //学分
public:
    Student(){coursename="#";classhour=0;credit=0;}
    virtual void Calculate(){credit=classhour/16;}
    void SetCourse(string str,int hour){
        coursename=str;
        classhour=hour;
    }
    int GetHour(){return classhour;}
    void SetCredit(int cred){credit=cred;}
    void Print(){cout<<coursename<<'\t'<<classhour<<"学时
    "<<'\t'<<credit<<"学分"<<endl;}
};
```

虚函数

【例8.6】虚函数计算本科生和研究生学分

```
class GradeStudent:public Student{
public:
    GradeStudent(){};
    void Calculate(){SetCredit(GetHour()/20);}
};
```

虚函数的覆盖函数

【例8.6】虚函数计算本科生和研究生学分

```
int main(){
    Student s,*ps;
    GradeStudent g;
    s.SetCourse("物理",80); s.Calculate();  //80/16=5
    g.SetCourse("物理",80); g.Calculate();   //80/20=4
    cout<<"本科生：  "<<'\t'; s.Print();
    cout<<"研究生：  "<<'\t'; g.Print();
    s.SetCourse("数学",160);
    g.SetCourse("数学",160);
    ps=&s; ps->Calculate(); //160/16=10
    cout<<"本科生：  "<<'\t';
    ps->Print();
    ps=&g; ps->Calculate();  //160/20=8
    cout<<"研究生：  "<<'\t';
    ps->Print();
    return 0;
}
```

```
void Calfun(Student &,string,int);
```

```
本科生：物理 80学时 5学分
研究生：物理 80学时 4学分
本科生：数学 160学时 10学分
研究生：数学 160学时 8学分
```

【例8.7】虚函数计算本科生和研究生学分

```
void Calfun(Student &ps,string str,int hour){
    ps.SetCourse(str,hour);
    ps.Calculate();
    ps.Print();
}
int main(){
    Student s;
    GradeStudent g;
    cout<<"本科生:";
    Calfun(s,"物理",80);
    cout<<"研究生:";
    Calfun(g,"物理",80);
//派生类对象作为基类引用的实参，只有calculate()为虚函数才能实现
    动态的多态性
    return 0;
}
```

本科生：物理 80学时 5学分
研究生：物理 80学时 4学分

虚函数的使用

- 不能被定义为虚函数
  - 静态成员函数(static)：所有对象公有，不属于某个对象
  - 内联函数(inline)：每个对象独享函数代码
  - 构造函数：调用构造函数时类对象尚未完成实例化

- 通常被定义为虚函数
  - 析构函数：实现撤销对象时的多态性

```
A *pa; //A类派生出B1和B2类
pa=new B1();
delete pa;      //调用B1析构函数
pa=new B2();
delete pa;      //调用B2析构函数
```

纯虚函数

- 虚函数(virtual function)
  - 基类指针指向基类对象时，虚函数调用基类的
  - 基类指针指向派生类对象时，虚函数调用派生类的
- 纯虚函数(pure virtual function)
  - 基类指针只能指向派生类对象
- 抽象类(abstract class)
  - 含有纯虚函数的类是抽象类（只要有一个纯虚函数，就是抽象类）
  - 抽象类不能定义类对象，只能作为派生类的基类
  - 抽象类中的纯虚函数用于定义基类中无法定义的函数如：Person::CalMark() 可以对学生/教师考核，但是对“人”考核无意义

纯虚函数的定义

virtual 返回类型 函数名（参数表）{...}; =0;

virtual 返回类型 函数名（参数表）=0;

- =0表示基类中不定义该函数

纯虚函数与虚函数的区别

	基类成员函数f()定义为虚函数	基类成员函数f()定义为纯虚函数
派生类中 不定义覆盖函数f()	基类指针指向派生类对象 调用的f()为基类定义版本	派生类仍然为抽象类， 无法定义派生类对象
派生类中 定义覆盖函数f()	基类指针指向派生类对象 调用的f()为派生类定义版本，实现运行时多态	

纯虚函数的使用

- 基类：Person，CalMark无意义，定义为纯虚函数
- 派生类：Student，重新定义CalMark
- 派生类：Teacher，重新定义CalMark
- 使用
  - Person类指针指向Student类对象，调用对应CalMark
  - Person类指针指向Teacher类对象，调用对应CalMark

基类Person定义

```
class Person{
   int MarkAchieve;
   string  Name;
public:
   Person(string name){ Name=name; MarkAchieve=0;}
   void SetMark(int mark){MarkAchieve=mark;};
   void Print(){ cout<<Name<<MarkAchieve<<endl;}
   virtual void CalMark()=0; //纯虚函数,Person为抽象类
};
```

派生类Student/Teacher定义

```
class Student:public Person{
   int credit,grade;  //学分和成绩
public:
   Student(string name,int cred,int grad):Person(name)
   {credit=cred; grade=grad; }
   void CalMark()
   {SetMark(credit*grade); }  //重新定义纯虚函数
};
class Teacher:public Person{
   int credit,classhour,studnum;  //学分、授课学时、学生人数
public:
   Teacher(string name, int cred, int ch,int sn):Person(name)
   {credit=cred; classhour=ch; studnum=sn; }
   void CalMark()
   {SetMark(cred*classhour*studnum); }  //重新定义纯虚函数
};
```

主函数

```
int main(){
    Person *pp;
    Student s1("张成",2, 80);
    Teacher t1( “范英明” ,2, 64, 60);
    pp=&s1;
    pp->CalMark();  //计算学生成绩
    pp->Print();        //张成160=2*80
    pp=&t1;
    pp->CalMark();  //计算教师工作量
    pp->Print();        //范英明7680=2*64*60
    return 0;
}
```

可以定义抽象类(Person)指针，但不能定义抽象类的类对象

Person类指针pp指向Student类对象s1

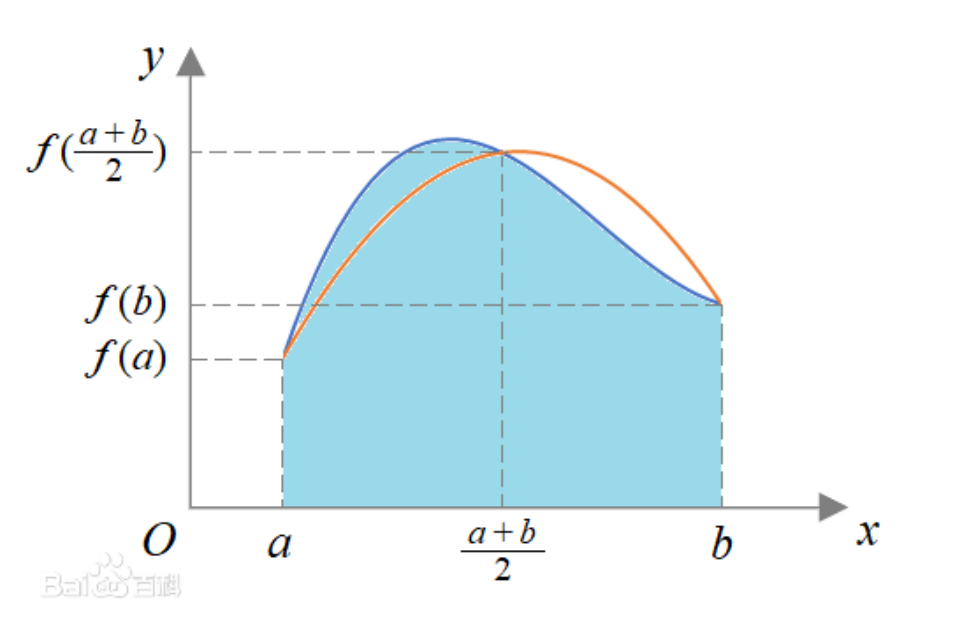
Person类指针pp指向Teacher类对象t1

张成160
范英明7680

【例8.9】虚函数实现辛普生法求函数定积分

设计辛普生法求函数的定积分的类模板，可求任意函数的定积分。在梯形法中是用直线来代替曲边梯形的曲边，在辛普生法中是用抛物线来代替曲边梯形的曲边，得出的公式如下（采用等区间分割，分区数量n为偶数）：

$$\int_a^b f(x)dx \approx \frac{1}{3}\Delta x\left[y_0 + y_n + 4\left(y_1 + y_3 + \ldots + y_{n-1}\right) + 2\left(y_2 + y_4 + \ldots + y_{n-2}\right)\right]$$



【例8.9】虚函数实现辛普生法求函数定积分

```
#include<iostream>
#include<cmath>
using namespace std;
class Simpson{
    double Intevalue,a,b; //Intevalue积分值，a积分下限，b积分上限
public:
    virtual double fun(double x)=0;  //被积函数声明为纯虚函数
    Simpson(double ra=0,double rb=0){
        a=ra;
        b=rb;
        Intevalue=0;
    }
    void Print(){cout<<"积分值="<<Intevalue<<endl;}
```

【例8.9】虚函数实现辛普生法求函数定积分

```
    void Integrate(){
        double dx;
        int i;
        dx=(b-a)/2000;
        Intevalue=fun(a)+fun(b);
        for(i=1;i<2000;i+=2) Intevalue+=4*fun(a+dx*i);
        for(i=2;i<2000;i+=2) Intevalue+=2*fun(a+dx*i);
        Intevalue*=dx/3;
    }
};
```

【例8.9】虚函数实现辛普生法求函数定积分

```
class A:public Simpson{
public:
    A(double ra,double rb):Simpson(ra,rb){};
    double fun(double x){return sin(x);}
};

class B:public Simpson{
public:
    B(double ra,double rb):Simpson(ra,rb){};
    double fun(double x){return exp(x);}
};
```

【例8.9】虚函数实现辛普生法求函数定积分

```
int main(){
    A a1(0.0,3.1415926535/2.0);
    Simpson *s=&a1;
    s->Integrate();//动态
    B b1(0.0,1.0);
    b1.Integrate();//静态
    s->Print();
    b1.Print();
    return 0;
}
```

积分值=1
积分值=1.71828

本章小结(1)

- 继承性（8.1, 8.2, 8.5节）
  - 概念：基类vs.派生类；父类/超类vs.子类
  - 类继承层次结构：直接基类（单继承/多重继承）/间接基类
  - 派生过程：吸收→（改造）→扩展→重写
  - 派生类访问限定
    - 公有/私有/保护
    - 派生类成员函数访问基类成员（内部访问）
    - 派生类对象访问基类成员（外部访问）
  - 构造函数/析构函数定义格式
  - 多重继承/虚基类
  - 继承与聚合
  - 赋值兼容规则
    - 内涵：派生类当基类用（兼容），反之不行
    - 三种兼容场景：类对象赋值；指针访问；初始化引用
    - 应用 ： 复制构造函数；赋值操作符重载函数

本章小结(2)

- 多态性（8.6节）
  - 概念：接口相同，功能不同（基类与多个派生类间）
  - 实现机制：虚函数
    - 特殊的覆盖函数
    - 与普通覆盖函数的异同
    - 多态性与赋值兼容规则的不同
  - 更进一步：纯虚函数
    - 抽象类
    - 与虚函数的异同

End

虚函数的使用

- 背景
  - 派生类由基类公有派生而来
  - 派生类定义了基类中成员函数的同名覆盖函数
  - 由同一个基类指针指向不同派生类的对象，并调用该函数
- 问题
  - 此时调用的哪个函数版本（基类/派生类）？

情况1：该函数不是虚函数

- 调用版本：基类
- 原因：赋值兼容规则

情况2：该函数是虚函数

- 调用版本：派生类
- 原因：多态性