



中国矿业大学(北京)
China University of Mining and Technology (Beijing)

面向对象程序设计与应用

第五章 继承

授课教师：张潇

机电与信息工程学院


计算机系






第五章 继承


继承使软件复用变得简单、易行，可以通过继承复用已有的程序资源，缩短软件开发的周期。本章主要介绍：




继承的概念、继承的方式



基类与派生类（对象）的关系



构造函数、析构函数

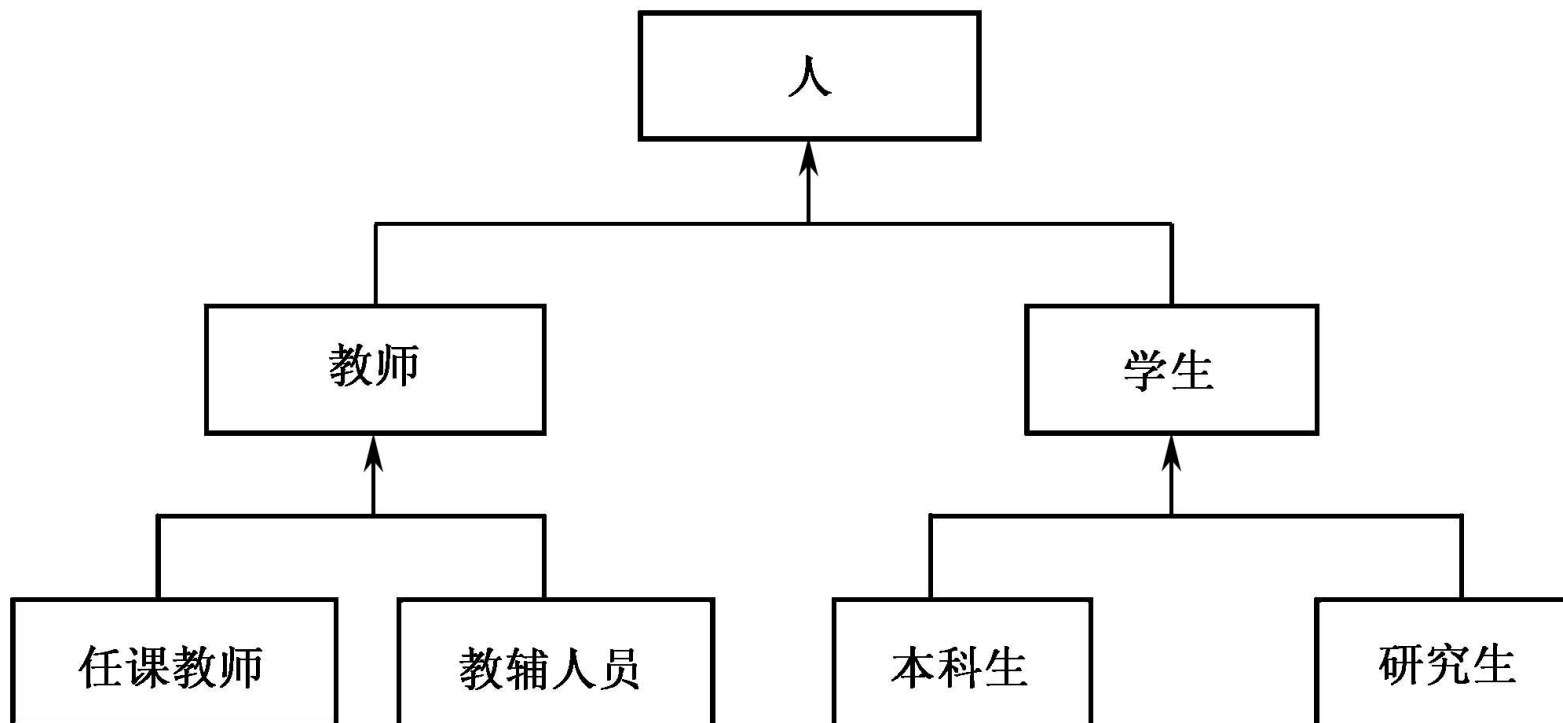


多重继承、虚拟继承



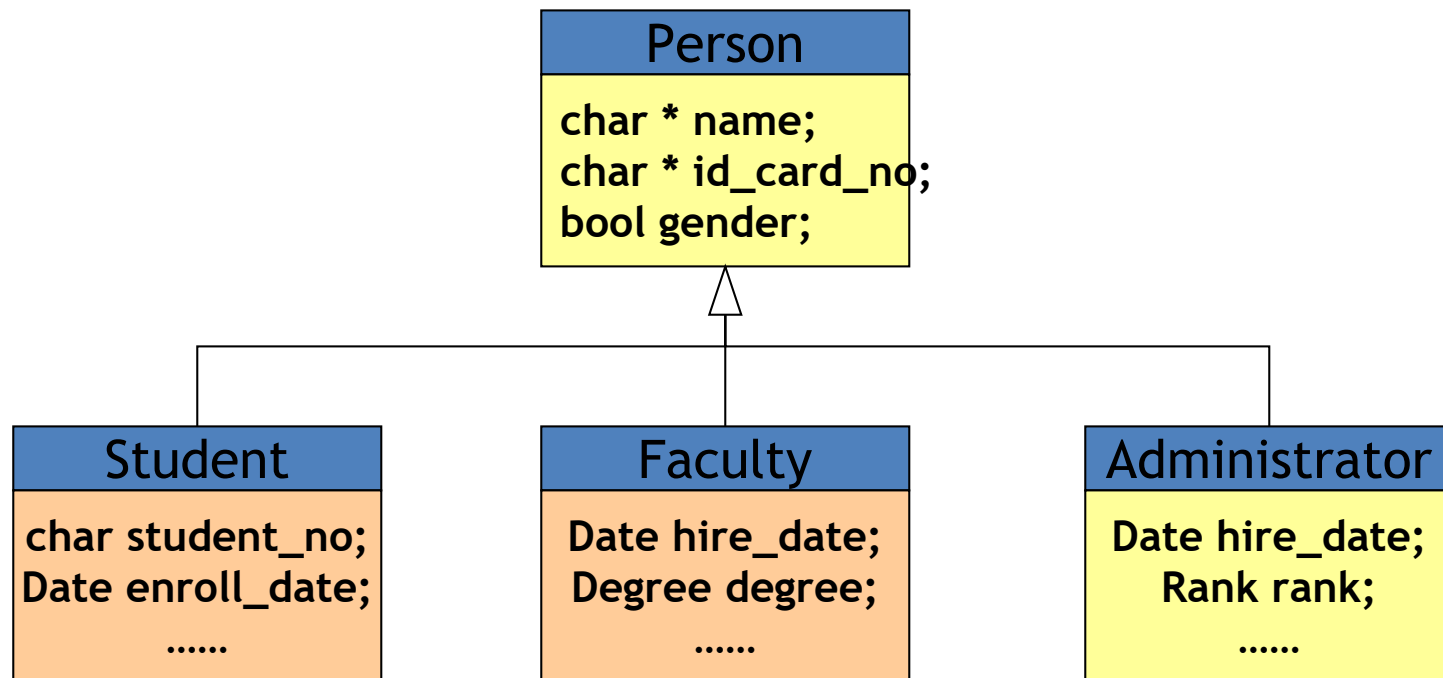
5.1 继承的概念

1、继承的概念





- 以存在的类为基础定义新的类，新类即拥有基类的
数据成员和成员函数。





2、继承目的

- 代码重用code reuse
- 描述能力：类属关系广泛存在
- Is-A vs. Has-A

3、派生的目的

- 当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。



4、派生类可实施的对基类的改变

- 增加新的数据成员和成员函数。
- 重载基类的成员函数。
- 重定义基类已有的成员函数。
- 改变基类成员在派生类中的访问属性。

5、派生类不能继承基类的以下内容

- 基类的构造函数和析构函数。
- 基类的友元函数。
- 静态数据成员和静态成员函数



- 继承语法形式

```
class B {.....};
```

```
class D : [private | protected | public] B
```

```
{
```

```
.....
```

```
};
```

派生类对象

基类子对象
派生类新定义成员

继承部分

派生部分



5.2.1 继承方式

- **C++的继承类型**

可分为公有继承、保护继承和私有继承，也称为公有派生、保护派生和私有派生。

不同继承方式会不同程度地影响基类成员在派生类中的访问权限。



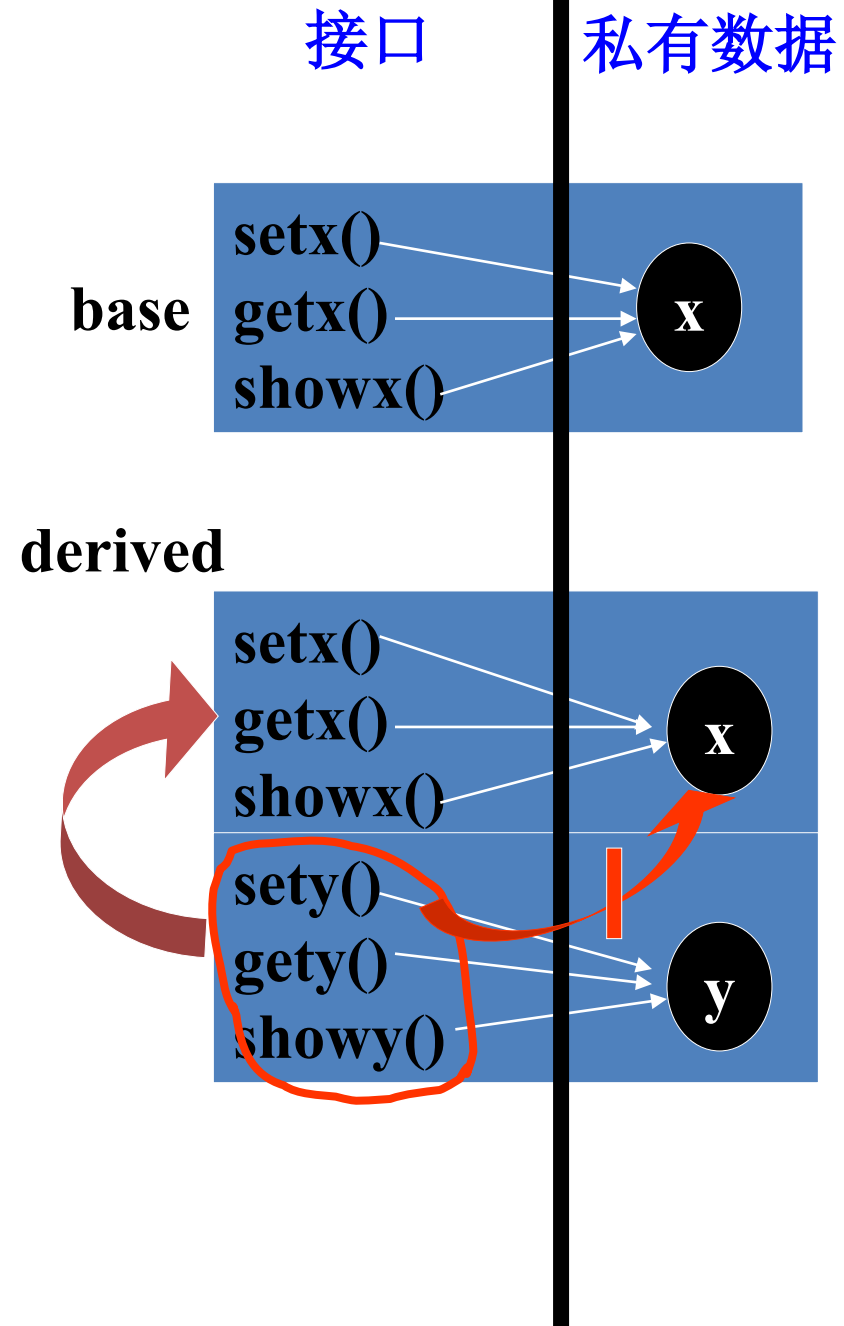
5.2.2 公有继承

- public
 - 最常见的派生方式
 - 维持基类成员的可访问性，公有继承不改变基类成员在派生类中的访问权限
 - 派生类中的成员函数不可直接访问基类的private成员，可以直接访问基类中的public和protected成员
 - 通过派生类的对象只能访问基类的public成员

- 例题ch5_1.cpp

```
class base{  
    int x;  
public:  
    void setx(int n){    x=n;    }  
    int getx(){    return x;    }  
    void showx()        {  
        cout<<x<<endl;    }  
};
```

```
class derived:public base{  
    int y;  
public:  
    void sety(int n){    y=n;    }  
    void gety(){ y=getx();    }  
    void showy()  
    {    cout<<y<<endl;    }  
};
```





```
int main()
{
    derived obj;
    obj.setx(10); //从base继承
    obj.sety(20);
    obj.getx(); //从base继承
    obj.gety();
    obj.showx(); //从base继承
    obj.showy();
}
```



5.2.3 私有继承

- **private**
 - 基类中的public和protected的成员在派生类中会变成private, private成员在派生类中不可访问。
 - 派生类中的成员函数**不可直接访问**基类的private成员, 可以直接访问基类中的public和protected成员
 - 通过派生类的对象**不能访问基类的任何成员**



【例】 私有继承的例子

```
#include <iostream>
using namespace std;
class Base{
    int x;
public:
    void setx(int n){x=n; }
    int getx(){return x; }
    void showx(){cout<<x<<endl; }
};
```

```

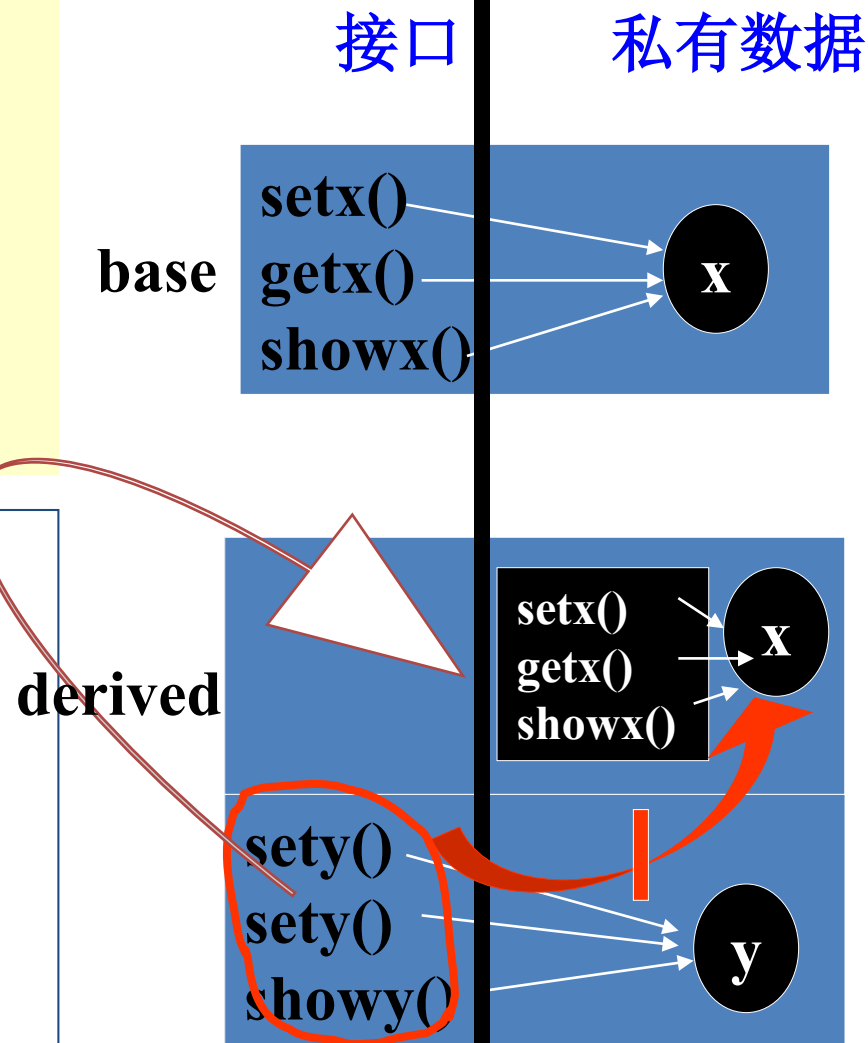
class derived:private base{
    int y;
public:
    void sety(int n){y=n;    }
    void gety(){ y=getx();  }
    void showy()    {
        cout<<y<<endl;  }
};

```

```

int main(){
    derived obj;
    obj.setx(10); // cannot access
    obj.sety(20);
    obj.showx(); // cannot access
    obj.showy();
}

```





5.2.4 保护继承

- 派生方式为protected的继承称为保护继承，在这种继承方式下，基类的public成员在派生类中会变成protected成员，基类的protected和private成员在派生类中保持原来的访问权限。基类的private成员不可访问。
- 派生类中的成员函数不可直接访问基类的private成员，可以直接访问基类中的public和protected成员
- 通过派生类的对象不能访问基类的任何成员



【例】 保护继承的例子

```
#include <iostream>
using namespace std;
class Base{
    int x;
protected:
    int getx(){ return x; }
public:
    void setx(int n){ x=n; }
    void showx(){ cout<<x<<endl; }
};
```



```

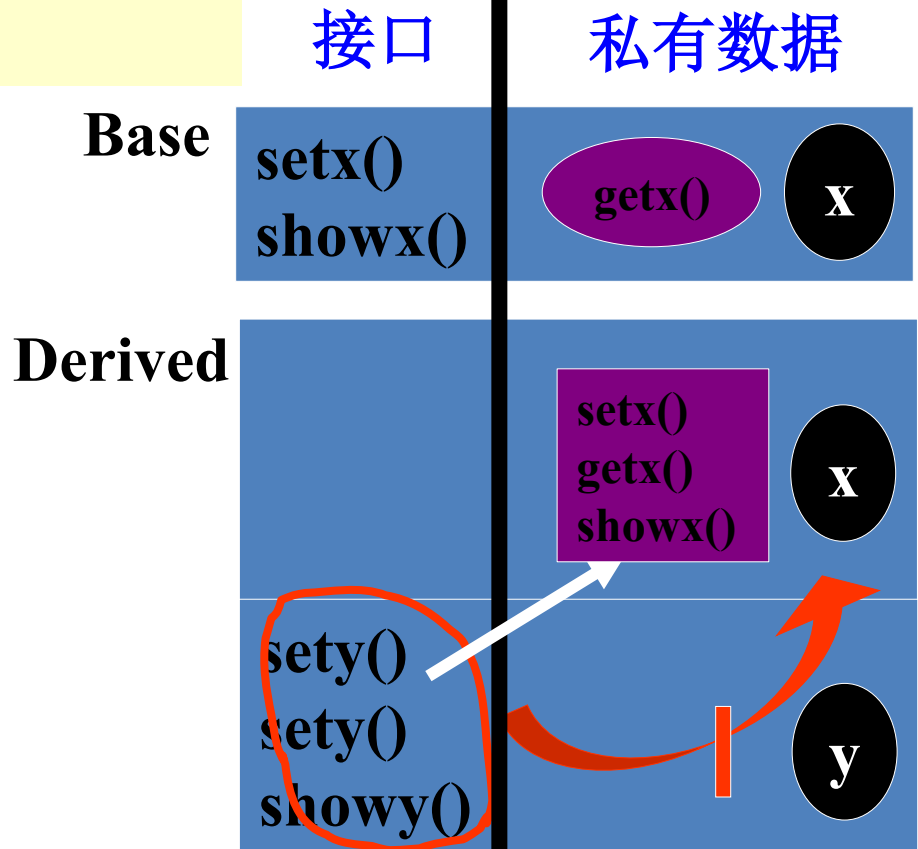
class Derived:protected Base{
    int y;
public:
    void sety(int n){ y=n; }
    void gety(){ y=getx();}  //访问基类的保护成员
    void showy(){ cout<<y<<endl; }
};

```

```

int main(){
    Derived obj;
    obj.setx(10); //错误
    obj.sety(20);
    obj.showx(); //错误
    obj.showy();
}

```





基类成员在派生类中的访问权限

派生类 基类	public继承			protected继承			private继承		
	public	protected	private	public	protected	private	public	protected	private
public	√				√				√
protected		√			√				√
private			√			√			√



例1、如果有，下面声明中哪些是错误的？

- `class Base { ... };`
- `(a) class Derived : public Derived { ... };`
- `(b) class Derived : Base { ... };`
- `(c) class Derived : private Base { ... };`
- `(d) class Derived : public Base;`
- `(e) class Derived inherits Base { ... };`



例2、分析程序中的访问权限,并回答所提的问题。

```
#include <iostream>
class A
{
    public:
        void f1();
    protected:
        int j1;
    private:
        int i1;
};

void main()
{
    B b1;
    C c1;
}
```

```
class B:private A
{
    public:
        void f2();
    protected :
        int j2;
    private:
        int i2;
};
class C:private B
{
    public:
        void f3();
};
```

.



例3、分析下列程序,并回答所提的问题。

```
#include <iostream>
```

```
class A
```

```
{
```

```
    public:
```

```
        void f(int i) {cout<<i<<endl;}
```

```
        void g() {cout<<"g\n";}
```

```
};
```

```
class B:A
```

//缺省的继承方式表示为private

```
{
```

```
    public:
```

```
        void h() {cout<<"h\n";}
```

```
        A::f;
```

//将基类中的公有成员说明为派生类的公有成员

```
};
```

```
void main()
```

```
{
```

```
    B d1;
```

```
    d1.f(6);
```

```
    d1.g();
```

```
    d1.h();
```

```
}
```

• • • • •



5.2.5 保护成员

● 基类中protected的成员

- 类内部：可以访问
- 类的对象：不能访问
- 类的派生类内部：三种继承方式都可以访问



5.3 基类与派生类的对应关系

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



5.3.1 成员函数的重定义和名字隐藏

- **同名覆盖原则：当派生类与基类中有相同成员时：**
- **若未强行指名，则通过派生类对象使用的是派生类中的同名成员。**
- **如要通过派生类对象访问基类中被覆盖的同名成员，应使用基类名限定。**
- **派生类对基类成员函数的重定义或重载会影响基类成员函数在派生类中的可见性，基类的同名成员函数会被派生类重载的同名函数所隐藏。**



5.3.2 访问基类成员

- **派生类对象组成**
 - 一部分是由基类继承而来的成员构成的基类子对象，一部分是派生类新定义的成员构造的派生子对象。
- **在派生类中可以直接访问由继承得到的基类子对象的public和protected成员。**



- **派生类对基类成员的访问方式：**
 - **通过派生类对象直接访问**
 - **public继承方式下，派生类对象可以直接访问基类的public成员**
 - **在派生类成员函数中直接访问基类成员**
 - **在三种继承方式下，派生类的成员函数可以访问基类的public和protected成员**
 - **通过基类名字限定访问被隐藏的基类成员**
 - **在派生类中被重载或重定义的基类成员，只有通过基类的类名限定才能访问**



5.4.1 派生类构造函数的定义

- 派生类可能有多个基类，也可能包括多个成员对象，在创建派生类对象时，派生类的构造函数除了要负责本类成员的初始化外，**还要调用基类和成员对象的构造函数，并向它们传递参数，以完成基类子对象和成员对象的建立和初始化。**



- **单一继承时的构造函数**

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

{

本类成员初始化赋值语句;

};



【例5-5】 派生类Derived以构造函数初始化列表的方式向基类构造函数提供参数。

```
#include <iostream>
using namespace std;
class Base{
private:
```

```
    int x;
```

```
public:
```

```
    Base(int a){x=a;}
```

```
};
```

```
Class Derived:public Base{
```

```
    int y;
```

```
Public:
```

```
    Derived(int a,int b):Base(a){y=b;}
```

```
};
```

.



- 多继承时的构造函数

派生类名::派生类名(基类1形参, 基类2形参,
...基类n形参, 本类形参):基类名1(参数), 基
类名2(参数), ...基类名n(参数)

{

 本类成员初始化赋值语句;

};



- **多继承且有内嵌对象时的构造函数**

**派生类名::派生类名(基类1形参, 基类2形参,
...基类n形参, 本类形参):基类名1(参数), 基
类名2(参数), ...基类名n(参数), 对象数据成
员的初始化**

{

本类成员初始化赋值语句;

};



5.4.2 构造函数和析构函数调用次序

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

基类构造顺序由派生层次决定：最远的基类最先构造

析构函数的析构顺序与构造相反



例：下列程序输出结果。

```
#include<iostream>
using namespace std;
class A {
public:
A() { cout << "construct A" << endl; }
~A() { cout << "destrcut A" << endl; }
};
class B:public A {
public:
B() { cout << "construct B" << endl; }
~B() { cout << "destrcut B" << endl; }
};
```

```
class C:public B {
public:
C() { cout << "construct C" << endl; }
~C() { cout << "destrcut C" << endl; }
B b;
A a;
};
void main() {
C c;
}
```



5.4.3 构造函数和析构函数的构造规则

1、派生类可以不定义构造函数的情况

当具有下述情况之一时，派生类可以不定义构造函数。

- 基类没有定义任何构造函数。
- 基类具有缺省参数的构造函数。
- 基类具有无参构造函数。



【例5-7】 没有构造函数的派生类。

```
#include <iostream>
using namespace std;
class A {
public:
    A(){ cout<<"Constructing A"<<endl; }
    ~A(){ cout<<"Destructing A"<<endl; }
};
class B:public A {
public:
    ~B(){ cout<<"Destructing B"<<endl; }
};
int main(){
    B b;
}
```



2、派生类必须定义构造函数的情况

当基类或成员对象所属类只含有带参数的构造函数时，即使派生类本身没有数据成员要初始化，它也必须定义构造函数，并以构造函数初始化列表的方式向基类和成员对象的构造函数传递参数，以实现基类子对象和成员对象的初始化。



【例5-8】 派生类构造函数的定义。

```
#include <iostream>
using namespace std;
class Point{
protected:
    int x,y;
public:
    Point(int a,int b=0) {x=a; y=b;}
};
class Line:public Point{
public:
    Line(int a, int b):Point(a,b){};
};
Void main(){Line l1;
```



3、派生类的构造函数只负责直接基类的初始化

- C++语言标准有一条规则：如果派生类的基类同时也是另外一个类的派生类，则每个派生类只负责它的直接基类的构造函数调用。
- 这条规则表明当派生类的直接基类只有带参数的构造函数，但没有默认构造函数时（包括缺省参数和无参构造函数），它必须在构造函数的初始化列表中调用其直接基类的构造函数，并向基类的构造函数传递参数，以实现派生类对象中的基类子对象的初始化。



4、构造函数的调用时间和次序

- 当派生类具有多个基类和多个对象成员，用初始化列表进行初始化，它们的构造函数将在创建派生类对象时被调用，调用次序如下：

基类构造函数→对象成员构造函数→派生类构造函数
(继承顺序) (声明顺序)



● 例：#include<iostream>

```
Class A{int i;
```

```
public: A(int a){a=i;} };
```

```
Class B:public A{int j;
```

```
public: B(int b){b=j;} };
```

```
Class C:public A{int k;};
```

```
Class D: public B, protected A{
```

```
public: C c1, c2;
```

```
D(int l, int m): A(l), B(m), c2(),c1(){} };
```

```
void main(){D d;}
```

.



- 例：对于下面的基类定义

```
class Base {  
    Base(int val) : id(val) {}  
protected:  
    int id;  
}
```

解释为什么下述每个构造函数是非法的。

```
(1) class C1 : public Base {  
    C1(int val) : id(val) {}  
};
```

• • • • •



```
(2)class C2 : public C1 {  
    C2(int val) : Base(val), C1(val) {}  
};
```

```
(3)class C3 : public C1{  
    C3(int val) : Base(val) {}  
};
```

```
(4)class C4 : public Base {  
    C4(int val) : Base(id+val) {}  
};
```

```
(5)class C5 : public Base {  
    C5() {}  
};
```



5.5 多重继承

- C++允许一个类从一个或多个基类派生。如果一个类只有一个基类，就称为单一继承。如果一个类具有两个或两个以上的基类，就称为多重继承。多重继承的形式如下：

class 派生类名:[继承方式] 基类名1,[继承方式] 基类名2,

...

{

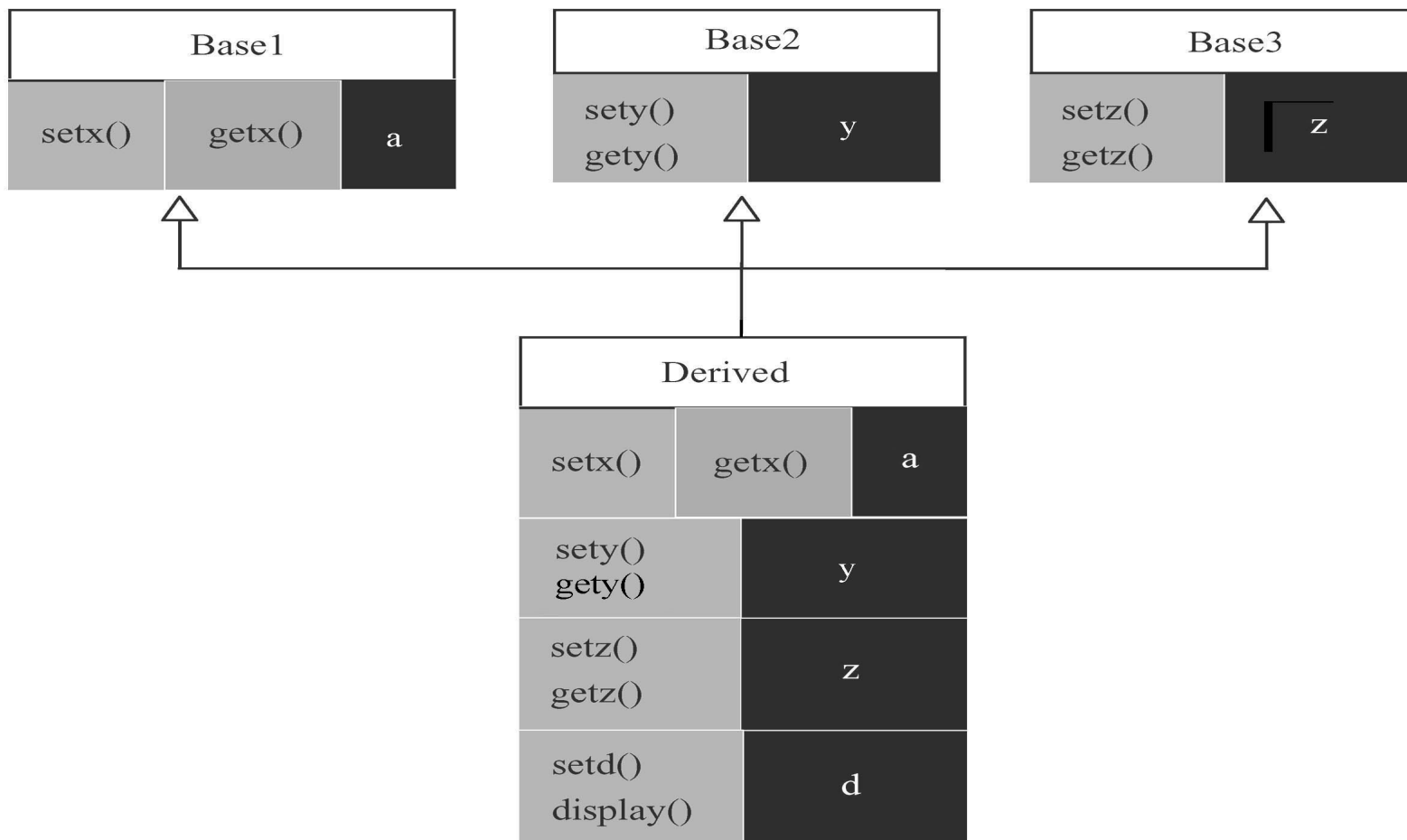
.....

};

- 其中，继承方式可以是public、protected、private



5.5.1 多继承的概念和应用





5.5.3 多继承的构造函数与析构函数

- 派生类必须负责为每个基类的构造函数提供初始化参数，构造的方法和原则与单继承相同。
- 构造函数的调用次序仍然是先基类，再对象成员，然后才是派生类的构造函数。
- 基类构造函数的调用次序与它们在被继承时的声明次序相同，与它们在派生类构造函数的初始化列表中的次序没有关系。
- 多继承方式下的析构函数调用次序仍然与构造函数的调用次序相反。



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

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

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

A->B->C->X->Y->Z->MI

```
class X {...};
```

```
class Y {...};
```

```
class Z:public X,public Y {...};
```

```
class MI :public C,public Z {...};
```

对于下面的定义，构造函数的执行次序是什么？

```
MI mi;
```

• • • • •



5.5.2 多继承下的二义性

- 在多继承时，基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用支配（同名覆盖）原则来解决。
- 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚拟继承来解决。



二义性问题举例

```
class A
{
    public:
        void f ( ) ;
};
class B
{
    public:
        void f ( ) ;
        void g ( ) ;
};
```

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

如果声明：C c1;

则 c1.f () ; 具有二义性

而 c1.g () ; 无二义性 (同名覆盖)



二义性的解决方法

- 解决方法一：用类名来限定

c1.A::f () 或 c1.B::f ()

- 解决方法二：同名覆盖

在C 中声明一个同名成员函数f () , f ()

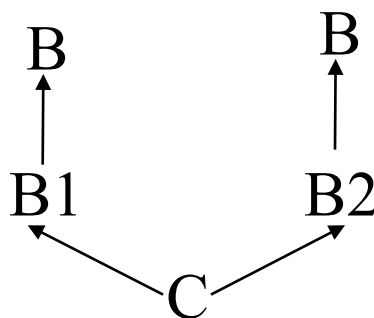
再根据需要调用 A::f () 或 B::f ()



二义性问题举例

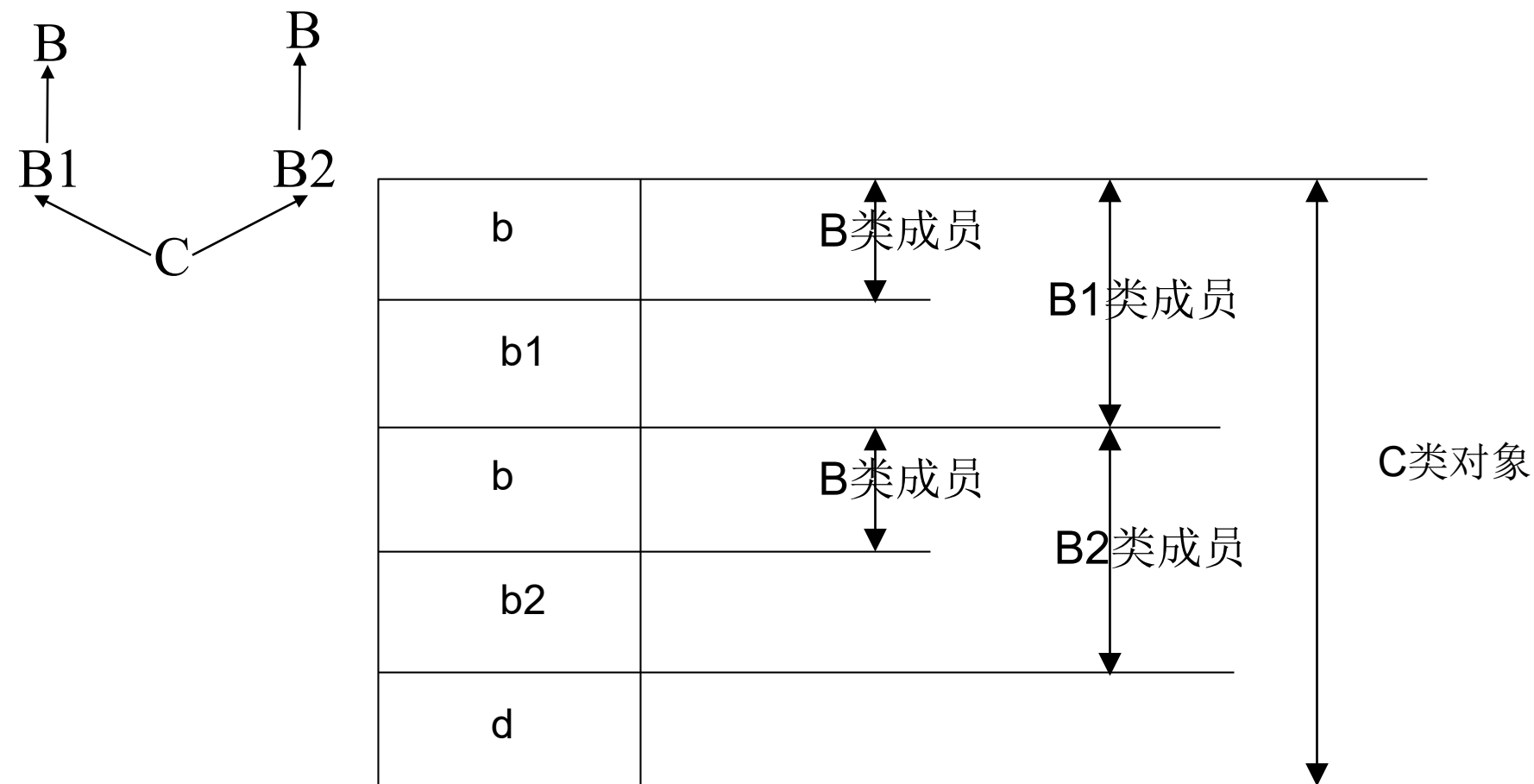
```
class B
{
    public:
        int b;
}
class B1 : public B
{
    public:
        int b1;
}
class B2 : public B
{
    public:
        int b2;
};
```

```
class C : public B1, public B2
{
    public:
        int f ( ) ;
    private:
        int d;
}
```





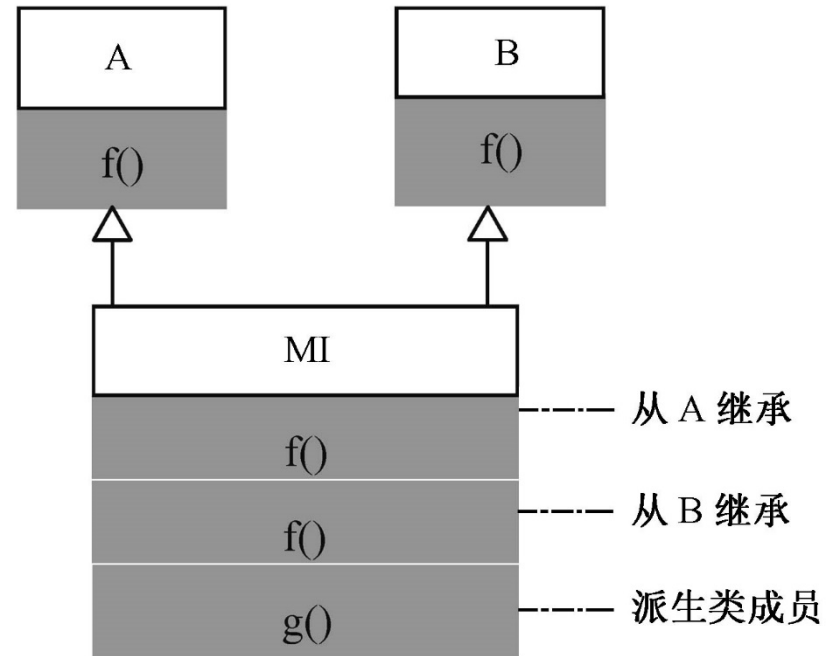
派生类C的对象的存储结构示意图：





【例5-12】 类A和类B是MI的基类，它们都有一个成员函数f，在类MI中就有通过继承而来的两个同名成员函数f。

```
#include<iostream>
using namespace std;
class A {
public:
void f(){ cout<<"From A"<<endl;}
};
class B {
public:
void f() { cout<<"From B"<<endl;}
};
class MI: public A, public B {
public:
void g(){ cout<<"From MI"<<endl;}
};
void main(){
    MI mi;
    mi.f();
    mi.A::f();
}
```



//错误

//正确

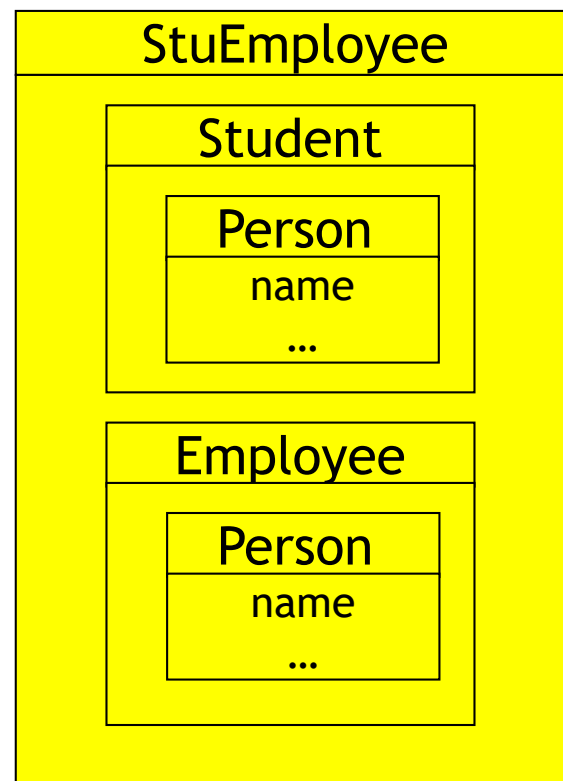
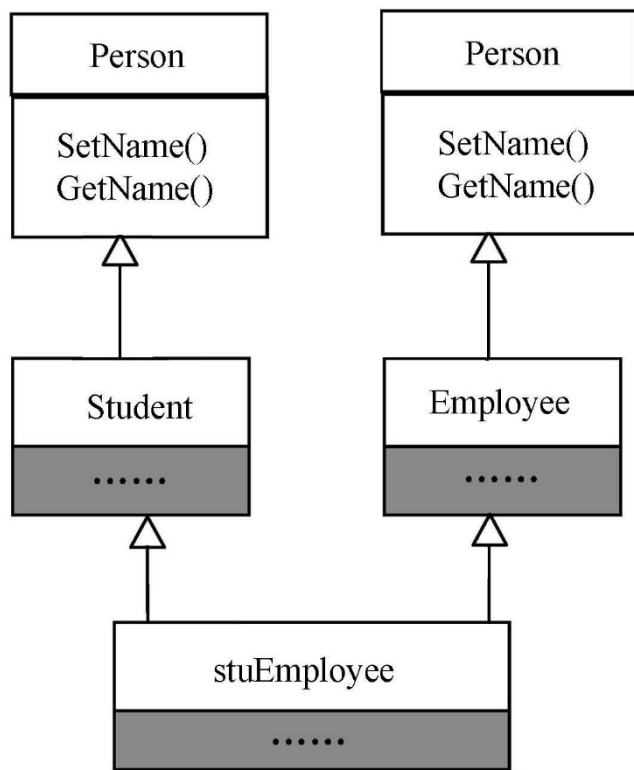




5.6 虚拟继承

5.6.1 引入的原因--重复基类

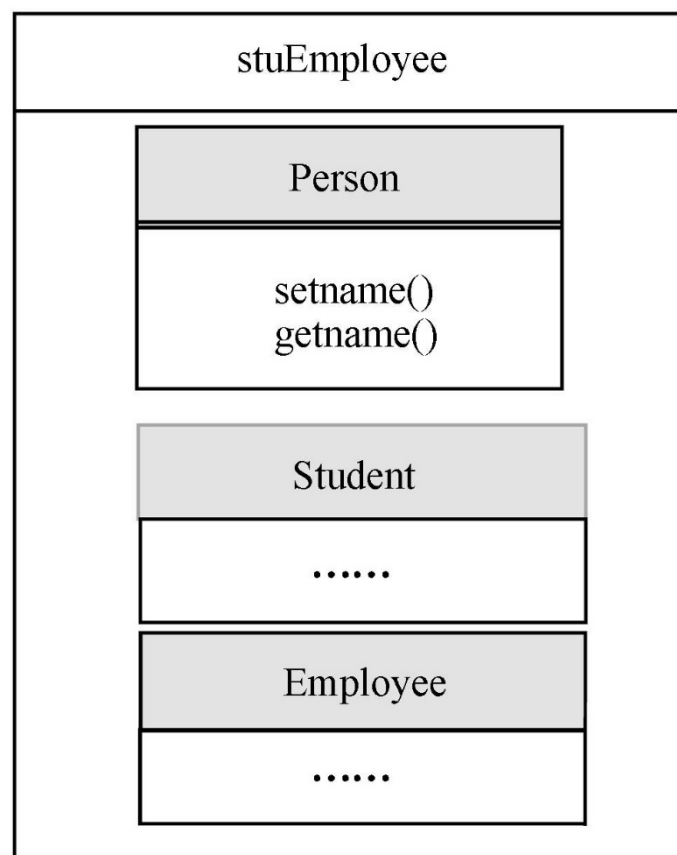
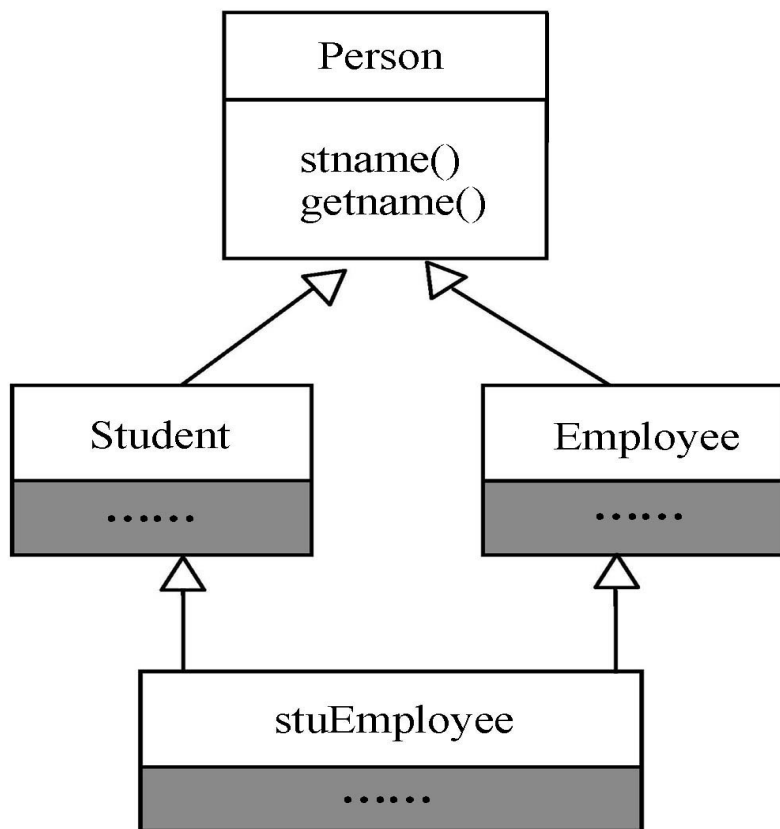
派生类间接继承同一基类使得间接基类 (Person) 在派生类中有多份拷贝，引发二义性。





5.6.1 引入的原因—重复基类

- 虚拟基类在派生类中只存在一份拷贝，解决了基类数据成员的二义性问题





5.6.2 虚拟继承的实现

1、虚拟继承virtual inheritance的定义

— 语法

- `class derived_class : virtual [...] base_class`

— 虚基类virtual base class

- 被虚拟继承的基类
- 在其所有的派生类中，仅出现一次

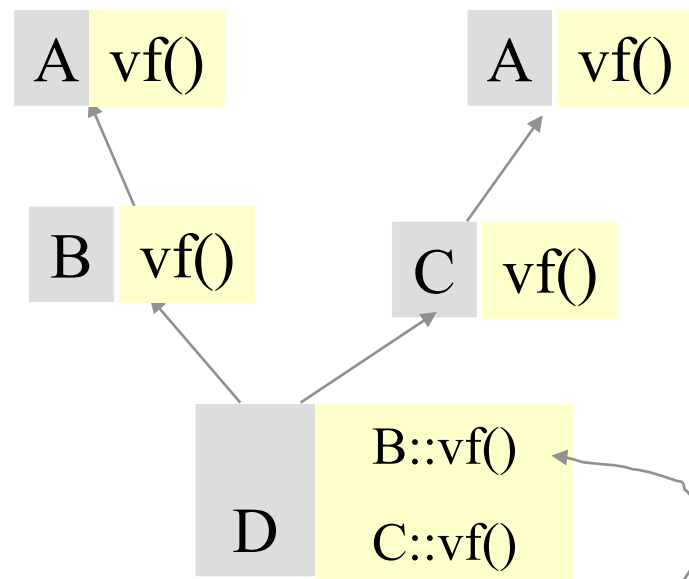


【例5-14】 类A是类B、C的基类，类D从类B、C继承，在类D中调用基类A的成员会产生二义性。

```
class A {  
public:  
    void vf() {  
        cout<<"I come from class A"<<endl;}  
};  
class B: public A{};  
class C: public A{};  
class D: public B, public C{};
```

```
int main()  
{  
    D d;  
    d.vf ();  
}
```

// error



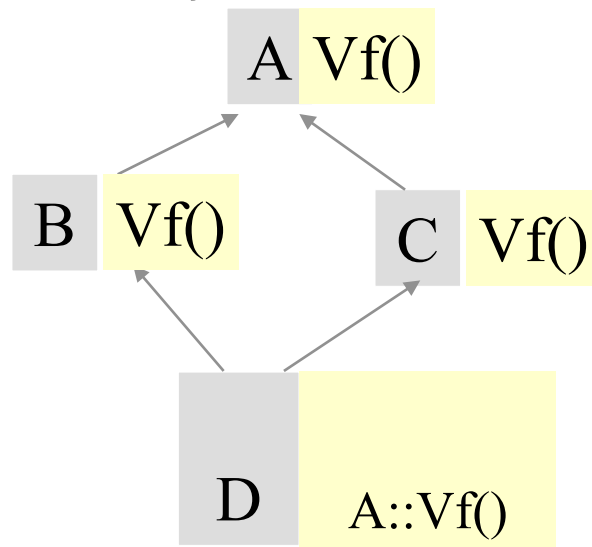


将【例5-14】 改为虚拟继承不会产生二义性。

```
class A {  
public:  
    void vf() {  
        cout<<"I come from class A"<<endl;    }  
};
```

```
class B: virtual public A{};  
class C: virtual public A{};  
class D: public B, public C{};
```

```
int main()  
{  
    D d;  
    d.vf();    // okay  
}
```





2、虚拟继承的构造次序

- 虚基类的初始化与一般的多重继承的初始化在语法上是一样的，但构造函数的调用顺序不同；
- 若基类由虚基类派生而来，则派生类必须提供对间接基类的构造（即在构造函数初始列表中构造虚基类，**无论此虚基类是直接还是间接基类**）
- 调用顺序的规定：
 - **先调用虚基类的构造函数，再调用非虚基类的构造函数**
 - 若同一层次中包含多个虚基类，这些虚基类的构造函数按它们的说明的次序调用
 - **若虚基类由非虚基类派生而来，则仍然先调用基类构造函数，再调用派生类构造函数**



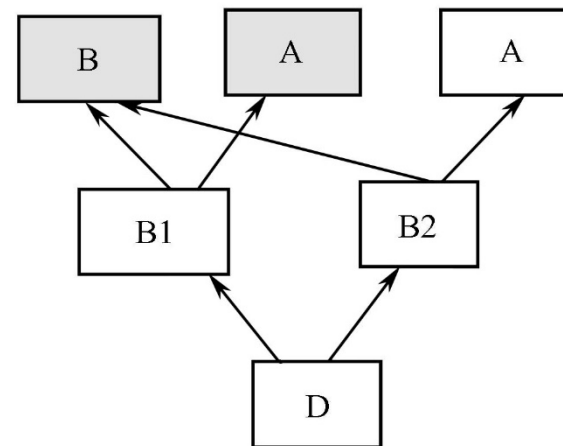
【例5-15】 虚基类的执行次序分析。

```
#include <iostream>
using namespace std;
class A {
    int a;
public:
    A(){ cout<<"Constructing A"<<endl; }
};
class B {
    int b;
public:
    B(){ cout<<"Constructing B"<<endl;}
    B(int i){b=i;}
};
```



```
class B1:virtual public B ,virtual public A{
public:
    B1(int i){ cout<<"Constructing B1"<<endl; }
};
class B2:public A,virtual public B {
public:
    B2(int j){ cout<<"Constructing B2"<<endl; }
};
class D: public B1, public B2 {
public:
    D(int m,int n): B1(m),B2(n){ cout<<"Constructing D"<<endl; }
    A a;
};

int main(){
    D d(1,2);
}
```



Constructing B
Constructing A
Constructing B1
Constructing A
Constructing B2
Constructing A
Constructing D



3、**虚基类由最终派生类初始化**

- 在没有虚拟继承的情况下，每个派生类的构造函数只负责其直接基类的初始化。但在虚拟继承方式下，**虚基类则由最终派生类的构造函数负责初始化。**
- 在虚拟继承方式下，若最终派生类的构造函数没有明确调用虚基类的构造函数，编译器就会尝试调用虚基类不需要参数的构造函数（包括缺省、无参和缺省参数的构造函数），如果没找到就会产生编译错误。



【例5-16】 类A是类B、C的虚基类，类ABC从B、C派生，是继承结构中的最终派生类，它负责虚基类A的初始化。

```
#include <iostream>
class A {
    int a;
public:
    A(int x) {
        a=x;
        cout<<"Virtual Bass A..."<<endl;
    }
};
```

.



```
class B:virtual public A {
public:
    B(int i):A(i){ cout<<"Virtual Bass B..."<<endl; }
};
class C:virtual public A{
    int x;
public:
    C(int i):A(i){
        cout<<"Constructing C..."<<endl;
        x=i;
    }
};
class ABC:public C, public B {
public:
    ABC(int i,int j,int k):C(i),B(j),A(i) //L1, 这里必须对A进行初始化
        { cout<<"Constructing ABC..."<<endl; }
};
int main(){
    ABC obj(1,2,3);
}
A→C→B→ABC
```

Virtual Bass A...
Constructing C...
Virtual Bass B...
Constructing ABC...



5.7 基类与派生类对象的关系

- **基类对象与派生类对象之间存在赋值相容性。包括以下几种情况：**
 - 把派生类对象赋值给基类对象。
 - 把派生类对象的地址赋值给基类指针。
 - 用派生类对象初始化基类对象的引用。
- **反之则不行**，即不能把基类对象赋值给派生类对象；不能把基类对象的地址赋值给派生类对象的指针；也不能把基类对象作为派生对象的引用。



【例5-17】 把派生类对象赋值给基类对象的例子。

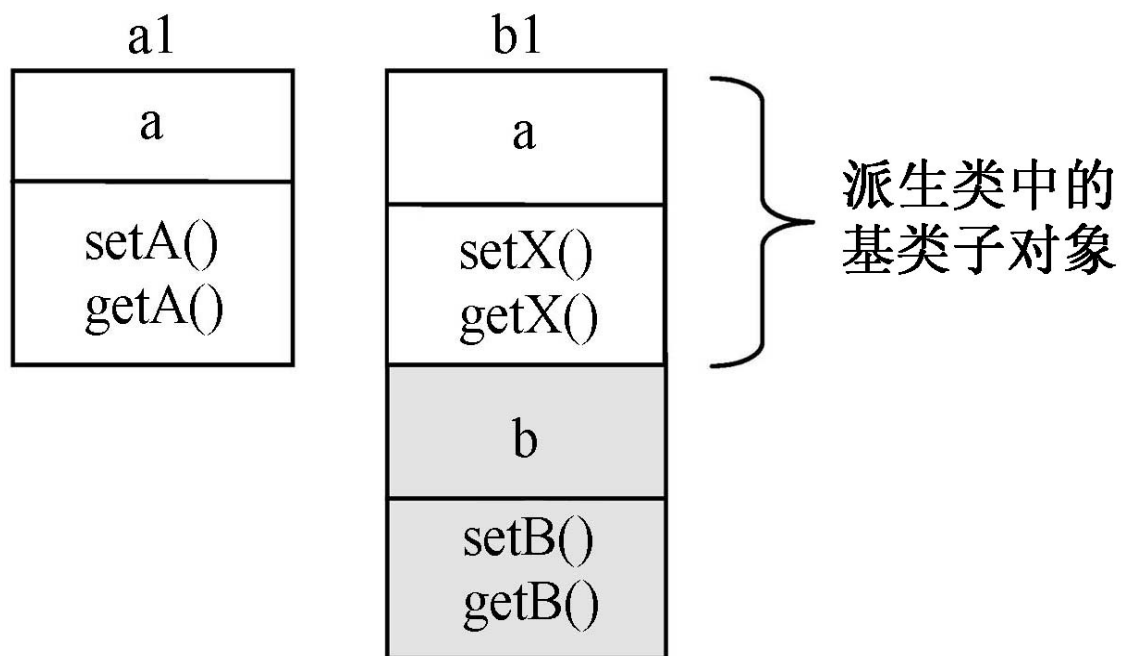
```
#include <iostream>
using namespace std;
class A {
    int a;
public:
    void setA(int x){ a=x; }
    int getA(){ return a;}
};
class B:public A{
    int b;
public:
    void setB(int x){ b=x; }
    int getB(){ return b;}
};
```





```
void f1(A a, int x){ a.setA(x); }
void f2(A *pA, int x){ pA->setA(x); }
void f3(A &rA, int x){ rA.setA(x); }
int main(){
    A a1,*pA;
    B b1,*pB;
    a1.setA(1);
    b1.setA(2);
    a1=b1;
    cout<<a1.getA()<<endl;
    cout<<b1.getA()<<endl;
    a1.setA(10);
    cout<<a1.getA()<<endl;
    cout<<b1.getA()<<endl;
    pA=&b1;
```

```
pA->setA(20);
cout<<pA->getA()<<endl;
cout<<b1.getA()<<endl;
A &ra=b1;
    ra.setA(30);
    cout<<pA->getA()<<endl;
    cout<<b1.getA()<<endl;
b1.setA(7);
    cout<<b1.getA()<<endl;
f1(b1,100);
    cout<<b1.getA()<<endl;
f2(&b1,200);
    cout<<b1.getA()<<endl;
f3(b1,300);
    cout<<b1.getA()<<endl; }
```





- 说明：
- ① 不论以哪种方式把派生类对象赋值给基类对象，都只能访问到派生类对象中的基类子对象部分的成员，不能访问派生类的自定义成员。
- ② 只能把派生类对象赋值给基类对象，不能把基类对象赋值给派生类对象。



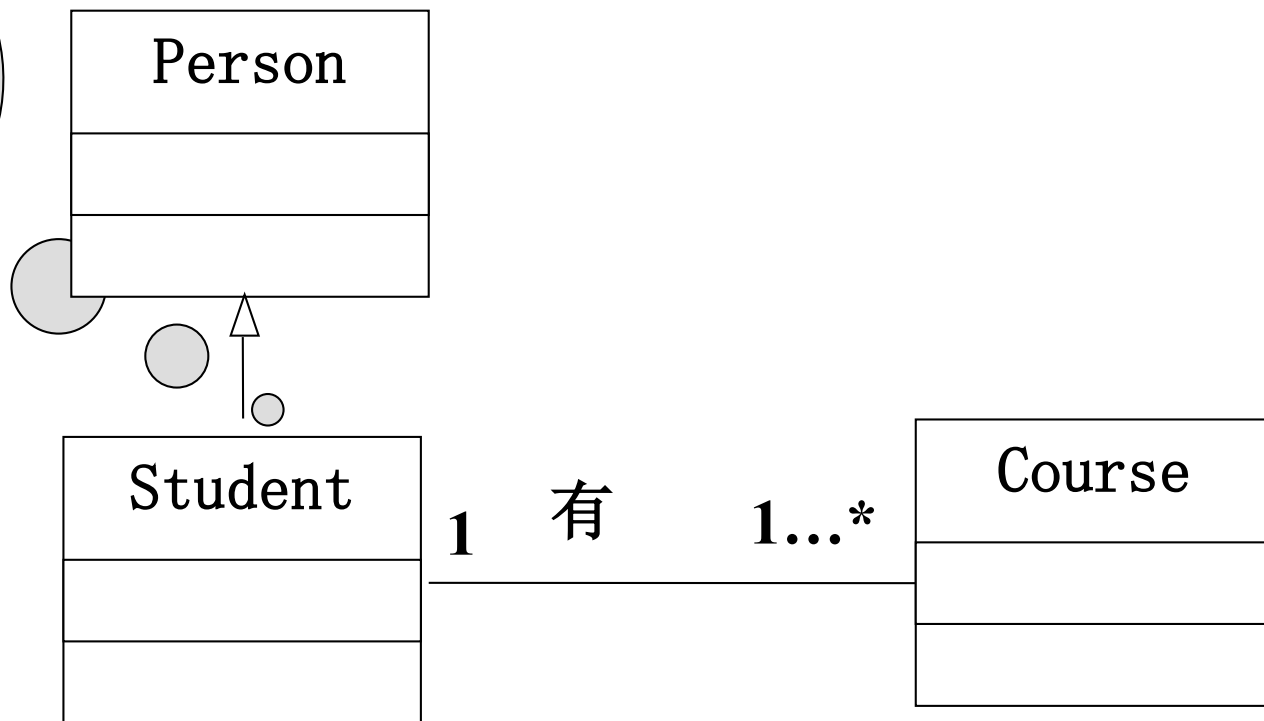
5. 8继承与组合

- **继承与组合（也称合成）是C++实现代码重用的两种主要方法。**通过继承，派生类可以获得基类的程序代码，从而达到代码重用的目的。而组合则体现了类之间的另一种关系，是指一个类可以包容另外的类，即用其他类来定义它的对象成员。
- **继承关系常被称为“Is-a”关系，即两个类之间若存在Is-a关系，就可以用继承来实现它。**比如，水果和梨，水果和苹果，它们就具有Is-a关系。因为梨是水果，苹果也是水果，所以梨和苹果都可以从水果继承，获得所有水果都具有的通用特征。
- **组合常用于描述类之间的“Has-a”关系，即一个类拥有另外一些类。**比如，图书馆有图书，汽车有发动机、车轮胎、座位等，计算机有CPU、存储器、显示器等，这些都可以用类的组合关系来实现。



- **【例5-18】** 设计一个成绩单管理的程序。其中包括学生信息，如姓名、身高、性别、学号、专业等，学生学习某门课程后将得到一个成绩，相关的信息有课程名称、课程编号学分和成绩等。

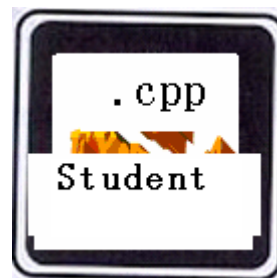
经过抽象与
继承，构造
出图所示的
类结构



- 将每个类的声明与实现分别独立保存在与类同名的.h头文件和.cpp文件中。
- Course类



- Student类



- 项目文件



1下列对派生类的描述中，（ ）是错误的。

- A. 一个派生类可以作为另一个派生类的基类
- B. 派生类至少有一个基类
- C. 派生类的成员除了它自己的成员外，还包含了它的基类成员
- D. 派生类中继承的基类成员的访问权限到派生类保持不变

2派生类的对象对它的哪一类基类成员是可以访问的？（ ）

- A. 公有继承的基类的公有成员
- B. 公有继承的基类的保护成员
- C. 公有继承的基类的私有成员
- D. 保护继承的基类的公有成员

3关于多继承二义性的描述，（ ）是错误的。

- A. 派生类的多个基类中存在同名成员时，派生类对这个成员访问可能出现二义性
- B. 一个派生类是从具有共同的间接基类的两个基类派生来的，派生类对该公共基类的访问可能出现二义性
- C. 解决二义性最常用的方法是作用域运算符对成员进行限定
- D. 派生类和它的基类中出现同名函数时，将可能出现二义性

4 多继承派生类构造函数构造对象时，（ ）被最先调用。

- A. 派生类自己的构造函数
- B. 虚基类的构造函数
- C. 非虚基类的构造函数
- D. 派生类中子对象类的构造函数

5 C++类体系中，能被派生类继承的是（ ）。

- A. 构造函数
- B. 虚函数
- C. 析构函数
- D. 友元函数

6 派生类的构造函数的成员初始化列表中，不能包含（ ）。

- A. 基类的构造函数
- B. 派生类中对象成员的初始化
- C. 基类的对象成员的初始化
- D. 派生类中一般数据成员的初始化

7 设有基类定义：

```
class Cbase { private: int a;    protected: int b;    public: int c; };
```

派生类采用何种继承方式可以使成员变量b成为自己的私有成员。

- A. 私有继承
- B. 保护继承
- C. 公有继承
- D. 私有、保护、公有均可

8 C++中的类有两种用法：一种是类的实例化，即生成类对象，并参与系统的运行；另一种是通过（ ）派生了新的类。

- A. 复用
- B. 继承
- C. 封装
- D. 引用

9 若要用派生类的对象访问基类的保护成员，以下观点正确的是
()

- A.不可能实现 B.可采用保护继承
C.可采用私有继承 D.可采用公有继承

10 下列程序输出结果是：

```
class A { public: int n; };  
class B:public A{};  
class C:public A{};  
class D:public B,public C { int getn(){return B::n;} };  
void main() {  
    D d;  
    d.B::n=10;  
    d.C::n=20;  
    cout<<d.B::n<<" "<<d.C::n<<endl;  
}
```

```
class Base{
    int i;
public:
    Base(int n){
        cout<<"Constucting base class"<<endl;
        i=n;
    }
    ~Base(){
        cout<<"Destructing base class"<<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 class"<<endl;
    }
    void show(){
        Base::showi();
        cout<<j<<","<<aa.Geti()<<endl;
    }
};

int main(){
    Derived obj(8,13,24);
    obj.show();
}

```

```

Constructing base class
Constructing base class
Constructing derived class
13,8,24
Destructing derived class
Destructing base class
Destructing base class
请按任意键继续. . .

```

```
class A{
public:
A(char *s) { cout<<s<<endl; }
~A() {}
};

class B:virtual public A{
public:
B(char *s1, char *s2):A(s1) {
cout<<s2<<endl;
}
};

class C: virtual public A{
public:
C(char *s1, char *s2):A(s1) {
cout<<s2<<endl;
}
};

class D:public B,public C{
public:
D(char *s1, char *s2, char *s3, char *s4):B(s1, s2), C(s1, s3), A(s1) {
cout<<s4<<endl;
}
};

void main() {
D *p=new D("class A", "class B", "class C", "class D");
delete p;
}
```



```

class parent{
int i;
protected:
int x;
public:
parent() {x=0;i=0;}
void change() {x++;i++;}
void display();
};
class son: public parent{
public:
void modify();
};
void parent::display() {cout<<"x="<<x<<endl;}
void son::modify() {x++;}
void main() {
son A;
parent B;
cout<<"Display derived class object A:\n";
A.display();
A.change();
A.display();
A.modify();
A.display();
cout<<"Display base class object B:\n";
B.change();
B.display();
}

```

```

Display derived class object A:
x=0
x=1
x=2
Display base class object B:
x=1
请按任意键继续. . .

```