

# 第4章 继承

## • 本章主要教学内容

- 继承的基本原理和实现技术，继承方式：公有继承，保护继承，私有继承，多继承，虚拟继承。
- 派生类与基类对象之间的关系：赋值相容和类型转换。
- 派生类构造函数如何提供对基类构造函数的调用。
- 通过继承与组合进行程序设计。

## • 本章教学重点

- 学会用继承解决类之间层次结构关系，进行代码复用，提高软件开发效率
- 继承方式，派生类与基类成员的关系，及其对基类成员的使用、重载、访问权限更改和函数功能修改
- 继承体系中静态成员的功能和设计方法
- 派生类构造函数、复制构造函数、赋值运算符函数设计，派生类、对象成员和基类构造函数和析构函数的调用次序

## • 教学难点

- 派生类的构造函数、复制构造函数、赋值函数设计
- 派生类与基类对象的类型转换与对象复制、赋值之间的区别和相关成员函数设计
- 多继承、多对象成员的派生类构造函数设计和调用问题
- 虚拟继承方式下的派生对象和基类对象构造函数的设计和执行过程分析

# 4.1 继承的概念

## 1. 继承的概念

- 以现有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的一份复制品，这就是**继承**。

基类

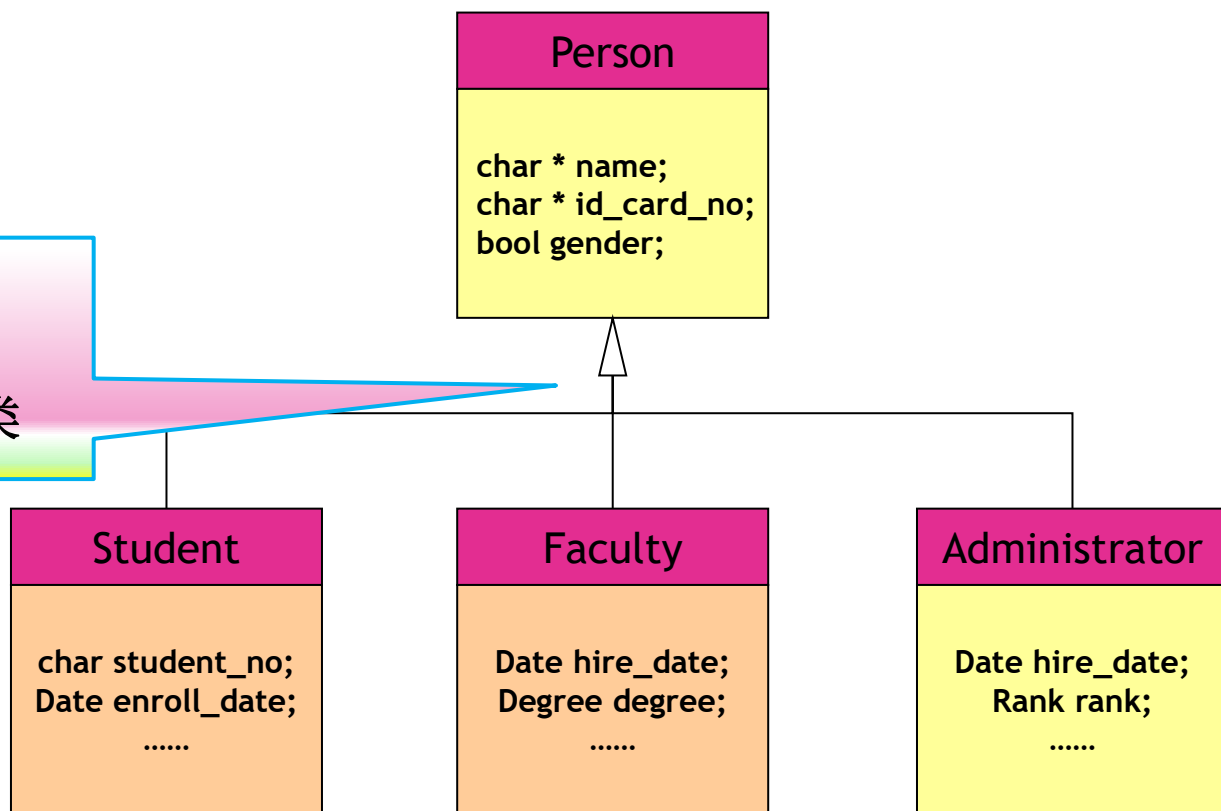
也叫超类，父类

继承方式：

单一继承：只有一基类  
多重继承：可以有多个基类

派生类

也叫子类



# 4.1 继承的概念

---

## 2、继承目的

- 代码重用**code reuse**
- 描述能力：类属关系广泛存在
- **IsA vs. HasA**

## 3、有关概念

基类，超类

派生类，子类

# 4.1 继承的概念

---

## 4、派生类可实施对基类的改变

- 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（覆盖）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      C++11之后可以继承
- 析构函数
- 基类的友元函数
- 静态数据成员和静态成员函数

## 4.2 **protected**和继承

---

- 基类中**protected**的成员

- 类内部：可以访问
- 类的使用者：不能访问
- 类的派生类成员：可以访问

【例4-1】 类B有数据成员i, j, k, 希望j可被派生类D和自身访问, 但不希望除此之外的其它函数访问。

分析: **protected**权限正好具有这样的访问控制能力

```
class B
```

```
{ private: int i;
```

```
protected: int j;
```

```
public: int k;
```

```
};
```

```
class D: public B
```

```
{public:
```

```
void f()
```

```
{ i=1; //cannot access
```

```
j=2; k=3; }
```

```
};
```

```
void main()
```

```
{ B b;
```

```
b.i = 1; //cannot access
```

```
b.j=2; //cannot access
```

```
b.k=3;
```

```
}
```

接口

私有数据

B

k

j

i

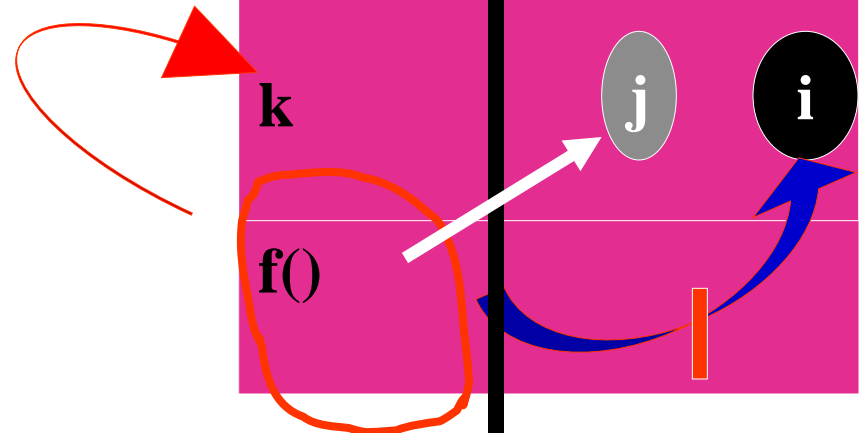
D

k

j

i

f()



# 4.3. 继承方式

## 1、C++的继承方式

- 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改变基类成员在派生类中的访问权限

## 2、继承语法形式

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

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

```
{
```

派生类对象

```
.....
```

```
};
```

基类子对象

继承部分

派生类新定义  
成员

派生部分

## 4.3. 继承方式

---

- 1、**public**继承

- 最常用的派生方式，派生类复制了基类数据成员和成员函数的一份复制品。
- 派生类从基类继承到的成员，维持基类成员的可访问性。即基类的**public**成员在派生类中也是**public**成员，可被派生类的外部函数访问。
- 同样，一个成员若在基类是**protected**或**private**属性，它在派生类中仍然是**protected**或**private**属性
- 在任何情况下，类的**private**成员只能被自身类的成员访问。因此，派生类不可直接访问基类的**private**成员



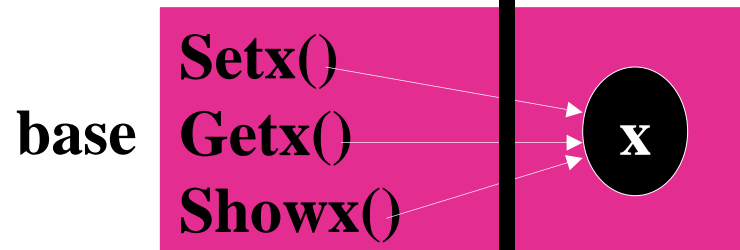
- 例题4\_2.cpp, 派生类Derived继承了基类Base的数据成员x和全部成员函数

接口

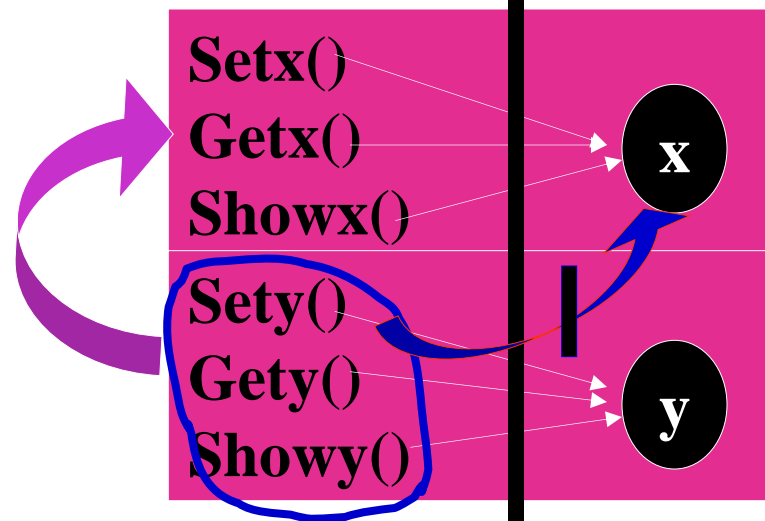
私有数据

```
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;    }
};
```



derived



## 4.3. 继承方式

```
void main()
{ Derived obj;
  obj.setx(10);
  cout<<obj.getx();;
  obj.showy();
  obj.sety();
  obj.sety(20);
  obj.showy();
  obj.showy();
}
```

**Derived**类并未定义**setx**等成员函数，但却调用了它们。原因是它从**Base**类继承了这些函数。

## 4.3. 继承方式

### 2.private继承

- 基类中的**public**、**protected**成员在派生类中是**private**,基类中的**private**成员在派生类中不可访问。

【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; }
};
```

## 4.3. 继承方式

### 2、private继承

```
class Derived:private Base{  
    .....  
}
```

- 在**private**派生方式下，派生复制了基类全部成员，但复制到的成员在派生类中**全部变成了private成员**。
- 在**private**派生方式下，虽然基类的**public**和**protected**成员在派生类中都变成了**private**成员，但它们仍然有区别：**派生类的成员函数不能直接访问基类的private成员，但可以直接访问基类的public和protected成员，并且通过它们访问基类本身的private成员。**

```
class derived:private base{
```

```
    int y;
```

```
public:
```

```
    void sety(int n){y=n;  }
```

```
    void gety(){  
        y=getx();  }
```

```
    void showy()  {  
        cout<<y<<endl;  }
```

```
};
```

```
void main(){
```

```
    derived obj;
```

```
    obj.setx(10); // cannot access
```

```
    obj.sety(20);
```

```
    obj.showx(); // cannot access
```

```
    obj.showy();
```

```
}
```

接口

私有数据

base

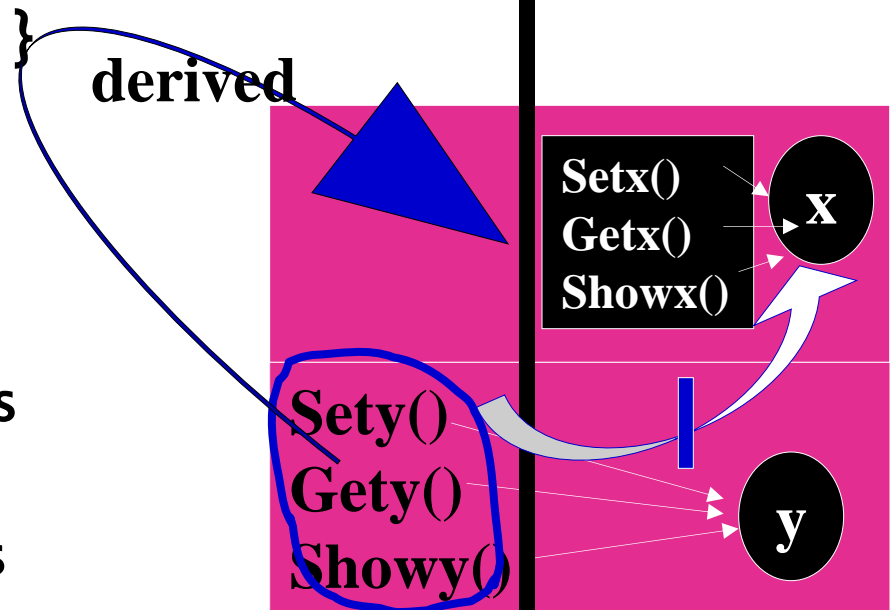
Setx()  
Getx()  
Showx()

x

derived

Sety()  
Gety()  
Showy()

y



## 4.3. 继承方式

### 3、protected继承

```
class Derived:protected Base{  
    .....  
}
```

- 在**protected**派生方式下，派生复制了基类全部成员，但复制到**public**成员在派生类中变成了**protected**成员，其余成员的访问权限保持不变。
- 在**protected**派生方式下，**派生类继承到的成员函数都不能被外部函数访问，但在派生类中可以直接访问基类的**public**和**protected**成员，并且通过它们访问基类本身的**private**成员。**

## 4.3. 继承方式

```
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; }
};
```

```
void main(){
    Derived obj;
    obj.setx(10);
    obj.showx();
    obj.sety(20);
    obj.showy();
}
```

//错误  
//错误,

```
#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; }
};
```

protected继承方式已将  
setx, showx改变成了  
protected成员, 不能在  
派生类外直接访问

# • 基类成员在派生类中的访问权限

基类	public继承			protected继承			private继承		
	public	protected	private	public	protected	private	public	protected	private
public	√				√				√
protected		√			√				√
private			√N/A			√N/A			√N/A

## □不能继承的基类内容

1. 构造函数      C++11标准以前（C++11起可以继承）
- 2.析构函数
- 3.友员关系
- 4.针对基类定义的一些特殊运算符，如new等。



## 4.3. 继承方式

### 4. 阻止继承

11C++

- 如果不想让一个类作为其它类的基类，可以用 **final** 关键字阻止它被继承。形式如下：

<code>class Base{.....}</code>	//可以被继承
<code>class NoDeri <b>final</b>{.....}</code>	//不能被继承
<code>class D <b>final</b>:Base{.....}</code>	//正确，D不能被继承
<code>class D1:NoDeri{.....}</code>	//错误，NoDeri不能被继承
<code>class D2:D{.....}</code>	//错误，D不能被继承

## 4.4 派生类对基类的扩展

### 1、派生类和基类的关系

- 派生类拷贝了基类数据成员和成员函数的一份副本，不用编程就具备了基类的程序功能。
- 在派生类对象中，具有一个基类子对象。

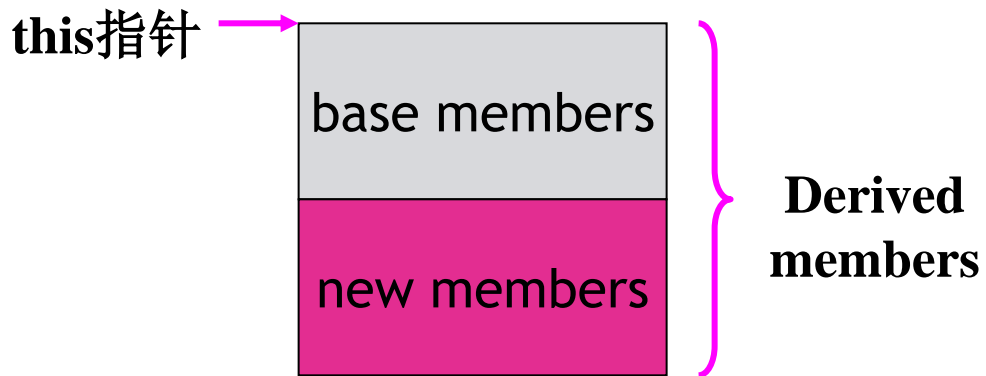
```
class Base
```

```
{base members;};
```

```
class Derived: Base
```

```
{new members;};
```

this指针



## 4.4 派生类对基类的扩展

---

### 2、派生类和基类成员的修改和扩展

– 派生类可以在继承基类成员的基础上

- ① 派生类可以增加新的数据成员和成员函数；
- ② 重载从基类继承到的成员函数；
- ③ 覆盖（重定义）从基类继承到的成员函数；
- ④ 改变基类成员在派生类中的访问属性。

### 3、派生类不能继承基类的以下成员

- ① 析构函数。
- ② 基类的友元函数。
- ③ 静态数据成员和静态成员函数。

## 4.4.1 成员函数的重定义和名字隐藏

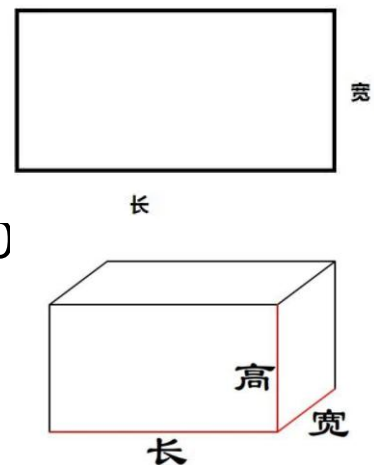
### 1、成员函数的重定义和名字隐藏

- 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。**

【例4-3】设计计算矩形与立方体面积和体积的类。

#### (1) 问题分析

- 矩形具有长和宽，**面积**=长×宽，没有体积，可以设置为0
- 具有高的矩形就是立方体，**面积**=2×底面积+2×侧面积1+2×侧面积2，**体积**=底面积×高。其中的**底面积就是矩形的面积。**
- 立方体是对矩形的扩展，矩形完成了长和宽的处理，在此基础上完成高的处理就能够实现其功能。这一关系可以通过**继承**实现。



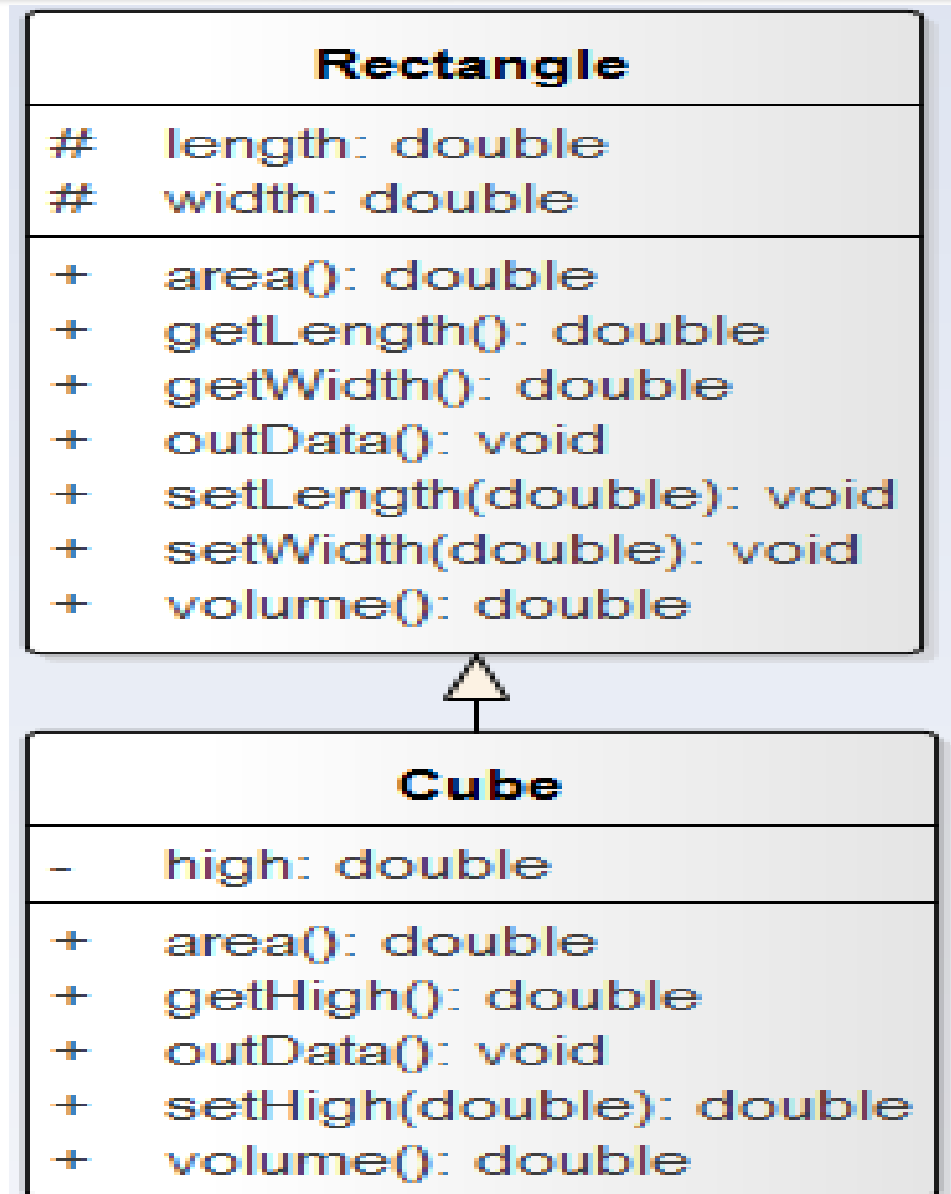
## 4.4.1 成员函数的重定义和名字隐藏

### (2) 数据抽象

- 将矩形抽象成类**Rectangle**，用数据成员**width**、**length**表示宽与长，为了方便其派生类访问数据成员，将它们设置为**protected** 访问权限；并设置成员函数**setWidth**、**setLength**、**getWidth**、**getLength**来设置和获取矩形的宽和长，**area**和**volume**成员函数计算矩形的面积和体积，成员函数**outData**输出矩形的长和宽。
- 将立方体抽象成类**Cube**，并从**Rectangle**类派生，由于**Rectangle**类已经完成了矩形长和宽的处理功能，所以只须增加数据成员**high**表示高，并设置**setHigh**和**getHigh**成员函数完成对高的读、写功能。
- 虽然**Rectangle**类已经设置了**area**、**volume**和**outData**成员函数计算面积、体积，或输出数据成员的值，但立方体的面积、体积和数据成员是不同的，需要**重新定义**它们

## 4.4.1 成员函数的重定义和名字隐藏

- 矩形和立方体的抽象结果
- 注意**Cube**对基类继承到的成员函数的重定义会**隐藏继承于基类的同名函数**



## 4.4.1 成员函数的重定义和名字隐藏

//Eg4-3.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
class Rectangle {
```

```
public:
```

```
    void setLength(double h) { length = h; }
```

```
    void setWidth(double w) { width = w; }
```

```
    double getLength() { return length; }
```

```
    double getWidth() { return width; }
```

```
    double area() { return length*width; }
```

```
    double volume() { return 0; }
```

```
    void outData(){cout<<"length="<<length<<"\t"
                    <<"width="<<width<< endl;
```

```
}
```

```
protected:
```

```
    double width;
```

```
    double length;
```

```
};
```

## 4.4.1 成员函数的重定义和名字隐藏

```
class Cube :public Rectangle{
public:
    void setHigh(double h) { high = h; }
    double getHigh() { return high; }
    double area()
        {return width*length*2+width*high*2+length*high*2; }    //L1
    double volume() { return Rectangle::area()*high; }
    void outData() {
        Rectangle::outData(); //L2 派生类访问基类同名成员的方法
        cout << "high=" << high << endl;
    }
private:
    double high;
};
```



## 4.4.1 成员函数的重定义和名字隐藏

```
void main() {  
    Cube cub1;  
    cub1.setLength(4);  
    cub1.setWidth(5);  
    cub1.setHigh(3);  
    cub1.Rectangle::outData();           //访问基类继承到的同名成员  
    cub1.outData();                       //访问派生类的同名成员  
    cout<<"立方体面积="<<cub1.area()<<endl;  
    cout<<"立方体底面积="<<cub1.Rectangle::area()<<endl;  
    cout << "立方体体积=" << cub1.volume() << endl;  
}
```

## 4.4.2 基类成员访问

---

- 派生类对基类成员的访问有以下形式
  1. 通过派生类对象直接访问基类成员，如  
`cub1.setWidth(5);`
  2. 在派生类成员函数中直接访问基类成员，如  
前面**Cube**类的**L1**语句。
  3. 通过基类名字限定访问被重载的基类成员名
    - 在派生类成员函数中访问基类同名成员函数
      - 如**Cube**的**volume()** 和**area()**成员函数
    - 在派生对象中访问基类同名成员函数
      - **cub1.Rectangle::outData();**

### 1. 派生对基类同名成员的隐藏

- 如果基类某个成员函数具有多个重载的函数版本，派生类又定义了同名成员，就会隐藏基类同名的全部重载函数。

### 2. 访问隐藏成员

- ① 一是使用基类名称限定要访问的成员函数
- ② 重载基类的所有同名函数，而这些重载函数的代码与基类完全相同
- ③ 用using声明使基类重载函数在派生类中可见。用法如下：  
**using 基类名称::被隐藏成员函数名;**

【例4-4】基类B的f1成员函数具有3个重载函数，派生D新增加了f1函数的功能，此f1会隐藏基类B中f1函数在派生类的可见性，用using将基类的f1引入到派生类作用域内。

```
#include <iostream>
using namespace std;
class B {
public:
    void f1(int a) { cout << a << endl; }
    void f1(int a,int b) { cout << a+b<< endl; }
    void f1() { cout << "B::f1" << endl; }
};
class D : public B {
public:
    using B::f1; //L1, 使基类的3个f1函数在此区域可见
    void f1(char * d) { cout << d << endl; }
};
void main() {
    D d;
    d.f1(); //L2, 正确, 调用基类成员
    d.f1(3); //L3, 正确, 调用基类成员
    d.f1(3, 5); //L4, 正确, 调用基类成员
    d.f1("Hellow c++!");
}
```

## 4.4.4 派生类修改基类成员的访问权限

### 1. 修改原由

- 不同继承方式可能会改变基类成员在派生类中的访问权限。
  - 比如，在`private`继承方式下，基类的`public`和`protected`成员在派生类中的访问权限都会被更改为`private`访问权限。
- 在某些时候，从类的整体设计上考虑，需要调整个别基类成员在派生类中的访问权限，使用`using`声明可以实现这一目的。

### 2. 修改方法

在派生类的`public`、`protected`或`private`权限区域内，使用`using`再次声明基类的非`private`成员，就可以重新设置它们在派生类中的权限为`using`语句所在区域的权限。

即`using`语句在`public`区域内即为`public`权限，在`protected`内即为`protected`权限，在`private`内则为`private`权限。

### 3. 限定内容

在派生类中不允许修改基类的`private`成员

## 4.4.4 派生类修改基类成员的访问权限

【例4-5】类D私有继承了类Base，修改基类Base成员在派生类中的访问权限，设置基类成员y在派生类的权限为private，其余成员在派生类中的权限保持与其在基类中的相同权限。

```
#include <iostream>
using namespace std;
class Base {
public:
    int x = 0;
    void setxyz(int a, double b, float c) {
        x = a; y = b; z = c;
    }
protected:
    double y = 0;
    float getZ() { return z; }
private:
    float z = 0;
};
```

## 4.4.4 派生类修改基类成员的访问权限

**Private**继承使基类成员在派生类中都成私有成员

```
class D :private Base {  
protected:
```

```
    using Base::getZ;
```

//指定getz为protected权限

```
    //using Base::z;
```

//错误, 不允许修改基类private成员

```
public:
```

```
    using Base::x;
```

//指定x为public权限

```
    using Base::setxyz;
```

//指定setxyz为public权限

```
    void display() {
```

```
        cout << "x=" << x << "\ty=" << y << "\tz=" << getZ() << endl;
```

```
    }
```

```
private:
```

```
    using Base::y;
```

//指定y为private权限

```
};
```

```
void main() {
```

```
    D d;
```

```
    d.setxyz(8, 9, 10);
```

```
    d.display();
```

```
}
```

## 4.4.5 友元与继承

- 友元不能被继承

- 每个类只能够负责控制自己的成员的访问权限。
- 因此，如果一个类继承了其它类，则它声明的友元也只能访问它自己的全体成员，包括它从基类继承到的public和protected成员。而它的基类和派生类并不认可这种友元关系，按照规则只能访问公有成员。

- 【例4-6】类Deri是基类Base的友元，函数f1和f2是类Deri的友元，分析下面程序中L4、L5、L7正确的原因，以及L8和L6错误的原因。



## 4.4.5 友元与继承

---

```
#include <iostream>
using namespace std;
class Base {
public:
    int x = 0;
protected:
    double y = 0;
private:
    float z = 0;
    friend class Deri;
};
```

//L1 Deri为Base的友元

//Base的全体成员可访问x,y,z

```
class Deri :public Base {  
protected: int dx = 1;  
public:
```

```
    friend void f1(Deri d);  
    friend void f2(Base b);  
    void f3(Base b)  
    { cout<<b.x<<b.y<<b.z<<endl;} //L4正确
```

```
};  
void f1(Deri d) {  
    cout << d.x << d.y <<d.dx<<endl;  
    //cout<<d.z<<endl;
```

//L5正确

//L6错误

```
}  
void f2(Base b) {  
    cout << b.x << endl;  
    //cout << b.y << endl;  
}
```

//L7正确

//L8错误

```
void main() {  
    Base b;  
    Deri d;  
    f1(d);  
    f2(b);  
}
```

```
class Base {  
    public:    int x = 0;  
    protected: double y = 0;  
    private:  float z = 0;  
  
    friend class Deri;  
};
```

## 4.4.6 静态成员与继承

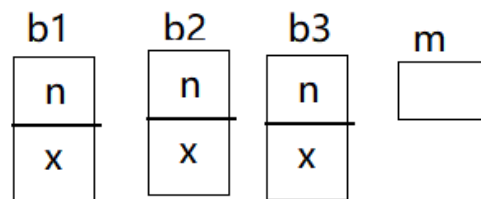
### 1. 基类静态成员为继承层次结构所有类共享

- 在继承体系中，如果基类定义了静态成员，则在**整个继承体系中只有该成员的唯一定义**，不论从该基类派生出了多少个或多少层次的派生类，静态成员都只有一个实例，为整个继承体系中的全体对象所共用。

### 2. 基类静态成员在继承结构中的应用

- 设计全类公用数据，或统计继承体系中的对象个数：**将共享数据或计数器设置为基类的静态成员**，就能够实现这样的目的。

```
class A{  
    int n;  
    static int m;  
.....}  
class B:public A{  
    int x;  
.....  
}  
B b1,b2,b3;
```



## 4.4.6 静态成员与继承

【例4-7】假设父亲生了儿子、女儿，儿子又生有孙子，构成了家族继承体系，统计家族成员的人数。

### 问题分析：

用Father、Son、Daughter、Grandson分别表示父亲类、儿子类、女儿类和孙子类，它们通过继承形成了层次结构的继承体系。

### 数据抽象

在Father类中设计静态成员personNum统计家族的人数，每构造一个对象人数就增加1，每析构一个对象就减少1；由于每个人都有姓名，因此在基类Father中设置name数据成员代表人名。为了便于派生类访问personNum和name成员，把它们设置为protected访问权限。

## 4.4.6 静态成员与继承

```
#include <iostream>
#include<string>
using namespace std;
class Father {
protected:
    string name;
    static int personNum;
public:
    Father(string Name = "") :name(Name) { personNum++; }
    ~Father() { personNum--; }
    static int getPersonNumber() { return personNum; }
};
int Father::personNum = 0;
class Son :public Father {
public:
    Son(string name) :Father(name) {}
};
```

```

class Daugther :Father {
public:
    Daugther(string name) :Father(name) {}
};
class Grandson :public Son {
public:
    Grandson(string name) :Son(name) {}
};
void main() {
    Father son("tom");
    Son sson("jack");
    Daugther dson("mike");
    {
        Grandson gson("s.jack");
        cout << son.getPersonNumber() << endl;
    }
    cout << son.getPersonNumber() << endl;
}

```

到L1语句时，总共定义了4个对象，都记在了由基类定义的共享静态成员personNum中，因此L1语句输出4

//L1，输出4

//L2，输出3

## 4.4.7 继承与类作用域

---

### 1. 基类类域

- 每个类都建立了属于自己的作用域，本类的全体成员都位于此作用域内，而且相互之间可以直接访问，不受定义先后次序的影响。
- 例如，一个成员函数可以调用在它后面定义的一个成员函数。

### 2. 派生类类域

- 派生类的作用域嵌套在基类作用域的内层。
- 在解析类成员名称时，如果在本类的作用域内没有找到，编译器就会接着在外层的基类作用域内继续寻找该成员名称的定义。

## 4.4.7 继承与类作用域

### 3. 编译器解析之后的派生类类域形式

— 例如类A、B、C继承形式如下

```
class A {  
    int g();.....  
};  
class B:public A{  
    int h(int);.....  
};  
class C:public B{  
    int c;  
    int h();  
    int f(int ); .....  
};
```



## 4.4.7 继承与类作用域

经编译器处理之后，形成类似于下面的块作用域：

```
A{  
  int g(){.....}  
  .....  
B {  
  int h(int ){.....};  
  .....  
C {  
  int c;  
  int h(){.....};  
  int f(int i){.....;return B::h(i);}  
  .....  
}  
}
```

```
C  xa;  
xa.g();  
xa.h(3);           //L2, 错误  
xa.B::h(3);        //L3, 正确
```

### 思考题：

1. 分析xa.g()的调用过程？
2. 分析L2错误的原因？

# 4.5 构造函数和析构函数

## 1. 为什么要设计构造函数？

- 在任何时候，只要定义类的对象，就需要调用适当的构造函数。因此，设计类时必须要考虑类的构造函数设计（包括派生类）。
- 有时，一个类没有构造函数也在使用，这种情况只能定义无参对象，而且它调用了编译器为它生成的默认构造函数（这种情况一定符合编译器为类自动生成默认构造函数的情况）。

## 2. 如何设计构造函数

- 在用类定义对象时，通常会用到默认构造函数（定义无参对象或对象数组）、拷贝构造函数（类对象作函数参数），赋值运算符函数（对象赋值），移动构造函数，移动拷贝构造函数和移动赋值运算符函数，当类没有定义任何构造函数时，编译器在需要来会自动为类生成这些成员函数。
- 在通常情况下，由编译器生成的上述成员函数已能够胜任对象的定义或复制了，即可以不定义这些成员函数。
- 但是，当类存在指针数据成员时，就很有可能需要显示定义这些成员函数，否则很有可能产生指针悬挂问题。

## 4.5.1 派生类构造函数的建立规则

1. 派生类必须为基类和对象成员提供构造函数初值。

- 派生类只能采用构造函数初始化列表的方式向基类构造函数提供初值。
- 派生类可以通过类内初始值或构造函数初始化列表向对象成员提供构造函数初值。
- 形式如下：

派生类构造函数名(参数表):基类构造函数名(参数表),对象成员名1(参数表),.....{

.....

}

## 4.5.1 派生类构造函数的建立规则

【例】 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;
        cout<<"B ctor x="<<x<<endl;
    }
    ~Base()
    { cout<<"~B dctor... " <<x<<endl; }
};
```

```
class Derived:public Base{
private:
    int y;
public:
    Derived(int a,int b):Base(a){ \\初始化列表
        y=b;
        cout<<"D ctor y="<<y<<endl;
    }
    ~Derived(){ cout<<"~D dctor..."<<endl; }
};
void main(){
    Derived d(1,2);
}
```

## 4.5.1 派生类构造函数的建立规则

---

### 2、派生类必须定义构造函数的情况

- 当基类或成员对象（未进行类内初始化）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。
- 派生类构造函数以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。

## 4.5.1 派生类构造函数的建立规则

### 【例4.8】 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;
        cout<<"constructing point("<<x<<","<<y<<")"<<endl;
    }
};
```

## 4.5.1 派生类构造函数的建立规则

```
class Line:public Point{
protected:
    int len;
public:
    Line(int a,int b,int l):Point(a,b) { //构造函数初始化列表
        len=l;
        cout<<"Constructing Line,len ..."<<len<<endl;
    }
};

void main(){
    Line L1(1,2,3);
}
```

**Line**类必须在构造函数列表中向基类**Point**类构造函数提供初值！

## 4.5.1 派生类构造函数的建立规则

---

### 3、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。
  - ① 基类没有定义任何构造函数。
  - ② 基类具有缺省参数的构造函数。
  - ③ 基类具有无参构造函数。

【例4-9】 类A具有默认构造函数，其派生类B没有成员要初始化，不必定义构造函数。



## 4.5.1 派生类构造函数的建立规则

```
//Eg4-9.cpp
#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; }
};
void main(){
    B b;
}
```

程序运行结果:  
Constructing A  
Destructing B  
Destructing A

此结果表明，在定义对b时，调用了编译器为类B自动合成的默认构造函数，此函数类似于下面的形式：

**B::B():A(){ }**

## 4.5.1 派生类构造函数的建立规则

### 4、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。
- 基类的默认构造函数可以被自动调用。
- 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。
- 这条规则有一个例外情况，当派生类存在虚基类时，所有虚基类都由最后的派生类负责初始化。

```
class A {  
public:  
    A(int) {}  
};  
  
class B:A {  
public:  
    B(int i) :A(i)  
    {}  
};  
  
class C :B {  
    C(int j) :B(j)  
    {}  
};
```

**【例4-10】** 类C具有直接基类B和间接基类A，每个派生类只负责其直接基类的构造。

---

**//Eg4-10.cpp**

**#include <iostream>**

**using namespace std;**

**class A {**

**int x;**

**public:**

**A(int aa) {**

**x=aa;**

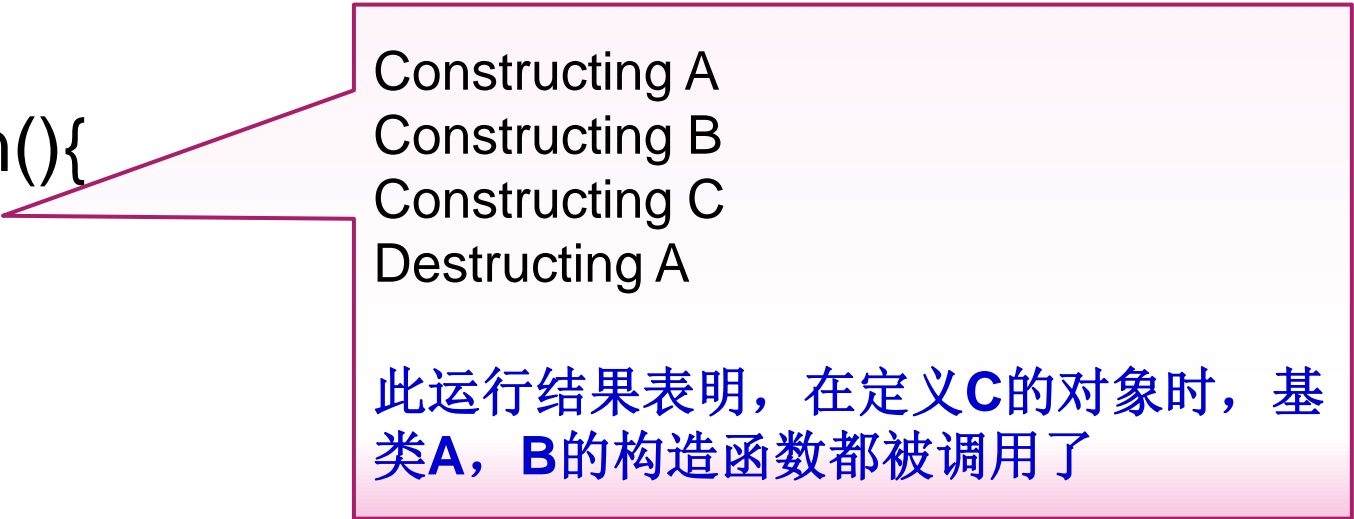
**cout<<"Constructing A"<<endl;**

**}**

**~A(){ cout<<"Destructing A"<<endl; }**

**};**

```
class B:public A {  
public:  
    B(int x):A(x){ cout<<"Constructing B"<<endl; }  
};  
class C :public B{  
public:  
    C(int y):B(y){ cout<<"Constructing C"<<endl; }  
};  
void main(){  
    C c(1);  
}
```



Constructing A  
Constructing B  
Constructing C  
Destructing A

此运行结果表明，在定义**C**的对象时，基类**A**，**B**的构造函数都被调用了

## 4.5.1 派生类构造函数的建立规则

- 5. 派生类继承基类的构造函数 11C<sub>++</sub>

- ① 关于构造函数继承

- C++11新增加标准（以前不允许继承构造函数）

- 解决的问题：

- 当基类具有多个重载构造函数，或构造函数具有较多参数，而派生类又没有数据成员需要初始化，但它却必须提供构造函数，其唯一目的是为基类构造函数提供初始化值。在这种情况下，派生类可以继承直接基类的构造函数。

## 4.5.1 派生类构造函数的建立规则

### ② 构造函数继承方法

用**using**在派生类中声明基类构造函数名即可。形式如下：

```
class Base:{.....}
```

```
class Derived: [public] Base{ //也可以是private或protected继承
```

```
.....
```

```
using Base::Base;
```

```
//继承基类构造函数
```

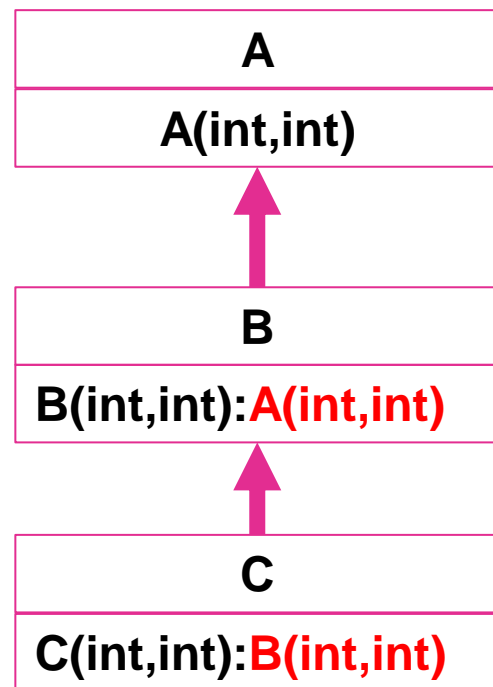
```
}
```

## 4.5.1 派生类构造函数的建立规则

**例4-11】**类A具有数据成员x、y，并且定义了初始化它们的构造函数；类B从A派生，没有任何成员要初始化；类C从类B派生，具有新定义数据成员c。设计A、B、C的构造函数。

**问题分析：**

- 按照规则，类B虽然没有数据成员要初始化，但是它**必须为基类A的构造函数提供初值**（除非A具有默认构造函数）
- 现在，可以通过**继承A的构造函数使问题更简单**。
- 类C要定义构造函数以便初始化其成员c，同时还必须为直接基类B提供构造初值。



```
///Eg4-11.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
    int x, y;
```

```
public:
```

```
    A(int aa) :x(aa) { cout << "Constructing A:x=\t" << x << endl; }
```

```
    A(int a, int b) :x(a), y(b) {
```

```
        cout << "Constructing A:x=\t" << x << endl;
```

```
    }
```

```
};
```

```
class B :public A {
```

```
public:
```

```
    using A::A;      //L1
```

```
    /* B(int x) :A(x) { //L2
```

```
        cout << "Constructing B\t" << endl;
```

```
    */
```

```
};
```

**L1**声明类**B**继承了**A**的构造函数，编译会为类**B**自动生成相应的程序代码，类似于：

**B::B(int a):A(a){**

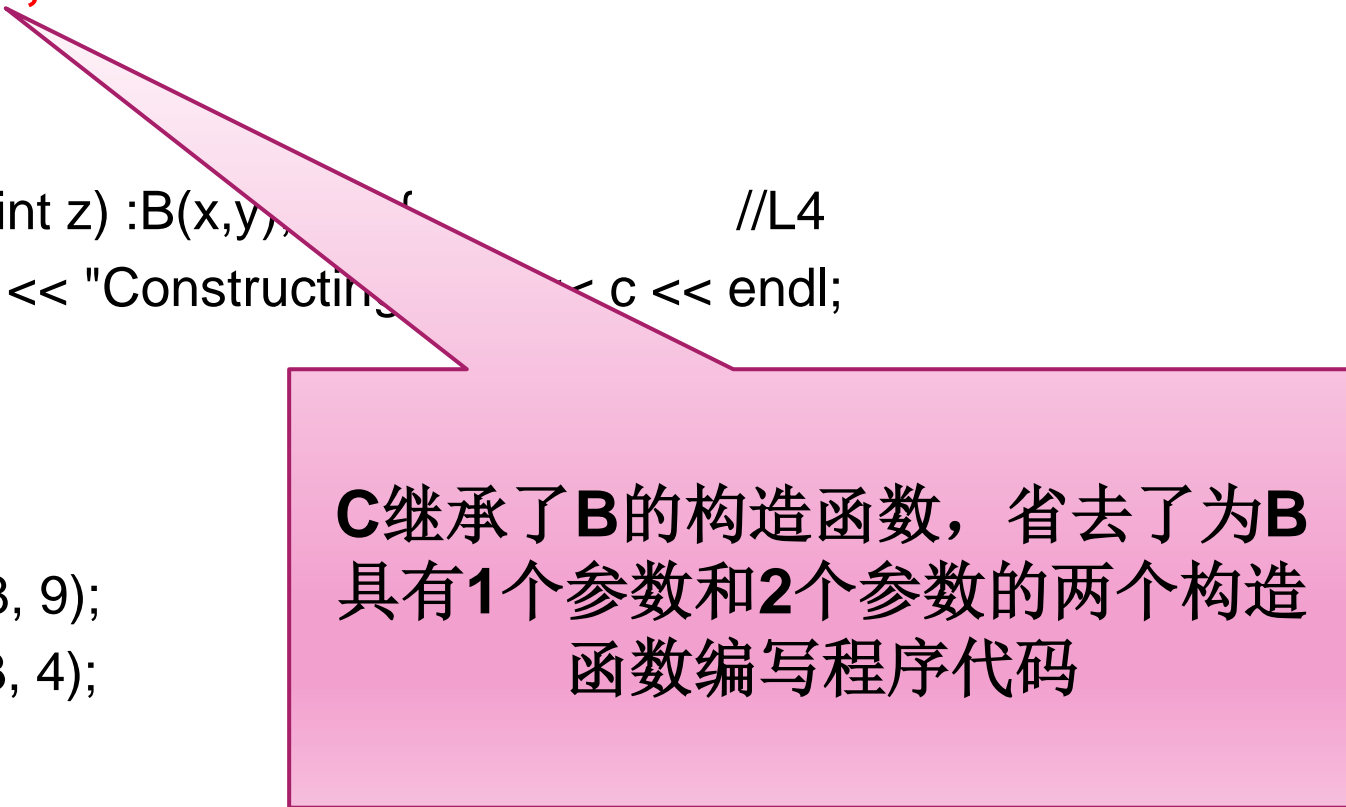
**B::B(int a,int b):A(a,b){**

不论类**A**有多少构造函数，系统都会自动生成。如果没有**L1**，就需像**L2**一样为每个基类构造函数提供程序代码。



## 4.5.1 派生类构造函数的建立规则

```
class C : public B {  
    using B::B; //L3  
    int c;  
public:  
    C(int x, int y, int z) : B(x, y), { //L4  
        cout << "Constructing C" << c << endl;  
    }  
};  
void main() {  
    B b1(1), b2(8, 9);  
    C c1(1), c2(3, 4);  
}
```



**C继承了B的构造函数，省去了为B具有1个参数和2个参数的两个构造函数编写程序代码**

## 4.5.1 派生类构造函数的建立规则

### ③ 构造继承的几点说明

- a) “Base::Base” 即为**基类名**和**基类构造函数的名称**，using语句说明了派生类要**继承基类的构造函数**。如果基类有多个构造函数，则using语句会在派生类中为**每个基类构造函数生成一个与之对应的构造函数**，并具有与基类构造函数相同的访问权限。
- b) using**不受访问权限制约**，放在public、protected或private区域中没有区别。
- c) 用using在派生类中声明基类的构造函数和其它成员有所不同，**声明其它成员**只是使该成员在指定的派生类权限区域可见，**并不生成代码**。而用**using继承基类构造函数**，则会使编译器在派生类中**生成基类构造函数的一份副本**。
- d) 基类的默认构造函数、拷贝构造函数和移动构造函数**不能够被继承**
- e) 若派生类在继承基类构造函数的同时，还需要定义其它构造函数，必须在构造函数初始化列表中为基类构造函数提供初始化值（除非基类有默认构造函数）。

## 4.5.1 派生类构造函数的建立规则

- f) 如果基类构造函数具参数默认值，继承将为派生类生成多个构造函数，每个构造函数的参数依次少一个。例如，

```
class A {  
    int x, y;  
public:  
    A(int a , int b = 2) :x(a), y(b) { cout<< "a=" <<a <<"\tb="<<b<<endl;  
}  
};  
class B :public A {  
public:  
    using A::A;  
};
```

— 继承将为类B生成构造函数：

B(int a): **A(a,2)** 和 B(int a,int b): **A(a,b)**

## 4.5.2 派生类构造函数的定义

### • 1、派生类构造函数的设计原则

- 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。
  - **派生类只能采用构造函数初始化列表的方式向基类构造函数传递参数。**
  - **但派生类可以通过构造函数初始化列表，也可以通过类内初始值向对象成员传递构造参数。**
- 形式如下：

派生类构造函数名(参数表):基类构造函数名(参数表),成员对象名1(参数表),...{  
    //.....  
}

## 4.5.2 派生类构造函数和析构函数的调用次序

### 2、构造函数的调用原则和次序

- 创建派生类对象时，基类、对象成员及派生构造函数都会被调用，调用次序：

基类构造函数→对象成员构造函数→派生类构造函数

- (1) 当有多个基类时，按照它们在继承方式中的**声明次序调用**，与它们在构造函数初始化列表中的次序无关。
- (2) 当有多个对象成员时，将按它们在派生类中的**声明次序调用**，与它们在构造函数初始化列表中的次序无关。
- (3) 当构造函数初始化列表中的基类和对象成员的构造函数调用完成之后，才执行派生类构造函数体中的程序代码。

```
class A {};  
class B {};  
class C {  
    public:C(int) {}  
};  
class D :public A, B  
{  
    C c=C(1);  
    A a1, a2;  
public:  
    D():B(), A(), a2()  
    a1() {...}  
};
```

## 4.5.2 派生类构造函数和析构函数的调用次序

【例4-12】 类D从类B派生，并具有用类A和C建立的对象成员。分析创建D的对象时，基类、对象成员和派生类构造函数和析构函数的调用次序。

//Eg4-12.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
    int x;
```

```
public:
```

```
    A(int i=0):x(i){          cout<<"Construct A----"<<x<<endl;}
```

```
    ~A() { cout <<"Des A----"<<x<<endl; }
```

```
};
```

## 4.5.2 派生类构造函数和析构函数的调用次序

---

```
class B {  
    int y;  
  
public:  
    B(int i):y(i) { cout<<"Construct B----"<<y<<endl; }  
    ~B() { cout <<"Des B----"<<y<<endl; }  
  
};  
  
class C {  
    int z;  
  
public:  
    C(int i):z(i) { cout<<"Construct C----"<<z<<endl; }  
    ~C() { cout<<"Des C----"<<z<<endl; }  
  
};
```

## 4.5.2 派生类构造函数和析构函数的调用次序

```
class D : public B {  
public:  
    C c1, c2;  
    A a0, a4;  
    D():a4(4),c2(2),c1(1),B(1) {  
        cout<<"Construct D----5"  
        <<endl;  
    }  
    ~D() { cout<<"Des D----5"<<endl;  
};  
void main() {  
    D d;  
}
```

运行结果如下，分析每个输出的来源

```
Construct B----1  
Construct C----1  
Construct C----2  
Construct A----0  
Construct A----4  
Construct D----5  
Des D----5  
Des A----4  
Des A----0  
Des C----2  
Des C----1  
Des B----1
```



## 4.5.3 派生类的赋值、拷贝和移动操作

### 1. 派生类赋值、拷贝和移动操作对基类的职责

(1) **派生类**的赋值函数和拷贝构造函数，以及移动赋值和移动构造函数不但要执行派生类成员的拷贝和移动，而且**还要负责基类部分数据成员的拷贝和移动**。

(2) 如果一个类**没有定义**赋值运算、拷贝构造函数、移动赋值和移动构造函数，编译器将会为它们**自动**生成对应的函数版本。但以下两种情况除外：

- ① 当一个类**有虚析构函数**时，即使没有定义这些函数，编译器也不会合成它们。
- ② 如果一个类**定义了赋值运算符或拷贝构造函数**，编译器也不会为它合成移动赋值和移动构造函数。

(3) 派生类在定义赋值函数、拷贝构造函数和它们的移动函数版本时，**要负责对基类成员进行相应的处理**，即应当调用基类与之对应的赋值函数、拷贝构造函数和移动函数来完成基类成员的相应处理。

## 4.5.3 派生类的赋值、拷贝和移动操作

【例4-13】类A具有数据成员x，并定义了赋值函数，拷贝构造函数和它们的移动函数版本，以实现对象间的赋值、拷贝或移动操作，类B从类A派生，并有数据成员y。设计类B的赋值、拷贝构造函数和移动函数，实现派生类B的对象间的赋值、拷贝和移动操作。

设计思路：

根据前面的规则，当一个类设计了赋值运算符函数、拷贝构造函数和移动函数时，就需要在这些函数中提供对基类对应函数的初始化支持。因此，在类B的相应函数设计中，要提供对基类A对应函数的初始化列表。

## 4.5.3 派生类的赋值、拷贝和移动操作

### //Eg4-13.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
    int x;
```

```
public:
```

```
    A(int a =0, int b = 2) :x(a){}
```

```
    A &operator=(A& o) {
```

```
        x = o.x;
```

```
        cout << "In A =(A&)" << endl;
```

```
        return *this;
```

```
    }
```

```
    A& operator=(A &&o) = default; //使用默认的合成移动赋值函数
```

```
    A(A &o):x(o.x) { cout << "In A(&)"<<endl; }
```

```
    A(A &&o):x(std::move(o.x)) { cout<<"In A(&&)"<<endl; }
```

```
};
```

## 4.5.3 派生类的赋值、拷贝和移动操作

```
class B :public A {
    int y;
public:
    B(int a=0, int b=0) :A(a),y(b){}
    B& operator=(B& o) {
        A::operator=(o);
        cout<<"In B=(B&)"<<endl;
        return *this;
    }
    B& operator=(B &&o) { A::operator=(std::move(o));
        cout << "In B =(B&&)" << endl;
        return *this;
    }
    B(B &o):A(o) { cout << "In B(&)" << endl; }
    B(B &&o):A(std::move(o)) { cout<<"In B(&&)"<< endl; }
};
```

## 4.5.3 派生类的赋值、拷贝和移动操作

```
void main() {  
    B b, b1(1,2);  
    b = b1;                //L1  
    B b2(b);               //L2  
    B b3=std::move(B(8, 9)); //L3  
    b1 =std::move(b3);      //L4  
}
```

程序运行结果如下：

In A =(A&)	//L1的输出
In B =(B&)	//L1的输出
In A(&)	//L2的输出
In B(&)	//L2的输出
In A(&&)	//L3的输出
In B(&&)	//L3的输出
In B =(B&&)	//L4的输出

## 4.6 基类与派生类对象的关系

---

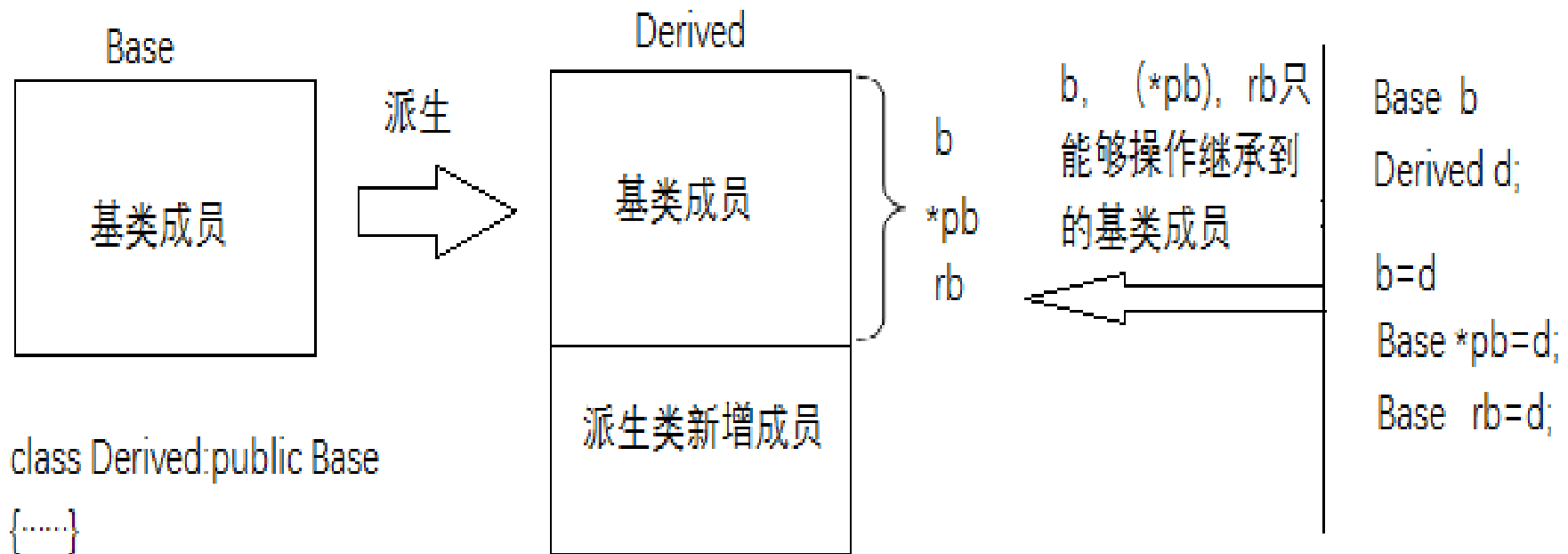
### 1. 派生对象与基类对象的赋值相容关系

- 派生类通过继承获得了基类成员的一份拷贝，这份拷贝构成了派生类对象内部的一个基类子对象。
- 因此，**公有派生方式下，凡是需要基类对象的地方都可以使用派生类对象**。基类对象能够解决的问题，用派生类对象也能够解决。称为赋值相容。包括下面三种情况：
  - ① 把派生类对象赋值给基类对象；
  - ② 把派生类对象的地址赋值给基类指针；
  - ③ 或者用派生类对象初始化基类对象的引用。

# 4.6 基类与派生类对象的关系

## 2. 派生类与基类赋值相容的处理方式

- 因为任何一个派生类对象的内部都包含有一个基类子对象，在进行派生类对象向基类对象的赋值时，C++采用截取的方法从派生类对象中复制其基类子对象并将之赋值给基类对象。



## 4.6.1 派生类对象对基类对象的赋值和初始化

- 派生类中基类之间的对象复制关系

- 以下两种操作并不存在从派生类向基类的类型转换，本质上是执行基类对象的复制构造函数或赋值运算符函数，通过它们把派生类对象中从基类继承到的数据成员复制给基类对象

① 在把派生类对象赋值给基类对象

② 用派生类对象初始化基类对象

注意：不存在基类对象向派生类对象的复制关系

**【例4-14】** 类B从类A派生，设计类B的复制构造函数和赋值运算符函数，并验证把派生对象赋值给基类对象或通过它初始化基类对象时，相关函数的调用情况。



//Eg4-14.cpp

#include <iostream>

using namespace std;

class A {

int a;

public:

void setA(int x) { a = x; }

int getA() { return a; }

A() :a(0) { cout<< "A::A()"<<endl; }

A(A& o):a(o.a) { cout<<"A::A(&o)"<<endl; }

A& operator=(A o)

{ a=o.a; cout<< "A::operaotor="<<endl; return \*this; }

};

class B :**public** A {

int b;

public:

void setB(int x) { b = x; }

int getB() { return b; }

B():b(0) { cout << "B::B()" << endl; }

B(B& o):b(o.b) { cout << "B::B(&o)" << endl; }

B& operator=(B o)

{ b=o.b; cout<<"B::operaotor="<<endl; return \*this; }

};

## 4.6.1 派生类对象对基类对象的赋值和初始化

```
void main() {  
    A a1, *pA;  
    B b1, *pB;  
    b1.setA(2);  
    a1 = b1;  
    b1.setA(10);  
    A a2 = b1;  
    a2.setA(1);  
    cout << a1.getA() << endl;//L1, 输出 2  
    cout << b1.getA() << endl;//L2, 输出 10  
    cout << a2.getA() << endl;//L3, 输出 1  
    //a2.setB(5);           //L4, 错误  
    //b1 = a1;              //L5, 错误  
}
```

a1

a

b1

a

b

a2

a

程序运行结果如下:

A::A()

A::A()

B::B()

A::A(&o)

A::operator=

A::A(&o)

2

10

1

请据上面的复制和赋值原则, 分析此程序结果的函数调用情况

## 4.6.2 派生类对象与基类对象的类型转换

### 1. 派生类和基类之间的类型转换关系

(1) 可以把派生类对象转换成基类对象，不能把基类对象转换成派生类对象  
(无法转换出派生类新增加的成员)

#### (2) 派生类对象到基类对象的隐式类型转换

- 用派生类对象赋值或初始化基类对象时，实际是通过赋值运算符函数或拷贝构造函数完成的，并没有执行类型转换；当把基类对象的指针或引用绑定到派生对象时，编译器会自动执行从派生类对象到基类对象的隐式类型转换。

例如，对于例4-14的基类A和派生类B，下面的语句段会发生类型转换。

B b,b1,b2;

A \*pa=&b1;               //正确，执行派生类向基类的转换

A &rA=b2               //正确，执行派生类向基类的转换

A a=b;               //正确，没有类型转换，通过基类拷贝构造函数初始化a

#### • 注意：

不论以哪种方式把派生类对象赋值给基类对象，都只能够访问到派生类对象中的基类子对象的成员，不能访问派生类的自定义成员

## 4.6.2 派生类对象与基类对象的类型转换

### (3) 基类对象到派生类对象的类型转换

- 实际上，不能把基类对象直接转换成派生类对象。但是，当基类对象的指针或引用实际绑定的是一个派生类对象时，则可以将它再次转换成派生类对象。
- 若要进行上面所说的类型转换，只能进行强制类型转换，编译器是不会进行这种转换的隐式转换的。
- 例如，对例4-14中的基类A和派生类B，

```
A a, *pa;
```

```
B b, b1,b2,*pb,
```

```
pa=&b1;           //正确，执行派生类向基类的转换
```

```
A &rA=b2          //正确，执行派生类向基类的转换
```

```
b=a;             //错误，不允许从基类向派生类的转换
```

```
pb=pa;           //错误，不能把基类对象的地址赋值给指向派生类对象的指针
```

```
B &rB=rA;         //错误，不能把基类对象作为派生类对象的引用
```

```
pb =static_cast<B*> (pa);    //正确，强制转换
```

```
B &rB =static_cast<B&> (rA);  //正确，强制转换
```

## 4.6.2 派生类对象与基类对象的类型转换

### 3. 对象、指针和引用的区别

- 把派生类对象赋值给基类对象或用派生类对象初始化基类对象，完成赋值或初始化操作后，**基类对象与派生对象就没有关系了**
- 把基类对象的指针或引用绑定到派生类对象时，指针或引用从来就没有生成新对象，它们**操作的是派生类对象内部的基类子对象。**
- 如对例4-14的基类A和派生类B，有下面的代码段

```
void main() {  
    B b,b1;  
    A a=b,*pa = &b1, &rA = b1;    //L1  
    b.setA(10);                    //L2  
    a.setA(9);                     //L3  
    pa->setA(20);                   //L4  
    rA.setA(1);                     //L5  
    cout << b.getA();               //L6, 输出10, 并未受L3的影响  
    cout << b1.getA();             //L7, 输出1, L5设置引起的值变化  
}
```

## 4.6.2 派生类对象与基类对象的类型转换

### 4. 派生类对象作为函数参数传递给基类对象

如果函数形式参数是基类对象，也可以用派生类对象作为实参。

— 比如，对例4-14的类A和类B，有下面的程序段

```
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); }
void main() {
    B b;
    b.setA(1);
    f1(b,10);           //b.a未被f1修改,仍然为1
    f2(&b,10);          //b.a被f2修改为10
    f3(b,15);           //b.a被f3修改为15
}
```

## 4.6.2 派生类对象与基类对象的类型转换

---

### (1) 形参是基类对象

- 不能修改实参对象的值。
- 参数传递方式：调用实参对象（若实参是派生类对象，则调用该对象的基类拷贝构造函数）的拷贝构造函数把实参的数据成员复制给形参对象，参数传递完成后，实参和形参就没有关系了，因此不能改变实参对象的值。

### (2) 形参是基类对象的引用或指针

- 能够修改实参对象的值。
- 参数传递方式：将形参绑定到实参对象（若形参是派生类对象，编译器将进行隐式类型转换，形参引用或指针被绑定到派生类实参对象内部的基类子对象上），形参操作的实际上是实参对象本身。因此，这两种参数传递方式都能够修改实参对象的值。

# 4.7 多重继承

## 4.7.1 多继承的概念和应用

### 1. 概念

- 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**。

### 2. 多继承的形式如下：

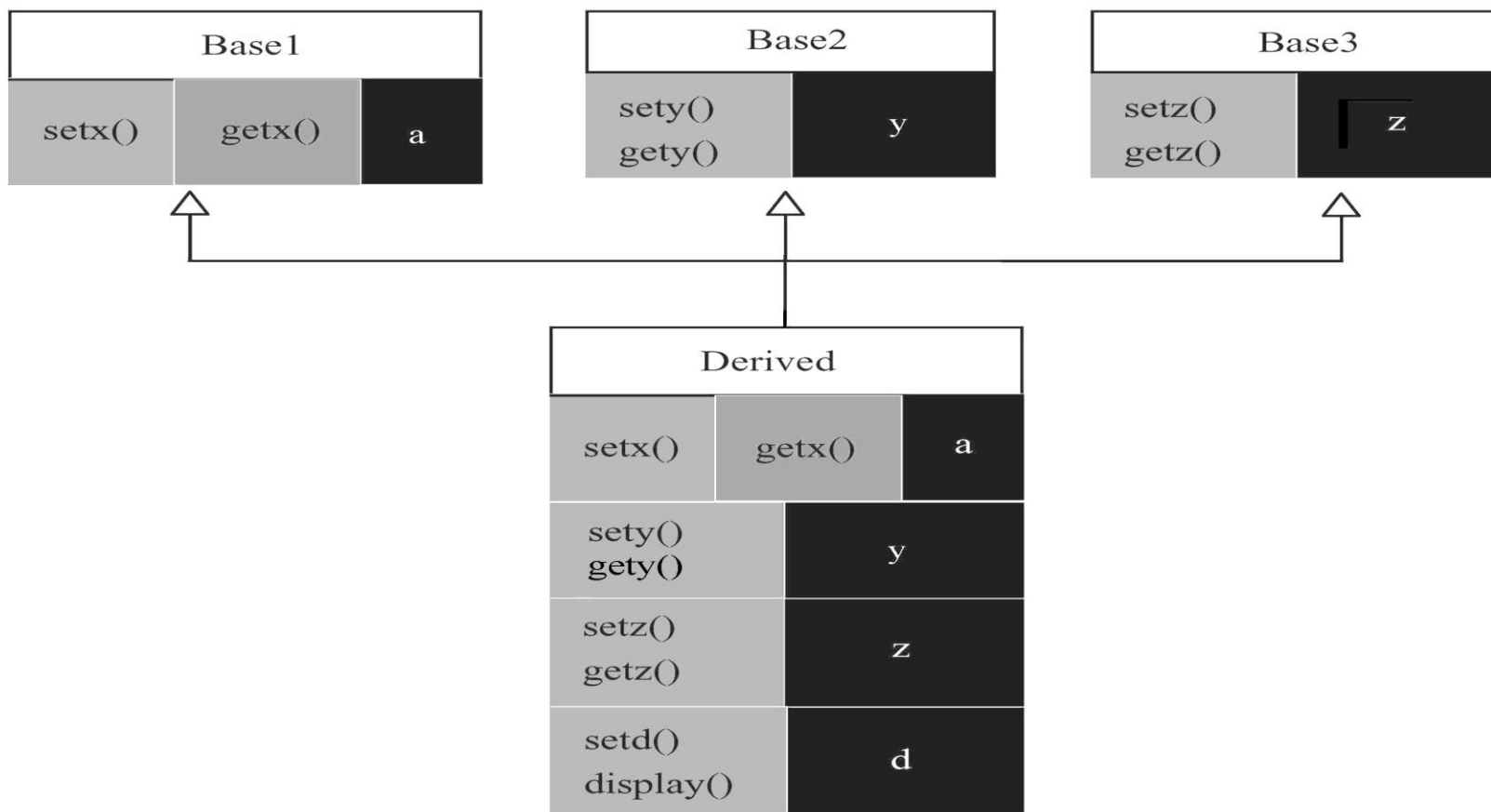
```
class 派生类名:[继承方式] 基类名1,[继承方式] 基类名2,  
...  
{  
    .....  
};
```

- 其中，继承方式可以是public、protected、private



## 4.7.1 多继承的概念和应用

【例4-15】假设有3个类Base1、Base2、Base3，其中Base1有公有成员函数setx，保护成员函数getx，私有数据成员a；Base2有公有成员函数sety和gety，私有数据成员y；Base3有公有成员函数setz和getz，私有成员z；类Derived从这3个类派生，且有自己的公有成员函数setd、display和私有数据成员d



## 4.7.1 多继承的概念和应用

---

【例4-15】 上图的简单程序。

**//Eg4-15.cpp**

**#include <iostream>**

**using namespace std;**

**class Base1{**

**private:**

**int x;**

**protected:**

**int getx(){ return x; }**

**public:**

**void setx(int a=1){ x=a; }**

**};**

## 4.7.1 多继承的概念和应用

---

```
class Base2{
private:
    int y;
public:
    void sety(int a){ y=a; }
    int gety(){ return y; }
};
class Base3{
private:
    int z;
public:
    void setz(int a){ z=a; }
    int getz(){ return z; }
};
```

## 4.7.1 多继承的概念和应用

```
class Derived:public Base1,public Base2,public Base3{
private:
    int d;
public:
    void setd(int a){ d=a; }
    void display();
};

void Derived::display(){
    cout<<"Base1....x="<<getx()<<endl;
    cout<<"Base2....y="<<gety()<<endl;
    cout<<"Base3....z="<<getz()<<endl;
    cout<<"Derived..d="<<d<<endl;
}

void main(){
    Derived obj;
    obj.setx(1);
    obj.sety(2);
    obj.setz(3);
    obj.setd(4);
    obj.display();
}
```

运行结果如下：

Base1....x=1

Base2....y=2

Base3....z=3

Derived..d=4

Derived类通过多继承具备了3个基类的成员，即使没有添加任何程序代码，也具有3个基类合起来才有的强大功能！

## 4.7.2 多重继承方式下成员名的二义性

---

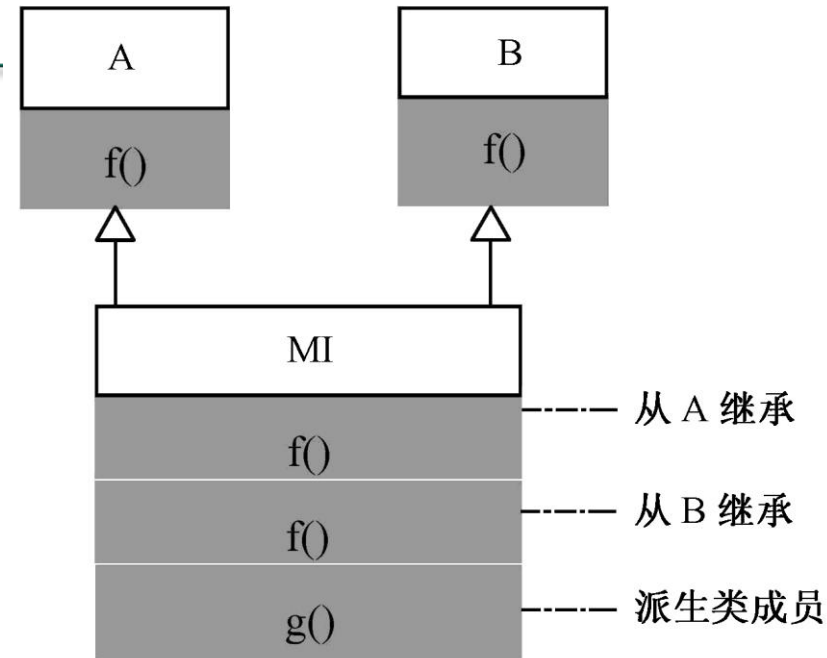
- 在多继承方式下，派生类继承了多个基类的成员，当两个不同基类拥有同名成员时，容易产生名字冲突问题。

**【例4-16】** 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，在调用时易产生冲突。

```

//Eg4-16.cpp
#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();           //L1: 错误
    mi.A::f();        //L2: 正确
}

```



从MI的类图可以看出，它有2个f()函数，因此在调用时，需要像L2语句那样指出调用函数出自的基类

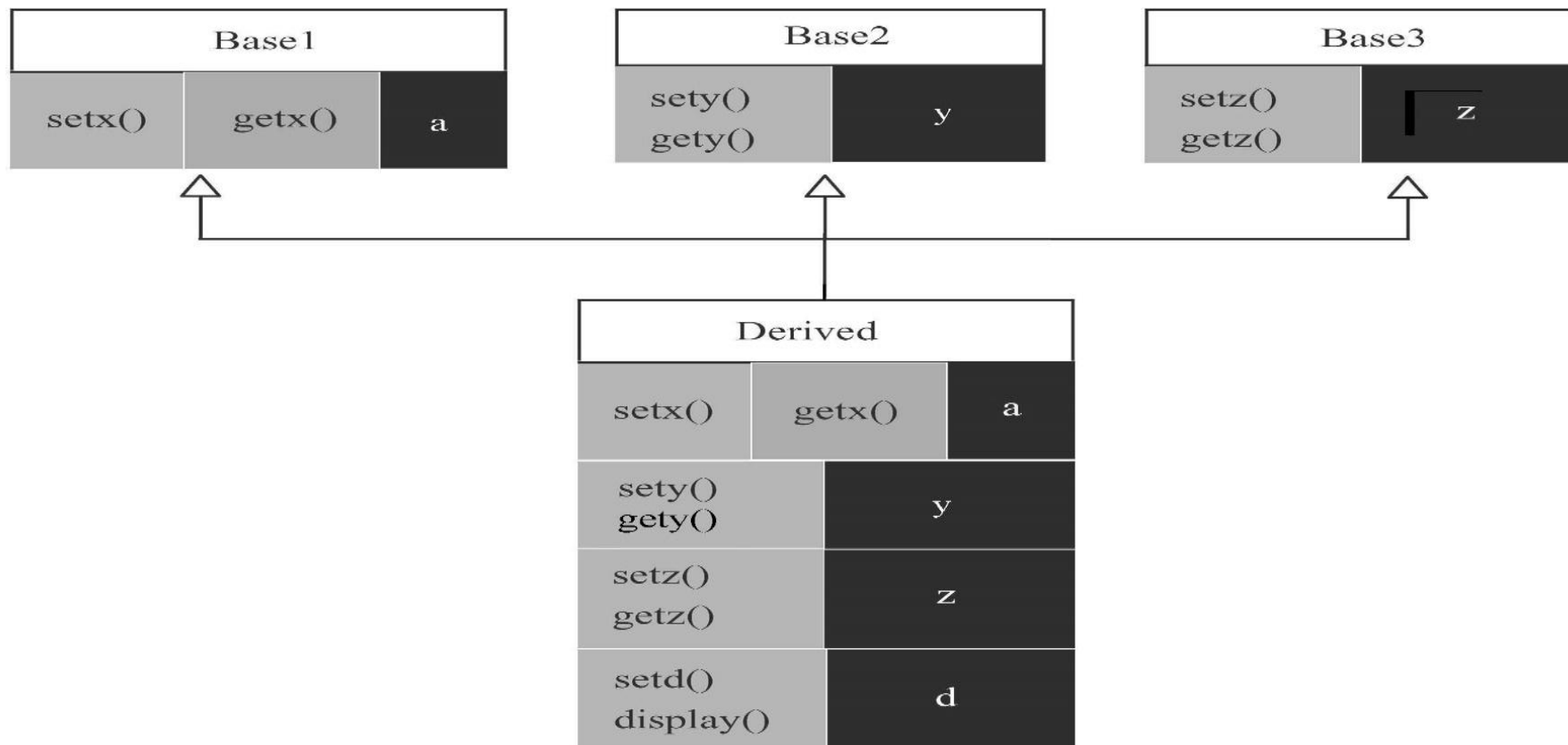
## 4.7.3 多继承的构造函数与析构函数

---

1. 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。
2. 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。
3. 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。
4. 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。

## 4.7.3 多继承的构造函数与析构函数

【例4-17】 类Base1、Base2、Base3、Derived的继承关系如图所示，验证其构造函数和析构函数的调用次序。





## 4.7.3 多继承的构造函数与析构函数

---

```
//Eg4-17.cpp
#include <iostream>
using namespace std;
class Base1{
private:
    int x;
public:
    Base1(int a=1){
        x=a;
        cout<<"Base1 constructor x="<<x<<endl;
    }
    ~Base1(){ cout<<"Base1 destructor..."<<endl; }
};
```

```
class Base2{
```

```
private:
```

```
    int y;
```

```
public:
```

```
    Base2(int a){
```

```
        y=a;
```

```
        cout<<"Base2 constructor y="<<y<<endl;
```

```
    }
```

```
    ~Base2(){ cout<<"Base2 destructor..."<<endl; }
```

```
};
```

```
class Base3{
```

```
private:
```

```
    int z;
```

```
public:
```

```
    Base3(int a){
```

```
        z=a;
```

```
        cout<<"Base3 constructor z="<<z<<endl;
```

```
    }
```

```
    ~Base3(){ cout<<"Base3 destructor..."<<endl; }
```

```
};
```

## 4.7.3 多继承的构造函数与析构函数

```
class Derived:public Base1,protected Base2,private Base3{
```

```
private:
```

```
    int y;
```

```
public:
```

```
    Derived(int a,int b,int c)
```

```
        :Base3(b),Base2(a){
```

```
            y=c;
```

```
            cout<<"Derived constructor y="
```

```
                <<y<<endl;
```

```
        }
```

```
        ~Derived(){ cout<<"Derived destructor..."
```

```
                <<endl; }
```

```
};
```

```
void main(){
```

```
    Derived d(2,3,4);
```

```
}
```

本程序的运行结果如下：

Base1 constructor x=1

Base2 constructor y=2

Base3 constructor z=3

Derived constructor y=4

Derived destructor...

Base3 destructor...

Base2 destructor...

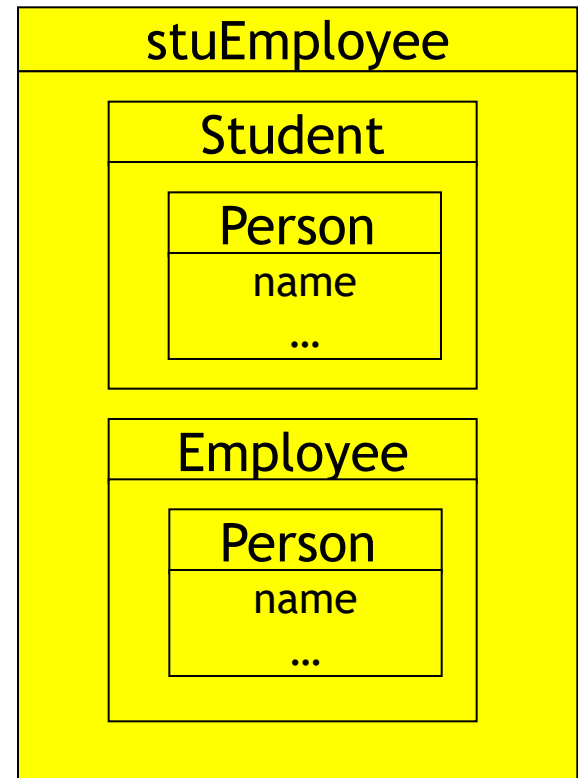
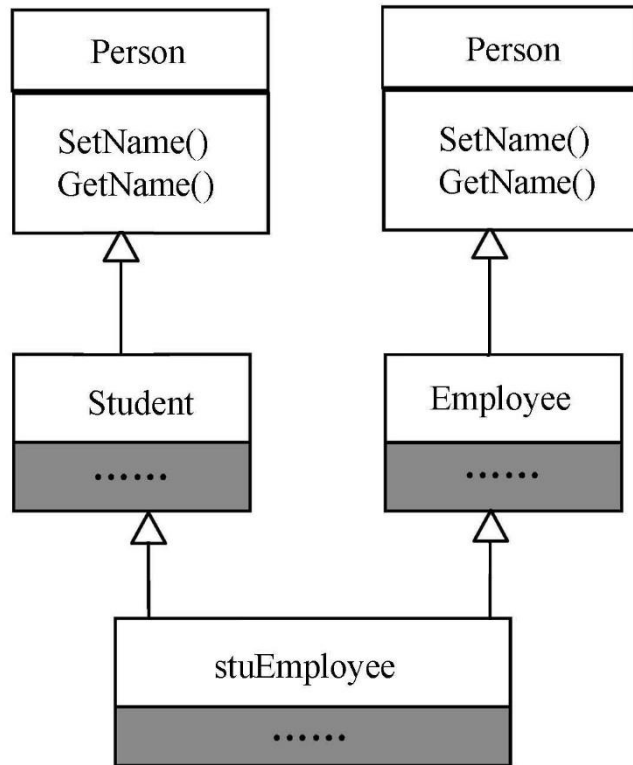
Base1 destructor...

分析此结果的产生过程！

# 4.8 虚拟继承

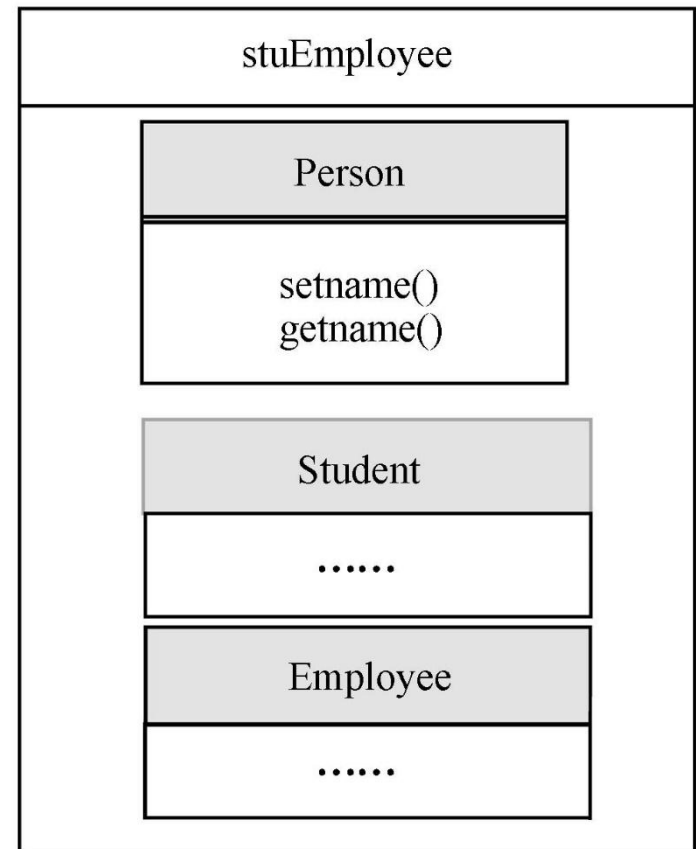
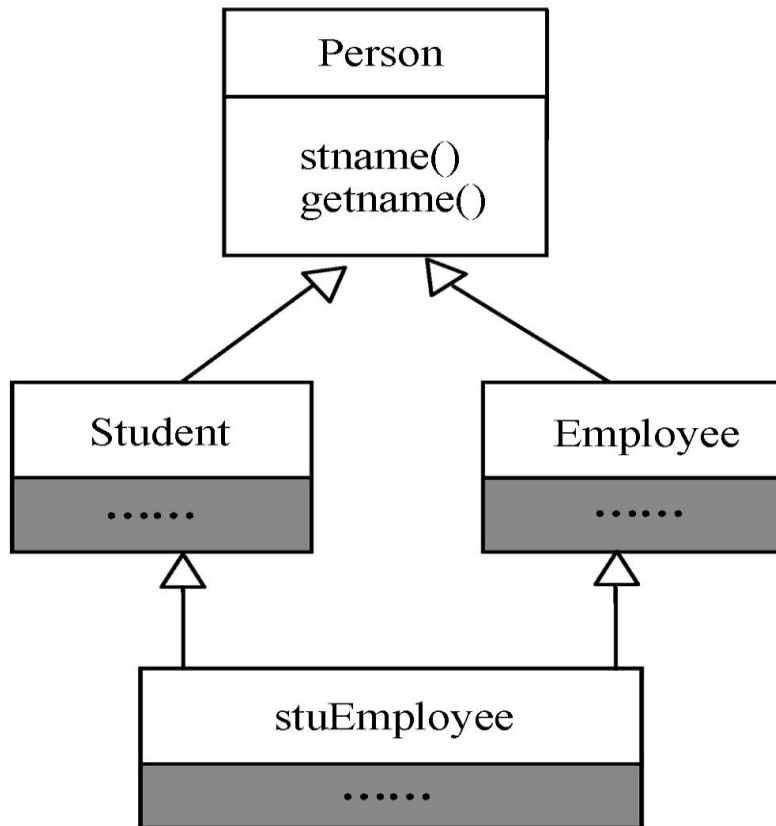
## 1. 引入虚拟继承的原因

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。



## 4.8 虚拟继承

- 虚拟继承使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



# 4.8 虚拟继承

## 2. 虚拟继承的定义方式

class 派生类名: **virtual** [继承方式] 基类名1, **virtual** [继承方式]  
基类名2,...{

派生类成员声明与定义;

};

- 关键字**virtual**限定继承方式，将基类指定为虚拟基类，就使**该基类的成员在派生类中只有一份拷贝**。
- 前面的**stuEmployee**类虚拟继承**Person**的形式如下：

```
class Student: virtual public Person{.....}    //Person为虚基类
class Employee: virtual public Person{.....}    //Person为虚基类
class StuEmployee:public Student,public Employee{.....}
```

**【例】** 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{};
```

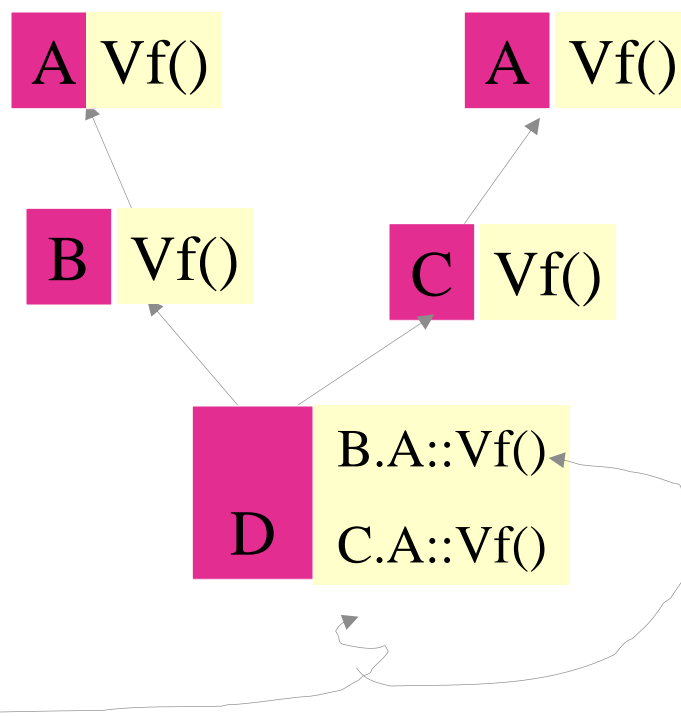
```
class D: public B, public C{};
```

```
void main()  
{
```

```
    D d;
```

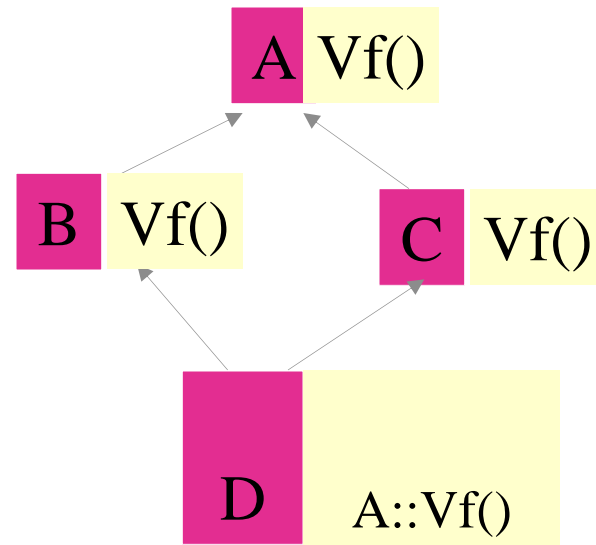
```
    d.vf ();    // error
```

```
}
```



将【例】 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{};  
  
void main()  
{  
    D d;  
    d.vf();    // okay  
}
```





## 4.8 虚拟继承

### 3、虚拟继承的构造次序

- 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**）
- 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**；
  - ① 先调用虚基类的构造函数，再调用非虚基类的构造函数
  - ② 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继承次序调用
  - ③ 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

## 4.8 虚拟继承

【例4-18】 虚基类的执行次序分析。

```
//Eg4-18.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
    int a;
```

```
public:
```

```
    A(){ cout<<"Constructing A"<<endl; }
```

```
};
```

```
class B {
```

```
public:
```

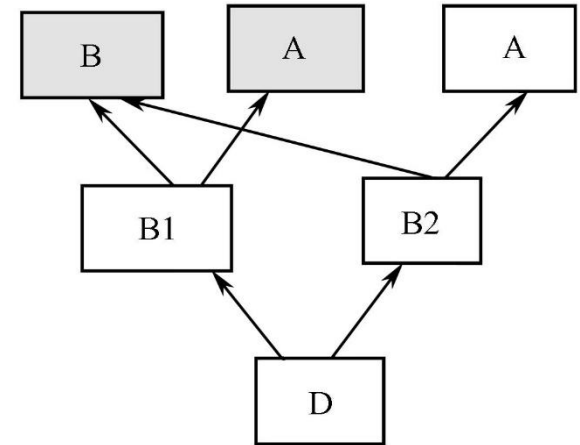
```
    B(){ cout<<"Constructing B"<<endl;}
```

```
};
```

## 4.8 虚拟继承

```
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;
};

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



程序的运行结果如下：

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

此运行结果表明：D的间接虚拟基类B只被构造了1次

# 4.8 虚拟继承

## 4、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。

## 4.8 虚拟继承

**【例4-19】** 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，它必须负责虚基类A的初始化。

```
//Eg4-19.cpp
#include <iostream.h>
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; }
```

```
};
```

```
void main(){
```

```
    ABC obj(1,2,3);
```

```
}
```

程序的运行结果如下:

**Virtual Bass A...**

**Constructing C...**

**Virtual Bass B...**

**Constructing ABC...**  
**B**

派生类**ABC**必须为其祖先类**A**提供构造函数初始化列表，否则报编译错误！

## 4.8 虚拟继承

### 5、成员函数冲突与优先级

- 如果虚基类和派生类中都有同名成员函数，仍然有可能**会产生命名冲突**。
- 假设类A具有函数f，类B和C都从A虚拟派生，类ABC继承了类B和类C，详见例4-19。对于ABC类对象的f成员函数调用，存在以下几种情况：
  - ① 如果类B、C和ABC都没有定义f函数，在类ABC的对象中只有1个来源于虚基类A中的f函数，没有冲突。但是，若**B、C都不是虚拟继承于A**；或者其中**一个虚拟继承于A，另一个非虚拟继承于A**；则在ABC对象中的f函数有多个，**会产生冲突**。
  - ② 如果类B、C中有一个定义了f函数，则在调用ABC对象的f函数时，B或C中的f函数具有优先权。例4-19中“obj.f()”的输出证明它调用的是类B中的f函数。
  - ③ 不论上面哪种情况，如果ABC类定义了f函数，当ABC的对象调用f函数时，不会有冲突，ABC中的f函数具有优先权。

# 4.9 继承与组合

## 1、继承与组合在OOP中的应用场景

- 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。

## 2、继承解决的主要问题

- 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。

## 3、组合解决的主要问题

- 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。



## 4.9 继承与组合

【例4-20】设计学生选学课程程序，要求能够管理指定课程的选课学生名单。

### (1) 问题分析

- 在本问题中，主要涉及的对象包括学生、课程，可以设计**课程类**，**学生类**分别管理课程信息和学生信息。
- 学生选修课程则和课程和学生都有关系，可以设计**选修课程类**来管理它。由于选修时需要知道选修的课程信息和学生名单。这一关系可以用**类的包含关系**处理，即选修课程类的内部包含课程和学生类的对象。

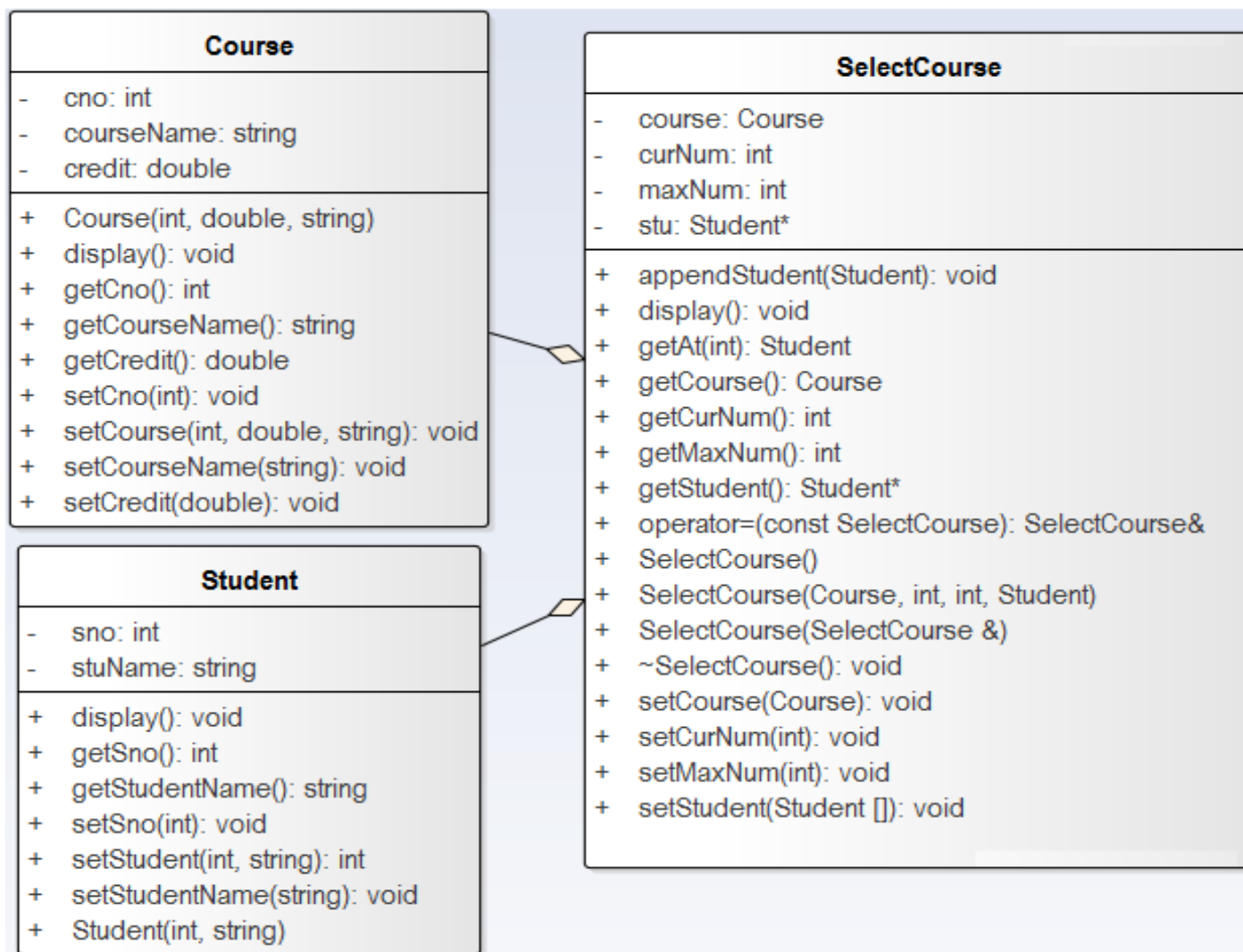
## 4.9 继承与组合

### (2) 数据抽象

- 用**Coure**表示课程类，包括课程编号、学分和课程名称，分别用courseName, cno和 credit表示它们，并设计 setCno/getCno, setCredit/getCredit.....来设置和读取数据成员的值，能够一次性设置全体数据成员值的setCourse成员函数和默认构造函数，以及显示成员值的display函数。
- 用**Student**表示学生类，分别用sno和stuName表示学号和姓名。用setSno/getSno.....设置和读取sno和stuName的值，用display函数显示成员的值。
- 课程选修类是本题的**核心类**，用**SelectCourse**表示，用数据成员course、maxNum、curNum分别表示选学的课程、最多允许选课的人数、实际选课人数，用指针成员stu存取选课学生名单，它指向Student类型的动态数组，该数组由构造函数分配。

## 4.9 继承与组合

### (3) 数据抽象结果



```
#include <iostream>
```

```
#include<string>
```

```
using namespace std;
```

```
class Course { //编译器会为该类生成合成的拷贝构造函数和赋值运算符函数
```

```
public:
```

```
    void setCno(int cNumber) { cno = cNumber; }
```

```
    int getCno() { return cno; }
```

```
    void setCred(double crd) { credit = crd; }
```

```
    double getCred() { return credit; }
```

```
    void setName(string cname) { courseName = cname; }
```

```
    string getCourseName(){return courseName; }
```

```
    Course(int Cno=0, double cre=0, string cName="")
```

```
    { setCourse(Cno, cre, cName); }
```

```
    void display() {
```

```
        cout<<"课程号: "<<cno<<"\t课程名称: "<<courseName
```

```
                <<"\t学分: " <<credit<<endl; }
```

```
    void setCourse(int Cno = 0, double cre = 0, string cName = "")
```

```
    { cno = Cno; credit = cre; courseName = cName; }
```

```
private:
```

```
    int cno;
```

```
    double credit;
```

```
    string courseName;
```

```
};
```

## 4.9 继承与组合

//student的程序代码，编译器会为该类合成默认的拷贝构造函数和赋值函数

```
class Student {  
public:  
    void setSno(int Snumber) { sno = Snumber; }  
    int getSno() { return sno; }  
    void setStudentName(string Sname) { stuName = Sname; }  
    string getStudentName() { return stuName; }  
    Student(int Sno = 0, string SName = "")  
        { setStudent(Sno, SName); }  
    void display() {cout<<"学号: "<<sno <<"\t姓名: "<<stuName<<endl;}  
    void setStudent(int Sno = 0, string Sname = "")  
        { sno = Sno; stuName = Sname; }  
private:  
    int sno;  
    string stuName;  
};
```

```
class SelectCourse {
```

```
private:
```

```
    int maxNum=10, curNum=0;
```

```
    Course course;
```

```
    Student *stu=nullptr;
```

```
public:
```

```
    void setCourse(Course c) { course = c; }
```

```
    Course getCourse(){return course;}
```

```
    void setMaxNum(int n) { maxNum = n; }
```

```
    int getMaxNum() { return maxNum; }
```

```
    void setCurNum(int n) { curNum = n; }
```

```
    int getCurNum() { return curNum; }
```

```
    Student* getStudent() { return stu; }
```

```
    void setStudent(Student s[]) { stu = s; }
```

```
    Student getAt(int n) { return stu[n]; }
```

```
    void appenStudent(Student s){if(curNum<maxNum)stu[curNum++]=s; }
```

```
    void display() {
```

```
        course.display();
```

```
        cout<< "最多选课人数:" << maxNum << "\t实选人数:" << curNum << endl;
```

```
        cout<< "选课学生名单:" << endl;
```

```
        for (int i = 0; i < curNum;i++)
```

```
            stu[i].display();
```

```
    }
```

```
SelectCourse() { stu = new Student[maxNum]; }
```

```
SelectCourse(Course c,int mNum,int cNum ,Student s[])
```

```
    :course(c),maxNum(mNum),curNum(cNum),stu(new Student[maxNum])
```

```
    { for (int i = 0; i < cNum; i++)          stu[i] = s[i]; }
```

```
SelectCourse(const SelectCourse &o):course(o.course)
```

```
    ,maxNum(o.maxNum), curNum(o.curNum) {
```

```
        stu = new Student[o.maxNum];
```

```
        for (int i = 0; i < o.curNum; i++)          stu[i] = o.stu[i];
```

```
    }
```

```
SelectCourse& operator=(const SelectCourse o) {
```

```
    course = o.course;
```

```
    maxNum = o.maxNum;
```

```
    curNum = o.curNum;
```

```
    for (int i = 0; i < o.curNum; i++)
```

```
        stu[i] = o.stu[i];
```

```
    return *this;
```

```
}
```

```
~SelectCourse() { delete []stu; }
```

```
};
```

```
void main(){
```

```
//下面的代码段，测试SelectCourse类构造函数和显示函数的运行情况
```

```
    Course course;  
    course.setCourse(101, 3.5, "C++面向对象程序设计");  
    Student s[2],s1;  
    s[0].setStudent(10, "高大山");  
    s[1].setStudent(11, "李明育");  
    SelectCourse sc(course,10,2,s);  
    cout<<"-----sc-----"<<endl;  
    sc.display();
```

```
//下面的代码段测试SelectCourse类的拷贝构造函数和添加选课学生函数的运行情况
```

```
    SelectCourse sc2, sc1 = sc;  
    s1.setStudent(14,"黄始仁");  
    sc1.appenStudent(s1);  
    cout<<"-----sc1(sc)-----"<<endl;  
    sc1.display();
```

```
//下面的代码段测试SelectCourse类的赋值运算符函数的运行情况
```

```
    sc2 = sc1;  
    cout <<"-----sc2=sc1-----"<<endl;  
    sc2.display();
```

```
//下面的代码段测试SelectCourse类中获取学生名单和人数的成员函数的运行情况
```

```
    Student *sname = sc2.getStudent();  
    cout <<"-----sc2.getStudent()-----"<<endl;  
    for (int i = 0; i < sc2.getCurNum();i++)  
        (sname++)->display();
```

```
}
```



## 4.10 编程实作：继承编程应用

【例4-21】某校每位学生都要学习英语、语文、数学三门公共课程以及不同的专业课程。会计学专业要学习会计学 and 经济学两门课程，化学专业要学习有机化学和化学分析两门课程。编程序管理学生成绩，计算公共课的总分和平均分，以及所有课程的总成绩。

### (1) 问题分析

- 在问题描述中，由于英语、语文、数学三门公共课程是所有学生都要学习的，可以将它抽象成为一个**基类comFinal**，由它来管理这三门基础课程的成绩。另外两个专业则分别抽象成类**Account**和**Chemistry**，分别管理会计学 and 化学两专业的课程成绩。
- 在整个问题域中还涉及学生，应该抽象出学生类**Student**来管理学生的档案。为简化问题，此处忽略了学生类的设计，仅用一个姓名都代表学生，并将此名字作为**comFinal**类的一个数据成员。

## 4.10 编程实作：继承编程应用

### (2) 数据抽象

– 基类`comFinal`抽象

- ① 用`name`、`english`、`chinese`、`math`分别表示学生姓名、英语、语文和数学成绩，并为每个数据成员设计`set/get`成员函数以修改/读取其值
- ② 设计成员函数`getTotal`、`getAverage`、`show`，分别用于计算总分、平均分、输出学生的各科成绩。设计构造函数`ComFinal`为各数据成员提供初始化值。
- ③ 由于设计了具有参数的构造函数，编译器就不会再为本类合成默认构造函数，为了便于`ComFinal`的继承及构造，以及本类无参对象和数组的定义，重定义了本类的默认构造函数。

## 4.10 编程实作：继承编程应用

- 会计学**Account**抽象

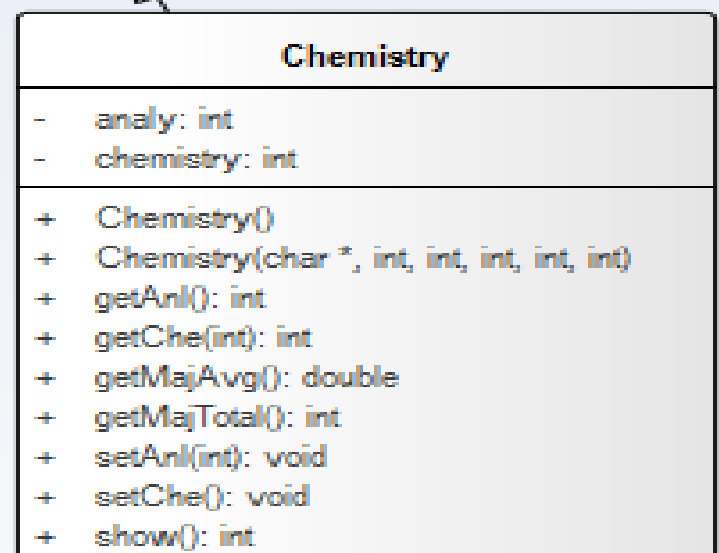
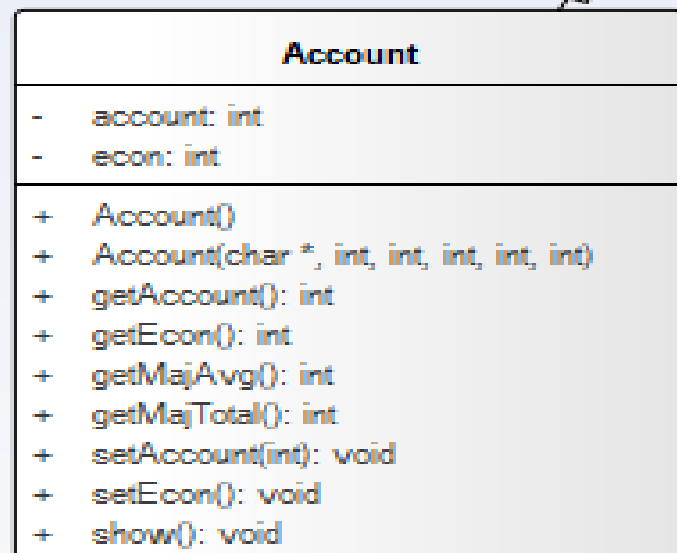
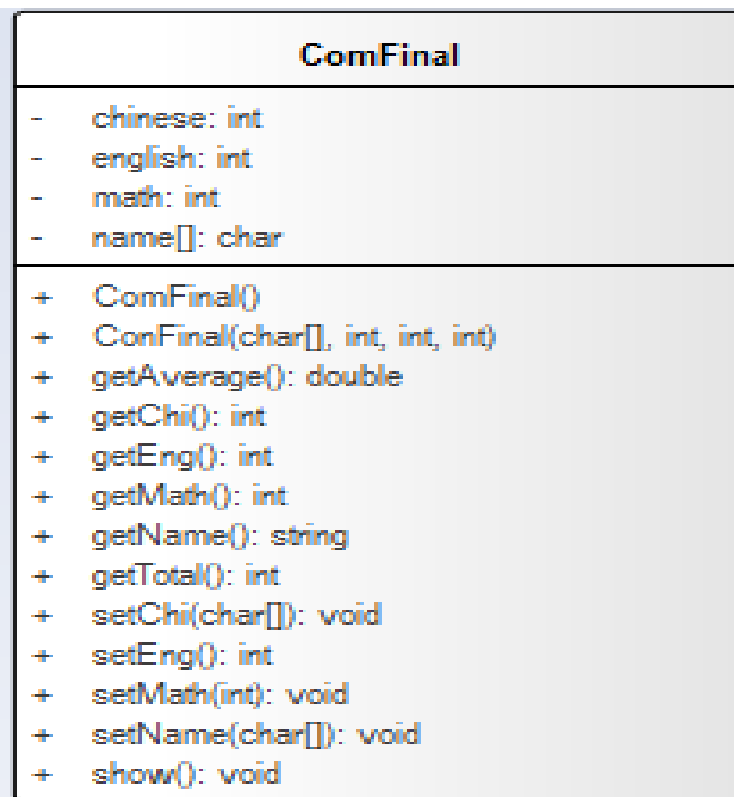
- ① 从**ComFinal**派生，继承基类的姓名及公共课处理能力，需要在构造函数初始化列表中对**ComFinal**类提供构造初值。
- ② 分别用数据成员**account**和**econ**表示会计学 and 经济学两门主科，并为之设置**set/get**成员函数；
- ③ 设计成员函数**getMajTotal**、**getMajAvg**和**show**，分别用来计算两主科的总分、平均分，以及显示输出学生的各项成绩数据。

- 化学类**Chemistry**抽象

- ① 用数据成员**analy**和 **chemistr**表示化学化析和化学两门主科
- ② 按照与设计**Account**类相同的方法设计**Chemistry**类。

- 三个类的**构造函数**、**析构函数**和**赋值运算符函数**设计

- ① 由于三个类都没有指针成员，也没有在构造函数中为任何数据成员分配动态存储空间，**不必设计析构函数**中进行动态存储空间的回收。
- ② 由于没有指针成员需要特别处理，也**不必定义拷贝构造函数、赋值运算符函数和析构函数**。编译器会为它们生成对应的默认合成函数，这些合成函数能够正确完成对象的构造、拷贝、赋值和析构。



# 4.10 编程实作： 继承编程应用

## (3) 编程过程

### 1. 在项目中添加各类的空头文件和源码文件

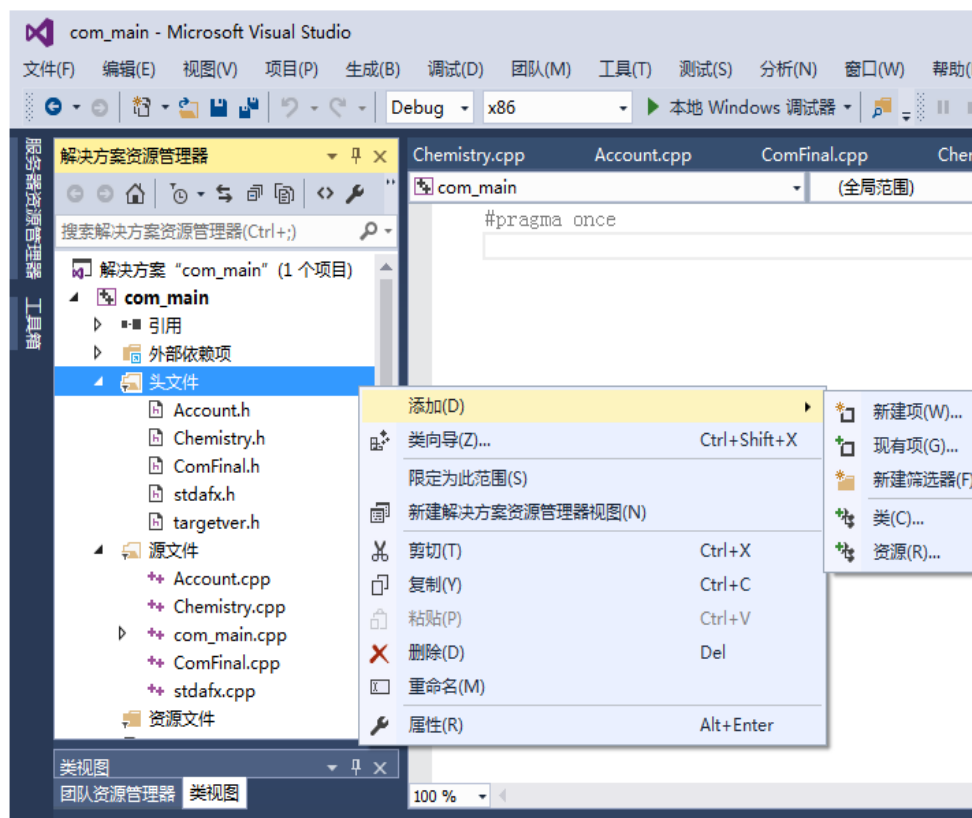
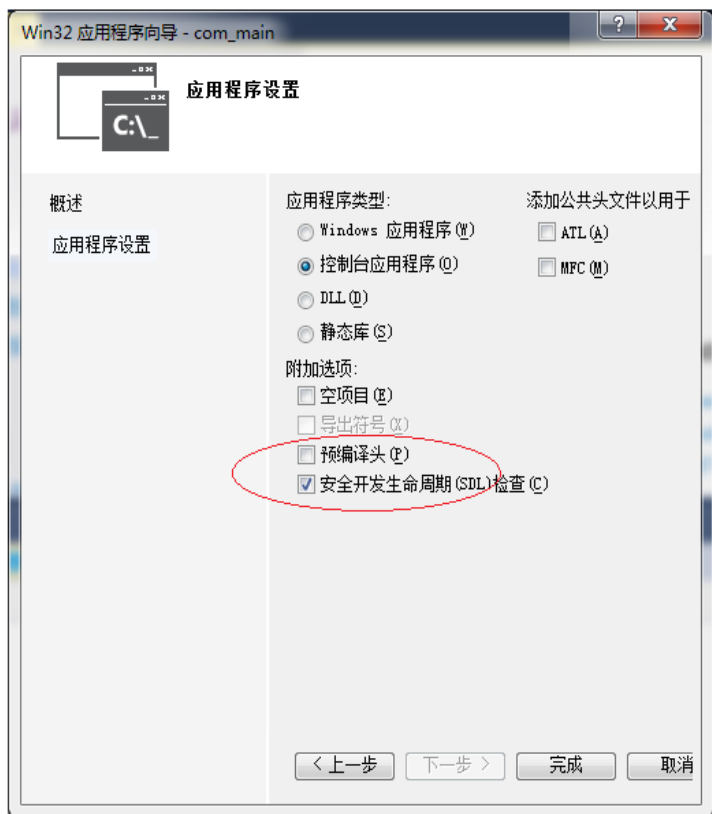
(1) 在C盘建立目录C:\course，用于保存本程序中的所有文件。然后启动Visual C++ 2015，选择“新建”|“项目”|“Visual C++”，在弹出的“新建项目”对话框中，指定“位置”为“c:\course”，在“名称”编辑框中输入“com\_main”，单击“确定”。

(2) 在弹出的向导对话中，单击“下一步”，显示出“应用程序设置”对话框，取消“附加项”下面的“预编译头”和“安全开发生命周期（SDL）检查”的复选设置。如图4-16所示。取消复选框中的“勾选”后单击“完成”按钮，进入Visual C++2015集成开发环境，如图4-17所示

# 4.10 编程实作：继承编程应用

(3) 在“解决方案资源管理器”中，右击“头文件”|“添加”|“新建项”，然后在弹出的对话框中选中“头文件”，并在名称中输入ComFinal.h。按照同样的方法将Account.h，Chemistry.h添加到项目中。

(4) 按照与添加头文件相同的方法，将各类的源文件ComFinal.cpp，Account.cpp，Chemistry.cpp添加到项目中。项目的结构如图4-17所示，只不过到目前为止，这些头文件和源文件都是空文件。



## 4.10 编程实作：继承编程应用

### 2. 编写各类头文件和源文件的程序代码

(1) 建立ConFinal类。

① 在comFinal.h头文件中输入如下内容：

```
//comFinal.h
```

```
#ifndef comFinal_h
```

```
#define comFinal_h
```

```
class comFinal {
```

```
protected:
```

```
    char name[20];
```

```
    int english, chinese, math;
```

```
//学生姓名
```

```
//公共课成绩及总分
```

## 4.10 编程实作： 继承编程应用

public:

```
comFinal(char *n, int Eng, int Chi, int Mat);  
comFinal() {};  
char *getName() { return name; }  
int getEng() { return english; }  
int getChi() { return chinese; }  
int getMat() { return math; }  
void setEng(int x) { english = x; }  
void setChi(int x) { chinese = x; }  
void setMat(int x) { math = x; }  
int getTotal() { return english + chinese + math; }  
double getAverage() { return (english + chinese + math) / 3; }  
void show();    //显示学生各公共课的成绩、平均分和总分  
};  
#endif
```



## ② 在ComFinal.cpp源文件中输入如下内容:

```
//ComFinal.cpp
#include <iostream>
#include "comFinal.h"
using namespace std;
comFinal::comFinal(char *n, int Eng, int Chi, int Mat) {
    english = Eng; chinese = Chi; math = Mat;
    strcpy(name, n);
}
void comFinal::show() {
    cout << "学生姓名 :" << getName() << endl;
    cout << "英语成绩: " << getEng() << endl;
    cout << "语文成绩: " << getChi() << endl;
    cout << "数学成绩: " << getMat() << endl;
    cout << "基础课总分: " << getTotal() << endl;
    cout << "基础课平均成绩: " << getAverage() << endl << endl;
}
```

## (2) Account类的建立

---

① 在Account.h头文件中输入如下内容:

```
//Account.h
#include "comFinal.h"
#ifndef Account_h
#define Account_h
class Account :public comFinal {
protected:
    int account;           //会计学成绩
    int econ;              //经济学成绩
```

## 4.10 编程实作： 继承编程应用

---

public:

```
Account(char *n, int Eng, int Chi, int Mat, int Acc, int Eco);
```

```
Account() {};
```

```
int getMajtotal() { return econ + account; }
```

```
float getMajave() { return float((account + econ) / 2); }
```

```
int getAccount() { return account; };
```

```
int getEcon() { return account; }
```

```
void setAccount(int x) { account = x; }
```

```
void setEcon(int x) { econ = x; }
```

```
void show();
```

```
};
```

```
#endif
```

## ② 在Account.cpp源文件输入如下内容:

```
//Account.cpp
```

---

```
#include "account.h"
```

```
#include<iostream>
```

```
using namespace std;
```

```
Account::Account(char *n, int Eng, int Chi, int Mat, int Acc,  
    int Eco) :comFinal(n, Eng, Chi, Mat) {  
    econ = Eco; account = Acc;  
}
```

```
void Account::show() {  
    comFinal::show();  
    cout << "会计学成绩: " << account << endl;  
    cout << "经济学成绩: " << econ << endl;  
    cout << "总分 " << getTotal() + account + econ << endl;  
}
```

# (3) 建立Chemistry类

---

① 在Chemistry.h头文件中输入如下内容:

```
//Chemistry.h
#include "comFinal.h"
#ifndef chemistry_h
#define chemistry_h
class Chemistry :public comFinal {
protected:
    int chemistr;
    int analy;
```

//化学成绩

//化学分析成绩

## (3) 建立Chemistry类

---

public:

```
Chemistry(char *n, int Eng, int Chi, int Mat, int Chem, int Anal);  
Chemistry() {};  
int getMajtotal() { return analy + chemistr; }  
float getMajave() { return float((chemistr + analy) / 2); }  
int getChe() { return chemistr; };  
int getAnl() { return analy; }  
void setChe(int x) { chemistr = x; }  
void setAnl(int x) { analy = x; }  
void show();  
};  
#endif
```

## ② 在Chemistry.cpp源文件中输入如下内容：

```
//#include "stdafx.h"
//Chemistry.cpp
#include<iostream>
#include"Chemistry.h"
using namespace std;
Chemistry::Chemistry(char *n, int Eng, int Chi, int Mat,
                    int Chem, int Anal) :comFinal(n, Eng, Chi, Mat)
{
    chemistr = Chem; analy = Anal;
}

void Chemistry::show() {
    comFinal::show();
    cout << "有机化学: " << chemistr << endl;
    cout << "化学分析: " << analy << endl;
    cout << "总分 " << getTotal() + chemistr + analy << endl;
}
```

## (4) 建立主程序并运行程序

---

在com\_main.cpp中，输入下面的程序代码：

```
//com_main.cpp
#include "Chemistry.h"
#include "Account.h"
#include <iostream>
using namespace std;
void main() {
    Account a1("张三星", 98, 78, 97, 67, 87);
    Chemistry c1("光红顺", 89, 76, 34, 56, 78);
    a1.show();
    cout << "-----" <<endl;
    c1.setAnl(100);
    c1.show();
}
```



# 编译并运行该程序，输出结果如下：

---

学生姓名 :张三星  
英语成绩: **98**  
语文成绩: **78**  
数学成绩: **97**  
基础课总分: **273**  
基础课平均成绩: **91**

会计学成绩: **67**  
经济学成绩: **87**  
总分 **427**

-----  
学生姓名 :光红顺  
英语成绩: **89**  
语文成绩: **76**  
数学成绩: **34**  
基础课总分: **199**  
基础课平均成绩: **66**

有机化学: **56**  
化学分析: **100**  
总分 **355**