



## § 17. 继承和派生

### 1. 基本概念

重用：事物不加修改或稍作修改即可重复使用

软件代码的可重用性：软件代码可重复利用

★ 一些经典算法及经典结构可以重用

★ 用户界面部分一般超过70%的代码可重用

继承：C++中实现重用的机制

类的继承：在一个已有类的基础上建立一个新类，新类可以从已有类那里获得已有特征，这种现象称为类的继承

类的派生：对继承而言，可以理解为已有类派生出新类

教务处：

```
class student {
```

```
    private:
```

```
        int  sno;          //学号
        char sname[20];    //姓名
        char ssex;         //性别
        int  sage;         //年龄
        char sdept[16];    //系部
        char saddr[64];    //地址
```

```
    public:
```

```
        各种函数
```

```
};
```

学生处：

```
class student_2 {
```

```
    private:
```

```
        int  sno;
        char sname[20];
        char ssex;
        int  sage;
        char sdept[16];
        char saddr[64];
```

```
        char ssc; //奖惩情况
```

```
    public:
```

```
        各种函数
```

```
};
```

目前是两个独立的类  
但有很多成员相同  
能否在student的基础上  
定义student\_2  
这样可以少做很多工作



## § 17. 继承和派生

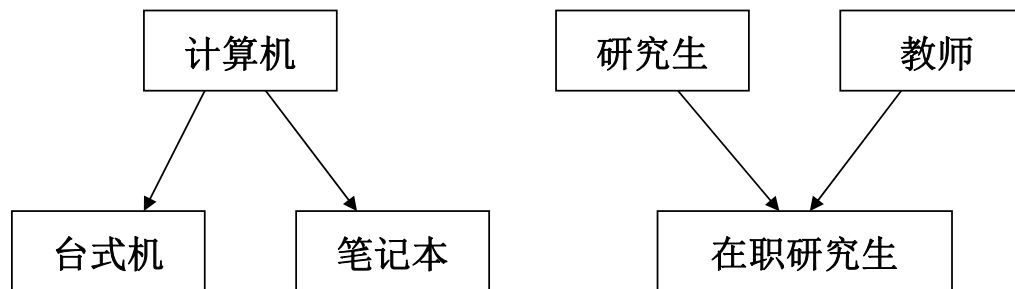
### 1. 基本概念

基类与派生类：基类(父类)，指已有的类

派生类(子类)，新建立的类

★ 派生类是基类的细化，基类是派生类的抽象

单继承与多继承：一个派生类从一个基类派生，则称为单继承，否则称为多继承





## § 17. 继承和派生

### 2. 派生类的声明方式

形式:

```
class 派生类名:private/public 基类名1, private/public 基类名2, ..., private/public 基类名n {
```

```
    private:
```

```
        私有成员;
```

```
    public:
```

```
        公有成员;
```

```
};
```

当只有一个基类名时，单继承  
当多于一个基类名时，多继承

★ 基类名前的private/public称为**基类存取限定符**根据限定符的不同分别称为私有继承和公有继承

★ 若基类前不加限定符，缺省是private

★ 根据需要加入派生类自己特有的成员

```
class 派生类名 : 基类名 {
    private:
        私有成员;
    public:
        公有成员;
};
```



## § 17. 继承和派生

### 2. 派生类的声明方式

- ★ 基类名前的private/public称为**基类存取限定符**根据限定符的不同分别称为私有继承和公有继承
- ★ 若基类前不加以限定符，缺省是private
- ★ 根据需要加入派生类自己特有的成员

教务处:

```
class student {  
    private:  
        int sno;           //学号  
        ...  
        char saddr[64];    //地址  
    public:  
        各种函数  
};
```

基类的成员表示  
为私有不太妥当，  
先这样表示

学生处:

```
class student_2:public student {  
    private:  
        char ssc; //奖惩情况  
    public:  
        ...  
};
```

student类:

- ① 可定义对象/指针/数组/引用
- ② 可做函数参数、返回值
- ③ 可通过student类的对象访问student类的  
数据成员及成员函数(按访问规则)

**特别强调:** student作为一个类，可以正常  
使用，不要误解为被student\_2  
继承后，只能用student\_2

student\_2类

- ① 可定义对象/指针/数组/引用
- ② 可做函数参数、返回值
- ③ 可通过student\_2类的对象访问student\_2  
类的数据成员及成员函数(按访问规则)
- ④ 可通过student\_2类的对象访问student类  
的数据成员及成员函数(具体待讨论)



## § 17. 继承和派生

### 2. 派生类的声明方式

形式:

```
class 派生类名:private/public 基类名1, private/public 基类名2, ..., private/public 基类名n {  
    private:  
        私有成员;  
    public:  
        公有成员;  
};
```

★ 对于多继承，派生类对每个基类的继承方式可不同



```
class A {  
    ...  
};  
class B {  
    ...  
};  
class C:public A, private B {  
    ...  
};
```

★ 暂不讨论基类的数据成员与成员函数与派生类同名的问题



## § 17. 继承和派生

### 3. 派生类的构成

派生类对象所占的空间:

基类数据成员所占空间总和 + 派生类数据成员所占空间的总和

派生类可访问的成员函数:

基类成员函数 + 派生类成员函数

<pre>class A {     priavte:         int  a;         short b;         char  c;     public:         void f1();         int  f2(); };</pre>	<pre>class B:public A {     priavte:         int  d;         short e;         char  f;     public:         void f3();         int  f4(); };</pre>	<div>B b1;</div>   
--	---	--

★ 继承基类的全部数据成员 (不一定都可以访问)

★ 继承基类除构造函数和析构函数外的全部成员函数 (不一定都可以访问)

★ 友元不能继承

★ 派生类的数据成员/成员函数允许和基类的同名, 不同的继承方式访问方法不同 (暂不讨论)



## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4.1. 派生类和基类对成员的访问

普通类的访问规则：

类的成员函数访问类的成员 : 全部

作用域外访问类的成员 : public

派生类与基类的访问规则：

基类的成员函数访问基类的成员 : 全部

基类的成员函数访问派生类的成员 : 不允许

作用域外通过基类访问基类的成员 : public

作用域外通过基类访问派生类的成员 : 不允许

派生类的成员函数访问基类的成员 : 分析讨论

派生类的成员函数访问派生类的成员 : 全部

作用域外通过派生类访问基类的成员 : 分析讨论

作用域外通过派生类访问派生类的成员 : public



## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4.1. 派生类和基类对成员的访问

派生类与基类的访问规则：

派生类的成员函数访问基类的成员      : 分析讨论

作用域外通过派生类访问基类的成员   : 分析讨论

```
class A {  
    private:  
        int a;  
        void f1();  
    public:  
        int b;  
        void f2();  
};  
  
class B:private/public A {  
    private:  
        int c;  
        void f3();  
    public:  
        int d;  
        void f4();  
};
```

基类A的成员函数  
没有讨论价值

void A::f2()

a=10;	✓
f1();	✓
b=15;	✓
c=10;	✗
d=15;	✗
f3();	✗
f4();	✗

派生类B的成员函数  
部分需要讨论

void B::f4()

a=10;	讨论
f1();	讨论
b=15;	讨论
f2();	讨论
c=10;	✓
d=15;	✓
f3();	✓

基类的作用域外，  
没有讨论价值  
派生类的作用域外，  
部分需要讨论

int main()

a1.a=10;	✗
a1.f1();	✗
a1.b=10;	✓
a1.f2();	✓

b1.a=10;	讨论
b1.f1();	讨论
b1.b=10;	讨论
b1.f2();	讨论
b1.c=10;	✗
b1.f3();	✗
b1.d=10;	✓
b1.f4();	✓





## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4.2. 公有派生

形式: `class 派生类名 : public 基类名 {`  
    `private:`  
        `...`  
    `public:`  
        `...`  
};

★ 基类的`private`成员被继承, 但不可访问

★ 基类的`public`成员被继承为派生类的`public`

#### 4.3. 私有派生

形式: `class 派生类名 : private 基类名 {`  
    `private:`  
        `...`  
    `public:`  
        `...`  
};

★ 基类的`private`成员被继承, 但不可访问

★ 基类的`public`成员被继承为派生类的`private`

不可访问: 不能被直接访问, 但仍然  
可以通过公有函数等形式  
进行间接访问  
(下同)



## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4.1. 派生类和基类对成员的访问

派生类与基类的访问规则：

派生类的成员函数访问基类的成员 : 分析讨论

作用域外通过派生类访问基类的成员 : 分析讨论

```
class A {  
    private:  
        int a;  
        void f1();  
    public:  
        int b;  
        void f2();  
};  
  
class B:private/public A {  
    private:  
        int c;  
        void f3();  
    public:  
        int d;  
        void f4();  
};
```

#### B公有继承A

```
void B::f4()    int main()  
{  
    a=10; 讨论 ×    A a1;  
    f1(); 讨论 ×    B b1;  
    b=15; 讨论 ✓    a1.a=10; ×  
    f2(); 讨论 ✓    a1.f1(); ×  
    c=10; ✓          a1.b=10; ✓  
    f3(); ✓          a1.f2(); ✓  
    d=15; ✓          b1.a=10; 讨论 ×  
                b1.f1(); 讨论 ×  
                b1.b=10; 讨论 ✓  
                b1.f2(); 讨论 ✓  
                b1.c=10; ×  
                b1.f3(); ×  
                b1.d=10; ✓  
                b1.f4(); ✓  
}
```

#### B私有继承A

```
void B::f4()    int main()  
{  
    a=10; 讨论 ×    A a1;  
    f1(); 讨论 ×    B b1;  
    b=15; 讨论 ✓    a1.a=10; ×  
    f2(); 讨论 ✓    a1.f1(); ×  
    c=10; ✓          a1.b=10; ✓  
    f3(); ✓          a1.f2(); ✓  
    d=15; ✓          b1.a=10; 讨论 ×  
                b1.f1(); 讨论 ×  
                b1.b=10; 讨论 ×  
                b1.f2(); 讨论 ×  
                b1.c=10; ×  
                b1.f3(); ×  
                b1.d=10; ✓  
                b1.f4(); ✓  
}
```



## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4.4. 派生类的多级派生

```
class A {  
    private:  
        int a;  
        void f1();  
    public:  
        int b;  
        void f2();  
};  
  
class B:public A {  
    private:  
        int c;  
        void f3();  
    public:  
        int d;  
        void f4();  
};  
  
class C:public B {  
    private:  
        int e;  
        void f5();  
    public:  
        int f;  
        void f6();  
};
```

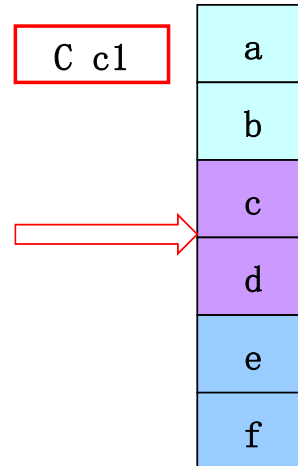
#### ★ 基类分为直接基类和间接基类

C的直接基类是B，间接基类是A

B的直接基类是A

#### ★ 派生类包含所有基类的数据成员 (不一定可访问)

#### ★ 基类的成员的被访问关系逐级确定





## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4. 4. 派生类的多级派生

#### ★ 基类的成员的被访问关系逐级确定

```
class A {
    private:
        int a;
        void f1();
    public:
        int b;
        void f2();
};

class B : 继承方式 A {
    private:
        int c;
        void f3();
    public:
        int d;
        void f4();
};

class C : 继承方式 B {
    private:
        int e;
        void f5();
    public:
        int f;
        void f6();
};
```

B公有继承A, C公有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ✕	c1.a=10; ✕
f1(); ✕	c1.f1(); ✕
b=10;	c1.b=10;
f2();	c1.f2();
c=10; ✕	c1.c=10; ✕
f3(); ✕	c1.f3(); ✕
d=10;	c1.d=10;
f4();	c1.f4();
e=10;	c1.e=10; ✕
f5();	c1.f5(); ✕
f=10;	c1.f=10;
f6();	c1.f6();
}	}

B私有继承A, C公有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ✕	c1.a=10; ✕
f1(); ✕	c1.f1(); ✕
b=10; ✕	c1.b=10; ✕
f2(); ✕	c1.f2(); ✕
c=10; ✕	c1.c=10; ✕
f3(); ✕	c1.f3(); ✕
d=10;	c1.d=10;
f4();	c1.f4();
e=10;	c1.e=10; ✕
f5();	c1.f5(); ✕
f=10;	c1.f=10;
f6();	c1.f6();
}	}

B公有继承A, C私有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ✕	c1.a=10; ✕
f1(); ✕	c1.f1(); ✕
b=10;	c1.b=10; ✕
f2();	c1.f2(); ✕
c=10; ✕	c1.c=10; ✕
f3(); ✕	c1.f3(); ✕
d=10;	c1.d=10; ✕
f4();	c1.f4(); ✕
e=10;	c1.e=10; ✕
f5();	c1.f5(); ✕
f=10;	c1.f=10;
f6();	c1.f6();
}	}

B私有继承A, C私有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ✕	c1.a=10; ✕
f1(); ✕	c1.f1(); ✕
b=10; ✕	c1.b=10; ✕
f2(); ✕	c1.f2(); ✕
c=10; ✕	c1.c=10; ✕
f3(); ✕	c1.f3(); ✕
d=10;	c1.d=10; ✕
f4();	c1.f4(); ✕
e=10;	c1.e=10; ✕
f5();	c1.f5(); ✕
f=10;	c1.f=10;
f6();	c1.f6();
}	}



## § 17. 继承和派生

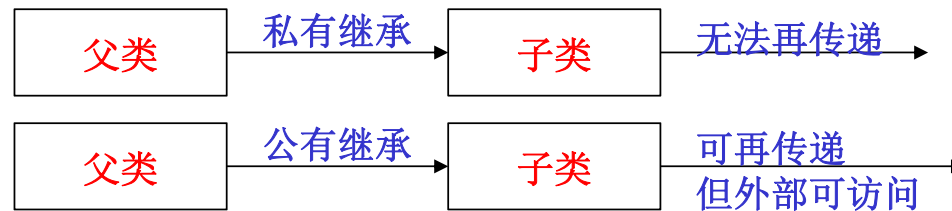
### 4. 派生类的成员访问属性

#### 4.5. 保护段与保护继承

引入：在多级继承中，基类的private不可访问，public部分被私有继承则不可再传递，若公有继承则对于整个继承序列的外部均可访问，为了加强灵活性，引入保护段及保护继承

保护段的定义：

```
class 类名 {  
    private:  
    ...  
    protected:  
    ...  
    public:  
    ...  
};
```



问题：(1) 父类的成员在派生类的继承序列中内部能够访问  
(2) ..... 外部不能.....

保护段的使用：

- ★ protected段的成员对外不可访问，对内可被任意访问，其成员函数可任意访问类的其它成员 (等同于private属性)
- ★ protected段的成员继承后可被派生类访问 (protected被private继承后当做派生类的private，protected被public继承后当做派生类的protected)
- ★ 若该类不被其它类所继承，则声明保护段无意义 (不被继承的情况下与声明为private等价)



## § 17. 继承和派生

### 4. 派生类的成员访问属性

#### 4.5. 保护段与保护继承

保护继承的定义：

```
class 派生类名 : protected 基类名 {  
    private:  
        ...  
    protected:  
        ...  
    public:  
        ...  
};
```

保护继承的使用：

- ★ 基类的private成员被继承，但不可访问
- ★ 基类的protected/public成员被继承为派生类的protected
- ★ 若该类不被其它类所再次传递继承，则声明保护继承无意义(不传递继承的情况下与private继承等价)



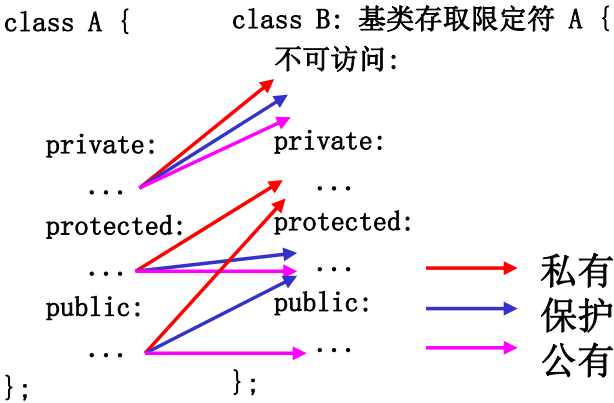
# § 17. 继承和派生

4. 派生类的成员访问属性

4. 6. 继承方式与访问属性

派生类的最终定义形式:

```
class 派生类名: private/protected/public 基类名 {
    private:
        ...
    protected:
        ...
    public:
        ...
};
```



继承与访问关系表:

继承 基类 \	private	protected	public
private	继承, 但不可访问	继承, 但不可访问	继承, 但不可访问
protected	private	protected	protected
public	private	protected	public

继承与派生类成员函数/派生类作用域外的访问关系表:

继承 基类 \	private		protected		public	
	成员	域外	成员	域外	成员	域外
private	✗	✗	✗	✗	✗	✗
protected	✓	✗	✓	✗	✓	✗
public	✓	✗	✓	✗	✓	✓



## § 17. 继承和派生

### 4. 派生类的成员访问属性

### 4.6. 继承方式与访问属性

#### 继承方式:private

B::fun()	int main()
{	{ B b1;
a=10; ✗	b1.a=10; ✗
f1(); ✗	b1.f1(); ✗
b=10;	b1.b=10; ✗
f2();	b1.f2(); ✗
c=10;	b1.c=10; ✗
f3();	b1.f3(); ✗
d=10;	b1.d=10; ✗
f4();	b1.f4(); ✗
e=10;	b1.e=10; ✗
f5();	b1.f5(); ✗
f=10;	b1.f=10;
f6();	b1.f6();
}	}

#### 继承方式:protected

B::fun()	int main()
{	{ B b1;
a=10; ✗	b1.a=10; ✗
f1(); ✗	b1.f1(); ✗
b=10;	b1.b=10; ✗
f2();	b1.f2(); ✗
c=10;	b1.c=10; ✗
f3();	b1.f3(); ✗
d=10;	b1.d=10; ✗
f4();	b1.f4(); ✗
e=10;	b1.e=10; ✗
f5();	b1.f5(); ✗
f=10;	b1.f=10;
f6();	b1.f6();
}	}

#### 继承方式:public

B::fun()	int main()
{	{ B b1;
a=10; ✗	b1.a=10; ✗
f1(); ✗	b1.f1(); ✗
b=10;	b1.b=10; ✗
f2();	b1.f2(); ✗
c=10;	b1.c=10;
f3();	b1.f3();
d=10;	b1.d=10; ✗
f4();	b1.f4(); ✗
e=10;	b1.e=10; ✗
f5();	b1.f5(); ✗
f=10;	b1.f=10;
f6();	b1.f6();
}	}

```
class A {
private:
    int a;
    void f1();
protected:
    int b;
    void f2();
public:
    int c;
    void f3();
};
```

```
class B:继承方式 A{
private:
    int d;
    void f4();
protected:
    int e;
    void f5();
public:
    int f;
    void f6();
    void fun();
};
```





## § 17. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

继承传递与成员函数/派生类作用域外的访问关系表:

(假设基本类A, 类B继承A, 类C继承B) (共9种情况)

列解释: B私 => B私有继承A  
成员=>在C的成员函数内访问  
域外=>在C的作用域外访问  
行解释: A:私=>A的私有成员在各种  
方式下的可访问性

继承 基类	B私 C私		B私 C保		B私 C公		B保 C私		B保 C保		B保 C公		B公 C私		B公 C保		B公 C公	
	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外
A: 私																		
A: 保																		
A: 公																		
B: 私																		
B: 保																		
B: 公																		
C: 私																		
C: 保																		
C: 公																		

请完成此表, 填×或✓, 表示不可访问/可访问



## § 17. 继承和派生

### 4. 派生类的成员访问属性

### 4.6. 继承方式与访问属性

继承传递与成员函数/派生类作用域外的访问关系表:

(假设基本类A, 类B继承A, 类C继承B) (共9种情况)

上页表格九种情况之一的实例表示, 其它请自行理解  
B私有继承A, C私有继承B

<pre>void C::fun() {   A私 a1=10; × fa1(); ×   A保 a2=10; × fa2(); ×   A公 a3=10; × fa3(); ×    B私 b1=10; × fb1(); ×   B保 b2=10;  fb2();   B公 b3=10;  fb3();    C私 c1=10;  fc1();   C保 c2=10;  fc2();   C公 c3=10;  fc3(); }</pre>	<pre>int main() {   C c;   A私 c.a1=10; × c.fa1(); ×   A保 c.a2=10; × c.fa2(); ×   A公 c.a3=10; × c.fa3(); ×    B私 c.b1=10; × c.fb1(); ×   B保 c.b2=10; × c.fb2(); ×   B公 c.b3=10; × c.fb3(); ×    C私 c.c1=10; × c.fc1(); ×   C保 c.c2=10; × c.fc2(); ×   C公 c.c3=10;  c.fc3(); }</pre>
--	--

```
class A {
    priavte:
        int a1;
        void fa1();
    protected:
        int a2;
        void fa2();
    public:
        int a3;
        void fa3();
};

class B : private A {
    priavte:
        int b1;
        void fb1();
    protected:
        int b2;
        void fb2();
    public:
        int b3;
        void fb3();
};

class C : public B {
    priavte:
        int c1;
        void fc1();
    protected:
        int c2;
        void fc2();
    public:
        int c3;
        void fc3();
        void fun();
};
```



## § 17. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

### ★ 多级派生总结

- 如果在多级派生时都采用**私有**继承方式，则传递一次后基类的所有成员**均不可访问**
- 如果在多级派生时都采用**保护**继承方式，则传递一次后基类的所有成员**均只能在类的作用域内访问/不可访问**
- 如果在多级派生时都采用**公有**继承方式，则传递多次后基类的**公有及保护成员仍能访问且访问属性不变**

=> **实际使用**中：成员属性：protected / public  
继承方式：public



## § 17. 继承和派生

4. 派生类的成员访问属性

4.7. 派生类与基类的成员同名

数据成员同名：

★ 派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

成员函数同名：

参数的个数、类型完全相同

★ 与数据成员同名的处理方式相同

参数的个数、类型不同

★ 与数据成员同名的处理方式相同

三种情况规则一样

派生类优先于基类，称为支配规则

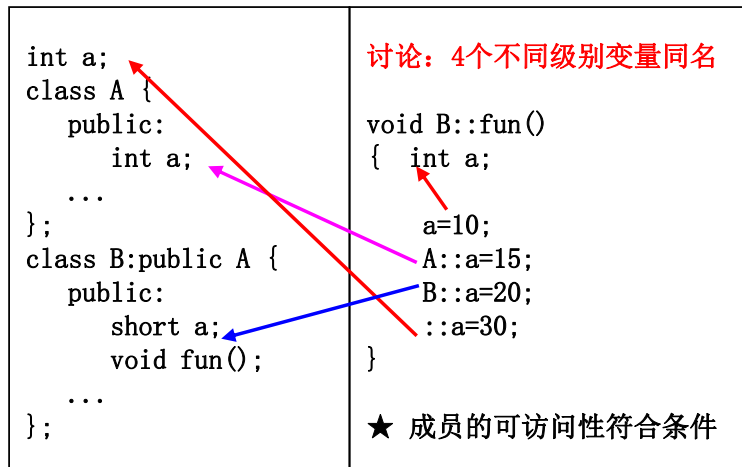
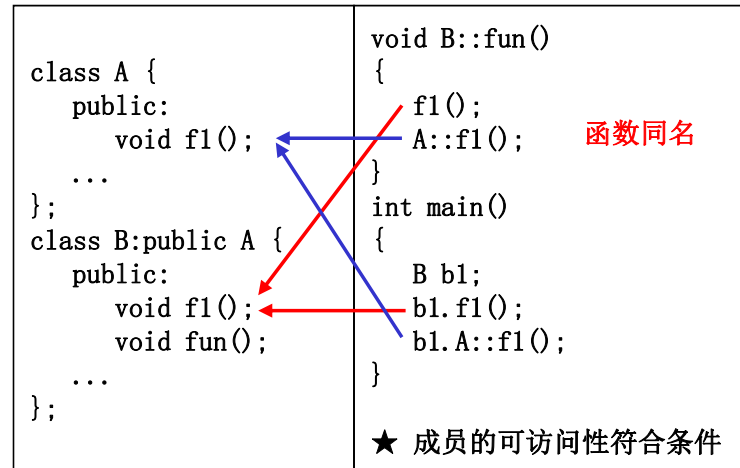
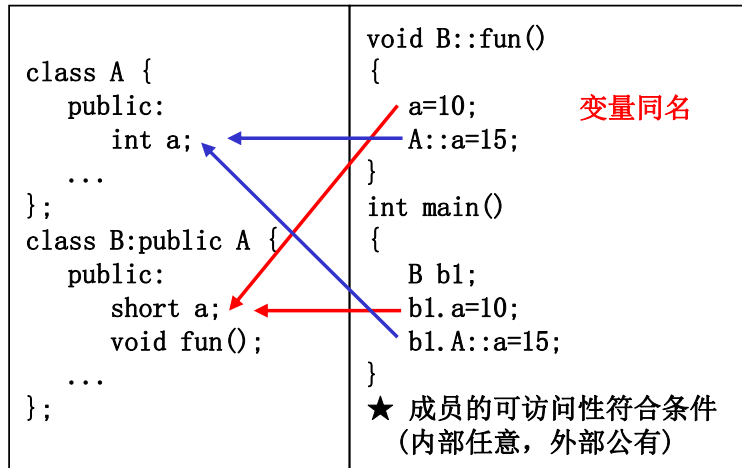


## § 17. 继承和派生

### 4. 派生类的成员访问属性

### 4.7. 派生类与基类的成员同名

★ 派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员





## § 17. 继承和派生

### 4. 派生类的成员访问属性

### 4.7. 派生类与基类的成员同名

★ 派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

函数同名，非重载，适用支配规则

<pre>class A { public:     void f1(int x);     ... }; class B:public A { public:     void f1();     void fun();     ... };</pre>	<pre>void B::fun() {     f1();     f1(10); //编译错误 }  int main() {     B b1;     b1.f1();     b1.f1(10); //编译错误 }</pre>
★ 成员的可访问性符合条件	

函数同名，非重载，适用支配规则

<pre>class A { public:     void f1(int x);     ... }; class B:public A { public:     void f1();     void fun();     ... };</pre>	<pre>void B::fun() {     f1();     A::f1(10); }  int main() {     B b1;     b1.f1();     b1.A::f1(10); }</pre>
★ 成员的可访问性符合条件	



## § 17. 继承和派生

4. 派生类的成员访问属性

4.8. 基类对象不可访问部分的强制访问

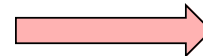
问题引入:

- 任何继承方式派生类均不能访问基类的private
- 派生类的对象中包含了基类对象(包括private部分)

=> 不可访问, 又占空间?

解决方法:

继承后, 基类的私有部分不可被派生类直接访问, 但是可以通过基类的  
**可访问的成员函数**进行间接访问



- 1、那句话编译错?
- 2、删除错误语句后的运行结果

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

class A {
private:
    int a;
public:
    void set(int x) {
        a=x;
    }
    void show() {
        cout << a << endl;
    };
};

class B:public A {
public:
    void fun() {
        a=10;
        set(10);
    }
};

int main()
{
    B b1;
    b1.set(15);
    b1.show();
    b1.fun();
    b1.show();
}
```



## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.1. 简单的派生类构造函数

含义：在派生类的数据成员中不包含基类的对象

构造函数的实现方法：

★ 构造函数不能继承，同时派生中又含有基类的成员，需要能同时初始化基类和派生类的数据成员，因此在派生类的构造函数中激活基类的构造函数

形式：

派生类名(参数):基类名(参数)

初始化表方式

{

函数体(一般是对派生类新增成员的初始化)

}

构造函数的调用顺序：

★ 先基类，再派生类

(在派生类的构造函数中先自动激活基类的构造函数)





## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.1. 简单的派生类构造函数

```
class Time {
protected:
    int hour, minute, second;
public:
    Time(int h, int m, int s) {
        hour = h; minute = m; second = s;
    }
};
class Date:public Time {
protected:
    int year, month, day;
public:
    Date(int y, int mo, int d, int h, int m, int s):Time(h,m,s) {
        year = y;
        month = mo;
        day = d;;
    }
};
```

只有参数名，没有类型，  
从派生类的形参表中取

体内实现

```
class Date:public Time {
...
public:
    Date(int y, int mo, int d, int h, int m, int s);
};
Date::Date(int y, int mo, int d, int h, int m, int s):Time(h,m,s)
{
    year = y;
    month = mo;
    day = d;;
}
```

对体外实现，声明时不要，  
实现时给出即可

体外实现

```
int main()
{
    Date d1 (2021 5, 14, 15, 16, 17);
    ...
}
```



## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.2. 含有子对象的派生类构造函数

含义：在派生类定义中包含了基类的对象

形式：

```
派生类名(参数):基类名(参数),子对象名(参数)
{
    函数体(一般是对派生类新增成员的初始化)
}
```

构造函数的调用顺序：

★ 先基类，次子对象，再派生类

★ 不再深入讨论



## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.2. 含有子对象的派生类构造函数

```
class student {  
    protected:  
        int num;  
        char name[10];  
    public:  
        student(int n, char *nam)  
        {  
            num = n;  
            strcpy(name, nam);  
        }  
    ...  
};
```

```
class stu:public student {  
    protected:  
        int age;  
        char addr[30];  
        student monitor;  
    public:  
        stu(int n, char *nam, int n1, char *nam1, int a, char *ad): student(n, nam), monitor(n1, nam1)  
        {  
            age = a;  
            strcpy(addr, ad);  
        }  
};
```

```
int main()  
{  
    stu s1(10001, "张三", 10009, "李四", 20, "上海");  
    ...  
}
```

班长

对于stu类的对象

stu s1;

所占空间为16+36+16=68字节，即双份student，其中1份继承，1份是monitor

也可体外实现(略)



## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.3. 多层派生时的构造函数

形式:

派生类名(参数):直接基类名(参数)

{

函数体(一般是对派生类新增成员的初始化)

}

★ 间接基类不需要出现在初始化表中

构造函数的调用顺序:

★ 按继承顺序依次调用, 派生类放在最后

(派生类中先激活直接基类, 直接基类再先激活其直接基类)



## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.3. 多层派生时的构造函数

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

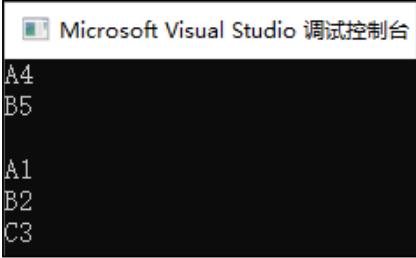
class A {
public:
    A(int x) { cout << "A" << x << endl; }
};

class B : public A {
public:
    B(int x, int y):A(x) { cout << "B" << y << endl; }
};

class C : public B {
public:
    C(int x, int y, int z):B(x, y) { cout << "C" << z << endl;}
};

int main()
{
    B b1(4,5);
    cout << endl;
    C c1(1,2,3);
}
```

A4  
B5  
A1  
B2  
C3





## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.4. 派生类构造函数的特殊形式

★ 若派生类的初始化表中不出现基类，则系统自动调用基类的无参构造函数

(允许基类不定义构造函数，此时调用系统缺省的无参空体构造函数)

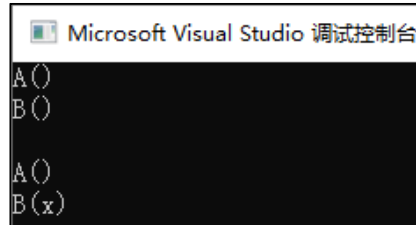
```
#include <iostream>
using namespace std;
```

调用A的无参构造函数

```
class A {
public:
    A()      { cout << "A()" << endl; }
    A(int x) { cout << "A(x)" << endl; }
};
class B:public A {
public:
    B()      { cout << "B()" << endl; }
    B(int x) { cout << "B(x)" << endl; }
};
```

```
int main()
{
    B b1;
    cout << endl;
    B b2(100);
}
```

A()  
B()  
  
A()  
B(x)



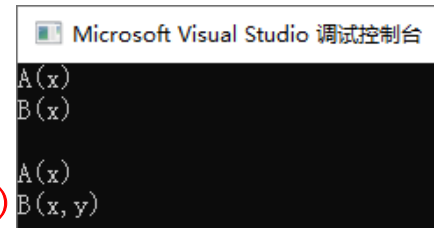
```
#include <iostream>
using namespace std;
```

调用A的有参构造函数

```
class A {
public:
    A()      { cout << "A()" << endl; }
    A(int x) { cout << "A(x)" << endl; }
};
class B:public A {
public:
    B(int x):A(x)      { cout << "B(x)" << endl; }
    B(int x, int y):A(x) { cout << "B(x,y)" << endl; };
};
```

```
int main()
{
    B b1(10);
    cout << endl;
    B b2(100, 200);
}
```

A(x)  
B(x)  
  
A(x)  
B(x, y)





## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.4. 派生类构造函数的特殊形式

★ 若派生类的初始化表中不出现基类，则系统自动调用基类的无参构造函数

(允许基类不定义构造函数，此时调用系统缺省的无参空体构造函数)

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

class A {
public:
    A() { cout << "A()" << endl; }
    A(int x) { cout << "A(x)" << endl; }
};

class B:public A {
public:
    B() { cout << "B()" << endl; }
    B(int x) { cout << "B(x1)" << endl; }
    B(int x):A(x) { cout << "B(x2)" << endl; }
    B(int x, int y):A(x) { cout << "B(x,y)" << endl; }
};

int main()
{
    B b1;
    cout << endl;
    B b2(100);
    cout << endl;
    B b3(4, 5);
}
```

编译时会报函数重载错误

注释掉两个函数中的第1个，给出程序的运行结果

注释掉两个函数中的第2个，给出程序的运行结果

编译时这两句报函数重载错误，为什么？



## § 17. 继承和派生

### 5. 派生类的构造函数和析构函数

#### 5.5. 派生类的析构函数

析构函数的调用：

在派生类对象出作用域时，**自动调用**派生类的析构函数，在其中再**自动调用**基类的析构函数

析构函数的调用顺序：

★ 先派生类，再基类**(与构造函数的顺序相反)**

析构函数的使用：

★ 派生类的数据成员无动态内存申请的情况下一般不需要定义派生类的析构函数

★ 若基类有动态内存申请，派生类无，则基类需要定义，派生类不需要





## § 17. 继承和派生

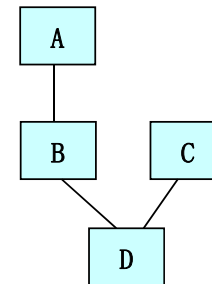
### 6. 多重继承

#### 6.1. 多重继承的声明及使用

```
class 派生类名: private/protected/public 基类名1, ... , private/protected/public 基类名n {  
    private:  
        私有成员;  
    protected:  
        保护成员;  
    public:  
        公有成员;  
};
```

- ★ 每个基类的基类存取限定符允许不同
- ★ 每个基类被继承后的可访问性由其基类存取限定符确定
- ★ 派生类继承所有基类的数据成员
- ★ 派生类继承所有基类除构造函数和析构函数外的所有成员函数
- ★ 派生类对象所占的空间 = 所有基类的数据成员之和 + 派生类的数据成员之和
- ★ 多重继承允许多层派生

```
class A {  
    private:  
        int a1;  
    protected:  
        int a2;  
    public:  
        int a3;  
};  
  
class B:public A {  
    private:  
        int b1;  
    protected:  
        int b2;  
    public:  
        int b3;  
};  
  
class C {  
    private:  
        int c1;  
    protected:  
        int c2;  
    public:  
        int c3;  
};  
  
class D:private B, public C {  
    ...  
};
```





## § 17. 继承和派生

### 6. 多重继承

#### 6.2. 多重继承的派生类构造函数和析构函数

构造函数的形式:

```
派生类名(参数):基类名1(参数),..., 基类名n(参数)
{
    函数体(一般是对派生类新增成员的初始化)
}
```

构造函数的使用:

- ★ 基类名出现的顺序无限制
- ★ 若调用基类的无参构造函数, 则该基类可不出现在初始化参数表中

```
class A {
public:
    A(int x) { ... }
};
class B {
public:
    B(int y) { ... }
};
class C:public A, private B {
public:
    C(int a, int b, int c):A(a), B(b) { ... }
};
```

B(b), A(a)

```
class A {
public:
    A() { ... }
};
class B {
public:
    B(int y) { ... }
};
class C:public A, private B {
public:
    C(int a, int b, int c):B(b) { ... }
};
```

激活A的无参构造



## § 17. 继承和派生

### 6. 多重继承

#### 6.2. 多重继承的派生类构造函数和析构函数构造函数的使用:

★ 若是多层派生, 只出现直接基类

```
class A {
public:
    A(int x) { ... }
};
class B {
public:
    B(int y) { ... }
};
class C:public B {
public:
    C(int z):B(z) { ... };
};
class D:public A, private C {
public:
    D(int a, int c, int d):A(a), C(c) { ... }
};
```

```
graph TD
    A[A] --> D[D]
    B[B] --> C[C]
    C[C] --> D[D]
```

B不需要出现

★ 若派生中含有基类的实例对象(子对象), 则子对象也可以出现在初始化参数表中

```
class A {
public:
    A(int x) { ... }
};
class B {
public:
    B(int y) { ... }
};
class C:public A, private B {
public:
    B bb;
    C(int a, int b, int b1):A(a), B(b), bb(b1) { ... }
};
```



## § 17. 继承和派生

### 6. 多重继承

#### 6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序：

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

与构造函数中基类出现的顺序无关

```
class D: ... {  
    public:  
        D():A(),B(),C() {...}  
           B(),A(),C()  
           C(),B(),A()  
};  
构造函数的调用顺序都是A、B、C、D
```

```
#include <iostream>  
using namespace std;  
class A {  
    public:  
        A() {cout << "A()" << endl;}  
};  
  
class B {  
    public:  
        B() {cout << "B()" << endl;}  
};  
  
class C {  
    public:  
        C() {cout << "C()" << endl;}  
};  
  
class D:public A, protected B, private C {  
    public:  
        D() {cout << "D()" << endl;}  
};  
  
int main()  
{  
    D d1;  
}
```

构造函数的调用顺序：

A()  
B()  
C()  
D()

两者一致

D对象的内存映像：

A的数据成员
B的数据成员
C的数据成员
D的数据成员



## § 17. 继承和派生

### 6. 多重继承

#### 6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序:

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

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

class A {
public:
    int a;
    A() { a=1; }
};

class B {
public:
    int b;
    B() { b=2; }
};

class C {
public:
    int c;
    C() { c=3; }
};
```

```
class D:public A, protected B, private C {
public:
    int d;
    D() { d=4; }
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;
    int *p = (int *)&d1;

    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;

    return 0;
}
```

证明D对象的内存映像与  
D对象的构造函数调用顺序  
一致的测试程序

D对象的内存映像:
A的数据成员
B的数据成员
C的数据成员
D的数据成员

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

class A {
public:
    A() {cout << "A()" << endl;}
};

class B {
public:
    B() {cout << "B()" << endl;}
};

class C {
public:
    C() {cout << "C()" << endl;}
};

class D:public A, protected B, private C {
public:
    D() {cout << "D()" << endl;}
};

int main()
{
    D d1;
}
```

构造函数的调用顺序:

A()  
B()  
C()  
D()

两者一致



## § 17. 继承和派生

### 6. 多重继承

#### 6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序:

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

F对象的内存映像:

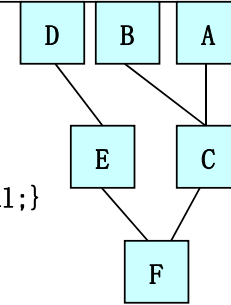
D的数据成员
E的数据成员
B的数据成员
A的数据成员
C的数据成员
F的数据成员

两者一致

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

class A {
public:
    A() {cout << "A()" << endl;}
};
class B {
public:
    B() {cout << "B()" << endl;}
};
class C:public B, public A {
public:
    C() {cout << "C()" << endl;}
};
class D {
public:
    D() {cout << "D()" << endl;}
};
class E:public D {
public:
    E() {cout << "E()" << endl;}
};
class F:public E, public C {
public:
    F() {cout << "F()" << endl;}
};

int main()
{
    F f1;
}
```



D()  
E()  
B()  
A()  
C()  
F()



## § 17. 继承和派生

### 6. 多重继承

#### 6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序：

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

析构函数的使用：

★ 若数据成员没有动态内存申请，一般不需要定义

析构函数的调用顺序：

★ 先派生类的析构函数，再按声明顺序的反序依次调用基类的析构函数



## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名:

按单继承的方式进行处理

(支配规则:派生类直接访问, 基类加作用域符)

两个以上的基类中的成员同名:

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名:

派生类直接访问, 不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名:

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则





## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名:

按单继承的方式进行处理

(支配规则:派生类直接访问, 基类加作用域符)

两个以上的基类中的成员同名:

分别加不同的基类作用域符区分

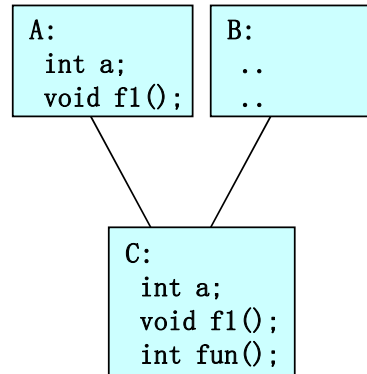
两个以上的基类与派生类的成员同名:

派生类直接访问, 不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名:

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则



```
int C::fun()
{
    a=10;
    f1();
    A::a=15;
    A::f1();
}
```

```
int main()
{
    C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
}
```

下面测试程序编译通过,  
则证明上面的例子是正确的

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

class A {
public:
    int a;
    void f1() { return; }
};

class B {
public:
    int b;
};

class C:public A, public B {
public:
    int a;
    void f1() { return; }
    void fun();
};
```

```
void C::fun()
{
    a = 10;
    f1();
    A::a = 15;
    A::f1();
}

int main()
{
    C c1;

    c1.a = 10;
    c1.f1();
    c1.A::a = 15;
    c1.A::f1();

    return 0;
}
```



## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名:

按单继承的方式进行处理

(支配规则:派生类直接访问, 基类加作用域符)

两个以上的基类中的成员同名:

分别加不同的基类作用域符区分

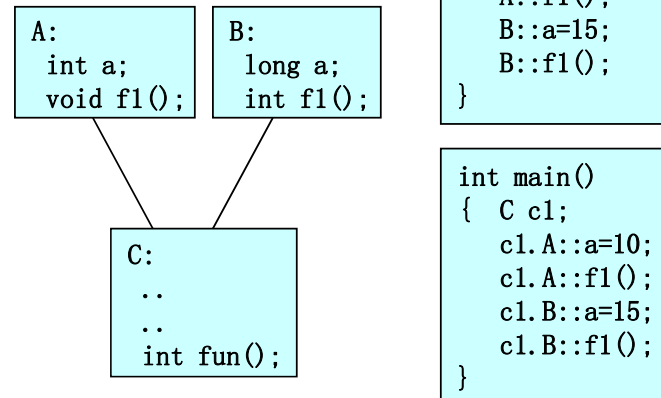
两个以上的基类与派生类的成员同名:

派生类直接访问, 不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名:

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则



可自行构造测试程序来验证



## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名:

按单继承的方式进行处理

(支配规则:派生类直接访问, 基类加作用域符)

两个以上的基类中的成员同名:

分别加不同的基类作用域符区分

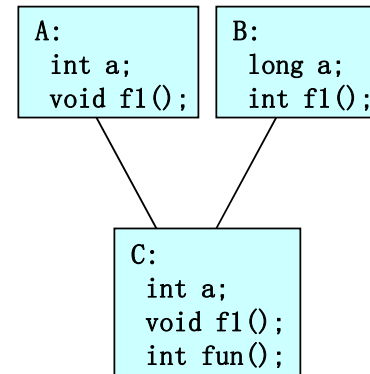
两个以上的基类与派生类的成员同名:

派生类直接访问, 不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名:

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则



```
int C::fun()
{
    a=10;
    f1();
    A::a=15;
    A::f1();
    B::a=20;
    B::f1();
}
```

```
int main()
{
    C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
    c1.B::a=20;
    c1.B::f1();
}
```

可自行构造测试程序来验证



## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名:

按单继承的方式进行处理

(支配规则:派生类直接访问, 基类加作用域符)

两个以上的基类中的成员同名:

分别加不同的基类作用域符区分

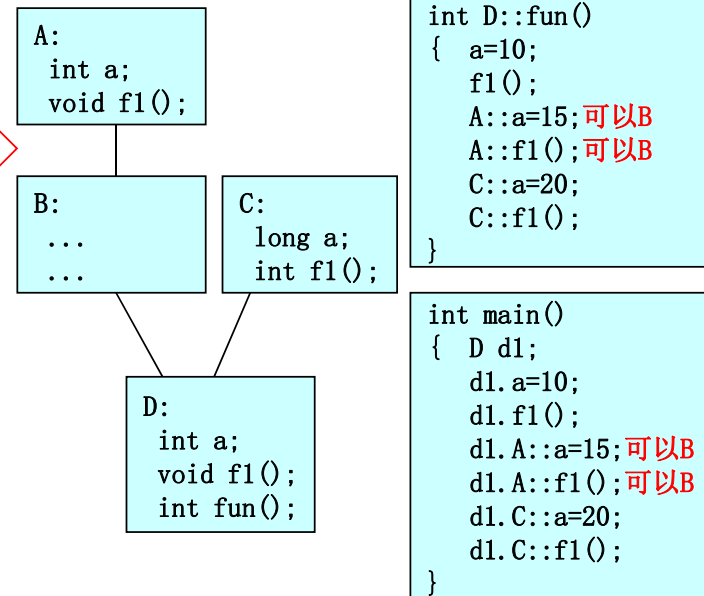
两个以上的基类与派生类的成员同名:

派生类直接访问, 不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名:

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则



可自行构造测试程序来验证



## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

例：写出右侧两个程序的运行结果（左侧代码相同）

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

class A {
public:
    int a;
    void f1() { return; }
};

class B:public A {
public:
    int b;
};

class C {
public:
    long a;
    int f1() { return 0; }
};

class D:public B, public C {
public:
    int a;
    char f1() { return 'A'; }
    void fun();
};
```

```
void D::fun()
{
    a = 10;
    f1();
    A::a = 15; // B::a = 15;
    A::f1();   // B::f1();
    C::a = 20;
    C::f1();
    int *p = (int *)this;
    cout << sizeof(*this) << endl;
    cout << *p      << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
}

int main()
{
    D d1;
    d1.fun();
    return 0;
}
```

?

```
void D::fun()
{
    return;
}

int main()
{
    D d1;
    d1.a = 10;
    d1.f1();
    d1.A::a = 15; // B::a = 15;
    d1.A::f1();   // B::f1();
    d1.C::a = 20;
    d1.C::f1();
    int *p = (int *)&d1;
    cout << sizeof(d1) << endl;
    cout << *p      << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```

?



## § 17. 继承和派生

### 6. 多重继承

#### 6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名:

按单继承的方式进行处理

(支配规则:派生类直接访问, 基类加作用域符)

两个以上的基类中的成员同名:

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名:

派生类直接访问, 不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名:

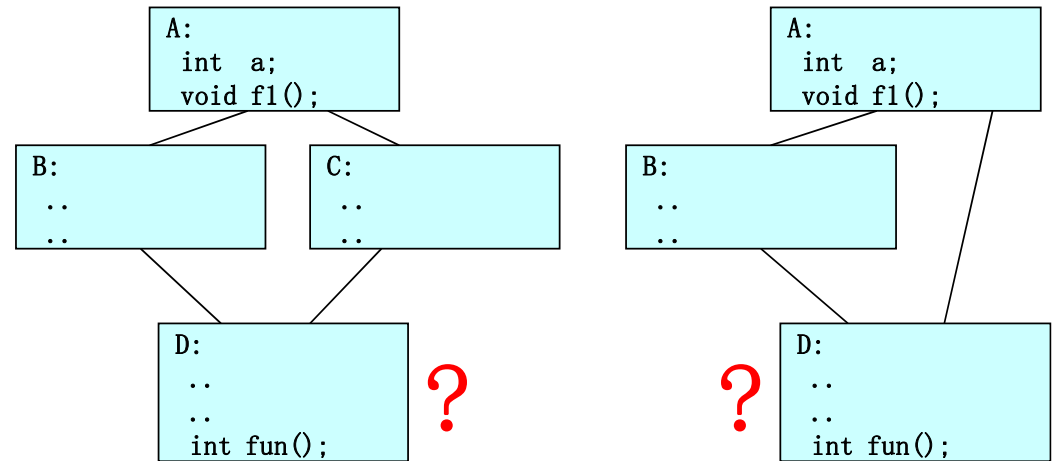
通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则

下面这两个例子中: A类中的成员在间接基类D中可能多次继承而产生重复问题

问1: 在派生类中是否有重复的间接基类数据成员?

问2: 如何处理多重继承引起的二义性问题(成员同名)





## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

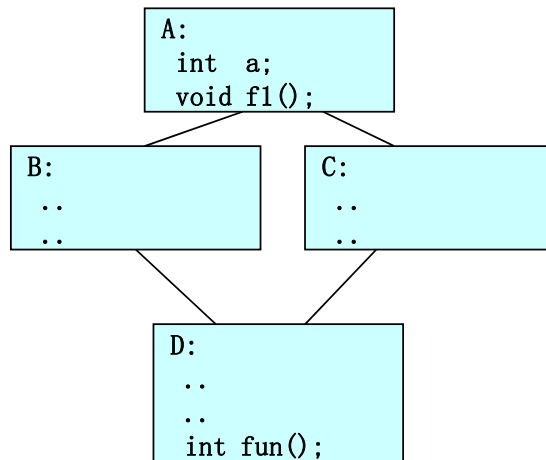
引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？ -> 答：有

问2：如何处理多重继承引起的二义性问题(成员同名)

D对象的内存映象(两份A)

A的数据成员	B
B的数据成员	
A的数据成员	C
C的数据成员	
D的数据成员	D



- 针对上面的继承序列，运行右侧的测试程序
- 1、先通过sizeof(d1)的结果确定A中的int a在d1对象中有2份
  - 2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
#include <iostream>
using namespace std;
static int x=0;
class A {
public:
    int a;
    A() { a = ++x;
        cout << "A(" << a << ")" << endl; }
};

class B :public A {
public:
    int b;
    B() { b = 20;
        cout << "B()" << endl; }
};

class C :public A {
public:
    int c;
    C() { c = 30;
        cout << "C()" << endl; }
};
```

```
class D :public B, public C {
public:
    int d;
    D()
    {    d = 40;
        cout << "D()" << endl;
    }
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;

    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    cout << *(p+4) << endl;

    return 0;
}
```

```
Microsoft
A(1)
B()
A(2)
C()
D()
20
1
20
2
30
40
```



## § 17. 继承和派生

### 6. 多重继承

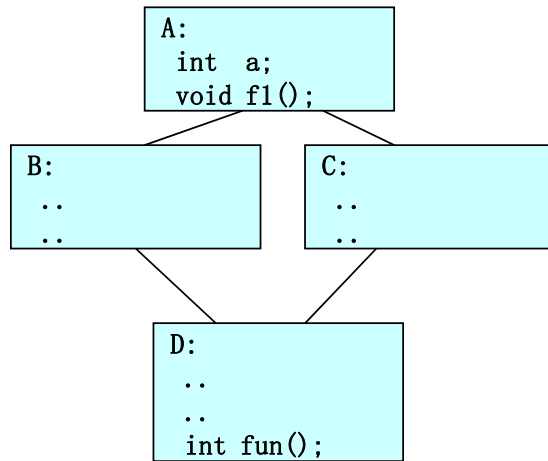
#### 6.4. 虚基类

引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？

问2：如何处理多重继承引起的二义性问题(成员同名)

答：通过可区分的类的作用域符来进行区分



针对上面的继承序列，运行右侧的测试程序

- 1、先通过sizeof(d1)的结果确定A中的int a在d1对象中有2份
- 2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
int D::fun()
{
    B::a=10;
    B::f1();
    C::a=15;
    C::f1();
    return 0;
} //不能/建议不用A::
```

```
int main()
{
    D d1;
    d1.B::a=10;
    d1.B::f1();
    d1.C::a=15;
    d1.C::f1();
    return 0;
} //不能/建议不用A::
```

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

static int x=0;
class A {
public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl; }
};

class B :public A {
public:
    int b;
};

class C :public A {
public:
    int c;
};

class D :public B, public C {
public:
    int d;
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;
    d1.A::f1();
    d1.B::f1();
    d1.C::f1();
    d1.f1();
    return 0;
}
```

四编译器分别编译  
哪些语句报错？





## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

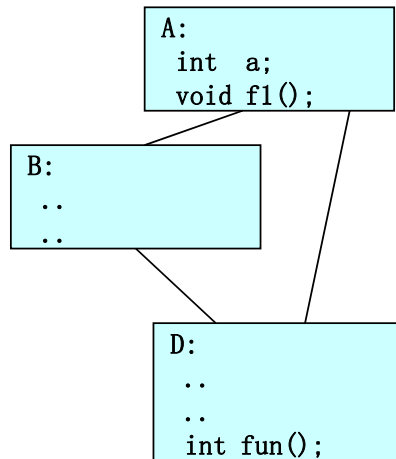
引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？ -> 答：有

问2：如何处理多重继承引起的二义性问题(成员同名)

D对象的内存映象(两份A)

A的数据成员	B
B的数据成员	
A的数据成员	C
D的数据成员	D



- 针对上面的继承序列，运行右侧的测试程序
- 1、先通过sizeof(d1)的结果确定A中的int a在d1对象中有2份
  - 2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

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

static int x = 0;

class A {
public:
    int a;
    A()
    { a = ++x;
      cout << "A(" << a << ")" << endl;
    }
};

class B :public A {
public:
    int b;
    B()
    { b = 20;
      cout << "B()" << endl;
    }
};
```

```
class D :public B, public A {
public:
    int d;
    D()
    { d = 40;
      cout << "D()" << endl;
    }
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;

    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;

    return 0;
}
```

[Warning] direct base 'A' inaccessible in 'D' due to ambiguity

warning C4584: "D": 基类 "A" 已是 "B" 的基类

```
A(1)
B()
A(2)
D()
16
1
20
2
40
```



## § 17. 继承和派生

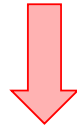
### 6. 多重继承

#### 6.4. 虚基类

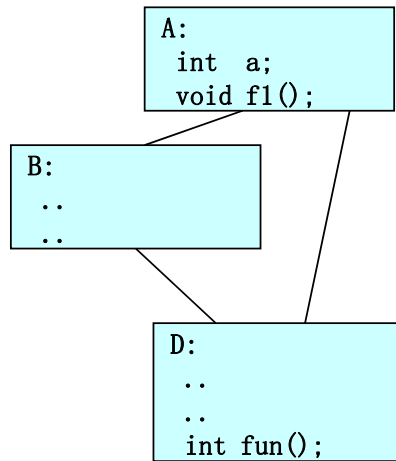
引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？

问2：如何处理多重继承引起的二义性问题(成员同名)



答：通过可区分的类的作用域符来进行区分, 部分无法区分



针对上面的继承序列，运行右侧的测试程序

- 1、先通过sizeof(d1)的结果确定A中的int a在d1对象中有2份
- 2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
int D::fun()
{
    A::a=10; /*/不建议
    A::f1(); /*/不建议
    B::a=15;
    B::f1();
    a=15;    x
    f1();    x
}
```

```
int main()
{
    D d1;
    d1.A::a=10; /*/不建议
    d1.A::f1(); /*/不建议
    d1.B::a=15;
    d1.B::f1();
    d1.a=15;    x
    d1.f1();    x
}
```

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

static int x=0;

class A {
public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl;}
};

class B :public A {
public:
    int b;
};

class D :public B, public A {
public:
    int d;
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;

    d1.A::f1();
    d1.B::f1();
    d1.f1();

    return 0;
}
```

四编译器分别编译  
哪些语句报错？



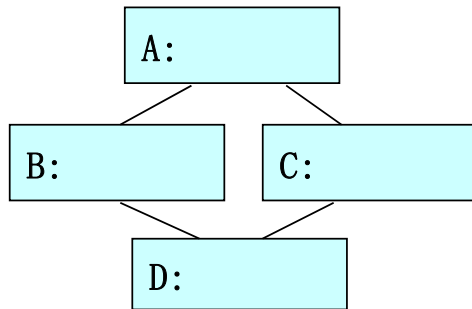
## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

含义：针对某个间接基类被多次继承而产生的多个无名对象，从而导致派生类中有多份相同的数据成员拷贝的情况，引入虚基类，使相同基类只保留一份数据成员

=> 目前D中有两份A，引入虚基类后，D中只有一份A



D对象内存映像(两份A)

A的数据成员	B
B的数据成员	
A的数据成员	C
C的数据成员	
D的数据成员	D

声明：

```
class 派生类名: virtual 存取限定符 基类名 {  
    ...  
};
```



## § 17. 继承和派生

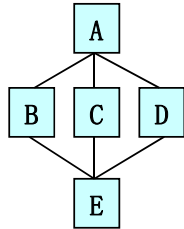
### 6. 多重继承

#### 6.4. 虚基类

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

```
class A {
public:
    A() {cout << "A()" << endl;}
};
class B:public A {
public:
    B() {cout << "B()" << endl;}
};
class C:public A {
public:
    C() {cout << "C()" << endl;}
};
class D:public A {
public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
public:
    E() {cout << "E()" << endl;}
};

int main()
{
    E e1;
}
```



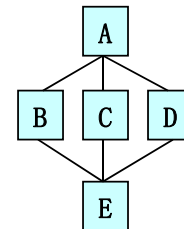
A() : 第1次  
B()  
A() : 第2次  
C()  
A() : 第3次  
D()  
E()

若定义: E e1 , 则E中只有三份A的拷贝

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

```
class A {
public:
    A() {cout << "A()" << endl;}
};
class B:virtual public A {
public:
    B() {cout << "B()" << endl;}
};
class C:virtual public A {
public:
    C() {cout << "C()" << endl;}
};
class D:virtual public A {
public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
public:
    E() {cout << "E()" << endl;}
};

int main()
{
    E e1;
}
```



A()  
B()  
C()  
D()  
E()

若定义: E e1 , 则E中只有一份A的拷贝



## § 17. 继承和派生

### 6. 多重继承

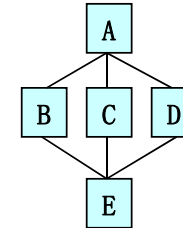
#### 6.4. 虚基类

声明:

```
class 派生类名: virtual 存取限定符 基类名 {  
    ...  
};
```

★ 所有继承该基类的直接派生类都应声明为虚基类  
才能保证只有一份数据拷贝

```
#include <iostream>  
using namespace std;  
  
class A {  
public:  
    A() {cout << "A()" << endl;}  
};  
class B: virtual public A {  
public:  
    B() {cout << "B()" << endl;}  
};  
class C: virtual public A {  
public:  
    C() {cout << "C()" << endl;}  
};  
class D: public A { //此处无virtual  
public:  
    D() {cout << "D()" << endl;}  
};  
class E: public B, public C, public D {  
public:  
    E() {cout << "E()" << endl;}  
};  
  
int main()  
{  
    E e1;  
}
```



A()  
B()  
C()  
A()  
D()  
E()

若定义: E e1 , 则e1中有两份A的拷贝



## § 17. 继承和派生

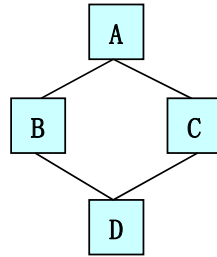
### 6. 多重继承

#### 6.4. 虚基类

虚基类的构造函数:

★ 由派生类直接负责给出间接虚基类的初始化参数若派生类中未给出, 则缺省调用无参构造

```
class A {
public:
    A()      { cout << "A()" << endl; } //无参构造
    A(int x) { cout << "A(" << x << ")" << endl; } //一参构造
};
class B:virtual public A {
public:
    B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
public:
    C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
public:
    D(int x, int y, int z):A(x), B(y), C(z) { cout << "D()" << endl; }
};
```



```
int main()
{
    D d1(1, 2, 3);
    cout << endl;
    C c1(4);
    cout << endl;
    B b1(5);
}
```

A(1)  
B()  
C()  
D()

A(4)  
C()

A()  
B()

如果通过D激活, 应该A(1)  
如果通过B激活, 应该A()  
如果通过C激活, 应该A(3)



## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

虚基类的构造函数:

★ 由派生类直接负责给出间接虚基类的初始化参数若派生类中未给出, 则缺省调用无参构造

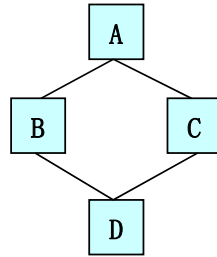
```
class A {  
public:  
    A()      { cout << "A()" << endl; } //无参构造  
    A(int x) { cout << "A(" << x << ")" << endl; } //一参构造  
};
```

```
class B:virtual public A {  
public:  
    B(int y) { cout << "B()" << endl; }  
};
```

```
class C:virtual public A {  
public:  
    C(int z):A(z) { cout << "C()" << endl; }  
};
```

```
class D:public B, public C {  
public:  
    D(int x, int y, int z):B(y),C(z) { cout << "D()" << endl; }  
};
```

```
int main()  
{  
    D d1(1,2,3);  
    cout << endl;  
    C c1(4);  
    cout << endl;  
    B b1(5);  
}
```



此处无A(x), 则调A的无参构造

A()  
B()  
C()  
D()

A(4)  
C()

A()  
B()



## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

虚基类的构造函数:

★ 由派生类直接负责给出间接虚基类的初始化参数若派生类中未给出, 则缺省调用无参构造

★ 调用时, 由派生类直接激活间接虚基类的构造函数, 其直接基类不再自动激活虚基类的构造

```
class A {
public:
    A() { cout << "A()" << endl; } //无参构造
    A(int x) { cout << "A(" << x << ")" << endl; } //一参构造
};
class B:virtual public A {
public:
    B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
public:
    C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
public:
    D(int x, int y, int z):A(x), B(y), C(z) { cout << "D()" << endl; }
};
```

假设 B b1(1) / C c1(1):  
(直接基类是虚基类的情况)

与非虚基类一样, b1/c1生成时  
会自动激活A的无参/有参构造  
函数

假设 D d1(1, 2, 3):

- ① d1对象生成时会自动激活A、  
B、C的构造函数(顺序不讨论),  
再调用D的构造函数
- ② B/C的构造函数被调用时,  
不再自动激活A的构造函数

★ 包含虚基类的构造函数及析构函数的调用顺序与普通的继承不同, 不再讨论  
(可自行查阅有关资料)





## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

不再讨论的内容(只给出现象, 原因自行研究):

★ 包含虚基类的派生类对象所占空间与普通继承不同, 不再讨论

★ 包含虚基类的派生类构造函数及析构函数的调用顺序与普通继承不同, 不再讨论

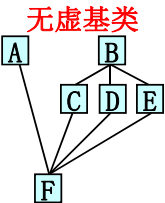
```
#include <iostream>
using namespace std;

class A {
public:
    int a;
    A() {cout << "A()" << endl; }
};

class B {
public:
    int b;
    B() {cout << "B()" << endl; }
};

class C : public B {
public:
    int c;
    C() {cout << "C()" << endl; }
};

class D : public B {
public:
    int d;
    D() {cout << "D()" << endl; }
};
```



```
class E : public B {
public:
    int e;
    E() {cout << "E()" << endl; }
};

class F : public A, public C,
          public D, public E {
public:
    int f;
    F() {cout << "F()" << endl; }
};

int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
    cout << sizeof(E) << endl;
    cout << sizeof(F) << endl;

    F f1;
    return 0;
}
```

无虚基类

```
Microsoft
4
4
8
8
8
8
32
A()
B()
C()
B()
D()
B()
E()
F()
```



## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

不再讨论的内容(只给出现象, 原因自行研究):

★ 包含虚基类的派生类对象所占空间与普通继承不同, 不再讨论

★ 包含虚基类的派生类构造函数及析构函数的调用顺序与普通继承不同, 不再讨论

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

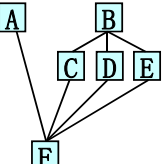
class A {
public:
    int a;
    A() {cout << "A()" << endl; }
};

class B {
public:
    int b;
    B() {cout << "B()" << endl; }
};

class C : virtual public B {
public:
    int c;
    C() {cout << "C()" << endl; }
};

class D : virtual public B {
public:
    int d;
    D() {cout << "D()" << endl; }
};
```

部分虚基类



```
class E : public B {
public:
    int e;
    E() {cout << "E()" << endl; }
};

class F : public A, public C,
          public D, public E {
public:
    int f;
    F() {cout << "F()" << endl; }
};

int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
    cout << sizeof(E) << endl;
    cout << sizeof(F) << endl;

    F f1;
    return 0;
}
```

深度思考:

- 1、如何理解有virtual后sizeof的大小多了4字节? 4字节存了什么?
- 2、如何理解总大小的变化(32=>36)

无虚基类

部分虚基类

无虚基类	部分虚基类
4	4
4	4
8	12
8	12
8	8
32	36
A()	B()
B()	A()
C()	C()
B()	D()
D()	B()
B()	E()
E()	F()
F()	



## § 17. 继承和派生

### 6. 多重继承

#### 6.4. 虚基类

不再讨论的内容(只给出现象, 原因自行研究):

★ 包含虚基类的派生类对象所占空间与普通继承不同, 不再讨论

★ 包含虚基类的派生类构造函数及析构函数的调用顺序与普通继承不同, 不再讨论

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

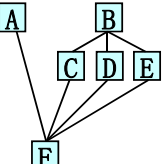
class A {
public:
    int a;
    A() {cout << "A()" << endl; }
};

class B {
public:
    int b;
    B() {cout << "B()" << endl; }
};

class C : virtual public B {
public:
    int c;
    C() {cout << "C()" << endl; }
};

class D : virtual public B {
public:
    int d;
    D() {cout << "D()" << endl; }
};
```

全部虚基类



```
class E : virtual public B {
public:
    int e;
    E() {cout << "E()" << endl; }
};

class F : public A, public C,
        public D, public E {
public:
    int f;
    F() {cout << "F()" << endl; }
};

int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
    cout << sizeof(E) << endl;
    cout << sizeof(F) << endl;

    F f1;
    return 0;
}
```

深度思考:

- 1、如何理解有virtual后sizeof的大小多了4字节? 4字节存了什么?
- 2、如何理解总大小的变化(32=>36=>36)

无虚基类

```
Microsoft
4
4
8
8
8
32
A()
B()
C()
B()
D()
B()
E()
F()
```

部分虚基类

```
Microsoft
4
4
12
12
8
36
B()
A()
C()
D()
B()
E()
F()
```

全部虚基类

```
Microsoft
4
4
12
12
12
36
B()
A()
C()
D()
E()
F()
```



## § 17. 继承和派生

### 7. 基类与派生类的转换

例:

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

class A {
public:
    int a;
};

class B:public A {
public:
    int b;
};

int main()
{
    A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl;    10
    a1 = b1;
    cout << a1.a << endl;    15
}
```

问题:

- 1、第16章概念, 系统会缺省对=进行重载, 但要求两侧都是相同类对象, 即  $A=A$  /  $B=B$
- 2、既未定义  $A=B$  的重载, 也未定义B转A的转换构造函数, 为什么此句不错?
- 3、换成  $B=A$  会怎样?  
即:  $b1=a1$ ;

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

class A {
public:
    int a;
};

class B:public A {
public:
    int b;
};

int main()
{
    A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl;
    b1 = a1;    //编译错!!!
    cout << a1.a << endl;
}
```

(21,12): error C2679: 二元“=”: 没有找到接受“A”类型的右操作数的运算符(或没有可接受的转换)  
(12,1): message : 可能是“B &B::operator =(B &B)”  
(12,1): message : 或 “B &B::operator =(const B &)”  
(21,12): message : 尝试匹配参数列表“(B, A)”时



## § 17. 继承和派生

### 7. 基类与派生类的转换

赋值兼容规则：在需要基类对象的**任何位置**，均可以使用**公有继承**的派生类对象

★ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问，因此不可用

★ 使用时，将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃



## § 17. 继承和派生

### 7. 基类与派生类的转换

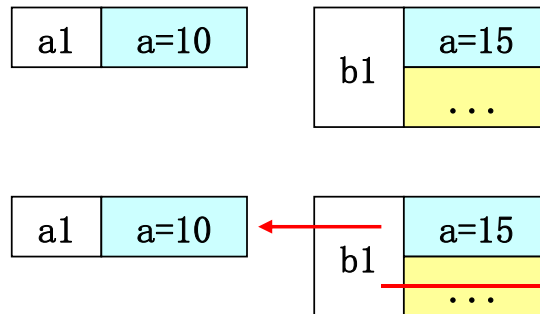
例：单继承下的赋值兼容规则

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

class A {
public:
    int a;
};

class B:public A {
public:
    int b;
};

int main()
{
    A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl; 10
    a1 = b1; 赋值兼容规则
    cout << a1.a << endl; 15
}
```



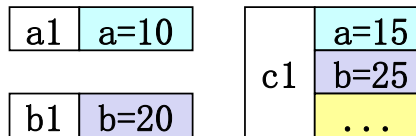


## § 17. 继承和派生

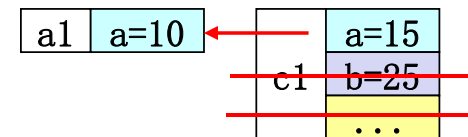
### 7. 基类与派生类的转换

例：多继承下的赋值兼容规则

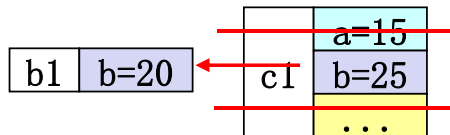
```
#include <iostream>
using namespace std;
class A {
public:
    int a;
};
class B {
public:
    int b;
};
class C:public A, public B {
public:
    int c;
};
int main()
{
    A a1;  B b1;  C c1;
    a1.a = 10;
    b1.b = 20;
    c1.a = 15;
    c1.b = 25;
}
```



```
cout << a1.a << endl;  10
a1 = c1;                赋值兼容规则
cout << a1.a << endl;  15
```



```
cout << b1.b << endl;  20
b1 = c1;                赋值兼容规则
cout << b1.b << endl;  25
```





## § 17. 继承和派生

### 7. 基类与派生类的转换

赋值兼容规则：在需要基类对象的任何位置，均可以使用公有继承的派生类对象

★ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问，因此不可用

★ 使用时，将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃

- 如果不希望直接对应拷贝，则可根据需要自行定义=重载或复制构造函数
- 当基类中包含动态申请内存时，赋值兼容规则可能出错（请参考=重载及复制构造函数）

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

class A {
public:
    int a;
};

class B:public A {
public:
    int b;
};

int main()
{
    B b1;
    b1.a = 15;
    A a1(b1);    //复制构造
    A a2;
    a2 = b1;     //重载
    cout << a1.a << endl;    //15
    cout << a2.a << endl;    //15
}
```

- 1、由赋值兼容规则可得到输出为15
- 2、如果希望输出为b1.a的2倍（30），则赋值兼容规则不适用，怎么办？

```
class B; //提前声明
class A {
public:
    int a;
    A() {} //无参构造
    A(B &b); //不能体内实现
    A& operator=(B &b); //不能体内实现
};

class B:public A {
public:
    int b;
};

A::A(B &b) //一参构造
{
    a = b.a*2;
}

A& A::operator=(B &b) //重载
{
    a = b.a*2;
    return (*this);
}

int main()
{
    B b1;
    b1.a = 15;
    A a1(b1), a2; //一参构造，无参构造
    a2 = b1;     //重载
    cout << a1.a << endl;    //30
    cout << a2.a << endl;    //30
}
```

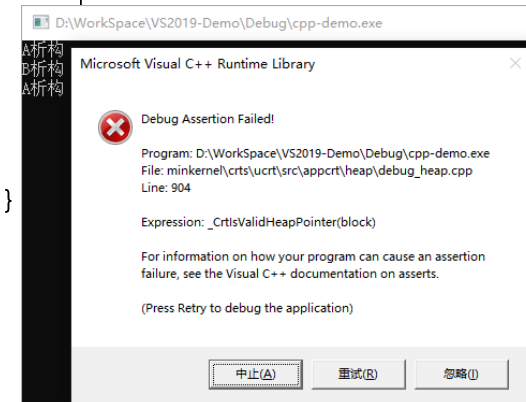
```
#include <iostream>
using namespace std;

class A {
private:
    char *s;
public:
    int a;
    A() { s = new char[20]; }
    ~A() {
        cout << "A析构" << endl;
        delete s;
    }
};

class B:public A {
public:
    int b;
    ~B() { cout << "B析构" << endl; }
};

int main()
{
    B b1;
    b1.a = 15;
    A a1(b1);    //复制构造
}
```

运行出错，具体请参考=重载及复制构造函数  
带动态申请时的错误分析（一定要自行弄懂!!!）







## § 17. 继承和派生

### 7. 基类与派生类的转换

赋值兼容规则：在需要基类对象的**任何位置**，均可以使用**公有继承**的派生类对象

★ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问，因此不可用

★ 使用时，将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃

● 如果不希望直接对应拷贝，则可根据需要自行定义=重载或复制构造函数

● 当基类中包含动态申请内存时，赋值兼容规则可能出错（请参考=重载及赋值构造函数）

★ 派生类对象可初始化基类的对象或引用

★ 派生类对象可出现在函数参数/返回值为基类的地方

★ 派生类对象可赋值给基类

★ 派生类对象的指针可出现在基类指针出现的位置

下面这两个例子，编译全部正确

```
class A { ... };
class B:public A { ... };
int main()
{
    B b1, *pb = &b1;
    A a1(b1); //需要A对象，用B对象替代
    A &a2 = b1; //需要A对象，用B对象替代
    A *pa;

    a1 = b1; //需要A对象，用B对象替代
    pa = pb; //需要A对象的地址，用B对象的地址替代
    pa = &b1; //需要A对象的地址，用B对象的地址替代

    return 0;
}
```

```
class A { ... };
class B:public A { ... };
void f1(A a1) { ... } //形参为A对象
void f2(A &a1) { ... } //形参为A对象的引用
void f3(A *pa) { ... } //形参为A对象的指针
A f4()
{ static B b1;
  return b1; //返回值为A对象
}
int main()
{ B b1;
  f1(b1); //B对象出现在要求A对象的位置
  f2(b1); //B对象出现在要求A对象引用的位置
  f3(&b1); //B对象出现在要求A对象指针的位置
}
```



## § 17. 继承和派生

### 8. 继承与组合

类的组合：一个类的对象是另一个类的数据成员

(两个类相互独立，无继承关系)

```
class Date {  
    private:  
        int year, month, day;  
    ...  
};
```

```
class student {  
    private:  
        Date birthday;  
};
```

★ 请注意两个类声明的顺序

★ 可能有提前声明