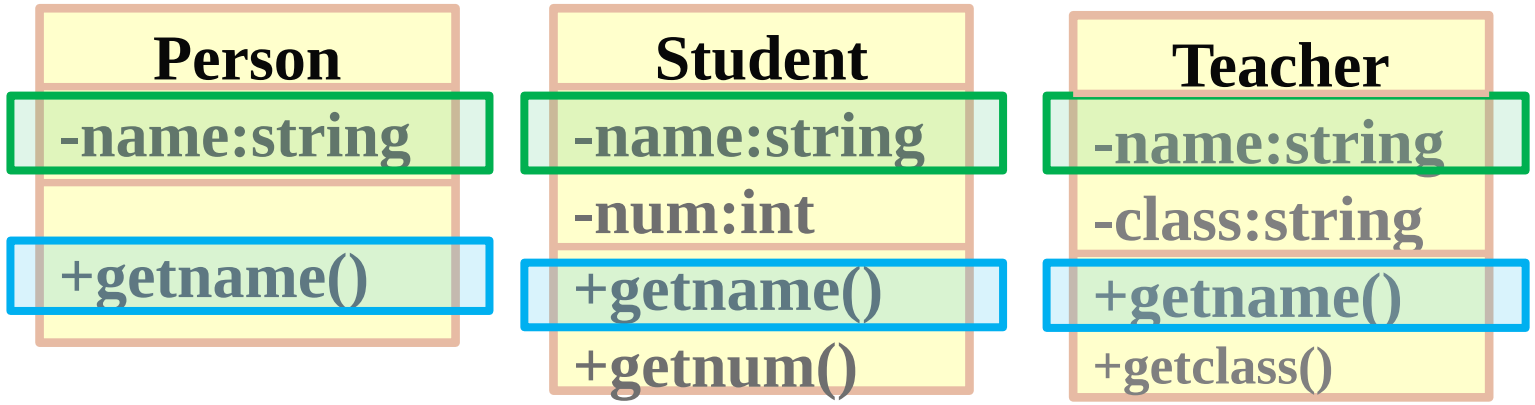
第2章

继承和派生

内容安排

- 2.1 继承和派生的概念
- 2.2 单继承
- 2.3 派生类的访问控制
- 2.4 派生类的构造函数和析构函数
- 2.5 多重继承

2.1 继承和派生的概念

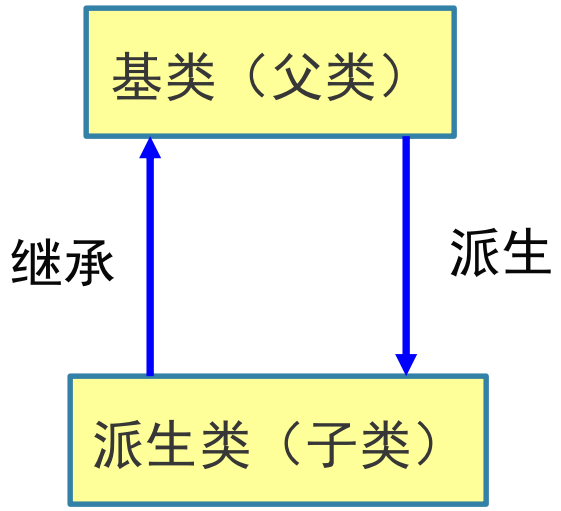


代码重复率太高！

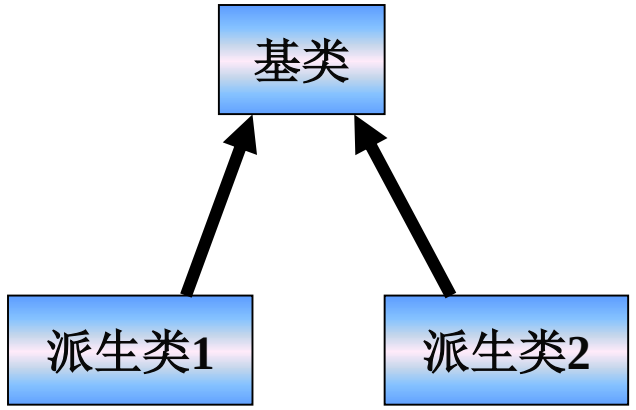
- 不利于代码维护
- 不利于代码重用

2.1 继承和派生的概念

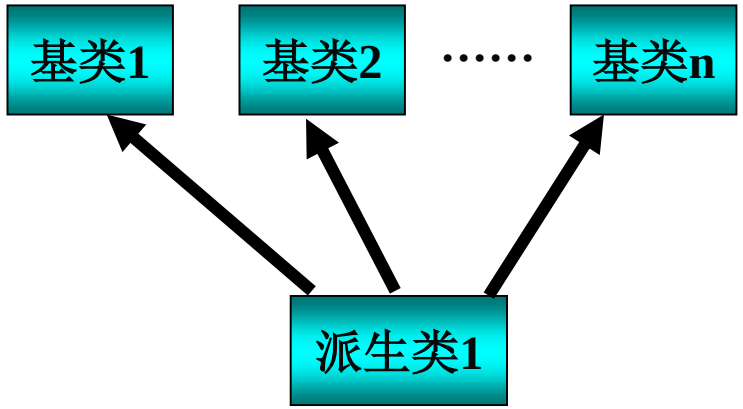
- 继承（Inheritance）就是在一个已有类的基础上建立一个新类，实质就是利用已有的数据类型定义出新的数据类型。
- 在继承关系中：
  - 被继承的类称为基类（Base class）（或父类）
  - 直接基类：在继承层次关系中，直接派生出新类的类
  - 间接基类：在继承层次关系中，处在直接基类之上的类
  - 定义出来的新类称为派生类（Derived class）（子类）



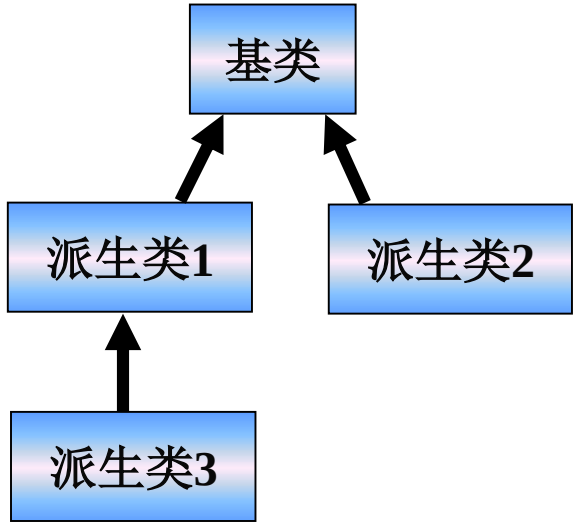
2.1 继承和派生的概念



单继承—派生类只有一个直接基类。



多重继承——派生类同时有多个直接基类。



派生类不仅可以继承原来类的成员，还可以通过以下方式扩充新的成员：

（1）增加新的数据成员

（2）增加新的成员函数

（3）重新定义已有成员函数

（4）改变现有成员的属性

2.2 单继承

❖ 单继承下派生类的定义格式：

class 派生类名: 继承方式 基类名

{ 派生类中的新成员 }

- 其中：

（1）class是定义类的关键字；派生类名由用户自己命名。

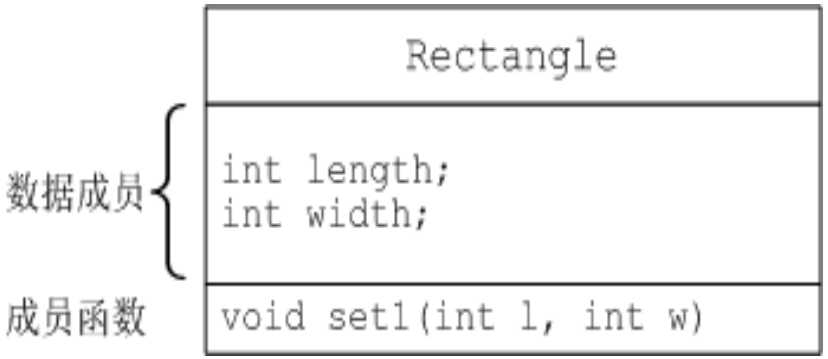
（2）在冒号“:”后的内容表明派生类是从哪个基类派生的以及在派生时的继承方式是什么。

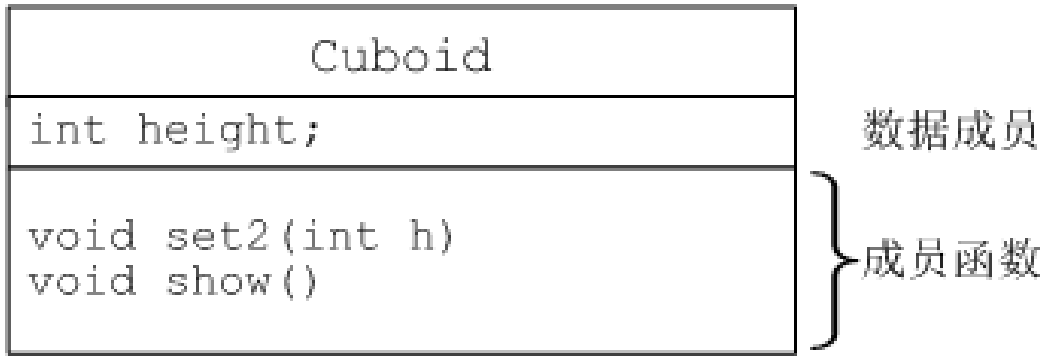
（3）“继承方式”即访问方式，可为：公有继承（public）、保护继承（protected）或私有继承（private），如果省略，则默认为private方式。

（4）“基类名”必须是已有的一个类的类名。

（5）大括号内的内容是派生类中新定义的成员或是对基类成员的改造。

- 【例2-1】 定义长方形类Rectangle
- 数据成员length和width分别表示长方形的长度和宽度，
- 成员函数只有set1，用于设置数据成员的值。
- 以Rectangle作为基类来定义派生类Cuboid（长方体类）
- 其中增加数据成员height、成员函数set2和show。



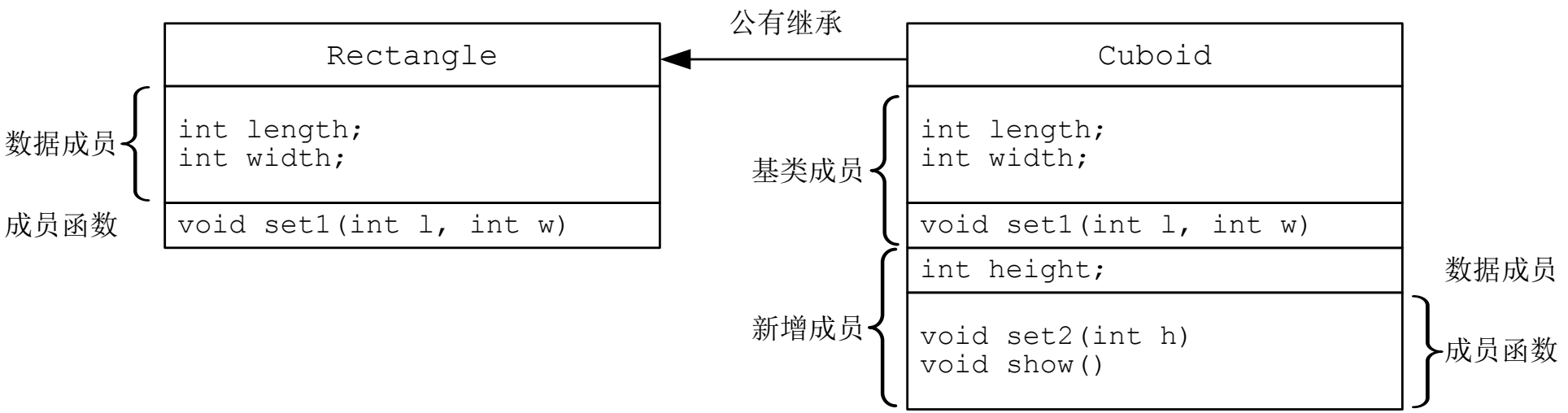


```
#include <iostream>
using namespace std;
class Rectangle
{
protected:
        int lenghth;
        int width;
public:
        void set1(int l, int w)
        {
                lenghth = l;
                width = w;
        }
};
```

公有继承

```
class Cuboid: public Rectangle
{
private:
        int height;
public:
        void set2(int h)
        {
                height = h;
        }
        void show()
        {
                cout<<"长度：  "<<lenghth<<endl;
                cout<<"宽度：  "<<width<<endl;
                cout<<"高度：  "<<height<<endl;
        }
};
```

int length;
int width;
void set1(int l int w)




```
长度：  20
宽度：  10
高度：  5
```

2.3 派生类的访问控制

- 2.3.1 派生类的成员构成
- 2.3.2 继承方式对基类成员的访问属性控制

2.3.1 派生类的成员构成

❖派生新类经历了三个步骤：

（1）吸收基类成员

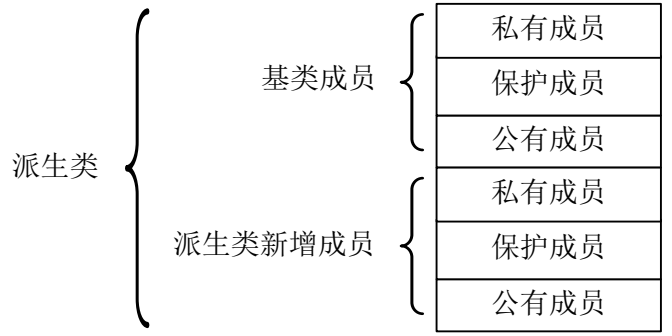
- 派生类继承和吸收了基类的全部数据成员和除了构造函数、析构函数之外的全部成员函数。

（2）添加新成员

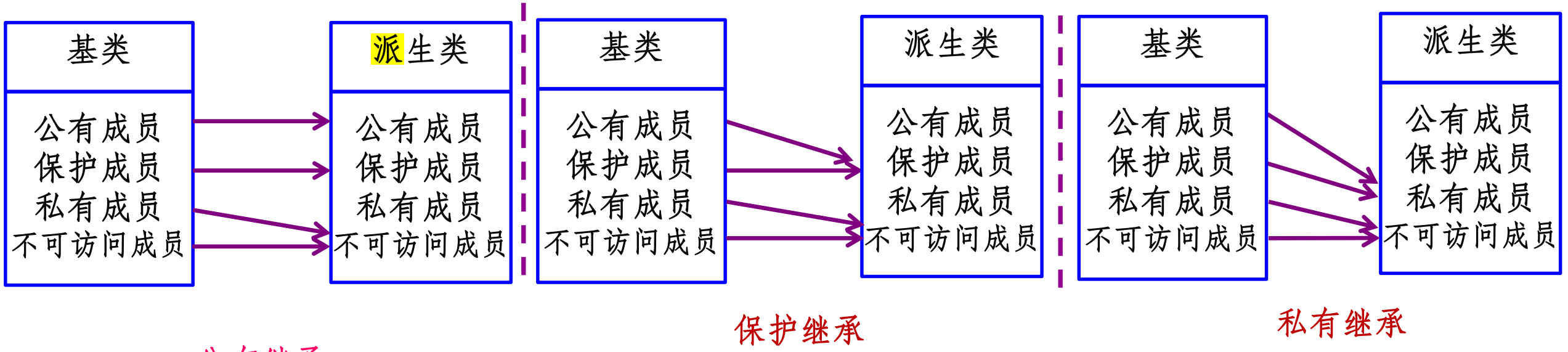
- 添加新的数据成员和成员函数，保证了派生类在功能上比基类有所发展。

（3）改造基类成员

- 一是基类成员的访问方式问题；
- 二是对基类数据成员或成员函数的覆盖。



2.3.2 继承方式对基类成员的访问属性控制

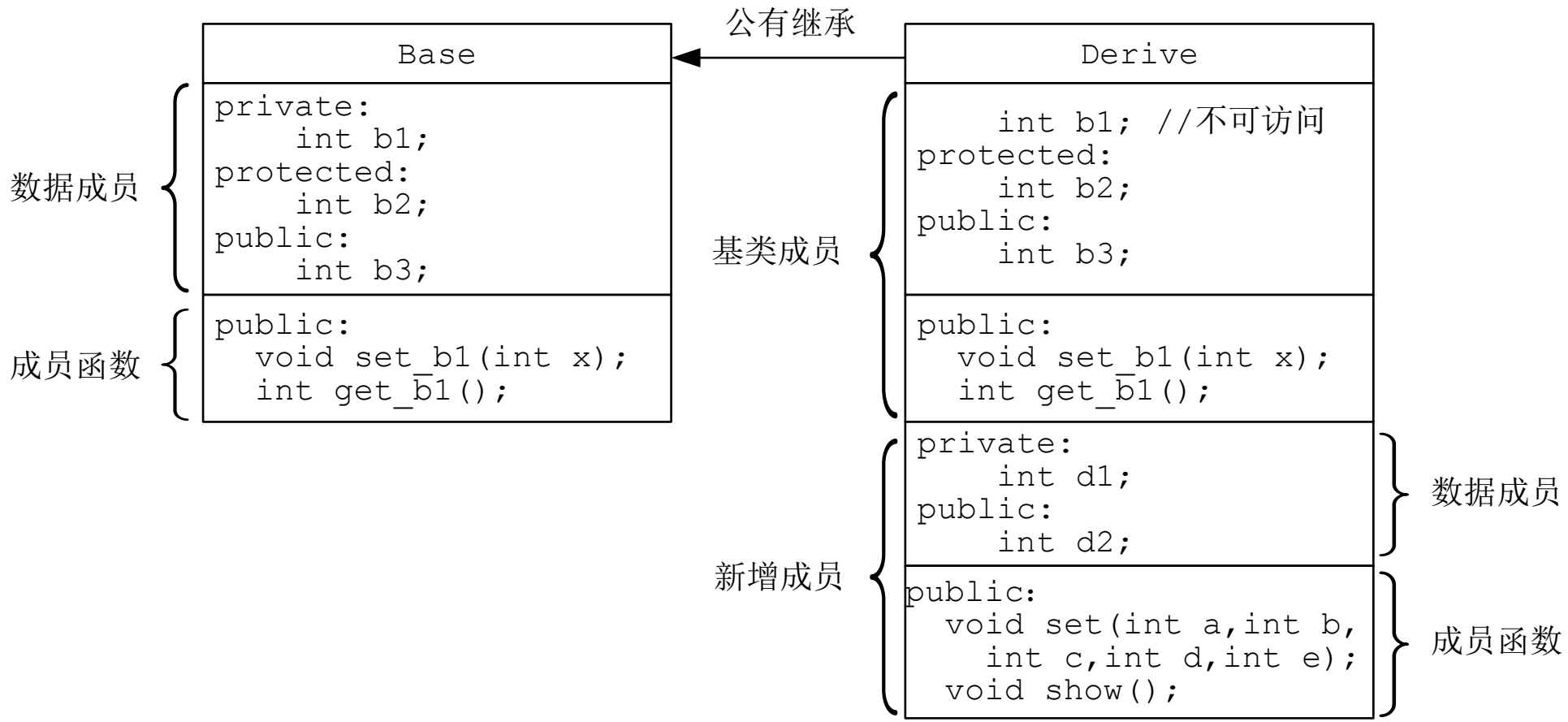


私有继承相当于终止了继续向下派生的路

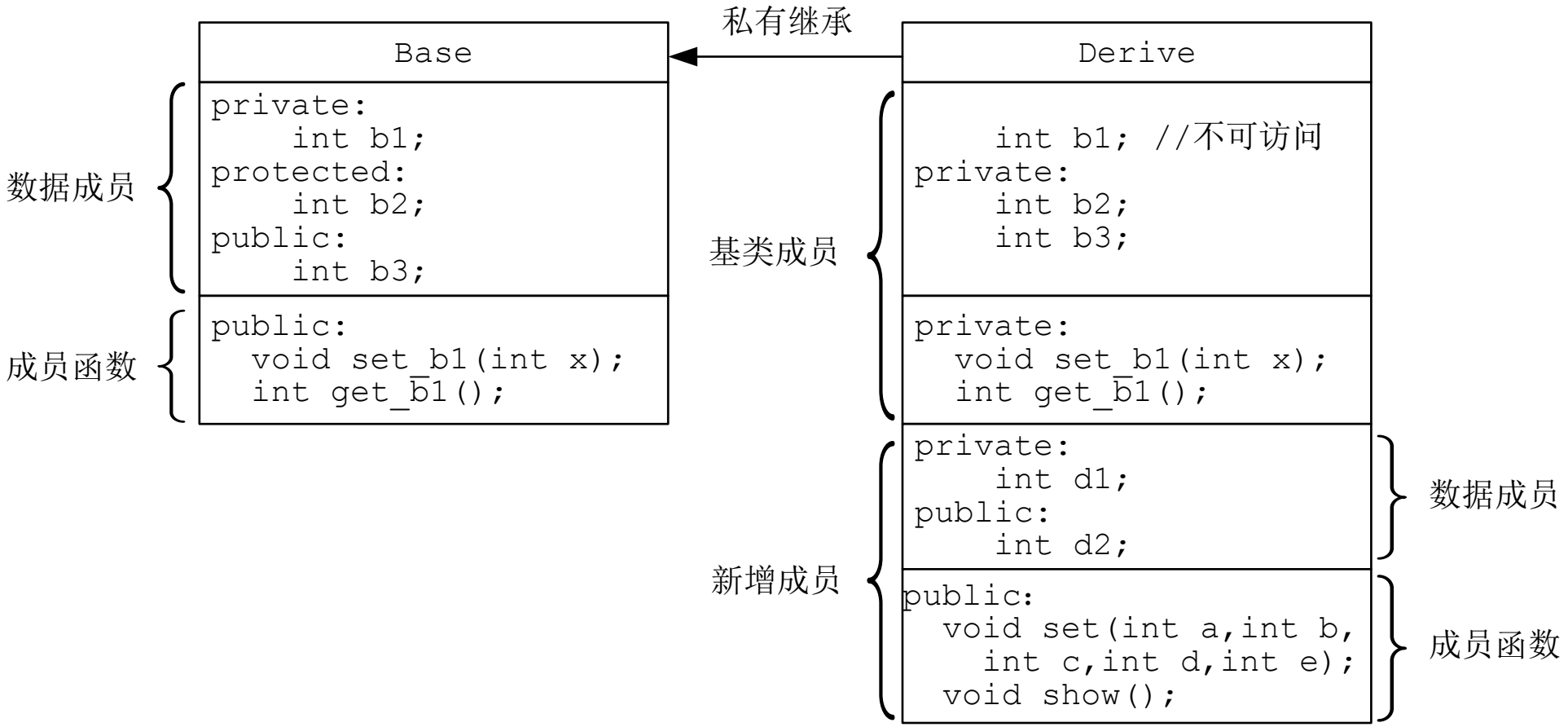
2.3.2 继承方式对基类成员的访问属性控制

基类成员	作为派生类成员		
	公有继承	保护继承	私有继承
public成员	public成员	protected成员	private成员
protected成员	protected成员	protected成员	private成员
private成员	不可访问	不可访问	不可访问
不可访问成员	不可访问	不可访问	不可访问

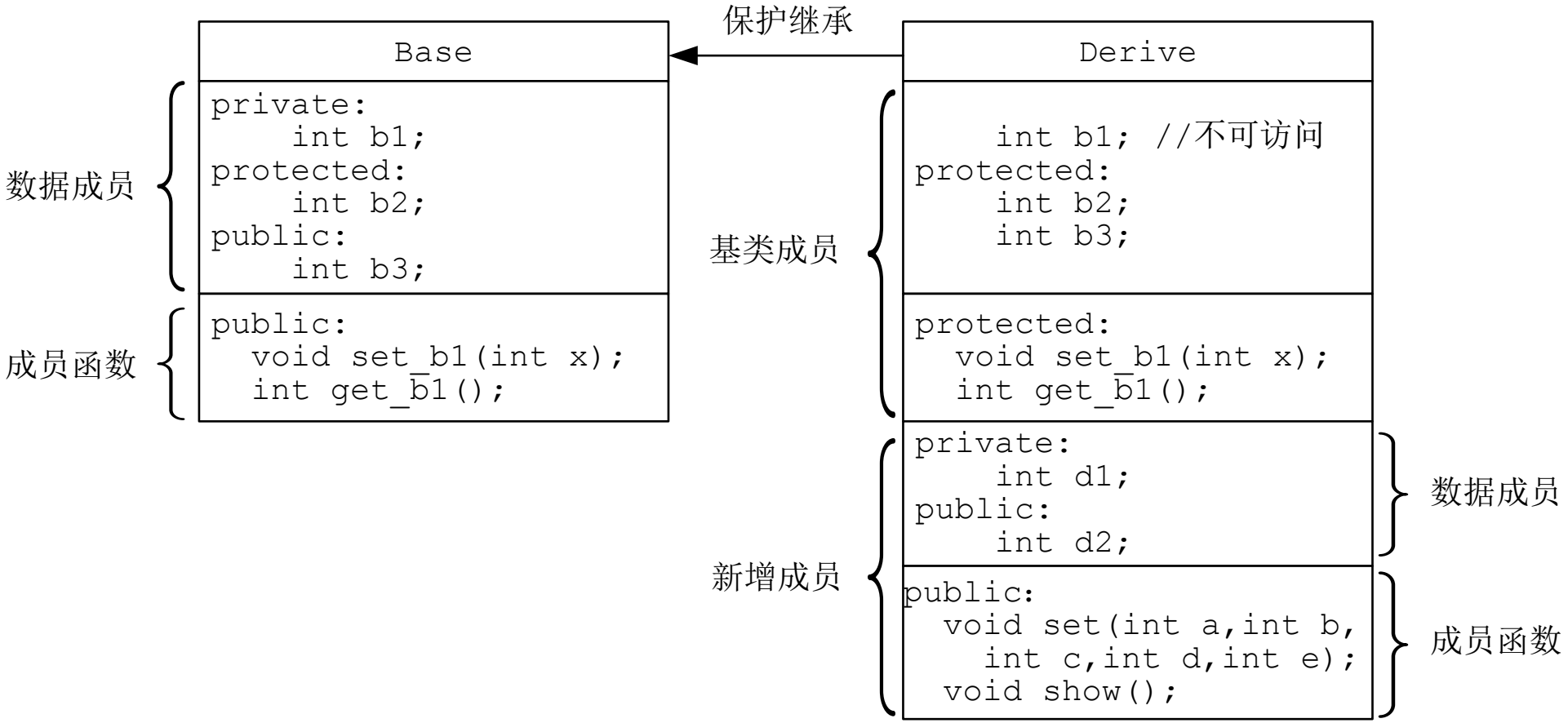
【例2-2】验证公有继承方式下，基类和派生类的成员的访问特性。



私有继承方式


私有继承
Base
Derive
数据成员
private:
int b1;
protected:
int b2;
public:
int b3;
成员函数
public:
void set_b1(int x);
int get_b1();
基类成员
int b1; //不可访问
private:
int b2;
int b3;
private:
void set_b1(int x);
int get_b1();
新增成员
private:
int d1;
public:
int d2;
数据成员
public:
void set(int a,int b,
int c,int d,int e);
void show();
成员函数

保护继承方式


保护继承
Base
private:
int b1;
protected:
int b2;
public:
int b3;
数据成员
public:
void set_b1(int x);
int get_b1();
成员函数
Derive
int b1; //不可访问
protected:
int b2;
int b3;
基类成员
protected:
void set_b1(int x);
int get_b1();
private:
int d1;
public:
int d2;
数据成员
新增成员
public:
void set(int a,int b,
int c,int d,int e);
void show();
成员函数

```
#include <iostream>
using namespace std;
class Base
{
private:
        int b1;
protected:
        int b2;
public:
        int b3;
        void set_b1(int x)
        {
                b1 = x;
        }
        int get_b1()
        {
                return b1;
        }
};
```

```
class Derive : public Base
{
private:
        int d1;
public:
        int d2;
        void set(int a, int b, int c, int d, int e)
        {
                set_b1(a);  //b1在派生类中不可访问
                b2 = b;
                b3 = c;
                d1 = d;
                d2 = e;
        }
        void show()
        {
                cout<<"b1 = "<<get_b1()<<endl;
                cout<<"b2 = "<<b2<<endl;
                cout<<"b3 = "<<b3<<endl;
                cout<<"d1 = "<<d1<<endl;
                cout<<"d2 = "<<d2<<endl;
        }
};
```

Derive类有哪些成员

在类外通过Derive类对象能访问哪些成员

```
int main()
{
        Derive obj;
        obj.set(1,2,3,4,5);
        obj.show();
//在类外通过派生类的对象只能访问派生类的公有成员，所以b1,b2,d1不可访问
//      cout<<"b1 = "<<obj.b1<<endl;
//      cout<<"b2 = "<<obj.b2<<endl;
        cout<<"b3 = "<<obj.b3<<endl;
//      cout<<"d1 = "<<obj.d1<<endl;
        cout<<"d2 = "<<obj.d2<<endl;
        return 0;
}
```

返回

2.4 派生类的构造函数和析构函数

- 2.4.1 派生类的构造函数
- 2.4.2 有子对象的派生类的构造函数
- 2.4.3 派生类的析构函数

2.4.1 派生类的构造函数

- 派生类的构造函数：
  - 需要负责调用基类的构造函数对基类成员进行初始化；
- 派生类的构造函数定义格式：

是「直接基类」sp哦。

派生类名::派生类构造函数名（总参数列表）：基类构造函数名（参数列表）

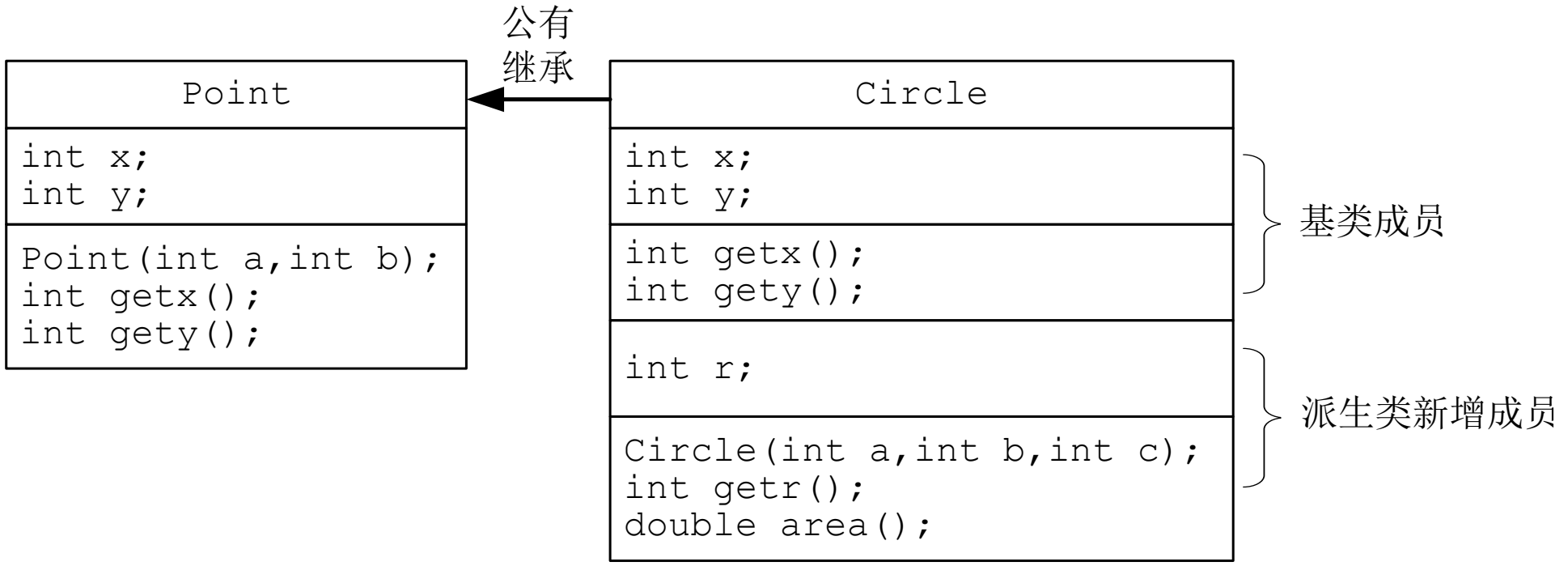
{派生类中新增数据成员初始化语句}

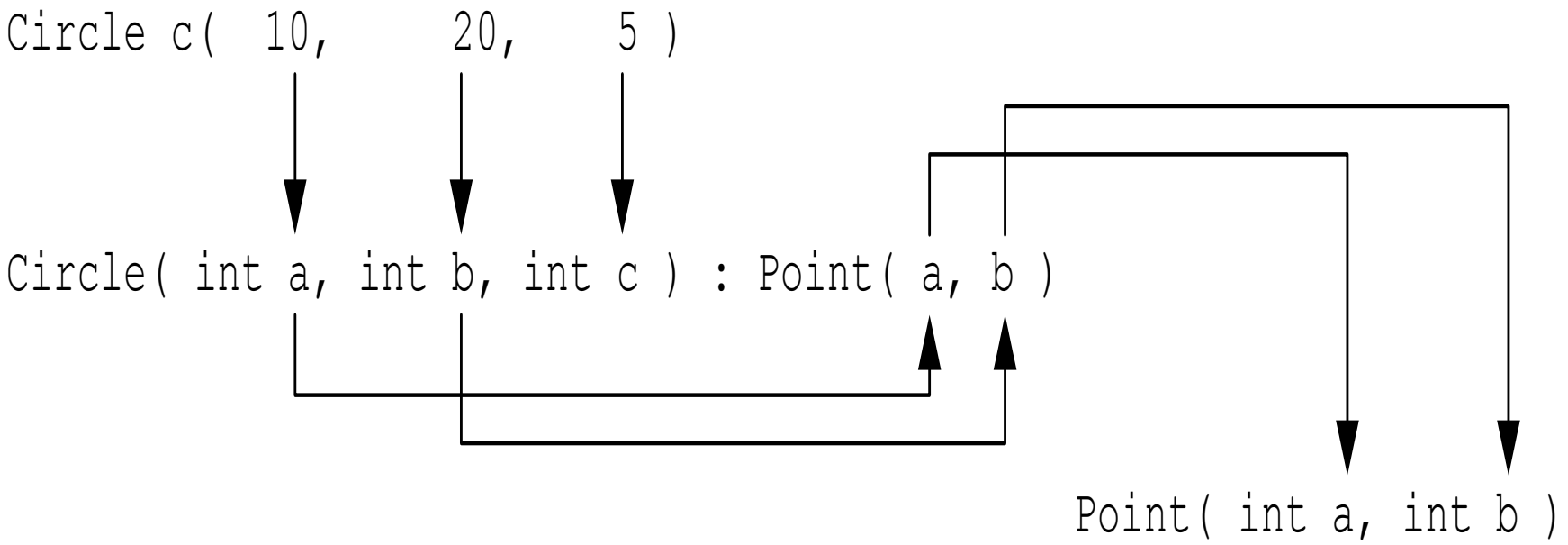
- 【例2-3】设计一个点类Point，包含横、纵坐标两个数据成员，由它派生出圆类Circle，添加一个半径数据，并求其面积。测试基类的构造函数和派生类的构造函数的执行顺序。

```
#include <iostream>
using namespace std;
class Point
{
private:
        int x;
        int y;
public:
        Point(int a, int b)
        {
                cout<<"基类Point的构造函数被执行\n";
                x = a;
                y = b;
        }
        int getx()
        {
                return x;
        }
        int gety()
        {
                return y;
        }
};
```

```
class Circle : public Point
{
private:
        int r;
public:
        Circle(int a, int b, int c):Point(a,b)
        {
                cout<<"派生类Circle的构造函数被执行\n";
                r = c;
        }
        int getr()
        {
                return r;
        }
        double area()
        {
                return 3.14159*r*r;
        }
};
int main()
{
        Circle c(10,20,5);
        cout<<"圆的圆心为：("<<c.getx()<<","<<c.gety()<<")"<<"\n";
        cout<<"圆的半径为： "<<c.getr()<<"\n";
        cout<<"圆的面积为： "<<c.area()<<endl;
        return 0;
}
```

```
基类Point的构造函数被执行
派生类Circle的构造函数被执行
圆的圆心为：(10,20)
圆的半径为： 5
圆的面积为： 78.5397
```


公有
继承
Point
int x;
int y;
Point(int a,int b);
int getx();
int gety();
Circle
int x;
int y;
int getx();
int gety();
int r;
Circle(int a,int b,int c);
int getr();
double area();
基类成员
派生类新增成员


Circle c(10, 20, 5)
Circle(int a, int b, int c) : Point(a, b)
Point(int a, int b)

- 【例2-4】定义一个描述圆的类Circle和一个描述圆柱体的类Cylinder，并计算其相应的面积。

```
#include <iostream>
using namespace std;
class Circle
{
protected:
        double r;
public:
        Circle(double ra)
        {
                r = ra;
        }
        double area()
        {
                return 3.14159*r*r;
        }
};
```

同名覆盖

```
class Cylinder : public Circle
{
protected:
        double h;
public:
        Cylinder(double rv, double hv);
        double area()
        {
                return 2*Circle::area()+2*3.14*r*h;
        }
};
Cylinder::Cylinder(double rv, double hv): Circle(rv)
{
        h = hv;
}
int main()
{
        Circle cir(5);
        Cylinder cylin(5,10);
        cout<<"圆的面积为： "<<cir.area()<<"\n";
        cout<<"圆柱体的底面积为： "<<cylin.Circle::area()<<"\n";
        cout<<"圆柱体的表面积为： "<<cylin.area()<<endl;
        return 0;
}
```

```
圆的面积为： 78.5397
圆柱体的底面积为： 78.5397
圆柱体的表面积为： 471.079
```

说明

在C++中，处理同名函数时有以下3种基本方法：

- ① 根据函数参数的特征进行区分。如：

  max(int,int)

  max(float,float)

- ② 根据类对象进行区分。

  cylinder.area()

  circle.area()

- ③ 使用作用域运算符“::”进行区分，如：

  Circle::area()

2.4.2 有子对象的派生类的构造函数

派生类名::派生类构造函数名(总参数列表)：基类构造函数名(参数列表),

子对象名(参数列表),……

{ 派生类中新增数据成员初始化语句 }

- 此时，构造函数执行的一般次序为：

① 调用基类的构造函数。

② 调用子对象的构造函数。当派生类中含有多个子对象时，各子对象的构造函数的调用顺序按照它们在类中说明的先后顺序进行。

③ 执行派生类构造函数的函数体。

2.4.3 派生类的析构函数

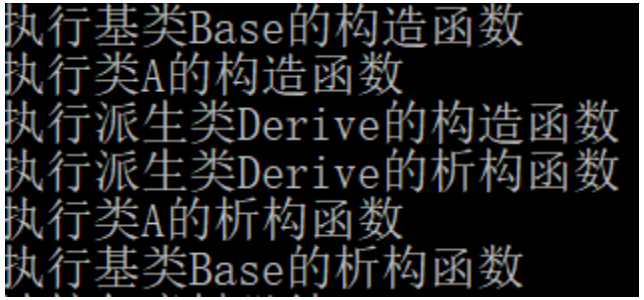
- 析构函数的作用是在对象撤销之前，进行必要的清理工作。
- 当对象被删除时，系统会自动调用析构函数。
- 析构函数的调用顺序与构造函数的调用顺序正好相反：
  - 先执行派生类自己的析构函数；
  - 然后调用子对象的析构函数；
  - 最后调用基类的析构函数。

- 【例2-5】 分析程序的执行结果。

```
#include <iostream>
using namespace std;
class Base
{
private:
        int b;
public:
        Base(int i)
        {
            cout<<"执行基类Base的构造函数\n";
            b = i;
        }
        ~Base()
        {
            cout<<"执行基类Base的析构函数\n";
        }
};
```

```
class A
{
private:
        int a;
public:
        A(int n)
        {
                a = n;
                cout<<"执行类A的构造函数\n";
        }
        ~A()
        {
                cout<<"执行类A的析构函数\n";
        }
};
class Derive : public Base
{
private:
        int d;
        A aobj;
public:
        Derive(int a, int b, int c): Base(b), aobj(a)
        {
                d = c;
                cout<<"执行派生类Derive的构造函数\n";
        }
        ~Derive()
        {
                cout<<"执行派生类Derive的析构函数\n";
        }
};
```

```
int main()
{
        Derive dobj(10,20,30);
        return 0;
}
```



2.5 多重继承

- 2.5.1 多重继承的定义方式
- 2.5.2 多重继承的二义性（即同名冲突）
- 2.5.3 虚基类及其派生类的构造函数

2.5.1 多重继承的定义方式

class 派生类名：继承方式 基类名1，继承方式 基类名2 ……

{ …… };

- 其中：

（1）继承方式为public、protected或private，功能同单一继承。

（2）在冒号“:”后要列出派生类的所有基类及其继承方式，并用逗号做分隔符。

- 派生类的构造函数

  派生类名::派生类构造函数名（参数表）：基类名1 (参数表1),基类名2(参数表2),……

  { 派生类中新增数据成员初始化语句 }

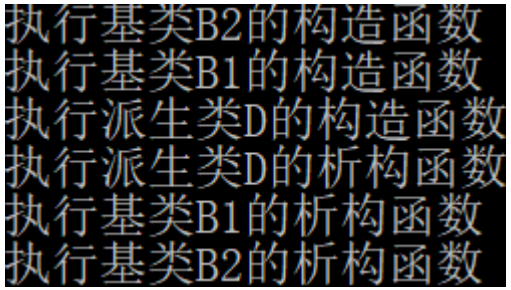
- 在多重继承下，系统首先执行各基类的构造函数，然后再执行派生类的构造函数；
- 处于同一层次的各基类构造函数的执行顺序与声明派生类时所指定的各基类顺序一致，而与派生类的构造函数定义中所调用的基类构造函数的顺序无关。
- 【例2-6】 测试在多重继承关系下析构函数的执行顺序。

```
#include <iostream>
using namespace std;
class B1
{
protected:
        int b1;
public:
        B1(int i)
        {
                cout<<"执行基类B1的构造函数\n";
                b1 = i;
        }
        ~B1()
        {
                cout<<"执行基类B1的析构函数\n";
        }
};
```

```
class B2
{
protected:
        int b2;
public:
        B2(int n)
        {
                b2 = n;
                cout<<"执行基类B2的构造函数\n";
        }
        ~B2()
        {
                cout<<"执行基类B2的析构函数\n";
        }
};
class Derive : public B2, public B1
{
protected:
        int d;
public:
        Derive(int a, int b, int c): B1(a), B2(b)
        {
                d = c;
                cout<<"执行派生类D的构造函数\n";
        }
        ~Derive()
        {
                cout<<"执行派生类D的析构函数\n";
        }
};
```

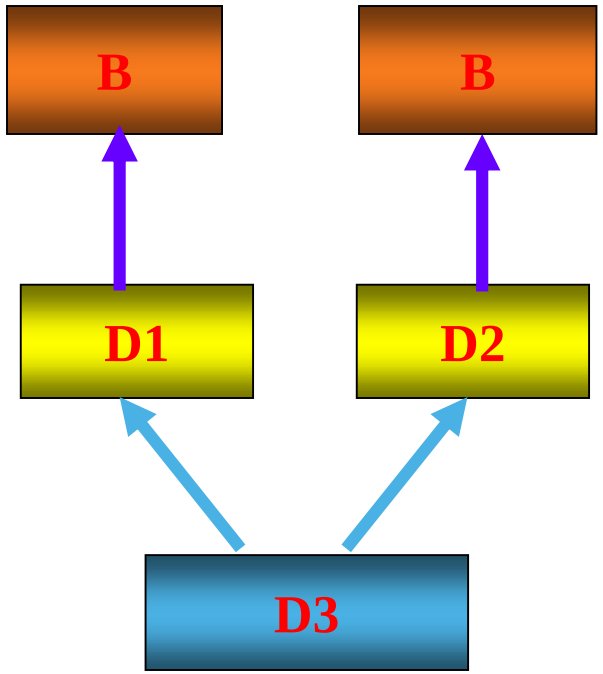
先执行哪个基类的构造函数

```
int main()
{
        Derive dobj(10,20,30);
        return 0;
}
```



2.5.2 多重继承的二义性（即同名冲突）

- 多重继承下，可能会产生一个类是通过多条路径从一个给定的类中派生出来的情况，会造成派生类对基类成员访问的不唯一性，即二义性。



通过派生类D3的对象访问类B的成员？

2.5.2 多重继承的二义性（即同名冲突）

- 同名成员的二义性。被继承的多个基类中具有同名成员，在派生类中对该同名成员的访问会产生二义性。 (修改li2_6)

  解决方法：使用作用域运算符

  直接基类名::数据成员名

  直接基类名::成员函数名（参数表）

- 同一个基类被多次继承产生的二义性。被继承的多个基类有一个共同的基类，在派生类中访问这个共同基类的成员会产生二义性。 (修改li2_7)

  解决方法：将直接基类的共同基类设置为虚基类

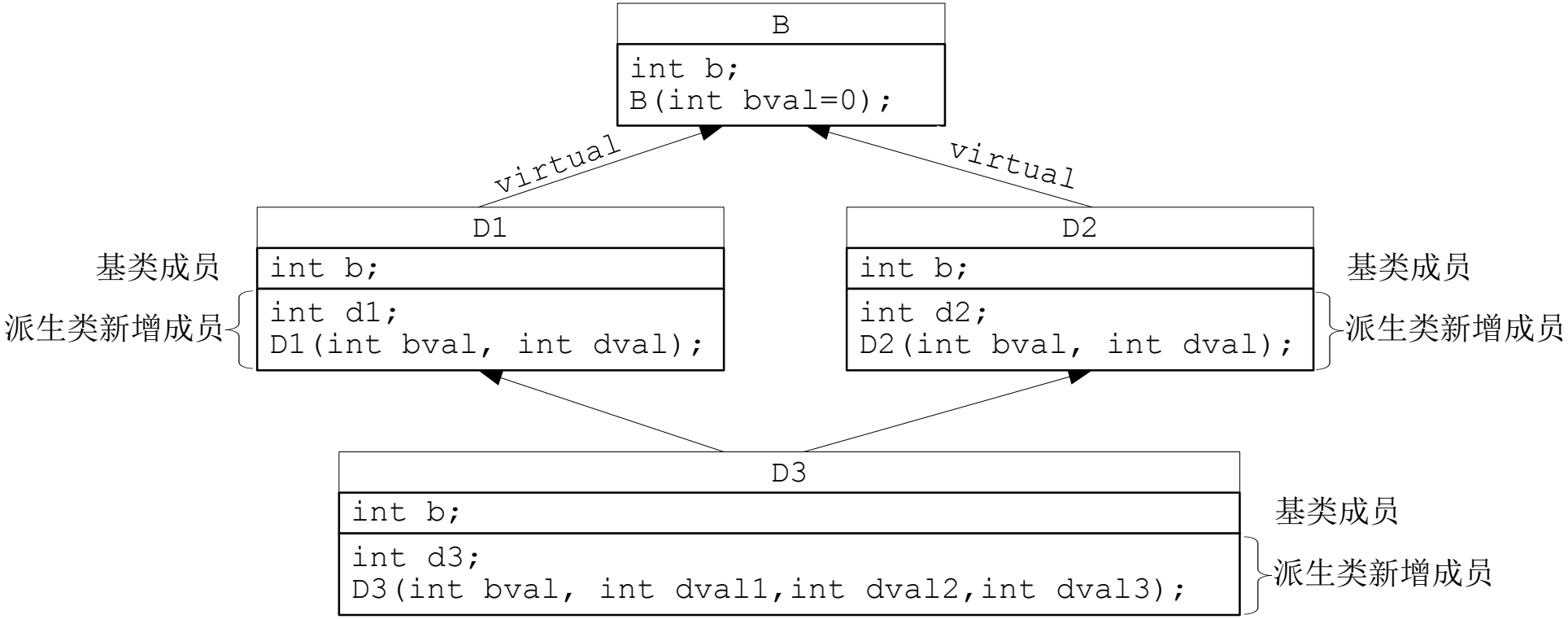
  class 派生类名： **virtual** 继承方式 基类名

  { …… };

2.5.3 虚基类及其派生类的构造函数

- 虚基类：虽然被一个派生类间接地多次继承，但派生类却只继承一份该基类的成员，这样就避免了在派生类中访问这些成员时产生二义性。
- 使用虚基类时，要特别注意派生类的构造函数。
  - 对于普通基类，派生类的构造函数负责调用其直接基类的构造函数以初始化其直接基类的数据成员；
  - 而对于虚基类的任何派生类，其构造函数不仅负责调用直接基类的构造函数，还需调用虚基类的构造函数。

❖【例2-7】按照下图所示的类之间的关系，设计对应的类和相关的构造函数，并将类B声明为虚基类，测试虚基类的作用。



```
#include <iostream>
using namespace std;
class B
{
protected:
        int b;
public:
        B(int bval)
        {
                cout<<"执行类B的构造函数\n";
                b = bval;
        }
};
class D1 : virtual public B
{
protected:
        int d1;
public:
        D1(int bval, int dval) : B(bval)
        {
                d1 = dval;
                cout<<"执行类D1的构造函数\n";
        }
};
```

```
class D2 : virtual public B
{
protected:
        int d2;
public:
        D2(int bval, int dval) : B(bval)
        {
                d2 = dval;
                cout<<"执行类D2的构造函数\n";
        }
};
class Derive : public D1, public D2
{
protected:
        int d3;
public:
        Derive(int bval, int dval1, int dval2, int dval3): D1(bval, dval1), D2(bval, dval2), B(bval)
        {
                 d3 = dval3;
                cout<<"执行类Derive的构造函数\n";
        }
        void show()
        {
                cout<<"成员b的值为： "<<b<<"\n";
                cout<<"成员d1的值为： "<<d1<<"\n";
                cout<<"成员d2的值为： "<<d2<<"\n";
                cout<<"成员d3的值为： "<<d3<<"\n";
        }
};
```

```
int main()
{
        Derive dobj(10,20,30,40);
        dobj.show();
        return 0;
}
```

```
执行类B的构造函数
执行类D1的构造函数
执行类D2的构造函数
执行类Derive的构造函数
成员b的值为: 10
成员d1的值为: 20
成员d2的值为: 30
成员d3的值为: 40
```

说明：

- 虚基类的构造函数只能被调用1次，把建立对象时所指定的类称为最终派生类，虚基类的构造函数是由最终派生类的构造函数调用的。
- 构造函数的调用顺序：
  - 所有虚基类的构造函数（按定义顺序）
  - 所有直接基类的构造函数（按定义顺序）
  - 所有对象成员的构造函数（按定义顺序）
  - 派生类自己的构造函数
- 析构函数的调用顺序与构造函数相反