

面向对象程序设计教程

继承与派生

苏州大学计算机科学与技术学院
面向对象与C++程序设计课程组

类与继承

- 继承的基础:类

- 继承

- ◆ 就是用已经存在的类创建新类，新类继承已经存在类的特性，此特性包括成员数据、成员函数和访问权限。

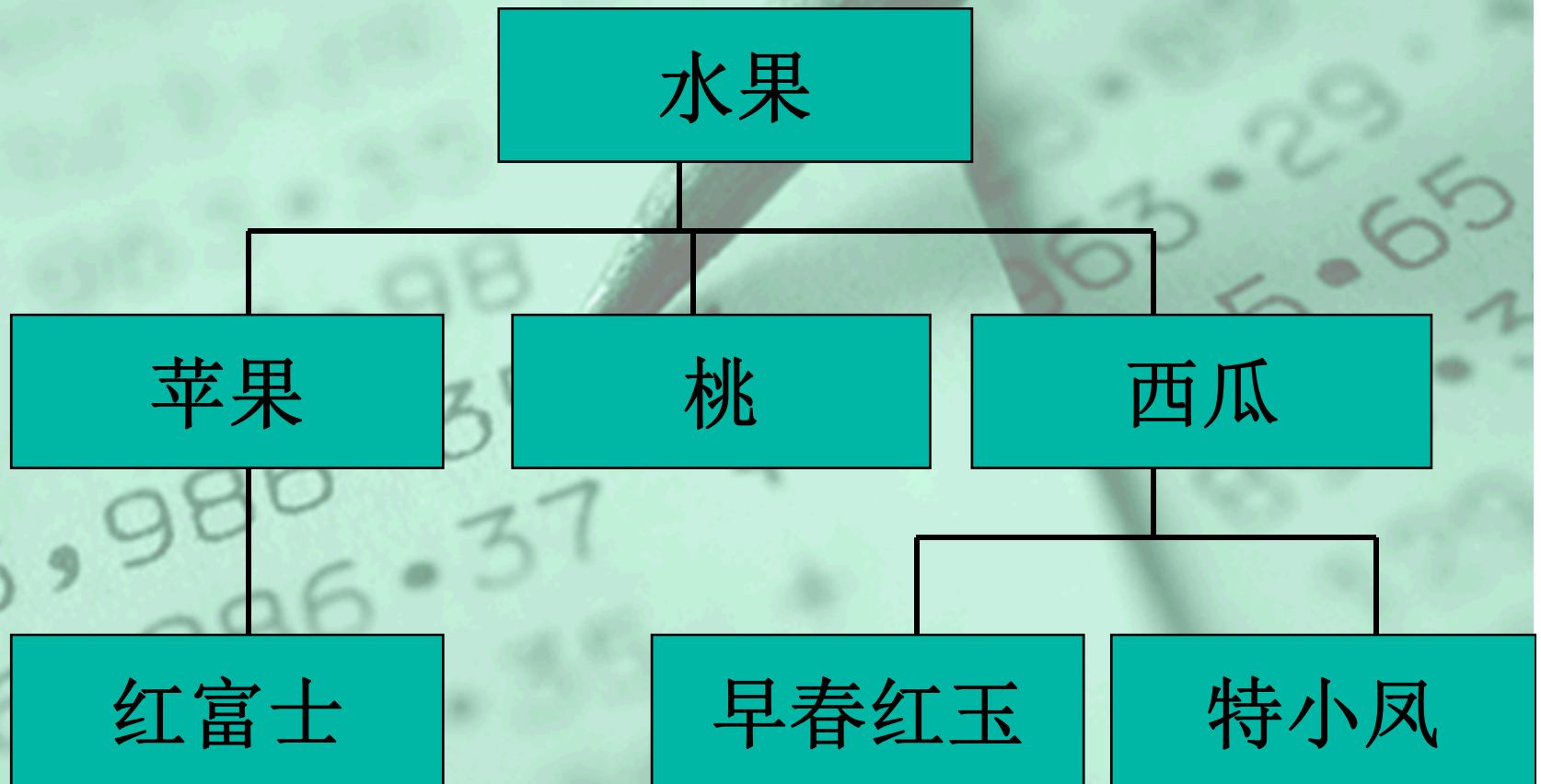
- 代码重用的重要手段

- 帮我们描述事物的层次关系

- 使程序更加容易理解与扩展

与现实世界相符

类的继承与派生关系与客观世界的共性与特性、一般与特殊关系一致



继承与派生的目的

- 继承的目的：实现代码重用。
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。

继承与派生的形式

- 单一继承

- **class** <派生类名>:<基类存取限定符><基类名>
{
 ...
};

- 多重继承

- **class** <派生类名>:<基类存取限定符><基类名>
 ,<基类存取限定符><基类名> ...
{
 ...
};

单一继承与多重继承



继承方式

- 不同继承方式的影响主要体现在：
 - 1、派生类成员对基类成员的访问控制。
 - 2、派生类对象对基类成员的访问控制。
- 三种继承方式
 - ◆ 公有继承
 - ◆ 私有继承
 - ◆ 保护继承

公有继承(public)

- 基类的**public**和**protected**成员的访问属性在派生类中保持不变，但基类的**private**成员不可访问。
- 派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但不能访问基类的**private**成员。
- 通过派生类的对象只能访问基类的**public**成员。

例1 公有继承举例

```
class Point //基类Point类的声明
{public:      //公有函数成员
    void InitP(float xx=0, float yy=0)
    {X=xx;Y=yy;}
    void Move(float xOff, float yOff)
    {X+=xOff;Y+=yOff;}
    float GetX ( ) {return X;}
    float GetY ( ) {return Y;}
private:    //私有数据成员
    float X,Y;
};
```

```
class Rectangle: public Point //派生类声明
{
public:      //新增公有函数成员
    void InitR(float x, float y, float w, float h)
    {InitP(x,y);W=w;H=h;}//调用基类公有成员函数
    float GetH ( ) {return H;}
    float GetW ( ) {return W;}
private:   //新增私有数据成员
    float W,H;
};
```

```
#include<iostream>
using namespace std;
int main ( )
{ Rectangle rect;
  rect.InitR(2,3,20,10);
  //通过派生类对象访问基类公有成员
  rect.Move(3,2);
  cout<<rect.GetX ( ) <<','
    <<rect.GetY ( ) <<','
    <<rect.GetH ( ) <<','
    <<rect.GetW ( ) <<endl;
  return 0;
}
```

私有继承(private)

- 基类的**public**和**protected**成员都以**private**身份出现在派生类中，但基类的**private**成员不可访问。
- 派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但不能访问基类的**private**成员。
- 通过派生类的对象不能访问基类中的任何成员。

例2 私有继承举例

```
class Rectangle: private Point           //派生类声明
{public:           //新增外部接口
    void InitR(float x, float y, float w, float h)
    {InitP(x,y);W=w;H=h;} //访问基类公有成员
    void Move(float xOff, float yOff)
    {Point::Move(xOff,yOff);}
    float GetX ( ) {return Point::GetX ( ) ;}
    float GetY ( ) {return Point::GetY ( ) ;}
    float GetH ( ) {return H;}
    float GetW ( ) {return W;}
private:           //新增私有数据
    float W,H;
};
```

```
#include<iostream>
using namespace std;
int main ( )
int main ( )
{ //通过派生类对象只能访问本类成员
    Rectangle rect;
    rect.InitR(2,3,20,10);
    rect.Move(3,2);
    cout<<rect.GetX ( ) <<','
        <<rect.GetY ( ) <<','
        <<rect.GetH ( ) <<','
        <<rect.GetW ( ) <<endl;
    return 0;
}
```

保护继承(protected)

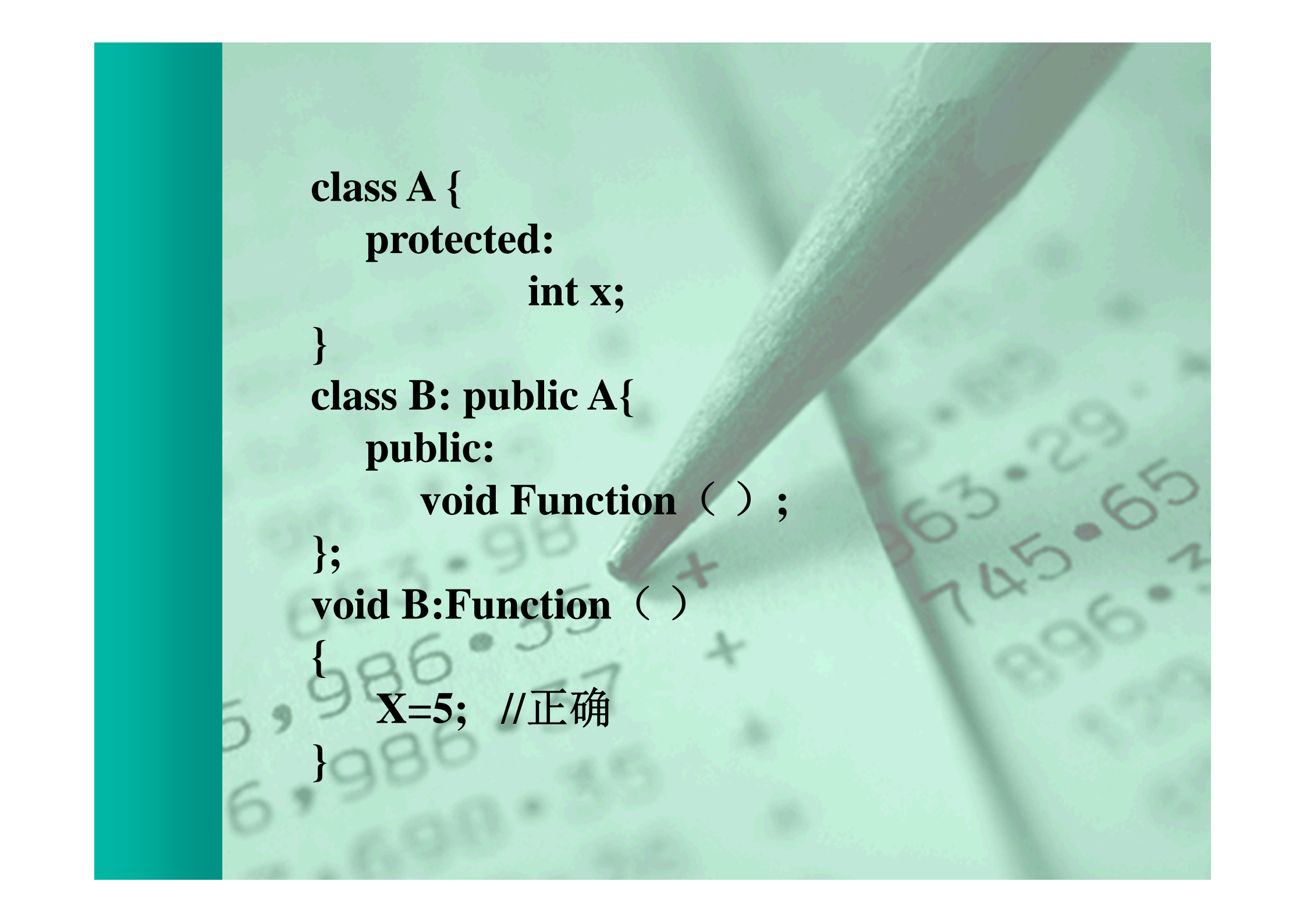
- 基类的**public**和**protected**成员都以**protected**身份出现在派生类中，但基类的**private**成员不可访问。
- 派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但不能访问基类的**private**成员。
- 通过派生类的对象不能访问基类中的任何成员

protected 成员的特点与作用

- 对建立其所在类对象的模块来说（水平访问时），它与 **private** 成员的性质相同。
- 对于其派生类来说（垂直访问时），它与 **public** 成员的性质相同。
- 既实现了数据隐藏，又方便继承，实现代码重用。

例3 protected 成员举例

```
class A {  
    protected:  
        int x;  
}  
  
int main ( )  
{  
    A a;  
    a.X=5; //错误  
}
```

The background of the slide features a close-up, slightly blurred image of a pencil resting on a calculator. The calculator's display shows several numbers, including 963.29, 745.65, and 896.3. The pencil is positioned diagonally across the frame, pointing towards the bottom right. The overall color scheme is a muted teal or green, which is also reflected in a solid vertical bar on the left side of the slide.

```
class A {  
    protected:  
        int x;  
}  
class B: public A{  
    public:  
        void Function ( ) ;  
};  
void B:Function ( )  
{  
    X=5; //正确  
}
```

公有派生、私有派生与保护派生（续）

| 派生方式 | 基类中的访问权限 | 基类成员在派生类中的访问权限 | 外部函数能否使用 |
|------------------|------------------|------------------|----------|
| public | public | public | 可以访问 |
| | protected | protected | 不可以访问 |
| | private | 不可以访问 | 不可以访问 |
| private | public | private | 不可以访问 |
| | protected | private | 不可以访问 |
| | private | 不可以访问 | 不可以访问 |
| protected | public | protected | 不可以访问 |
| | protected | protected | 不可以访问 |
| | private | 不可以访问 | 不可以访问 |

基类与派生类的构造函数

- 基类的构造函数不被继承，需要在派生类中自行声明。
- 派生类的对象时，将自动执行基类与派生类的构造函数
 - ◆ 基类的构造函数在派生类构造函数之前执行
- 传递参数
 - ◆ 可以通过派生类构造函数显式调用基类的构造函数，从而为基类构造函数传递参数

单一继承时的构造函数

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

{

本类成员初始化赋值语句;

};

基类与派生类的构造函数(续1)

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

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

基类与派生类的构造函数(续2)

```
class A {  
private:  
    int va;  
public:  
    A(int i)  
    {  
        va = i;  
    }  
};
```

```
class B :public A {  
private:  
    int vb;  
public:  
    B(int j, int i):A(i)  
    {  
        vb = j;  
    }  
};
```

多继承时的构造函数

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

{

 本类成员初始化赋值语句;

};

多继承且使用组合时的构造函数

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

{

 本类成员初始化赋值语句;

};

基类与派生类的析构函数

■ 执行顺序

- ◆ 调用析构函数的过程与构造函数相反
- ◆ 首先执行派生类的析构函数，然后执行基类的析构函数

基类与派生类的名字冲突

同名覆盖原则

与基类中有相同成员时：

- 若未强行指名，则通过派生类对象使用的是派生类中的同名成员。
- 如要通过派生类对象访问基类中被覆盖的同名成员，应使用基类名限定。

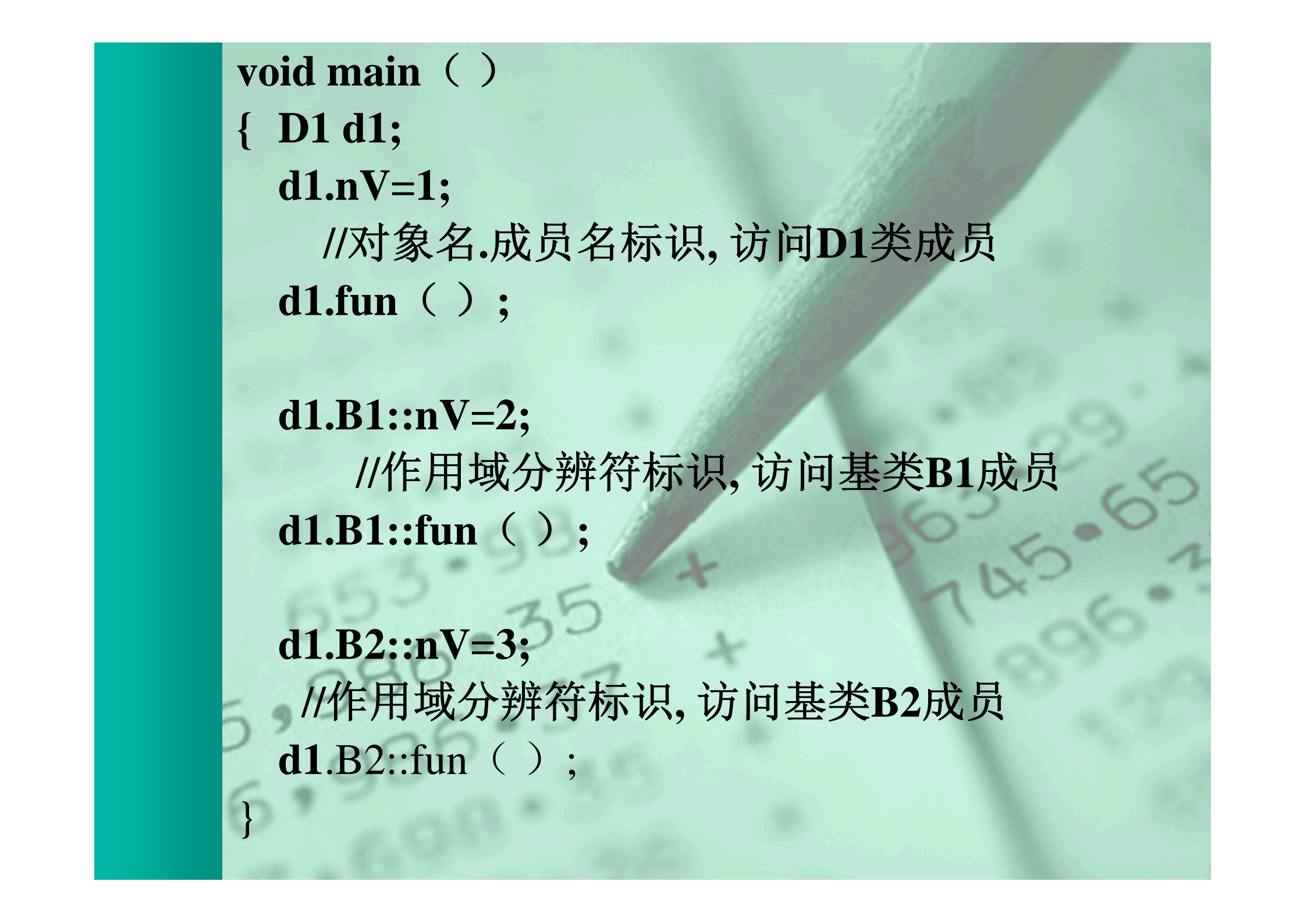
例4 多继承同名覆盖举例

```
#include <iostream.h>

class B1    //声明基类B1
{
public:    //外部接口
    int nV;
    void fun ( ) {cout<<"Member of
B1"<<endl;}
};
```



```
class B2    //声明基类B2
{ public:   //外部接口
    int nV;
    void fun ( ) {cout<<"Member of B2"<<endl;}
};
class D1: public B1, public B2
{ public:
    int nV;    //同名数据成员
    void fun ( ) {cout<<"Member of D1"<<endl;}
    //同名函数成员
};
```

The background of the slide features a light green, semi-transparent image of a calculator and a pen. The calculator is positioned diagonally, with its screen and buttons visible. A pen is also visible, resting on the calculator. The overall aesthetic is clean and professional, suitable for a technical presentation.

```
void main ( )  
{ D1 d1;  
  d1.nV=1;  
  //对象名.成员名标识, 访问D1类成员  
  d1.fun ( ) ;  
  
  d1.B1::nV=2;  
  //作用域分辨符标识, 访问基类B1成员  
  d1.B1::fun ( ) ;  
  
  d1.B2::nV=3;  
  //作用域分辨符标识, 访问基类B2成员  
  d1.B2::fun ( ) ;  
}
```

基类与派生类的名字冲突（续）

```
class A {  
public:  
    int get () { return 1;}  
};  
class B:public A {  
public:  
    int get () { return 2;}  
};
```

```
void main()  
{  
    A a;  
    B b;  
    int i = a.get();  
    int j = b.get();  
    int k = b.A::get();  
}
```

作用域分辨操作符

二义性问题

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

二义性问题举例

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

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

如果声明: C c1;
则 c1.f () ; 具有二义性
而 c1.g () ; 无二义性 (同名覆盖)

二义性的解决方法

- 解决方法一：用类名来限定

c1.A::f () 或 **c1.B::f ()**

- 解决方法二：同名覆盖

在**C** 中声明一个同名成员函数**f ()** ,

f () 再根据需要调用 **A::f ()** 或

B::f ()

二义性问题举例

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

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



下面的访问是二义性的:

C c;

c.b

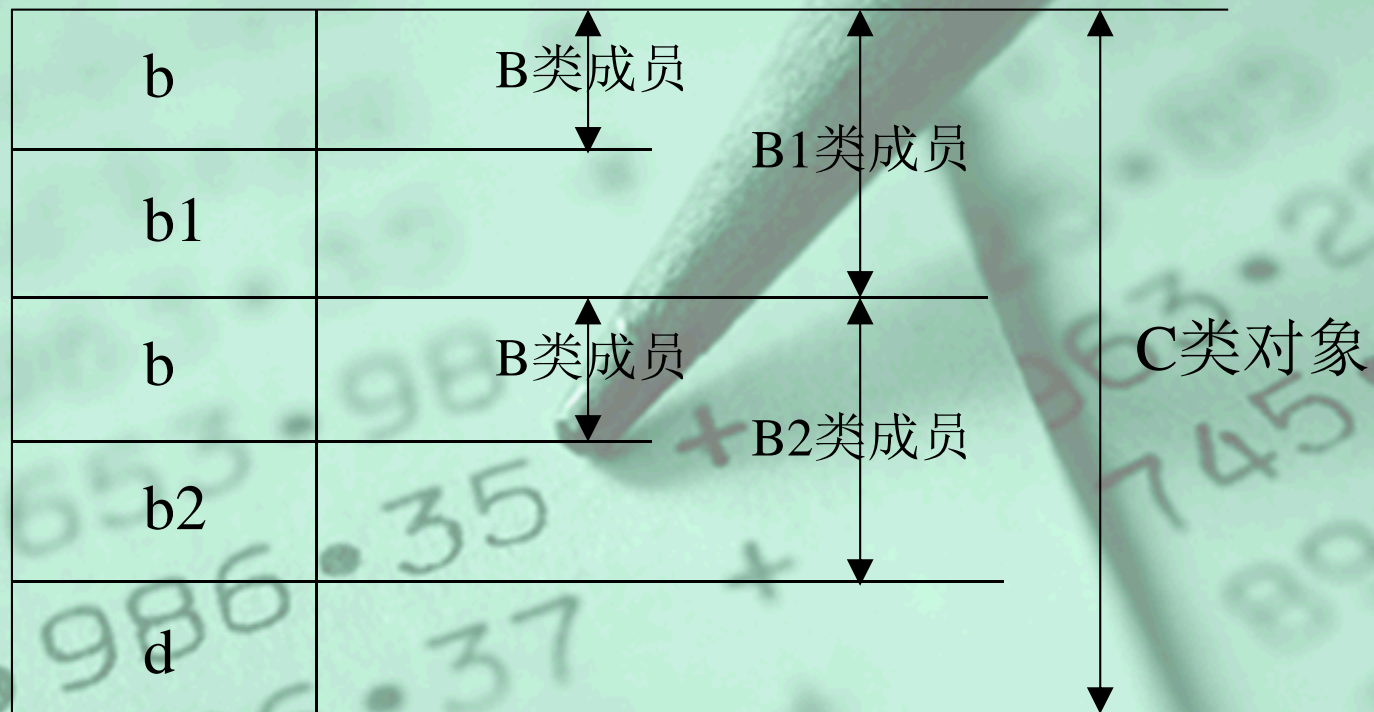
c.B::b

下面是正确的:

c.B1::b

c.B2::b

派生类C的对象的存储结构示意图:



虚拟继承

```
class A {  
public:  
    int value;  
};  
class B :public A {  
.....  
};  
class C:public A {  
.....  
}
```

```
class D:public B,public C {  
public:  
    int get() { return value};  
};
```

这个value值从何而来？

虚拟继承（续）

```
class A {  
public:  
    int value;  
};
```

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

```
class C:public virtual A {  
.....  
}
```

```
class D:public B,public C {  
public:  
    int get() { return value};  
};
```

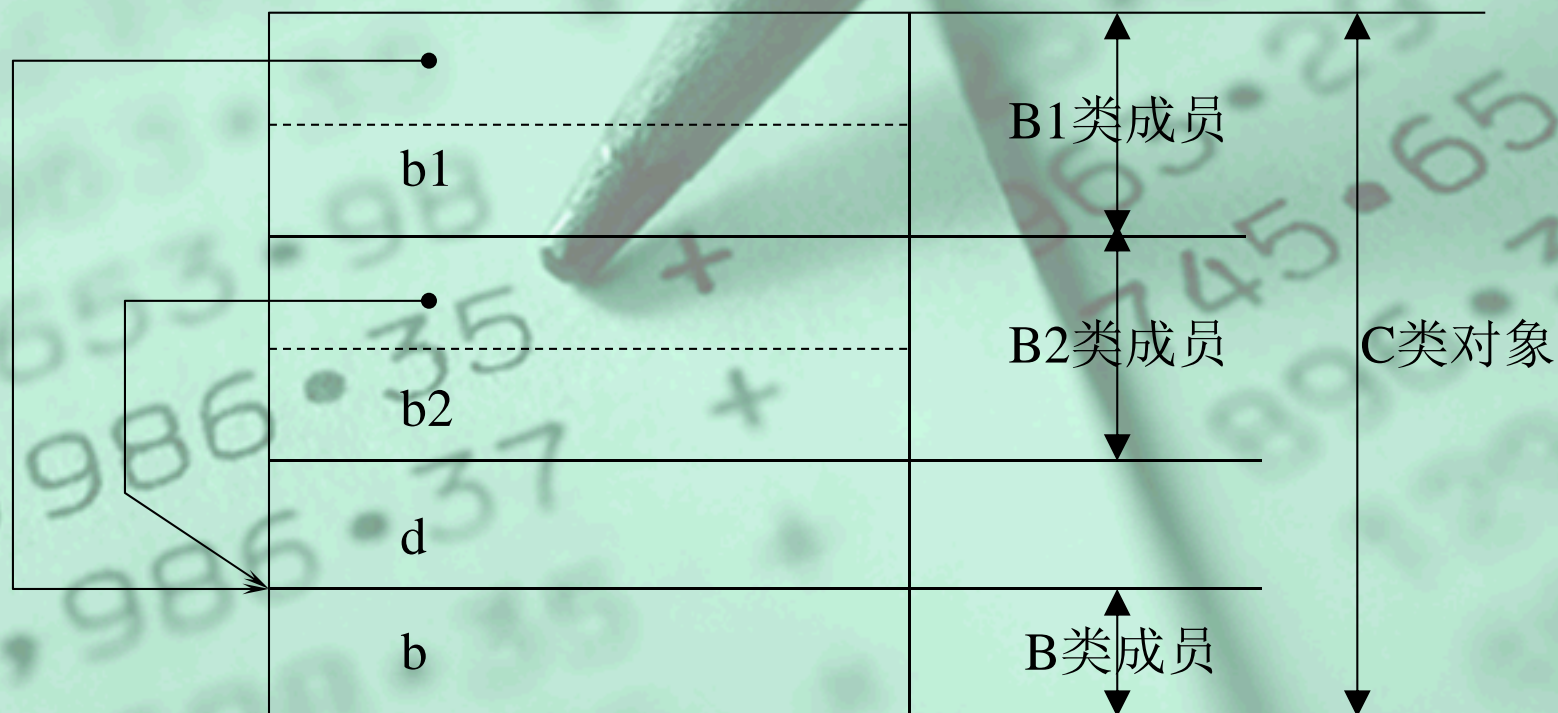
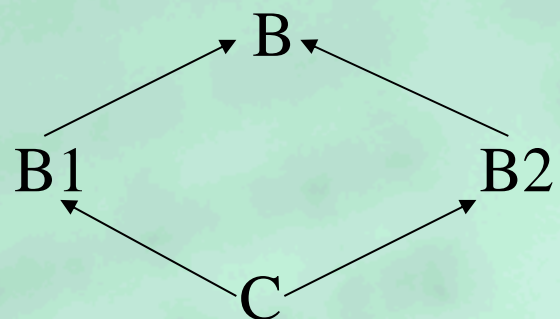
虚基类举例

```
class B{ private: int b;};  
class B1 : virtual public B { private: int b1;};  
class B2 : virtual public B { private: int b2;};  
class C : public B1, public B2{ private: float d;}
```

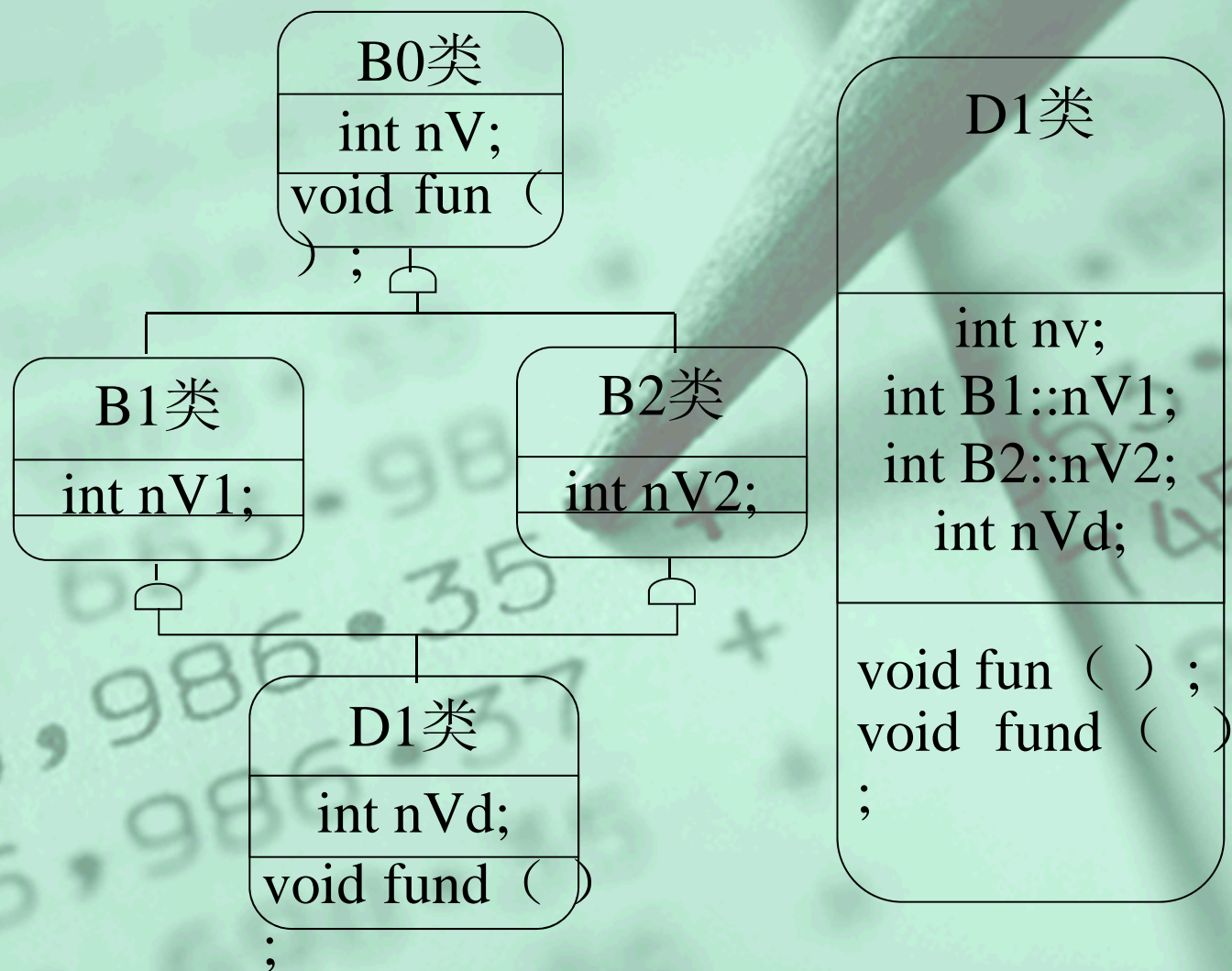
下面的访问是正确的:

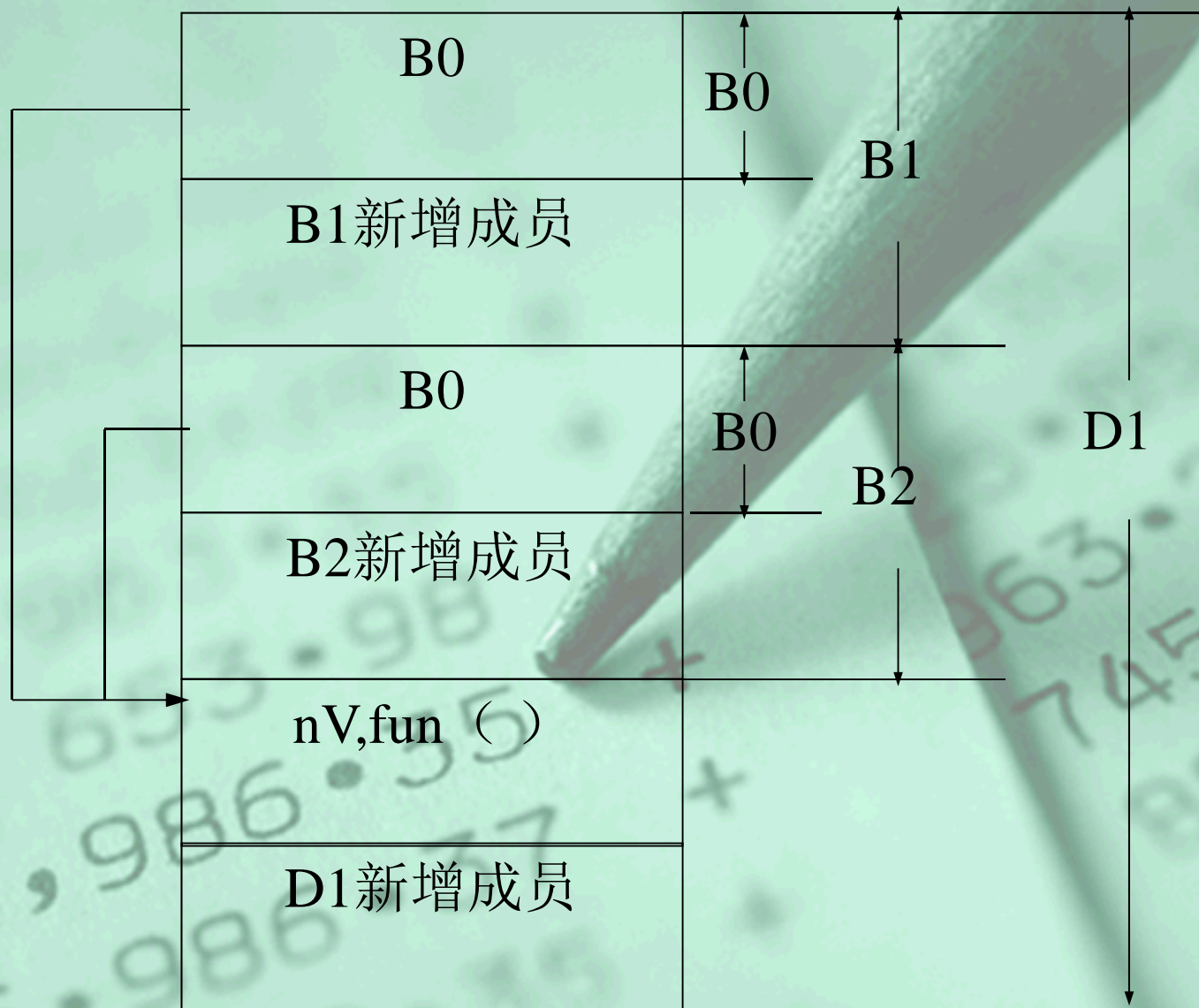
```
C cobj;  
cobj.b;
```


虚基类的派生类对象存储结构示意图：



例5 虚基类（虚拟继承）举例





```
#include <iostream.h>
class B0      //声明基类B0
{
public:      //外部接口
    int nV;
    void fun ( ) {cout<<"Member of B0"<<endl;}
};
class B1: virtual public B0 //B0为虚基类，派生B1类
{
public:      //新增外部接口
    int nV1;
};
```



```
class B2: virtual public B0 //B0为虚基类派生B2类
{ public:    //新增外部接口
    int nV2;
};
class D1: public B1, public B2    //派生类D1声明
{ public:    //新增外部接口
    int nVd;
    void fund ( ) {cout<<"Member of D1"<<endl;}
};
void main ( )    //程序主函数
{ D1 d1; //声明D1类对象d1
  d1.nV=2; //使用直接基类
  d1.fun ( ) ;
}
```

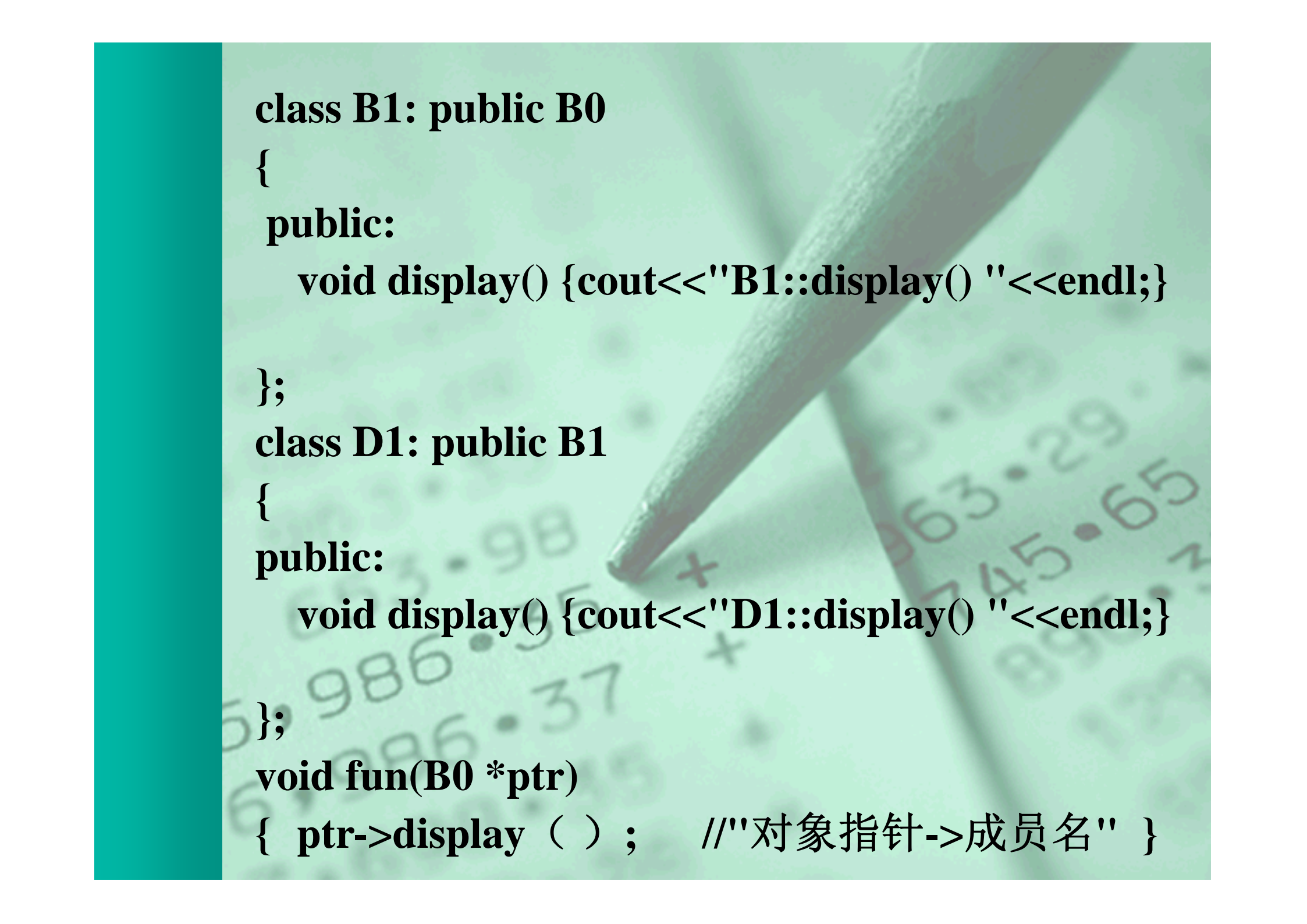
赋值兼容原则

- 一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：
 - ◆ 派生类的对象可以被赋值给基类对象。
 - ◆ 派生类的对象可以初始化基类的引用。
 - ◆ 指向基类的指针也可以指向派生类。

例6 赋值兼容规则举例

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

```
class B0    //基类B0声明
{ public:
    void display ( ) {cout<<"B0::display ( )
    "<<endl;}    //公有成员函数
};
```



```
class B1: public B0  
{  
    public:  
        void display() {cout<<"B1::display() "<<endl;}  
};  
class D1: public B1  
{  
    public:  
        void display() {cout<<"D1::display() "<<endl;}  
};  
void fun(B0 *ptr)  
{ ptr->display ( ) ;    //"对象指针->成员名" }
```



```
void main ( )    //主函数
{ B0 b0;    //声明B0类对象
  B1 b1;    //声明B1类对象
  D1 d1;    //声明D1类对象
  B0 *p;    //声明B0类指针
  p=&b0;    //B0类指针指向B0类对象
  fun(p);
  p=&b1;    //B0类指针指向B1类对象
  fun(p);
  p=&d1;    //B0类指针指向D1类对象
  fun(p);
}
```