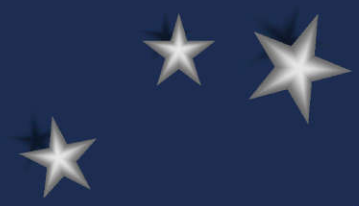
C++语言程序设计

# 第七章　继承与派生

# 本章主要内容

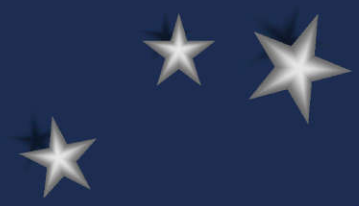
- 类的继承
- 类成员的访问控制
- 单继承与多继承
- 派生类的构造、析构函数
- 类成员的标识与访问



# 类的继承与派生

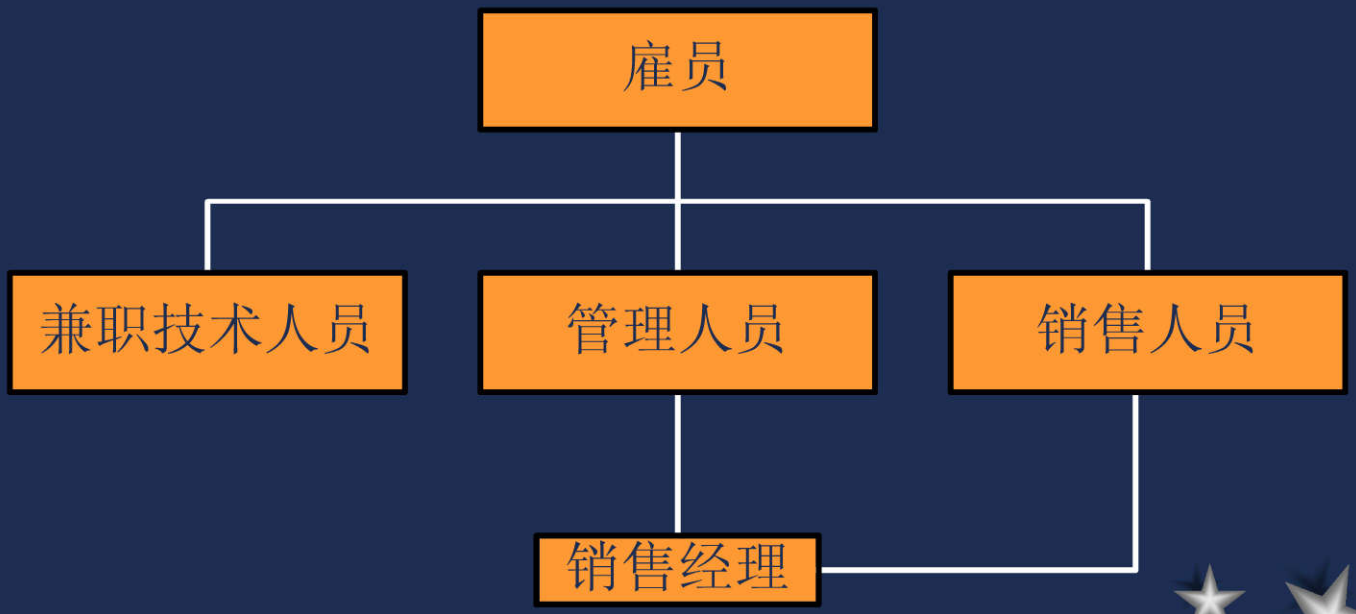


- 保持已有类的特性而构造新类的过程称为继承。
- 在已有类的基础上新增自己的特性而产生新类的过程称为派生。
- 被继承的已有类称为基类（或父类）。
- 派生出的新类称为派生类。



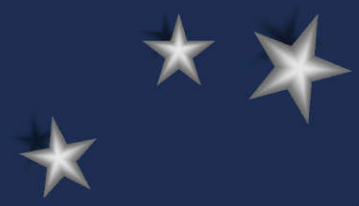
# 继承与派生问题举例

类的继承与派生



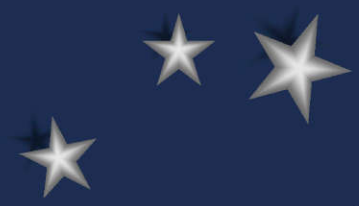
# 继承与派生的目的

- 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。



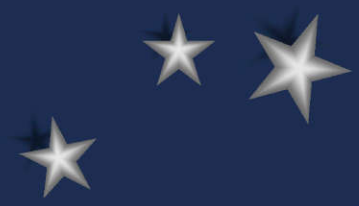
# 派生类的声明

```
class 派生类名：继承方式 基类名1
{
    成员声明;
}
```



# 继承方式

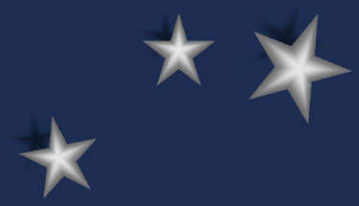
- 不同继承方式的影响主要体现在：
  - 派生类成员对基类成员的访问权限
  - 通过派生类对象对基类成员的访问权限
- 三种继承方式
  - 公有继承
  - 私有继承
  - 保护继承



# 公有继承(public)

类成员的访问控制

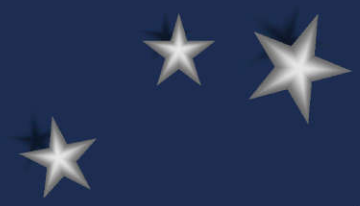
- 基类的public和protected成员的访问属性在派生类中保持不变，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象只能访问基类的public成员。



# 例7-1 公有继承举例

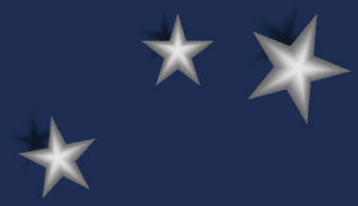
类成员的访问控制

```
class Point
{public:
    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:
    float X,Y;
};
```



类成员的访问控制

```
class Rectangle: public Point
{ public:
      void InitR(float w, float h)
        {W=w;H=h;}
      float GetH() {return H;}
      float GetW() {return W;}
  private:
      float W,H;
};
```



# 类成员的访问控制

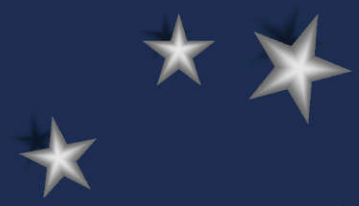
```
#include<iostream>
using namespace std;
void main()
{  Rectangle rect;
   rect.InitP(2,3);
   rect.InitR(20,10);
   rect.Move(3,2);
   cout<<rect.GetX()<<','<<rect.GetY()<<','
       <<rect.GetH()<<','<<rect.GetW();
}
```

继承的方法

# 私有继承(private)

类成员的访问控制

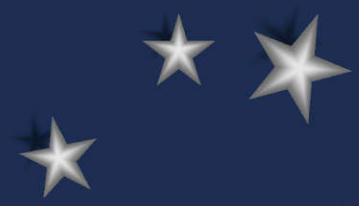
- 基类的public和protected成员都以private身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员。



# 例7-2 私有继承举例

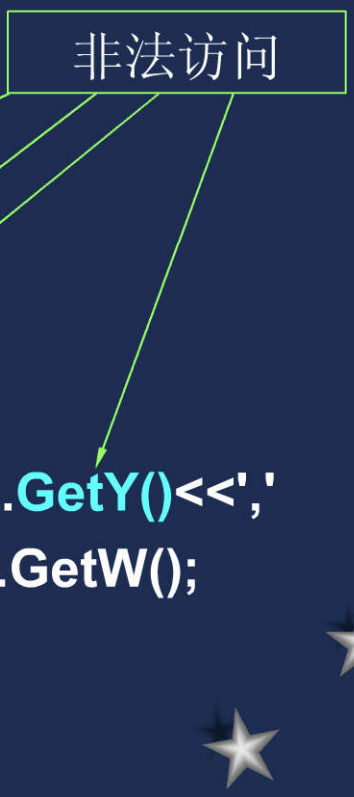
类成员的访问控制

```
class Rectangle: private Point
{ public:
      void InitR(float w, float h)
        {W=w;H=h;}
      float GetH() {return H;}
      float GetW() {return W;}
  private:
      float W,H;
};
```



# 类成员的访问控制

```
#include<iostream>
using namespace std;
void main()
{  Rectangle rect;
   rect.InitP(2,3);
   rect.InitR(20,10);
   rect.Move(3,2);
   cout<<rect.GetX()<<','<<rect.GetY()<<','
       <<rect.GetH()<<','<<rect.GetW();
}
```



类成员的访问控制

```
class Rectangle: private Point
{ public:
    void InitR(float w, float h)
     { W=w;H=h;}
    void InitP(float xx=0, float yy=0)
     { Point::InitP(xx,yy);}
    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:
    float W,H;
};
```

# 类成员的访问控制

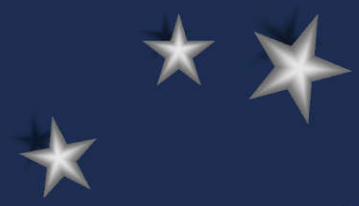
```
#include<iostream>
using namespace std;
void main()
{  Rectangle rect;
   rect.InitP(2,3);
   rect.InitR(20,10);
   rect.Move(3,2);
   cout<<rect.GetX()<<','<<rect.GetY()<<','
         <<rect.GetH()<<','<<rect.GetW();
}
```

访问派生类自身的方法

# 保护继承(protected)

类成员的访问控制

- 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成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员。



## 基类

| 成员访问修饰符 | public | protected | private |
| --- | --- | --- | --- |
| 成员函数间 | √ | √ | √ |
| 对象 | √ | | |

## public继承

| 基类成员访问修饰符 | public | protected | private |
| --- | --- | --- | --- |
| 派生类继承的成员的访问修饰符 | public | protected | private |
| 派生类对继承的成员的访问 | √ | √ | |
| 派生类对象对继承的成员的访问 | √ | | |

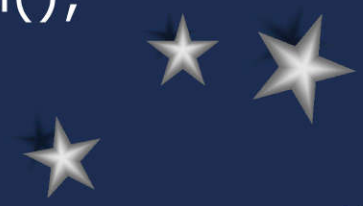
## private继承

| 基类成员访问修饰符 | public | protected | private |
| --- | --- | --- | --- |
| 派生类继承的成员的访问修饰符 | private | private | private |
| 派生类对继承的成员的访问 | √ | √ | |
| 派生类对象对继承的成员的访问 | | | |
| 以派生类作为基类再派生的新类 | | | |

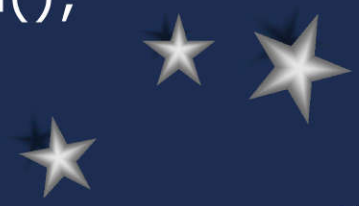
## protected继承

| 基类成员访问修饰符 | public | protected | private |
| --- | --- | --- | --- |
| 派生类继承的成员的访问修饰符 | protected | protected | private |
| 派生类对继承的成员的访问 | √ | √ | |
| 派生类对象对继承的成员的访问 | | | |
| 以派生类作为基类再派生的新类 | √ | √ | |

```
#include<iostream>
using namespace std;
class A
{ protected:
    int x;
};
class B: protected A
{ public:
    void Function();
};
class C: private     B
{ public:
    void Function();
};
void B::Function()
{ x=5;
  cout<<x<<endl;
}
void C::Function()
{ x=15;
  cout<<x<<endl;
}
void main()
{ B b;
  b.Function();
  C c;
  c.Function();
}
```



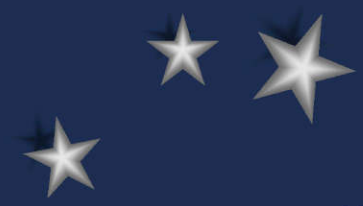
```
#include<iostream>
using namespace std;
class A
{ protected:
    int x;
};
class B: private    A
{ public:
    void Function();
};
class C: protected  B
{ public:
    void Function();
};
void B::Function()
{ x=5;
  cout<<x<<endl;
}
void C::Function()
{ x=15;
  cout<<x<<endl;
}
void main()
{ B b;
  b.Function();
  C c;
  c.Function();
}
```



# 类型兼容规则

类型兼容

- 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：
  - 派生类的对象可以赋值给基类对象。
  - 派生类的对象可以初始化基类的引用。
  - 派生类对象的地址可以赋值给指向基类的指针。
- 替代之后，派生类对象就可以作为基类的对象使用，但只能使用从基类继承的成员。派生类对象发挥基类对象的作用。



类型兼容

```
#include <iostream>
using namespace std;
class B0
{public:
   void display()
    {cout<<"B0::display()"<<endl;}
};
class B1: public B0
{public:
    void display()
     {cout<<"B1::display()"<<endl;}
    void output()
     {cout<<"class B1  B1::display()"<<endl;}
};
```

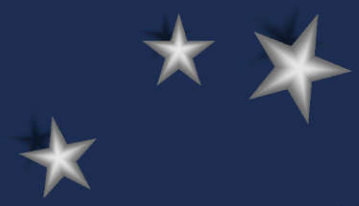
类型兼容

```
void main()
{ B0 b0;
  B1 b1;
  b0.display();      B0::display()
  b0=b1;
  b0.display();      B0::display()
  b1.display();      B1::display()
  b0.output();       编译错误
  b1.output();       class B1  B1::display()
}
```

P219 例7-4

# 基类与派生类的对应关系

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

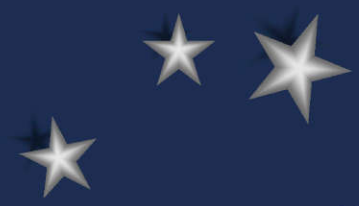


# 多继承时派生类的声明



```
class 派生类名：继承方式1 基类名1，继承方式2 基类名2，...
{
    成员声明;
}
```

注意：每一个“继承方式”，只用于限制其后的基类。



# 单继承与多继承

```
class A
{  public:
       void setA(int);
       void showA();
    private:
       int a;
};
class B
{  public:
       void setB(int);
       void showB();
   private:
       int b;
};
```

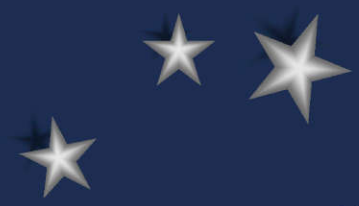
```
class C : public A, private B
{ public:
       void setC(int, int, int);
       void showC();
   private:
       int c;
};
```

```
void  A::setA(int x)
{   a=x;  }

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

void C::setC(int x, int y, int z)
{ setA(x);
  setB(y);
  c=z;
}
//其他函数实现略
```

```
void main()
{ C obj;
  obj.setA(5);
  obj.showA();
  obj.setC(6,7,9);
  obj.showC();
  obj.setB(6);
  obj.showB();
 }
```

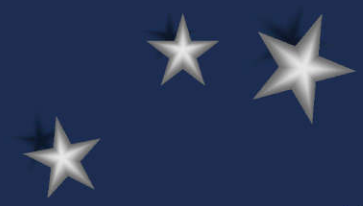
# 继承时的构造函数

- 基类的构造函数不被继承，派生类中需要声明自己的构造函数。
- 声明构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化，自动调用基类构造函数完成。
- 派生类的构造函数需要给基类的构造函数传递参数。



# 继承时的构造函数

- 当基类中有默认形式的构造函数或带默认形参值的构造函数时，派生类构造函数可以不向基类构造函数传递参数。
- 若基类中未声明构造函数，派生类中也可以不声明，全采用默认形式构造函数。
- 当基类声明有带形参的构造函数时，派生类也应声明带形参的构造函数，并将参数传递给基类构造函数。



# 单一继承时的构造函数

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

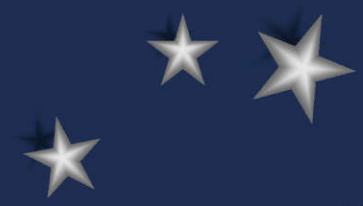
派生类C的构造函数有以下几种实现方式：

**C::C(){}或C::C(int i,int j){...}**

表示先调用B类默认的构造函数或带默认形参值的构造函数初始化B类数据成员。

**C::C(int i,int j):B(i){...}**

表示先调用B类带参数的构造函数初始化B类数据成员。



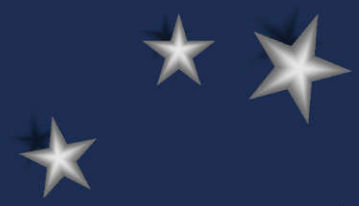
```
#include<iostream>
using namespace std;
class B
{public:
   B();
   B(int i);
    ~B();
 private:
   int b;
};
class C:public B
{public:
   C();
   C(int i,int j);
   ~C();
 private:
   int c;
};
```

```
void  main()
{C obj(3,5);
 }
```

```
B::B()
{b=0;
  cout<<"B's default constructor called."<<endl;}
B::B(int i)
{b=i;
  cout<<"B's constructor called." <<b<<endl;}
B::~B()
{cout<<"B's destructor called."<<endl;  }
C::C()
{c=0;
  cout<<"C's default constructor called."<<endl;}
C::C(int i,int j):B(i)
{c=j;
  cout<<"C's constructor called."<<c<<endl;}
C::~C()
{cout<<"C's destructor called."<<endl; }
```

# 多继承时的构造函数

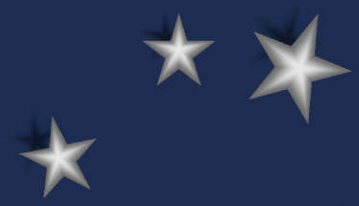
派生类名::派生类名(基类1形参，基类2形参，...基类n形参，本类形参)：
基类名1(参数), 基类名2(参数), ... , 基类名n(参数),
对象数据成员的初始化
{
　　本类成员初始化赋值语句；
};



# 构造函数的调用顺序

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

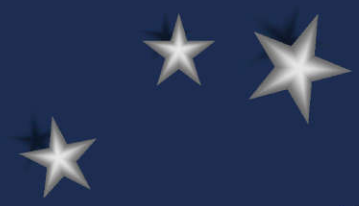
P222 例7-5



# 拷贝构造函数

- 若建立派生类对象时调用默认拷贝构造函数，则编译器将自动调用基类的默认拷贝构造函数。
- 若编写派生类的拷贝构造函数，则需要为基类相应的拷贝构造函数传递参数。例如：

```
C::C(C &c1):B(c1)
{…}
```



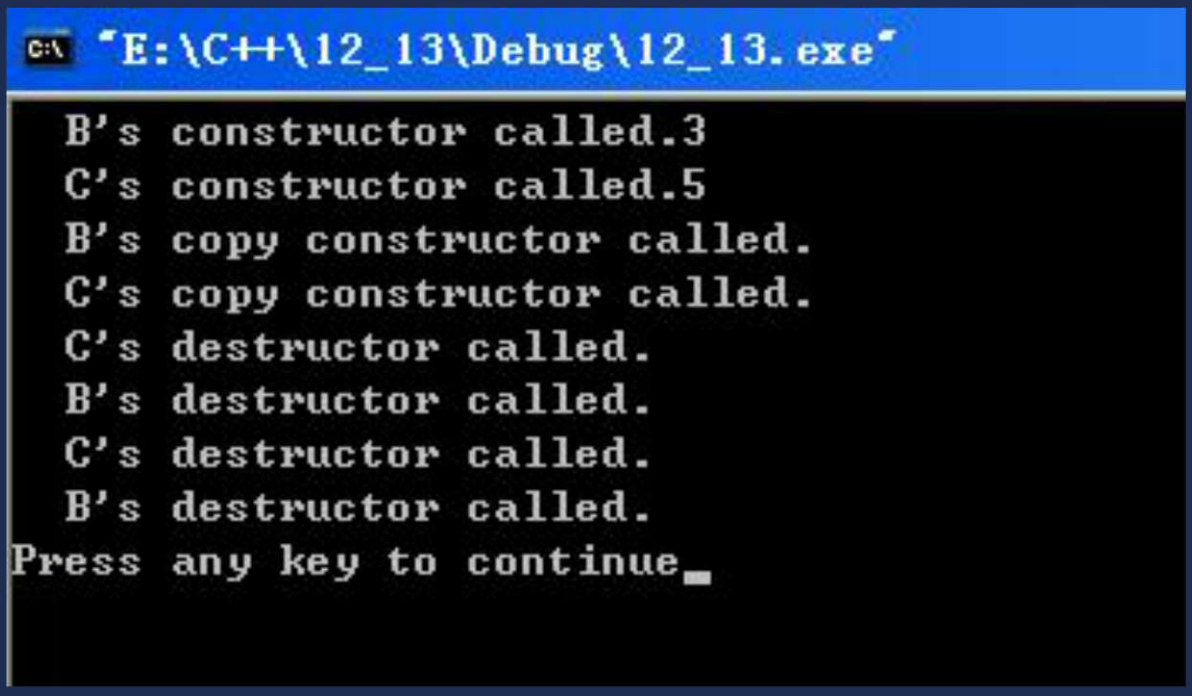
派生类的构造、析构函数

```
#include<iostream>
using namespace std;
class B
{public:
    B(int i);
    B(B&);
    ~B();
  private:
    int b;
};
class C:public B
{public:
    C(int i,int j);
    C(C&);
    ~C();
  private:
    int c;
};
```

## 派生类的构造、析构函数

```
B::B(B& b1)
{b=0;
  cout<<"B's copy constructor called."<<endl;}
B::B(int i)
{b=i;
  cout<<"B's constructor called." <<b<<endl;}
B::~B()
{cout<<"B's destructor called."<<endl;  }
C::C(C& c1):B(c1)
{c=0;
  cout<<"C's copy constructor called."<<endl;}
C::C(int i,int j):B(i)
{c=j;
  cout<<"C's constructor called."<<c<<endl;}
C::~C()
{cout<<"C's destructor called."<<endl; }
void  main()
{C obj(3,5);
  C obj2(obj);}
```

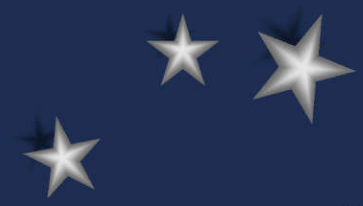
# 派生类的构造、析构函数


"E:\C++\12_13\Debug\12_13.exe"
B's constructor called.3
C's constructor called.5
B's copy constructor called.
C's copy constructor called.
C's destructor called.
B's destructor called.
C's destructor called.
B's destructor called.
Press any key to continue

# 继承时的析构函数

- 析构函数也不被继承，派生类自行声明
- 声明方法与一般（无继承关系时）类的析构函数相同。
- 不需要显式地调用基类的析构函数，系统会自动隐式调用。
- 析构函数的调用次序与构造函数相反。

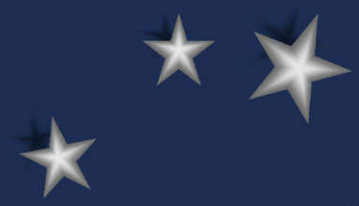
**P224 例7-6**



# 例7-6 派生类析构函数举例

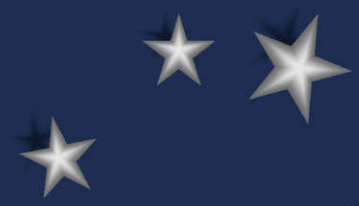
```
#include <iostream>
using namecpace std;
class B1          //基类B1声明
{ public:
    B1(int i)   {cout<<"constructing B1 "<<i<<endl;}
    ~B1() {cout<<"destructing B1 "<<endl;}
};
class B2          //基类B2声明
{public:
    B2(int j)    {cout<<"constructing B2 "<<j<<endl;}
    ~B2() {cout<<"destructing B2 "<<endl;}
};
class B3          //基类B3声明
{public:
    B3(){cout<<"constructing B3 *"<<endl;}
    ~B3() {cout<<"destructing B3 "<<endl;}
};
```

```
class C: public B2, public B1, public B3
{public:
   C(int a, int b, int c, int d):
    B1(a),memberB2(d),memberB1(c),B2(b){}
private:
   B1 memberB1;
   B2 memberB2;
   B3 memberB3;
};
void main()
{  C obj(1,2,3,4);   }
```



# 例7-6 运行结果

```
constructing B2 2
constructing B1 1
constructing B3 *
constructing B1 3
constructing B2 4
constructing B3 *
destructing B3
destructing B2
destructing B1
destructing B3
destructing B1
destructing B2
```



# 同名隐藏规则

当派生类与基类中有相同成员时：

- 若未强行指名，则通过派生类对象使用的是派生类中的同名成员。
- 在没有虚函数的情况下，如果派生类中声明了与基类成员函数同名的新函数，即使函数的参数表不同，从基类继承的同名函数的所有重载形式也都会被隐藏。
- 在没有虚函数的情况下，如果某派生类的多个基类拥有同名的成员，同时，派生类又新增这样的同名成员，派生类成员将隐藏所有基类的同名成员。
- 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7-7 多继承同名隐藏举例

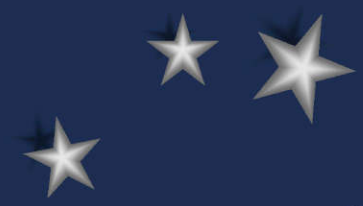
```
#include <iostream>
using namespace std;
class B1
{ public:
    int nV;
    void fun()  {cout<<"Member of B1 "<<nV<<endl;}
};
class B2
{ public:
    int nV;
    void fun()  {cout<<"Member of B2 "<<nV<< endl;}
};
class D1: public B1, public B2
{ public:
    int nV;
    void fun(){cout<<"Member of D1 "<<nV<< endl;}
};
```

# 派生类成员的标识与访问

```
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();
}
```

# 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（参见第8章）或同名隐藏规则来解决。
- 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（一）

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

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

如果声明：C  c1;

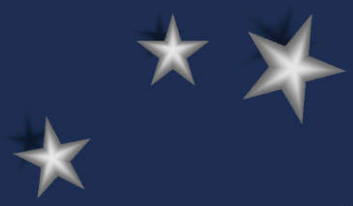
则  c1.f();  具有二义性

而  c1.g();  无二义性

（同名覆盖）

# 二义性的解决方法

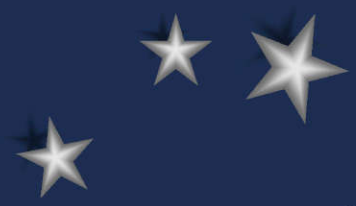
- 解决方法一：用类名来限定
  **c1.A::f()　或　c1.B::f()**
- 解决方法二：同名覆盖
  在C 中声明一个同名成员函数f()，f()
  再根据需要调用 **A::f()　或　B::f()**



# 二义性问题举例（二）

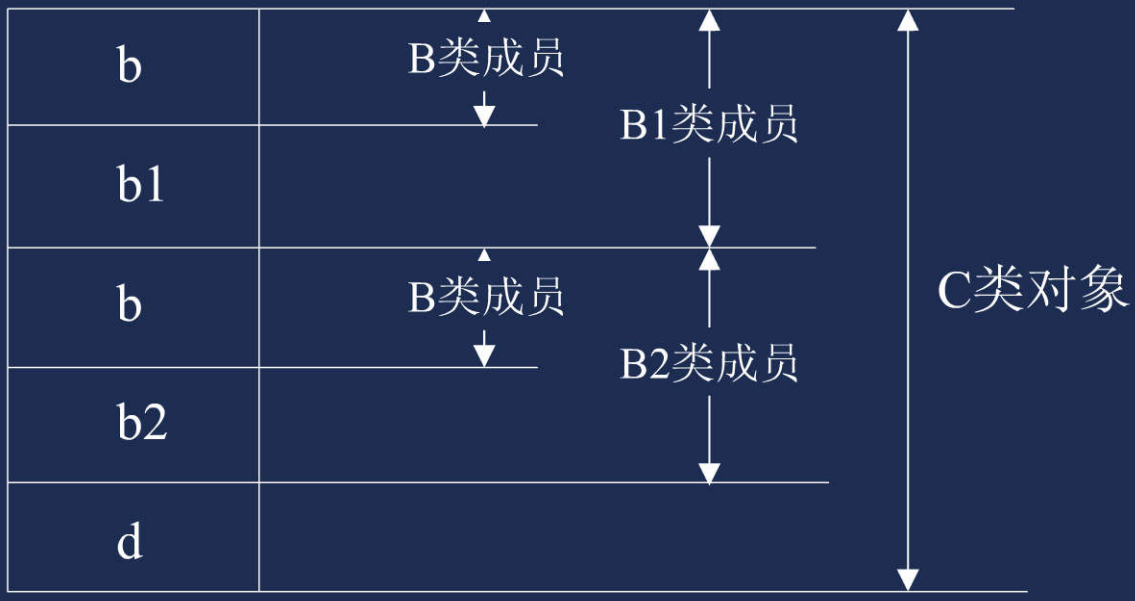
```
class B
{ public:
      int b;
};
class B1 : public B
{ private:
      int b1;
};
class B2 : public B
{ private:
      int b2;
};
```

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



# 派生类成员的标识与访问

## 派生类C的对象的存储结构示意图：




有二义性：

```
C  c;
c.b
c.B::b
```

无二义性：

```
c.B1::b
c.B2::b
```

# 虚基类

虚基类

- 虚基类的引入
  - 用于有共同基类的场合
- 声明
  - 以virtual修饰说明基类
    例：**class B1:virtual public B**
- 作用
  - 主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题。
  - 为最远的派生类提供惟一的基类成员，而不重复产生多次拷贝。
- 注意
  - 在第一级继承时就要将共同基类设计为虚基类。

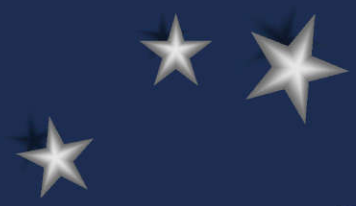
# 虚基类举例

虚基类

```
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;}
```

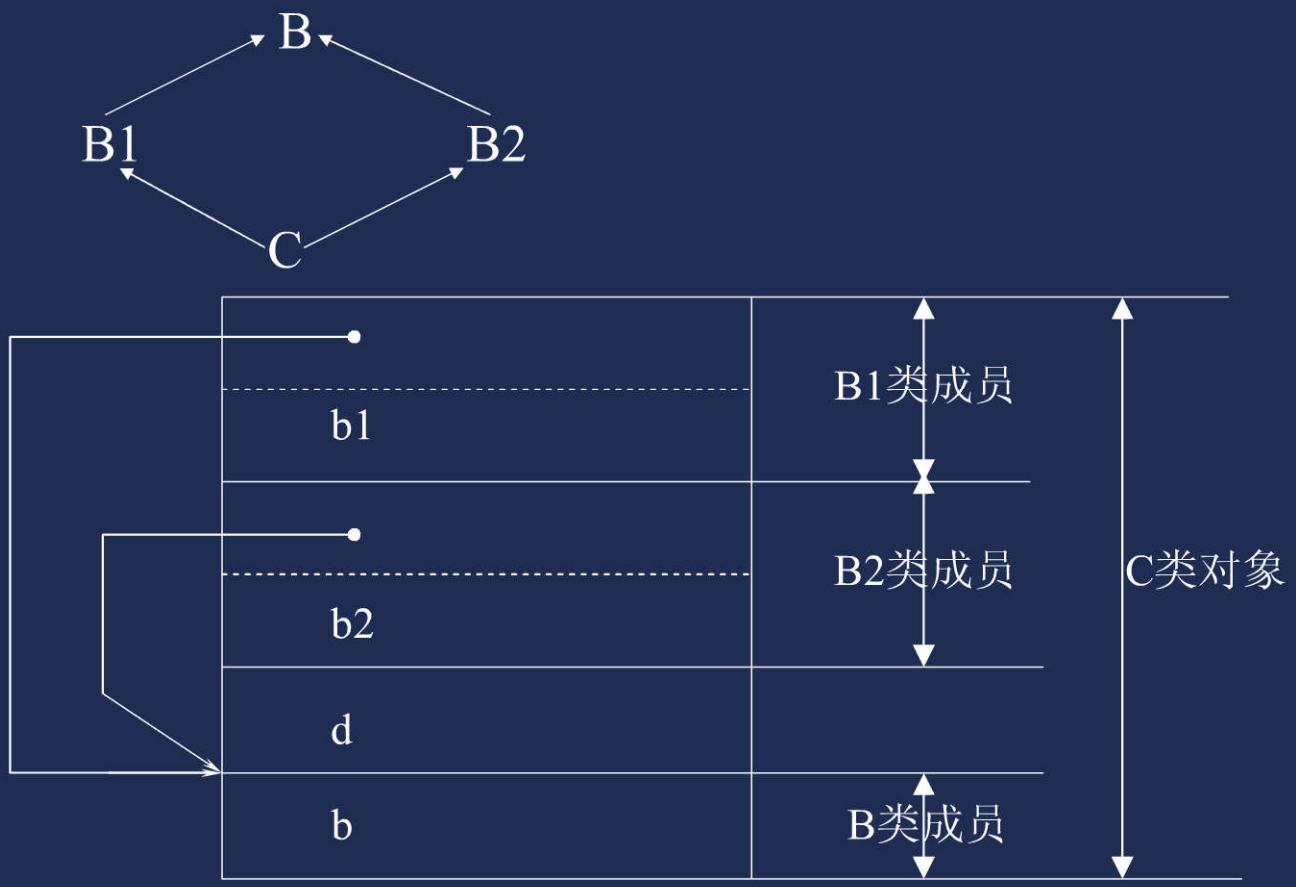
下面的访问是正确的：

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



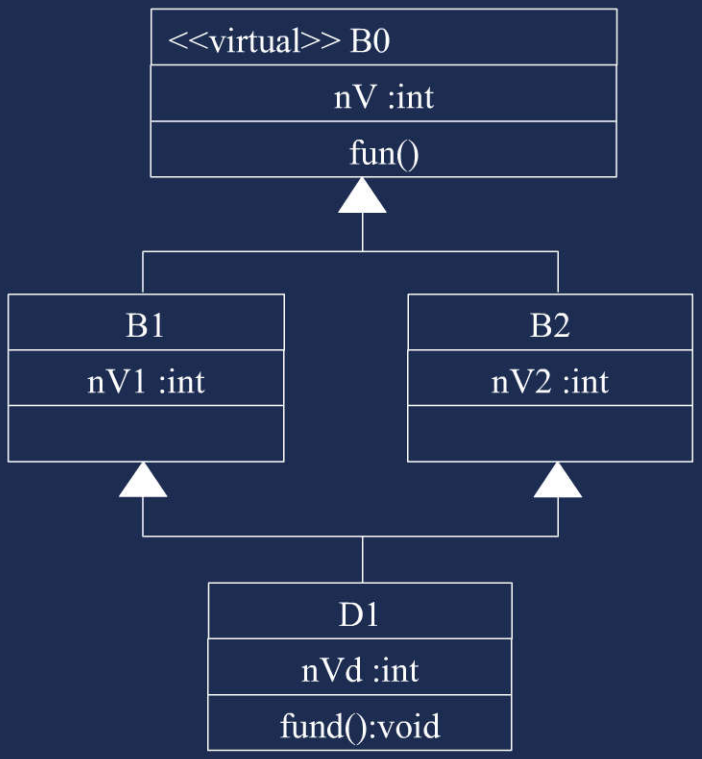
虚基类

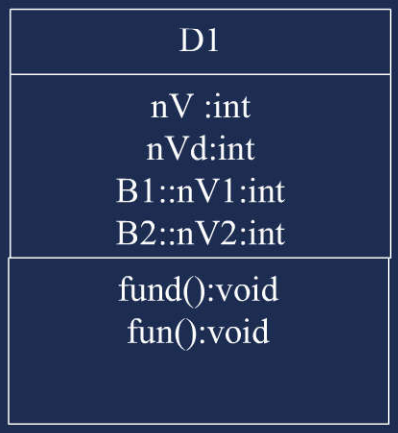
# 虚基类的派生类对象存储结构示意图：



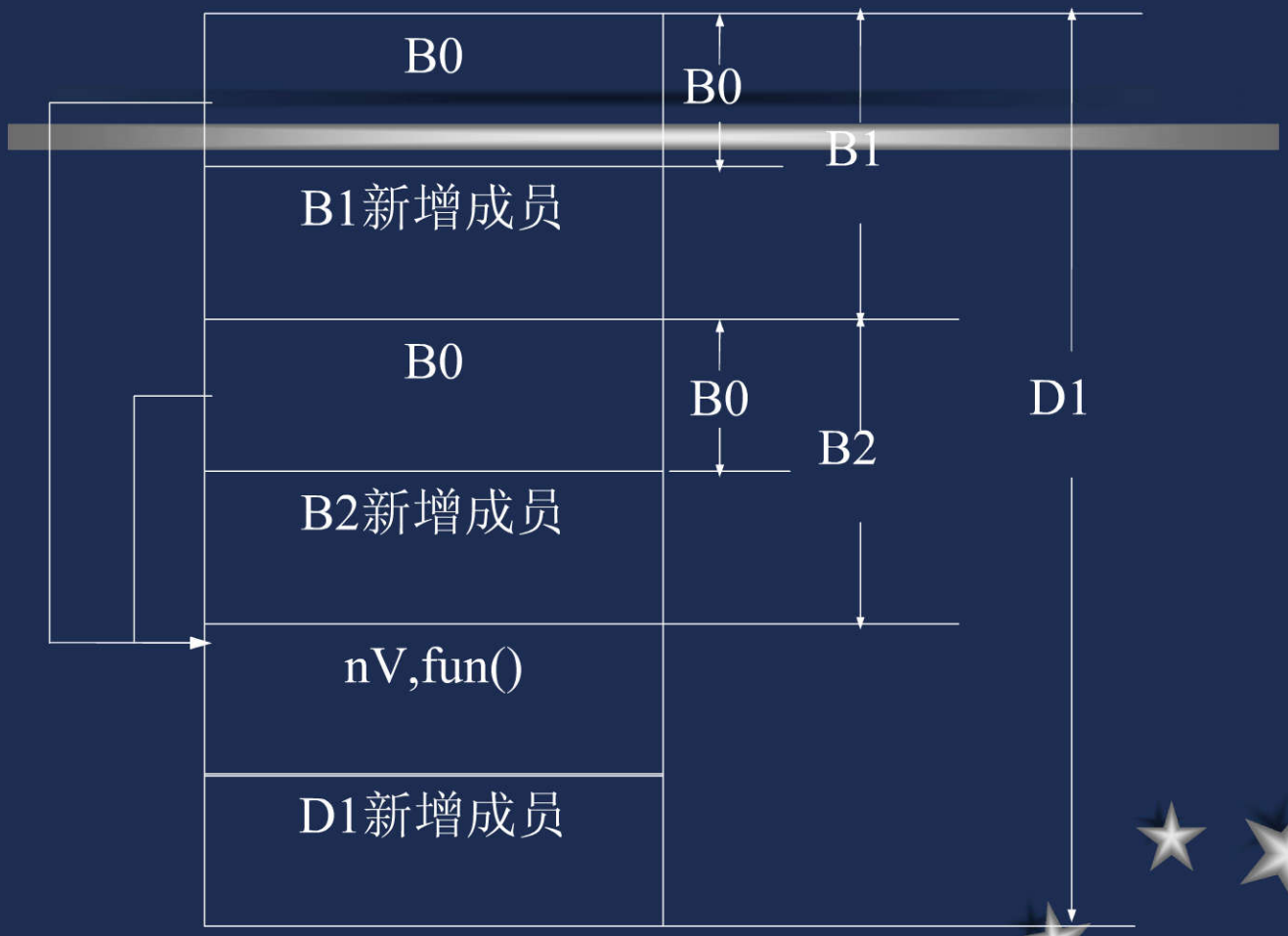
# 例7-8虚基类举例

虚基类



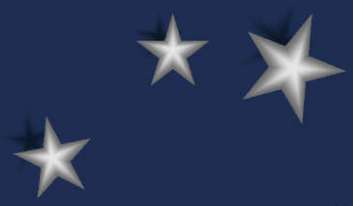



虚基类



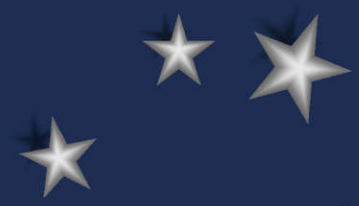
虚基类

```
#include <iostream>
using namespace std;
class B0
{ public:
    int nV;
    void fun(){cout<<"Member of B0"<<endl;}
};
class B1: virtual public B0
{ public:
    int nV1;
};
class B2: virtual public B0
{  public:
     int nV2;
};
```



虚基类

```
class D1: public B1, public B2
{  public:      //新增外部接口
      int nVd;
      void fund(){cout<<"Member of D1"<<endl;}
};
void main()
{   D1 d1;      //声明D1类对象d1
    d1.nV=2;
    d1.fun();
}
```



# 虚基类及其派生类构造函数

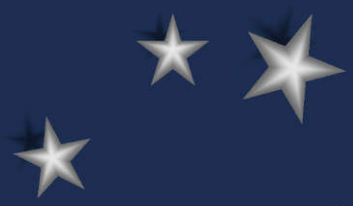
虚基类

- 建立对象时所指定的类称为最(远)派生类。
- 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。
- 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。
- 在建立对象时，只有最远派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略。

虚基类

```
#include <iostream>
using namespace std;
class B0
{ public:
      B0(int n){ nV=n;}
      int nV;
      void fun(){cout<<"Member of B0"<<endl;}
};
class B1: virtual public B0
{  public:
      B1(int a) : B0(a) { }
      int nV1;
};
class B2: virtual public B0
{  public:
      B2(int a) : B0(a) { }
      int nV2;
};
```

```
class D1: public B1, public B2
{
  public:
      D1(int a) : B0(a), B1(a), B2(a){ }
      int nVd;
      void fund(){cout<<"Member of D1"<<endl;}
};
void main()
{
    D1 d1(1);
    d1.nV=2;
    d1.fun();
}
```



# 综合实例

- 例7-10（课后阅读）
- 这个程序有两点不足：

  ①基类的成员函数pay() 的函数体为空，在实现部分仍要写出函数体，显得冗余。

  ②在main()函数中，建立了四个不同类的对象，对它们进行了类似的操作，但是却重复写了四遍类似的语句，程序不够简洁。

# 小结与复习建议

- 主要内容
  - 类的继承、类成员的访问控制、单继承与多继承、派生类的构造和析构函数、类成员的标识与访问
- 达到的目标
  - 理解类的继承关系，学会使用继承关系实现代码的重用。
- 实验任务
  - 实验七