第5章：类之继承性与派生类

计算机系 江志英 jiangzy@mail.buct.edu.cn

本章内容

1 基类和派生类

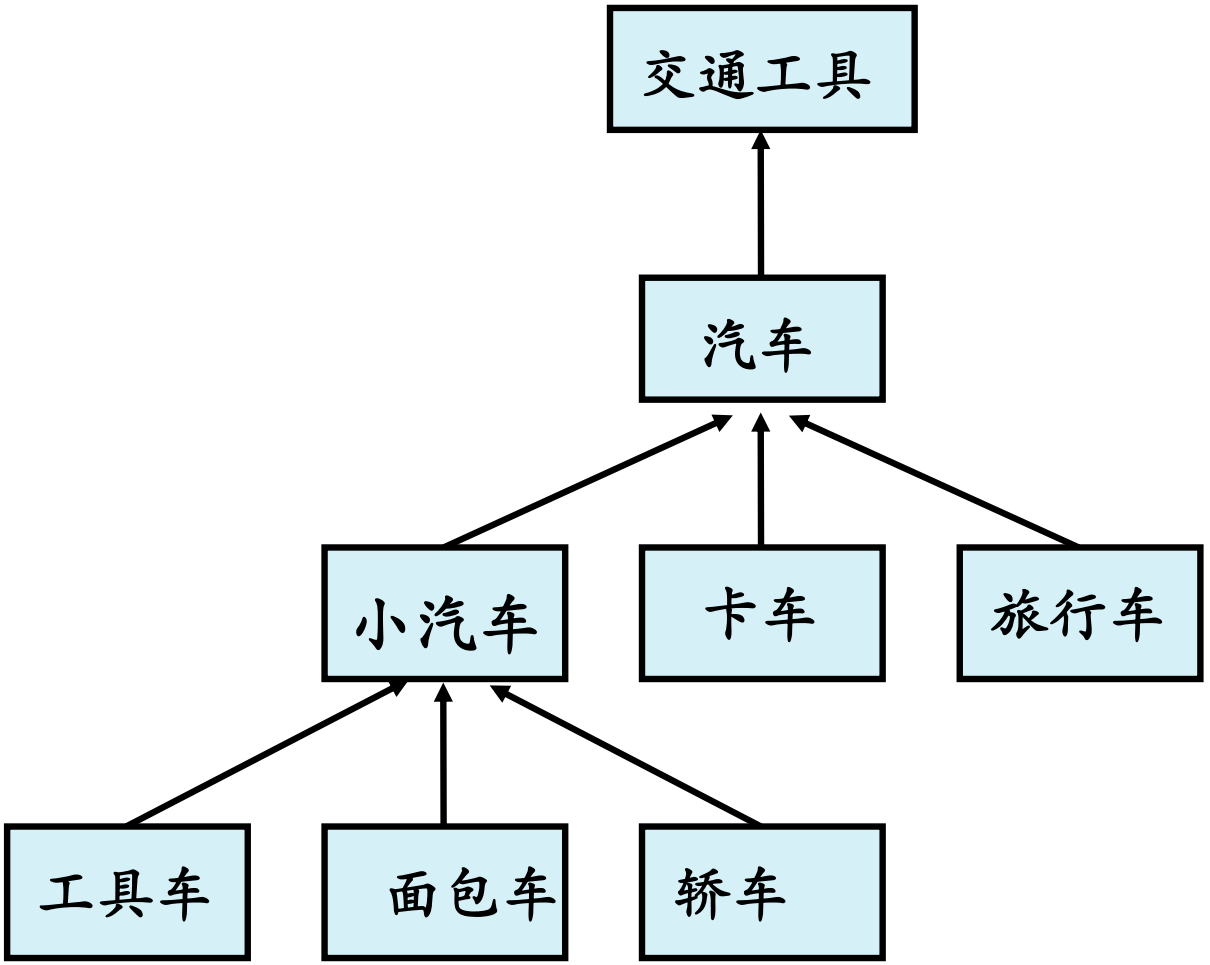
2 单继承

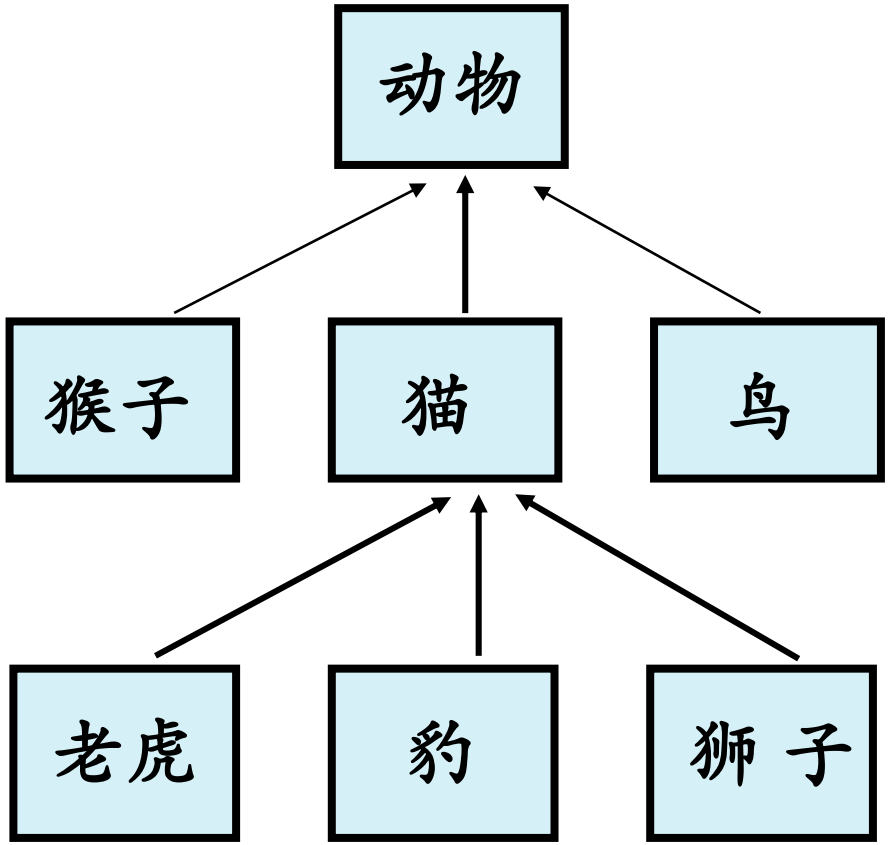
3 多继承

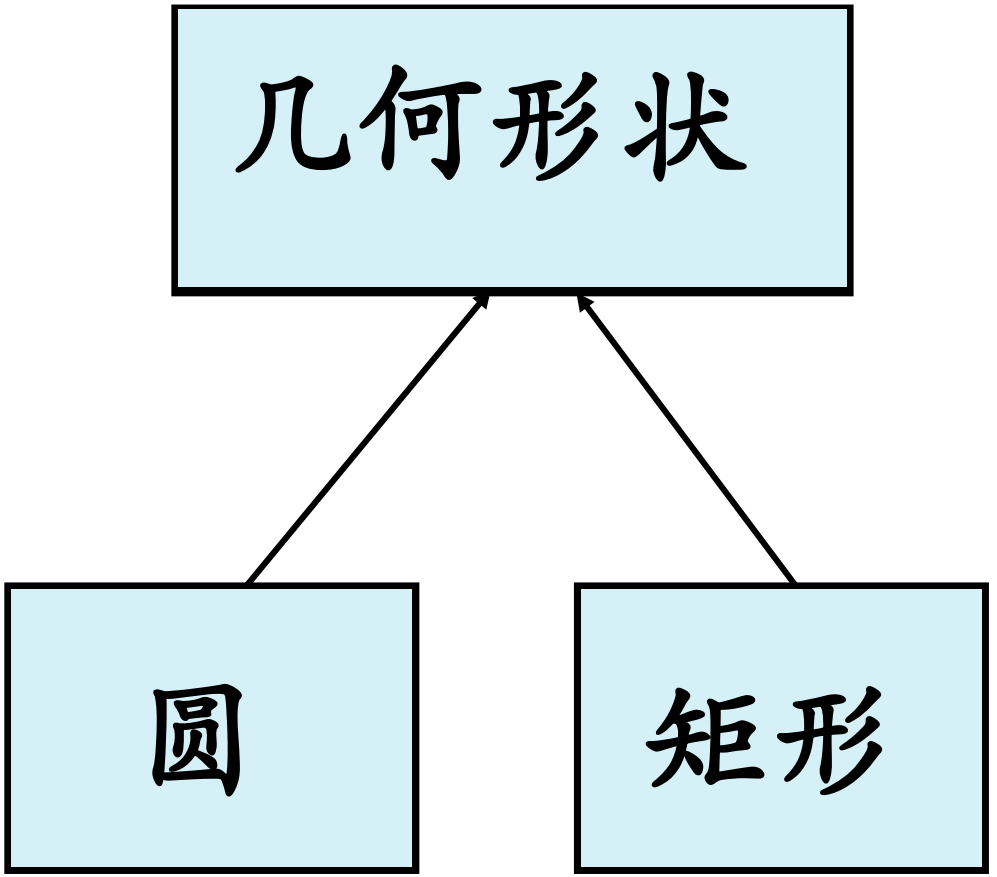
4 虚基类

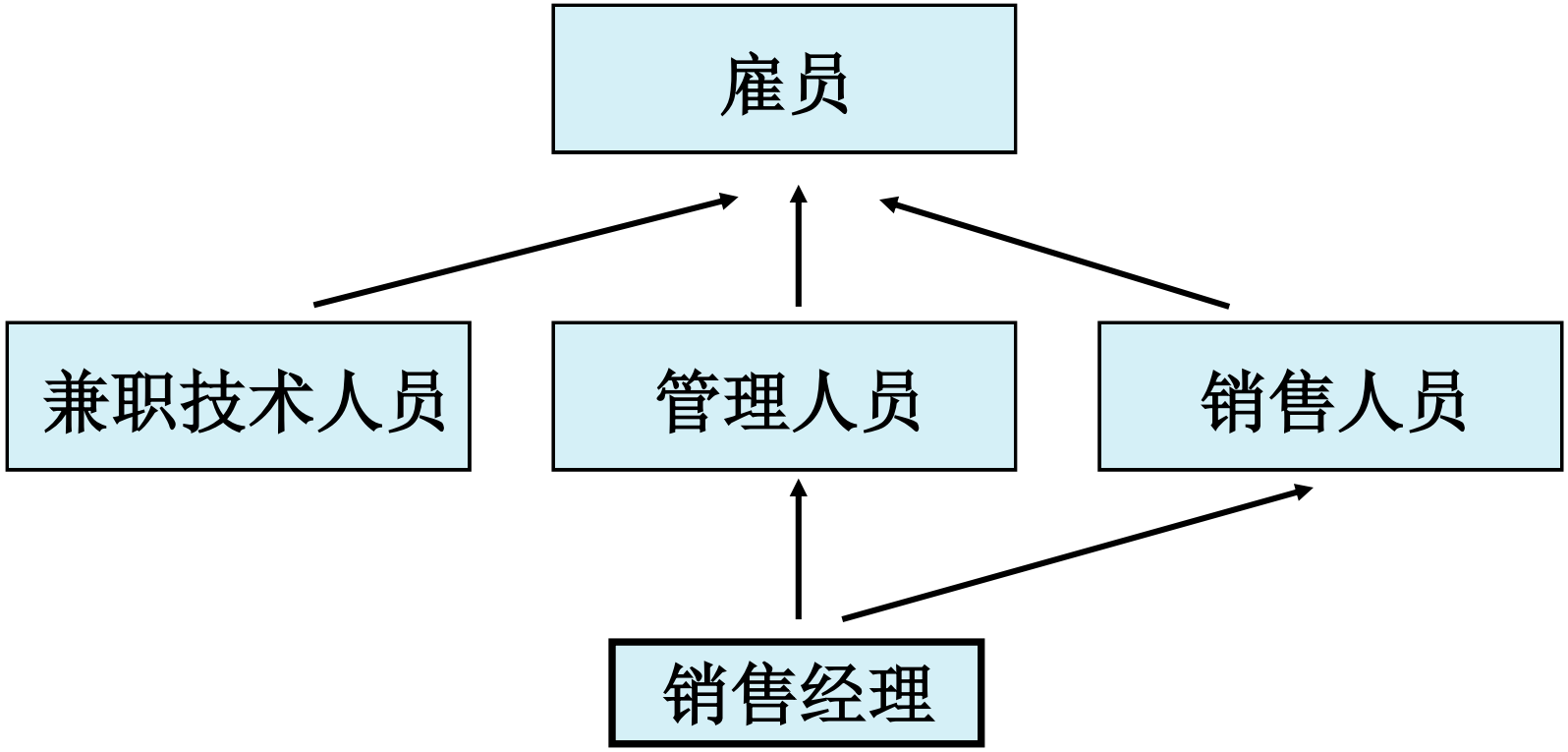
1 基类和派生类

- 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**。


交通工具
汽车
小汽车
卡车
旅行车
工具车
面包车
轿车


动物
猴子
猫
鸟
老虎
豹
狮 子


几何形状
圆
矩形


雇员
兼职技术人员
管理人员
销售人员
销售经理

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。

计算机系 江志英 jiangzy@mail.buct.edu.cn

类与类之间的关系

- 继承关系：

  描述了派生类与基类之间的“是”关系，派生类是基类中的一种，是它的具体化。

- 组合关系：

  又称包含关系，一个类中有另一个类中的对象，表现两者之间的“有”的关系。

继承的类型：单继承和多继承

继承可分为：

- 单重继承，指生成的派生类只有一个基类。
- 多重继承，是指生成的派生类有两个或两个以上的基类。

两者的区别仅在于所继承基类数不同

计算机系 江志英 jiangzy@mail.buct.edu.cn

派生类的定义

派生类定义格式如下：

```
class <派生类名>: <继承方式> <基类名>
{

    <派生类新增成员说明>

};
```

派生类的三种继承方式

- 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
- 继承方式包含以下3种：
  - public（公有的方式）；
  - private（私有的方式）；
  - protected（保护的方式）。
- 默认方式：
  - 对class来讲是private；
  - 对struct来讲是public。

计算机系 江志英 jiangzy@mail.buct.edu.cn

基类成员在派生类中的访问权限

① 基类中的私有成员无论哪种继承方式在派生类中都是不能直接访问的。

② 在公有继承方式下，基类中公有成员和保护成员在派生类中仍然是公有成员和保护成员。

③ 在私有继承方式下，基类中公有成员和保护成员在派生类中都为私有成员。

④ 在保护继承方式下，基类中公有成员和保护成员在派生类中都为保护成员。

基类成员在派生类中访问权限

基类中成员	公有继承	私有继承	保护继承
私成员	不可访问	不可访问	不可访问
公成员	公有	私有	保护
保护成员	保护	私有	保护

基类私有不可访问，公有不变，私有私有，保护保护。

成员访问权限的控制

(1) 公有继承方式

```
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>
#include<cmath>
using namecpace 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;
}
```


```
#include <iostream.h>
  class A
  {
   public:
     void f1();
   protected:
     int j1;
   private:
     int i1;
  };
```

```
class B:public A
 {
  public:
    void f2();
  protected:
    int j2;
  private:
    int i2;
 };
```

```
class C:public B
   {
    public:
      void f3();
   };
```

（2）私有继承方式

```
#include <iostream.h>
  class A
  {
   public:
     void f(int i)
     { cout<<i<<endl; }
     void g()
     { cout<<"A\n"; }
  };
```

```
class B:A //默认私有继承
  {
   public:
     void h()
     { cout<<"B\n"; }
     A::f;  //引入本作用域
  };
  void main()
  {
     B b;
     b.f(10);
     b.g( );  //错
     b.h( );
  }
```

(3) 保护继承方式

protected 成员的特点与作用：

- 对建立其所在类对象的模块来说，它与 private 成员的性质相同。
- 对于其派生类来说，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。


```
#include <iostream.h>
#include <string.h>
class A
{
 public:
   A(const char *str)
   {  strcpy(name,str);  }
 protected:
     char name[80];
};
```

```
class B:protected A
{
 public:
     B(const char *str):A(str)  {}
     void Print()
     {  cout<<name<<endl;  }
};
void main()
{
    B b("Zhang");
    b.Print();
}
```

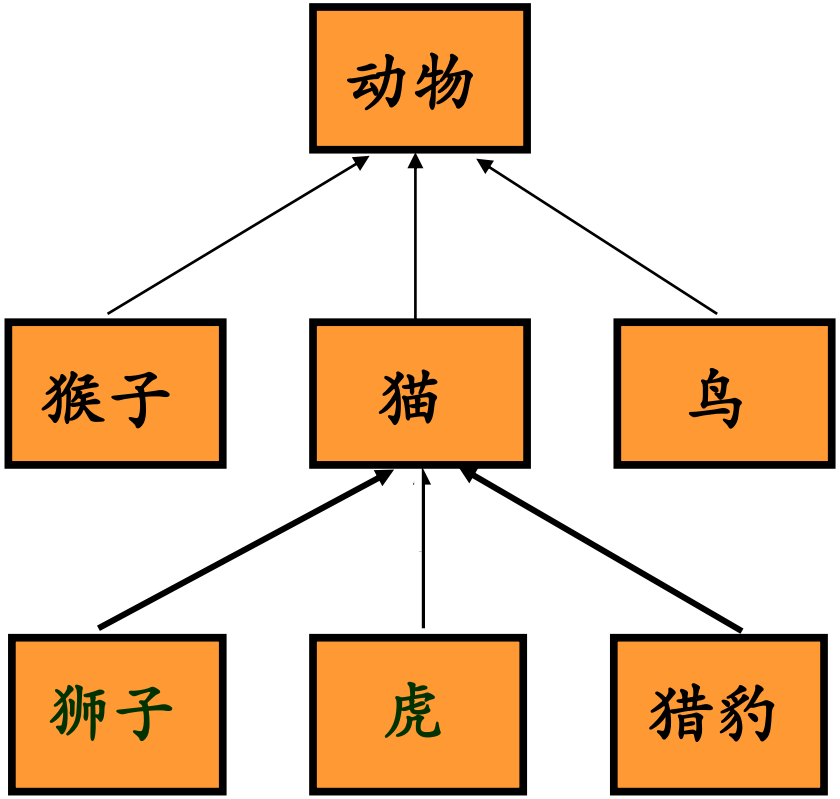
例 分析下列程序，并填写不同成员在不同类的访问权限。

```
#include <iostream.h>
  class A {
   public:
      void f1();
   protected:
      int a1;
   private:
      int a2;
  };
  class B:public A {
   public:
      void f2();
   protected:
      void f3();
   private:
      int b;
  };
  class C:protected B {
   public:
      void f4();
   protected:
      int c1;
   private:
      int c2;
  };
  class D:private C {
   public:
      void f5();
   protected:
      int d1;
   private:
      int d2;
  };
```

成员类	f1()	a1	a2	f2()	f3()	b	f4()	c1	c2	f5()	d1	d2
类A（基类）	公有	保护	私有	×	×	×	×	×	×	×	×	×
类B（公继）	公有	保护	不可访	公有	保护	私有	×	×	×	×	×	×
类C（保继）							公有	保护	私有	×	×	×
类D（私继）										公有	保护	私有

2 单继承

单重继承指的是仅有一个基类的派生继承。



派生类的构造函数和析构函数

- 构造函数不能继承。
- 派生类的构造函数承担着对基类中数据成员初始化和对派生类自身数据成员初始化的双重任务。

- 析构函数不能继承。
- 派生类的析构函数应包含着基类的析构函数，用来释放基类中的数据成员。

派生类的构造函数

•派生类的构造函数应该包含它的直接基类的构造函数。定义格式如下：

<派生类构造函数名>(<总参数表>):<基类构造函数名>(<参数表>)，<其他初始化项>

{

<派生类自身数据成员初始化>

}

掌握派生类构造函数时应注意如下两点

① 派生类构造函数的执行顺序如下:

- 先执行基类构造函数;
- 再执行子对象的构造函数（如有子对象的话）;
- 最后执行派生类构造函数的函数体。

② 派生类构造函数的成员初始化列表中应该显式地包含基类中带参数的构造函数，或者隐含地包含基类中的默认构造函数。

派生类的析构函数

- 由于析构函数不能继承，因此在派生类的析构函数中要包含它的直接基类的析构函数。
- 派生类析构函数的执行顺序如下：
  - 先执行派生类析构函数的函数体
  - 再执行子对象所在类的析构函数（如果有子对象的话）
  - 最后执行直接基类中的析构函数。

例 派生类构造函数的执行顺序

```
#include <iostream.h>
class A {
 public:
        A( ) {
           a=0;
           cout<<"Default constructor called."<<a<<endl;
        }
        A(int i) {
           a=i;
           cout<<"Constructor called."<<a<<endl;
        }
        ~A( ) {  cout<<"Destructor called."<<a<<endl;  }
        void Print( )  {  cout<<a<<',';  }
        int Geta( ) {  return a;  }
 private:
       int a;
};
```


```
class B:public A {
  public:
        B( ) {
           b=0;
           cout<<"Default constructor called."<<b<<endl;
        }
        B(int i,int j,int k):A(i),aa(j) {
           b=k;
           cout<<"Constructor called."<<b<<endl;
        }
        ~B( ) {  cout<<"Destructor called."<<b<<endl;  }
        void Print( )       {
           A::Print( );
           cout<<b<<','<<aa.Geta()<<endl;
        }
  private:
        int b;
        A aa;
};
```

```
void main( ) {
     B bb[2];
     bb[0]=B(7,8,9);
     bb[1]=B(12,13,14);
     for(int i=0;i<2;i++)
        bb[i].Print();
}
```

例 多层派生类构造函数和析构函数。

```
#include <iostream.h>
class A {
    public:
      A()
      { a=0; }
      A(int i)
      { a=i; }
      ~A()
      { cout<<"In A.\n"; }
      void Print()
      { cout<<a<<','; }
    private:
      int a;
};
```

```
class B:public A {
  public:
      B()
      { b1=b2=0; }
      B(int i)
      { b1=0;b2=i; }
      B(int i,int j,int k):A(i),b1(j),b2(k)
      { }
      ~B() { cout<<"In B.\n"; }
      void Print() {
          A::Print();
          cout<<b1<<','<<b2<<',';
      }
  private:
      int b1,b2;
};
```

```
class C:public B {
public:
        C( ) {  c=0;  }
        C(int i) {  c=i;  }
        C(int i,int j,int k,int l ):B(i,j,k),c( l )
        {  }
        ~C()
        {  cout<<"In C.\n";  }
        void Print() {
            B::Print();
            cout<<c<<endl;
        }
private:
        int c;
   };
```

```
void main( ) {
        C c1;
        C c2(10);
        C c3(10,20,30,40);
        c1.Print();
        c2.Print();
        c3.Print();
  }
```

子类型

- 当一个类型至少包含了另一个类型的所有行为，则称该类型是另一个类型的子类型。
  - 例如，在公有继承下，派生类是基类的子类型。
- 如果类型B是类型A的子类型，则称类型B适应于类型A，这时用类型A对象的操作也可以用于类型B的对象。因此，可以说类型B的对象就是类型A的对象。
- 子类型的关系是不可逆的。

类型适应

- 类型适应是指两种类型之间的关系
- B类型适应A类型是指B类型的对象能够用于A类型的对象所能使用的场合。
- 子类型与类型适应是一致的。

赋值兼容规则

- 当类型B是类型A的子类型时，则满足下述的赋值兼容规则。
  - ① B类的对象可以赋值给A类的对象。
  - ② B类的对象可以给A类对象引用赋值。
  - ③ B类的对象地址值可以给A类对象指针赋值。

- 通过基类对象名、指针、引用只能使用从基类继承的成员

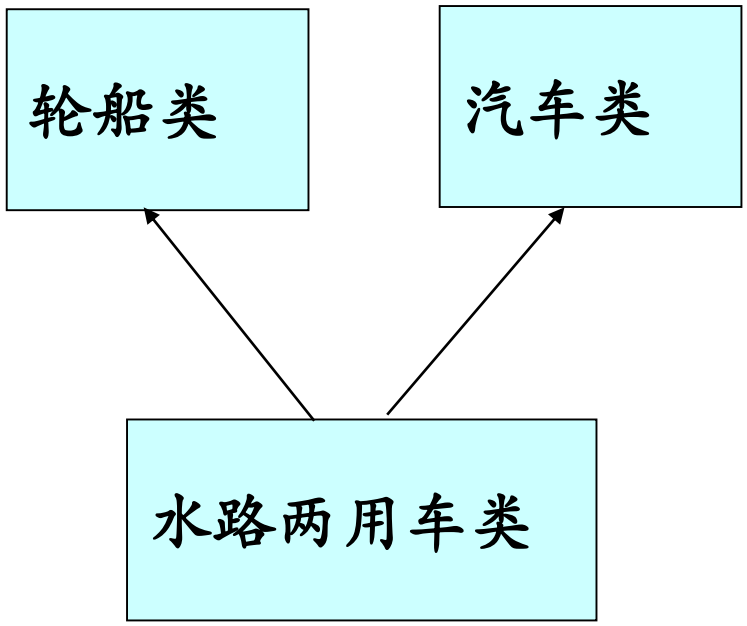
例 分析下列程序的输出结果，熟悉赋值兼容规则在该程序中的使用。

```
#include <iostream.h>
class A {
  public:
    A( )
    { a=0; }
    A(int i)
    { a=i; }
    void Print( )
    { cout<<a<<endl; }
    int Geta()
    { return a; }
  private:
    int a;
};
```

```
class B:public A {
public:
      B( ) { b=0; }
      B(int i,int j):A(i),b(j) { }
      void Print( ) {
          cout<<b<<',';
          A::Print();
      }
  private:
    int b;
};
void fun(A &a) {
    cout<<a.Geta( )+2<<',';
    a.Print();
}
```


```
void main( ) {
    A a1(10),a2;
    B b(10,20);
    b.Print();
    a2=b;
    a2.Print( );
    A *pa=new A(15);
    B *pb=new B(15,25);
    pa=pb;
    pa->Print();
    fun(*pb);
    delete pa;
}
```

多继承的概念



多继承的构造函数和析构函数

- 多继承派生类构造函数格式如下：

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

{

<派生类构造函数体>

}

计算机系 江志英 jiangzy@mail.buct.edu.cn

注意事项

- 在多重继承派生类构造函数中，先执行基类的构造函数。多个基类构造函数的执行顺序取决于定义派生类时规定的先后顺序，与派生类的成员初始化列表中顺序无关。
- 派生类构造函数中可以隐含着直接基类的默认构造函数。
- 多重继承派生类的析构函数中隐含着直接基类的析构函数，但其执行顺序与构造函数相反。

例　熟悉多重继承派生类构造函数和析构函数的用法

```
#include <iostream.h>
class A {
public:
        A(int i) {
            a=i;
            cout<<"Constructor
                        called.A\n";
        }
        ~A()
        {  cout<<"Destructor
                        called.A\n";  }
        void Print( )
        {  cout<<a<<endl;  }
private:
        int a;
};
```

例　熟悉多重继承派生类构造函数和析构函数的用法

```
class B {
public:
        B(int i) {
            b=i;
            cout<<"Constructor
                        called.B\n";
        }
        ~B()
        {  cout<<"Destructor
                        called.B\n";  }
        void Print()
        {  cout<<b<<endl;  }
private:
        int b;
};
```

```
class C {
public:
        C(int i) {
            c=i;
            cout<<"Constructor called.C\n";
        }
        ~C()
        {  cout<<"Destructor called.C\n";  }
        int Getc( ) {  return c;  }
private:
        int c;
};
```


计算机系 江志英 jiangzy@mail.buct.edu.cn

```
class D :public A,public B {
public:
    D(int i,int j,int k,int l):B(i),A(j),c(l) {
        d=k;
        cout<<"Constructor called.D\n";
    }
    ~D()
    {  cout<<"Destructor called.D\n";  }
    void Print()
    {
        A::Print();
        B::Print(); cout<<d<<','<<c.Getc()<<endl;
    }
  private:
    int d;
    C c;
};
```

```
void main( ) {
    D d(5,6,7,8);
    d.Print();
    B b(2);
    b=d;
    b.Print();
}
```

多继承的二义性

在多重继承时，在下面两种情况下，可能出现派生类对基类成员访问的二义性。

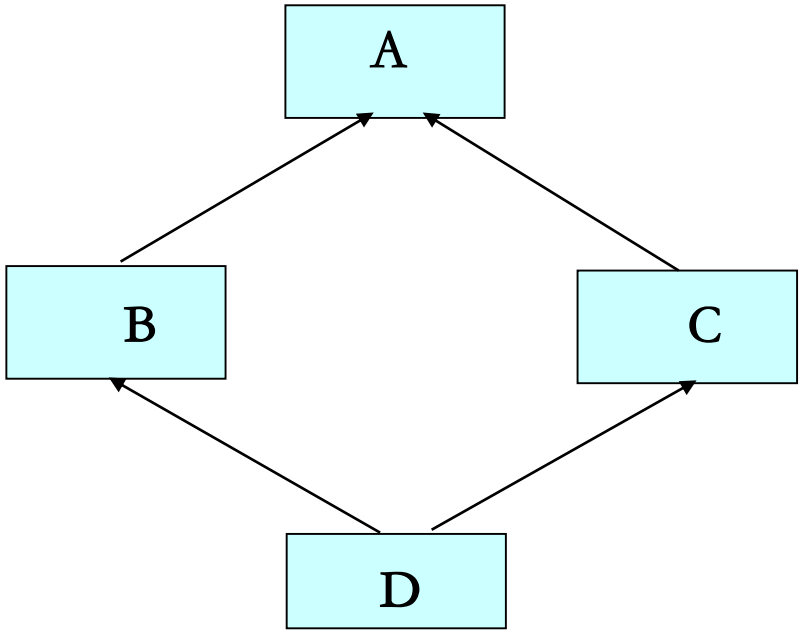
1．调用不同基类中的相同成员时可能出现二义性

例　分析下列程序的输出结果

```
#include <iostream.h>
class A
{
 public:
   void f()
   { cout<<"A.\n"; }
};
class B
{
 public:
   void f()
   { cout<<"B.\n"; }
   void g()
   { cout<<"BB.\n"; }
};
class D:public A,public B
{
 public:
   void g()
   { cout<<"DD.\n"; }
   void h()
   { B::f(); }
};
void main()
{
   D d;
   d.A::f();
   d.g();
   d.h();
}
```


计算机系 江志英 jiangzy@mail.buct.edu.cn

2. 当类层次结构如下时：

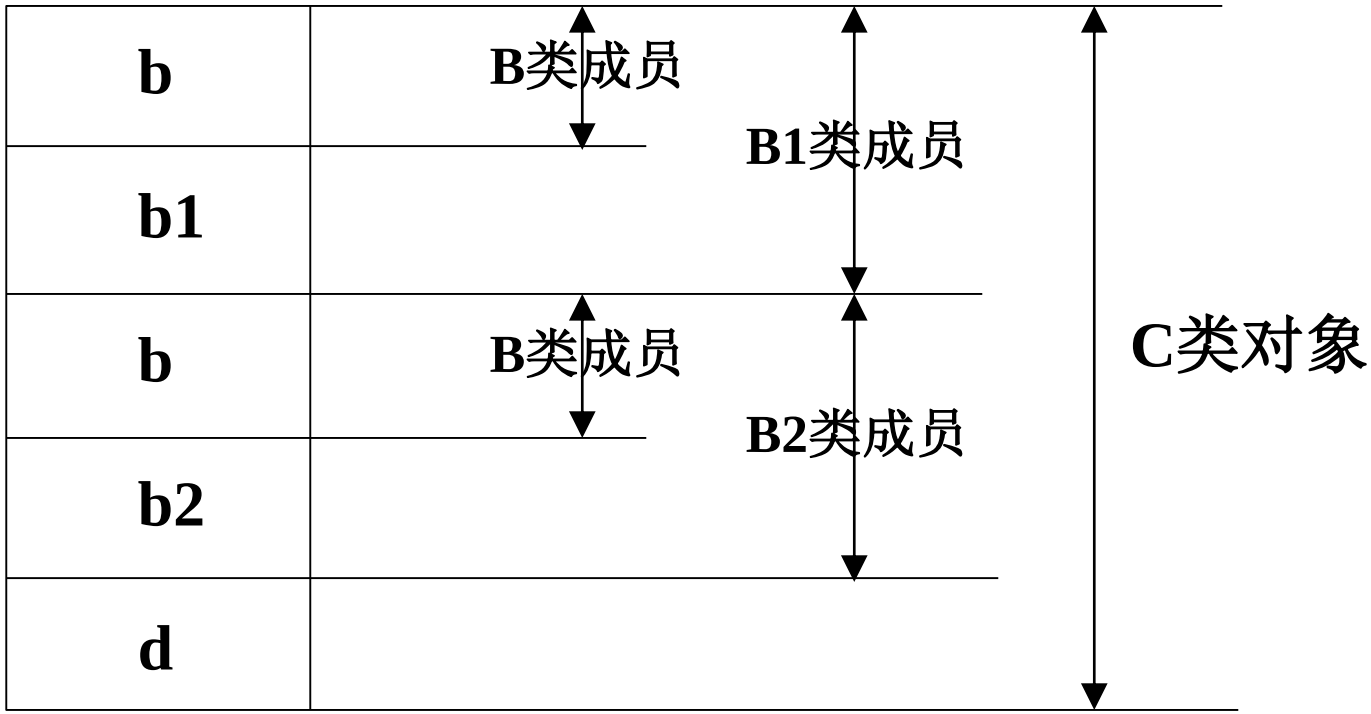


例

```
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;
}
```

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

虚基类的引入和说明

- 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唯一的基类成员，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。

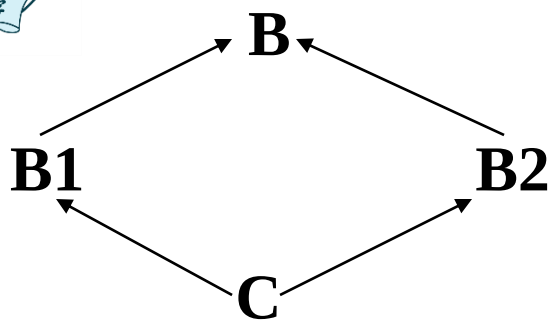
举例

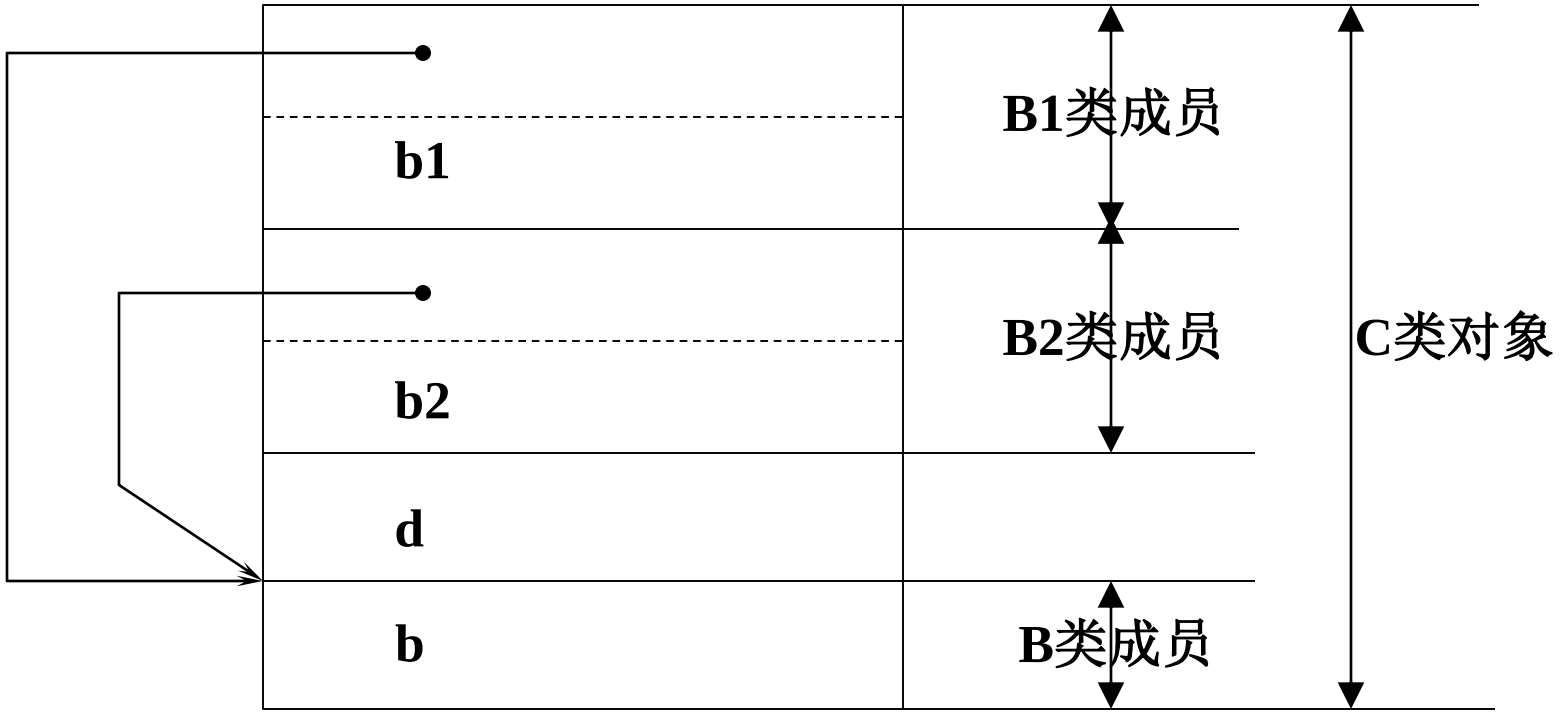
```
class B{ private: int b;};
class B1 : virtual public B { private: int b1;};
class B2 : virtual public B { private: int b2;};
class C : public B1, public B2{ private: float d;}
```

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：






```
#include <iostream>
using namecpace std;
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;}
};
int main()          //程序主函数
{       D1 d1;     //声明D1类对象d1
          d1.nV=2;          //使用最远基类成员
          d1.fun();
}
```

含有虚基类的派生类构造函数

- ✓ 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。
- ✓ 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。

```
#include <iostream>
using namecpace std;
class B0 //声明基类B0
{ public:            //外部接口
        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;}
};
int main()
{
        D1 d1(1);
        d1.nV=2;
        d1.fun();
}
```

小结

- 基类和派生类
- 派生类成员的访问权限
- 派生类的构造函数和析构函数
- 子类型及赋值兼容规则
- 多继承的二义性
- 虚基类