

2023

面向对象程序设计

第五讲： 继承和派生

李际军

lijijun@cs.zju.edu.cn



学习目标 / GOALS



- (1) 理解基类和派生类的概念;
- (2) 掌握派生类的声明、生成过程、继承方式和访问权限;
- (3) 掌握派生类的构造函数和析构函数;
- (4) 掌握多重继承的构造函数和析构函数、构造顺序和析构顺序及多重继承中的二义性处理方法;
- (5) 掌握虚基类的概念;
- (6) 理解子类型和赋值兼容规则;



01

类的继承与派生概念

02

基类与派生类

03

派生类的构造函数和析构函数

04

多重继承

05

子类型与赋值兼容规则

06

程序实例

01



类的继承与派生概念

- ◆ 继承与派生源于人们认识客观世界的过程，是自然界普遍存在的一种现象。
- ◆ 例如，“狗”和“黑狗”。
- ◆ 当人们谈及“狗”时，知道它有 4 条腿，1 条尾巴，喜欢吃骨头，为哺乳动物。如谈论“黑狗”时，人们又如何理解呢？通常人们把“黑狗”看作哺乳动物，也有 4 条腿，1 条尾巴，喜欢吃骨头，只不过增加了一个新的特征，即它的毛是黑色的。也就是说“黑狗就是毛色是黑色的狗”。在这里“狗”和“黑狗”之间存在一条重要内在的联系。“黑狗”是一类特殊的“狗”，“黑狗”从“狗”哪里继承了“狗”的全部特征，同时又增加了一个新特征。

◆所谓继承（ Inheritance ）就是在一个已存在的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（或父类， Base class ）；

◆ 定义出来的新类称为派生类（子类， Derived class ）；

◆继承是在已有的类的基础上定义新的类，从而形成类的层次和等级，体现面向对象程序设计的层次性概括方法

- ◆类的继承和派生层次结构，可以有助于人们对自然界的事物进行分类、分析和认识。
- ◆继承是软件可重用性的一种形式，新派生类通过继承从现有类中吸取其属性和行为，并对其进行覆盖或改写，产生新类所需要的功能。
- ◆派生类又可以作为另一个类的基类，派生出其他更“新”的类。

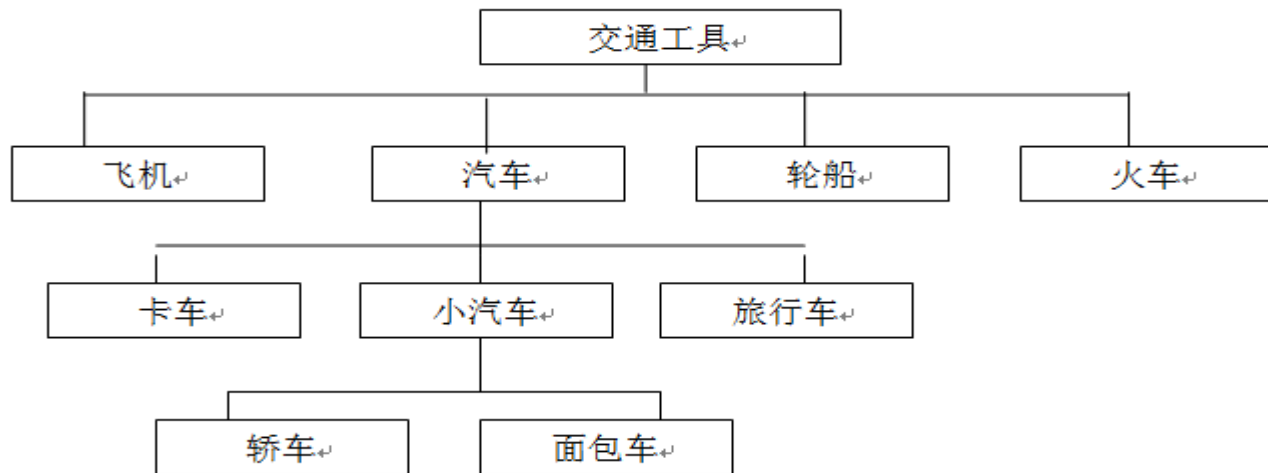


图 5-1 交通工具继承关系

继承的好处

◆ 软件重用

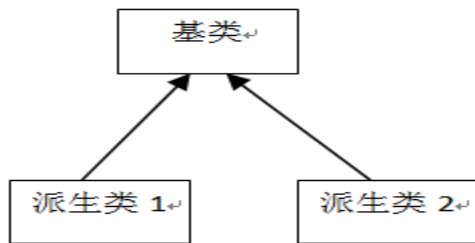
- ◆ 派生类可以直接利用基类已有的功能
- ◆ 具有从属关系的类可以通过继承机制联系起来，体系派生类对象和基类对象之间的 ‘ is-a-kind-of ’ 的关系
- ◆ 设计并测试好了的通用类可以组成类库重复使用

◆ 接口重用

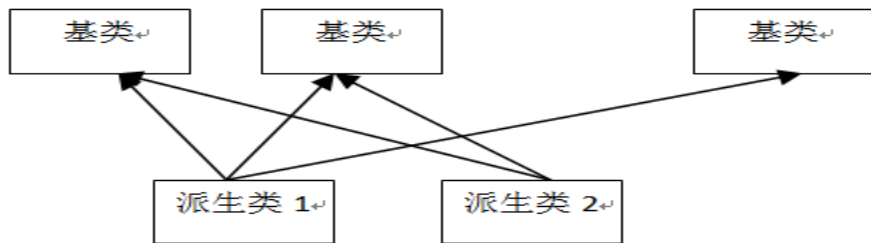
- ◆ 基类中定义的函数可以在派生类中重新定义
- ◆ 体现了接口与实现相分离的思想
- ◆ 继承是实现面向对象程序设计多态性概括方法的基本手段

单继承和多重继承

- ◆ 如果一个派生类只有一个直接的基类，那么称这种继承为单继承；
- ◆ 如果某个类的直接基类有两个或两个以上，则称该继承为多重继承；



a) 单一继承



b) 多重继承

图 5-2 类的单一继承和多重继承的 UML 结构图

02



基类与派生类

- ◆ 通过继承机制，可以利用已有数据类型来定义新的数据类型。
- ◆ 所定义的新的派生类，不仅拥有新定义的成员（数据成员、成员函数），而且还同时拥有旧的基类的成员。

派生类的声明

在 C++ 中，派生类的一般性声明语法如下：

```
class <派生类名> : <继承方式> <基类名>
{
    private:
        派生类成员声明;
    protected:
        派生类成员声明;
    public:
        派生类成员声明;
};
```

多继承下派生类的定义

多继承的情况下，派生类的定义格式与单继承时基本相同，只不过同时继承多个基类，继承方式也是分 **public**、**private** 和 **protected** 3 种情况。其格式为：

```
class 派生类名 : 继承方式 基类名 1, 继承方式 基类名 2, ..., 继承方式  
基类名 n  
{  
private:  
    新增数据成员和成员函数声明 ;  
public:  
    新增数据成员和成员函数声明 ;  
protected:  
    新增数据成员和成员函数声明 ;  
};
```

◆ 继承方式包含以下三种：

◆ `public` （公有继承方式）；

◆ `private` （私有继承方式）；

◆ `protected` （保护继承方式）。

- ◆ 派生类是在基类的基础上产生的。派生类的成员包括 3 种：
 - ◆ 吸收基类成员：派生类继承了基类的除了构造函数和析构函数以外的全部数据成员和函数成员。
 - ◆ 新增成员：增添新的数据成员和函数成员，体现了派生类与基类的不同和个性，是派生类对基类的发展。
 - ◆ 对基类成员进行改造，包含两层含义：一是，对基类成员的访问控制方式进行改造；二是，定义与基类同名的成员，即同名覆盖。

派生类的生成过程

◆ 派生类生成过程三个步骤:

◆ 继承基类成员;

◆ 对基类成员的改造;

◆ 添加派生类的新成员

【例 5-1】派生类的生成过程

```
class Point
{
protected:
    float x,y;          // 点的坐标 x,y
public:
    Point(int a,int b) {x=a; y=b; cout<<"Point..."<<endl; } // 构造函数
    void showX() {cout << "x="<<x<<endl;}
    void showY() {cout << "y="<<y<<endl;}
    void Show() {cout<<"x="<<x<<","<<y<<endl;}
    ~Point() {cout<<"Delete Point"<<endl;}
};
```

```
class Rectangle:public Point
{
    private:
        float H,W;    // 矩形的高和宽
    public:
        Rectangle(int a,int b,int h,int w):Point(a,b)
        {H=h;W=w;cout<<"Rectangle..."<<endl;}
        void ShowH() {cout<<"H="<<H<<endl;}
        void ShowW() {cout<<"W="<<W<<endl;}
        void Show() {cout<<"H="<<H<<","W="<<W<<endl;}
        ~Rectangle() {cout<<"Delete Rectangle!"<<endl;}
};
```

继承方式和派生类的访问权限

- ◆在声明派生类之后，派生类就继承了基类的数据成员和成员函数，但是这些成员并不都能直接被派生类所访问。采用不同的继承方式，决定了基类成员在派生类中的访问属性。
- ◆在 C++ 程序设计中，提供了三种继承方式：公有继承（`public`）、私有继承（`private`）、保护继承（`protected`）。
- ◆对于不同的继承方式，会导致基类成员原来的访问属性在派生类中发生变化。

派生方式	父类中的 访问权限	子类中的 访问权限
public (公有派生)	public	public
	protected	protected
	private	private
protected (保护派生)	public	protected
	protected	protected
	private	private
private (私有派生)	public	private
	protected	private
	private	private

继承方式和派生类的访问权限

- ◆ 在 C++ 程序设计中，访问来自两个方面：
 - ◆ 派生类的新增成员对从基类继承来的成员的访问；
 - ◆ 派生类外部（非类成员），通过派生类对象对从基类继承来的成员的访问。

表 5-1 访问属性与继承的关系

访问属性 继承方式	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

继承方式和派生类的访问权限 -1. 公有继承 (public inheritance)

- ◆当类的继承方式为 public （公有），基类的公有成员（ public ）和保护成员（ protected ）在派生类中保持原有访问属性，其私有成员（ private ）仍为基类私有。
- ◆派生类类内：可以访问基类中的公有成员和保护成员，而基类的私有成员则不能被访问。
- ◆派生类类外：只能通过派生类对象访问继承来的基类中的公有成员。

继承方式和派生类的访问权限 -1. 公有继承 (public inheritance)

表 5-2 公有继承中派生类及派生类对象对基类的访问属性

基类成员	基类成员在派生类中的访问属性（类内）	基类成员在派生类对象中的访问属性（类外）
公有成员	公有	公有
保护成员	保护	不能被访问
私有成员	不能被访问	不能被访问


```
#include<iostream>
#include<string>
using namespace std;
class Person
{
public:                                     // 基类公有成员函数
    Person(string nna="",char nsex='m',string nphonenum=""):
        name(nna),sex(nsex),phonenum(nphonenum){ }
    void Input();
    void Show();
private:                                  // 基类私有数据成员
    string name;
    char sex;
    string phonenum;
};
```

```
void Person::Input()
{
    cout<<"Input name:";cin>>name;
    cout<<"Input sex:"; cin>>sex;
    cout<<"Input phonenum:"; cin>>phonenum;
}

void Person::Show()
{
    cout<<"name="<<name<<endl;
    cout<<"sex="<<sex<<endl;
    cout<<"phonenum="<<phonenum<<endl;
}

class Teacher:public Person           // 派生类 Teacher 的声明
{
public:                                // 新增公有成员函数
    void Input_t();
    void Show_t();
```

```
private:                                // 新增私有数据成员
    string title;
    double wage;
};
void Teacher::Input_t(){
    Input();
    cout<<"Input title:"; cin>>title;
    cout<<"Input wage:"; cin>>wage;
}
void Teacher::Show_t(){
    Show();
    cout<<"title="<<title<<endl;
    cout<<"wage="<<wage<<endl;
}
class Cadre:public Person               // 派生类 Cadre 的声明
{
```



```
• public:                                // 新增公有成员函数
•     void Input_c();
•     void Show_c();
• private:                               // 新增私有数据成员
•     string post;
•     string political;
• };
• void Cadre::Input_c() {
•     Input();
•     cout<<"Input post: "; cin>>post;
•     cout<<"Input political: "; cin>>political;
• }
• void Cadre::Show_c()
• {
•     Show();
•     cout<<"post="<<post<<endl;
•     cout<<"political="<<political<<endl;
• }
```

```
int main()
{
    Teacher t;
    cout<<" 请输入教师的信息: "<<endl;
    t.Input_t();
    cout<<" 该教师的信息: "<<endl;
    t.Show_t();
    cout<<endl;
    Cadre c;
    cout<<" 请输入干部的信息: "<<endl;
    c.Input_c();
    cout<<" 该干部的信息: "<<endl;
    c.Show_c();
    return 0;
}
```

继承方式和派生类的访问权限 -2. 私有继承 (private inheritance)

- ◆当类的继承方式为 `private`（私有），基类的公有成员（`public`）和保护成员（`protected`）都以私有成员身份出现在派生类中，而基类私有成员（`private`）在派生类中仍不可访问。也就是说，基类的公有成员（`public`）和保护成员（`protected`）被继承后作为派生类的私有成员（`private`），派生类的其它成员可以直接访问它们。
- ◆派生类类内：可以访问基类中的公有成员和保护成员，而基类的私有成员则不能被访问。
- ◆派生类类外：通过派生类对象不能访问基类中的任何成员。

继承方式和派生类的访问权限 -2. 私有继承 (private inheritance)

表 5-3 私有继承中派生类及派生类对象对基类的访问属性

基类成员	基类成员在派生类中的访问属性（类内）	基类成员在派生类对象中的访问属性（类外）
公有成员	私有	不能被访问
保护成员	私有	不能被访问
私有成员	不能被访问	不能被访问



5.1

```
#include<iostream>
#include<string>
using namespace std;
class Person
{
public:                                     // 基类公有成员函数
    Person(string nna="",char nsex='m',string nphonenum=""):
        name(nna),sex(nsex),phonenum(nphonenum){ }
    void Input();
    void Show();

private:                                  // 基类私有数据成员
    string name;
    char sex;
    string phonenum;
};
void Person::Input()
{
    cout<<"Input name:";cin>>name;
    cout<<"Input sex:"; cin>>sex;
```




5.1

```
cout<<"Input phonenum:"; cin>>phonenum;
}
void Person::Show()
{
    cout<<"name="<<name<<endl;
    cout<<"sex="<<sex<<endl;
    cout<<"phonenum="<<phonenum<<endl;
}
class Teacher:private Person           // 派生类 Teacher 私有继承基类 Person
{
public:                                // 新增公有成员函数
    void Input_t();
    void Show_t();

private:                               // 新增私有数据成员
    string title;
    double wage;
};
```



基类与派生类【例 5.3】使用私有继承改写例

34

5.1

```
void Teacher::Input_t()
{
    Input();
    cout<<"Input title:"; cin>>title;
    cout<<"Input wage:"; cin>>wage;
}

void Teacher::Show_t()
{
    Show();
    cout<<"title="<<title<<endl;
    cout<<"wage="<<wage<<endl;
}

class Cadre:private Person           // 派生类 Cadre 私有继承基类 Person
{
public:                               // 新增公有成员函数
    void Input_c();
    void Show_c();
}
```



5.1

```
private:                                // 新增私有数据成员
```

```
    string post;  
    string political;
```

```
};
```

```
void Cadre::Input_c()
```

```
{
```

```
    Input();  
    cout<<"Input post:"; cin>>post;  
    cout<<"Input political:"; cin>>political;
```

```
}
```

```
void Cadre::Show_c()
```

```
{
```

```
    Show();  
    cout<<"post="<<post<<endl;  
    cout<<"political="<<political<<endl;
```

```
}
```



5.1

```
int main()
{
    Teacher t;
    cout<<" 请输入教师的信息: "<<endl;
    t.Input_t();
    cout<<" 该教师的信息: "<<endl;
    t.Show_t();
    cout<<endl;
    Cadre c;
    cout<<" 请输入干部的信息: "<<endl;
    c.Input_c();
    cout<<" 该干部的信息: "<<endl;
    c.Show_c();
    return 0;
}
```

继承方式和派生类的访问权限 -3. 保护继承 (protected inheritance)

- ◆ 当类的继承方式为 `protected`（保护），基类的公有成员（`public`）和保护成员（`protected`）都以保护成员身份出现在派生类中，而基类的私有成员（`private`）仍不可访问。也就是说，基类的保护成员只能被基类的成员函数或派生类的成员函数访问，不能被派生类以外的成员函数访问。
- ◆ 派生类类内：可以访问基类中的公有成员和保护成员，而基类的私有成员则不能被访问。
- ◆ 派生类类外：通过派生类对象不能访问基类中的任何成员。

继承方式和派生类的访问权限 -3. 保护继承 (protected inheritance)

表 5-4 保护继承中派生类及派生类对象对基类的访问属性

基类成员	基类成员在派生类中的访问属性	基类成员在派生类对象中的访问属性（类外）
公有成员	保护	不能被访问
保护成员	保护	不能被访问
私有成员	不能被访问	不能被访问

5.1

[illegible]



基类与派生类【例 5.4】采用保护继承改写例

5.1

40

```
void Person::Input()
{
    cout<<"Input name: ";cin>>name;
    cout<<"Input sex: "; cin>>sex;
    cout<<"Input phonenum: "; cin>>phonenum;
}
void Person::Show()
{
    cout<<"name="<<name<<endl;
    cout<<"sex="<<sex<<endl;
    cout<<"phonenum="<<phonenum<<endl;
}
string Person::GetName()
{
    return name;
}
string Person::GetPhonenum()
{
    return phonenum;
}
```




5.1

```
class Teacher:protected Person {    // 派生类 Teacher 保护继承基类 Person
public:                               // 新增公有成员函数
    void Input_t();
    void Show_t();
private:                             // 新增私有数据成员
    string title;
    double wage;
};
void Teacher::Input_t(){
    Input();
    cout<<"Input title:"; cin>>title;
    cout<<"Input wage:"; cin>>wage;
}
void Teacher::Show_t(){
    Show();
    cout<<"title="<<title<<endl;
    cout<<"wage="<<wage<<endl;
```



5.1

```
}  
class Cadre:protected Person           // 派生类 Cadre 保护继承基类 Person  
{  
public:                                // 新增公有成员函数  
    void Input_c();  
    void Show_c();  
private:                              // 新增私有数据成员  
    string post;  
    string political;  
};  
void Cadre::Input_c()  
{  
    Input();  
    cout<<"Input post:"; cin>>post;  
    cout<<"Input political:"; cin>>political;  
}
```



5.1

```
void Cadre::Show_c()
{
    Show();
    cout<<"post="<<post<<endl;
    cout<<"political="<<political<<endl;
}

int main(){
    Teacher t;
    cout<<" 请输入教师的信息： "<<endl;
    t.Input_t();
    cout<<" 该教师的信息： "<<endl;
    t.Show_t();
    cout<<t.GetName()<<endl;           // 不能访问，错误！
    cout<<t.GetPhonenum()<<endl;       // 不能访问，错误！
    cout<<endl;
    Cadre c;
    cout<<" 请输入干部的信息： "<<endl;
```



5.1

```
c.Input_c();  
cout<<" 该干部的信息: "<<endl;  
c.Show_c();  
cout<<c.GetName()<<endl;           // 不能访问, 错误!  
cout<<c.GetPhonenum()<<endl;       // 不能访问, 错误!  
return 0;  
}
```

继承方式和派生类的访问权限

◆ 派生类的三种继承方式及基类成员在派生类的访问权限

继承方式	基类特性	派生类特性	派生类中的成员函数	派生类的对象
公有继承	public protected private	public protected 不可访问	可访问基类中的公有成员 和保护成员	可访问基类和派生 类中的公有成员
私有继承	public protected private	private private 不可访问	可访问基类中的公有成员 和保护成员	不能访问基类中的 所有成员
保护继承	public protected private	protected protected 不可访问	可访问基类中的公有成员 和保护成员	不能访问基类中的 所有成员

00



调整派生类中的访问属性的其他方法

- C++ 允许派生类中说明的成员与基类的成员名字相同
- 派生类的同名成员覆盖了基类的同名成员

```
class X{  
    int a,c;  
public:  
    void print( );  
};
```

Y 类的 c 和 print 对类 X 的同名成员有覆盖！

```
class Y : public X {  
    int b,c;  
public:  
    void print( );  
    void show( )  
  
    { print(); X::print(); }  
};
```

- 继承中的私有派生改变基类的公有访问属性为私有，此时外界无法利用派生类的对象直接使用基类的成员。
- C++ 提供了调整机制——访问声明，可个别调整基类的某些成员，使之在派生类中保持原来的访问属性。
- 方法：在公有（`public`）区域添加
基类名 :: 成员名字；



2 访问声明（续）

49

- 访问声明举例

```
class X{  
    int a;  
public:  
    int a;  
    void print( );  
};
```

```
class Y : private X {  
    int b;  
public:  
    X::print;  
    X::a;  
    void show( )  
    { cout<<b<<endl; }  
};
```

03



派生类的构造函数和析构函数

- ◆ 基类的构造函数和析构函数是不能被继承，需要在派生类中重新定义。
- ◆ 由于派生类继承了基类的成员，在初始化时，也要同时初始化基类成员。
可通过调用基类的构造函数完成初始化。

在 C++ 中，派生类构造函数的一般性声明语法如下：

```
<派生类名>::<派生类名> (基类形参, 内嵌对象形参, 本类形参): <基类名> (参数表), <内嵌对象 1> (参数表 1), <内嵌对象 2> (参数表 2), ..., <内嵌对象 n> (参数表 n)
{
    本类成员初始化赋值语句;
    ...
};
```

- ◆ 派生类的构造函数名与派生类名相同。
- ◆ 冒号之后，列出需要使用参数进行初始化的基类名和内嵌成员对象名及各自的参数表，各项之间用逗号分隔。
- ◆ 对于基类成员，如使用默认构造函数，可以不给出基类名和参数表。
- ◆ 对于内嵌对象成员，如使用默认构造函数，也无需写出对象名和参数表。

◆ 派生类构造函数的执行

- ◆ 先调用基类的构造函数对继承成员进行初始化，再调用对新加成员初始化的部分。

◆ 若基类构造函数带有参数

- ◆ 必须由派生类构造函数的形式参数中为基类构造函数提供实参。

◆ 若基类构造不带参数

- ◆ 定义派生类构造函数时，可以不必显式的调用基类构造函数。

- ◆ C++ 编译程序认为调用的是基类中形式参数为空的构造函数。无参数的构造函数可以是 C++ 编译程序自动产生的缺省构造函数，也可以是程序员自己声明的。

◆ 基类构造函数带参数

- ◆ 定义派生类构造函数时必须显式调用基类构造函数，并用在派生类构造函数的形参部分为基类构造函数提供实际参数。
- ◆ 即使派生类本身的构造函数不带参数也必须在冒号“:”之后调用基类的构造函数，但这时传递给基类构造函数的实际参数通常是一些常量表达式。

【例 5-5】 构造函数例题

```
#include<iostream>
using namespace std;
class vehicle // 基类
{
private:      // 私有数据成员
    int wheels; // 轮子数量
    float weight; // 重量
public:
    vehicle(int input_wheels,float input_weight) // 基类构造函数
        { wheels=input_wheels; weight=input_weight;}
};
class car:public vehicle // 公有派生
{
private:
    int passenger_num; // 新增数据成员, 载客人数
public:
    car(int input_wheels,float input_weight,int num):vehicle(input_wheels,input_weight) // 派生类构造函数的定义
        { passenger_num=num;}
};
```


【例 5-5】 构造函数例题 2

```
#include <iostream>
using namespace std;
class X{
private:
    int x;
public:
    X(int i){x=i;} // 类 X 构造函数
};
class A{
    int a;
public:
    A(int i=0):a(i) {} // 类 A 构造函数
};
class B:public A{ // 公有继承
    int b; // 新增数据成员
    X x; // 新增内嵌成员对象
public:
    B(int i, int j, int k):A(i), x(j), b(k) {}
    // B 的构造函数，对基类 A、内嵌对象 x 和新增数据成员 b 的初始化
};
```

派生类析构函数的构建

◆当对象被删除时，派生类的析构函数被执行。在派生过程中，由于基类的析构函数也不能被继承，因此在执行派生类的析构函数时，基类的析构函数也将被调用。析构函数没有类型，也无参数，与构造函数相比，情况相对比较简单。如果未显示定义某个类的析构函数，系统会自动为每一个类都生成一个默认的析构函数。

◆派生类析构函数的的定义方法与没有继承关系的类中析构函数的定义方法完全相同，只需在派生类析构函数体中把派生类新增的成员（非成员对象）的清理工作做好，系统就会自己调用基类及成员对象的析构函数来对基类及成员对象进行清理。

派生类析构函数的构建

【例 5-6】析构函数例题。

```
#include <iostream>
using namespace std;
class A {
private:
    int a1,a2;
public:
    A() { a1=0;a2=0;}          // 基类 A 默认构造函数
    A(int i, int j) {a1=i;a2=j;}
    void print() {cout<<a1<<" "<<a2<<" ";}
    ~A() {cout<<"A destructor called. "<<endl;}
};
class B:public A {
private:
    int b;
public:
    B() {b=0;}               // 派生类构造函数
    B(int i, int j, int k):A(i, j) {b=k;}    // 派生类构造函数
    void print() {A::print();cout<<b<<endl;}
    ~B() {cout<<"B destructor called. "<<endl;}
};
```

派生类构造函数和析构函数执行顺序

◆ 派生类构造函数的执行顺序：

- ◆ 派生类构造函数执行顺序一般是：先祖先（基类），再客人（内嵌对象），后自己（派生类本身）。顺序如下：
 - ◆ 先调用基类的构造函数。
 - ◆ 然后按照数据成员（包括内嵌对象、常量、引用等必须初始化的成员）的声明顺序，依次调用数据成员的构造函数或初始化数据成员。
 - ◆ 最后执行派生类构造函数的函数体。

派生类构造函数和析构函数执行顺序

◆ 派生类析构函数的执行顺序：

- ◆ 派生类析构函数执行顺序与构造函数正好相反：先自己（派生类本身），再客人（内嵌对象），后祖先（基类）。顺序如下：
 - ◆ 先执行派生类的析构函数，对派生类新增普通成员进行清理。
 - ◆ 然后按着内嵌对象声明的相反顺序，依次调用内嵌对象的析构函数，对派生类新增的对象成员进行清理。
 - ◆ 最后调用基类的析构函数，对所有从基类继承来的成员进行清理。

【例 5-7】 派生类构造函数和析构函数的执行顺序例题

```
class B2 {
private:
    int b2;
public:
    B2(int i):b2(i){cout<<"construction B2 "<<b2<<endl;}
    ~B2() {cout <<"destructing B2 "<<endl;}
};
class C:public A {
private:
    int c;
    B1    b1;  // 内嵌对象
    B2    b2;  // 内嵌对象
public:
    C(int i,int j1,int j2,int k):A(i),b2(j2),b1(j1),c(k){cout<<"construction C "<<c<<endl;}
    ~C() {cout <<"destructing C "<<endl;} };
int main()
{
    C c1(1,2,3,4);
    return 0;
}
```

04

多重继承



- ◆ 多重继承可以看作是单继承的扩展。
- ◆ 所谓多重继承是指派生类具有多个基类，派生类与每一个基类之间关系可以看作是一个单继承。在现实生活中，很多继承均表现为多重继承。例如，两用沙发，它既是一个沙发，又是一张床。假如人们已经定义了沙发类和床类，那么两用沙发同时继承了沙发和床两个类的特征。

01

OPTION

多重继承的声明

◆ 在 C++ 中，多重继承的一般性声明语法如下：

```
class <派生类名> : <继承方式 1> <基类名 1> , <继承方式 2> <基类名 2> ,  
    ..., <继承方式 n> <基类名 n> {  
    派生类新增加的成员; };
```

◆ 其中：

- ◆ <继承方式 1>、<继承方式 2>、…、<继承方式 n> 是三种继承方式：
public(公有)、private(私有)和 protected(保护)。如果没有显式地指出继承方式，系统默认继承方式为 private(私有)。
- ◆ 各个基类之间用逗号隔开。
- ◆ 派生类继承了多个基类的成员，基类中的成员按照继承方式来确定其在派生类中的访问方式。

多重继承的构造函数和析构函数

◆ 在多重继承的情况下，派生类的构造函数一般性声明语法如下：

```
<派生类名>::<派生类名> (基类形参, 内嵌对象形参, 本类形参): <基类名 1>  
    (参数表), <基类名 2> (参数表), ..., <基类名 n> (参数表), <内嵌对象  
    1> (参数表 1), <内嵌对象 2> (参数表 2), ..., <内嵌对象 n> (参数表 n)  
{  
    本类成员初始化赋值语句;  
    ...  
};
```

【例 5-8】 多重继承派生类构造函数和析构函数例题。

```
#include <iostream>
using namespace std;
class A1          // 基类 A1 的声明
{
public:
    A1(int i)      {cout<<"constructing A1 "<<i<<endl;}    // 构造函数，带参数
    ~A1() {cout<<"destructing A1 "<<endl;}                // 析构函数
};
class A2          // 基类 A2 的声明
{
public:
    A2(int j)      {cout<<"constructing A2 "<<j<<endl;}    // 构造函数，带参数
    ~A2() {cout<<"destructing A2 "<<endl;}                // 析构函数
};
class A3  // 基类 A3 的声明
{
public:
    A3() {cout<<"constructing A3 *"<<endl;}                // 默认构造函数
    ~A3() {cout<<"destructing A3 "<<endl;}                // 析构函数
};
```

【例 5-8】 多重继承派生类构造函数和析构造函数例题。

```
class B: public A2, public A1, public A3           // 派生类 B，公有继承 A2、A1、A3
{
public:
    B(int a, int b, int c, int d): A1(a), memberA2(d), memberA1(c), A2(b)
        {cout<< "constructing B" }
    ~B() {cout<<"destructing B "<<endl;}

private:
    A1 memberA1;    // 内嵌对象成员 memberA1
    A2 memberA2;    // 内嵌对象成员 memberA2
    A3 memberA3;    // 内嵌对象成员 memberA3
};
int main()
{
    B obj(1, 2, 3, 4);
    return 0;
}
```

03 OPTION

多重继承中的二义性

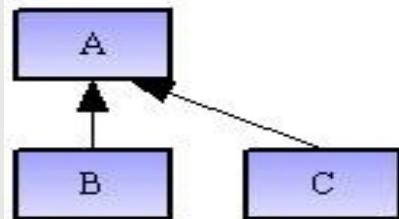
- ◆一般而言，在派生类中对基类成员的访问必须是唯一的。但是，由于多重继承方式下，派生类可能有多个直接基类或间接基类，这虽然充分体现了软件重用的优点，也可能造成对基类中某个成员的访问出现了不确定的情况，使得这种访问具有二义性。
- ◆在多重继承中，派生类对基类成员访问在下列两种情况下可能出现二义性。
 - ◆ 访问不同基类的相同成员时可能出现二义性
 - ◆ 访问共同基类中成员时可能出现二义性

03

OPTION

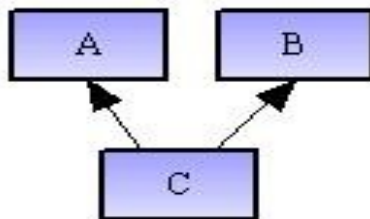
多重继承中的二义性

单重继承



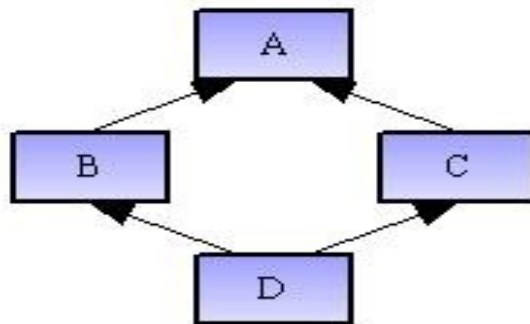
每一个派生类只有一个直接基类

多重继承



每个派生类具有两个或两个以上直接基类

重复继承

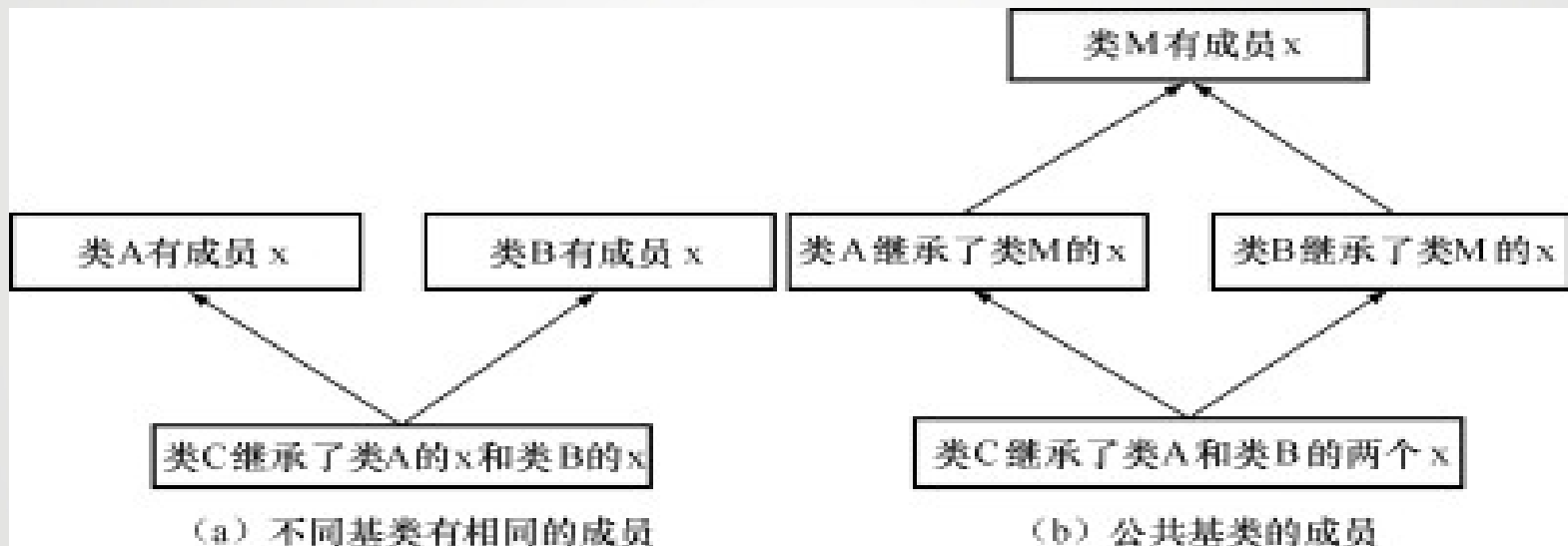


多重继承的一种特殊形式
派生类2次或2次以上重复继承某个祖先基类

03

OPTION

多重继承中的二义性



- ◆ 1. 派生类的不同基类有同名成员
 - ◆ 当派生类的不同基类有同名成员，则派生类中将产生二义性问题：
 - ◆ 派生类类内：访问同名基类成员时将产生二义性问题
 - ◆ 派生类类外：通过派生类对象访问同名的基类成员时将产生二义性问题。

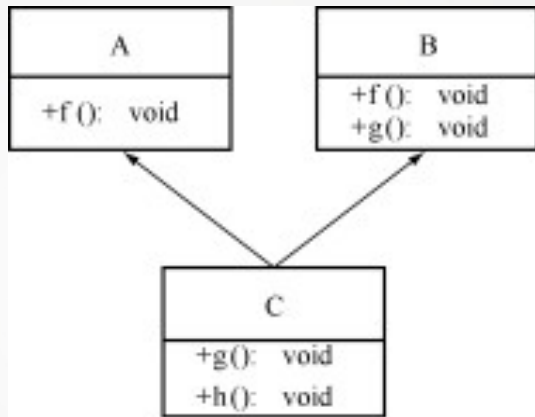
【例 5-9】 派生类的对象访问同名的基类成员产生二义性。

```
#include <iostream>
using namespace std;
class A
{
    public:
        void f() {cout<<"From  A"<<endl;}
};
class B
{
    public:
        void f() {cout<<"From  B"<<endl;}
        void g();
};
```

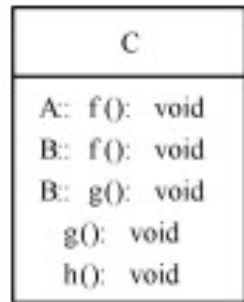
【例 5-9】 派生类的对象访问同名的基类成员产生二义性。

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

int main()
{
    C c1;
    c1.f();    // 产生二义性
    return 0;
}
```



(a) UML图



(b) 类C的成员结构

【例 5-10】 在派生类中访问同名的基类成员产生二义性。

```
#include <iostream>
using namespace std;
class A
{
    public:
        void f() {cout<<"From  A"<<endl;}
};
class B
{
    public:
        void f() {cout<<"From  B"<<endl;}
};
```

【例 5-10】 在派生类中访问同名的基类成员产生二义性。

```
class C: public A, public B
{
    public:
        void h() { f(); }    // 产生二义性
};

int main(int argc, char* argv[])
{
    C c1;
    c1.h();
    return 0;
}
```

03 OPTION

多重继承中的二义性

- ◆ 解决方法：
 - ◆ 使用作用域标识符
 - ◆ 使用同名覆盖的原则。
- ◆ （1）使用作用域标识符，进行成员限定消除二义性。
 - ◆ 使用作用域标识符 “::” 进行限定的一般格式为：
 - ◆ 对象名 . 基类名 :: 成员名
 - ◆ 对象名 . 基类名 :: 成员名（参数表）

如例 5-10 中，主函数中使用作用域标识符进行成员限定，告诉编译器调用的是哪个类的同名函数，即可消除二义性。

```
int main()
{
    C c1;
    c1.A::f();           // 或 c1.B::f(); 消除了二义性。
    return 0;
}
```

如例 5-10 中，派生类 C 中的成员函数 h() 调用 f() 时使用作用域标识符进行成员限定。

```
class C: public A, public B
{
    public:
        void h() {A::f();}    // 或 B::f();
};
```

- ◆（2）使用同名覆盖的原则。
- ◆在派生类中重新定义与基类同名的成员（如果是成员函数，在参数表也要相同，参数情况不同为重载），以隐蔽掉同名的基类成员。原因是同名隐藏规则，规定派生类的成员函数将覆盖基类中同名的成员。这样在访问同名成员时，使用的就是派生类中的成员，二义性问题得到解决。

【例 5-11】 同名覆盖原则。

```
#include <iostream>
using namespace std;
class A
{
    public:
        int x;
        void f() {cout<<"From  A"<<endl;}
};
```

```
class B
{
    public:
        int x;
        void f() {cout<<"From  B"<<endl;}
};
class C: public A, public B
{
    public:
        int x;
        void f() {A::f();}      // 或 B::f(); , 消除二义性
};
```

```
int main()
{
    C c1;
    c1.x=4;
    c1.A::x=8;
    c1.B::x=12;
    c1.f();
    return 0;
}
```

- ◆ 2. 在派生类中引用公共基类中成员时出现二义性
 - ◆ 在继承与派生的类层次结构中，被继承的多个基类如果有一个共同的基类，在派生类中访问这个共同基类的成员时也会产生二义性问题。
 - ◆ 解决这种二义性方法也有两种：
 - ◆ （1）使用作用域标识符
 - ◆ （2）虚基类

◆ (1) 使用作用域标识符

◆ 格式: 派生类对象. 基类 :: 基类成员 (类外)

◆ 基类 :: 基类成员 (类内)

◆ 【例 5-12】 派生类的直接基类全部或部分从另一个共同基类派生而出产生的二义性

```
#include <iostream>
using namespace std;
class L1
{
public:
    int m1;
    void f1() {cout<<"layer 1-> m1="<<m1<<endl;}
};
```

◆ 【例 5-12】 派生类的直接基类全部或部分从另一个共同基类派生而出产生的二义性

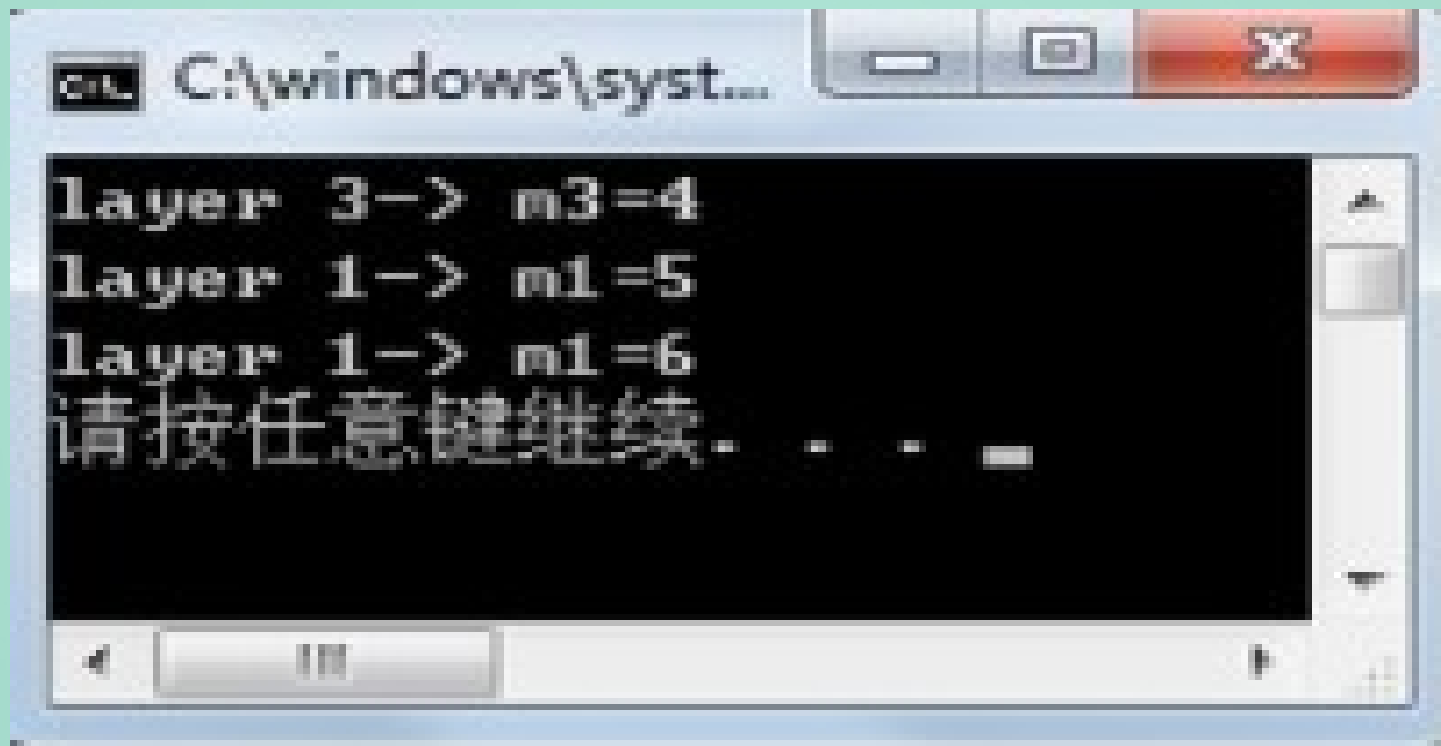
```
class L2_1:public L1
{
public:
    int m2_1;
};
```

```
class L2_2:public L1
{
public:
    int m2_2;
};
```

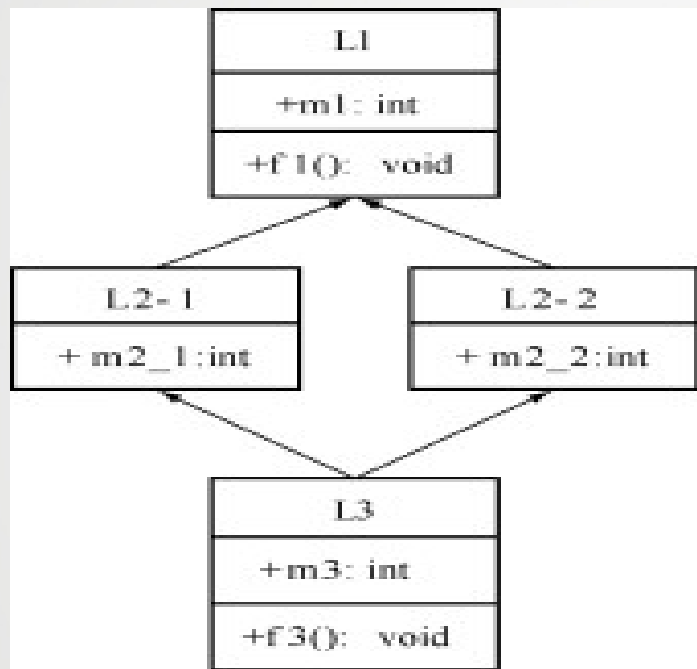
◆ 【例 5-12】 派生类的直接基类全部或部分从另一个共同基类派生而出产生的二义性

```
class L3:public L2_1,public L2_2
{
public:
    int m3;
    void f3() {cout<<"layer 3-> m3="<<m3<<endl;}
};
```

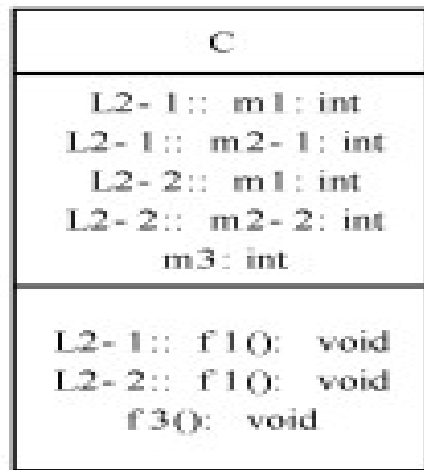
```
int main()
{
    L3 obj;
    obj.m3=4;
    obj.f3();
    obj.L2_1::m1=5; // 正确! 使用直接基类
    obj.L2_1::f1(); // 正确! 使用直接基类
    obj.L2_2::m1=6; // 正确! 使用直接基类
    obj.L2_2::f1(); // 正确! 使用直接基类
    // obj.m1=1;    // 错误! 产生二义性
    // obj.f1();    // 错误! 产生二义性
}
```

```
layer 3-> m3=4  
layer 1-> m1=5  
layer 1-> m1=6  
请按任意键继续. . .
```



(a) UML 图



(b) L3类成员结构

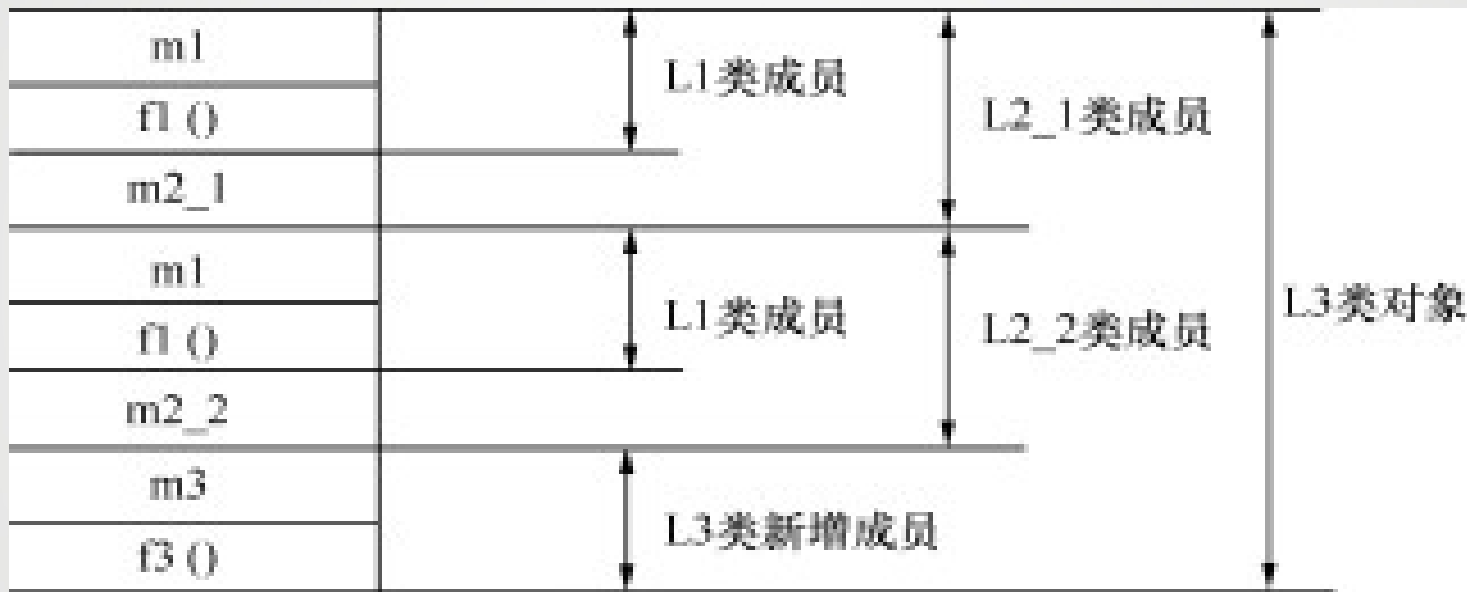
- ◆ 如图所示，间接基类 L1 的成员 m1 和 f1() 经过两次派生之后，通过两个不同途径以相同的名字出现在派生类 L3 中。这时，当通过派生类 L3 的对象或者在派生类 L3 中访问间接基类 L1 的成员时就会产生二义性。那么，对于同名成员 m1 和 f1() 如何进行标识和访问来消除二义性呢？可以使用作用域标识符方法解决，如果用基类名 L1 来限定，无法表明是从类 L2_1 还是类 L2_2 继承过来的。因此，必须使用 L3 的直接基类 L2_1 或者 L2_2 来限定，才能够唯一标识和访问同名成员。

```
L3 obj1;  
obj1.m1;           // 错误! 存在二义性  
obj1.f1();         // 错误! 存在二义性  
obj1.L1::m1;       // 错误! 存在二义性  
obj1.L1::f1();     // 错误! 存在二义性  
obj1.L2_1::m1;     // 正确!  
obj1.L2_1::f1();   // 正确!  
obj1.L2_2::m1;     // 正确!  
obj1.L2_1::f1();
```

◆ 使用作用域标识符存在问题:

◆ 上述使用作用域标识符虽然解决了访问基类 L1 的成员 m1 和 f1() 的二义性问题, 但是派生类 L3 对象在内存中同时拥有基类 L1 的成员 m1 和 f1() 的两份拷贝, 同一成员的多份拷贝增加了内存的开销。

◆ 对于这一问题 C++ 提供了虚基类来解决



04 OPTION 虚基类

◆由上节的例 5-12 可知，当某类的部分或全部直接基类是从另一个共同基类派生而来时，在这些直接基类中从上一级共同基类中继承来的成员就拥有相同的名称。在派生类的对象中，这些同名成员在内存中同时拥有多个副本。虽然可以使用作用域分辨符来唯一标识并访问它们，但增加了内存的开销。为了解决这一问题，C++ 提供了虚基类技术。具体做法是，将共同基类设置为虚基类，这样从不同的途径继承过来的同名成员只有一个副本，这样就不会再会引起二义性问题。

◆ 虚基类一般性声明语法如下：

```
class <派生类名>: virtual <继承方式> <基类名>  
{    //……    };
```

◆ 其中：

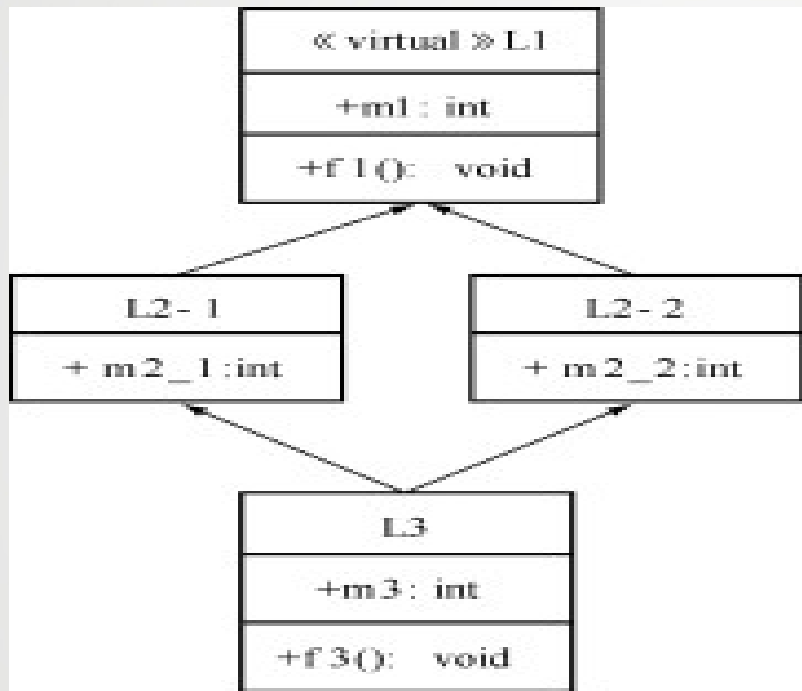
- ◆ virtual 是虚基类的关键字。
- ◆ 在多重继承方式下，虚基类关键字的作用范围只是对紧跟其后的基类起作用。
- ◆ 声明了虚基类后，在进一步派生过程中，虚基类的成员和派生类一起维护同一个内存数据拷贝。
- ◆ 在第一级继承时，就要将共同基类设计为虚基类。

◆ 【例 5-13】虚基类例题

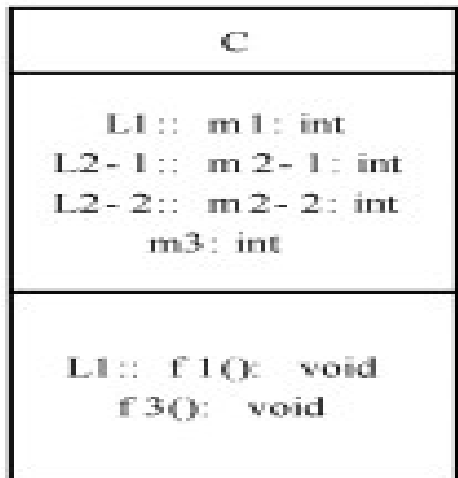
```
#include <iostream>
using namespace std;
class L1
{
public:
    int m1;
    void f1() {cout<<"layer 1-> m1="<<m1<<endl;}
};
class L2_1: virtual public L1 //L1 为虚基类，公有派生 L2_1 类
{
public:
    int m2_1;
};
class L2_2: virtual public L1 //L1 为虚基类，公有派生 L2_2 类
{
public:
    int m2_2;
};
```

◆ 【例 5-13】 虚基类例题

```
class L3:public L2_1,public L2_2 //L3 多重继承
{
public:
    int m3;
    void f3() {cout<<"layer 3-> m3="<<m3<<endl;}
};
```



(a) UML图



(b) L3 类成员结构

```
int main(int argc, char* argv[])
{
    L3 obj;
    obj.m3=4;
    obj.f3();
    obj.m1=5; // 正确! L1 为虚基类, 可以直接使用
    obj.f1(); // 正确!
    return 0;
}
```

◆在例 5-13 中，各类没有构造函数，使用的是编译器自动生成的默认构造函数。

如果虚基类定义有非默认构造函数（如带形参），情况就有所不同。此时，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列表中给出对虚基类的初始化。

◆ 【例 5-14】虚基类构造函数例题

```
#include <iostream>
using namespace std;
class L1
{
public:
    int m1;
    L1(int i){m1=i;cout << "Layer 1 -> m1="<<m1<<endl;} // 类 L1 的构造函数
};
```

◆ 【例 5-14】虚基类构造函数例题

```
class L2_1: virtual public L1    //L1 为虚基类，公有派生 L2_1 类
{
public:
    int m2_1;
    L2_1(int i):L1(i){m2_1=i;cout<<"Layer 2_1 -> m2_1="<<m2_1<<endl;}
    // 类 L2_1 的构造函数，需对虚基类 L1 进行初始化
};
```

◆ 【例 5-14】虚基类构造函数例题

```
class L2_2:virtual public
{
public:
    int m2_2;
    L2_2(int i):L1(i){m2_2=i;cout<<"Layer 2_2 -> m2_2="<<m2_2<<endl;}
};
class L3:public L2_1,public L2_2 {
public:
    int m3;
    L3(int i):L1(i),L2_1(i),L2_2(i)    {m3=i;cout <<"Layer 3-> m3="<<m3<<endl;}
};
int main(int argc, char* argv[])
{
    L3 obj(1);
    return 0;
}
```


◆分析:

- ◆ 在例 5-14 中，虚基类 L1 中的构造函数带有形参，因此从虚基类 L1 中直接继承（类 L2_1、类 L2_2）或间接继承（类 L3）的派生类，其构造函数的成员列表都要列出对虚基类 L1 构造函数的调用。
- ◆ 当观察程序时发现，当生成 L3 类对象时，通过 L3 构造函数，不仅直接调用了虚基类 L1 的构造函数，对从 L1 继承的成员 m1 进行初始化，而且还调用基类 L2_1 和 L2_2 的构造函数。而类 L2_1 和类 L2_2 的构造函数的初始化列表中也有对基类 L1 的初始化。看起来好像整个过程对从虚基类继承来的成员 m1 进行了三次初始化。上述问题，C++ 通过最终派生类的概念很好地解决了。

05



子类型与赋值兼容规则

- ◆ 子类型的概念涉及到行为的共享，它与继承和派生有着紧密的联系。
- ◆ 所谓子类型，是指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。
- ◆ 比如，在公有继承下，派生类是基类的子类型。子类型反映类型之间的一般和特殊的关系，并且子类型关系是不可逆的。

◆ 【例 5-15】公有继承实现子类型例题。

```
#include <iostream>
using namespace std;
class Base
{
public:
    void Print() {cout<< "Base::Print() !"<<endl;}
};
class Derived: public Base
{
public:
    void f() {};
};
void fun( Base& base1)  // 形参为基类 Base 的引用
{
    base1.Print();
}
int main()
{
    Derived derived1;
    fun(derived1);
}
```

- ◆ 类型适应是指两种类型之间的关系。如果类型 B 是类型 A 的子类型，则称类型 B 适应于类型 A，也就是说 B 类型的对象能够用于 A 类型的对象所能使用的场合。
- ◆ 比如，在公有继承方式下，派生类是基类的子类型，派生类必适应于基类，而且派生类的对象是基类的对象。
- ◆ 引入子类型的重要性是为了减轻程序员的编程负担。原因在于一个函数可以适应于某个类型的对象，则它同样也适用于该类型的各个子类型的对象，这样就大可不必为处理这些子类型的对象去重载该函数。

- ◆ 所谓赋值兼容规则就是在公有继承方式下，对于某些场合，一个派生类的对象可以作为基类的对象来使用。也就是说在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。包括以下三种情况：
- ◆ （1）派生类的对象可以赋值给基类对象。如：
`Derived d; Base b; b=d;`
- ◆ （2）派生类的对象可以初始化基类的引用。如：
`Derived d; Base &br=d;`
- ◆ （3）派生类对象的地址可以赋给指向基类的指针。如：
`Derived d; Base *pb=&d;`

◆ 【例 5-16】赋值兼容规则使用情况例题。

```
#include <iostream>
using namespace std;
class Point // 基类 Point
{
protected:
    int x,y;
public:
    Point(int i,int j){x=i;y=j;}
    void show(){cout<<"x="<<x<<" , y="<<y<<endl;}
};
class Rectangle:public Point // 公有派生类 Rectangle
{
private:
    int H,W;
public:
    Rectangle(int i,int j,int m,int n);
    void show1(){cout<<"Error!"<<endl;}
    void show(){cout<<"x="<<x<<" , y="<<y<<" , H="<<H<<" , W="<<W<<endl;}
};
```

◆ 【例 5-16】赋值兼容规则使用情况例题。

```
Rectangle::Rectangle(int i,int j,int m,int n):Point(i, j)
{
    H=m;W=n;
}
int main()
{
    Point p1(1, 2);           // 基类对象 p1
    Rectangle r(3, 4, 5, 6);   // 派生类对象 r
    p1.show();
    r.show();
    Point& br=r;               // 正确！派生类的对象初始化基类的引用
    br.show();                 // 正确！调用基类 show()
    Point *p=&r;               // 正确！派生类对象的地址赋给指向基类的指针
    p->show();                  // 正确！调用基类 show()
    //p->show1();              // 错误！试图调用派生类成员
    Rectangle *pb=&r;          // 正确！派生类指针 pb
    pb->show();                 // 正确！调用派生类 show()
    p1=r;                      // 正确！用派生类对象属性更新基类对象的属性
    p1.show();                 // 正确！调用基类 show()，显示更新后的对象 p1

    //Rectangle *pr=&p1;        // 错误！试图将派生类指针 pr 指向基类对象
}
```

属性值

06

虚基类



多继承层次结构可能很复杂，而且可能会出现二义性问题。为了避免出现基类的两个副本，应使用虚基类。虚基类用在多继承层次结构中，可以避免同一数据成员的不必要重复。

声明虚基类的格式如下：

`class 派生类名 ::virtual 继承方式 基类名`

【例 5.17】虚基类举例

```
#include<iostream>
using namespace std;
class A
{
protected:
    int a;
public:
    A(){a=50;}
    void f(){cout<<"In class A : "<<a<<endl;}
};
```

```
class B:virtual public A{
protected:
    int b;
public:
    B(){b=60;}
    void g() {
        a=10;
        cout<<"In class B : "<<a<<","<<b<<endl;
    }
};

class C:virtual public A{
protected:
    int c;
public:
    C(){c=70;}
    void g() {
        a=20;
        cout<<"In class C : "<<a<<","<<c<<endl;
    }
};
```

```
class D: public B,public C
{
private:
    int d;
public:
    D(){d=80;}
    void g()
    {
        a=30;
        b=40;
        c=50;
        cout<<"In class D : "<<a<<","<<b<<","<<c<<","<<d<<endl;
    }
};

int main()
{
    D d1;
    d1.f();  // 编译正确，没有二义性
    d1.B::g();
}
```

•

```
d1.C::g();  
d1.g();  
return 0;  
}
```

程序运行结果为：

In class A : 50

In class B : 10,60

In class C : 20,70

In class D : 30,40,50,80

使用虚基类后，类 D 对象中只存在一个虚基类 A 成员的副本，故下面的访问是正确的。

```
D d1;  
d1.f();    // 正确
```

定义虚基类就是要保证派生类对象中只有一个虚基类对象，要求虚基类的构造函数只能被调用一次。因此，虚基类的出现改变了构造函数的调用顺序。在初始化任何非虚基类之前，将先初始化虚基类。这时，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列出对虚基类的初始化。如果存在多个虚基类，初始化顺序由它们在继承结构中的位置决定，其顺序是从上到下、从左到右。调用析构函数也遵守相同的规则，但是顺序相反。

【例 5.18】用虚基类改写例 5.10，分析程序的运行结果。

```
#include<iostream>
#include<string>
using namespace std;
class Person
{
public:
```

```
Person(string nna,char nsex,string nphonenum): name(nna),sex(nsex), phonenum
(nphonenum){ }

    void Show()                                // 基类 Show() 函数
    {
        cout<<"name="<<name<<endl;
        cout<<"sex="<<(sex=='m'?" 男 ":" 女 ")<<endl;
        cout<<"phonenum="<<phonenum<<endl;
    }
    ~Person(){ }

private:
    string name;
    char sex;
    string phonenum;
};

class Teacher:virtual public Person    // 虚基类
{
```

public:

```
Teacher(string nna,char nsex,string nphonenum,string ntitle,double nwage):
```

```
Person(nna,nsex,nphonenum)
```

```
{
```

```
    title=ntitle;
```

```
    wage=nwage;
```

```
}
```

```
void Show()
```

//Teacher 类的 Show() 函数

```
{
```

```
    Person::Show();
```

```
    cout<<"title="<<title<<endl;
```

```
    cout<<"wage="<<wage<<endl;
```

```
}
```

```
~Teacher(){ }
```

private:

```
    string title;
```



```
double wage;
};
class Cadre:virtual public Person    // 虚基类
{
public:
    Cadre(string nna,char nsex,string nphonenum,string npost,string npolitical): Person(nna,nsex,nphonenum)
    {
        post=npost;
        political=npolitical;
    }
    void Show()                      //Cadre 类的 Show() 函数
    {
        Person::Show();
        cout<<"post="<<post<<endl;
        cout<<"political="<<political<<endl;
    }
    ~Cadre(){ }
```



- private:
- string post; // 职务
- string political; // 政治面貌
- };
- class Tea_Ca:public Teacher,public Cadre
- {
- public:
- Tea_Ca(string nna,char nsex,string nphonenum,string ntitle,double nwage,string npost,string npolitical):
- Teacher(nna,nsex,nphonenum,ntitle,nwage), Cadre(nna,nsex,nphonenum,npost,npolitical),
- Person(nna,nsex,nphonenum){ } // 由 Tea_Ca 类调用虚基类的构造函数
- void Show() //Tea_Ca 类的 Show() 函数
- {
- Teacher::Show();
- Cadre::Show();
- }
- ~Tea_Ca(){ }
- };

```
int main()
{
    Tea_Ca tc(" 李明 ", 'm', "13189783326", " 教授 ", 5000, " 主任 ", " 党员 "); // 定义派生类对象
    tc.Person::Show();           // 编译正确，从 Person 类间接继承的 Show() 函数唯一
    return 0;
}
```

例 5.18 在主函数中定义了一个派生类 Tea_Ca 的对象 tc，其构造函数和析构函数的执行顺序如下：

- (1) 执行虚基类 Person 的构造函数；
- (2) 执行类 Teacher 和类 Cadre 的构造函数；
- (3) 执行类 Tea_Ca 自己的构造函数；
- (4) 销毁对象 tc 时，调用析构函数，调用析构函数的顺序与调用构造函数的顺序相反。

虽然类 Teacher 和类 Cadre 相对类 Person 来说也是派生类，但因其基类是虚基类，且已经被构造，因此就不再重复调用基类 Person 的构造函数。

虚基类的初始化与一般多继承的初始化在语法上是一样的，但构造函数的执行顺序不同。虚基类及派生类构造函数的执行顺序如下：

- (1) 虚基类的构造函数在所有非虚基类之前执行；
- (2) 若同一层次中包含多个虚基类，这些虚基类的构造函数按它们声明的次序调用；
- (3) 若虚基类由非虚基类派生而来，则仍然先调用基类构造函数，再调用派生类的构造函数。



5.19]

```
#include <iostream>
using namespace std;
class A1                                // 声明基类 A1
{
public:
    A1(){cout<<"A1 类默认构造函数 ;"<<endl;}
    ~A1(){cout<<"A1 类析构函数 ;"<<endl;}
    void Print(){cout<<" 在 A1 中 ;"<<endl;}
};
class A2:public A1                       // 声明基类 A2
{
public:
    A2(int i){a=i;cout<<"A2 类构造函数 ; a="<<a<<endl;}
    ~A2(){cout<<"A2 类析构函数 ; a="<<a<<endl;}
private:
    int a;
};
```

```
class B1:virtual public A2           //A2 为虚基类， 派生类 B1
{
public:
    B1(int i,int j):A2(i){b1=j;cout<<"B1 类构造函数 ; b1="<<b1<<endl;}
    ~B1(){cout<<"B1 类析构函数 ; b1="<<b1<<endl;}
private:
    int b1;
};

class B2:virtual public A2           //A2 为虚基类， 派生类 B2
{
public:
    B2(int i,int j):A2(i){b2=j;cout<<"B2 类构造函数 ; b2="<<b2<<endl;}
    ~B2(){cout<<"B2 类析构函数 ; b2="<<b2<<endl;}
private:
    int b2;
};

class C:public B1,public B2          // 声明派生类 C
```

```
{
public:
    C(int i,int j,int k,int t):A2(i),B1(i,j),B2(i,k)
    {
        c=t;
        cout<<"C 类构造函数 ; c="<<c<<endl;
    }
    ~C(){cout<<"C 类析构函数 ; c="<<c<<endl;}
private:
    int c;
};

int main()
{
    C c1(1,2,3,4);
    c1.Print();
    return 0;
}
```

程序运行结果为：

A1 类省略构造函数；
A2 类构造函数； a=1
B1 类构造函数； b1=2
B2 类构造函数； b2=3
C 类构造函数； c=4
在 A1 中；
C 类析构函数； c=4
B2 类析构函数； b2=3
B1 类析构函数； b1=2
A2 类析构函数； a=1
A1 类析构函数；



虚基类：虚基类的应用【例5.20】

128

```
#include<iostream>
#include<string>
using namespace std;
class people
{
public:
    people(char *n="",char *i="",char s='m',int a=19);
    void Pdisplay();
private:
    char name[20];
    char ID[20];
    char sex;
    int age;
};
people::people(char *n,char *i,char s,int a)
{
```



```
strcpy(name,n);
strcpy(ID,i);
sex=s;
age=a;
}
void people::Pdisplay()
{
    cout<<" 人员： \n 身份证号 ---"<<ID<<endl;
    cout<<" 姓名 ---"<<name<<endl;
    if(sex=='m' || sex=='M')cout<<" 性别 ---"<<" 男 "<<endl;
    if(sex=='f' || sex=='F')cout<<" 性别 ---"<<" 女 "<<endl;
    cout<<" 年龄 ---"<<age<<endl;
}
class job:virtual public people
{
public:
```

```
job(char *n,char *i,char s,int a,int num=0,char *dep="");
void Jdisplay();
private:
    int number;                // 工作证号
    char department[20];       // 工作部门
};
job::job(char *n,char *i,char s,int a,int num,char *dep):people(n,i,s,a)
{
    number=num;
    strcpy(department,dep);
}
void job::Jdisplay()
{
    cout<<" 工作人员： "<<endl;
    cout<<" 编号  ---"<<number<<endl;
    cout<<" 工作单位  ---"<<department<<endl;
}
```

```
class student: virtual public people
```

```
{
```

```
public:
```

```
    student(char *n,char *i,char s,int a,int sn=0,int cn=0):people(n,i,s,a)
```

```
    {
```

```
        snum=sn;
```

```
        classnum=cn;
```

```
    }
```

```
    void Sdisplay()
```

```
    {
```

```
        cout<<" 在校学生 "<<endl;
```

```
        cout<<" 学号 ="<<snum<<endl;
```

```
        cout<<" 班级 ="<<classnum<<endl;
```

```
    }
```

```
private:
```

```
    int snum;
```

```
    int classnum;
```

```
};
```

```
class job_student:public job,public student
{
public:
    job_student(char *n,char *i,char s='m',int a=19,int mn=0,char *md="",int no=0,int
sta=1):job(n,i,s,a,mn,md),student(n,i,s,a,no,sta),people(n,i,s,a){ }
    void Tdisplay();
};
void job_student::Tdisplay()
{
    cout<<" 在职学生 "<<endl;
}
int main()
{
    job_student w(" 张国媛 ","122334571908655",'f',22,102," 民族学院 ",5282,2004);
    w.Tdisplay();
    w.Pdisplay();
}
```

- `w.Jdisplay();`
 - `w.Sdisplay();`
 - `return 0;`
 - `}`
-
- 例中首先设计基类 `people`，表示一般人员的信息
 - ，再设计一个表示工作人员的类 `job`，接下来设计一个表示学生的类 `student`，在职学生类 `job_student` 以
 - 这些类为基类。

程序运行结果为：

在职学生
人员：
身份证号 ---
122334571908655
姓名 --- 张国媛
性别 --- 女
年龄 ---22
工作人员：
编号 ---102
工作单位 --- 民族学院
在校学生
学号 =5282
班级 =2004 数；

07



基类和派生类的转换

- 基类和派生类之间也能进行类似的类型转换。由于派生类包含从基类继承的成员，因此可以将派生类对象赋值给基类对象，反之不能。具体表现为下面 3 种形式。

1. 派生类的对象可以赋值给基类的对象

用派生类对象给基类对象赋值时，派生类中从基类继承的数据成员对应赋值给基类的数据成员，派生类自己新增的数据成员则舍弃。赋值只是对数据成员，成员函数是不存在赋值的。只能用派生类对象对基类对象赋值，不能用基类对象对派生类对象赋值，原因是基类对象不包含派生类的数据成员，所以无法对派生类对象赋值。例如：

```
Base b;           // 定义基类对象
Derived d;        // 定义派生类对象
b=d;              // 正确，派生类对象给基类对象赋值
d=b;              // 错误！
```

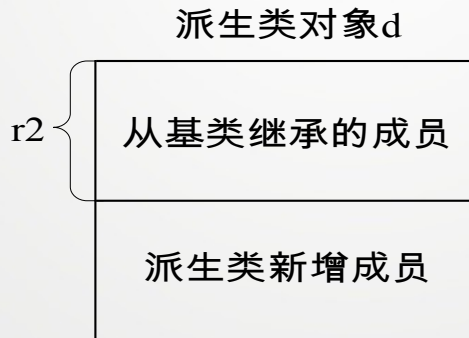
在公有继承下，派生类的对象可作为基类的对象使用，但只能使用从基类继承的成员。

2. 派生类的对象可以初始化基类对象的引用

如果定义了基类对象的引用，可以用基类对象初始化，也可以用派生类对象初始化。如：

```
Base b;      // 定义基类对象
Derived d;   // 定义派生类对象
Base &r1=b;   // 定义基类对象的引用 r1，用基类对象 b 初始化
Base &r2=d;   // 定义基类对象的引用 r2，用派生类对象 d 初始化
```

基类对象的引用 r1 是 b 的别名，b 和 r1 共享同一存储空间。但 r2 并不是 d 的别名，它只是 d 中基类部分的别名，r2 与 d 中的基类部分共享同一存储空间。



3. 派生类对象的地址可以赋给基类指针变量

若定义了指向基类对象的指针变量，也可以用派生类的对象取地址给它赋值，即指向基类对象的指针变量可以指向派生类对象。例如：

```
Base b;           // 定义基类对象
Derived d;        // 定义派生类对象
Base *p=&d;       // 定义基类的指针 p，指向派生类的对象 d
```

以上的表现形式同样可用在函数参数中。如果函数的形参是基类的对象、基类对象的引用或是指向基类对象的指针变量，那么实参可以是一个派生类的对象。

【例 5.21】分析下列程序的运行结果。

```
#include<iostream>
#include<string>
using namespace std;
class Person
{
public:
    Person(string nna,char nsex,string nphonenum):name(nna),sex(nsex),
phonenum(nphonenum){ }
    void Show()
    {
        cout<<"name="<<name<<endl;
        cout<<"sex="<<(sex=='m'?" 男 ":" 女 ")<<endl;
        cout<<"phonenum="<<phonenum<<endl;
    }
private:
    string name;
```

```
        char sex;
        string phonenum;
};
class Teacher:public Person
{
public:
    Teacher(string nna,char nsex,string nphonenum,string ntitle,double nwage):
    Person(nna,nsex,nphonenum)
    {
        title=ntitle;
        wage=nwage;
    }
    void Show()
    {
        Person::Show();
        cout<<"title="<<title<<endl;
        cout<<"wage="<<wage<<endl;
    }
}
```

private:

```
    string title;  
    double wage;
```

```
};
```

```
int main()
```

```
{
```

```
    Teacher t(" 李明 ", 'm', "13189783326", " 教授 ", 5000);
```

```
    Person *p=&t; // 定义了指向基类对象的指针，用派生类对象的地址给它赋值
```

```
    p->Show();
```

```
    return 0;
```

```
}
```

08

程序实例



- ◆ 【例 5-22】继承与派生例题。
- ◆ 编写一个程序，有一个汽车类 `vehicle`，它具有带参数的构造函数，类中的数据成员为车轮个数 `wheels` 和车重 `weight` 放在保护段中；小车类 `car` 是汽车类 `vehicle` 私有派生类，其中包含载客人数 `passenger_load`；卡车类 `truck` 是汽车类 `vehicle` 私有派生类，其中包含载客人数 `passenger_load` 和载重量 `payload`。

```
#include <iostream>
using namespace std;
class vehicle // 汽车类
{
protected:
    int wheels;
    float weight;
public:
    vehicle(int input_wheels, float input_weight); // 汽车类, 构造函数
    int get_wheels(); //
    float get_weight();
    float wheel_load();
    void print();
};
```



```
class car:private vehicle // 私有派生小汽车
{
private:
    int passenger_load;
public:
    car(int input_wheels,float input_weight,int
input_passenger_load=4); // 小汽车构造函数
    int get_passenger_load();
    void print();
};
```

```
class truck:private vehicle // 私有派生卡车类
{
private:
    int passenger_load;
    float payload;
public:
    truck(int input_wheels,float input_weight,int
input_passenger_load=2,
float input_payload=320000);
// 卡车类构造函数
    int get_passenger_load();
    float efficiency();    // 计算卡车效率
    void print();
};
```

```
vehicle::vehicle(int input_wheels, float input_weight)
{
    wheels=input_wheels;
    weight=input_weight;
}
int vehicle::get_wheels()
{
    return wheels;
}
float vehicle::get_weight()
{
    return weight/wheels;
}
```

```
void vehicle::print()
{
    cout<<" 车轮: "<<wheels<<" 个 "<<endl;
    cout<<" 重量:  "<<weight<<" 公斤 "<<endl;
}

car::car(int input_wheels, float input_weight, int
input_passenger_load)
:vehicle(input_wheels, input_weight)
{
    passenger_load=input_passenger_load;
}

int car::get_passenger_load()
{
    return passenger_load;
}
```

```
void car::print()
{
    cout<<" 小车: " <<endl;
    vehicle::print();
    cout<<" 载人: " <<passenger_load<<" 人 " <<endl;
    cout<<endl;
}

truck::truck(int input_wheels, float input_weight, int input_passenger_load, float
input_payload)
:vehicle(input_wheels, input_weight)
{
    passenger_load=input_passenger_load;
    payload=input_payload;
}

int truck::get_passenger_load()
{
    return passenger_load;
}
```

```
float truck::efficiency()
{
    return payload/(payload+weight);
}
void truck::print()
{
    cout<<" 卡车: "<<endl;
    vehicle::print();
    cout<<" 载人: "<<passenger_load<<" 人 "<<endl;
    cout<<" 载重量: "<<payload<<" 公斤 "<<endl;
    cout<<" 效率: " (载重量 / (载重量 + 车
重) ) : "<<efficiency()*100<<"%"<<endl;
    cout<<endl;
}
```

```
int main()
{
    car car1(4., 900, 5);
    truck truck1(8, 10000, 3, 300000);
    car1.print();
    truck1.print();
    return 0;
}
```

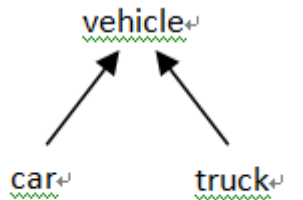


图 5-20 继承关系

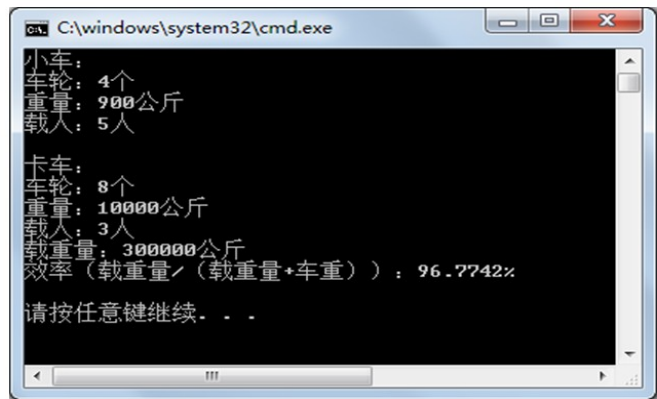


图5-19 例5-17程序运行结果