第5章

继承与派生

继承是面向对象程序设计的第二个重要特性，通过继承实现了数据抽象基础上的代码重用。继承的作用是减少代码冗余，通过协调来减少接口和界面。

继承是面向对象程序设计的关键。它的好处非常多，最重要的有以下两点。

(1) 抽取对象类之间的共同点，消除冗余。仅仅用处于同一层次的类来构建软件是不可取的，实际上很多类之间都存在共同点，忽略这些共同点将会带来很大的冗余。

(2) 继承带来了软件的复用。用已经实现的类为基类，派生出新的类，可以做到“站在巨人的肩膀上”，能快速开发出高质量的程序。

继承反映了类的层次结构，并支持对事物从一般到特殊的描述。继承使得程序员可以以一个已有的较一般的类为基础建立一个新类，而不必从零开始设计。建立一个新的类，可以从一个或多个先前定义的类中继承数据成员和成员函数，而且可以重新定义或加进新的数据成员和成员函数，从而建立了类的层次或等级，这个新类称为派生类或子类，而先前有的类称为基类、超类或父类。

在C++语言中，有两种继承方式：单一继承(参见图1.3单一继承示例)和多重继承(参见图1.4多重继承示例)。对于单一继承，派生类只能有一个直接基类；对于多重继承，派生类可以有多个直接基类。

5.1 单一继承

5.1.1 继承与派生的概念

C++语言的重要目标之一是代码重用。为了实现这个目标，C++语言采用两种方法：对象成员和继承。在面向对象的程序设计中，大量使用继承和派生。类的继承实际上是一种演化、发展过程，即通过扩展、更改和特殊化，从一个已知类出发建立一个新类。通过类的派生可以建立具有共同关键特征的对象家族，从而实现代码重用。这种继承和派生的机制对于已有程序的发展和改进是极为有利的。

派生类同样也可以作为基类再派生新的类，这样就形成了类的层次结构。类的继承和派生的层次结构，可以说是人们对自然界中的事物进行分类、分析和认识的过程在程序设计中的体现。现实世界中的事物都是相互联系、相互作用的，人们在认识自然的过程中，根据事物的实际特征，抓住其共同特性和细小差别，利用分类的方法分析和描述这些实体或概念之间的相似点和不同点。

派生类具有如下特点。

(1) 新的类可在基类的基础上包含新的成员。

(2) 新的类中可隐藏基类的成员函数。

(3) 可为新类重新定义成员函数。

基类与派生类的关系如下。

(1) 派生类是基类的具体化。

(2) 派生类是基类定义的延续。

(3) 派生类是基类的组合。

5.1.2 派生类的定义

在 C++语言中,派生类的定义如下:

```
class 派生类名:[继承方式]基类名
{
// 派生类成员声明
};
```

其中:

(1) 派生类名是新生成类的类名。

(2) 继承方式规定了如何访问从基类继承的成员。继承方式关键字为 private、public 和 protected,分别表示私有继承、公有继承和保护继承,默认情况下是私有(private)继承。类的继承方式指定了派生类成员以及类外对象对于从基类继承来的成员的访问权限,将在 5.1.3 节中详细介绍。

(3) 派生类成员除了包括从基类继承来的所有成员之外,还包括新增加的数据成员和成员函数。这些新增加的成员正是派生类不同于基类的关键所在,是派生类对基类的发展。重用和扩充已有的代码,就是通过在派生类中新增成员来添加新的属性和功能的。

例如,定义如下的汽车类及其派生类:小汽车类和卡车类。

```
class Vehicle                                          //定义基类 Vehicle
{
public:                                                //公有成员函数
    void init_Vehicle(int in_wheels,float in_weight);  //给数据成员初始化
    int  get_wheels();                                 //取车轮数
    float get_weight();                                //取汽车质量
    float wheelloading();                              //车轮承重
private:                                               //私有数据成员
    int wheels;                                        //车轮数
    float weight;                                      //表示汽车载重
};
```

在基类 Vehicle 的基础上,定义了如下的派生类 Car 和 Truck,在派生类中新增了一些数据成员和成员函数。

```
class Car:public Vehicle                               //定义派生类 Car
{
public:                                                //新增公有成员函数
    void intitialize(int in_wheels,float in_weight,int people = 4);
    int passengers();
```

```
private:
    int passenger_load;                                    //新增私有数据成员,表示载客数
};

class Truck:public Vehicle                                 //定义派生类 Truck
{
public:                                                    //新增公有成员函数
    void init_Truck(int,float);
    int passengers();
    float weight_loads();
private:                                                   //新增私有数据成员
    int passenger_load;
    float weight_load;
};
```

在C++语言程序设计中,进行派生类的定义,给出该类的成员函数的实现之后,整个类就定义好了,这时就可以由它来生成对象,进行实际问题的处理。派生新类的过程,经历了三个步骤:吸收基类成员、改造基类成员和添加新的成员,下面分别加以介绍。

1. 吸收基类成员

面向对象的继承和派生机制,其最主要的目的是实现代码的重用和扩充。吸收基类成员就是一个重用的过程,而对基类成员进行调整、改造以及添加新成员就是原有代码的扩充过程,二者是相辅相成的。

C++语言的类继承,首先是基类成员的全盘吸收,这样,派生类实际上就包含了它的所有基类的除构造函数和析构函数之外的所有成员。很多基类的成员,特别是非直接基类的成员,尽管在派生类中很可能根本不起作用,也被继承下来,在生成对象时要占据一定的内存空间,造成资源浪费,要对其进行改造。

2. 改造基类成员

对基类成员的改造包括两方面,一是依靠派生类的继承方式来控制基类成员的访问,二是对基类数据成员或成员函数的覆盖,即在派生类中定义一个和基类数据成员或成员函数同名的成员,由于作用域不同,产生成员覆盖(member overridden,又叫同名覆盖,即当一个已在基类中声明的成员名又在派生类中重新声明所产生的效果),基类中的成员就被替换成派生类中的同名成员。

成员覆盖使得派生类的成员掩盖了从基类继承得到的同名成员。这种掩盖既不是成员的丢失,也不是成员的重载。因为经类作用域声明后仍可引用基类的同名成员函数,而且可以由派生类的成员函数去引用从基类继承来的同名成员(参见例5.6),这是重载所没有的效果。基于这一成员覆盖的机制,可以充分体现派生的优越性。那就是派生类可以在继承基类的基础上继续扩充原设计中未考虑到的内容,从而使一个软件系统的生命周期大大地延长,这便是可重用软件设计思想的最终目标。

3. 添加新的成员

继承与派生机制的核心是在派生类中加入新的成员,程序员可以根据实际情况的需要,给派生类添加适当的数据成员和成员函数,来实现必要的新功能。在派生的过程中,基类的构造函数和析构函数是不能被继承下来的。同时,在派生类中,一些特殊的初始化和扫尾清理工作,也需要重新定义新的构造函数和析构函数。

5.1.3 类的继承方式

在面向对象程序中,基类的成员可以有 public(公有)、protected(保护)和 private(私有)三种访问类型。在基类内部,自身成员可以对任何一个其他成员进行访问,但是通过基类的对象,就只能访问基类的公有成员。

派生类继承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,但是这些成员的访问属性在派生的过程中是可以调整的。从基类继承的成员,其访问属性由继承方式控制。

类的继承方式有 public(公有)继承、protected(保护)继承和 private(私有)继承三种。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有所变化。表 5.1 列出了不同继承方式下基类成员访问属性的变化情况。

表 5.1 不同继承方式下基类成员的访问属性

继承方式	访问属性		
	public	protected	private
public	public	protected	不可访问的
protected	protected	protected	不可访问的
private	private	private	不可访问的

说明:

表 5.1 中第一列给出三种继承方式,第一行给出基类成员的三种访问属性,其余单元格内容为基类成员在派生类中的访问属性。

从表中可以看出以下几点:

(1) 基类的私有成员在派生类中均是不可访问的,它只能由基类的成员访问。

(2) 在公有继承方式下,基类中的公有成员和保护成员在派生类中的访问属性不变。

(3) 在保护继承方式下,基类中的公有成员和保护成员在派生类中均为保护的。

(4) 在私有继承方式下,基类中的公有成员和保护成员在派生类中均为私有的。

注意:

保护成员与私有成员唯一的不同是当发生派生后,基类的保护成员可被派生类直接访问,而私有成员在派生类中是不可访问的。在同一类中私有成员和保护成员的用法完全一样。

1. 公有继承

公有继承方式创建的派生类对基类各种成员的访问权限如下。

(1) 基类公有成员相当于派生类的公有成员,即派生类可以像访问自身公有成员一样访问从基类继承的公有成员。

(2) 基类保护成员相当于派生类的保护成员,即派生类可以像访问自身的保护成员一样,访问基类的保护成员。

(3) 基类的私有成员,派生类内部成员无法直接访问。派生类使用者也无法通过派生类对象直接访问。

【例 5.1】 公有继承示例。

从基类 Vehicle(汽车)公有派生 Car(小汽车)类,Car 类继承了 Vehicle 类的全部特征,

同时，Car类自身也有一些特点，这就需要在继承Vehicle类时添加新的成员。

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

class Vehicle                                    //基类 Vehicle 类的定义
{
public:                                          //公有成员函数
    Vehicle(int in_wheels,float in_weight)
    {
        wheels = in_wheels;weight = in_weight;
    }
    int get_wheels()
    {
        return wheels;
    }
    float get_weight()
    {
        return weight;
    }
private:                                         //私有数据成员
    float weight;
    int wheels;
};

class Car:public Vehicle                         //派生类 Car 类的定义
{
public:                                          //新增公有成员函数
    Car(int in_wheel,float in_weight,int people = 5):Vehicle(in_wheel,in_weight)
    {
        passenger_load = people;
    }
    int get_passengers()
    {
        return passenger_load;
    }
private:                                         //新增私有数据成员
    int passenger_load;
};

int main()
{
    Car car1(4,1000);                            //声明 Car 类的对象
    cout <<"The message of car1(wheels,weight,passengers):"<< endl;
    cout << car1.get_wheels()<<",";              //访问派生类从基类继承来的公有函数
    cout << car1.get_weight()<<",";              //访问派生类从基类继承来的公有函数
    cout << car1.get_passengers()<< endl;        //访问派生类的公有函数

    return 0;
}
```

程序的运行结果如图 5.1 所示。

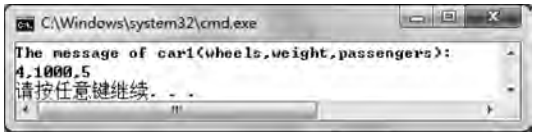

图 5.1　例 5.1 的运行结果

这里首先声明了基类 Vehicle。派生类 Car 继承了 Vehicle 类的全部成员(构造函数和析构函数除外)。因此,派生类实际所拥有的成员就是从基类继承过来的成员与派生类新声明的成员的总和。继承方式为公有继承,这时基类中的公有成员在派生类中的访问属性保持不变,派生类的成员函数及派生类对象可以访问基类的公有成员,但是无法访问基类的私有数据(例如基类的 wheels 和 weight)。

基类原有的外部接口(如基类的 get_wheels()函数和 get_weight()函数)变成了派生类外部接口的一部分。当然,派生类自己新增的成员之间都是可以互相访问的。

Car 类继承了 Vehicle 类的成员,也就实现了代码的重用。同时,通过新增成员,加入了自身的独有特征,实现了程序的扩充。

2. 私有继承

派生类对基类各种成员访问权限如下。

(1) 基类公有成员和保护成员都相当于派生类的私有成员,派生类只能通过自身的成员函数访问它们。

(2) 基类的私有成员,无论派生类内部成员或派生类的对象都无法直接访问。

【例 5.2】 私有继承示例。

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

class Vehicle                                    //基类 Vehicle 类的定义
{
public:                                          //公有成员函数
    Vehicle(int in_wheels,float in_weight)
    {
        wheels = in_wheels;weight = in_weight;
    }
    int get_wheels()
    {
        return wheels;
    }
    float get_weight()
    {
        return weight;
    }
private:                                         //私有数据成员
    int wheels;
    float weight;
```

```
};

class Car:private Vehicle                          //定义派生类 Car 类
{
public:                                            //新增公有成员函数
    Car(int in_wheels,float in_weight,int people = 5):Vehicle(in_wheels,in_weight)
    {
        passenger_load = people;
    }
    int get_wheels()                               //重新定义 get_wheels()
    {
        return Vehicle::get_wheels();
    }
    float get_weight()                             //重新定义 get_weight()
    {
        return Vehicle::get_weight();
    }
    int get_passengers()
    {
        return passenger_load;
    }
private:                                           //新增私有数据成员
    int passenger_load;
};

int main()
{
    Car car1(4,1000);                              //定义 Car 类对象
    cout <<"The message of car1(wheels,weight,passengers):"<< endl;
    cout << car1.get_wheels()<<","                 //输出小汽车的信息
        << car1.get_weight()<<","
        << car1.get_passengers()<< endl;

    return 0;
}
```

程序的运行结果同图 5.1。

在私有继承情况下，为了保证基类的部分外部接口特征保留在派生类中，就必须在派生类中重新定义同名的成员函数。

同例 5.1 相比较，本例对程序修改的只是派生类的内容，基类和主函数部分没有做任何改动。由此可以看到面向对象程序设计封装性的优越性：Car 类的外部接口不变，内部成员的实现做了改造，根本没有影响到程序的其他部分，这正是面向对象程序设计可重用性与可扩充性的一个实际体现。

3. 保护继承

保护继承方式创建的派生类对基类各种成员访问权限如下。

(1) 基类公有成员和保护成员都相当于派生类的保护成员，派生类可以通过自身的成员函数或其子类的成员函数访问它们。

(2) 基类的私有成员,无论派生类内部成员或派生类的对象都无法直接访问。

【例 5.3】 保护继承示例。

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

class Vehicle                                    //定义基类 Vehicle
{
public:                                          //公有成员函数
    Vehicle(int in_wheels,float in_weight)
    {
        wheels = in_wheels;weight = in_weight;
    }
    int get_wheels()
    {
        return wheels;
    }
    float get_weight()
    {
        return weight;
    }
private:                                         //私有数据成员
    int wheels;
protected:                                       //保护数据成员
    float weight;
};

class Car:protected Vehicle                      //定义派生类 Car
{
public:                                          //新增公有成员函数
    Car(int in_wheels,float in_weight,int people = 5):Vehicle(in_wheels,in_weight)
    {
        passenger_load = people;
    }
    int get_wheels()                             //重新定义 get_wheels()
    {
        return Vehicle::get_wheels();
    }
    float get_weight()                           //重新定义 get_weight()
    {
        return weight;
    }
    int get_passengers()
    {
        return passenger_load;
    }
private:                                         //新增私有数据成员
    int passenger_load;
};
```

```
int main()
{
    Car car1(4,1000);                                    //定义 Car 类的对象
    cout <<"The message of car1(wheels,weight,passengers):"<< endl;
    cout << car1.get_wheels()<<","                       //输出小汽车的信息
        << car1.get_weight()<<","
        << car1.get_passengers()<< endl;

    return 0;
}
```

程序的运行结果同图 5.1。

在保护继承情况下，为了保证基类的部分外部接口特征保留在派生类中，就必须在派生类中重新定义同名的成员函数。根据同名覆盖的原则，在主函数中调用的是派生类的成员函数。

总之，不论是哪种继承方式，派生关系具有下述特征。

(1) 派生类没有独立性，即派生类不能脱离基类而独立存在。

(2) 派生类对其所继承的基类成员的可访问程度因继承方式的不同而不同。

(3) 无论派生类能否直接访问其所继承的基类成员，基类成员都是派生类成员。

5.1.4 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不能被继承，在派生类中，如果对派生类新增的成员进行初始化，就必须加入新的构造函数，与此同时，对所有从基类继承来的成员的初始化工作，还是应由基类的构造函数完成，因此必须在派生类中对基类的构造函数所需要的参数进行设置。同样，对派生类对象的扫尾、清理工作也需要编写析构函数。

1. 派生类的构造函数

在下面两种情况下，必须定义派生类的构造函数：一是派生类本身需要构造函数，二是在定义派生类对象时，其相应的基类对象需调用带有参数的构造函数。

派生类对象的初始化也是通过派生类的构造函数实现的。具体来说，就是为该类的数据成员赋初值。派生类的数据成员由所有基类的数据成员与派生类新增的数据成员共同组成，如果派生类新增成员中包括有内嵌的其他类对象，派生类的数据成员中实际上还间接包括了这些对象的数据成员。

因此，初始化派生类对象，就要对基类数据成员、新增数据成员和对象成员的数据成员进行初始化。派生类的构造函数需要以合适的初值作为参数，隐含调用基类和新增的内嵌对象成员的构造函数来初始化它们各自的数据成员，然后再加入新的语句对新增普通数据成员进行初始化。

派生类构造函数声明的一般语法形式如下所示。

```
派生类构造函数(参数表):基类构造函数(参数表),对象成员 1(参数表),…,对象成员 n(参数表)
{
    派生类新增成员的初始化语句;
}
```

其中：

(1) 派生类的构造函数名与派生类名相同。

(2) 参数表需要列出初始化基类数据、新增内嵌对象数据及新增一般数据成员所需要的全部参数。

(3) 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名及各自的参数表，各项用逗号分隔开。

在定义派生类对象时构造函数的执行顺序是先祖先(基类，调用顺序按照它们继承时说明的顺序)，再客人(对象成员，调用顺序按照它们在类中说明的顺序)，后自己(派生类本身)。

【例 5.4】 派生类构造函数示例。

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

class ST_COM
{
public:
    ST_COM(char * na, unsigned int n, float ma, float en, float ph): num(n),math(ma),english
(en),physics(ph)
    {
        strcpy_s(name,na);
    }
protected:
    char name[10];
    unsigned int num;
    float mathematics,English,Physics;
};

class EL_DEP:public ST_COM
{
public:
    EL_DEP(char * na,unsigned int n,float ma,float en,float ph,float pe,float el,float d): ST_
COM(na,n,ma,en,ph), exchange(pe), elnet(el),dst(d)
    { }
    void show()
    {
        cout <<"Name:"<< name <<"          Number:"<< num << endl;
        cout <<"Mathematics Score:"<< math << endl;
        cout <<"English Score      :"<< english << endl;
        cout <<"Physics Score      :"<< physics << endl;
        cout <<"Exchange Score     :"<< exchange << endl;
        cout <<"Elec_net Score     :"<< elnet << endl;
        cout <<"Data_structure Score :"<< dst << endl;
    }
private:
    float exchange,elnet,dst;
};
```

```
int main()
{
    EL_DEP aStudent("LiQiang",1,71,72,73,81,82,83);
    aStudent.show();

    return 0;
}
```

程序的运行结果如图 5.2 所示。

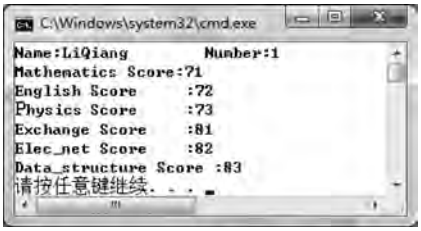

图 5.2 例 5.4 的运行结果

【程序解析】

本例中 ST_COM 类是学生公共课(数学、英语、物理)的数据结构。EL_DEP 可以代表某系所开课(程控交换、电信网络、数据结构)的数据结构。EL_DEP 类继承了 ST_COM 类的成员而成为其派生类。在定义 EL_DEP 类的对象时初始化了基类的全部数据成员。

【例 5.5】 构造函数的调用顺序。

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

class Data
{
public:
    Data(int x)
    {
        Data::x = x;
        cout <<"class Data\n";
    }
private:
    int x;
};

class A
{
public:
    A(int x):d1(x)
    {
        cout <<"class A\n";
    }
private:
    Data d1;
};

class B:public A
{
public:
```

```
    B(int x):A(x),d2(x)
    {
        cout <<"class B\n";
    }
private:
    Data d2;
};

class C:public B
{
public:
    C(int x):B(x)
    {
        cout <<"class C\n";
    }
};

int main()
{
    C object(5);

    return 0;
}
```

程序的运行结果如图 5.3 所示。

图 5.3 例 5.5 的运行结果

从程序运行的结果可以看出,构造函数的调用严格地按照先祖先、再客人、后自己的顺序执行。

【例 5.6】 类派生引出的成员覆盖的示例。

在例 5.4 的两个类中各插入一个同名的显示函数和用于记录平均成绩的数据成员 avg 来观察成员覆盖后的情形。

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

class ST_COM
{
public:
    ST_COM(char *na, unsigned int n, float ma, float en, float ph): num(n),math(ma),english
(en),physics(ph)
    {
        strcpy_s(name,na);
        avg = (math + english + physics)/3;
    }
    void show()
    {
        cout <<"Name:"<< name <<"          Number:"<< num << endl;
        cout <<"Mathematics Score:"<< math << endl;
```

```
        cout <<"English Score      :"<< english << endl;
        cout <<"Physics Score      :"<< physics << endl;
    }
protected:
    char name[10];
    unsigned int num;
    float math,english,physics,avg;
};

class EL_DEP:public ST_COM
{
public:
    EL_DEP(char * na,unsigned int n,float ma,float en,float ph,float pe,float el,float d): ST_
COM(na,n,ma,en,ph), exchange(pe), elnet(el),dst(d)
    {
        avg = ((exchange + elnet + dst)/3 + ST_COM::avg)/2;
    }
    void show()
    {
        ST_COM::show();
        cout <<"Exchange Score     :"<< exchange << endl;
        cout <<"Elec_net Score     :"<< elnet << endl;
        cout <<"Data_struct Score  :"<< dst << endl;
        cout <<"Average Score      :"<< avg << endl;
    }
private:
    float exchange,elnet,dst;
};

int main()
{
    EL_DEP aStudent("LiQiang",1,71,72,73,81,82,83);
    aStudent.show();

    return 0;
}
```

程序的运行结果如图 5.4 所示。

通过以上几个例子,可以得到以下结论。

(1) 当基类构造函数不带参数时,派生类不一定需要定义构造函数,然而当基类的构造函数哪怕只带有一个参数时,派生类都必须定义构造函数,甚至所定义的派生类构造函数的函数体可能为空,仅仅起传递参数的作用。

(2) 如果派生类的基类也是一个派生类,每个派生类只需负责其直接基类的构造,依次上溯。

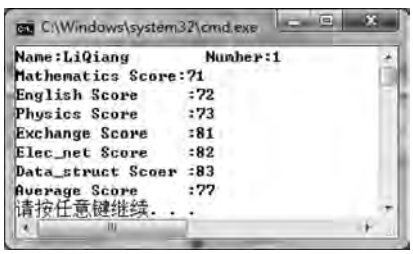

图 5.4 例 5.6 的运行结果

(3) 构造函数的执行顺序：在调用派生类的构造函数时将优先调用声明在成员初始化表内的基类构造函数，也就是先初始化由基类派生来的成员，然后再执行自身的构造函数。即使有意把调用的基类构造函数部分写在一系列的初始化表的最后面也不会改变这种调用顺序。

(4) 默认调用关系：如在派生类构造函数的成员初始化表中没有指明要调用的基类构造函数，则一定会调用基类的无参构造函数(若类没有无参构造函数，则调用默认构造函数)。另一方面，当然不能在派生类的成员初始化表中去无中生有地调用不存在的基类构造函数，甚至是其他属于基类的成员函数。

2. 派生类的析构函数

派生类析构函数与基类的析构函数没有什么联系，彼此独立，它们只做各自类对象消亡前的善后工作。派生类析构函数的功能与没有继承关系的类中析构函数的功能一样，也是在对象消亡之前进行一些必要的清理工作。在派生过程中，基类的析构函数不能继承，如果需要析构函数的话，就要在派生类中重新定义。析构函数没有类型，也没有参数，如果没有显式定义过某个类的析构函数，系统会自动生成一个默认的析构函数，完成清理工作。

派生类析构函数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，只要在函数体中负责把派生类新增的非对象成员的数据成员的清理工作做好就够了，系统会自己调用基类及对象成员的析构函数来对基类及对象成员进行清理。

析构函数的执行顺序和构造函数正好严格相反：先自己(派生类本身)，再客人(对象成员)，后祖先(基类)。

【例 5.7】 析构函数的调用顺序的示例。

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

class Person
{
public:
    Person()
    {  cout <<"the constructor of class Person!\n";  }
    ~Person()
    {  cout <<"the destructor of class Person!\n"; }
private:
    char *name;
    int age;
    char *address;
};

class Student:public Person
{
public:
    Student()
    {  cout <<"the constructor of class Student!\n";  }
    ~Student()
    {  cout <<"the destructor of class Student!\n";  }
```

```
private:
    char *department;
    int level;
};

class Teacher:public Person
{
public:
    Teacher()
    {  cout<<"the constructor of class Teacher!\n";  }
    ~Teacher()
    {  cout<<"the destructor of class Teacher!\n";  }
private:
    char *major;
    float salary;
};

int main()
{
    Student student1;
    Teacher teacher1;
    cout<<" ------ main function finished ------ "<<endl;

    return 0;
}
```

程序的运行结果如图 5.5 所示。

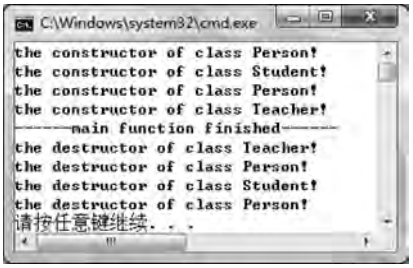

图 5.5 例 5.7 的运行结果

注意：由于析构函数是不带参数的，在派生类中是否要定义析构函数与它所属的基类无关，基类的析构函数不会因为派生类没有析构函数而得不到执行，它们是各自独立的。

5.1.5 派生类对基类成员的继承

下面主要介绍对于某些特殊的情况，如何来调整派生类的访问权限。

1. 如何访问基类的私有成员

派生类无权访问基类的私有成员，不管是私有派生还是公有派生，派生类要想使用基类的私有成员，只能通过调用基类的成员函数的方法实现，也就是使用基类所提供的接口。这种方式对于要频繁访问基类私有成员的派生类来说，使用起来不方便，每次访问都需要进行

函数调用。可以采用以下两种方式访问基类的私有成员。

(1) 在类定义体中增加保护段。

为了便于派生类的访问,可以将基类私有成员中需提供给派生类访问的部分定义为保护段成员。保护段成员可以被它的派生类访问,但是对于外界是隐藏起来的。这样,既方便了派生类的访问,又禁止外界对私有成员的访问。

这种方式的缺点是在公有派生的情况下,如果把成员设为保护访问控制,则为外界访问基类的保护段成员提供了机会,而三种派生方式中,经常使用的是公有派生。

(2) 将派生类声明为基类的友元类。

这样派生类中的所有成员函数均成为基类的友元函数,可以访问基类的私有成员。此外也可直接将需访问基类私有成员的派生类的部分成员函数声明为基类的友元。

2. 通过访问声明调整访问域

在定义私有派生类时,基类中的公有成员在派生类中变为私有成员,必要时可通过访问声明来改变这种情况,调整其访问域(即调整基类中的公有成员在派生类中的访问控制权限),但需遵守以下的规则。

(1) 访问声明仅仅调整名字的访问,不可为它说明任何类型;成员函数在访问声明时,也不准说明任何参数,例如:

```
class Base                      //定义基类
{
public:
    int b;
    int f(int i,int j);
private:
    int a;
};

class Derive:base               //定义私有派生类
{
public:
    int base::b;                //错误,说明了数据类型,应改为 base::b;
    base::f(int i,int j);       //错误,访问声明不应说明函数参数,应改为 base::f;
private:
    int c;
};
```

通过调整访问域,基类中的公有成员在私有派生类中变为公有成员。

(2) 访问声明只能调整基类的保护段和公有段成员在派生类中的访问域,不能改变基类的私有段成员在派生类中的访问域,这样可以保持封装性。

(3) 在 Visual C++ 6.0 中,可以在派生类中降低基类成员的可访问性,也可以把保护段成员提升为 public 成员。

(4) 对重载函数的访问声明将调整基类中具有该名的所有函数的访问域。若基类中的这些重载函数处在不同的访问域,那么,在派生类中就不能调整其访问域,例如:

```
class Base
```

```
{
public:
    x();
    x(int a);
    x(char * p);
};

class Derive:Base
{
public:
    Base::x;                    //基类中的所有名为 x 的重载函数在派生类中将变为公有的
};
```

(5) 若派生类中具有与基类同名的函数,则基类中的此函数不能在派生类中进行访问声明,因为此时基类的同名函数在派生类的作用域中不可见,例如:

```
class base
{
public:
    f();
    f(int a);
    f(char * p);
};

class derive:base
{
public:
    void f(int s);
    base::f;
/* 注意: 此种情况编译时不会出错,但这样的代码段是不健壮的. 若定义 derive 类的对象 d,语句
d.f("abcd");在编译时会出错. d 只能访问 derive 类中的 f 成员,而不能访问 base 类中的 f 成员,即
使写成 d.base::f("abcd");也是错误的 */
};
```

5.2 多重继承

5.2.1 多重继承的概念和定义

在派生类的声明中,基类可以有一个,也可以有多个。如果只有一个基类,则这种继承方式称为单一继承;如果基类有多个,则这种继承方式称为多重继承,这时的派生类同时得到了多个已有类的特征。在多重继承中,各个基类名之间用逗号隔开。多重继承定义的语法如下:

```
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
{
    //定义派生类自己的成员;
};
```

从这个一般形式可以看出,每个基类有一个继承方式来限制其中成员在派生类中的访问权限,如果省略,则默认为 private 继承。其规则和单一继承情况是一样的,多重继承可以看作是单一继承的扩展,单一继承可以看作是多重继承的一个最简单的特例。

在派生过程中,派生出来的新类也同样可以作为基类再继续派生新的类。此外,一个基类可以同时派生出多个派生类。也就是说,一个类从父类继承来的特征也可以被其他新的类所继承,一个父类的特征,可以同时被多个子类继承。这样就形成了一个相互关联的类的家族,称为类族。在类族中,直接参与派生出某类的基类称为直接基类;基类的基类甚至更高层的基类称为间接基类。

5.2.2 二义性和支配规则

1. 二义性的两种情况

(1) 当一个派生类是多重派生也就是由多个基类派生而来时,假如这些基类中的成员有成员名相同的情况,如果使用一个表达式引用了这些同名的成员,就无法确定是引用了哪个基类的成员,这种对基类成员的访问就是二义性的。

要避免此种情况,可以使用成员名限定来消除二义性,也就是在成员名前用对象名及基类名来限定。

【例 5.8】 多重继承中的二义性问题示例。

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

class Bed
{
public:
    Bed(){}
    void sleep()
    {  cout <<"sleeping...\n";  }
    void setWeight(int i)
    {  weight = i;  }
protected:
    int weight;
};

class Sofa
{
public:
    Sofa(){  }
    void watchTV()
    { cout <<"Watching TV.\n"; }
    void setWeight(int i)
    { weight = i; }
protected:
    int weight;
};
```

```
class SleeperSofa :public Bed, public Sofa                    //多重继承
{
public:
    SleeperSofa(){}
    void foldOut(){ cout <<"Fold out the sofa.\n"; }
};

int main()
{
    SleeperSofa ss;
    ss.watchTV();
    ss.foldOut();
    ss.sleep();
    ss.setWeight(20);                                         //出现二义性

    return 0;
}
```

【程序解析】

① 此例中 SleeperSofa 类是 Bed 类和 Sofa 类的公有派生类，而 Bed 类和 Sofa 类中均有成员函数 setWeight()，对象 ss 调用 setWeight()函数时存在二义性，所以当程序编译时将出现如图 5.6 所示的错误提示信息。

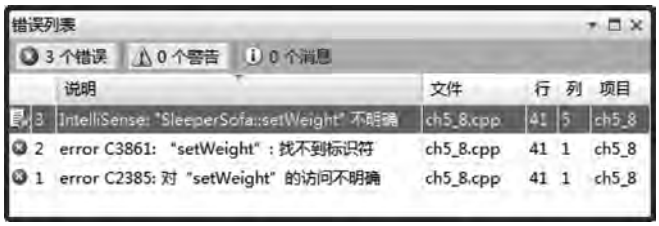

图 5.6 编译时的错误提示信息

② 使用作用域运算符限定成员名可以消除上述二义性。例如：

```
ss.Bed::setWeight(10);
ss.Sofa::setWeight(10);
```

(2) 如果一个派生类从多个基类中派生，而这些基类又有一个共同的基类，则在这个派生类中访问这个共同基类中的成员时会产生二义性。要避免此种情况，可以利用 5.3 节讲到的虚基类。

2. 作用域规则

当基类中的成员名字在派生类中再次声明时，派生类中的名字就屏蔽掉基类中相同的名字(也就是派生类的自定义成员与基类成员同名时，派生类的成员优先)。如果要使用被屏蔽的成员，可由作用域操作符实现。它的形式是：类名::类成员标识符。作用域操作符不仅可以用在类中，也可以用在函数调用时。

3. 支配规则

一个派生类中的名字将优先于它基类中相同的名字，这时二者之间不存在二义性，当选择该名字时，使用支配者(派生类中)的名字，称为支配规则。

5.2.3 赋值兼容规则

所谓赋值兼容规则就是在公有派生的情况下,一个派生类的对象可以作为基类的对象来使用的地方(在公有派生的情况下,每一个派生类的对象都是基类的一个对象,它继承了基类的所有成员并没有改变其访问权限)。

具体地说,有三种情况可以把一个公有派生类的对象作为基类对象来使用。

(1) 派生类对象可以赋予基类的对象,例如(约定类 derived 是从类 base 公有派生而来的):

```
Derived d;
Base b;
b = d;
```

(2) 派生类对象可以初始化基类的引用,例如:

```
Derived d;
Base &br = d;
```

(3) 派生类对象的地址可以赋予指向基类的指针,例如:

```
Derived d;
Base * pb = &d;
```

5.3 虚 基 类

5.3.1 虚基类的概念

当在多条继承路径上有一个公共的基类时,在这些路径中的某几条路径汇合处,这个公共的基类就会产生多个实例(或多个副本),若想只保存这个基类的一个实例,可以将这个公共基类说明为虚基类。从基类派生新类时,使用关键字 virtual 可以将基类说明成虚基类。一个基类,在定义它的派生类时,在作为某些派生类的虚基类的同时,又可以作为另一些派生类的非虚基类。

为了初始化基类的对象,派生类的构造函数要调用基类的构造函数。对于虚基类来讲,由于派生类的对象中只有一个虚基类对象。为保证虚基类对象只被初始化一次,这个虚基类构造函数必须只被调用一次。由于继承结构的层次可能很深,规定将在建立对象时所指定的类称为最直接派生类。虚基类对象是由最直接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。如果一个派生类有一个直接或间接的虚基类,那么派生类的构造函数的成员初始列表中必须列出对虚基类构造函数的调用,如果未被列出,则表示使用该虚基类的默认构造函数来初始化派生类对象中的虚基类对象。

C++语言规定,在一个成员初始化列表中出现对虚基类和非虚基类构造函数的调用,则虚基类的构造函数先于非虚基类的构造函数执行。

从虚基类直接或间接继承的派生类中的构造函数的成员初始化列表中都要列出这个虚基类构造函数的调用。但是,只有用于建立对象的那个派生类的构造函数调用虚基类的构

造函数，而该派生类的基类中所列出的对这个虚基类的构造函数的调用在执行中被忽略，这样便保证了对虚基类对象只初始化一次。

【例 5.9】 利用虚基类避免产生二义性示例。

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

class Furniture                                    //定义家具类
{
public:
    Furniture(){}
    void setWeight(int i){ weight = i; }
    int getWeight(){ return weight; }
protected:
    int weight;
};

class Bed:virtual public Furniture                 //Furniture 类作为 Bed 类的虚基类
{
public:
    Bed(){}
    void sleep(){ cout <<"sleeping...\n"; }
};

class Sofa :virtual public Furniture               //Furniture 类作为 Sofa 类的虚基类
{
public:
    Sofa(){}
    void watchTV(){ cout <<"Watching TV.\n"; }
};

class SleeperSofa :public Bed, public Sofa
{
public:
    SleeperSofa():Sofa(),Bed(){}
    void foldOut(){ cout <<"Fold out the sofa.\n"; }
};

int main()
{
    SleeperSofa ss;
    ss.watchTV();
    ss.foldOut();
    ss.sleep();
    ss.setWeight(20);
    cout <<"weight:"<< ss.getWeight()<< endl;

    return 0;
}
```

程序的运行结果如图 5.7 所示。

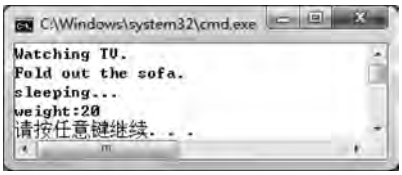

图 5.7　例 5.9 的运行结果

【程序解析】

此例中,Furniture 类作为 Sofa 类和 Bed 类的虚基类,如果不加关键字 virtual,在编译时会出现如图 5.8 所示的错误提示信息。

图 5.8　编译时的错误提示信息

5.3.2　多重继承的构造函数和析构函数

多重继承情况下,严格按照派生类定义时从左到右的顺序来调用构造函数,而析构函数的调用顺序刚好与构造函数的相反。如果基类中有虚基类,则构造函数的调用顺序采用下列规则。

(1) 虚基类的构造函数在非虚基类构造函数之前调用。

(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照它们说明的次序调用。

(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。

需要特别注意,当一个派生类同时有多个基类时,所有需要给予参数进行初始化的基类,都要显式给出基类名和参数表。对于使用默认构造函数的基类,可以不给出类名。同样,对于对象成员,如果是使用默认构造函数,也不需要写出对象名和参数表,而对于单一继承,只需要写一个基类名就可以了。

【例 5.10】　虚基类使用示例。

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

class Base
{
```

```
public:
    Base()
    {  cout <<"This is Base class!\n";  }
};

class Base2
{
public:
    Base2()
    {  cout <<"This is Base2 class!\n";  }
};

class Level1:public Base2,virtual public Base
{
public:
    Level1()
    {  cout <<"This is Level1 class!\n";  }
};

class Level2:public Base2,virtual public Base
{
public:
    Level2()
    {  cout <<"This is Level2 class!\n";  }
};

class TopLevel:public Level1,virtual public Level2
{
public:
    TopLevel()
    {  cout <<"This is TopLevel class!\n";  }
};

int main()
{
    TopLevel topObject;

    return 0;
}
```

程序的运行结果如图 5.9 所示。

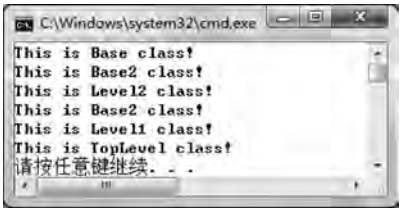

图 5.9　例 5.10 的运行结果

【例 5.11】 多重继承中构造函数和析构函数的调用顺序示例。

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

class OBJ1
{
public:
    OBJ1()
    {  cout <<"Constructing OBJ1"<< endl;  }
    ~OBJ1()
    {  cout <<"Destructing OBJ1"<< endl;  }
};

class OBJ2
{
public:
    OBJ2()
    {  cout <<"Constructing OBJ2"<< endl;  }
    ~OBJ2()
    {  cout <<"Destructing OBJ2"<< endl;  }
};

class Base1
{
public:
    Base1()
    {  cout <<"Constructing Base1"<< endl;  }
    ~Base1()
    {  cout <<"destructing Base1"<< endl;  }
};

class Base2
{
public:
    Base2()
    {  cout <<"Constructing Base2"<< endl;  }
    ~Base2()
    {  cout <<"Destructing Base2"<< endl;  }
};

class Base3
{
public:
    Base3()
    {  cout <<"Constructing Base3"<< endl;  }
    ~Base3()
    {  cout <<"Destructing Base3"<< endl;  }
};
```

```
class Base4{
public:
    Base4()
    {   cout <<"Constructing Base4"<< endl;   }
    ~Base4()
    {   cout <<"Destructing Base4"<< endl;   }
};

class Derive:public Base1,virtual public Base2,public Base3,virtual public Base4
{
public:
    Derive():Base4(),Base3(),Base2(),Base1(),obj2(),obj1()
    {
        cout <<"Constructing Derive"<< endl ;
    }
    ~Derive()
    {   cout <<"Destructing Derive"<< endl;   }
protected:
    OBJ1 obj1;
    OBJ2 obj2;
};

int main()
{
    Derive deriveObject;
    cout <<" ----------------------- "<< endl;

    return 0;
}
```

程序的运行结果如图 5.10 所示。

图 5.10 例 5.11 的运行结果

5.4 类 模 板

在第 2 章讲到模板的作用和函数模板，本节讲解类模板。

类模板为类定义一种模式，使得类中的某些数据成员、某些成员函数的参数、某些成员

函数的返回值,能取任意类型(包括系统预定义的和用户自定义的数据类型)。

如果一个类中数据成员的类型不能确定,或者是某个成员函数的参数或返回值的类型不能确定,就必须将此类声明为模板,它的存在不是代表一个具体的实际的类,而是代表着一类类。

C++语言编译系统根据类模板和特定的数据类型来产生一个类,即模板类(这是一个类)。类模板是一个抽象的类,而模板类是实实在在的类,是由类模板和实际类型结合后由编译系统产生的一个实实在在的类。这个类名就是抽象类名和实际数据类型的结合,如:TclassName < int > 整体是一个类名,包括尖括号和 int。通过这个类才可以产生对象(类的实例)。

对象是类的实例,而模板类是类模板的实例。

类模板由 C++语言的关键字 template 引入,定义的语法形式如下:

```
template < class 类属参数 1,class 类属参数 2,…>
class name
{
    //类定义体
}

template < class 类属参数 1,class 类属参数 2,…>
<返回类型><类名><类型名表>::<成员函数 1>(形参表)
{
    //成员函数定义体
}
```

其中,用尖括号括起来的是形式类属参数表,它列出类属类的每个形式类属参数,多个类属参数之间用逗号隔开,每个类属参数由关键字 class 或 typename 引入。

类模板必须用类型参数将其实例化为模板类后,才能用来生成对象。其表示形式一般为:

```
类模板名 <类型实参表> 对象名(值实参表)
```

其中,类型实参表表示将类模板实例化为模板类时所用到的类型(包括系统固有的类型和用户已定义类型),值实参表表示将该模板类实例化为对象时模板类构造函数所用到的变量。一个类模板可以用来实例化多个模板类。

下面通过一个简单的例子来介绍如何定义和使用类模板及模板类。

【例 5.12】 类模板和模板类的使用示例。

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

template < class T >
class ClassTemplateTest
{
```

```
public:
    ClassTemplateTest(T);
    ~ClassTemplateTest()
    {
        cout <<"call deconstructor function"<< endl;
    }
    T getSize()
    {
        return size;
    }
private:
    T size;
};

template < class T >
ClassTemplateTest < T >::ClassTemplateTest(T n)
{
    size = n;
    cout <<"call constructor function"<< endl;
}

int main()
{
    ClassTemplateTest < int > object1(5);
    ClassTemplateTest < double > object2(6.66);
    cout <<" ---------------------------------- "<< endl;
    cout <<"the class name of object1 is: "<< typeid(object1).name()<< endl;
    cout <<"object1.getSize() is "<< object1.getSize()<< endl;
    cout <<"the class name of object2 is: "<< typeid(object2).name()<< endl;
    cout <<"object2.getSize() is "<< object2.getSize()<< endl;
    cout <<" ---------------------------------- "<< endl;

    return 0;
}
```

程序的运行结果如图 5.11 所示。

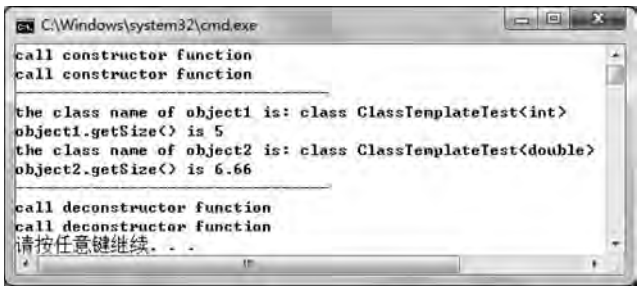

图 5.11 例 5.12 的运行结果

这个例子中定义了一个类模板 ClassTemplateTest，它有一个类型参数 T，类模板的定义由 template < class T >开始，下面的类模板定义体部分与普通类定义的方法相同，只是在

部分地方用参数 T 表示数据类型。

类模板的每一个非内联函数的定义是一个独立的模板,同样以 template < class T >开始,函数头中的类名由类模板名加模板参数构成,如例子中的 ClassTemplateTest < T >。

类模板在使用时必须指明参数,形式为:类模板名<模板参数表>。主函数中的 ClassTemplateTest < int >即是如此,编译器根据参数 int 生成一个实实在在的类,即模板类,理解程序时可以将 ClassTemplateTest < int >看成是一个完整的类名。在使用时 ClassTemplateTest 不能单独出现,它总是和尖括号中的参数表一起出现。

上面的例子中模板只有一个表示数据类型的参数,多个参数以及其他类型的参数也是允许的。

【例 5.13】 定义一个单向链表的模板类,分别实现增加、删除、查找和打印操作。

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

template<class T>                          //定义类模板
class List
{
public:
    List();
    void addNode(T&);                      //增加结点
    void removeNode(T&);                   //删除结点
    T* findNode(T&);                       //查找结点
    void printList();                      //打印输出链表各结点值
    ~List();
protected:
    struct  Node
    {
        Node* pNext;
        T* Pt;
    };
    Node*   pFirst;
};

template<class T>
List<T>::List()
{
    pFirst = 0;
}

template<class T>
void List<T>::addNode(T& t)
{
    Node* temp = new Node;
    temp->Pt = &t;
    temp->pNext = pFirst;
    pFirst = temp;
}
```

```
template < class T >
void List < T >::removeNode(T&t)
{
    Node  * q = 0;
    if( * pFirst - > Pt == t)
    {
        q = pFirst;
        pFirst = pFirst - > pNext;
    }
    else
    {
        for(Node * p = pFirst;p - > pNext;p = p - > pNext)
            if( * (p - > pNext - > Pt) == t)
            {
                q = p - > pNext;
                p - > pNext = q - > pNext;
                break;
            }
    }
    if(q)
    {
        cout <<"delete node of value "<< t << endl;
        delete q - > Pt;
        delete q;
    }
    else
        cout <<"In removeNode function: No node of value "<< t << endl;
}

template < class T >
T *   List < T >::findNode(T& t)
{
    for(Node *  p = pFirst;p;p = p - > pNext)
    {
        if( * (p - > Pt) == t)
            return p - > Pt;
    }
    return  0;
}

template < class T >
void List < T >::printList()
{
    cout <<"链表中各结点值为: ";
    for(Node *  p = pFirst;p;p = p - > pNext)
    {
        if(p!= pFirst)
            cout <<" - >";
```

```
        cout << * (p -> Pt);
    }
    cout << endl;
}

template < class T >
List < T >::~List()
{
    Node * p = pFirst;
    while (p)
    {
        pFirst = pFirst -> pNext;
        delete p -> Pt;
        delete p;
        p = pFirst;
    }
}

int main()
{
    List < int > intList;                    //intList 是模板类 List < int >的对象
    for(int i = 1;i < 7;i++)
    {
        intList.addNode( * new int(i));      //向链表中插入结点
    }
    intList.printList();
    int  b = 9;
    int * pa = intList.findNode(b);
    if(pa)
        intList.removeNode( * pa);
    else
        cout <<"In main function: No code of value "<< b << endl;
    intList.removeNode( * new int(5));
    intList.removeNode( * new int(1));
    intList.removeNode( * new int(10));
    intList.printList();

    return 0;
}
```

程序的运行结果如图 5.12 所示。

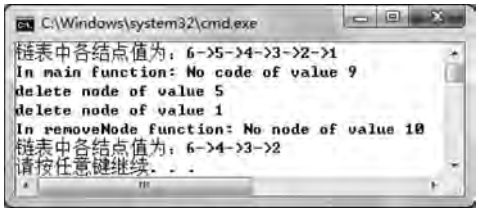

图 5.12　例 5.13 的运行结果

5.5 应用示例

【例 5.14】 本例定义了一个基类 Person 类及其两个派生类(Teacher 类和 Student 类),显示不同对象的相关信息。

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

class Person
{
public:
    Person(char *name,int age,char sex)
    {
        strcpy_s(m_strName, name);
        m_nSex = (sex == 'm'?0:1 );
        m_nAge = age;
    }
    void ShowMe()
    {
        cout <<"  姓    名: "<< m_strName << endl;
        cout <<"  性    别: "<<(m_nSex == 0?"男":"女")<< endl;
        cout <<"  年    龄: "<< m_nAge << endl;
    }
protected:
    char m_strName[20];
    int m_nSex;
    int m_nAge;
};

class Teacher: public Person
{
public:
    Teacher(char *name,int age,char sex,char *dept,int salary)
        :Person(name,age,sex)
    {
        strcpy_s(m_strDept, dept);
        m_fSalary = salary;
    }
    void ShowMe()
    {
        Person::ShowMe();
        cout <<"  工作单位: "<< m_strDept << endl;
        cout <<"  月    薪: "<< m_fSalary << endl;
    }
private:
    char m_strDept[30];
    int  m_fSalary;
};
```

```
class Student: public Person
{
public:
    Student(char * name,int age,char sex,char * ID,char * Class)
        :Person(name,age,sex)
    {
        strcpy_s(m_strID, ID);
        strcpy_s(m_strClass, Class);
    }
    void ShowMe()
    {
        cout <<"  学     号: "<< m_strID << endl;
        Person::ShowMe();
        cout <<"  班     级: "<< m_strClass <<"\n";
    }
private:
    char m_strID[12];
    char m_strClass[50];
};

int main()
{
    Teacher teacher1("李强",38,'m',"信息工程学院",3800);
    Student student1("马丽",20,'f',"03016003","计科 173");
    cout <<"------ 教师信息如下: ------ "<< endl;
    teacher1.ShowMe();
    cout <<"------ 学生信息如下: ------ "<< endl;
    student1.ShowMe();

    return 0;
}
```

程序的运行结果如图 5.13 所示。

图 5.13　例 5.14 的运行结果

【例 5.15】 某单位所有员工根据领取薪金的方式分为几类：时薪工(hourlyworker)、计件工(pieceworker)、经理(manager)、佣金工(commissionworker)。时薪工按工作的小时支付工资，对于每周超过 40 小时的加班时间，按照附加 50%薪水支付工资。按生产的每件

产品给计件工支付固定工资，假定该工人仅制造一种产品。经理每周得到固定的工资。佣金工每周得到少许的固定保底工资，加上该工人在一周内总销售的固定百分比。试编制一个程序来实现该单位的所有员工类，并加以测试。

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

class Employee                                  //雇员类
{
public:
    //设置雇员的基本信息
    void setInfo(char * empname,int empsex,char * empid)
    {
        strcpy_s(name,empname);
        strcpy_s(emp_id,empid);
    }
    void getInfo(char * empname,char * empid) //取得雇员的基本信息
    {
        strcpy_s(empname,strlen(name) + 1,name);
        strcpy_s(empid,strlen(emp_id) + 1,emp_id);
    }
    double getSalary()                          //取得所应得的总薪金数
    {
        return salary;
    }
protected:
    char name[10];                              //姓名
    char emp_id[8];                             //职工号
    double salary;                              //薪金数
};

class HourlyWorker:public Employee              //时薪工类
{
public:
    HourlyWorker()
    {
        hours = 0;
        perHourPay = 15.6;
    }
    int getHours()                              //取得某人工作的小时数
    {
        return hours;
    }
    void setHours(int h)                        //设置某人工作的小时数
    {
        hours = h;
    }
    double getperHourPay()                      //取得每小时应得的报酬
    {
```

```
        return perHourPay;
    }
    void setperHourPay(double pay)          //设置每小时应得的报酬
    {
        perHourPay = pay;
    }
    void computePay()                       //计算工资
    {
        if(hours < = 40)
            salary = perHourPay * hours;
        else
            salary = perHourPay * 40 + (hours - 40) * 1.5 * perHourPay;
    }
protected:
    int hours;                              //工作的小时数
    double perHourPay;                      //每小时应得的报酬
};

class PieceWorker:public Employee           //计件工
{
public:
    PieceWorker()
    {
        pieces = 0; perPiecePay = 26.8;
    }
    int getPieces()
    {return pieces;}
    void setPieces(int p)                   //设置生产的工件总数
    {pieces = p;}
    double getperPiecePay()
    {return perPiecePay;}
    void setperPiecePay(double ppp)
    { perPiecePay = ppp;}
    void computePay()
    { salary = pieces * perPiecePay;}
protected:
    int pieces;                             //每周所生产的工件数
    double perPiecePay;                     //每个工件所应得的工资数
};

class Manager:public Employee               //经理类
{
public:
    void setSalary(double s)                //设置经理的工资数
    {   salary = s;   }
};

class CommissionWorker:public Employee      //佣金工类
{
public:
    CommissionWorker()
```

```
    {
        baseSalary = 500;
        total = 0;
        percent = 0.01;
    }
    double getBase()
    { return baseSalary;}
    void setbase(double base)
    { baseSalary = base;}
    double getTotal()
    { return total;}
    void setTotal(double t)
    { total  = t;}
    double getPercent()
    { return percent;}
    double setPercent(double p)
    { percent = p;}
    void computePay()
    { salary = baseSalary + total * percent;}
protected:
    double baseSalary;                          //保底工资
    double total;                               //一周内的总销售额
    double percent;                             //提成的额度
};

int main()
{
    char name[10],emp_id[9];

    HourlyWorker hworker;                       //时薪工
    hworker.setInfo("John",0,"001");
    hworker.setHours(65);
    hworker.getInfo(name,emp_id);
    hworker.computePay();
    cout <<" ------ HourlyWorker ------ "<< endl
        <<"    "<< name <<"'s id is "<< emp_id << endl
        <<"    "<< name <<"'s salary is "<< hworker.getSalary()<< endl;

    PieceWorker pworker;                        //计件工
    pworker.setInfo("Mark",0,"002");
    pworker.setPieces(100);
    pworker.computePay();
    pworker.getInfo(name,emp_id);
    cout <<" ------ PieceWorker ------ "<< endl
        <<"    "<< name <<"'s id is "<< emp_id << endl
        <<"    "<< name <<"'s salary is:"<< pworker.getSalary()<< endl;

    CommissionWorker cworker;                   //佣金工
    cworker.setTotal(234.6);
    cworker.setInfo("Jane",0,"003");
    cworker.computePay();
```

```
    cworker.getInfo(name,emp_id);
    cout <<" ------ CommissionWorker ------ "<< endl
        <<"    "<< name <<"'s id is "<< emp_id << endl
        <<"    "<< name <<"'s salary is:"<< cworker.getSalary()<< endl;

    Manager manager;                          //经理
    manager.setInfo("Mike",1,"004");
    manager.setSalary(3500);
    manager.getInfo(name,emp_id);
    cout <<" ------ Manager ------ "<< endl
        <<"    "<< name <<"'s id is "<< emp_id << endl
        <<"    "<< name <<"'s salary is:"<< manager.getSalary()<< endl;

    return 0;
}
```

程序的运行结果如图 5.14 所示。

【例 5.16】 从二叉排序树中删除一个结点。

图 5.14 例 5.15 的运行结果

分析：

(1) 被删除结点为叶结点，则只需修改其双亲结点对应指针为 NULL，并释放该结点。

(2) 被删结点只有左子树或右子树，此时只要将其左子树或右子树直接变成其双亲结点的左子树或右子树。

(3) 若被删结点 K 左右子树均不空，需循该结点的右子树根结点 M 的左子树，一直向左子树找，直到某结点 x 的左子树为空，则把 x 的右子树改为其父结点的左子树，而用 x 结点取代 K 结点。递归的说法：用结点 x 取代 K，再从右子树中删去结点 x。实际就是用 K 结点的中序下一结点 x 取代 K 结点，也可以用 K 结点中序前一结点 y 来取代 K 结点，一句话，用最接近被删结点值的结点来代替它。

源程序如下：

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

template < typename T >
class BinaryTree;

template < typename T >
class Node
{
public:
    Node()
    {
        lChild = NULL; rChild = NULL;
```

```
    }
    Node(T data,Node<T> * left = NULL,Node<T> * right = NULL)
    {
        info = data;
        lChild = left;
        rChild = right;
    }
    friend class BinaryTree<T>;
private:
    Node<T> * lChild, * rChild;
    T info;
};

template<typename T>
class BinaryTree
{
public:
    BinaryTree()                                          //构造函数
    {  root = NULL;  }
    ~BinaryTree()                                         //析构函数
    {  destroyTree(root);  }
    void creatTree(T* data,int n);                        //建立(排序)二叉树
    void inOrder()
    {  inOrder(root);  }                                  //中序遍历
    void removeNode(const T &data)                        //删除结点
    {  removeNode(data,root,root);  }
private:
    Node<T> * root;                                       //二叉树的根指针
    void inOrder(Node<T> * Current);                      //中序遍历
    void insertNode(const T &data,Node<T> * &b);          //插入结点,第二个参数为引用
    void removeNode(const T &data,Node<T> * &a,Node<T> * &b);     //删除结点
    void destroyTree(Node<T> * Current);                  //删除树
};

template<typename T>
void BinaryTree<T>::destroyTree(Node<T> * Current)
{
    if(Current!= NULL)
    {
        destroyTree(Current->lChild);
        destroyTree(Current->rChild);
        delete Current;                                   //后序释放根结点
    }
}

template<typename T>
void BinaryTree<T>::insertNode(const T &data,Node<T> * &b)
{
    if(b == NULL)                                         //空树,插入
    {
        b = new Node<T>(data);
```

```
        if(b == NULL)
        {
            cout <<"空间不足"<< endl;
            exit(1);
        }
    }
    else if(data < b -> info)                    //小于,向左子树去插入
        insertNode(data,b -> lChild);
    else                                         //大于或等于,向右子树去插入
        insertNode(data,b -> rChild);
}

template < typename T >
void BinaryTree < T >::creatTree(T * data,int n)   //建立一棵二叉排序树
{
    for(int i = 0;i < n;i++)
        insertNode(data[i],root);
}

template < typename T >
void BinaryTree < T >::inOrder(Node < T > * Current)
{
    if(Current!= NULL)                           //递归终止条件
    {
        inOrder(Current -> lChild);              //中序遍历左子树
        cout << Current -> info <<"  ";          //访问根结点,注意所放位置
        inOrder(Current -> rChild);              //中序遍历右子树
    }
}

template < typename T >
void BinaryTree < T >::removeNode(const T &data,Node < T > * &a,Node < T > * &b)
{
    Node < T > * temp1, * temp2;
    if(b == NULL)
        return;
    if(data < b -> info)
        removeNode(data,b,b -> lChild);          //所查数小,去左子树
    else if(data > b -> info)
        removeNode(data,b,b -> rChild);          //所查数大,去右子树
    else if (b -> lChild!= NULL&&b -> rChild!= NULL)
    {
        //查到值为 data 的结点,它有两个子树
        temp2 = b;
        temp1 = b -> rChild;                     //向右一步
        if(temp1 -> lChild!= NULL)
        {
            while(temp1 -> lChild!= NULL)
            {
                //向左到极左的结点,将要用来取代被删除结点
                temp2 = temp1;
```

```
                temp1 = temp1 -> lChild;
            }
            temp2 -> lChild = temp1 -> rChild;
            //把选中结点的右子树或 NULL,接到该结点的父结点的左子树上
        }
        else
            temp2 -> rChild = temp1 -> rChild;    //向右一步后无左子树
        b -> info = temp1 -> info;
        delete temp1;
    }
    else
    {   //只有一个子树或是叶结点
        temp1 = b;
        if(b -> rChild!= NULL)                    //只有右子树
        {
            temp1 = b -> rChild;
             b -> info = temp1 -> info;
             b -> rChild = temp1 -> rChild;
             b -> lChild = temp1 -> lChild;
        }
        else if(b -> lChild!= NULL)               //只有左子树
        {
            temp1 = b -> lChild;
            b -> info = temp1 -> info;
            b -> rChild = temp1 -> rChild;
            b -> lChild = temp1 -> lChild;
        }
        else if(b == root) root = NULL;           //叶结点,仅有根结点
        else if(a -> rChild == temp1) a -> rChild = NULL;   //被删除结点在父结点右边
        else
            a -> lChild = NULL;                   //被删除结点在父结点左边
        delete temp1;
    }
}

int main()
{
    const int n = 15;
    int i,a[n] = {10,5,15,8,3,18,13,12,14,16,20,1,4,6,9};
    BinaryTree < int > btree;
    btree.creatTree(a,n);
    cout <<"输入数据: "<< endl;
    for(i = 0;i < n;i++)
    cout << a[i]<<"  ";
    cout << endl <<"中序: "<< endl;
    btree.inOrder();                              //中序遍历,升序输出
    btree.removeNode(a[13]);                      //删除叶结点
    cout << endl <<"中序: "<< endl;
    btree.inOrder();                              //中序遍历,升序输出
    btree.removeNode(a[3]);                       //删除子树结点
    cout << endl <<"中序: "<< endl;
```

```
    btree.inOrder();
    btree.removeNode(a[9]);                    //删除叶结点
    cout << endl <<"中序: "<< endl;
    btree.inOrder();
    btree.removeNode(a[2]);                    //被删除结点的右子树根结点无左子树
    cout << endl <<"中序: "<< endl;
    btree.inOrder();
    btree.removeNode(a[0]);                    //删除根结点
    cout << endl <<"中序: "<< endl;
    btree.inOrder();
    int a1[1] = {10};                          //仅有根结点
    BinaryTree < int > btree1;
    btree1.creatTree(a1,1);
    cout <<"\n 输入数据: "<<'\t'<< a1[0]<<'\t';
    cout << endl <<"中序: "<<'\t';
    btree1.inOrder();
    btree1.removeNode(a[0]);                   //删除叶结点
    cout << endl <<"中序: "<< endl;
    btree1.inOrder();                          //中序遍历,升序输出,输出为空

    return 0;
}
```

程序的运行结果如图 5.15 所示。

图 5.15　例 5.16 的运行结果

习　　题

1. 什么是类的继承与派生?

2. 类的三种继承方式之间的区别是什么?

3. 派生类能否直接访问基类的私有成员? 若否,应如何实现?

4. 派生类构造函数和析构函数的执行顺序是怎样的? 在多重继承中,派生类构造函数和析构函数的执行顺序又是怎样的?

5. 派生类的构造函数和析构函数的作用是什么？

6. 多重继承一般应用在哪些场合？

7. 在类的派生中为何引入虚基类？在含有虚基类的派生类中，当创建它的对象时，构造函数的执行顺序如何？

8. 设计一个大学的类系统，学校中有学生、教师、职员，他们之间有相同的地方，又有自己的特性。利用继承机制定义这个系统中的各个类及类中必需的操作。

9. 假定车可分为货车和客车，客车又可分为轿车、面包车和公共汽车。请设计相应的类层次结构。

10. 设计一个能细分为矩形、三角形、圆形和椭圆形的图形类。使用继承将这些图形分类，找出能作为基类部分的共同特征（如宽、高、中心点等）和方法（如初始化、求面积等），并看看这些图形能否进一步划分为子类。

11. 考虑大学的学生情况，试利用单一继承来实现学生和毕业生两个类，设计相关的数据成员及函数，编写程序对继承情况进行测试。

提示：作为学生一定有学号、姓名、性别、学校名称及入学时间等基本信息，而毕业生除了这些信息外，还应有毕业时间、所获学位的信息，可根据这些内容设计类的数据成员，也可加入一些其他信息，除了设计对数据进行相应操作的成员函数外，还要考虑到成员类型、继承模式，并在 main()函数中进行相应测试。可设计多种继承模式来测试继承的属性。

12. 定义一个哺乳动物类，再由此派生出人类、狗类和猫类，这些类中均有 speak()函数，观察在调用过程中，到底使用了哪一个类的 speak()函数。

13. 通过多重继承定义研究生类，研究生既有学生的属性，又有教师的属性。

14. 定义类模板 SortedSet，即元素的有序集合，集合元素的类型和集合元素的最大个数可由使用者确定。要求该类模板对外提供以下三种操作。

(1) insertElement()：插入一个新的元素到合适的位置上，并保证集合元素的值不重复。

(2) getAddress()：返回比给定值大的最小元素的地址。若不存在，返回 0。

(3) deleteElement()：删除与给定值相等的那个元素，并保持剩余元素的有序性。

15. 定义一堆栈类模板，使其具有如下操作。

(1) void push(T)；//压栈操作

(2) T pop()；//弹栈操作

(3) bool empty()；//判空操作

(4) T display(int)；//显示指定位置的元素值