第5单元　简单的公司人员体系

继承(泛化)是一种基于已知类(父类)来定义新类(子类)的方法,或者说是一种将特性从一个类传递到另一个类的机制。这一单元介绍从一个已知类创建新类的基本方法,以及特性传递的基本形式。

5.1　公司人员的类层次结构

一个简单的公司人员体系,可能涉及3类对象:人(person)、员工(employee)和管理者(manager)。本题要建立这三者之间的联系。

5.1.1　问题建模

在公司里,普通员工(employee)与管理人员(manager)都是员工(employee),都具有员工特征;而在公司工作的人员与不在公司工作的人员,都是人(person),都具有人的特征。如果声明3个类,则它们之间具有如下关系:Person通常具有姓名(name)、年龄(age)和性别(sex)等属性;Employee,要在Person的基础上增加两个属性:职工号(workerID)和工资(salary);而Manager又要在Employee的基础上再增加一个属性——职位(post)。这种以一个类为基础,通过继承(inheritance),建立一个新的类的过程称为派生(derived)。派生所得到的类称为派生类(derived class)。作为派生其他类的类称为基类(base class)。派生类不仅继承了基类的属性,还继承了基类的行为。此外,在派生类中还可以追加其他的属性和行为。图5.1所示的类图,描述了本题中3个类之间的派生关系。

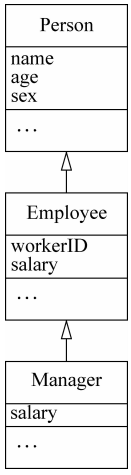


图5.1　公司人员体系模型

5.1.2　类层次结构声明

在C++语言中,派生关系用下面的格式描述。这里,public表示以公有方式派生。

```
class 派生类名: public 基类名
{
    新增成员列表
};
```

【代码5-1】 图5.1中派生关系的C++描述代码。

```
#include<string>
```

```
class Person {
private:
    std::string    name;
    int    age;
    char    sex;

public:
Person (std::string theName,int theAge,char theSex):name (theName),age (theAge),sex
(theSex)
    {}

    ~Person (void) {}

     std::string    getName (void)const                    //const 成员函数
    {return name;}

    int       getAge (void)const                           //const 成员函数
    {return age;}

    char getSex (void)const                                //const 成员函数
    {return sex;}

    void output (void);
};

//声明 Person 类的派生类 Employee
class Employee:public Person {
private:
    unsigned int  workerID;
    double  salary;
public:
    Employee (std::string theName,int theAge,char theSex
    ,unsigned int theWorkerID, double theSalary)
    :Person (theName,theAge,theSex),workerID (theWorkerID), salary (theSalary)
    {}

    ~Employee (void) {}

    unsigned int    getWorkerID (void)const                //const 成员函数
    {return workerID;}

    double    getSalary (void)const                        //const 成员函数
    {return salary;}

    void    output (void);
};

//声明 Employee 类的派生类 Manager
class Manager:public Employee {
```

```
private:
    std::string    post;
public:
    Manager (std::string theName,int theAge,char theSex,
             unsigned int theWorkerID,double theSalary,string thePost)
             :Employee(theName,theAge,theSex,theWorkerID,theSalary),post (thePost)
    {}

    ~Manager (void) {}

    std::string getPost (void)const                              //const 成员函数
     {return post;}

    void    output (void);
};
```

说明：

(1) 在具有泛化关系的类层次结构中，派生类对象包含基类对象，所以在声明派生类构造函数时，需要调用基类的构造函数，即要在初始化列表中列出对于基类构造函数的调用。

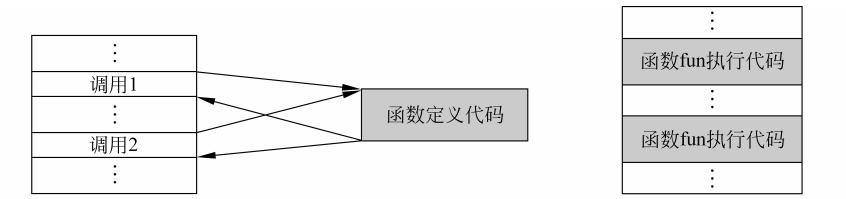
(2) 基类中私密成员的访问。如前所述，一个对象的公开成员，不仅可以被本对象的成员函数访问，也可以被本对象外部的函数访问。但是，一个对象的私密成员则只可以被本对象的成员函数访问，不能被任何外部函数访问，即使是派生类对象中的成员函数也不可。派生类对象的成员函数只能间接地通过基类成员函数访问基类的私密成员。这种间接访问可以通过两种形式进行：

- 派生类对象的新增成员函数调用基类公开函数。
- 由于派生类继承了基类的公开成员函数，因此可以将基类公开成员函数作为派生类对象的分量直接调用。

(3) 关键字 const 放在成员函数的函数头后面，将成员函数声明成为 const 成员函数。这样，就不允许在所定义的成员函数中出现修改数据成员值的语句，也不能调用非 const 成员函数，只能调用 const 成员函数。在本例中，凡是仅返回数据成员值的函数都不允许修改数据成员，所以都定义为 const 成员函数。

(4) 在派生类中，可以重新定义基类的同名成员函数。

(5) 在本例的各个类中，除了 output()外，其他成员函数都是在声明的同时，在类中给出了定义。这样的函数称为内联(inline)函数。内联函数与普通函数运行机理的区别如图 5.2 所示。普通函数的代码具有公共性，每遇一次调用，就把流程转移到这段公共代码一



(a) 普通函数fun的调用情况　　(b) 内联函数fun的代码嵌入情况

图 5.2　编译后的普通函数和内联函数

次；而内联函数经过编译后，会在所有调用该函数的语句处，嵌入该内联函数的代码。普通函数调用时会有一个调用—返回的过程，这要耗费一些系统资源。而内联函数没有这些过程，所以效率比较高。因此，代码比较短且会多次调用的函数，常被写成内联函数。

内联函数的定义有两种形式：隐式形式和显式形式。上述定义在类中的内联函数称为隐式内联函数。如果在函数声明语句中冠以关键字 inline，这种内联函数就是显式的。显式内联函数的定义也可以写在类声明之外，这时函数头部的关键字 inline 是可选的。

【代码 5-2】 显式内联函数的例子。

```
class Circle {
public:
    ⋮
    inline double calcPerimeter (void);
    ⋮
};

inline double Circle :: calcPerimeter (void)  {       //关键字 inline 可选
    return 2 * PI * radius;
}
```

注意：内联函数中不宜有复杂的控制结构，有些编译器不支持内联函数中包含循环或 switch 结构，有的只接受一两个 if 语句。

5.1.3 在派生类中重定义基类成员函数

在类层次中，派生类继承了基类的成员函数（或数据成员）。但是，在派生类中往往有不同于基类中的功能补充，例如在一个人事管理系统中，在每一层都需要有一个显示人员数据的函数，为了便于记忆，可以使用相同的名字。然而，每一层显示的内容不相同，可能在父类只要显示：职工号、姓名、岗位，而在子类下一层还需要增加职位……，为此需要在子类中对这个显示函数重新定义，示例见代码 5-3。

【代码 5-3】 output 函数的重定义与实现。

```
#include<iostream>

class Person {
private:
    ⋮
    void output (void);
};

class Employee:public Person {
    ⋮
    void output (void);
};

class Manager:public Employee {
    ⋮
```

```
    void output (void);
};

void Person::output (void) {
    std::cout<<"姓名:"<<getName()<<std::endl;
    std::cout<<"年龄:"<<getAge()<<std::endl;
    std::cout<<"性别:"<<getSex()<<std::endl;
}

void Employee::output (void) {
    std::cout<<"姓名:"<<getName()<<std::endl;
    std::cout<<"年龄:"<<getAge()<<std::endl;
    std::cout<<"性别:"<<getSex()<<std::endl;
    std::cout<<"工号:"<<getWorkerID()<<std::endl;
    std::cout<<"工资:"<<getSalary()<<std::endl;
}

void Manager::output (void) {
    std::cout<<"姓名:"<<getName()<<std::endl;
    std::cout<<"年龄:"<<getAge()<<std::endl;
    std::cout<<"性别:"<<getSex()<<std::endl;
    std::cout<<"工号:"<<getWorkerID()<<std::endl;
    std::cout<<"工资:"<<getSalary()<<std::endl;
    std::cout<<"职位:"<<getPost()<<std::endl;
}
```

这样,在派生类中重定义了基类中的成员函数后,派生类对象用这个名字调用的是派生类中用这个名字定义的成员函数版本,基类对象用这个名字调用的是基类中用这个名字定义的成员函数版本,实现了一种多态性。例如:

```
Person per;
Employee emp;
Manager mang;
emp.output();                    //调用 Employee 类的 output()
per.output();                    //调用 Person 类的 output()
mang.output();                   //调用 Manager 类的 output()
```

也就是说,一旦在派生类中重定义了基类的一个成员函数,则在派生类中这个基类的成员函数就被屏蔽——不可见。不过,在派生类中如果还想访问基类中的函数版本,也并非不可能,只是需要使用作用域操作符指定其作用域。例如:

```
Manager m1(…);
m1.output();                     //调用 Manager 类的 output()
m1.Employee::output();           //调用 Employee 类的 output()
m1.Person::output();             //调用 Person 类的 output()
```

5.1.4 类层次结构中成员函数执行规则

【代码 5-4】 测试类层次结构中成员函数执行规则的程序。

```
#include<iostream>
#include<string>

class Person {
private:
    std::string    name;
    int        age;
    char    sex;
public:
    Person (std::string theName,int theAge,char theSex)
    :name (theName),age (theAge),sex (theSex) {
        std::cout<<"执行 Person 构造函数。"<<std::endl;        //输出提示
    }

    ~Person (void) {
        std::cout<<"执行 Person 析构函数。"<<std::endl;        //输出提示
    }

    std::string getName (void)const {return name;}
    int getAge (void)const {return age;}
    char getSex (void)const {return sex;}
    void output (void);
};

//声明 Person 类的派生类 Employee
class Employee:public Person {
private:
    unsigned int   workerID;
    double        salary;
public:
    Employee (std::string theName,int theAge,char theSex,unsigned int theWorkerID
    ,double theSalary)
    :Person (theName,theAge,theSex),workerID (theWorkerID),salary (theSalary) {
        std::cout<<"执行 Employee 构造函数。"<<std::endl;      //输出提示
    }

    ~Employee (void) {
        std::cout<<"执行 Employee 析构函数。"<<std::endl;      //输出提示
    }

    unsigned int getWorkerID (void)const {return workerID;}
    double getSalary (void)const {return salary;}
    void output();
};

//声明 Employee 类的派生类 Manager
class Manager:public Employee {
private:
    std::string post;
```

```
public:
    Manager (std::string theName,int theAge,char theSex,unsigned int theWorkerID
        ,double theSalary, std::string thePost)
        :Employee (theName,theAge,theSex,theWorkerID,theSalary),post (thePost) {
        std::cout<<"执行 Manager 构造函数。"<<std::endl;     //输出提示
    }

    ~Manager() {
        std::cout<<"执行 Manager 析构函数。"<<std::endl;     //输出提示
    }

    std::string getPost (void)const {return post;}
    void output();
};

//output()函数的实现
//…(同前,略)

//测试主函数
int main (void) {
    std::cout<<"\n-------初始化 Manager 对象时构造函数调用顺序--------\n";
    Manager m1 ("AAAAA",26,'f',555555,5432.10,"部长");
    std::cout<<"\n-------执行 m1.output()的情形----------------------\n";
    m1.output();
    std::cout<<"\n-------执行 m1.Employee::output()的情形-----------\n";
    m1.Employee::output();
    std::cout<<"\n-------执行 m1.Person::output()的情形-------------\n";
    m1.Person::output();
    std::cout<<"\n-------撤销 Manager 对象时析构函数调用顺序----------\n";
    return 0;
}
```

测试结果：

```
-------初始化Manager对象时构造函数调用顺序--------
执行Person构造函数。
执行Employee构造函数。
执行Manager构造函数。

-------执行m1.output ()的情形----------------------
姓名：AAAAA
年龄：26
性别：f
工号：555555
工资：5432.1
职位：部长

-------执行m1.Employee::output ()的情形-----------
姓名：AAAAA
年龄：26
性别：f
工号：555555
工资：5432.1

-------执行m1.Person::output ()的情形-------------
姓名：AAAAA
年龄：26
性别：f

-------撤销Manager对象时析构函数调用顺序----------
执行Manager析构函数。
执行Employee析构函数。
执行Person析构函数。
```

创建Manager m1对象过程中，构造函数的执行顺序

撤销Manager m1对象过程中，构造函数的执行顺序

说明：

(1) 在多层结构的类层次结构中，派生类的构造函数要调用直接基类的构造函数，不能调用间接基类的构造函数，构造函数的初始化列表中只能列出直接基类构造函数的调用。

(2) 构造函数的执行过程分为两个阶段：第一阶段是调用阶段，即按照从派生类到基类的方向，调用上一层的构造函数，直到最基层的类(没有可调用)为止。第二阶段是从最基类开始逆着调用的方向执行各层的构造函数。在每一层中，如采用初始化列表，则按照从左到右的方向依次执行。也就是说，声明一个派生类对象，意味着要先自动创建一个基类对象，再创建一个派生类对象。如图 5.3 所示，当声明一个 Manager 对象时，首先执行 Person 类构造函数，自动创建一个 Person 类对象；再执行 Employee 类构造函数，自动创建一个 Employee 类对象；最后执行 Manager 类构造函数，创建一个 Manager 类对象。

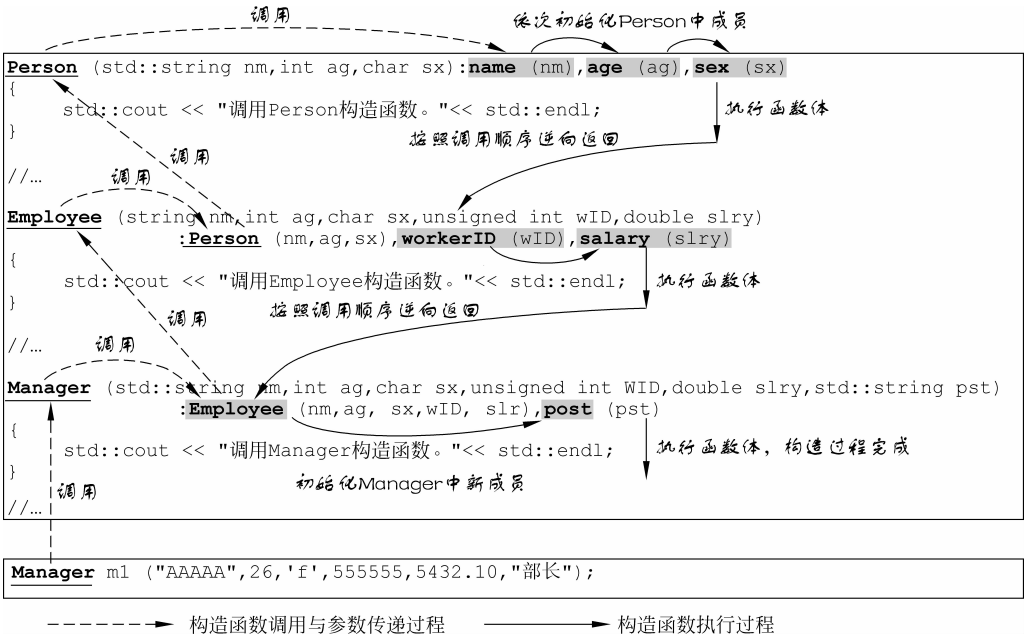


图 5.3　生成 Manager m1 过程中构造函数的调用过程

(3) 析构函数的执行顺序与构造函数的执行顺序相反，即当销毁派生类对象时，析构函数的执行顺序则是从下向上进行，即先执行派生类的析构函数，撤销派生类对象，然后执行基类的析构函数，撤销基类对象。

(4) 在创建派生类对象时，由于首先执行基类构造函数，所以也就首先创建了基类对象，为该对象中的数据成员分配存储空间，只不过这个对象是一个无名对象。接着再为派生类中所增添的数据成员分配存储空间。图 5.4 展示生成 Manager 对象时存储空间的分配过程。

(5) 当派生类与基类中有同名函数时，除非用作用域操作符指定，否则派生类对象调用该同名函数时，将按照派生类优先的原则调用派生类的那个同名函数。

(6) 派生类对象可以调用基类的公开成员，但不可调用基类的私密成员。

图 5.4　Manager m1 对象的内存分配过程

5.1.5　基于血缘关系的访问控制——protected

前面介绍了当一个类的成员采用 private 和 public 访问保护，在 public 派生时，基类的 private 成员在派生类中不可访问。这就带来许多不便。例如在代码 5-4 中，派生类要访问基类的私密成员，必须先调用基类的一个公开成员。那么如何才能做到在一个类层次结构中，使有些成员可以被各个类对象共同访问，而在该类层次结构的外部，不能访问，即做到血缘内外有别呢？这就要用 protected 进行访问控制。这类用 protected 进行访问控制的成员称为保护成员。一个基类的保护成员进行 public 派生后，在派生类中仍然是保护成员。

这样，访问控制就被分为 3 个级别了：

（1）private：访问权限仅限于本类的成员。

（2）protected：访问权限扩大到本血统内部。

（3）public：访问权限扩大到血统外部。

读者可以将代码 5-4 中 3 个类中的 private 改为 protected，观察运行结果。

5.2　指针与引用

5.2.1　指针＝基类型＋地址

1. 变量及其存储空间存储

一个变量（或对象）一经生成，系统就会给其分配一个由其类型所确定的存储空间。于是，变量与这个存储区域之间有了如下联系。

（1）一个名字。

（2）一个该存储区域的首地址。

（3）一个由数据类型决定的存储区域大小和存储方式。

（4）在这个存储区间中存放的数据值。

图 5.5 表明了 3 个变量在内存中的存储情况。先看变量 i 和 sh，它们的区别一是具有不同的存储地址 0x3004 和 0x3006，二是它们所占有的存储空间大小不同，分别是 4B 和 2B——这是因为它们具有不同的类型。

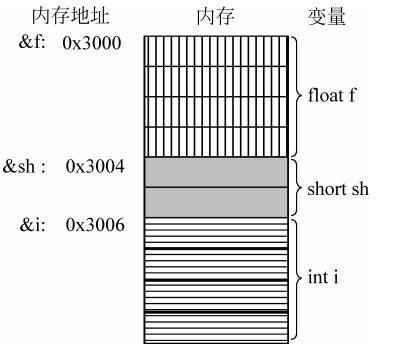


图 5.5　变量及其存储地址

再看变量 i 和 f,它们的区别首先是具有不同的存储地址,此外它们虽然具有相同的存储空间,但存储方式不同——因为它们的类型不同。

2. C++ 程序对于一个存储空间中数据的存取方式

C++ 程序对于内存中一个数据的存取,可以通过两种方式进行:

(1) 类型+地址方式。即知道了一个数据在内存中的存储地址(区间的首地址)、存储空间大小以及存储格式,就可以对数据进行存取。

(2) 变量名方式。即使用变量名进行存取。

实际上,在程序编译时,变量名被编译系统解释为地址+类型方式。因此从方便用户的角度,许多高级语言不向用户开放地址+类型方式。但是,C/C++ 从提供程序效率的角度,向用户开放了地址+类型方式并将之称为指针,即有关系

指针=基类型+内存地址

这个公式说明,类型对于指针是非常重要的。要使用一个指针,首先要求它必须确定是指向什么类型的数据。指针所指向数据的类型,称为指针的基类型。

3. 用变量名计算地址

一个变量一经定义,便有了地址。C++ 允许使用操作符 & 取得名字中所隐含的地址,并在程序中使用这些地址。例如对于图 5.5 中的变量 f,可以用表达式 &f 取得其地址 0x3000。& 称为取地址操作符,是一种单目操作符,具有较高的优先级和从右向左的结合性。

4. 指针变量的定义与初始化

如前所述,类型对于指针是非常重要的。一个指针可以不指向某个具体地址,但必须首先确定它指向什么类型。所以指针可以用下面的格式定义。

```
基类型 *指针变量名;
```

这里,用(*)表明所定义的名字是一个指针变量名。例如:

```
double *pd;
```

定义了一个指向 double 类型的指针。

注意,表示指针的符号(*),可以靠近基类型,也可以靠近指针变量名。不同的写法表明了程序员对于符号(*)理解上的差异,但它们的语法都是正确的。例如:

```
double* pd1;                    //定义了一个变量 pd1,它是一个指向 double 类型的指针
double *pd2;                    //定义了一个指针变量 pd2,其基类型是 double
```

若把 double* 当作一种类型——"指向 double 类型的指针类型",有时会引起理解上的错误。例如:

```
double* pd1,pd2,pd3;
```