第5单元　继　　承

继承（inheritance）相当于生物界的血缘关系。在面向对象程序设计中，通过继承形成类与类之间的层次关系，使一个下层（新）类自动地拥有上层（既有）类的成员，并以此为基础进行扩充或修改，实现代码复用。通常把既有类称为基类（base class）或超类（super class），把新类称为派生类（derived class）或子类（subclass）。它们之间的关系，可以称之为：子类继承自超类（有时也称父类）或基类派生子类。

5.1　单 基 继 承

派生类只有一个基类，称为单基派生或单一继承。

5.1.1　公司人员的类层次结构模型

一个简单的公司人员体系，可能涉及 3 类对象：人（person）、职员（employee）和管理者（manager）。不管是普通员工（employee），还是管理人员（manager），都是员工（employee），都具有员工特征；而不管是在公司工作的人员，还是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）。从而形成一种包含关系：人⊇职员⊇管理者。图 5.1 所示的类图用向上的箭头描述了本题中 3 个类之间的继承关系，管理者继承自职员，职员继承自人；也描述了由特殊（specialization）向一般（generalization）的关系，即泛化关系，职员是管理者的泛化，人是职员的泛化。这一单元将介绍这种类之间关系的描述和其中的规则。

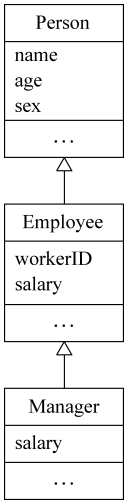

图 5.1　公司人员体系模型

5.1.2　C++继承关系的建立

在 C++语言中，派生关系用下面的格式描述。

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

下面看一个从 person 类派生 Employee 类的例子。

代码 5-1 Person 类声明。

```
//文件名：person.h
#include <string>
using namespace std;

class Person{
private:
    string  name;
    int     age;
    char    sex;
public:
    Person (string name,int age,char sex);
    ~Person(){}

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

代码 5-2 Person 类实现。

```
//文件名：person.cpp
#include "person.h"
#include "employee.h"
#include <iostream>

Person::Person (string name,int age,char sex):name (name),age (age),sex (sex) {}
void Person::output(){
    cout << "姓名：" << name << endl;
    cout << "年龄：" << age << endl;
    cout << "性别：" << sex << endl;
}
```

定义了 Person 类之后，可以以其为基类，派生类 Employee。

代码 5-3 派生类 Employee 的声明。

```
//文件名：employee.h
class Employee : public Person {
private:
    unsigned int   workerID;                                    //新增成员
    double         basePay;                                     //新增成员
public:
    Employee (string name,int age,char sex,unsigned int workerID, double basePay);
    ~Employee(){}
    unsigned int   getWorkerID() const {return workerID;}  //新增成员
    double         getBasePay() const {return basePay;}    //新增成员
};
```

代码 5-4 Employee 类的实现。

```
//文件名：employees。cpp
#include "person.h"
#include "employee.h"

Employee :: Employee (string name, int age, char sex, unsigned int workerID, double basePay)
              :Person (name, age, sex), workerID (workerID), basePay (basePay) {}
```

代码 5-5 测试代码。

```
#include "person.h "
#include "employee.h"
#include <iostream>

int main(){
    Employee e1("AAAAA", 26, 'f', 123456, 1234.56);
    cout << "\n-------执行 e1.output()的情形----------------------\n";
    e1.output ();
    cout << "\n-------执行 e1.Person::output()的情形-------------\n";
    e1.Person::output ();
    cout << "\n-------执行 e1.Employee::output()的情形-----------\n";
    e1.Employee::output ();

    return 0;
}
```

测试结果：

```
-------执行e1.output ()的情形----------------------
姓名：AAAAA
年龄：26
性别：f

-------执行e1.Person::output ()的情形-------------
姓名：AAAAA
年龄：26
性别：f

-------执行e1.Employee::output ()的情形-----------
姓名：AAAAA
年龄：26
性别：f
```

说明：派生类是通过对基类的继承、修改和扩充而形成的。下面对这三点进一步说明。

（1）在本例中，派生方式使用了关键字 public。这种派生称为公有派生。公有派生具有如下基本特点。

- 基类的公开成员被派生类继承为公开成员。所以可以使用表达式 e1.output()。
- 基类的私密成员虽然也被派生类继承，但成为隐藏的成员。所以不可以使用表达式 e1.name。
- 构造函数和折构函数都不可继承。所以 Employee 类中需要定义自己的构造函数。

除了公有派生，C++还允许使用 private（私有）派生和 protected（保护）派生。不同的派生方式使得派生类对象的特征有所不同。例如私有派生将使基类的公开成员成为派生类

的私密成员。由于私有派生和保护派生很少使用，所以本书不作介绍。

（2）在派生类中除了必须添加自己需要的构造函数外，还可以增添其他数据成员和成员函数。这就是对基类的扩充。如在代码 5-2 中，扩充了成员变量 wokerID 和 basePay，以及成员函数 getWokerID( )和 getBasePay( )。

（3）在派生类中还可以修改基类中的成员函数。例如再增添一个与基类中同名的 output( )，但功能是增添了 wokerID 和 basePay 的输出。关于这部分将在 5.1.3 节中介绍。

（4）“::”称为作用域运算符。在本例中基类 Person 和派生类 Employee 中都有 output( )，操作符::用于区分调用的 output( )是哪个类的成员函数。

（5）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。

（6）unsigned int 称为无符号 int 类型，其只取正值，即最小为 0，最大为 int 的 2 倍。

（7）在本例的各个类中，有一些成员函数都在声明的同时，在类中给出了定义。这样的函数称为内联（inline）函数。如图 5.2 所示，非内联函数有一个调用—返回的过程。每调用一次，就把流程转移到函数代码一次。函数代码执行结束后，要返回调用处。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，不再形成调用—返回过程，效率比较高，适合代码比较短且会多次调用的函数。

(a) 非内联函数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

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 5-6　显式内联函数的例子。

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

inline double Circle :: calcPerimeter() { //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。

（8）派生类 Employee 还可以再派生新的类 Manager。

代码 5-7 派生类 Manager 的声明。

```
//文件名：manager.h
class Manager : public Employee {
private:
    string  post;                                  //新增成员
public:
     Manager (string name,int age,char sex,
              unsigned int workerID,double basePay,string post)
     ~Manager () {}
     string getPost ()const {return post;}      //新增成员

};
```

代码 5-8 派生类 Manager 的实现。

```
//文件名：person.cpp
#include "person.h"
#include "employee.h"
#include "manager.h"

Manager :: Manager (string name,int age,char sex,unsigned int workerID,double basePay,
           string post):Employee(name,age,sex,workerID, basePay),post (post)
```

从继承体系中类的组织可以看出，按照项目并把每个类分成声明和实现，非常便于程序的扩展。

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。

代码 5-9 output 函数的重定义与实现。

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

class Person {
    …
    void output();
};
```

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

class Manager:public Employee {
    …
    void output();
};

void Person::output(){
    cout << "姓名：" << name << endl;          //直接使用
    cout << "年龄：" << age << endl;           //直接使用
    cout << "性别：" << sex << endl;           //直接使用
}

void Employee::output(){
    Person::output();
    cout << "工号：" << workerID << endl;
    cout << "工资：" << basePay << endl;       //直接使用
}

void Manager::output(){
    Employee::output();
    cout << "职位：" << post << endl;          //直接使用
}
```

说明：

（1）从上述代码可以看出，派生类虽然可以继承基类的成员，但它毕竟对于基类来说是“外部”，因此对于基类的私密成员，只能继承，不可由其成员直接调用，要使用基类的私密成员也须借助基类的公开函数间接使用。

（2）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。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会被覆盖——屏蔽。如果还想在派生类中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来指定其作用域。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()
```

（3）重定义与重载是两个不同的概念。重载是靠参数区分，而重定义函数靠类域区分，因为重定义函数的名字和参数必须相同，而重载参数要求名字相同，参数必须不同。

（4）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，否则在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同名函数。

5.1.4 基于血缘关系的访问控制——protected

前面介绍了若一个类的成员采用 private 和 public 访问保护，在 public 派生时，基类的 private 成员在派生类中将不可访问。这就带来了许多不便。例如，在代码 5-9 中，派生类若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，使某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，则不能访问这些成员，即做到血缘内外有别呢？这就要用 protected 进行访问控制了。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，访问控制就被分为下述 3 个级别了。

（1）private：访问权限仅限于本类的成员。

（2）proteted：访问权限扩大到本血缘关系内部。

（3）public：访问权限扩大到本血缘关系外部。

代码 5-10 若将上述 Person 类和 Employee 类中的 private 改为 protected，则代码 5-9 可以改写为如下形式。

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

class Person {
protected:
    string  name;
    int     age;
    char    sex;
public:
   …
    void output();
};

class Employee:public Person {
protected:
    unsigned int   workerID;
    double         basePay;
public:
    …
    void output();
};
```

```
class Manager:public Employee {
private:
    string  post;                           //新增成员
public:
    …
    void output();
};

void Person::output(){
    cout << "姓名：" << name << endl;
    cout << "年龄：" << age << endl;
    cout << "性别：" << sex << endl;
}

void Employee::output(){
    cout << "姓名：" << name << endl;       //保护成员，直接调用
    cout << "年龄：" << age << endl;        //保护成员，直接调用
    cout << "性别：" << sex << endl;        //保护成员，直接调用
    cout << "工号：" << workerID << endl;
    cout << "工资：" << basePay << endl;
}

void Manager::output(){
    cout << "姓名：" << name << endl;       //保护成员，直接调用
    cout << "年龄：" << age << endl;        //保护成员，直接调用
    cout << "性别：" << sex << endl;        //保护成员，直接调用
    cout << "工号：" << workerID << endl;   //保护成员，直接调用
    cout << "工资：" << basePay << endl;    //保护成员，直接调用
    cout << "职位：" << post << endl;
}
```

5.1.5 类层次结构中构造函数和析构函数的执行顺序

在 Person 类、Employee 类和 Manager 类的构造函数、析构函数中分别加入输出语句：

```
cout << "执行 Person 构造函数。" << endl;
cout << "执行 Person 析构函数。" << endl;
cout << "执行 Employee 构造函数。" << endl;
cout << "执行 Employee 析构函数。" << endl;
cout << "执行 Manager 构造函数。" << endl;
cout << "执行 Manager 析构函数。" << endl;
```

如：

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

```
    Person (string name,int age,char sex):name(name),age(age),sex(sex){
         cout << "执行 Person 构造函数。"<< endl;     //输出提示
    }

    ~Person () {
     cout << "执行 Person 析构函数。"<< endl;         //输出提示
    }
//其他代码
};
```

代码 5-11 测试公司人员类层次中构造函数和析构函数的执行顺序。

```
#include <iostream>
int main(){
    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");
    cout << "\n-------执行 m1.output()的情形---------------------\n";
    m1.output();
    cout << "\n-------执行 m1.Employee::output()的情形-----------\n";
    m1.Employee::output();
    cout << "\n-------执行 m1.Person::output()的情形-------------\n";
    m1.Person::output();
    cout << "\n-------撤销 Manager 对象时析构函数调用顺序----------\n";

    return 0;
}
```

测试结果如下。

```
-------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，构造函数执行顺序

撤销Manager m1对象过程中，析构函数执行顺序

说明：

（1）在多层类层次结构中，派生类的构造函数须调用直接基类的构造函数，以构建所

继承的基类分量，但不能调用间接基类的构造函数。即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——初始从信息传递阶段，即沿继承链而上，把初始化信息传递到上层各类的构造函数，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，沿派生链向下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，在声明一个 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。

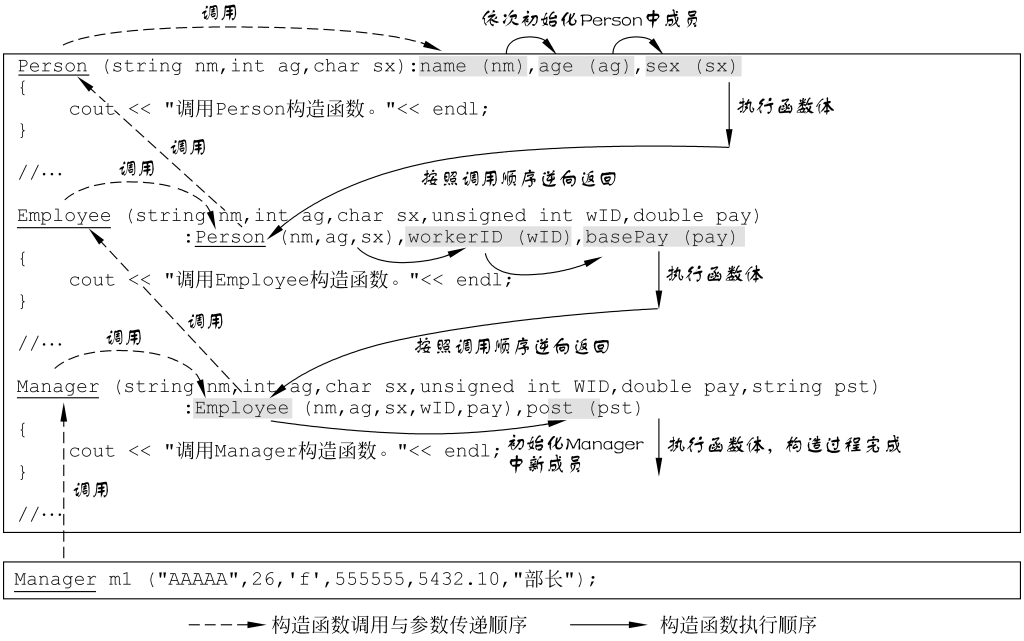

图 5.3　生成 Manager m1 过程中构造函数的调用顺序

（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是从下向上进行的，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。

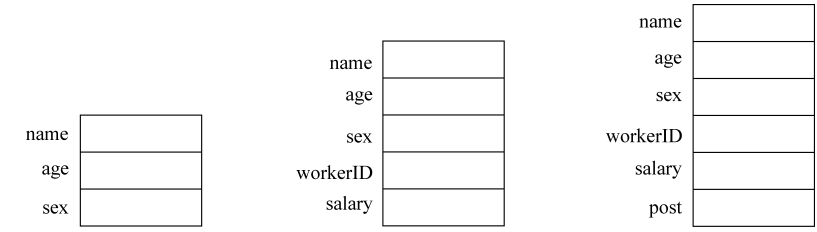

图 5.4　Manager m1 对象的内存分配过程