第5章 继承与派生

本章要点：

- 继承与派生的概念
- 派生类的构造函数和析构函数的执行顺序与规则
- 多继承的声明与实现
- 基类成员访问原则
- 赋值兼容性
- 虚基类的概念

计算机软件技术发展到今天，各种软件系统已经越来越复杂、软件开发也越来越困难，软件复用(software reuse)技术日益受到人们的重视，软件复用一方面能够降低软件开发的工作量和成本、提高开发效率，另一方面也能够提高软件的可靠性。

传统的非面向对象程序设计也在软件复用的问题上做了一定的努力，当用户定义的已有数据结构和功能无法满足新的需求时，通常的办法就是改写甚至重写这些已经写好的程序，C 语言在这方面常用的解决方法就是代码复制或者是使用程序库，但是这样的代码重用效率很低，造成了不必要的资源浪费；另外，传统的程序设计不能很好地体现程序间层次关系的思想。

继承(inheritance)机制是面向对象技术提供的另一种解决软件复用问题的途径，即在定义一个新的类时，先把一个或多个已有类的功能全部包含进来，然后再给出新功能的定义或对已有类的某些功能重新定义。继承不需要修改已有软件代码，很好地体现了程序的相关性，又实现了程序的可扩充性，它是种基于目标代码的复用机制，本章介绍有关继承的基本内容。

5.1 继承与派生的概念

继承是面向对象程序设计中重要的特性。继承主要是指在已有类(或称为基类)的基础上创建新类的过程，这个新类就是派生类。派生类自动包含了基类的成员，包括所有的数据和操作，而且它还可以增加自身新的成员。

在 C++语言中，一个派生类可以从一个基类派生，也可以从多个基类派生，从一个基类派生的称为单继承，如图 5.1 所示。图 5.2 中的树状结构图可以体现学生体系的概念。

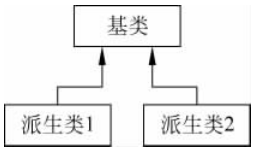

图 5.1　单继承

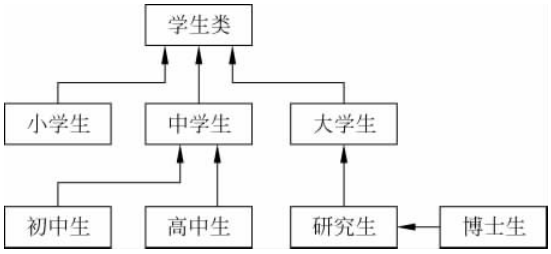

图 5.2　类的层次树状结构图

一个派生类从两个或多个基类派生则称为多继承，如图 5.3 所示，它使一个类可以融合多个类的特征，例如在现实生活中的在职研究生的概念就是一个多继承的例子，他是在职人员，又是研究生，如图 5.4 所示，从图中还可以看到一个有趣的现象，在职人员类本身是单继承的基类，教师和职员都是它的具体子类，而其又是在职研究生的多重基类，提供在职人员的基本特征。

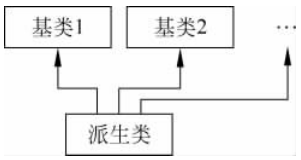

图 5.3　多继承

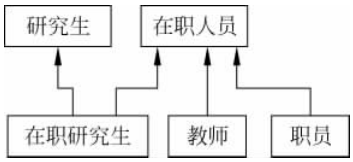

图 5.4　单继承与多继承

由以上可以看出，只要处理好类的层次关系，就可以生成各种有用的类，最大程度地实现软件重用。

继承机制除了支持软件复用外，还具备以下 3 个作用：

（1）对事物进行分类。可以把基类看成子类的共性抽象，换一个角度，即基类表达了该种类型的一般概念，而子类则表达了一个特殊概念。

（2）支持软件的增量开发。软件开发通常不是一次完成的，往往是一个逐步升级、细化、改进的过程，派生类和基类的关系恰好体现了这种改进和被改进的关系。

（3）对概念进行组合。前面内容提过可以为类定义对象成员，这就是一种组合关系，而继承机制中，基类成为派生类的一部分，其实也是一种组合的概念，尤其在多继承关系上，多个基类共同构成同一派生类，此问题体现得更加明显。

前面已知如何定义类和如何实现类的抽象与封装，通常在不同的类中，数据成员和函数成员都是不同的，但对于某些特定的问题，有时候两个类的基本或大部分内容是相同的，在图 5.2 中给出了学生体系的概念，我们利用现有知识可以首先声明一个类来描述学生这一基本概念，代码如下：

```
class Student
{
private:
    int number;                    //学号
    string name;                   //姓名
public:
```

```
    Student()
    {
        number = 0;                    //学号初值=0
        name = "";                     //姓名初值为空字符串
    }
    void SetValue(int n, string s1) //修改成员变量
    {
        number = n;
        name = s1;
    }
    void Print()                       //打印输出学号、姓名
    {
        cout << "Number:" << number << endl;
        cout << "Name:" << name << endl;
    }
};
```

如果现在需要一个新类 Undergraduate 来描述大学生的概念,除上述的基本成员外,还需要用到年龄、年级等信息,可以如下定义此类:

```
class Undergraduate
{
private:
    int number;
    string name;
    int age;
    int grade;
public:
    void Print()
    {
        cout << "Number:" << number << endl;
        cout << "Name:" << name << endl;
        cout << "Age:" << age << endl;
        cout << "Grade:" << grade << endl;
    }
};
```

观察以上两个类,可以看到二者存在一定的关系,类 Undergraduate 是在 Student 类基础上扩充而来的,也可以说 Undergraduate 是 Student 的一种特例,具有 Student 的特征,并拥有自己的特殊特征。两个类的程序也存在着很大的相似性,那么利用继承机制该如何处理此问题呢?

5.2 派生类的声明

在 C++语言中,类的继承关系可以用如下语法表示:

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

要求基类名必须是一个已经声明的类。其中{}内的部分用来定义派生类新增加的成员,或者是基类中原来已有但是在派生类做了一定的修改的成员。继承方式,也称访问控制方式,用来限定紧随其后的基类,包括三种方式:public、protected、private,如果没有显式使用这三个关键字之一进行声明,则系统默认为私有继承(private)。类的继承方式指定了派生类成员函数以及类的对象对于从基类继承来的成员的访问权限,或者说决定了是基类成员在派生类中访问控制方式的变化。

需要注意的是,基类的构造函数和析构函数不能被派生类继承,派生类若要初始化基类的数据成员必须在构造函数中初始化。

5.1节介绍了Student和Undergraduate类,可以用继承机制来改写Undergraduate类。

【例5.1】 用继承重新定义Undergraduate类。

```
/* 01_01.cpp */
#include<iostream>
#include<string>
using namespace std;
//定义基类 Student
class Student
{private:
    int number;                                    //学号
    string name;                                   //姓名
public:
    Student()
    {
        number=0; name="";                         //学号、姓名设置初值
    }
    void SetValue(int n,string s1)                 //修改成员变量
    {
        number=n;name=s1;
    }
    void Print()                                   //打印输出学号、姓名
    {
        cout<<"Number:"<<number<<endl;
        cout<<"Name:"<<name<<endl;
    }
};
//定义派生类 Undergraduate
class Undergraduate : public Student
{
private:
    int  age;                                      //新增成员:年龄
    int  grade;                                    //新增成员:年级
public:
    Undergraduate()                                //不带参数构造函数
    {age=0; grade=1;
    }
    Undergraduate(int n, string s1, int a, int g)  //带参数构造函数
    {
        SetValue(n, s1);                           //调用基类成员函数修改学号、姓名
```

```
        age = a; grade = g;
    }
    void PrintExtra()                          //打印新增的数据成员信息
    {
        cout <<"Age:"<< age << endl;
        cout <<"Grade:"<< grade << endl;
    }
};
//下面用主函数进行测试:
int main()
{
    Undergraduate st1(100,"wang",18,1);
    st1.Print();                               //调用基类的函数
    st1.PrintExtra();                          //调用派生类新定义的函数
    system("pause");
    return 0;
}
```

```
Number:100
Name:wang
Age:18
Grade:1
请按任意键继续. . .
```

图 5.5 例 5.1 的调试运行结果

程序的调试运行结果如图 5.5 所示。

由以上程序可以看出,扩展原有的类变得非常容易,派生类可以在自己的成员函数中访问来自基类的成员,也可以通过“派生类对象.成员”的方式来访问基类成员,这时基类成员已经变成了派生类的成员。如图 5.6 所示,派生类中成员分为两部分,一部分继承自基类,一部分是新增加的,每一部分都有数据成员和函数成员。

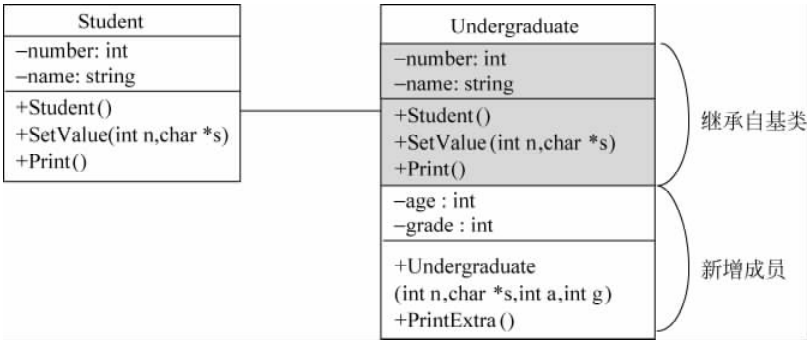

图 5.6 基类与派生类中的成员

在派生过程中,一个基类可以同时派生出多个派生类。此外,派生出来的新类也同样可以作为基类再继续派生新的类,也就是说,一个类从父类继承来的特征也可以继续向下传递,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族体系。在这样的类族体系中,直接派生出某类的基类称为该类的直接基类,基类的基类甚至更高层的基类也称为间接基类。如图 5.2 中,大学生类是研究生类的直接基类,而学生类则是研究生类的间接基类,在例 5.1 基础上,可以继续定义研究生类:

```
class GraduateStudent : public Undergraduate        //研究生类派生自大学生类
{
protected:
```

```
    string researchField;                    //新增成员：研究领域
public:
    GraduateStudent(string sf)               //构造函数
    {
        researchField = sf;
    }
};
```

5.3 派生类的访问属性

除了基类的构造函数、析构函数和赋值运算符函数，派生类继承了基类所有的数据成员和其他成员函数，继承之后，基类成员的访问控制权限在派生类中会发生一定变化。如基类中的 private 成员在派生类中的访问权限变得更低，已经无法访问，而基类的 public 成员在派生类中不一定还是 public 权限，这些变化会直接影响到派生类对于基类成员的访问。

在不同的继承方式下，基类成员的访问权限在派生类中将会发生什么变化？先暂不考虑静态函数和友元函数这两种情况，针对普通的成员函数，主要着眼于两个方面：派生类的成员函数和派生类的对象能够访问基类中哪些权限的成员？

类的成员可以分为 public（公有）、protected（保护）和 private（私有）三种访问权限。类的非静态成员函数可以访问类中的所有成员，但是通过类的“对象.成员”方式（在类的作用域之外），则只能访问该类的公有成员。类的继承方式，有公有继承（public）、保护继承（protected）和私有继承（private）三种。不同的继承方式导致原有基类成员在派生类中的访问属性也有所不同。表 5.1 中列出三种继承方式下，派生类对于基类成员的访问控制规则。

表 5.1 不同继承方式下的访问控制权限

基类成员的权限	继承方式		
	public	**protected**	**private**
public	在派生类中为 public	在派生类中为 protected	在派生类中为 private
	派生类的成员函数和类的作用域之外，都可以直接访问	派生类的成员函数可以直接访问	派生类的成员函数可以直接访问
protected	在派生类中为 protected	在派生类中为 protected	在派生类中为 private
	派生类的成员函数可以直接访问	派生类的成员函数可以直接访问	派生类的成员函数可以直接访问
private	在派生类中被隐藏，无法访问	在派生类中被隐藏，无法访问	在派生类中被隐藏，无法访问
	任何方式都不能直接访问，但可以通过基类的 public、protected 成员函数间接访问	任何方式都不能直接访问，但可以通过基类的 public、protected 成员函数间接访问	任何方式都不能直接访问，但可以通过基类的 public、protected 成员函数间接访问

从表 5.1 中可以看出三种不同继承方式下的访问控制权限：

(1) public 继承时，基类成员的访问控制权限除私有成员外，在派生类中保持不变。

派生类的成员函数可以直接访问基类中的 public、protected 成员，以及本类所有权限的成员，基类的 private 私有成员虽然已经继承到派生类里，但是却无法实现直接访问，可以通过基类的 public、protected 成员函数访问。

在类的作用域之外的派生类对象只能访问基类的 public 成员和本类的 public 成员。

(2) protected 继承时，基类成员的 public 访问权限在派生类中变为 protected。

派生类的成员函数可以直接访问基类中的 public、protected 成员，以及本类所有权限的成员，不能访问的是基类的 private 成员，可以通过基类的 public、protected 成员函数访问。

在类的作用域之外的派生类对象不能访问基类所有权限的成员，但可以访问本类的 public 成员。

(3) private 继承时，基类成员的 public 和 protected 访问权限在派生类中变为 private。

派生类的成员函数可以直接访问基类中的 public、protected 成员，以及本类所有权限的成员，不能访问的是基类的 private 成员，但可以通过 public、protected 成员函数访问。

在类的作用域之外的派生类对象不能访问基类所有权限的成员，但可以访问本类的 public 成员。

另外要注意的一点是，如果继承时不写继承方式，则默认为私有继承，例如：

```
class Derived: Base
{  …  };
```

则该继承等价于：

```
class Derived: private Base
{  …  };
```

总结这三种继承方式的访问控制权限，可以发现：

(1) 基类的 private 成员在基类中任何方式都不能直接访问，只能通过基类的成员函数。

(2) 在 private、protected 继承方式下，基类成员的权限都发生较大的变化，只有在特殊要求的情况下才会使用，如希望在派生类的对象在类的作用域之外无法访问基类的 public 成员，就可以采用 protected 方式，再或者为了防止派生类还能被继续派生，就可以利用 private 方式把基类的访问权限都变为 private，这样如果本派生类被非法取得并被继续派生，则原基类的成员在新派生类中就全部都无法访问了，这在一定程度上对原基类的使用许可起到了保护的作用。

对于静态成员来说，与普通成员函数组合，将产生以下两种情况：

(1) 派生类中静态函数对基类中静态成员的访问；

(2) 派生类的普通成员函数要访问基类中的静态成员。

静态成员的访问控制变化完全遵循表 5.1 的规则，这两种情况和派生类中普通成员函数访问基类中普通成员没有区别。

为了说明访问控制的变化情况，以公有继承 public 为例，基类的 public、protected 成员

在派生类中保持访问控制权限不变，而基类的 private 成员不可直接访问。在类的作用域之外的派生类对象，则只能访问从基类继承得到的 public 成员，以及派生类自己的 public 成员。

【例 5.2】 公有继承时的访问控制权限。

```
/* 05_02.cpp */
#include <iostream>
using namespace std;
class Base                                  //定义基类 Base
{private:
    int a;                                  //基类私有成员变量 a
    void Fun1()                             //基类私有成员函数 Fun1()
    {  cout << a << endl;  }
protected:
    int b;                                  //基类保护成员变量 b
    void Fun2()                             //基类保护成员函数 Fun2()
    {  cout << c << endl;  }
public:
    int c;                                  //基类公有成员变量 a
    void Fun3()                             //基类公有成员函数 Fun3()
    {  cout << b << endl;  }
    void Seta(int i)                        //共有成员函数 Seta(),可以修改私有成员 a 的值
    {   a = i;  }
    int   Geta()                            //公有成员函数 Geta(),返回私有成员 a 的值
    {  return a; }
    Base(int i, int j, int k)               //基类的构造函数
    {  a = i; b = j; c = k;  }
};
class Sub : public Base                     //定义派生类
{private:
    int d;                                  //派生类的私有成员 d
public:
    Sub(int i, int j, int k, int m) :Base(i,j,k)  //派生类构造函数,调用基类构造函数
    {  d = m;  }
    void Test()
    {   //cout << a << endl;               //错误,无法访问基类的私有成员
        cout << b << endl;                  //正确,可以访问基类的保护成员
        cout << c << endl;                  //正确,可以访问基类的公有成员
        //Fun1();                           //错误,无法访问基类的私有成员
        Fun2();                             //正确,可以访问基类的保护成员
        Fun3();                             //正确,可以访问基类的公有成员
        Seta(10);                           //正确,间接访问基类成员 a
        cout << d << endl;                  //正确,可以访问派生类的私有成员
    }
};
int main()
{
    Base b1(5,6,7);                         //定义基类对象 b1
    //cout << b1.a;                         //错误,无法访问对象的私有成员
    //cout << b1.b;                         //错误,无法访问对象的保护成员
```

```
    cout << b1.c << endl;                    //正确,可以访问对象的公有成员 c
    cout << b1.Geta()<< endl;                //正确,间接访问对象的私有成员 a
    Sub s1(11,15,19,22);                     //定义派生类对象 s1
    s1.Test();                               //正确,可以访问对象的公有成员
    s1.c = 200;                              //正确,可以访问对象的公有成员
    s1.Fun3();                               //正确,可以访问对象的公有成员
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.7 所示。

```
7
5
15
19
19
15
22
15
请按任意键继续. . .
```

图 5.7 例 5.2 的调试运行结果

在上述程序中一些语句已经用“//”注释了,这些语句不符合访问控制,如果去掉“//”在编译时就会出错。在派生类的公有成员函数 Test()中,可以访问本类所有的成员,以及基类中除了 private 权限的成员。因此对于基类数据成员 a 和成员函数 f 的访问是不允许的,但是对于基类中 public、protected 权限成员 c、b、Fun2 ()和 Fun3 ()的访问能够正常进行,另外,对于基类中的私有成员,可以通过基类中 public 或 protected 函数来实现访问,如 Base 类中定义的 Geta()和 Seta()就实现了对私有变量 a 的间接访问。在主函数 main()还实现了“对象.成员”方式的访问,派生类对象利用这种方式就实现了对基类公有成员的访问。

对于保护继承和私有继承这两种方式,本书就不再进行举例说明了,感兴趣的读者可以修改例 5.2 的程序,进行实验,以加深对于三种继承方式的理解。

5.4 派生类的构造函数和析构函数

由前面的内容我们知道,用户在声明类时如果不定义构造函数,系统会自动提供一个默认的构造函数,在定义类对象时会自动调用这个默认的构造函数。这个构造函数实际上是一个空函数,没有形参,也不会执行任何操作。需要自己定义构造函数才能实现对数据成员的初始化。

在继承机制中,基类的构造函数和析构函数是不能继承的,也就是说,基类的构造函数不能作为派生类的构造函数,派生类的构造函数负责对来自基类数据成员和新增加的数据成员进行初始化,所以在执行派生类的构造函数时,需要调用基类的构造函数。

5.4.1 派生类构造函数和析构函数的执行顺序

通过继承,派生类得到了基类的成员,因此派生类对象中既包括自身类的数据成员还包括通过继承从基类中得到的数据成员。在派生类中还可用其他类来定义对象作为成员,又涉及派生类中对象成员的构造问题,则当用派生类定义对象后,派生类对象、对象成员、基类对象的构造函数的调用顺序如下:

(1) 基类的构造函数;

(2) 对象成员的构造函数(如果有的话)有多个时按声明的顺序;

(3) 派生类的构造函数。

【例 5.3】 派生类构造示例。

```
#include<iostream>
using namespace std;
class B                                    //基类 B
{
public:
    B(){  cout<<"Construct B"<<endl;  } //基类 B 的构造函数
};
class C                                    //C 类
{
public:
    C(){  cout<<"Construct C"<<endl;  } //C 类的构造函数
};
class D : public B                         //派生类 D
{
private:
    C c1;                                  //对象成员 c1
public:
    D(){  cout<<"Construct D"<<endl;  } //派生类构造函数
};
int main()
{
    D d1;                                  //定义派生类对象 d1
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.8 所示。

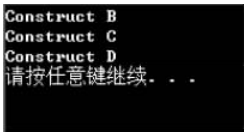

图 5.8 例 5.3 的调试运行结果

析构函数与构造函数执行的顺序相反,将按如下顺序执行:

(1) 派生类的构造函数;

(2) 对象成员的构造函数(如果有的话)有多个时与声明的顺序相反;

(3) 基类对象的析构函数。

对例 5.3 中的程序进行改造,为每一个类添加析构函数,然后再进行测试。

```
class B
{
public:
    B(){  cout << "Construct B" << endl;  }     //基类 B 构造函数
    ~B(){  cout << "Destruct B" << endl;  }    //基类 B 析构函数
};
class C
{
public:
    C(){  cout << "Construct C" << endl;  }     //C 类构造函数
    ~C(){  cout << "Destruct C" << endl;  }    //C 类析构函数
};
```

```
class D : public B                                  //派生类 D
{
private:
    C c1;                                           //对象成员 c1
public:
    D(){   cout << "Construct D" << endl;   }       //派生类构造函数
    ~D(){   cout << "Destruct D" << endl;   }       //派生类析构函数
};

int main()
{
    D d1;                                           //定义派生类对象 d1
    return 0;
}
```

程序的运行结果如图 5.9 所示,注意:在 VS 2015 中,此处用到非调试运行。

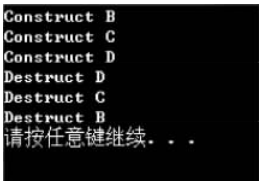

图 5.9 例 5.3 的调试运行结果

5.4.2 派生类构造函数和析构函数的构造规则

在 C++语言中,类的机制非常清楚、严格地划分了各自的权限和责任。是哪个类的操作,必须由哪个类调用;是谁的对象,就必须由该类的构造函数来完成对其构造的工作。因此,对派生类中基类成员的构造,必须由基类构造函数完成,而不能由派生类的构造函数越权去构造。派生类构造函数主要负责调用基类构造函数并提供基类构造函数所需的参数。

下面分两种情况讨论派生类对象的构造:

(1) 如基类中定义了默认构造函数,且该默认构造函数能够完成派生类对象中基类成员的构造,则派生类构造函数无须显式调用基类构造函数,直接调用基类的默认构造函数即可,这是一种较为简单的情况,例 5.3 中的继承就属于此类情况,下面的例 5.4 也可以说明这一点。

【例 5.4】 分析下面程序的输出结果。

```
/* 05_04.cpp */
#include<iostream>
using namespace std;
class Base                                          //基类
{
public:
    Base(){ a=0; }                                  //不带参数的缺省构造函数
    Base(int i){ a=i; }                             //带参数的构造函数
protected:
```

```
    int a;
};
class Derived : public Base                    //派生类
{
public:
    Derived(){ b = 0; }                        //派生类的不带参数缺省构造函数
    Derived(int i) { b = i; }                  //派生类的带参数构造函数
    void Print()                               //打印数据
    {   cout <<"a = "<< a <<",b = "<< b << endl;  }
private:
    int b;
};
int main()
{
    Derived d1;                                //定义派生类对象 d1,调用不带参数构造函数
    Derived d2(12);                            //定义派生类对象 d2,调用带参数构造函数
    d1.Print();
    d2.Print();
    system("pause");
    return 0;
}
```

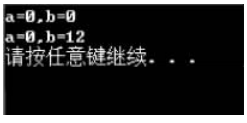

图 5.10　例 5.4 的调试运行结果

程序的调试运行结果如图 5.10 所示。

说明：在程序中派生类 Derived 内定义了两个构造函数，都没有显式地调用基类的构造函数，其实它们都隐式地调用了基类 Base 中的不带参数的缺省构造函数，由于不需要任何参数，所以在派生类的构造函数省略了对它的调用语句。

注意：如果基类中没有重新定义任何构造函数，本程序也是完全合法的，构造基类对象会使用系统提供的默认构造函数。但是如果基类中，没有重新定义缺省的构造函数，却定义了其他带参数的构造函数，本程序在编译时就会发生错误，原因就是派生类构造函数中隐式地调用基类的缺省构造函数，但是后者并不存在，因此产生编译错误。

（2）若基类中定义了有参数的构造函数，或者所定义的默认构造函数不能完成基类成员的构造，则必须通过派生类构造函数显式地调用基类的构造函数，向带参数的构造函数传递参数，这需要用到“成员初始化列表”的语法。

另外，对于派生类中普通数据成员的初始化，以及对象成员的构造，也可以放在成员初始化列表中完成。此时，这些以逗号隔开的各种初始化项的顺序可以是任意的。

因此派生类构造函数定义的一般格式如下：

```
派生类名(参数列表):基类构造函数(参数列表 1),子对象成员(参数列表)…
{
    派生类构造函数体
}
```

【例 5.5】 派生类构造函数示例。

```
/* 05_05.cpp */
#include <iostream>
```

```
#include <cstring>
using namespace std;
class Date                                          //日期类
{
private:
    int year,month,day;                             //年、月、日成员变量
public:
    Date(int y = 2009,int m = 6,int d = 10)         //构造函数
    {
        year = y;month = m;day = d;
    }
    void Print()                                    //输出数据,格式为: 年 - 月 - 日
    {
        cout << year <<" - "<< month <<" - "<< day << endl;
    }
};
class Student                                       //定义学生基类
{
protected:
    int number;                                     //数据成员
    string name;
    char sex;
public:
    Student()                                       //重定义的默认构造函数
    {
        number = 0;
        name = "No name";                           //默认名字
        sex = 'M';                                  //默认性别,男性 (Male)
    }
    Student(int n, string s, char x)                //带参数的构造函数
    {
        number = n;
        name = s;
        sex = x;
    }
};
//大学生派生类
class Undergraduate : public Student
{
public:
    Undergraduate(int n, string s, char x, int a, int y, int m, int d) :
        Student(n, s, x), birth(y, m, d)            //调用基类构造函数和对象成员的构造函数
    {
        age = a;
    }
    Undergraduate()                                 //此处省略了 Student()调用
    {
        age = 0;
    }
    void Print()                                    //输出信息
    {
```

```
        cout <<"number:"<< number << endl;
        cout <<"name:"<< name << endl;
        cout <<"sex:"<< sex << endl;
        cout <<"age:"<< age << endl;
        cout <<"birthday:";
        birth.Print();
    }
private:
    int age;
    Date birth;                                    //对象成员
};
//主函数
int main()
{
    Undergraduate st1;                             //用派生的默认构造函数定义对象
    Undergraduate st2(1001,"Zhang",'F',20,2009,6,11);  //带参数构造
    st1.Print();
    st2.Print();
    system("pause");
    return 0;
}
```

图 5.11　例 5.5 的调试运行结果

程序的调试运行结果如图 5.11 所示。

说明：在以上程序中，定义了 3 个类，Undergraduate 和 Student 类构成继承关系，在 Undergraduate 类中定义了带参数的构造函数和默认构造函数，并且还定义了一个日期类 Date 的对象成员，Undergraduate 类带参数的构造函数中，利用参数初始化表实现了基类对象的构造和对象成员的构造。

5.4.3　C++ 11 继承构造函数

1. C++ 11 继承构造函数

在 C++ 11 标准中，派生类能够重用其直接基类定义的构造函数。构造函数并非以常规的方式继承而来，一个类只初始化它的直接基类，出于同样的原因，一个类也只继承其直接类的构造函数。类不能继承默认、拷贝和移动构造函数。如果派生类没有直接定义这些构造函数，则编译器将为派生类生成它们。

派生类 Derived 继承基类 Base 构造函数的方式是提供一条注明了（直接）基类的 using 声明语句：

```
calss Derived: public Base
{
  public:
    using Derived::Base;                           //继承 Base 的构造函数
    …
};
```

通常情况下，using 声明语句只是令某个名字在当前作用域内可见。而当作用于构造

函数时，using 声明语句将令编译器产生代码。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。即对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。

编译器生成的构造函数形如：

```
Derived(parms): Base(args)
{  }
```

其中，Derived 是派生类的名字，Base 是基类的名字，parms 是构造函数的形参列表，args 将派生类构造函数的形参传递给基类的构造函数。如果派生类含有自己的数据成员，则这些成员将被默认初始化。

2. 继承的构造函数的特点

(1) 继承的构造函数不会改变访问属性。不管 using 声明出现在哪儿，基类的私有构造函数在派生类中还是一个私有构造函数，受保护的构造函数和公有构造函数也是同样的规则。

(2) 当一个基类构造函数含有默认实参时，这些实参并不会被继承。相反，派生类将获得多个继承的构造函数，其中每个构造函数分别省略一个含有默认实参的形参。例如，如果基类有一个接收两个形参的构造函数，其中第二个形参含有默认实参，则派生类将获得两个构造函数：一个构造函数接收两个形参(没有默认实参)，另一个构造函数只接收一个形参，它对应于基类中最左侧的没有默认值的那个实参。

(3) 如果基类含有几个构造函数，则除了两个例外情况，大多数时候派生类会继承所有这些构造函数。

第一个例外是派生类可以继承一部分构造函数，而为其他构造函数定义自己的版本。如果派生类定义的构造函数与基类的构造函数具有相同的参数列表，则该构造函数将不会被继承。定义在派生类中的构造函数将替换继承而来的构造函数。

第二个例外是默认、拷贝和移动构造函数不会被继承。这些构造函数按照正常规则被合成。继承的构造函数不会作为用户定义的构造函数来使用，因此，如果一个类只含有继承的构造函数，则它也将拥有一个合成的默认构造函数。

有关 C++ 11 构造函数新增特性的内容，请参阅 C++ 11 标准。

5.5 多继承

5.5.1 多继承的声明

前面讲述了单继承中派生类和基类之间的关系。这一节讨论多继承问题。多继承可以看作是单继承的扩展，所谓多继承是指派生类具有多个基类。派生类与每个基类之间的关系仍可看作是一个单继承，而多继承本质是实现多个概念的合并。

多继承下派生类的声明格式如下：

class 派生类名:继承方式 1 基类名 1,继承方式 2 基类名 2,…

```
{
    派生类类体;
};
```

其中，继承方式 1、继承方式 2、……是三种继承方式 public、private 和 protected 之一。以下为最基本的定义形式：

```
class B1
{
    …
};
class B2
{
    …
};
class D:public B1,public B2
{
    …
};
```

派生类 D 具有两个基类(类 B1 和类 B2)，因此，类 D 是多继承的。按照继承的规定，派生类 D 的成员包含了基类 B1 中成员和基类 B2 中成员以及该类本身的成员，参见图 5.12 所示的 UML 图。

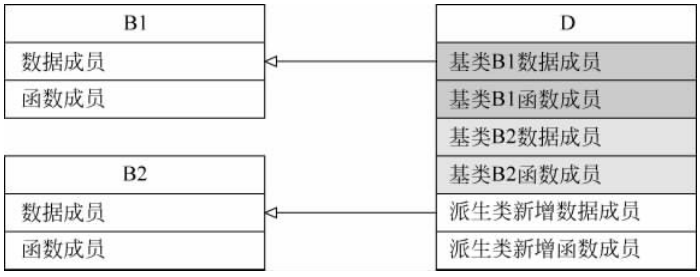

图 5.12　多继承类 D 中的成员

5.5.2　多继承的构造函数与析构函数

在多继承的情况下，派生类的构造函数格式如下：

```
派生类名(参数列表):基类名 1(参数表 1),基类名 2(参数表 2)…,子对象名(参数表 n)…
(
    派生类构造函数体;
}
```

其中，构造函数的参数列表中各个参数包含了其后的各个分参数表中所需的参数。

多继承下派生类的构造函数与单继承下派生类构造函数相似，它必须同时负责该派生类所有基类构造函数的调用，同时，派生类的参数个数必须包含完成所有基类初始化所需的参数个数。

派生类构造函数执行顺序是先执行所有基类的构造函数，再执行派生类本身的构造函

数。处于同一层次的各基类构造函数的执行顺序取决于声明派生类时所指定的各基类顺序,与派生类构造函数中所定义的成员初始化列表的各项顺序无关。相对应的是,析构函数的调用顺序与构造函数完全相反。

以图 5.4 中的在职研究生为例,说明多继承中派生类构造函数的构成及其执行顺序。

【例 5.6】 多继承示例。

```
/* 05_06.cpp */
#include <iostream>
#include <cstring>
using namespace std;
//定义研究生基类
class GStudent
{
protected:
    int number;                                          //学号
    char name[20];                                       //名字
    char sex;                                            //性别,男性: M,女性: F
public:
    GStudent (int n,char *s,char x)                      //带参数的构造函数
    {
        number = n;
        strcpy(name,s);                                  //此函数可以用 strcpy_s()替换
        sex = x;
        cout <<"Construct GStudent."<< endl;
    }
    ~ GStudent()                                         //析构函数
    {  cout <<"Destruct GStudent."<< endl;  }
};
class Employee                                           //职员类
{
protected:
    char ename[20];                                      //职员名字
    char jobname[20];                                    //工作名
public:
    Employee(char *sn,char *sj)                          //构造函数
    {
        strcpy(ename,sn);                                //此函数可以用 strcpy_s()替换
        strcpy(jobname,sj);                              //此函数可以用 strcpy_s()替换
        cout <<"Construct Employee."<< endl;
    }
    ~ Employee ()                                        //析构函数
    {  cout <<"Destruct Employee."<< endl;  }
};
//在职研究生类,从两个基类派生
class GStudentHasJob: public GStudent,public Employee
{
public:
    GStudentHasJob (int n,char *s,char x,char *sj):
                    GStudent (n,s,x),Employee(s,sj)      //调用两个基类构造函数
    {  cout <<"Construct GStudentHasJob."<< endl;  }
```

```
    ~GStudentHasJob ()                                    //析构函数
    {   cout <<"Destruct GStudentHasJob."<< endl;   }
    void Print()                                          //输出信息
    {
        cout <<"number:"<< number << endl;
        cout <<"name:"<< name << endl;
        cout <<"sex:"<< sex << endl;
        cout <<"job:"<< jobname << endl;
    }
};
//主函数
int main()
{   //定义一个在职研究生对象,并对其初始化
    GStudentHasJob st(1001,"zhang",'F',"teacher");
    st.Print();
    return 0;
}
```

```
Construct GStudent.
Construct Employee.
Construct GStudentHasJob.
number:1001
name:Zhang
sex:F
job:teacher
Destruct GStudentHasJob.
Destruct Employee.
Destruct GStudent.
请按任意键继续. . .
```

图 5.13　例 5.6 的调试运行结果

程序的运行结果如图 5.13 所示,在 VS 2015 中,此处用到非调试运行。

说明:在派生类 GStudentHasJob 的构造函数中调用了两个基类 GStudent 和 Employee 的构造函数,并且把派生类构造函数的形参传递给这两个基类构造函数,两个基类的构造函数执行顺序是先构造 GStudent 基类对象,再构造 Employee 基类对象,其执行顺序与定义派生关系时声明基类的顺序相同。如果在某一个基类中定义了不带参数的默认构造函数,则在派生类的构造函数的成员初始化表中可以省略对该基类构造函数的显式调用,从而实现对该默认构造函数的隐式调用。

由以上程序的运行结果可以看出,各析构函数执行顺序与构造函数相反,先执行派生类的析构函数,然后按照定义派生关系时声明基类的相反顺序执行各基类的析构函数。

在 C++ 11 新增构造函数的特性中,允许派生类从它的一个或几个基类中继承构造函数。同样也需要使用 using 语句声明,但是如果从多个基类中继承了相同的构造函数(即形参列表完全相同)则程序将产生错误,这个派生类必须为该构造函数定义它自己的版本。有关详细说明请参考 C++ 11 标准。

5.6　基类成员访问和赋值兼容性

5.6.1　基类成员名的限定访问和名字覆盖

若多个基类中定义有同名成员,则派生类对这些同名成员的访问可能存在冲突。为避免可能出现的成员访问冲突,需要用成员名限定的方法显式地指定要访问的成员。

【例 5.7】　成员访问冲突。

```
/* 05_07.cpp */
```

```
#include <iostream>
using namespace std;
class MP3Player                          //MP3 播放器类
{
public:
    void Play()                          //播放音乐操作
    {  cout <<"Play mp3 music."<< endl;  }
};
class VideoPlayer                        //视频播放器类
{
public:
    void Play()                          //播放视频操作
    {  cout <<"Play video."<< endl;  }
};
//新型的 MP4 播放器类
class MP4Player: public MP3Player,public VideoPlayer
{
public:
    /* ..... */
};
int main()
{
    MP4Player mp4;
    //mp4.Play();                        //去掉注释,本行将产生 Play()函数访问不明确的错误
    mp4.MP3Player::Play();
    mp4.VideoPlayer::Play();
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.14 所示。

图 5.14 例 5.7 的调试运行结果

在例 5.7 程序中,MP3Player 和 VideoPlayer 类中都有一个成员函数 Play(),这样在派生类 MP4Player 中就会同时拥有两个名为 Play 的成员函数。在 main()函数中,如果把注释行的"//"去掉,访问 MP4Player 类的成员函数 Play()时,会因为对该成员函数访问的不明确而出现二义性错误。为解决上述成员访问冲突问题,在 main ()函数中,采用了成员名限定的方法对具体基类的同名成员进行访问,格式如下:

基类名::成员名

即在成员名 Play()前显式指定该成员所属基类,这样就有效避免了对该成员访问的二义性错误,同样对于基类的数据成员,该原则同样适用。

对于多继承,如果在不同的基类中定义了同名的成员,在派生类中要区分成员的来源,就必须使用成员名限定方法,即在成员名前加上各自基类的访问域限制即可。

【例 5.8】 多继承中成员限定法。

```
/* 05_08.cpp */
#include <iostream>
using namespace std;
```

```
class B1                                    //基类 B1
{
public:
    int m;                                  //成员变量 m
    B1()                                    //构造函数
    {  m = 0;  }
};
class B2                                    //基类 B2
{
public:
    int m;                                  //成员变量 m
    B2()                                    //构造函数
    {  m = 100;  }
};
class D:public B1,public B2                 //派生类 D
{
public:
    void Test()
    {
        //cout << m << endl;                //此语句将引起二义性错误
        cout <<"B1::m = "<< B1::m << endl;  //输出基类 B1 中的成员 m 的值
        cout <<"B2::m = "<< B2::m << endl;  //输出基类 B2 中的成员 m 的值
    }
};
int main()                                  //主函数
{
    D d1;                                   //派生类对象
    d1.Test();
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.15 所示。

```
B1::m=0
B2::m=100
请按任意键继续. . .
```

图 5.15　例 5.8 的调试运行结果

说明：程序中分别定义了两个基类 B1、B2，在两个类中都定义了数据成员 m，在派生类 D 中如果直接引用成员 m，就会引发二义性错误，编译器无法确定是对哪个基类中的 m 进行访问，因此必须显式地给出基类的访问域限制，如 B1::m，B2::m。

5.6.2　名字覆盖

当派生类中定义了与基类中同名的成员时，则从基类中继承得到的成员被派生类的同名成员覆盖，派生类对基类成员的直接访问将被派生类中该成员取代，为访问基类成员，必须采用成员名限定方法。

【例 5.9】　成员访问冲突。

```
/* 05_09.cpp */
#include <iostream>
using namespace std;
class Circle                                //定义圆类
```

```
{
protected:
    float radius;                                   //半径
public:
    Circle (float r)                                //构造函数
    {   radius = r;  }
    float Area()                                    //求圆面积
    {  return 3.14f * radius * radius;  }
};
class Cylinder: public Circle                       //圆柱体派生类
{
private:
    float height;                                   //高度
public:
    Cylinder (float r,float h) :Circle(r)           //构造函数
    {   height = h;  }
    float Area()                                    //求圆柱体面积,覆盖了基类的 Area()函数
    {   float botarea = 0,sidearea = 0;
        botarea = Circle::Area() * 2;               //底面积 * 2
        sidearea = 2 * 3.14f * radius * height;     //侧面积
        return botarea + sidearea;
    }
    float Volume()                                  //圆柱体积
    {   return Circle::Area() * height;  }          //基类求面积乘高度
};
int main()
{
    Cylinder cy1(10,5);                             //定义圆柱体对象
    cout <<"BottomArea = "<< cy1.Circle::Area()<< endl;  //访问基类成员
    cout <<"Area = "<< cy1.Area()<< endl;           //访问派生类成员
    cout <<"Volume = "<< cy1.Volume()<< endl;
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.16 所示。

从上述程序可以看出,用派生类对象访问与基类中同名的成员时,会调用本类中的成员,而不会访问基类成员。为访问基类的同名成员,需要以成员名限定的方法来指明,如在 Cylinder 类中的求面积和体积函数就用到了基类的 Circle::Area()函数。

另外,要注意的一点是,在派生类中如果定义了与基类中同名的函数,则基类中所有的同名的重载函数都将被覆盖,即在派生类中或通过派生类对象都无法直接访问基类的任何一个同名函数,如图 5.17 中定义了一个基类和派生类。

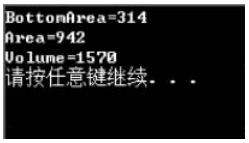

图 5.16 例 5.9 的调试运行结果

图 5.17 派生类对基类的名字覆盖

有如下程序段落：

```
Derived D1;                    //定义派生类对象
D1.Fun();                      //调用派生类中的 Fun()函数
D1.Fun(5);                     //该语句错误,基类中的函数名被覆盖,无法直接调用
D1.Base::Fun();                //调用派生类中的 void Fun()函数
D1.Base::Fun(5);               //调用派生类中的 void Fun(int a)函数
```

5.6.3 赋值兼容规则

在派生类中包含了基类的全部成员，因此可以认为派生类对象在一定程度上就是基类对象，可以使用基类的任何地方也可以使用派生类，而实现了兼容的效果。为实现这种兼容性，应该允许把派生类对象形态对应赋值给基类对应的对象形态，此处对象形态包括普通对象、对象指针和对象引用。例如，可以把派生类对象赋值给基类对象。

由于在继承之后，派生类对象中包含有比基类更多的成员，因此可以把派生类对象看作基类对象。反之则不允许，这是因为基类对象中不包含派生类新增加的成员。

因此，在派生类对象和基类对象之间赋值时需要注意赋值的方向，即这些赋值操作需要满足赋值兼容规则。赋值兼容规则包括：

(1) 基类对象可以赋值给基类对象，也可以把派生类对象赋值给基类对象。

(2) 基类指针可以指向基类对象，也可以指向派生类对象。

(3) 基类引用可以指向基类对象，也可以指向派生类对象。

例如，有基类 Base 和其派生类 Derived，可以定义相应的对象、指针：

```
Base b1;
Base * pb;
Derived d1;
```

根据赋值兼容规则，在基类 Base 对象可以出现的任何地方都可以用派生类 Derived 对象来替代。

(1) 派生类对象可以赋值给基类对象，即派生类对象中来自基类成员，逐个赋值给基类对象的成员：

```
b1 = d1;
```

(2) 派生类的对象也可以初始化基类对象的引用：

```
Base &rb = d1;
```

(3) 基类的指针赋值为派生类对象的地址：

```
pb = &d1;
```

由于赋值兼容规则的引入，对于基类及其公有派生类的对象可以使用相同的函数统一进行处理(如当函数形参为基类对象形态时，实参可以是派生类对应的对象形态)而没有必要为每一个类设计单独的模块，大大提高了程序的效率。为 C++的运行时多态性打下了重要基础，相关内容将在第 6 章中详细介绍。

【例 5.10】 赋值兼容示例。

```
/* 05_10.cpp */
#include<iostream>
using namespace std;
class Base                          //基类 Base
{
protected:
    int member;
public:
    Base()
    {  member=0;  }
    void Show()                     //公有成员函数
    {  cout<<"Base::Show() :"<<member<<endl;  }
};
class Derived1:public Base          //第 1 个派生类 Derived1
{
public:
    Derived1(int a)
    {  member=a;  }
    void Show()                     //重写公有成员函数 Show
    {  cout<<"Derived1::Show() :"<<member<<endl;  }
};
class Derived2:public Derived1      //第 2 个派生类 Derived2
{
public:
    Derived2(int a): Derived1(a)
    { }
    void Show()                     //重写公有成员函数 Show
    {  cout<<"Derived2::Show() :"<<member<<endl;  }
};
void Test(Base *pb)                 //测试函数,用基类指针作参数
{  pb->Show();  }
void Test(Base &br)                 //重载测试函数,用基类引用作参数
{  br.Show();  }
int main()                          //主函数
{
    Base b0;                        //基类 Base 对象
    Derived1 d1(5);                 //派生类 Derived1 的对象
    Derived2 d2(10);                //派生类 Derived2 的对象
    Base *pb0;                      //基类指针 pb0
    pb0=&b0;                        //基类指针 pb0 指向基类对象 b0
    Test(pb0);
    b0=d1;                          //基类对象赋值为子类对象
    Test(pb0);                      //测试输出
    pb0=&d1;                        //基类指针 pb0 指向其第一派生类 Derived1 的对象 d1
    Test(pb0);
    Test(d2);                       //第 2 派生类 Derived2 的对象 d2 的引用作参数传给 Test 函数
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.18 所示。

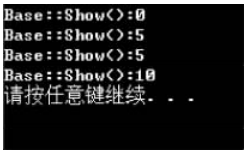

图 5.18　例 5.10 的调试运行结果

说明：在程序中分别测试了用基类指针指向基类对象、把基类对象赋值为派生类对象、基类指针指向派生类对象、把派生类对象赋值初始化为基类引用 4 种情况，验证了继承机制中的赋值兼容性。

从上述结果也应该看到，由于指针 pb0 是基类对象指针，在用该指针指向派生类对象时，只能访问到这些派生类对象中包含的基类成员函数 Show()，而不是派生类中重写的 Show()函数，因此输出结果都是相同的。如何通过基类指针访问派生类成员，就是第 6 章要讨论的内容。

5.7　虚基类

5.7.1　提出问题

在多继承关系中，如果某个派生类 D 的多个基类(如类 B1 和 B2)派生自另一个公共基类 B0，则在派生类对象中，会通过不同的继承路径多次得到基类 B0 的成员。通过派生类 D 的对象访问这些成员时，会出现对这些成员的访问冲突，即二义性问题。为解决冲突问题可以使用成员名限定的方法来唯一标识某个成员所属的基类，但是这不能从根本上解决问题，即派生类对象中存在基类成员的多个副本，如图 5.19 所示。

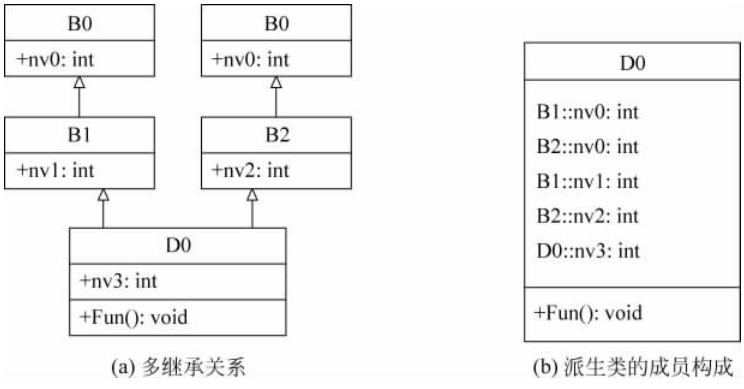

图 5.19　多继承关系及派生类的成员构成 UML 图

在如图 5.22 所示的继承关系中，类 B0 是一个公共基类，派生类 D0 是多重继承的派生类，它有两个基类 B1 和 B2。从这种继承关系中可以看出，在派生类 D0 中含有两个从不同路径得到的间接基类 B0 的成员 nv0。在引用这些同名成员时必须在派生类对象名后增加直接基类名进行类名限定，以避免产生二义性，那么有没有更好的解决办法呢？

5.7.2　虚基类的概念

为使得公共基类 B0 在派生类 D0 中只产生一份基类成员，则需要将这个共同基类 B0 设置为虚基类，让基类 B1 和 B2 从基类 B0 虚拟继承，这时从不同的路径继承过来的同名数

据成员在派生类中就只有一个副本,同一个函数名也只有一个映射。这样就解决了同名成员的唯一标识问题。

使用虚基类,可以使公共基类的成员在其间接派生类中只保留一份。使用虚基类后,4个类之间的关系如图 5.20(a)所示,这时派生类中的成员如图 5.20(b)所示。

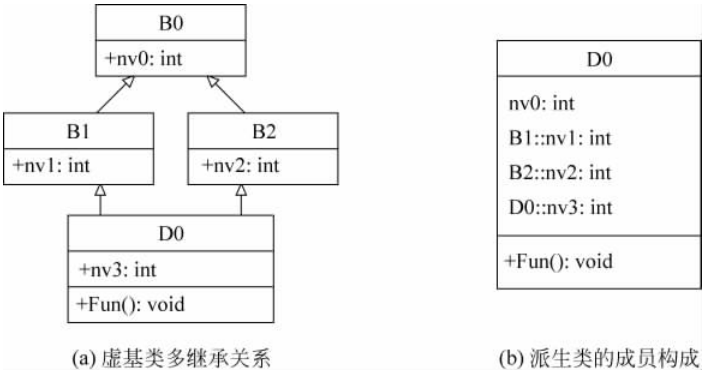

图 5.20 虚基类多继承关系及派生类的成员构成 UML 图

定义虚基类的格式如下:

```
class 派生类名: virtual 继承方式  基类名称
{
    …
};
```

定义虚基类使用关键字 virtual,在说明派生类时加在基类名的继承方式前边,经过虚基类说明后,当公共基类经过多条派生路径被一个派生类继承时,该派生类中只保留一次公共基类的成员。则在图 5.20 中的几个类可以写成如下形式:

```
class B0                          //公共基类
{
    public:
    int nv0;
};
class B1: virtual public B0       //基类 B1,虚拟继承自 B0
{
    public:
    int nv1;
};
class B2: virtual public B0       //基类 B2,虚拟继承自 B0
{
    public:
    int nv2;
};
class D0: public B1,public B2     //派生类 D0
{
    public:
    int nv3;
};
```

在以上程序中，B0 类就被定义为虚基类，D0 类中就只含有 B0 类中的一次成员，这样就避免了二义性问题。

5.7.3 虚基类的初始化

关于虚基类的初始化，有如下两条规则：

(1) 所有从虚基类直接或者间接派生的类必须在该类构造函数的成员初始化列表列出对虚基类构造函数的调用，但是只有实际构造对象的类的构造函数才会引发对虚基类构造函数的调用，而其他基类在成员初始化列表中对虚基类构造函数的调用都会被忽略，从而保证了派生类对象中虚基类成员只会被初始化一次。

(2) 若某类构造函数的成员初始化列表中同时列出对虚基类构造函数和非虚基类构造函数的调用，则会优先执行虚基类的构造函数。

正如上面所示，在图 5.23 具有虚基类的多层派生继承结构中，对其虚基类 B0 的初始化应该由最后派生类 D0 来完成。如果由它的直接派生类完成，即由类 B1 和类 B2 中构造函数进行初始化，这时有可能使得类 B0 中一个数据成员先后接收不同的初始化参数而产生矛盾。如果由派生类来完成便可避免这一矛盾。为此要在定义派生类 D0 的构造函数中增加一个给虚基类的初始化项，而给两个直接基类 B1 和 B2 的初始化仍然保留。

【例 5.11】 设置虚基类以解决二义性。

```
/* 05_11.cpp */
#include <iostream>
using namespace std;
class Base                                  //虚基类 Base
{
public:
    Base(int a)                             //构造函数
    {  val = a;  }
    void Print()                            //输出成员 val 值的函数
    {  cout << val << endl;  }
protected:
    int val;                                //成员变量 val
};
class Derived1:virtual public Base          //第一个派生类 Derived1
{
public:
    Derived1(int x,int y):Base(x),dv1(y)    //调用了基类的构造函数
    {  }
protected:
    int dv1;
};
class Derived2:virtual public Base
{
public:
    Derived2(int x,int y):Base(x),dv2(y)    //调用了基类的构造函数
```

```
    {  }
protected:
    int dv2;
};
//最终的多重继承类 DerivedFinal
class DerivedFinal:public Derived1,public Derived2
{
public:
    DerivedFinal(int x,int y,int z):
        Derived1(x,y),Derived2(y,z),Base(z)
    {  }
};
int main()
{
    DerivedFinal df(7,18,22);              //定义对象
    df.Print();                            //输出虚基类 Base 的成员 val 的值
    system("pause");
    return 0;
}
```

程序的调试运行结果如图 5.21 所示。

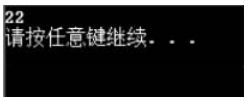

图 5.21 例 5.11 的调试运行结果

在上述程序中,从输出结果可以看出,在构造 DerivedFinal 类的对象 df 时,只有派生类 DerivedFinal 的构造函数的初始化表中列出的虚基类 Base 的构造函数被调用了,而且仅被调用了一次,在 Derived1 类和 Derived1 类的构造函数中对于基类 Base 构造函数的调用全部被忽略了。

5.8 本章小结

本章讨论了继承机制中类的继承方式、派生类对象的构造与析构的顺序与原则、继承中的几个主要问题,以及虚基类的应用。

1. 类的继承方式

类的继承方式有 public(公有继承)、protected (保护继承)和 private (私有继承)3 种,不同的继承方式,导致原来具有不同访问属性的基类成员在派生类中的访间属性也有所不同。这时访问规则有两类:一是派生类中非 static 成员函数和友元函数对基类成员的访问;二是在派生类作用域外的对象对基类成员的访问。

2. 派生类对象的构造与析构的顺序

构造派生类的对象时,就要对基类数据成员、派生类自身的数据成员和对象成员进行初始化。由于基类的构造函数不能被继承下来,要完成这些工作,就必须给派生类添加新的构造函数。派生类构造函数执行的一般顺序如下:

(1) 调用基类构造函数,调用顺序按照它们被继承时声明的顺序(从左向右)。

（2）调用对象成员的构造函数，调用顺序按照它们在类中声明的顺序。

（3）执行派生类的构造函数体中的内容。

派生类析构函数的功能是在该类对象消亡之前进行一些必要的清理工作。析构函数没有类型也没有参数，与构造函数相比情况略简单些。析构函数的执行次序和构造函数正好严格相反，首先调用派生类的析构函数，然后析构派生类的对象成员，最后调用基类的析构函数。

3. 继承中基类成员的访问和赋值兼容性

当多个基类中定义有同名成员，则派生类对这些同名成员的访问可能存在冲突和二义性，这时可采用成员名限定法来访问这种不明确的问题。

在多重继承的情况下，调用不同基类中的相同成员时可能也会出现二义性问题。C++语言规定，在派生类中重新声明的成员函数具有比基类同名成员函数更小的作用域，这时在可能出现二义性的地方，加上类名限定，就可避免出现名字冲突问题。

赋值兼容规则是指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。通过公有继承，派生类得到了基类中除构造函数、析构函数之外的所有成员。这样，公有派生类实际就具备了基类的所有功能，凡是基类能解决的问题公有派生类都可以解决。赋值兼容规则中所指的替代包括以下几种情况：

（1）派生类的对象可以赋值给基类的对象；

（2）派生类的对象可以初始化基类的引用；

（3）派生类对象的地址可以赋给指向基类的指针。

4. 虚基类

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，从上一级共同基类继承来的成员就拥有相同的名称。在派生类对象中这些同名数据成员在内存中同时拥有多个副本，同一个函数名会有多个映射。这时可以将共同基类设置为虚基类，那么从不同的路径继承过来的同名数据成员在内存中就只有一个副本，同一个函数名也只有一个映射。因而虚基类解决了同名成员的唯一标识问题。

习题

5-1　判断题下列描述的正确性，对的画（√）错的画（×）。

（1）C++语言中，包括单继承和多继承两类。（　　）

（2）C++语言的继承仅包括 public 和 private 两种方式。（　　）

（3）派生类从基类派生出来，它不能再生成新的派生类。（　　）

（4）在公有继承中，基类中的私有成员在派生类中都是可见的。（　　）

（5）在私有继承中，基类中只有公有成员对派生类是可见的。（　　）

（6）派生类仅是其基类的组合。（　　）

（7）基类的构造也可以被继承。（　　）

（8）基类的析构函数不能被继承。（　　）

(9) 若多继承情况下出现名字冲突,可以使用成员名限定法访问基类成员。 ( )

5-2 单项选择题。

(1) 派生类的构造函数的成员初始化列表中,不能包含( )。

A. 基类的构造函数 B. 派生类子对象的构造函数

C. 基类子对象的构造函数 D. 派生类中一般数据的初始化

(2) 派生类的对象可以访问它的( )基类成员。

A. 公有继承的公有成员 B. 公有继承的私有成员

C. 公有继承的保护成员 D. 私有继承的公有成员

(3) 若基类定义了两个重载函数 Fun()和 Fun(int a),则如下说法正确的是( )。

A. 在派生类中不可以再定义名为 Fun 的函数

B. 在派生类中只能定义和基类原型不同的 Fun 重载函数

C. 在派生类中只要定义了名为 Fun 的函数,基类的所有名为 Fun 重载函数都被覆盖

D. 在派生类中定义的 Fun 函数仅覆盖基类中相同原型的 Fun 函数

(4) 下面关于赋值兼容性错误的语句是( )。

A. 基类对象可以赋值给基类对象,也可以把派生类对象赋值给基类对象

B. 基类指针可以指向基类对象,也可以指向派生类对象

C. 基类引用可以指向基类对象,也可以指向派生类对象

D. 基类对象的地址也可以直接赋值给派生类指针变量

(5) 设置虚基类的目的是( )。

A. 消除二义性 B. 方便书写程序

C. 提高运行效率 D. 减小目标代码体积

(6) 关于保护成员的说法正确的是( )。

A. 在派生类中仍然是保护的

B. 具有私有成员和公有成员的双重特色

C. 在派生类中是私有的

D. 在派生类中是公有的

5-3 简述类的组合和继承的区别。

5-4 派生类能否直接访问基类的私有成员? 如果不能,用什么方式实现?

5-5 派生类的构造函数与析构函数的执行顺序是怎样的?

5-6 在类的继承机制中引入虚基类概念的原因是什么?

5-7 考虑赋值兼容性对程序中对象的组织和管理带来的影响。

5-8 程序改错。

(1)

```
class BC
{private:
    int x;
 public:
    BC(){x = 0;}
};
```

```
class DC
{ public:
    DC(){x = 100;}
};
```

(2)

```
class BC
{private:
    int x;
 public:
    BC(){ x = 0;}
};
class DC: public BC
{ public:
    DC(int i){ x = i;}
};
```

(3)

```
class BC
{private:
    int x;
public:
    BC(int i){x = i;}
};
class DC: public BC
{public:
    DC(){y = 0;}
 private:
    int y;
};
```

5-9 写出程序结果。

(1)

```
#include <iostream>
using namespace std;
class BC
{ public:
    void Fun(){ cout <<"base class"<< endl;}
};
class DC:public BC
{ public:
    void Fun(){ cout <<"vderived class"<< endl;}
};
int main()
{   BC * pbc;
    DC dc;
    pbc = &dc;
    pbc -> Fun();
    return 0;
}
```

(2)

```
#include <iostream>
using namespace std;
class Base
{
public:
    Base(int i,int j){ x = i; y = j;}
    void Offset(int a,int b){ x += a;y += b; }
    void Print() { cout <<"("<< x <<","<< y <<")"<< endl; }
protected:
    int x,y;
};
class Derived: public Base
{
public:
    Derived(int i,int j,int k,int l):Base(i,j)
    {  w = k; h = l;  }
    void Move() {  Offset(10,10);  }
    void Print()
    {  cout <<"("<< x <<","<< y <<" - "<< w <<","<< h <<")"<< endl;  }
    void Show()
    {  Base::Print();  }
private:
    int w,h;
};
int main()
{
    Base b1(10,5);
    b1.Print();
    Derived d1(3,9,18,33);
    d1.Print();
    d1.Move();
    d1.Base::Print();
    d1.Derived::Print();
    d1.Show();
    return 0;
}
```

5-10　编程题,定义一个形状 Shape 基类,包括整型数的成员变量 x、y 来表示位置,定义带参数的构造函数可以初始化成员变量 x、y,再由此定义出派生类:矩形类 Rect 和圆类 Circle,Rect 类增加宽和高 w、h 两个成员变量,Circle 类增加半径 r,分别定义两个派生类的构造函数,可以初始化各自的成员变量(包括基类成员变量),最后用主函数测试。

5-11　编程题,定义机动车类 Vehicle,包括的数据成员有出厂日期和售价,并定义成员函数可以设置这些数据成员,再定义 Print( )成员函数输出成员变量内容;然后定义 Car 类和 Truck 类,分别扩展各自的内容,如 Car 类增加乘客数量,Truck 类增加载重吨数,并都可以通过构造函数初始化各自成员变量和其基类成员,最后都能输出相关的信息。