

第四章 继承和派生

第二章和第三章的内容主要围绕在面向对象编程的“抽象性”（如何将问题空间的事物抽象成类，这是个建模的过程）和“封装性”（保证数据的私密性，只提供外部访问的接口），本章来探讨“继承性”。继承的概念不难理解，多少有点“**不劳而获**”的意思，实际也是如此，面向对象程序设计的一个重要特点就是可以在既有类的基础上定义新的类，而不用将既有类的内容重新书写一遍，这称为“继承”（inheritance），**既有类称为“父类”或“基类”**，在它的基础上建立的类称为“派生类”、“导出类”和“子类”，在本章节的描述中，统一使用“**基类**”和“**派生类**”的概念。

4.1 什么是继承

- 继承是很自然的概念，广泛存在于现实世界中，如家族图谱，动植物分类图等。
- 对面向对象的程序设计(OOP)而言，继承的引入意义巨大：首先，程序员可以按现实世界、按自然的方式去思考和解决问题，组织信息，提高了效率，其次，可以**复用**基类的代码，并可以在继承类中**增加**新代码或者**覆盖**基类的成员函数，为基类成员函数赋予新的意义，实现最大限度的代码复用。

4.1.1 简单示例

通过例子来看什么是继承，如下边代码。

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

class point
{
private:
    int xPos;
    int yPos;
public:
    point(int x = 0, int y = 0)           //构造函数，带缺省参数
    {
        xPos = x;
        yPos = y;
    }
    void disp()                          //成员disp函数，用来输出点的信息
    {
        cout << "( " << xPos << " , " << yPos << " )" << endl;
    }
    int GetX()                           //读取private成员xPos
    {
        return xPos;
    }
    int GetY()                           //读取private成员yPos
    {
        return yPos;
    }
};

class point3D:public point                //3维点类point3D，从point类继承而来
{
private:
    int zPos;                            //在point类基础上增加了zPos坐标信息
public:
    point3D(int x, int y, int z):point(x, y)//派生类构造函数，初始化表中调用基类构造函数
    {
        zPos = z;
    }
    void disp()                          //隐藏了基类中的同名disp函数
    {
        cout << "( " << GetX() << " , " << GetY() << " , " << zPos << " )" << endl;
    }
    int calcSum()                        //增添了计算3个数据成员和的函数
    {
        return GetX() + GetY() + zPos;
    }
};
```

```

int main()
{
    point pt1(7, 8);           //建立point类对象pt1
    pt1.disp();               //显示pt1的信息

    point3D pt2(3, 4, 5);     //建立point3D类对象pt2
    pt2.disp();              //显示pt2的信息

    int res = pt2.calcSum();   //计算pt2中3个坐标信息的和
    cout << res << endl;     //输出结果

    return 0;
}

```

执行结果:

```

< 7 , 8 >
< 3 , 4 , 5 >
12
Press any key to continue

```

上边的代码中, point 类是二维点类, 三维点类 point3D 是从 point 类继承而来的, 则 point 类称为“基类”、point3D 类称为“派生类”。在 point3D 类内不用再对 xPos 和 yPos 进行定义性声明, 只要增加一个 private 成员 zPos 即可, **还可在 point3D 类内定义与 point 类某个成员函数同名的函数以实现功能覆盖**, 如 point3D 中的 disp 函数实现了 point 类中 disp 函数不同的功能。根据需要可再 point3D 类增加其他一些成员函数和数据成员, 如 calcSum 函数。

4.1.2 继承的层次性

任何一个类都可以派生出新类, 派生类还可以再派生出新的类, 因此, 基类和派生类是相对而言的。一个基类可以是另一个基类的派生类, 这样便构建了层次性的类结构, 如图所示, 类 B 是 A 的派生类, 同时又派生了新类 C, B 又可以看作是 C 的基类。

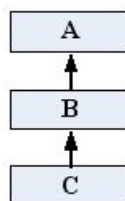


图 10.1 继承的层次性

4.2 派生方式

派生有 public、protected、private 三种方式，不同的派生方式下，派生类对基类成员的访问权限以及外部对基类成员的访问权限有所不同，本节详细讨论不同的派生方式。

4.2.1 public 派生与 private 派生

C++中，利用基类派生其子类（派生类）的基本格式为：

```
class 派生类名: 派生方式 基类名
{
    private:
        新增私有成员列表;
    public:
        新增公开成员列表;
};
```

通过继承，派生类自动得到了除基类私有成员以外的其它所有数据成员和成员函数，在派生类中可以直接访问，从而实现了代码的复用。派生方式是指 public、protected 和 private 派生，3 种派生方式的比较见下表。

3种派生方式下的访问权限

基类成员	private	protected	public	private	protected	public	private	protected	public
派生方式	private			protected			public		
派生类中	不可见	private	private	不可见	private	protected	不可见	protected	public
外部	不可见	不可见	不可见	不可见	不可见	不可见	不可见	不可见	可见

4.2.2 派生类访问权限

private 成员是私有成员，只能被本类的成员函数所访问，派生类和类外都不能访问。

public 成员是公有成员，在本类、派生类和外部都可访问

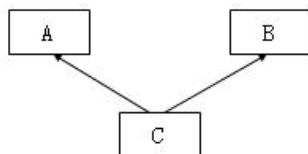
protected 成员是保护成员，只能在本类和派生类中访问。是一种区分血缘关系内外有别的成员。

派生类的访问权限规则如下：

- 基类的 `private` 成员在外部和派生类中均不可访问
- `private` 派生使得基类中的所有成员都变成 `private` 成员，在外部和其派生类中都不可访问
- `protected` 派生使得基类中的所有非 `private` 成员都降一级：`public` 降为 `protected`，`protected` 降为 `private`
- `public` 派生时，基类的所有成员在派生类中的访问属性不变

4.3 多基派生

派生类只有一个基类时，称为单基派生，在实际运用中，我们经常需要派生类同时具有多个基类，这种方法称为多基派生或多重继承，下面是双基继承的示意，在实际应用中，还允许使用三基甚至是更多基继承。



4.3.1 多基派生的声明和定义

在 C++ 中，声明和定义具有两个以上基类的派生类与声明单基派生类的形式类似，只需将要继承的多个基类用逗号分开即可，如

派生类名(参数表):基类名 1(参数表 1), 基类名 2(参数表 2), ..., 基类名 n(参数表 n)

```
{  
private:  
    新增私有成员列表;  
public:  
    新增公开成员列表;  
};  
例程:
```

```

#include <iostream>
using namespace std;

class A          //类A的定义
{
public:
    void print()  //A中定义了print函数
    {
        cout << "Hello,this is A" << endl;
    }
};
class B          //类B的定义
{
public:
    void show()   //B中定义了show函数
    {
        cout << "Hello,this is B" << endl;
    }
};
class C : public A, public B  //类C由类A和类B共同派生而来
{
public:
    void disp()
    {
        show();    //在类内部基类的成员函数
    }
};
int main()
{
    C c;
    c.print();      //在类外部 通过 派生类对象 访问 基类 成员
    c.disp();       //在类外部 访问 派生类对象 的新添加的成员
    return 0;
}

```

执行结果:

```

Hello,this is A
Hello,this is B
Press any key to continue

```

4.3.2 多基派生的二义性

一般来说，在派生类中对基类成员的访问应当具有唯一性，但在多基继承时，如果多个基类中存在同名成员的情况，造成编译器无从判断具体要访问的哪个基类中的

成员，则称为对基类成员访问的二义性问题。

例程：

```
//多基类继承时的二义性问题
#include <iostream>
using namespace std;

class A          //类A的定义
{
public:
    void print()  //A中定义了print函数
    {
        cout<<"Hello,this is A"<<endl;
    }
};

class B          //类B的定义
{
public:
    void print()  //B中同样定义了print函数
    {
        cout<<"Hello,this is B"<<endl;
    }
};

class C : public A, public B  //类C由类A和类B共同派生而来
{
public:
    void disp()
    {
        print();  //编译器无法决定采用A类中定义的版本还是B类中的版本
    }
};

int main()
{
    C c;
    c.disp();      //派生类内访问对象成员的二义性
    c.print();     //外部通过派生类对象访问基类成员的二义性
    return 0;
}
```

以上代码会报错

4.3.3 多基派生的二义性解决方案

若两个基类中具有同名的数据成员或成员函数，应使用成员名限定来消除二义性，如：

```
void disp()
{
    A::print();    //加成员名限定 A::
}
```



```

//多基类继承时的二义性问题的解决方案，见代码备注中的1和2
#include <iostream>
using namespace std;

class A          //类A的定义
{
public:
    void print()  //A中定义了print函数
    {
        cout<<"Hello,this is A"<<endl;
    }
};

class B          //类B的定义
{
public:
    void print()  //B中同样定义了print函数
    {
        cout<<"Hello,this is B"<<endl;
    }
};

class C : public A, public B    //类C由类A和类B共同派生而来
{
public:
    void disp()
    {
        A::print();           //1. 指明采用A类中定义的版本
    }

    //重载print函数，根据条件调用不同基类的print函数
    void print()
    {
        if (true)
            A::print();
        else
            B::print();
    }
};

int main()
{
    C c;
    c.disp();                //派生类内访问对象成员的二义性得到解决

    c.A::print();           //2.1 指明采用B类中定义的版本 c.(A::print());
    c.print();              //2.2 在C类中重载print函数

    return 0;
}

```

运行结果：


```

Hello,this is A
Hello,this is A
Hello,this is A
Press any key to continue

```

上述代码明确指明要在 disp 函数内要调用的是从类 A 继承来的 print 函数，但即使做了如此修改，仍无法编译通过，问题出在语句“c.print();”上，虽然也可以通过添加作用域限定符，诸如“c.A::print();”来解决，但最好在类 C 中也定义一个同名 print 函数，根据需要调用 A::print() 还是 B::print()，从而实现对基类同名函数的隐藏。

4.4 共同基类(虚基类)

多基派生中，如果在多条继承路径上有一个共同的基类，如图 10.4 所示，不难看出，在 D 类对象中，会有来自两条不同路径的共同基类（类 A）的双重拷贝。

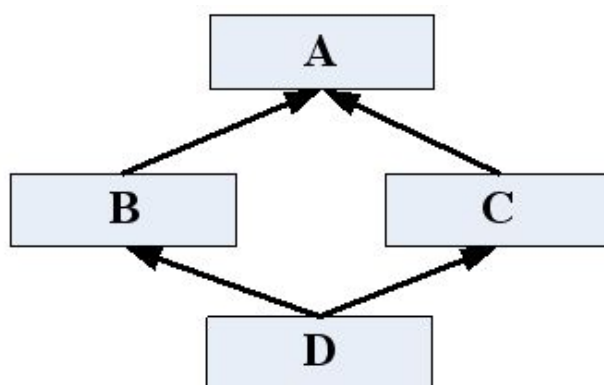
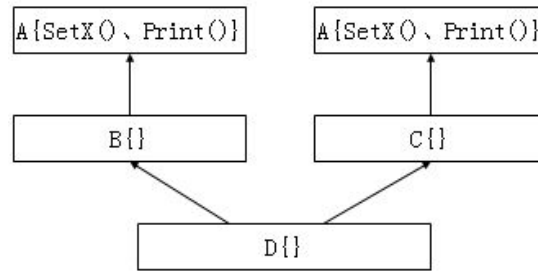


图 10.4 多基继承时的共同基类问题

4.4.1 共同基类的二义性

共同基类和多基派生的共同作用，使得在派生类中会出现多个共同基类的拷贝，这很容易带来二义性问题

如下图所示，D 类通过 B 类和 C 类各继承了一次 A 类的函数 SetX() 和 Print()，从而产生了二份拷贝，进而导致了二义性。



例程：

```

//共同基类带来的二义性
#include <iostream>
using namespace std;

class A                                //公共基类
{
protected:                            //protected成员列表
    int x;
public:                                 //public成员列表
    A(int xp = 0)                       //构造函数
    {
        x = xp;
    }
    void SetX(int xp)                   //设置protected成员x的值
    {
        x = xp;
    }
    void print()
    {
        cout << "this is x in A: " << x << endl;
    }
};

class B: public A                       //类B由类A派生而来
{
};

class C: public A                       //类C由类A派生而来
{
};

class D : public B, public C            //类D由类B和类C派生而来
{
};

int main()
{
    D d;                               //声明一个D类对象exD
    d.SetX(5);                          //SetX()具有二义性，系统不知道是调用B类的还是C类的SetX()函数
    d.print();                          //print()具有二义性，系统不知道是调用B类的还是C类的print()函数
    return 0;
}

```

4.4.2 共同基类的二义性的解决方案

使用关键字 `virtual` 将共同基类 A 声明为虚基类，可有效解决上述问题。

在定义由共同基类直接派生的类（中的类B和类C）时，使用下列格式定义：
class 派生类名 : virtual 派生方式 基类名

```
{
    //类定义
};
```

虚函数：由 virtual 声明，它允许在派生类中被重写。

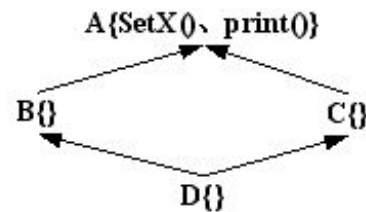


图 10.6 虚基派生 DAG 示意图

例程：

```
//使用虚基类解决共同基类带来的二义性问题
#include <iostream>
using namespace std;

class A          //公共虚基类A
{
protected:      //protected成员列表
    int x;
public:
    A(int xp = 0)    //构造函数，带缺省构造参数
    {
        x=xp;
    }
    void SetX(int xp)//SetX函数用以设置protected成员x
    {
        x = xp;
    }
    void print()     //print函数输出信息
    {
        cout << "this is x in A: " << x << endl;
    }
};

class B: virtual public A    //B由A虚基派生而来
{
};

class C: virtual public A    //C由A虚基派生而来
{
};

class D: public B, public C    //D由B和C共同派生
{
};
```

```

int main()
{
    D d;           //声明一个D类对象exD

    d.SetX(5);      //SetX函数，因为virtual派生，在D中只有一个版本，不会二义
    d.print();      //print函数，因为virtual派生，在D中只有一个版本，不会二义

    d.A::print();   //还可类名显式说明调用函数的版本
    d.B::print();
    d.C::print();

    return 0;
}

```

执行结果：

```

this is x in A: 5
this is x in A: 5
this is x in A: 5
this is x in A: 5
Press any key to continue

```

4.4.3 小结：共同基类派生二义性与多基派生二义性不同

尽管看起来很相似，但（共同基类）虚基派生和多基派生带来的二义性有些细微的差别：

- （1）多基派生的二义性主要是成员名的二义性，通过加作用域限定符来解决。
- （2）虚基派生（共同基类）的二义性则是共同基类成员的多重拷贝带来的存储二义性，使用 virtual 派生来解决。

4.5 单基派生类的构造函数和析构函数

派生时，构造函数和析构函数是不能继承的，为了对基类成员进行初始化，必须对派生类重新定义构造函数和析构函数，并在构造函数的初始化列表中调用基类的构造函数。

由于派生类对象通过继承而包含了基类数据成员，因此，创建派生类对象时，系统首先通过派生类的构造函数来调用基类的构造函数，完成基类成员的初始化，而后对派生类中新增的成员进行初始化。

4.5.1 单基派生类的构造函数

派生类构造函数的一般格式为：

派生类名(总参数表)：基类构造函数(参数表)

```
{  
    //函数体  
};
```

必须将基类的构造函数放在派生类的初始化表达式中，以调用基类构造函数完成基类数据成员的初始化，派生类构造函数实现的功能，或者说调用顺序为：

- (1) 完成对象所占整块内存的开辟，由系统在调用构造函数时自动完成。
- (2) 调用基类的构造函数完成基类成员的初始化。
- (3) 若派生类中含对象成员、const 成员或引用成员，则必须在初始化表中完成其初始化。
- (4) 派生类构造函数体执行。

例子：

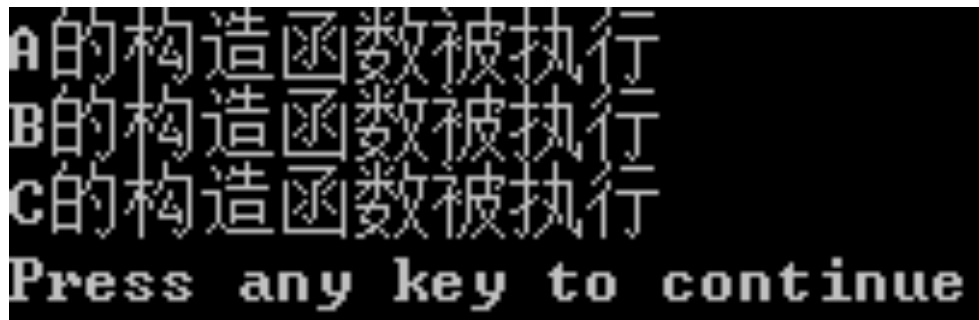
```
//派生类构造函数的调用顺序  
#include <iostream>  
using namespace std;  
  
class A //类A的定义  
{  
private: //private成员列表  
    int x;  
public:  
    A(int xp=0) //构造函数，带缺省参数  
    {  
        x = xp;  
        cout << "A的构造函数被执行" << endl;  
    }  
};  
class B //类B定义  
{  
public:  
    B() //无参构造函数  
    {  
        cout << "B的构造函数被执行" << endl;  
    }  
};
```

```

class C:public A           //类C由类A派生而来
{
private:
    int y;
    B b;
public:
    C(int xp, int yp) : A(xp), b() //构造函数, 基类构造函数在初始化表中调用
    {
        y = yp;
        cout << "C的构造函数被执行" << endl;
    }
};
int main()
{
    C c(1, 2);           //创建C类对象c
    return 0;
}

```

结果:



```

A的构造函数被执行
B的构造函数被执行
C的构造函数被执行
Press any key to continue

```

4.5 .2 单基派生类的析构函数

当对象被删除时, 派生类的析构函数被执行。**析构函数同样不能继承**, 因此, 在执行派生类析构函数时, 基类析构函数会被自动调用。

执行顺序是先执行派生类的析构函数, 再执行基类的析构函数, 这和执行构造函数时的顺序正好相反。

例程:


```

//派生类的析构函数
#include <iostream>
using namespace std;

class A //A类的定义
{
private: //private成员列表
    int x;
public: //public成员列表
    A(int xp = 0) //构造函数，带缺省参数
    {
        x = xp;
        cout << "A的构造函数被执行" << endl;
    }
    ~A() //析构函数
    {
        cout << "A的析构函数被执行" << endl;
    }
};

class B //B类的定义
{
public: //public成员列表
    B() //无参构造函数
    {
        cout << "B的构造函数被执行" << endl;
    }
    ~B() //析构函数
    {
        cout << "B的析构函数被执行" << endl;
    }
};

class C:public A //类C由类A派生而来
{
private:
    int y;
    B b; //对象成员
public:
    C(int xp, int yp):A(xp), b() //派生类的构造函数，基类和对象成员都在初始化表中完成初始化
    {
        y = yp;
        cout << "C的构造函数被执行" << endl;
    }
    ~C() //析构函数
    {
        cout << "C的析构函数被执行" << endl;
    }
};

int main()
{
    C c(1, 2); //声明一个C类对象c
    return 0; //main函数执行完毕，c撤销，析构函数触发执行
}

```


执行结果：



```
A的构造函数被执行
B的构造函数被执行
C的构造函数被执行
C的析构函数被执行
B的析构函数被执行
A的析构函数被执行
Press any key to continue
```

4.6 多基派生类的构造函数和析构函数

多基派生时，派生类的构造函数格式如（假设有 N 个基类）：

派生类名(总参数表)：基类名 1(参数表 1)，基类名 2(参数表 2)，……，基类名 N(参数表 N)

```
{
    //函数体
}
```

- 和前面所讲的单基派生类似，总参数表中包含了后面各个基类构造函数需要的参数。
- 多基派生和单基派生构造函数完成的任务和执行顺序并没有本质不同，唯一一点区别在于首先要执行所有基类的构造函数，再执行派生类构造函数中初始化表达式的其他内容和构造函数体，各基类构造函数的执行顺序与其在初始化表中的顺序无关，而是由定义派生类时指定的派生类顺序决定的。
- 析构函数的执行顺序同样是与构造函数的执行顺序相反。

4.7 继承不是万金油

前面已提及继承的重要性，使得代码结构清晰，大大提高了程序的可复用性，因此，很多初学者容易犯的错误就是把继承当成灵丹妙药，不管三七二十一拿来继承一下再说，其实，在面向问题空间的对象组织方面，不只有继承，还有对象组合，更高阶的结构还有聚合等，但从 C++ 的本质来看，本节讨论下继承和组合的关系。

如果两个类没有关联，仅仅是为了使一个类的功能更多而让其去继承另一个类，这种方法要不得，继承不是万金油，毫无疑问的继承就像乱拉亲戚，会让条理有序的关系变得一团糟。从逻辑上说，继承使一种 a kind of 的关系（AKO）或者说 IS-A（“是一个”），汽车是车，因此，汽车类可以从普通的车类继承而来，轮子类就不能从汽车类继承来，轮子是汽车的一个部件，轮子可以作为汽车类的对象成员，这就是“组合”（composition）。

4.7.1 组合

某类以另一个类对象作数据成员，称为组合，在逻辑上，如果类 A 是类 B 的一部分（a part of），不要从 A 类派生出类 B，而应当采用组合的方式，《高质量 C++ 编程指南》中“眼睛、鼻子、嘴巴、耳朵和头部”的范例很好地解释了组

合的本质：眼睛、鼻子、嘴巴、耳朵分别是头部的一部分，头部并不是从眼睛、鼻子、嘴巴、耳朵继承来的。

例子：

```
//组合还是继承
#include <iostream>
using namespace std;

class Eye    //眼睛
{
public:
    void Look() {cout << "Eye.Look()." << endl;}
};
class Nose   //鼻子
{
public:
    void Smell() {cout << "Nose.Smell()." << endl;}
};
class Mouth  //嘴
{
public:
    void Eat() {cout << "Mouth.Eat()." << endl;}
};
class Ear    //耳朵
{
public:
    void Listen() {cout << "Ear.Listen()." << endl;}
};

//组合方式：逻辑很清晰，后续扩展很方便。
class Head
{
private:
    Eye m_eye;
    Nose m_nose;
    Mouth m_mouth;
    Ear m_ear;
public:
    void Look()
    {
        m_eye.Look();
    }
    void Smell()
    {
        m_nose.Smell();
    }
    void Eat()
    {
        m_mouth.Eat();
    }
    void Listen()
    {
        m_ear.Listen();
    }
};
```

```

//继承方式，会给后续设计带来很多逻辑上的问题
class HeadX : public Eye, public Nose, public Mouth, public Ear
{
};

int main()
{
    Head h;
    h.Look();
    h.Smell();
    h.Eat();
    h.Listen();

    cout << endl;

    HeadX hx;
    hx.Look();
    hx.Smell();
    hx.Eat();
    hx.Listen();

    cout << endl;

    return 0;
}

```

执行结果:

```

Eye.Look().
Nose.Smell().
Mouth.Eat().
Ear.Listen().

Eye.Look().
Nose.Smell().
Mouth.Eat().
Ear.Listen().

Press any key to continue

```