第 5 章 多态性与虚函数

多态性是面向对象程序设计的重要特征之一。如果一种语言只支持类而不支持多态，是不能称为面向对象语言的，只能说是基于对象的，如 Visual Basic、Ada 就属于此类。C++ 支持多态性，在 C++ 程序设计中应用多态性机制，可以设计和实现一个易于扩展的系统。

本章介绍多态性的概念及分类，以及虚函数、纯虚函数和抽象类的概念、定义及使用方法。

5.1 什么是多态性

多态性(polymorphism)是面向对象程序设计的一个重要特性。在面向对象方法中一般是这样表述多态性的：向不同的对象发送同一个消息，不同的对象在接收时会有不同的反应，产生不同的动作。也就是说，每个对象可以用自己的方式去响应共同的消息。

在 C++ 程序设计中，多态性是指用一个名字定义不同的函数，这些函数执行不同但又类似的操作，从而可以使用相同的调用方式来调用这些具有不同功能的同名函数。这样，就可以达到用同样的接口访问不同功能的函数，从而实现“一个接口，多种方法”。

C++ 中的多态性可以分为 4 类：参数多态、包含多态、重载多态和强制多态。前面两种统称为通用多态，而后面两种统称为专用多态。

参数多态如函数模板和类模板(在本书第 8 章介绍)。由函数模板实例化的各个函数都具有相同的操作，而这些函数的参数类型却各不相同。同样地，由类模板实例化的各个类都具有相同的操作，而操作对象的类型是各不相同的。

包含多态是研究类族中定义于不同类中的同名成员函数的多态行为，主要是通过虚函数来实现的。

重载多态如函数重载、运算符重载等。前面学习过的普通函数及类的成员函数的重载都属于重载多态。运算符重载将在第 7 章介绍。

强制多态是指将一个变元的类型加以变化，以符合一个函数(或操作)的要求，如加法运算符在进行浮点数与整型数相加时，首先进行类型强制转换，把整型数变为浮点数再相加的情况，就是强制多态的实例。

5.2 向上类型转换

根据赋值兼容规则，可以使用派生类的对象代替基类对象。向上类型转换就是把一个派生类的对象作为基类的对象来使用。向上类型转换中有 3 点需要特别注意。第一，

向上类型转换是安全的；第二，向上类型转换可以自动完成；第三，向上类型转换的过程中会丢失子类型信息。下面通过一个程序来加深对它的理解。

【例 5-1】 向上类型转换示例。

```
#include <iostream>
using namespace std;
class Point
{public:
    Point(double a =0, double b =0) {  x =a; y =b;  }
    double area()
    {   cout <<"Call Point's area function. " <<endl;
        return 0. 0;
    }
protected:
    double x, y;          //点的坐标值
};
class Rectangle: public Point
{public:
    Rectangle(double a =0, double b =0, double c =0, double d =0): Point(a, b)
    {  x1 =c; y1 =d;   }
    double area()
    {   cout <<"Call Rectangle's area function. " <<endl;
        return (x1 -x) * (y1 -y);
    }
protected:
    double x1, y1;        //长方形右下角点的坐标值，基类中 x、y 为左上角点的坐标值
};
class Circle: public Point
{public:
    Circle(double a =0, double b =0, double c =0): Point(a, b){  r =c;  }
    double area()
    {   cout <<"Call Circle's area function. " <<endl;
        return 3. 14 * r * r;
    }
protected:
    double r;             //圆半径，基类中 x、y 为圆心坐标点的坐标值
};
double calcArea(Point &ref){   return ( ref. area() );  }
int main()
{   Point p(0, 0);
    Rectangle r(0, 0, 1, 1);
    Circle c(0, 0, 1);
    cout <<calcArea(p) <<endl;
    cout <<calcArea(r) <<endl;
```

```
    cout <<calcArea(c) <<endl;
    return 0;
}
```

程序运行结果如下：

```
Call Point's area function.
0
Call Point's area function.
0
Call Point's area function.
0
```

函数 calcArea 接收一个 Point 类的对象，但也不拒绝任何 Point 派生类的对象。在 main 函数中，可以看出，无需类型转换，就能将 Rectangle 类或 Circle 类的对象传给 calcArea。这是可接受的，在 Point 类中有的接口必然存在于 Rectangle 类和 Circle 类中，因为 Rectangle 类和 Circle 类都是 Point 类的公用派生类。Rectangle 类和 Circle 类到 Point 类的向上类型转换会使 Rectangle 类和 Circle 类的接口“变窄”，但不会窄过 Point 类的整个接口。

从运行结果来看，3 次调用都是调用的 Point::area()，这不是用户所希望的输出。用户希望通过使用指向基类对象的指针或基类对象的引用能够调用基类和派生类对象的成员，即想得到如下运行结果：

```
Call Point's area function.
0
Call Rectangle's area function.
1
Call Circle's area function.
3.14
```

也就是当通过指向基类对象的引用 ref 调用 area 时，如果 ref 是 Point 类对象的引用，就调用 Point 类中定义的 area 函数；如果 ref 是 Rectangle 类对象的引用或 Circle 类对象的引用，就调用 Rectangle 类中定义的 area 函数或 Circle 类中定义的 area 函数，而不是都调用 Point 类中定义的 area 函数。为了解决这个问题，需要知道绑定这个概念。

5.3　功能早绑定和晚绑定

多态从实现的角度来讲可以划分为两类：编译时的多态和运行时的多态。前者是在编译的过程中确定了同名操作的具体操作对象，而后者则是在程序运行过程中才动态地确定操作所针对的具体对象。这种确定操作的具体对象的过程就是绑定(binding)。绑定是指计算机程序自身彼此关联的过程，也就是把一个标识符名和一个存储地址联系在一起的过程；用面向对象的术语讲，就是把一条消息和一个对象的方法相结合的过程。按照绑定进行的阶段的不同，可以分为两种不同的绑定方法：功能早绑定和功能晚绑定，这

两种绑定方法分别对应着多态的两种实现方式。

绑定工作在编译连接阶段完成的情况称为功能早绑定。因为绑定过程是在程序开始执行之前进行的。在编译、连接过程中，系统就可以根据类型匹配等特征确定程序中操作调用与执行该操作的代码的关系，即确定了某一个同名标识到底是要调用哪一段程序代码。有些多态类型，其同名操作的具体对象能够在编译、连接阶段确定，通过功能早绑定解决，比如重载多态、强制多态和参数多态。对于例5-1中的calcArea函数，在程序编译阶段，通过基类Point类的引用ref调用的area函数被绑定到Point类的函数上，因此，在执行函数calcArea中的ref.area()操作时，每次都执行Point类的area函数。这是功能早绑定的结果。

与功能早绑定相对应，绑定工作在程序运行阶段完成的情况称为功能晚绑定。在编译、连接过程中无法解决的绑定问题，要等到程序开始运行之后再来确定，包含多态中操作对象的确定就是通过功能晚绑定完成的。

一般而言，编译型语言（如C、PASCAL）都采用功能早绑定，而解释性语言（如LISP、Prolog）都采用功能晚绑定。功能早绑定要求在程序编译时就知道调用函数的全部信息，因此，这种绑定类型的函数调用速度很快，效率高，但缺乏灵活性；而功能晚绑定的方式恰好相反，采用这种绑定方式，一直要到程序运行时才能确定调用哪个函数，它降低了程序的运行效率，但增强了程序的灵活性。C++由C语言发展而来，为了保持C语言的高效性，C++仍是编译型的，仍采用功能早绑定。好在C++的设计者想出了“虚函数”的机制，解决了这个问题。利用虚函数机制，C++可部分地采用功能晚绑定。这就是说，C++实际上是采用了功能早绑定和功能晚绑定相结合的编译方法。

在C++中，编译时的多态性主要是通过函数重载和运算符重载实现的。运行时的多态性主要是通过虚函数来实现的。函数重载在前面章节中已作了介绍，运算符重载将在第7章介绍，本章重点介绍虚函数以及由它们提供的多态性机制。

5.4 实现功能晚绑定——虚函数

虚函数提供了一种更为灵活的多态性机制。虚函数允许函数调用与函数体之间的联系在运行时才建立，也就是在运行时才决定如何动作，即所谓的功能晚绑定。

5.4.1 虚函数的定义和作用

虚函数的定义是在基类中进行的，在成员函数原型的声明语句之前冠以关键字virtual，从而提供一种接口。一般虚成员函数的定义语法是：

```
virtual 函数类型 函数名(形参表)
{
    函数体
}
```

当基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、

参数类型的顺序，都必须与基类中的原型完全相同。

虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过指向基类对象的指针或基类对象的引用来访问基类和派生类中的同名函数。下面的程序将例 5-1 中的函数 area 定义为虚函数，以达到预期的效果。

【例 5-2】 虚函数的作用示例。

```
#include <iostream>
using namespace std;
class Point
{public:
    Point(double a =0, double b =0) {  x =a; y =b;   }
    virtual double area()
    {   cout <<"Call Point's area function. " <<endl;
        return 0. 0;
    }
protected:
    double x, y;          // 点的坐标值
};
class Rectangle: public Point
{public:
    Rectangle(double a =0, double b =0, double c =0, double d =0): Point(a, b)
    {   x1 =c; y1 =d;   }
    double area()
    {   cout <<"Call Rectangle's area function. " <<endl;
        return (x1 -x) * (y1 -y);
    }
protected:
    double x1, y1;        //长方形右下角点的坐标值，基类中 x、y 为左上角点的坐标值
};
class Circle: public Point
{public:
    Circle(double a =0, double b =0, double c =0): Point(a, b){  r =c;  }
    double area()
    {   cout <<"Call Circle's area function. " <<endl;
        return 3. 14 * r * r;
    }
protected:
    double r;             //圆半径，基类中 x、y 为圆心坐标点的坐标值
};
double calcArea(Point &ref){  return(ref. area() );  }
int main()
{
    Point p(0, 0);
    Rectangle r(0, 0, 1, 1);
```

```
    Circle c(0, 0, 1);
    cout <<calcArea(p) <<endl;
    cout <<calcArea(r) <<endl;
    cout <<calcArea(c) <<endl;
    return 0;
}
```

程序运行结果如下：

```
Call Point's area function.
0
Call Rectangle's area function.
1
Call Circle's area function.
3.14
```

为什么把基类中的 area 函数定义为虚函数时，程序的运行结果就正确了呢？这是因为，关键字 virtual 指示 C++ 编译器，函数调用 ref. area()要在运行时确定所要调用的函数，即要对该调用进行功能晚绑定。因此，程序在运行时根据引用 ref 所引用的实际对象，调用该对象的成员函数。

可见，继承、虚函数、指向基类对象的指针或基类对象的引用的结合可使 C++ 支持运行时的多态性，而多态性对面向对象的程序设计是非常重要的，实现了在基类中定义派生类所拥有的通用接口，而在派生类中定义具体的实现方法，即常说的“同一接口，多种方法”，它帮助程序员处理越来越复杂的程序。

下面再通过一个例子来说明虚函数在实际编程中的作用。

【例 5-3】 有一个交通工具类 Vehicle，将它作为基类派生出汽车类 MotorVehicle，再将汽车类 MotorVehicle 作为基类派生出小汽车类 Car 和卡车类 Truck，声明这些类并定义一个虚函数用来显示各类信息。程序如下：

```
#include <iostream>
using namespace std;
class Vehicle                          //声明基类 Vehicle
{public:
    virtual void message()             //虚成员函数
    {  cout <<"Call Vehicle's message function. " <<endl;  }
private:
    int wheels;                        //车轮个数
    float weight;                      //车的质量
};
class MotorVehicle: public Vehicle  //声明 Vehicle 类的公用派生类 MotorVehicle
{public:
    void message(){  cout <<"Call MotorVehicle's message function. " <<endl;  }
private:
    int passengers;                    //承载人数
```

```
};
class Car: public MotorVehicle        //声明 MotorVehicle 类的公用派生类 Car
{public:
    void message(){  cout <<"Call Car's message function. " <<endl;  }
private:
    float  engine;                    //发动机的马力数
};
class Truck: public MotorVehicle      //声明 MotorVehicle 类的公用派生类 Truck
{public:
    void message(){  cout <<"Call Truck's message function. " <<endl;  }
private:
    int loadpay ;                     //载重量
};
int main()
{    Vehicle v, * p =nullptr;         //声明 Vehicle 类对象 v 和基类指针 p
     MotorVehicle m;                  //声明 MotorVehicle 类对象 m
     Car c;                           //声明 Car 类对象 c
     Truck t;                         //声明 Truck 类对象 t
     p =&v;                           // Vehicle 类指针 p 指向 Vehicle 类对象 v
     p ->message();                   //调用基类成员函数
     p =&m;                           // Vehicle 类指针 p 指向 MotorVehicle 类对象 m
     p ->message();                   //调用 MotorVehicle 类成员函数
     p =&c;                           // Vehicle 类指针 p 指向 Car 类对象 c
     p ->message();                   //调用 Car 类成员函数
     p =&t;                           // Vehicle 类指针 p 指向 Truck 类对象 t
     p ->message();                   //调用 Truck 类成员函数
     return 0;
}
```

程序运行结果如下：

```
Call Vehicle's message function.
Call MotorVehicle's message function.
Call Car's message function.
Call Truck's mMessage function.
```

程序只在基类 Vehicle 中显式定义了 message 为虚函数。C++ 规定，如果在派生类中，没有用 virtual 显式地给出虚函数声明，这时系统就会遵循以下的规则来判断一个成员函数是不是虚函数：

(1) 该函数与基类的虚函数有相同的名称。

(2) 该函数与基类的虚函数有相同的参数个数及相同的对应参数类型。

(3) 该函数与基类的虚函数有相同的返回类型或者满足赋值兼容规则的指针、引用型的返回类型。

派生类的函数满足了上述条件，就被自动确定为虚函数。因此，在本程序的派生类

MotorVehicle、Car 和 Truck 中 message 仍为虚函数。

下面对虚函数的定义做几点说明：

（1）通过定义虚函数来使用 C++ 提供的多态性机制时，派生类应该从它的基类公用派生。之所以有这个要求，是因为在赋值兼容规则的基础上来使用虚函数的，而赋值兼容规则成立的前提条件是派生类从其基类公用派生。

（2）必须首先在基类中定义虚函数。由于“基类”与“派生类”是相对的，因此，这项说明并不表明必须在类等级的最高层类中声明虚函数。在实际应用中，应该在类等级内需要具有动态多态性的几个层次中的最高层类内首先声明虚函数。

（3）在派生类中对基类声明的虚函数进行重新定义时，关键字 virtual 可以写也可以不写。但为了增强程序的可读性，最好在对派生类的虚函数进行重新定义时也加上关键字 virtual。

如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

（4）虽然使用对象名和点运算符的方式也可以调用虚函数，如语句 c.message()可以调用虚函数 Car::message()。但是这种调用是在编译时进行的功能早绑定，它没有充分利用虚函数的特性。只有通过指向基类对象的指针或基类对象的引用访问虚函数时才能获得运行时的多态性。

（5）一个虚函数无论被公用继承多少次，它仍然保持其虚函数的特性。

（6）虚函数必须是其所在类的成员函数，而不能是友元函数，也不能是静态成员函数，因为虚函数调用要靠特定的对象来决定该激活哪个函数。但是虚函数可以在另一个类中被声明为友元函数。

（7）内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时仍将其看作是非内联的。

（8）构造函数不能是虚函数。因为虚函数作为运行过程中多态的基础，主要是针对对象的，而构造函数是在对象产生之前运行的，因此虚构造函数是没有意义的。

（9）析构函数可以是虚函数，而且通常说明为虚函数。

5.4.2 虚析构函数

在析构函数前面加上关键字 virtual 进行说明，则称该析构函数为虚析构函数。虚析构函数的声明语法为：

```
virtual ~类名();
```

看下面的例子。

【例 5-4】 在交通工具类 Vehicle 中使用虚析构函数。

```
#include <iostream>
using namespace std;
class Vehicle                          //声明基类 Vehicle
{public:
```

```
    Vehicle(){ }                          //构造函数
    virtual ~Vehicle()                    //虚析构函数
    {   cout <<"Vehicle :: ~Vehicle()" <<endl;   }
private:
    int wheels;
    float weight;
};
class MotorVehicle: public Vehicle  //声明 Vehicle 的公用派生类 MotorVehicle
{public:
    MotorVehicle(){ }                     //派生类构造函数
    ~MotorVehicle()                       //派生类析构函数
    {   cout <<"MotorVehicle :: ~MotorVehicle()" <<endl;   }
private:
    int passengers;
};
int main()
{     Vehicle *p =nullptr;                //声明 Vehicle 类指针 p
      p =new MotorVehicle;
      delete p;
      return 0 ;
}
```

程序运行结果如下：

```
MotorVehicle :: ~MotorVehicle()
Vehicle :: ~Vehicle()
```

先调用了派生类 MotorVehicle 的析构函数，再调用了基类 Vehicle 的析构函数，符合用户的愿望。

如果类 Vehicle 中的析构函数不用虚函数，则程序运行结果如下：

```
Vehicle :: ~Vehicle()
```

系统只执行基类 Vehicle 的析构函数，而不执行派生类 MotorVehicle 的析构函数。

如果将基类的析构函数声明为虚函数时，由该基类所派生的所有派生类的析构函数也都自动成为虚函数，即使派生类的析构函数与基类的析构函数名字不相同。

特别提醒：

当基类的析构函数为虚函数时，无论指针指向的是同一类族中的哪一个类对象，系统都会采用动态关联，调用相应的析构函数，对该对象所涉及的额外内存空间进行清理工作。最好把基类的析构函数声明为虚函数，这将使所有派生类的析构函数自动成为虚函数。这样，如果程序中显式地用了 delete 运算符准备删除一个对象，而 delete 运算符的操作对象用了指向派生类对象的基类指针，则系统会首先调用派生类的析构函数，再调用基类的析构函数，这样整个派生类的对象被完全释放。

5.4.3 虚函数与重载函数的比较

在一个派生类中重新定义基类的虚函数不同于一般的函数重载：

（1）函数重载处理的是同一层次上的同名函数问题，而虚函数处理的是同一类族中不同派生层次上的同名函数问题，前者是横向重载，后者可以理解为纵向重载。但与重载不同的是：同一类族的虚函数的首部是相同的，而函数重载时函数的首部是不同的（参数个数或类型不同）。

（2）重载函数可以是成员函数或普通函数，而虚函数只能是成员函数。

（3）重载函数的调用是以所传递参数序列的差别作为调用不同函数的依据；虚函数是根据对象的不同去调用不同类的虚函数。

（4）虚函数在运行时表现出多态功能，这是C++的精髓；而重载函数则在编译时表现出多态性。

5.5 纯虚函数和抽象类

抽象类是带有纯虚函数的类。为了学习抽象类，我们先来了解纯虚函数。

1. 纯虚函数

有时，基类往往表示一种抽象的概念，它并不与具体的事物相联系。如下面的例5-5中定义一个公共基类Shape，它表示一个封闭图形。然后，从Shape类可以派生出三角形类、矩形类和圆类，这个类等级中的基类Shape体现了一个抽象的概念，在Shape中定义一个求面积的函数和显示图形信息的函数显然是无意义的，但是可以将其说明为虚函数，为它的派生类提供一个公共的接口，各派生类根据所表示的图形的不同重新定义这些虚函数，以提供求面积的各自版本。为此，C++引入了纯虚函数的概念。

纯虚函数是一个在基类中说明的虚函数，它在该基类中没有定义，但要求在它的派生类中必须定义自己的版本，或重新说明为纯虚函数。

纯虚函数的定义形式如下：

```
class  类名
{  …
   virtual  函数类型  函数名(参数表) =0;
   …
};
```

此格式与一般的虚函数定义格式基本相同，只是在后面多了“=0”。声明为纯虚函数之后，基类中就不再给出函数的实现部分。纯虚函数的函数体由派生类给出。

【例5-5】 定义一个公共基类Shape，它表示一个封闭平面几何图形。然后，从Shape类派生出三角形类Triangle、矩形类Rectangle和圆类Circle，在基类中定义纯虚函数show和area，分别用于显示图形信息和求相应图形的面积，并在派生类中根据不同的图形实现相应的函数。要求实现运行时的多态性。