第 5 章　多态性与虚函数

一种语言如果不支持多态就不能被称为面向对象程序设计语言，多态性是面向对象程序设计的重要特征之一。多态又可分为编译时的多态和运行时的多态，从第 3 章学到的运算符重载就属于编译时的多态，本章主要讨论函数重载和建立在虚函数基础上的运行时的多态。

5.1　多　态　性

生活中也常存在多态性，例如学校的上课铃声响了，这时，教师会去上课，学生会回到教室，校广播站人员会关掉广播。不同的人员对同一个事件产生了不同的行为，这就是多态性在日常生活中的表现。

面向对象程序设计中的多态性，指的是不同的对象在得到相同的消息时产生了不同的行为，也就是说不同的对象以自己的方式去响应相同的消息。消息主要指的是函数的调用，不同的行为是指执行不同的函数。

5.1.1　多态的类型

C++ 的多态性可以分为静态多态性和动态多态性两类。静态多态性是系统在程序编译过程中根据函数形参的不同来决定到底调用哪个函数，因而也被称为编译时的多态。例如第 3 章讲的运算符重载就是具体的静态多态性的表现。动态多态性则是程序在编译、链接过程中还无法确定要调用哪个函数，必须要等到运行时才能确定具体操作的对象，因此也被称为运行时的多态。动态多态性需要通过虚函数来实现。

5.1.2　函数重载

C++ 支持函数的重载。

1. 非派生重载

函数名相同，但函数的形式参数表互不相同。例如下面几个求和函数：

(1) int sum(int a,int b);

(2) int sum(int a,int b,int c);

(3) float sum(float a,float b,float c)。

其中(1)和(2)是形参的个数不同，(2)和(3)是形参的类型不同，(1)和(3)是形参的个数和类型都不相同。重载函数的返回类型可以相同也可以不同，但不可以是形参的个数和类型都相同而函数的返回类型不同。

【例 5-1】　函数重载。

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

```
int main()
{
    int sum(int a,int b);                                   //函数声明
    int sum(int a,int b,int c);                             //函数声明
    float sum(float a,float b,float c);                     //函数声明
    int i,i1,i2,i3;
    float f,f1,f2,f3;
    cin>>i1>>i2>>i3;                                        //输入三个整数
    cin>>f1>>f2>>f3;                                        //输入三个实数
    i=sum(i1,i2);                                           //求两个整数之和
    cout<<"sum_int2="<<i<<endl;
    i=sum(i1,i2,i3);                                        //求三个整数之和
    cout<<"sum_int3="<<i<<endl;
    f=sum(f1,f2,f3);                                        //求三个实数之和
    cout<<"sum_float3="<<f<<endl;
    return 0;
}
int sum(int a,int b)                                        //定义求两个整数之和的函数
{
    return(a+b);
}
int sum(int a,int b,int c)                                  //定义求三个整数之和的函数
{
    return(a+b+c);
}
float sum(float a,float b,float c)                          //定义求三个实数之和的函数
{
    return(a+b+c);
}
```

程序运行情况如下：

```
1 3 5 2.1 4.1 6.1↙
sum_int2=4
sum_int3=9
sum_float3=12.3
```

虽然定义了三个同名的函数 sum()，但在编译过程中，系统会根据调用函数时的实际参数个数、类型自动匹配相应的函数，而不会出现错误。例如在编译“i＝sum(i1,i2,i3);”语句时，因为函数有三个参数，所以不可能是第 1 个函数，又因为实际参数 i1、i2、i3 都是基本整型数据，所以确认当前语句调用的只可能是第 2 个函数。

2. 派生类重载基类函数

基类成员函数也可以在派生类中重载。

【例 5-2】 重载基类函数。

```
#include<iostream>
```

```
using namespace std;
class Animal
{
private:
string name;
public:
Animal(string n)
{
name=n;
}
void enjoy()
{
cout<<"叫声"<<endl;
}
};
class Cat: public Animal
{
public:
string eyesColor;
Cat(string n,string c): Animal(n)
{
eyesColor=c;
}
void enjoy()
{
cout<<"喵喵"<<endl;
}
void enjoy(int i)
{
for(;i>=1;i--)cout<<"喵";
cout<<endl;
}
};
void main()
{
Animal a("cat");
Cat c("catname","blue");
a.enjoy();                          //调用 Animal 类的 enjoy()函数,对 a 的数据操作
c.enjoy();                          //调用 Cat 类的 enjoy()函数,对 c 的数据操作
c.enjoy(5);                         //调用 Cat 类的 enjoy(int i)函数,对 c 的数据操作
c.Animal::enjoy();                  //调用 Animal 类的 enjoy()函数,对 c 的数据操作
}
```

输出结果如下：

```
叫声
```

喵喵
喵喵喵喵喵
叫声

通过例 5-2 可以把基类成员函数在派生类中重载的编译区分方法做一个总结。

(1) 参数的特征,即根据参数的类型、个数、顺序的不同来加以区分。

例如: c. enjoy()没有参数,而 c. enjoy(5)有一个参数且是整数类型,所以它们不是同一函数,编译时能够区分。这一点和非派生重载的区分方法是相同的。

(2) 使用::加以区分。

例如: c. Animal::enjoy()有别于 c. enjoy()。类 Cat 继承了类 Animal 中的 enjoy()函数,同时类 Cat 又自己定义了一个无参数的同名函数 enjoy(),如果直接通过 Cat 的对象c 来调用 enjoy()函数,实际上调用的是 Cat 自己定义的函数 enjoy()。如果需要调用基类的 enjoy()函数,则必须使用::号,即 c. Animal::enjoy()。

当然,如果派生类是从多个基类继承了相同名字的函数,也就构成了多重继承的二义性在成员函数上的体现。也需要使用::来加以区分,即对象名. 基类名::函数名。

(3) 根据类对象加以区分。

例如: a. enjoy()调用 Animal 类的 enjoy()函数,c. enjoy()调用 Cat 类的 enjoy()函数,以类对象加以区分,不会产生歧义。

5.1.3 联编

多态的实现是通过联编来完成的。在执行存在多态性的语句时,必须确定要执行什么样的行为,也就是确定到底调用哪个函数。确定多态性的语句要调用哪个函数的过程称为联编。

联编有两种方式: 静态联编和动态联编。

1. 静态联编

在程序开始运行之前的编译、链接过程中,系统就能确定多态性语句要调用哪个函数的过程称为静态联编。由于联编是在程序运行之前完成的,所以也称静态联编为早期联编。前文中讲到的函数重载和运算符重载都属于静态联编。

2. 动态联编

如果在程序编译、链接过程中还无法确定要调用哪个函数,那么联编需要在程序运行时才能完成,这时的联编被称为动态联编或晚期联编。动态联编需要通过虚函数来实现。

5.2 虚 函 数

在讲解虚函数之前,先看一下例 5-3。

【例 5-3】 通过基类的指针访问派生类对象。

```
#include<iostream>
using namespace std;
class Animal
{
```

```
private:
string name;
public:
Animal(string n)
{
name=n;
}
void enjoy()
{
cout<<"叫声"<<endl;
}
};
class Cat: public Animal
{
public:
string eyesColor;
Cat(string n,string c): Animal(n)
{
eyesColor=c;
}
void enjoy()
{
cout<<"喵喵"<<endl;
}
};
class Dog: public Animal
{
private:
string furColor;
public:
Dog(string n,string c): Animal(n)
{
furColor=c;
}
void enjoy()
{
cout<<"汪汪"<<endl;
}
};
void main()
{
Cat c("catname","blue");
Dog d("dogname","black");
Animal * pet;
pet=&c;
```

```
pet->enjoy();
pet=&d;
pet->enjoy();
}
```

输出结果如下：

```
叫声
叫声
```

有时，我们会指着小猫问孩子：这个小动物怎么叫？实际上，问话中已经隐含着具体的小动物即小猫，同样的问话我们也可以用在其他动物上，例如小狗、小鸭等。当然，不同的动物叫声是不同的。如果我们把动物看成是基类，那么猫、狗等就是这个基类的派生类。动物的叫声可以看成是基类的函数，猫、狗等的叫声是派生类的函数。

例 5-3 就是这样的问题，虽然实验已经证明基类对象的指针可以用派生类对象的地址进行初始化，但还不能达到我们希望通过基类的指针 pet 来调用派生类 Dog 的成员函数 enjoy()的目的，指针 pet 引用的对象的函数只能是基类 Animal 的 enjoy()函数，而不是派生类 Dog 的成员函数 enjoy()。这种调用关系是在编译的过程中完成的，因此不是运行时的多态，而是静态多态性的体现。

如果想实现运行时的多态，除了具备基类的指针可以访问基类及派生类的同名函数这一条件外，还必须将基类的同名函数定义为虚函数。

声明虚函数需要使用关键字 virtual，格式为：

```
virtual 函数类型 函数名称(形式参数表)
```

引入虚函数的声明后，例 5-3 可以改为例 5-4。

【例 5-4】 虚函数的应用。

```
#include<iostream>
using namespace std;
class Animal
{
private:
string name;
public:
Animal(string n)
{
name=n;
}
virtual void enjoy()
{
cout<<"叫声"<<endl;
}
};
class Cat: public Animal
{
```

```
public:
string eyesColor;
Cat(string n,string c): Animal(n)
{
eyesColor=c;
}
void enjoy()
{
cout<<"喵喵"<<endl;
}
};
class Dog: public Animal
{
private:
string furColor;
public:
Dog(string n,string c): Animal(n)
{
furColor=c;
}
void enjoy()
{
cout<<"汪汪"<<endl;
}
};
void main()
{
Cat c("catname","blue");
Dog d("dogname","black");
Animal * pet;
pet=&c;
pet->enjoy();
pet=&d;
pet->enjoy();
}
```

输出结果如下：

```
喵喵
汪汪
```

通过虚函数的使用，成功实现了通过基类的指针访问派生类对象的函数。两次调用只采用了同一种调用方式 pet->enjoy()，却实现了对不同对象的同名函数的调用，也就是对于同一个消息，不同的对象以不同的方式进行了响应。另外，这种多态不是在编译和链接时联编的，而是要等到程序运行时才能确定具体要执行的是哪个函数，这种多态属于运行时的多态，需要动态联编来实现。

在没有引入虚函数之前，例 5-4 的运行结果也可以做到，例如可以通过对象名来调用 enjoy()函数，即 c.enjoy()和 d.enjoy()，也可以分别定义指向 Cat 类和 Dog 类的指针 pc 和 pd，让 pc 和 pd 分别指向对象 c 和 d 后，再用 pc->enjoy()和 pd->enjoy()调用。这两种方法都是正确和可行的，但是，如果基类有多个派生类甚至多级派生类，这些派生类又都有同名的函数 enjoy()，则在调用不同类的 enjoy()函数时这两种方法显然都很烦琐而且容易出错。这时，使用虚函数和指向基类指针结合的方式却能很好地解决这一问题。具体方法总结如下：

(1) 在基类中声明虚函数，在派生类中重新定义同名的函数，并且函数类型和参数要和基类的虚函数完全一致。此时，派生类的同名函数会自动成为虚函数。

(2) 定义基类对象的指针变量，并通过这个指针指向不同派生类对象就可以实现对不同对象的同名函数的调用。

通过例 5-3 和例 5-4 我们发现，静态联编时，执行的是变量定义时所使用的那个类中的成员函数。动态联编时，执行的是指针所指向的那个变量所属类中的成员函数。当且仅当满足以下 4 个条件时为动态联编，否则为静态联编。

(1) 有继承；

(2) 有函数重载；

(3) 基类中定义虚函数；

(4) 基类指针指向派生类对象。

5.3 纯虚函数和抽象类

基类中不需要定义具体行为的函数可以定义为纯虚函数。反映一类事物共有特性的类可以定义为抽象类。

5.3.1 纯虚函数

例 5-4 中的基类 Animal，实际上并不是具体的动物，只是为了派生其他具体的动物类而建立的，在对这个基类和它的派生类做比较后发现，这个类中有一个函数 enjoy()有些奇怪，因为在派生类中 enjoy()都是用来显示具体动物类是如何发声的，如 Cat 类的叫声是“喵喵”，Dog 类的叫声是“汪汪”，那么基类 Animal 的叫声是什么呢，是“叫声”吗？显然这是不合理的，这是因为基类 Animal 是一个抽象的概念，而这个叫声是无法具体确定的，也是没有意义的。之所以在基类中定义这样一个函数是考虑到将来在派生类中的使用。

在实际编程中，会将基类中有些成员函数定义为虚函数，其实这并不是基类自己的需要，而是为派生类做的准备工作。基类中的函数只是定义了一个函数名，具体要实现什么样的功能都留给派生类去完成。上文中提到基类 Animal 的函数 enjoy()之所以奇怪就是因为这个函数其实根本没有必要去完成任何没有意义的操作，只需要将其定义成一个虚函数就可以了，没有意义的函数体也不用写，只在函数原型后加上“=0”就行，具体的函数形式变为：

```
virtual void enjoy()=0;
```

这其实就是声明 enjoy()为一个纯虚函数的形式。

纯虚函数声明的格式是：

```
virtual 函数类型 函数名(参数表)=0;
```

对于纯虚函数声明的几点说明：

(1) 纯虚函数只能在基类中声明；

(2) 纯虚函数不可以被调用；

(3) 纯虚函数不是函数体为空，而是没有函数体；

(4) 包含纯虚函数的派生类都应该重载该纯虚函数，否则继承来的纯虚函数在派生类中仍是纯虚函数。

5.3.2 抽象类

抽象类就是包含了一个或多个纯虚函数的类。如果上文中提到的 Animal 类中的 enjoy 函数已经被声明为纯虚函数，基类 Animal 就成了一个抽象类。作为抽象类的 Animal 类是不能定义对象的，它只能作为专门的基类被派生，例如派生出类 Cat 和类 Dog。

在处理实际问题时，基类通常都是比较抽象的概念，如水果、衣物、动物等。抽象类其实就是这种不能用来定义对象而专门用来做派生用途的类。

在使用抽象类时应注意以下几点：

(1) 抽象类不能声明对象。

(2) 可以声明一个抽象类的指针和引用，以便于访问派生类的成员，实现运行时的多态。

(3) 抽象类中可以定义一个或多个纯虚函数，也可以定义有具体功能的函数并继承给派生类对象使用。

(4) 若抽象类的派生类中仍没有实现纯虚函数的功能，则派生类依然是抽象类。只有派生类实现了所有纯虚函数的功能才可以用来创建对象。

习　　题

5-1　什么是函数重载？函数重载有何作用？

5-2　什么是面向对象程序设计中的多态性？C++ 是如何实现多态的？

5-3　可以生成抽象类的对象吗？为什么？

5-4　重载与虚函数在实现方法上有什么相同和不同的地方？

5-5　什么是抽象类？抽象类有何作用？抽象类的纯虚函数必须在派生类中实现吗？

5-6　定义表示形状的抽象类 Shape，以它为基类派生出三角形类 Triangle、圆形类 Circle 和矩形类 Rectangle，并用虚函数 area 分别计算这三种图形的面积。