第12章 多态性与虚函数

12.1 多态性的概念

多态性(**polymorphism**)是面向对象程序设计的一个重要特征。利用多态性可以设计和实现一个易于扩展的系统。

在**C++**程序设计中，多态性是指具有不同功能的函数可以用同一个函数名，这样就可以用一个函数名调用不同内容的函数。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产生不同的行为(即方法)。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，在不同的类中定义了其响应消息的方法，那么使用这些类时，不必考虑它们是什么类型，只要发布消息即可。

从系统实现的角度看，多态性分为两类**:** 静态多态性和动态多态性。以前学过的函数重载和运算符重载实现的多态性属于静态多态性，在程序编译时系统就能决定调用的是哪个函数，因此静态多态性又称编译时的多态性。静态多态性是通过函数的重载实现的**(**运算符重载实质上也是函数重载**)**。动态多态性是在程序运行过程中才动态地确定操作所针对的对象。它又称运行时的多态性。动态多态性是通过虚函数**(virtual function)**实现的。

有关静态多态性的应用已经介绍过了，在本章中主要介绍动态多态性和虚函数。

要研究的问题是：当一个基类被继承为不同的派生类时，各派生类可以使用与基类成员相同的成员名，如果在运行时用同一个成员名调用类对象的成员，会调用哪个对象的成员？也就是说，通过继承而产生了相关的不同的派生类，与基类成员同名的成员在不同的派生类中有不同的含义。也可以说，多态性是“一个接口，多种方法”。

12.2 一个典型的例子

下面是一个承上启下的例子。一方面它是有关继承和运算符重载内容的综合应用的例子，通过这个例子可以进一步融会贯通前面所学的内容，另一方面又是作为讨论多态性的一个基础用例。

例**12.1** 先建立一个**Point**(点)类，包含数据成员**x,y**(坐标点)。以它为基类，派生出一个**Circle**(圆)类，增加数据成员**r**(半径)，再以**Circle**类为直接基类，派生出一个**Cylinder**(圆柱体)类，再增加数据成员**h**(高)。要求编写程序，重载运算符“**<<**”和“**>>**”，使之能用于输出以上类对象。

对于一个比较大的程序，应当分成若干步骤进行。先声明基类，再声明派生类，逐级进行，分步调试。

(1) 声明基类**Point**类

可写出声明基类**Point**的部分如下：

```
#include <iostream>

//声明类Point
class Point
{public:
  Point(float x=0,float y=0);//有默认参数的构造函数
```

```
  void setPoint(float,float);              // 设置坐标值
  float getX( ) const {return x;}          // 读x坐标
  float getY( ) const {return y;}          // 读y坐标
  friend ostream & operator<<(ostream &,const Point &);//重载运算符“<<”
protected:                                 // 受保护成员
  float x,y;
};
//下面定义Point类的成员函数

//Point的构造函数
Point::Point(float a,float b)              // 对x,y初始化
{x=a;y=b;}
//设置x和y的坐标值
void Point::setPoint(float a,float b)      //为x,y赋新值
{x=a;y=b;}
//重载运算符“<<”，使之能输出点的坐标
ostream & operator<<(ostream &output,const Point &p)
{output<<"["<<p.x<<","<<p.y<<"]"<<endl;
 return output;
}
```

以上完成了基类**Point**类的声明。

现在要对上面写的基类声明进行调试，检查它是否有错，为此要写出**main**函数。实际上它是一个测试程序。

```
int main( )
{Point p(3.5,6.4);//建立Point类对象p
 cout<<"x="<<p.getX( )<<",y="<<p.getY( )<<endl;//输出p的坐标值
 p.setPoint(8.5,6.8);                    // 重新设置p的坐标值
 cout<<"p(new):"<<p<<endl;                    // 用重载运算符“<<”输出p点坐标
}
```

程序编译通过，运行结果为

```
x=3.5,y=6.4
p(new):[8.5,6.8]
```

测试程序检查了基类中各函数的功能，以及运算符重载的作用，证明程序是正确的。

(2) 声明派生类Circle

在上面的基础上，再写出声明派生类Circle的部分：

```
class Circle:public Point//circle是Point类的公用派生类
{public:
 Circle(float x=0,float y=0,float r=0); //构造函数
 void setRadius(float);              //设置半径值
 float getRadius( ) const;           //读取半径值
 float area ( ) const;               //计算圆面积
 friend ostream &operator<<(ostream &,const Circle &);//重载运算符“<<”
private:
 float radius;
};

//定义构造函数，对圆心坐标和半径初始化
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }
//设置半径值
void Circle::setRadius(float r)
{radius=r;}
//读取半径值
float Circle::getRadius( ) const {return radius;}
//计算圆面积
float Circle::area( ) const
```

```
{return 3.14159*radius*radius;}
//重载运算符“<<”，使之按规定的形式输出圆的信息
ostream &operator<<(ostream &output,const Circle &c)
{output<<"Center=["<<c.x<<","<<c.y<<"],r="<<c.radius<<",area="<<c.area( )<<endl;
 return output;
}
```

为了测试以上**Circle**类的定义，可以写出下面的主函数：

```
int main( )
{Circle c(3.5,6.4,5.2);//建立Circle类对象c，并给定圆心坐标和半径
 cout<<"original circle:\\nx="<<c.getX()<<", y="<<c.getY()<<", r="<<c.getRadius( )
    <<", area="<<c.area( )<<endl;    //输出圆心坐标、半径和面积
 c.setRadius(7.5);                  // 设置半径值
 c.setPoint(5,5);                  // 设置圆心坐标值x,y
 cout<<"new circle:\\n"<<c;         //用重载运算符“<<”输出圆对象的信息
 Point &pRef=c;                     // pRef是Point类的引用变量，被c初始化
 cout<<"pRef:"<<pRef;               //输出pRef的信息
 return 0;
}程序编译通过，运行结果为original circle:(输出原来的圆的数据)
x=3.5, y=6.4, r=5.2, area=84.9486
new circle:                         ( 输出修改后的圆的数据)
Center=[5,5], r=7.5, area=176.714
pRef:[5,5]                          ( 输出圆的圆心“点”的数据)
```

(3) 声明Circle的派生类Cylinder

前面已从基类Point派生出Circle类，现在再从Circle派生出Cylinder类。

```
class Cylinder:public Circle// Cylinder是Circle的公用派生类
{public:
  Cylinder (float x=0,float y=0,float r=0,float h=0);//构造函数
  void setHeight(float);                    // 设置圆柱高
  float getHeight( ) const;                 // 读取圆柱高
  float area( ) const;                      // 计算圆表面积
  float volume( ) const;                    // 计算圆柱体积
  friend ostream& operator<<(ostream&,const Cylinder&);//重载运算符“<<”
 protected:
  float height;                             // 圆柱高
};
//定义构造函数
Cylinder::Cylinder(float a,float b,float r,float h)
    :Circle(a,b,r),height(h){}
//设置圆柱高
void Cylinder::setHeight(float h){height=h;}
//读取圆柱高
float Cylinder::getHeight( ) const {return height;}
```

```
//计算圆表面积
float Cylinder::area( ) const
{ return 2*Circle::area( )+2*3.14159*radius*height;}
//计算圆柱体积
float Cylinder::volume() const
{return Circle::area()*height;}
//重载运算符“<<”
ostream &operator<<(ostream &output,const Cylinder& cy)
{output<<"Center=["<<cy.x<<","<<cy.y<<"],r="<<cy.radius<<",h="<<cy.height
      <<"\\narea="<<cy.area( )<<", volume="<<cy.volume( )<<endl;
 return output;
}
```

可以写出下面的主函数：

```
int main( )
{Cylinder cy1(3.5,6.4,5.2,10);//定义Cylinder类对象cy1
 cout<<"\\noriginal cylinder:\\nx="<<cy1.getX( )<<", y="<<cy1.getY( )<<", r="
    <<cy1.getRadius( )<<", h="<<cy1.getHeight( )<<"\\narea="<<cy1.area()
    <<",volume="<<cy1.volume()<<endl;//用系统定义的运算符“<<”输出cy1的数据
 cy1.setHeight(15);                  // 设置圆柱高
 cy1.setRadius(7.5);                 // 设置圆半径
 cy1.setPoint(5,5);                  // 设置圆心坐标值x,y
 cout<<"\\nnew cylinder:\\n"<<cy1;       //用重载运算符“<<”输出cy1的数据
 Point &pRef=cy1;                    // pRef是Point类对象的引用变量
```

```
 cout<<"\\npRef as a Point:"<<pRef;     //pRef作为一个“点”输出
 Circle &cRef=cy1;                  // cRef是Circle类对象的引用变量
 cout<<"\\ncRef as a Circle:"<<cRef;    //cRef作为一个“圆”输出
 return 0;
}
```

运行结果如下：

```
original cylinder:                  ( 输出cy1的初始值)
x=3.5, y=6.4, r=5.2, h=10           ( 圆心坐标x,y。半径r，高h)
area=496.623, volume=849.486        ( 圆柱表面积area和体积volume)

new cylinder:                       ( 输出cy1的新值)
Center=[5,5], r=7.5, h=15           ( 以[5,5]形式输出圆心坐标)
area=1060.29, volume=2650.72        ( 圆柱表面积area和体积volume)

pRef as a Point:[5,5]               ( pRef作为一个“点”输出)
cRef as a Circle: Center=[5,5], r=7.5, area=176.714(cRef作为一个“圆”输出)
```

在本例中存在静态多态性，这是运算符重载引起的。可以看到，在编译时编译系统即可以判定应调用哪个重载运算符函数。稍后将在此基础上讨论动态多态性问题。

12.3 虚函数

12.3.1 虚函数的作用

在类的继承层次结构中，在不同的层次中可以出现名字相同、参数个数和类型都相同而功能不同的函数。编译系统按照同名覆盖的原则决定调用的对象。在例**12.1**程序中用**cy1.area( )**调用的是派生类**Cylinder**中的成员函数**area**。如果想调用**cy1**中的直接基类**Circle**的**area**函数，应当表示为**: cy1.Circle::area( )**。用这种方法来区分两个同名的函数。但是这样做很不方便。

人们提出这样的设想，能否用同一个调用形式，既能调用派生类又能调用基类的同名函数。在程序中不是通过不同的对象名去调用不同派生层次中的同名函数，而是通过指针调用它们。例如，用同一个语句**“pt->display( );”**可以调用不同派生层次中的**display**函数，只需在调用前给指针变量**pt**赋以不同的值(使之指向不同的类对象)即可。

C++中的虚函数就是用来解决这个问题的。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，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。

请分析例**12.2**。这个例子开始时没有使用虚函数，然后再讨论使用虚函数的情况。

例12.2 基类与派生类中有同名函数。

在下面的程序中**Student**是基类，**Graduate**是派生类，它们都有**display**这个同名的函数。

```
#include <iostream>
#include <string>
using namespace std;
//声明基类Student
class Student
{public:
  Student(int, string,float);//声明构造函数
  void display( );                          // 声明输出函数
 protected:                          // 受保护成员，派生类可以访问
  int num;
  string name;
  float score;
 };
//Student类成员函数的实现
Student::Student(int n, string nam,float s)          // 定义构造函数
 {num=n;name=nam;score=s;}

void Student::display( )                          // 定义输出函数
{cout<<"num:"<<num<<"\\nname:"<<name<<"\\nscore:"<<score<<"\\n\\n";}
```

```
//声明公用派生类Graduate
class Graduate:public Student
{public:
  Graduate(int, string, float, float);                    // 声明构造函数
  void display( );                                        // 声明输出函数
private:
 float pay;
};
// Graduate类成员函数的实现
void Graduate::display( )                                 // 定义输出函数
 {cout<<"num:"<<num<<"\\nname:"<<name<<"\\nscore:"<<score<<"\\npay="<<pay<<endl;}

Graduate::Graduate(int n, string nam,float s,float p):Student(n,nam,s),pay(p){ }

//主函数
int main()
 {Student stud1(1001,"Li",87.5);                          // 定义Student类对象stud1
  Graduate grad1(2001,"Wang",98.5,563.5);                 //定义Graduate类对象grad1
  Student *pt=&stud1;                                     // 定义指向基类对象的指针变量pt
  pt->display( );
  pt=&grad1;
  pt->display( );
  return 0;
 }
```

运行结果如下，请仔细分析。

num:1001(stud1的数据)
name:Li
score:87.5

num:2001 (grad1 中基类部分的数据)
name:wang
score:98.5

下面对程序作一点修改，在**Student**类中声明**display**函数时，在最左面加一个关键字**virtual**，即

virtual void display( );

这样就把**Student**类的**display**函数声明为虚函数。程序其他部分都不改动。再编译和运行程序，请注意分析运行结果：

num:1001(stud1的数据)
name:Li
score:87.5

num:2001 **(grad1 中基类部分的数据)**
name:wang
score:98.5
pay=1200 **(这一项以前是没有的)**

由虚函数实现的动态多态性就是：同一类族中不同类的对象，对同一函数调用作出不同的响应。虚函数的使用方法是：

(1) 在基类用**virtual**声明成员函数为虚函数。这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。

在类外定义虚函数时，不必再加**virtual**。

(2) 在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。

C++规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加**virtual**，也可以不加，但习惯上一般在每一层声明该函数时都加**virtual**，使程序更加清晰。

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。

(3) 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。

(4) 通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。

通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。这就如同前面说的，不断地告诉出租车司机要去的目的地，然后司机把你送到你要去的地方。

需要说明；有时在基类中定义的非虚函数会在派生类中被重新定义(如例**12.1**中的**area**函数)，如果用基类指针调用该成员函数，则系统会调用对象中基类部分的成员函数；如果用派生类指针调用该成员函数，则系统会调用派生类对象中的成员函数，这并不是多态性行为(使用的是不同类型的指针),没有用到虚函数的功能。

以前介绍的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)。

12.3.2 静态关联与动态关联

编译系统要根据已有的信息，对同名函数的调用作出判断。对于调用同一类族中的虚函数，应当在调用时用一定的方式告诉编译系统，你要调用的是哪个类对象中的函数。这样编译系统在对程序进行编译时，即能确定调用的是哪个类对象中的函数。

确定调用的具体对象的过程称为关联**(binding)**。在这里是指把一个函数名与一个类对象捆绑在一起，建立关联。一般地说，关联指把一个标识符和一个存储地址联系起来。

前面所提到的函数重载和通过对象名调用的虚函数，在编译时即可确定其调用的虚函数属于哪一个类，其过程称为静态关联**(static binding)**，由于是在运行前进行关联的，故又称为早期关联**(early binding)**。函数重载属静态关联。

在上一小节程序中看到了怎样使用虚函数，在调用虚函数时并没有指定对象名，那么系统是怎样确定关联的呢？是通过基类指针与虚函数的结合来实现多态性的。先定义了一个指向基类的指针变量，并使它指向相应的类对象，然后通过这个基类指针去调用虚函数(例如**“pt->display( )”**)。显然，对这样的调用方式，编译系统在编译该行时是无法确定调用哪一个类对象的虚函数的。因为编译只作静态的语法检查，光从语句形式是无法确定调用对象的。

在这样的情况下，编译系统把它放到运行阶段处理，在运行阶段确定关联关系。在运行阶段，基类指针变量先指向了某一个类对象，然后通过此指针变量调用该对象中的函数。此时调用哪一个对象的函数无疑是确定的。例如，先使**pt**指向**grad1**，再执行**“pt->display( )”**，当然是调用**grad1**中的**display**函数。由于是在运行阶段把虚函数和类对象“绑定”在一起的，因此，此过程称为动态关联**(dynamic binding)**。这种多态性是动态的多态性，即运行阶段的多态性。

在运行阶段，指针可以先后指向不同的类对象，从而调用同一类族中不同类的虚函数。由于动态关联是在编译以后的运行阶段进行的，因此也称为滞后关联**(late binding)**。

12.3.3 在什么情况下应当声明虚函数

使用虚函数时，有两点要注意：

(1)只能用**virtual**声明类的成员函数，使它成为虚函数，而不能将类外的普通函数声明为虚函数。因为虚函数的作用是允许在派生类中对基类的虚函数重新定义。显然，它只能用于类的继承层次结构中。

(2) 一个成员函数被声明为虚函数后，在同一类族中的类就不能再定义一个非**virtual**的但与该虚函数具有相同的参数(包括个数和类型)和函数返回值类型的同名函数。

根据什么考虑是否把一个成员函数声明为虚函数呢？主要考虑以下几点：

(1) 首先看成员函数所在的类是否会作为基类。然后看成员函数在类的继承后有无可能被更改功能，如果希望更改其功能的，一般应该将它声明为虚函数。

(2) 如果成员函数在类被继承后功能不需修改，或派生类用不到该函数，则不要把它声明为虚函数。不要仅仅考虑到要作为基类而把类中的所有成员函数都声明为虚函数。

(3) 应考虑对成员函数的调用是通过对象名还是通过基类指针或引用去访问，如果是通过基类指针或引用去访问的，则应当声明为虚函数。

(4) 有时，在定义虚函数时，并不定义其函数体，即函数体是空的。它的作用只是定义了一个虚函数名，具体功能留给派生类去添加。在**12.4**节中将详细讨论此问题。

需要说明的是**:** 使用虚函数，系统要有一定的空间开销。当一个类带有虚函数时，编译系统会为该类构造一个虚函数表**(virtual function table**，简称**vtable)**，它是一个指针数组，存放每个虚函数的入口地址。系统在进行动态关联时的时间开销是很少的，因此，多态性是高效的。

12.3.4 虚析构函数

析构函数的作用是在对象撤销之前做必要的“清理现场”的工作。当派生类的对象从内存中撤销时一般先调用派生类的析构函数，然后再调用基类的析构函数。但是，如果用**new**运算符建立了临时对象，若基类中有析构函数，并且定义了一个指向该基类的指针变量。在程序用带指针参数的**delete**运算符撤销对象时，会发生一个情况**:** 系统会只执行基类的析构函数，而不执行派生类的析构函数。

例12.3 基类中有非虚析构函数时的执行情况。为简化程序，只列出最必要的部分。

```
#include <iostream>
using namespace std;
class Point//定义基类Point类
{public:
  Point( ){ }                                    //Point  类构造函数
  ~Point(){cout<<"executing Point destructor"<<endl;}//Point类析构函数
};

class Circle:public Point                        // 定义派生类Circle类
{public:
  Circle( ){ }                                   //Circle  类构造函数
  ~Circle( ){cout<<"executing Circle destructor"<<endl;}//Circle类析构函数
 private:
  int radius;
};

int main( )
{ Point *p=new Circle;                           // 用new开辟动态存储空间
delete p;                                  // 用delete释放动态存储空间
return 0;
}
```

这只是一个示意的程序。**p**是指向基类的指针变量，指向**new**开辟的动态存储空间，希望用**detele**释放**p**所指向的空间。但运行结果为

executing Point destructor

表示只执行了基类**Point**的析构函数，而没有执行派生类**Circle**的析构函数。原因是以前介绍过的。如果希望能执行派生类**Circle**的析构函数，可以将基类的析构函数声明为虚析构函数，如

```
virtual ~Point(){cout<<"executing Point destructor"<<endl;}
```

程序其他部分不改动，再运行程序，结果为

executing Circle destructor

executing Point destructor

先调用了派生类的析构函数，再调用了基类的析构函数，符合人们的愿望。当基类的析构函数为虚函数时，无论指针指的是同一类族中的哪一个类对象，系统会采用动态关联，调用相应的析构函数，对该对象进行清理工作。

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。

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，则系统会调用相应类的析构函数。

虚析构函数的概念和用法很简单，但它在面向对象程序设计中却是很重要的技巧。专业人员一般都习惯声明虚析构函数，即使基类并不需要析构函数，也显式地定义一个函数体为空的虚析构函数，以保证在撤销动态分配空间时能得到正确的处理。

构造函数不能声明为虚函数。这是因为在执行构造函数时类对象还未完成建立过程，当然谈不上函数与类对象的绑定。

12.4 纯虚函数与抽象类

12.4.1 纯虚函数

有时在基类中将某一成员函数定为虚函数，并不是基类本身的要求，而是考虑到派生类的需要，在基类中预留了一个函数名，具体功能留给派生类根据需要去定义。例如在本章的例**12.1**程序中，基类**Point**中没有求面积的**area**函数，因为“点”是没有面积的，也就是说，基类本身不需要这个函数，所以在例**12.1**程序中的**Point**类中没有定义**area**函数。但是，在其直接派生类**Circle**和间接派生类**Cylinder**中都需要有**area**函数，而且这两个**area**函数的功能不同，一个是求圆面积，一个是求圆柱体表面积。

有的读者自然会想到，在这种情况下应当将**area**声明为虚函数。可以在基类**Point**中加一个**area**函数，并声明为虚函数：

```
virtual float area( ) const {return 0;}
```

其返回值为**0**，表示“点”是没有面积的。其实，在基类中并不使用这个函数，其返回值也是没有意义的。为简化，可以不写出这种无意义的函数体，只给出函数的原型，并在后面加上**“=0”**，如

```
virtual float area( ) const =0;//纯虚函数
```

这就将**area**声明为一个纯虚函数**(pure virtual function)**。纯虚函数是在声明虚函数时被“初始化”为**0**的函数。声明纯虚函数的一般形式是

```
virtual 函数类型 函数名 (参数表列) =0;
```

注意：①纯虚函数没有函数体；②最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是纯虚函数”；③这是一个声明语句，最后应有分号。

纯虚函数只有函数的名字而不具备函数的功能，不能被调用。它只是通知编译系统：“在这里声明一个虚函数，留待派生类中定义”。在派生类中对此函数提供定义后，它才能具备函数的功能，可被调用。

纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。如果在基类中没有保留函数名字，则无法实现多态性。

如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数。

12.4.2 抽象类

如果声明了一个类，一般可以用它定义对象。但是在面向对象程序设计中，往往有一些类，它们不用来生成对象。定义这些类的惟一目的是用它作为基类去建立派生类。它们作为一种基本类型提供给用户，用户在这个基础上根据自己的需要定义出功能各异的派生类。用这些派生类去建立对象。

一个优秀的软件工作者在开发一个大的软件时，决不会从头到尾都由自己编写程序代码，他会充分利用已有资源(例如类库)作为自己工作的基础。

这种不用来定义对象而只作为一种基本类型用作继承的类，称为抽象类**(abstract class)**，由于它常用作基类，通常称为抽象基类**(abstract base class)**。

凡是包含纯虚函数的类都是抽象类。因为纯虚函数是不能被调用的，包含纯虚函数的类是无法建立对象的。抽象类的作用是作为一个类族的共同基类，或者说，为一个类族提供一个公共接口。

一个类层次结构中当然也可不包含任何抽象类，每一层次的类都是实际可用的，可以用来建立对象的。但是，许多好的面向对象的系统，其层次结构的顶部是一个抽象类，甚至顶部有好几层都是抽象类。

如果在抽象类所派生出的新类中对基类的所有纯虚函数进行了定义，那么这些函数就被赋予了功能，可以被调用。这个派生类就不是抽象类，而是可以用来定义对象的具体类**(concrete class)**。如果在派生类中没有对所有纯虚函数进行定义，则此派生类仍然是抽象类，不能用来定义对象。

虽然抽象类不能定义对象(或者说抽象类不能实例化)，但是可以定义指向抽象类数据的指针变量。当派生类成为具体类之后，就可以用这种指针指向派生类对象，然后通过该指针调用虚函数，实现多态性的操作。

12.4.3 应用实例

例**12.4** 虚函数和抽象基类的应用。

在本章例**12.1**介绍了以**Point**为基类的点—圆—圆柱体类的层次结构。现在要对它进行改写，在程序中使用虚函数和抽象基类。类的层次结构的顶层是抽象基类**Shape**(形状)。**Point**(点), **Circle**(圆), **Cylinder**(圆柱体)都是**Shape**类的直接派生类和间接派生类。

下面是一个完整的程序，为了便于阅读，分段插入了一些文字说明。

程序如下：

第(1)部分

```
#include <iostream>
using namespace std;
//声明抽象基类Shape
class Shape
{public:
 virtual float area( ) const {return 0.0;}//虚函数
 virtual float volume() const {return 0.0;}      //虚函数
 virtual void shapeName() const =0;              //纯虚函数
};
```

第(2)部分

```
//声明Point类
class Point:public Shape//Point是Shape的公用派生类
{public:
  Point(float=0,float=0);
  void setPoint(float,float);
  float getX( ) const {return x;}
  float getY( ) const {return y;}
  virtual void shapeName( ) const {cout<<"Point:";}     //对虚函数进行再定义
  friend ostream & operator<<(ostream &,const Point &);
protected:
  float x,y;
```

```
};
//定义Point类成员函数
Point::Point(float a,float b)
{x=a;y=b;}

void Point::setPoint(float a,float b)
{x=a;y=b;}

ostream & operator<<(ostream &output,const Point &p)
{output<<"["<<p.x<<","<<p.y<<"]";
return output;
}
```

第(3)部分

```
//声明Circle类
class Circle:public Point
{public:
  Circle(float x=0,float y=0,float r=0);
  void setRadius(float);
  float getRadius( ) const;
  virtual float area( ) const;
  virtual void shapeName( ) const {cout<<"Circle:";}//对虚函数进行再定义
  friend ostream &operator<<(ostream &,const Circle &);
protected:
```

```
 float radius;
};
//声明Circle类成员函数
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }

void Circle::setRadius(float r):radius(r){ }

float Circle::getRadius( ) const {return radius;}

float Circle::area( ) const {return 3.14159*radius*radius;}

ostream &operator<<(ostream &output,const Circle &c)
{output<<"["<<c.x<<","<<c.y<<"], r="<<c.radius;
 return output;
}
```

第(4)部分

```
//声明Cylinder类
class Cylinder:public Circle
{public:
  Cylinder (float x=0,float y=0,float r=0,float h=0);
  void setHeight(float);
  virtual float area( ) const;
  virtual float volume( ) const;
```

```
  virtual void shapeName( ) const {cout<<"Cylinder:";}//对虚函数进行再定义
  friend ostream& operator<<(ostream&,const Cylinder&);
 protected:
  float height;
};
//定义Cylinder类成员函数
Cylinder::Cylinder(float a,float b,float r,float h)
    :Circle(a,b,r),height(h){ }

void Cylinder::setHeight(float h){height=h;}

float Cylinder::area( ) const
{ return 2*Circle::area( )+2*3.14159*radius*height;}

float Cylinder::volume( ) const
{return Circle::area( )*height;}

ostream &operator<<(ostream &output,const Cylinder& cy)
{output<<"["<<cy.x<<","<<cy.y<<"], r="<<cy.radius<<", h="<<cy.height;
return output;
}
```

第(5)部分

```
//main函数
int main( )
{Point point(3.2,4.5);//建立Point类对象point
 Circle circle(2.4,1.2,5.6);                  // 建立Circle类对象circle
 Cylinder cylinder(3.5,6.4,5.2,10.5);         // 建立Cylinder类对象cylinder
 point.shapeName();                           // 静态关联
 cout<<point<<endl;

 circle.shapeName();                          // 静态关联
 cout<<circle<<endl;

 cylinder.shapeName();                        // 静态关联
 cout<<cylinder<<endl<<endl;

 Shape *pt;                                   // 定义基类指针

 pt=&point;                                   // 指针指向Point类对象
 pt->shapeName( );                            // 动态关联
 cout<<"x="<<point.getX( )<<",y="<<point.getY( )<<"\\narea="<<pt->area( )
    <<"\\nvolume="<<pt->volume()<<"\\n\\n";

 pt=&circle;                                  // 指针指向Circle类对象
```

```
 pt->shapeName( );                    // 动态关联
 cout<<"x="<<circle.getX( )<<",y="<<circle.getY( )<<"\\narea="<<pt->area( )
    <<"\\nvolume="<<pt->volume( )<<"\\n\\n";

 pt=&cylinder;                        // 指针指向Cylinder类对象
 pt->shapeName( );                    // 动态关联
 cout<<"x="<<cylinder.getX( )<<",y="<<cylinder.getY( )<<"\\narea="<<pt->area( )
    <<"\\nvolume="<<pt->volume( )<<"\\n\\n";
 return 0;
}
```

程序运行结果如下。请读者对照程序分析。

```
Point:[3.2,4.5](Point类对象point的数据: 点的坐标)
Circle:[2.4,1.2], r=5.6          (Circle 类对象circle的数据: 圆心和半径)
Cylinder:[3.5,6.4], r=5.5, h=10.5   (Cylinder类对象cylinder的数据：  圆心、半径和高)

Point:x=3.2,y=4.5                ( 输出Point类对象point的数据: 点的坐标)
area=0                          ( 点的面积)
volume=0                         ( 点的体积)

Circle:x=2.4,y=1.2               ( 输出Circle类对象circle的数据: 圆心坐标)
area=98.5203                     ( 圆的面积)
volume=0                        ( 圆的体积)
Cylinder:x=3.5,y=6.4               ( 输出Cylinder类对象cylinder的数据: 圆心坐标)
area=512.595                     ( 圆的面积)
volume=891.96                    ( 圆柱的体积)
```

从本例可以进一步明确以下结论：

(1) 一个基类如果包含一个或一个以上纯虚函数，就是抽象基类。抽象基类不能也不必要定义对象。

(2) 抽象基类与普通基类不同，它一般并不是现实存在的对象的抽象(例如圆形**(Circle)**就是千千万万个实际的圆的抽象)，它可以没有任何物理上的或其他实际意义方面的含义。

(3) 在类的层次结构中，顶层或最上面的几层可以是抽象基类。抽象基类体现了本类族中各类的共性，把各类中共有的成员函数集中在抽象基类中声明。

(4) 抽象基类是本类族的公共接口。或者说，从同一基类派生出的多个类有同一接口。

(5) 区别静态关联和动态关联。

(6) 如果在基类声明了虚函数，则在派生类中凡是与该函数有相同的函数名、函数类型、参数个数和类型的函数，均为虚函数**(**不论在派生类中是否用**virtual**声明**)**。

(7) 使用虚函数提高了程序的可扩充性。

把类的声明与类的使用分离。这对于设计类库的软件开发商来说尤为重要。开发商设计了各种各样的类，但不向用户提供源代码，用户可以不知道类是怎样声明的，但是可以使用这些类来派生出自己的类。

利用虚函数和多态性，程序员的注意力集中在处理普遍性，而让执行环境处理特殊性。

多态性把操作的细节留给类的设计者(他们多为专业人员)去完成，而让程序人员(类的使用者)只需要做一些宏观性的工作，告诉系统做什么，而不必考虑怎么做，极大地简化了应用程序的编码工作，大大减轻了程序员的负担，也降低了学习和使用C++编程的难度，使更多的人能更快地进入C++程序设计的大门。