第六章 多态性与虚函数

written by SCNU第一帅----->孔文康

什么是多态性

多态性是面向对象程序设计的一个重要特征。

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。所谓消息就是调用函数，不同的行为就是指不同的实现，即执行不同的函数。

例如函数的重载、运算符的重载都是多态性。

C++中，多态性表现形式之一是： 具有不同功能的函数可以用同一函数名，这样就可以用一个函数名调用不同内容的函数。

从系统的实现角度来看，多态性分为两类： 静态多态性和动态多态性。

一个典型的例子

先建立一个Point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。

```
class Point {
 public:
 Point (float a, float b):x(a),y(b) {}
 void setPoint (float, float);
 float getX () { return x; }
 float getY () { return y; }
 friend ostream & operator <<(ostream, const Point &);
 protected:
 float x,y;
};
// 设置x，y的坐标值
void Point::setPoint (float a, float b) {
 x = a;
 y = b;
}
// 重载运算符<<,使之能够输出坐标x,y
ostream & operator <<(ostream &output, const Point &p) {
 output<<"["<<p.x<<","<<p.y<<"]"<<endl;
 return output;
}
```

```
class Circle : public Point {
 public:
 Circle (float a, float b, float c) :
 Point(a,b),r(c) {}
 void setRadius(float);
 float getRadius() const { return r; }
 float area() const;
 friend ostream &operator <<(ostream &output, const Circle &);
 protected:
  float r;
 };
 void Circle::setRaduis (float a) {
  r = a;
 }
 float Circle::getRadius const () {
  return r;
 }
 float Circle::area const () {
  return 3.14159 * r * r;
 }
 ostream & operator << (ostream &output, const Circle &c) {
  output<<"Center=["<<c.x<<","<<c.y<<"],r="<<c.r<<",area="<<c.area()<<end
l;
  return output;
 }
```

```
class Cylinder : public Circle {
 public:
 Cylinder (float a, float b, float c, float d) :
 Circle (a,b,c), h(d) {}
 void setHeight (float);
 float getH () const { return h; }
 float volume () const;
 friend ostream & operator <<(ostream &, const Cylinder &);
 protected:
  float h;
 };
 void Cylinder::setHeight (float a) {
  h = a;
 }
```

```
float Cylinder::getH const () {
 return h;
}
// 重载圆表面积
float Cylinder::area () const {
 return Circle::area() + 2 * 3.14159 * r * h;
}
float Cylinder::volume () {
 return Circle::area() * h;
}
ostream &operator << (ostream &output, const Cylinder & c) {
 output<<"Center=["<<c.x<<","<<c.y<<"],r="<<c.r<<",h="<<c.h<<",area="<<
 c.area()<<",volume="<<c.volume()<<endl;
}

```

利用虚函数实现动态多态性

1、虚函数的作用

虚函数的作用就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的同名函数。

```
// 声明基类Student
class Student {
 public:
 Student (int n, string nam, float s) :
 num(n),name(nam),score(s) {}
 void display();
 protected:
 int num;
 string name;
 float score;
}
void Student::display () {
 cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\n\n";
}
```

```
class Graduate : public Student {
 public:
 Graduate (int n, string nam, float s, float w) :
 Student(n,nam,s),wage(w) {}
 void display();
 private:
 float wage;
}
void Graduate::display () {
  cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\nwage="<<wage<<"\n\n";
 }
```

```
int main () {
 Student stu1(1,"li",80.0);
 Graduate grad1(2,"wi", 91.0);
 Student* pt = &stu1; //定义指向基类对象的指针变量pt
 pt->display(); //输入Student基类对象stu1中的数据
 pt = &grad1; //pt指向Graduate类对象grad1
 pt->display(); //希望输出Graduate类对象grad1中的数据
 return 0;
}
```

请注意

此时pt->display()并没有输出grad1中的全部数据，而是grad1基类Student中的数据，所以并没有调用grad1中的display,而是调用了grad1基类中Student的display。

用虚函数可以解决这个问题。

```
virtual void display();
```

这样就把Student类的display函数声明为虚函数。

虚函数会突破这一限制。在基类中的display被声明为虚函数，在声明派生类时被重载，这时派生类的同名函数display就取代其基类中的虚函数。

因此在使基类指针指向派生类对象后，调用display函数时就调用了派生类的display函数。

虚析构函数

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，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。

在程序中用带指针的参数的delete运算符撤销该对象时，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。

```
class Point {
 public:
 Point () {}
 ~Point () {
 cout<<"point ~ functon"<<endl;
 }
}
class Circle : public Point {
 public:
  Circle () {}
  ~Circle () {
  cout<<"~ Circle function"<<endl;
  }
```

```
 private:
 int radius;
}
int main () {
 Point *p = new Circle(); //用new开辟动态内存空间
 delete p; //用delete释放动态内存空间
 return 0;
}
```

请注意

此时运行结果为

```
point ~ functon
```

如果希望执行Circle中的析构函数，可以考虑把析构函数定义为虚函数

```
virtual ~Point () {
 cout<<"point ~ functon"<<endl;
}
```

此时运行结果为

```
~ Circle function
point ~ functon
```

纯虚函数和抽象类

1、纯虚函数

有时候在基类中将某一成员函数定义为虚函数，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。

例如前面例子中，基类Point中没有求面积的area函数，因为点时没有面积的。也就是说，基类本身不需要这个函数，所以在例子中Point没有定义。但是在其派生类Circle和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。

所以这时候需要把area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，可以不要写这种无意义的函数体。

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

请注意

1. 纯虚函数没有函数体
2. 最后面的 = 0并不是函数返回值为0，他只是起到形式上的作用告诉编译系统这个是纯虚函数
3. 这是一个声明的语句，最后应该有;
4. 如果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虚函数。

2、抽象类

有一些类的目的是用他作为基类去建立派生类，他们不用来生成对象，这些类叫做抽象基类。

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，或者说为一个类提供一个公共接口。

3、应用实例

```
class Shape {
 public:
 virtual float area() { return 0.0; }
 virtual float volume() { reutrn 0.0; }
 virtual void shapeName() const = 0;
};
```

```
class Point : public Shape {
 public:
 Point (float a, float b) :
 x(a),y(b);
 void SetPoint(float,float);
 float getX() const { return x; }
 float getY() const { return y; }
 // 对虚函数进行再定义
 virtual void shapeName() const {
  cout<<"Point"<<endl; //
  }
  friend ostream & operator <<(ostream &,const Point &);
}
void Point::SetPoint (float a,float b) {
  x = a;
  y = b;
}
ostream & operator <<(ostream& output, const Point& p) {
  output<<"["<<p.x<<","<<p.y"]"<<endl;
}
```

Point类从Shape类继承了3个成员函数，由于点是没有面积和体积的，因此不必要重新定义area和volume。虽然Point类中用不到这两个函数，但是仍然继承了这两个函数。ShapeName函数再Shape类中是纯虚函数，所以在Point中要进行定义。

```
1
```