

C++大学基础教程

第11章 多态性

北京科技大学
信息基础科学系

◆多态性 (Polymorphism) 是面向对象程序设计的主要特征之一。
多态性对于软件功能的扩展和软件重用都有重要的作用。是学习面向对象程序设计必须要掌握的主要内容之一。

第十一章 多态性

- 11.1 多态性的概念
- 11.2 继承中的静态联编
- 11.3 虚函数和运行时的多态
- 11.4 纯虚函数和抽象类
- 11.5 继承和派生的应用
- 11.6 模板

11.1 多态性的概念

11.1.1 面向对象程序设计中多态的表现

- ◆ 总的来说，**不同对象对于相同的消息有不同的响应**，就是面向对象程序设计中的多态性。
- ◆ 具体在程序中，多态性有两种表现的方式：
 - 同一个对象调用名字相同、但是参数不同的函数，表现出不同的行为。在同一个类中定义的重载函数的调用，属于这种情况。
 - 不同的对象调用名字和参数都相同的函数，表现出不同的行为。在派生类的应用中，经常会看到这样的调用。

11.1.1 面向对象程序设计中多态的表现

◆ 面向对象程序设计中多态性表现为以下几种形式：

- 重载多态：通过调用相同名字的函数，表现出不同的行为。运算符重载也是一种重载多态。
- 运行多态：通过基类的指针，调用不同派生类的同名函数，表现出不同的行为。许多面向对象程序设计的书籍中所说的多态性，就是这种多态。
- 模板多态，也称为参数多态：

11.1.2 多态的实现：联编

◆ 一个具有多态性的程序语句，在执行的时候，必须确定究竟是调用哪一个函数。也就是说，在执行的时候调用哪个函数是唯一地确定的。确定具有多态性的语句究竟调用哪个函数的过程称为**联编**（Binding），有的资料也翻译成“**绑定**”。

11.1.2 多态的实现： 联编

- ◆ 联编有两种方式：**静态联编**和**动态联编**。
- ◆ 在**源程序编译的时候**就能确定具有多态性的语句调用哪个函数，称为静态联编。
- ◆ 对于**重载函数的调用**就是在编译的时候确定具体调用哪个函数，所以是属于静态联编。

11.1.2 多态的实现： 联编

- ◆ 动态联编则是必须在程序运行时，才能够确定具有多态性的语句究竟调用哪个函数。
- ◆ 用动态联编实现的多态，也称为**运行时的多态**。

11.2 继承中的静态联编

11.2.1 派生类对象调用同名函数

- ◆ 在派生类中可以定义和基类中同名的成员函数。这是对基类进行改造，为派生类增加新的行为的一种常用的方法。
- ◆ 通过不同的派生类的对象，调用这些同名的成员函数，实现不同的操作，也是多态性的一种表现。
- ◆ 在程序编译的时候，就可以确定派生类对象具体调用哪个同名的成员函数。这是通过静态联编实现的多态。

- ◆ 例11.1 定义Circle类和Rectangle类为Shape类的派生类，通过Circle类和Rectangle类的对象调用重载函数getArea()显示对象的面积。

```
// 例11.1: shape.h
```

```
#ifndef SHAPE_H
```

```
#define SHAPE_H
```

```
class Shape {
```

```
public:
```

```
    double getArea() const;
```

```
    void print() const;
```

```
};
```

```
// Shape类定义结束
```

```
class Circle : public Shape {
public:
    Circle( int = 0, int = 0, double = 0.0 );
    double getArea() const; // 返回面积
    void print() const; // 输出Circle 类对象
private:
    int x,y; // 圆心坐标
    double radius; // 圆半径
}; // 派生类Circle定义结束
```

```
class Rectangle : public Shape {
public:
    Rectangle( int = 0, int = 0); // 构造函数
    double getArea() const; // 返回面积
    void print() const; // 输出Rectangle类对象
private:
    int a,b; // 矩形的长和宽
}; // 派生类Rectangle定义结束
#endif
```

基类成员函数的定义

```
// 例11.1: shape.cpp
#include <iostream>
using namespace std;
#include "shape.h"
```

```
double Shape::getArea() const
{
    cout<<"基类的getArea函数，面积是 ";
    return 0.0;
} // Shape类getArea函数的定义
```

```
void Shape::print() const
{
    cout<<"Base class Object"<<endl;
} //Shape类print函数定义
```

```
Circle::Circle( int xValue, int yValue, double radiusValue )
```

```
{  
    x=xValue; y=yValue;  
    radius= radiusValue ;  
}
```

Circle类成员函数的定义

// Circle类构造函数

```
double Circle::getArea() const
```

```
{  
    cout<<"Circle类的getArea函数，面积是 ";  
    return 3.14159 * radius * radius;  
}
```

// Circle类getArea函数定义

```
void Circle::print() const
```

```
{  
    cout << "center is ";  
    cout<<"x="<<x<<" y="<<y;  
    cout << "; radius is " << radius<<endl;
```

// Circle类print函数定义

```
}
```

Rectangle类成员函数的定义

```
Rectangle::Rectangle( int aValue, int bValue )
{
    a=aValue; b=bValue;
} // Rectangle类构造函数
double Rectangle::getArea() const
{
    cout<<"Rectangle类的getArea函数，面积是 ";
    return a * b;
} // Rectangle类getArea函数定义
void Rectangle::print() const
{
    cout << "height is "<<a;
    cout<<"width is"<<b<<endl;
} // Rectangle类print函数定义
```

例11.1的主函数

```
// 例11.1: 11_1.cpp
#include <iostream>
using std::cout;
using std::endl;
```

调用的是Circle类的getarea函数，面积是38.4845
调用的是Rectangle类的getarea函数，面积是100

```
#include "shape.h"
void main()
{
```

// 包含头文件

```
    Circle circle( 22, 8, 3.5 ); // 创建Circle类对象
    Rectangle rectangle( 10, 10 ); // 创建Rectangle类对象
    cout << "调用的是 ";
    cout<<circle.getArea() << endl; // 静态联编
    cout << "调用的是";
    cout<<rectangle.getArea() << endl; // 静态联编
```

```
}
```

11.2.1 派生类对象调用同名函数

- ◆ 对于派生类对象调用成员函数，可以有以下的结论：
 - 派生类对象可以直接调用本类中与基类成员函数同名的函数，不存在二义性；
 - 在编译时就能确定对象将调用哪个函数，属于静态联编，不属于运行时的多态。

11.2.2通过基类指针调用同名函数

- ◆ 从继承的角度来看，派生类对象是基类对象的一个具体的特例。或者说，派生类对象是某一种特定类型的基类对象。
 - 例如，Circle类是Shape类的公有继承，“圆”是“图形”的一种特例。或者说，圆是一种特定的图形，具有图形的基本特征。
- ◆ 但是，这种关系不是可逆的。不可说基类的对象具有派生类对象的特征，基类对象也不是派生类对象的一个特例。

11.2.2 通过基类指针调用同名函数

◆ 在关于**基类对象**和**派生类对象**的操作上，可以允许以下的操作：

- 派生类对象可以赋值给基类对象；
- 派生类对象的地址可以赋值给基类对象的指针。或者说，可以用派生类对象的地址初始化基类对象的指针；
- 可以将基类对象的引用，定义为派生类对象的别名，或者说，用派生类对象初始化基类的引用。
- 通过派生类对象的地址初始化的**基类对象的指针**，可以**访问基类的公有成员**，也可以访问和基类成员函数同名的函数。

11.2.2通过基类指针调用同名函数

◆ 以下这些操作是不可以进行的：

- 不可以将基类对象赋值给派生类对象；
- 不可以用基类对象的地址初始化派生类对象的指针；
- 不可以将派生类对象的引用定义为基类对象的别名；
- 不可以通过用派生类对象初始化的**基类对象的指针**，访问**派生类新增加的**和**基类公有成员不重名的公有成员**。

- ◆ 例11.2 在例11.1所定义的类的基础上，观察通过派生类对象地址初始化的基类对象的指针访问getArea函数的结果。

```
#include <iostream>
using namespace std;
#include "shape.h"
void main()
{ Shape *shape_ptr;
  Circle circle( 22, 8, 3.5 );
  Rectangle rectangle( 10, 10 );
  shape_ptr = &circle;
  cout<<shape_ptr->getArea() << endl; // 静态联编
  shape_ptr = &rectangle;
  cout<<shape_ptr->getArea() << endl; // 静态联编
}
```

circle 对象初始化shape_ptr指针访问的getArea函数是基类的getArea函数，面积是 0
rectangle 对象初始化shape_ptr指针访问的getArea函数是基类的getArea函数，面积是 0

11.2.2通过基类指针调用同名函数

◆ 程序运行结果表明：

- 确实可以用派生类对象的地址初始化基类对象的指针；
- 通过用派生类对象地址初始化的基类对象指针，只能调用基类的公有成员函数。在以上例子中，就是调用基类的getArea函数，而不是派生类的getArea函数。
- 这种调用关系的确定，也是在编译的过程中完成的，属于静态联编，而不属于运行时的多态。

11.3 虚函数和运行时的多态

11.3 虚函数和运行时的多态

- ◆ 通过指向基类的指针访问基类和派生类的同名函数，是实现运行时的多态的必要条件，但不是全部条件。
- ◆ 除此以外，还必须将基类中的同名函数定义为**虚函数**。

11.3.1 虚函数

- ◆ 虚函数可以在类的定义中声明函数原型的时候来说明，格式如下：

`virtual` <返回值类型> 函数名(参数表);

- 在函数原型中声明函数是虚函数后，具体定义这个函数时就不需要再说明它是虚函数了。

- ◆ 如果在基类中直接定义同名函数，定义虚函数的格式是：

`virtual` <返回值类型> 函数名(参数表)
{<函数体>}

11.3.1 虚函数

- ◆ 基类中的同名函数声明或定义为虚函数后，派生类的同名函数无论是不是用virtual来说明，都将自动地成为虚函数。从程序可读性考虑，一般都会在这些函数的声明或定义时，用virtual来加以说明。
- ◆ 只要对例11.2中的头文件稍加修改，也就是将基类和派生类中的getArea函数都声明为虚函数，再重新编译和运行程序，就可以得到运行时的多态的效果。

- ◆ 例11.3 将例11.2进行修改，使得程序具有运行时的多态的效果。

```
// 例11.3: shape1.h
#ifndef SHAPE_H
#define SHAPE_H
class Shape {
public:
    virtual double getArea() const;
    void print() const;
};
// Shape类定义结束
```

```

class Circle : public Shape {
public:
    Circle( int = 0, int = 0, double = 0.0 );
    virtual double getArea() const;    // 返回面积
    void print() const;              // 输出Circle 类对象t
private:
    int x,y;                          // 圆心坐标
    double radius;                    // 圆半径
};                                    // 派生类Circle定义结束

class Rectangle : public Shape {
public:
    Rectangle( int = 0, int = 0);    // 构造函数
    virtual double getArea() const; // 返回面积
    void print() const;             // 输出Rectangle类对象
private:
    int a,b;                          // 矩形的长和宽
};                                    // 派生类Rectangle定义结束

#endif

```

◆ 例11.2 在例11.1所定义的类的基础上，观察通过派生类对象地址初始化的基类对象的指针访问getArea函数的结果。

```
#include <iostream>
using namespace std;
#include "shape1.h"
void main()
{ Shape *shape_ptr;
  Circle circle( 22, 8, 3.5 );
  Rectangle rectangle( 10, 10 );
  shape_ptr = &circle;
  cout<<shape_ptr->getArea() << endl; //动态联编
  shape_ptr = &rectangle;
  cout<<shape_ptr->getArea() << endl; //动态联编
}
```

circle 对象初始化 shape_ptr 指针访问的
getArea函数是
Circle类的getArea函数，面积是 38.4845
rectangle 对象初始化 shape_ptr 指针访问的
getArea函数是
Rectangle类的getArea函数，面积是 100

11.3.1 虚函数

- ◆ 这个结果和例11.2的结果大不相同。同样的 `shape_ptr->getArea()` 函数调用，当 `shape_ptr` 指针中是 `Circle` 类对象地址时，访问的是 `Circle` 类的 `getArea` 函数。而 `shape_ptr` 指针中是 `Rectangle` 类对象的地址时，访问的是 `Rectangle` 类的 `getArea` 函数。
- ◆ 这种方式的函数调用，在编译的时候是不能确定具体调用哪个函数的。只有程序运行后，才能知道指针 `shape_ptr` 中存放的是什么对象的地址，然后再决定调用哪个派生类的函数。是一种运行时决定的多态性。

11.3.1 虚函数

- ◆ 要实现运行时的多态，需要以下条件：
 - 必须通过指向基类对象的指针访问和基类成员函数同名的派生类成员函数；
 - 或者用派生类对象初始化的基类对象的引用访问和基类成员函数同名的派生类成员函数；
 - 派生类的继承方式必须是**公有继承**；
 - 基类中的同名成员函数必须定义为虚函数。

11.3.2 虚函数的使用

- ◆ 虚函数必须正确的定义和使用。否则，即使在函数原型前加了virtual的说明，也可能得不到运行时多态的特性。
 - 必须首先在基类中声明虚函数。在多级继承的情况下，也可以不在最高层的基类中声明虚函数。例如在第二层定义的虚函数，可以和第三层的虚函数形成动态联编。但是，一般都是在最高层的基类中首先声明虚函数。

11.3.2 虚函数的使用

- 基类和派生类的同名函数，必须函数名、返回值、参数表全部相同，才能作为虚函数来使用。否则，即使函数用virtual来说明，也不具有虚函数的行为。
- 静态成员函数不可以声明为虚函数。构造函数也不可以声明为虚函数。
- 析构函数可以声明为虚函数，即可以定义虚析构函数。

- ◆ 例11.4 虚函数的正确使用。分析以下程序，编译时哪个语句会出现错误？为什么？将有错误的语句屏蔽掉以后，程序运行结果如何？其中哪些调用是静态联编，哪些是动态联编？

```
#include <iostream.h>
class BB
{public:
virtual void vf1(){cout<<"BB::vf1被调用\n";}
virtual void vf2(){cout<<"BB::vf2被调用\n";}
void f(){cout<<"BB::f被调用\n";}
};
class DD:public BB
{public:
virtual void vf1(){cout<<"DD::vf1被调用\n";}
void vf2(int i){cout<<i<<endl;}
void f(){cout<<"DD::f\n被调用";}
};
```

```
void main()
{ DD d;
  BB *bp=&d;
  bp->vf1();
  bp->vf2();
  bp->vf2(10);
  bp->f();
}
```

函数调用bp->vf2(10);是错误的。因为派生类的vf2函数和基类的vf2函数的参数不同，派生类的vf2就不是虚函数。

其中bp->vf1()调用将动态联编注释掉后，运行bp->vf2()是静态联编结果将显示：
bp->f()也是静态联编
DD::vf1被调用
BB::vf2被调用
BB::f被调用

11.3.3 虚析构函数

- ◆ 如果用**动态创建**的派生类对象的地址初始化基类的指针，创建的过程不会有问题：仍然是先调用基类构造函数，再执行派生类构造函数。
- ◆ 但是，在用**delete**运算符删除这个指针的时候，由于指针是指向基类的，通过静态联编，只会调用基类的析构函数，释放基类成员所占用的空间。而**派生类成员所占用的空间将不会被释放**。

```

#include <iostream>
using namespace std;
class Shape {
public:
    Shape(){cout<<"Shape类构造函数被调用\n";}
    ~Shape(){cout<<"Shape类析构函数被调用\n";}
};
class Circle : public Shape {
public:
    Circle( int xx= 0, int yy= 0, double rr= 0.0 )
    {x = xx; y = yy; radius =rr;
    cout<<"Circle类构造函数被调用\n"; }
    ~Circle() {cout<<"Circle类析构函数被调用\n"; }
private:
    int x,y;
    double radius;
};
void main()
{Shape *shape_ptr;
shape_ptr = new Circle(3,4,5);
delete shape_ptr;
}

```

例11.5 定义简单的Shape类和Circle类，观察基类指针的创建和释放时如何调用构造函数和析构函数。

程序运行后在屏幕上显示：
Shape类构造函数被调用
Circle类构造函数被调用
Shape类析构函数被调用

11.3.3 虚析构函数

- ◆ 为了解决派生类对象释放不彻底的问题，必须将基类的析构函数定义为虚析构函数。格式是在析构函数的名字前添加virtual关键字。函数原型如下：

```
virtual ~Shape();
```

- ◆ 此时，无论派生类析构函数是不是用virtual来说明，也都是虚析构函数。
- ◆ 再用delete shape_ptr来释放基类指针时，就会通过动态联编调用派生类的析构函数。

11.3.3 虚析构函数

◆ 将例11.5程序中的~Shape析构函数作以上修改后，运行的结果将是：

Shape类构造函数被调用

Circle类构造函数被调用

Circle类析构函数被调用

Shape类析构函数被调用

◆ 11.4 纯虚函数和抽象类

11.4 纯虚函数和抽象类

- ◆ 在前面的几个例子中，基类Shape本身并不是一个具体的“形状”的抽象，而是各种实际的“形状”的抽象。
- ◆ 在C++中，对于那些在基类中不需要定义具体的行为的函数，可以定义为**纯虚函数**。
- ◆ 对于那些只是反映一类事物公共特性的类，在C++中可以定义为“**抽象类**”。

11.4 纯虚函数和抽象类

◆ 纯虚函数声明的格式是：

```
virtual <返回值类型> 函数名(参数表) = 0;
```

◆ 纯虚函数的声明和使用有以下的特点：

- 纯虚函数一定是在基类中声明的。
- 在多级继承的情况下，纯虚函数除了在最高层基类中声明外，也可以在较低层的基类中声明。
- 纯虚函数是没有函数体的。函数体是用“= 0”来代替了。
- 纯虚函数是不可以被调用的。凡是需要被调用的函数都不可以声明为纯虚函数。

11.4 纯虚函数和抽象类

- ◆ 抽象类的定义是基于纯虚函数的。
- ◆ 凡是带有一个或几个纯虚函数的类，就是抽象类。
- ◆ 抽象类定义的一般形式是：

```
class 类名
{public:
    virtual <返回值类型> 函数名(参数表) = 0;
    //其他函数的声明;
    //.....
};
```

11.4 纯虚函数和抽象类

◆ 抽象类的定义和使用具有以下的特点：

- 抽象类是不可以实例化的，也就是不可以定义抽象类的对象。
- 但是，可以定义抽象类的指针和抽象类的引用。目的是通过这些指针或引用访问派生类的虚函数，实现运行时的多态。
- 如果抽象类的派生类中没有具体实现纯虚函数的功能，这样的派生类仍然是抽象类。
- 抽象类中除了纯虚函数外，还可以定义其他的非纯虚函数。

11.4 纯虚函数和抽象类

- ◆ 虚函数、纯虚函数、多态性在面向对象程序设计中有很大的作用。可以增强程序的通用性、可扩展性和灵活性。

```
// 例11.6: shape2.h
#ifndef SHAPE_H
#define SHAPE_H
```

```
class Shape {                                //基类Shape的定义
public:
    virtual double getArea() const=0; //纯虚函数
    void print() const;
    virtual ~Shape(){}                    //虚析构函数
};                                          // Shape类定义结束
```

11.4 纯虚函数和抽象类

- ◆ 这个例子还显示了：抽象类中可以为各派生类定义一些通用的接口。这些通用的接口就是抽象类中的纯虚函数。新增加的派生类的对象，都可以使用这样的通用接口，表现派生类对象的行为特性。

总结

- ◆ 多态性的概念
- ◆ 继承中的静态联遍
 - 派生类对象调用同名函数
 - 通过基类指针或引用调用同名函数
- ◆ 虚函数和运行时的多态
- ◆ 纯虚函数和抽象类

作业及实验

- ◆ 第11章习题： 2
- ◆ 实验： 例11.1~5， 2， 3