第七讲 多态

多态的概念

多态（polymorphism）是指同样的消息被不同类型的对象接收时，会表现出不同的行为。实际使用中，即指调用相同函数名时，不同对象的函数体不同，因此实现的结果也不同。多态也是面向对象程序设计的重要特性之一。

多态在实现时又可以分为静态联编和动态联编两种：

➢ 静态联编指的是在编译、链接的过程中能够确定操作对象的多态；

➢ 动态联编指的是在编译、链接过程中无法确定操作对象，在运行时才能确定的多态

运算符重载

重载机制

运算符重载其实就是函数重载，实际上是通过重载运算符函数来实现的，运算符函数的名称与普通函数的格式不同，是一种特殊的函数，定义如下：

```
<返回类型说明符> operator <运算符符号> ( <参数表> )
{ < 函数体> }
```

例： 对于 14+47 和 48.5+3.7，代码实现过程中会先将指定的运算表达式转换为对运算符函数的调用，然后将运算的对象转换为运算符函数的实参，即会把两个表达式分别编译为：

```
operator +(14,47);
operator +(48.5,3.7);
```

两个表达式的实参类型不同，编译器会再根据参数类型和返回值类型确定应该调用哪个函数原型进行计算，由于运算符重载是在编译阶段就可以确定应该调用哪个函数，所以运算符重载属于静态联编。

重载规则

除了类属关系运算符“.”、成员指针运算符“.*”、作用域运算符“: :”、sizeof 运算符和三目运算符“?:”以外，C++中的所有运算符都可以重载。

可以重载的运算符：

```
算术运算符：+，-，*，/，%，++，--
位操作运算符：&，|，~，^，<<，>>
逻辑运算符：!，&&，||
比较运算符：<，>，<=，>=，= =，!=
赋值运算符：=，+=，=，*=，/=，%=，&=，|=，^=，<<=，>>=
其他运算符：[ ]，( )，->，,（逗号运算符），new，delete，new[ ]，delete[ ]，->*
```

- 重载运算符限制在 C++语言中已有的运算符范围内的允许重载的运算符之中，不能创建新的运算符
- 重载的运算符只能是用户自定义的类型，只能和用户自定义类型的对象一起使用
- 运算符重载是针对新类型数据的实际需要对原有运算符进行的适当的改造，重载的功能应当与原有功能相类似，不能改变该运算符用于内部类型对象的含义。
- 对自定义来说，运算符“=”可以不重载，当不重载=运算符时，编译器会生成一个缺省的赋值运算符函数，作用是通过位拷贝的方式把源对象的结果复制到目的对象中。赋值运算函数与拷贝运算符的相同之处：在于他们都是将一个对象的成员复制到另一个对象中；不同之处在于：拷贝构造函数需要构造一个新的对象，赋值运算函数函数是要改变一个已经存在的对象。

重载为类的成员函数

将运算符函数重载为类的成员函数，这样运算函数可以自由地访问本类的数据成员。重载运算符函数为类的成员函数语法形式为：

```
返回类型 类名:: operator 运算符 （ 形参表 ）
{
    函数体;
}
```

➢ 类名是要重载该运算符的类，如果在类中定义运算符函数，类名与作用域分辨符可以省略

➢ operator 与运算符构成运算符函数名

```
#include <iostream>
using namespace std;
//复数类
class Complex
{
private:
    double re; //复数的实部
    double im; //复数的虚部
public:
    Complex(double real = 0.0, double image = 0.0) //构造函数
    {
        re = real;
        im = image;
    }
    void Disp() //显示函数
    {
        cout << " ( " << this->re << " , " << this->im << " ) " <<
endl;
    }
    Complex operator +(Complex T) // +重载为成员函数
    {
        return Complex(re + T.re, im + T.im);
    };
    Complex operator -(Complex T) // - 重载为成员函数
    {
        return Complex(re - T.re, im - T.im);
    }
    Complex operator ++() // 前置++重载为成员函数
    {
        return Complex(++re, im);
    }
    Complex operator ++(int) // 后置++重载为成员函数
    {
        return Complex(re++, im);
    }
};
int main()
{
    Complex A(10.0, 100.0), B(20.0, 200.0), C; //定义三个复数，C取默认
值
    cout << " A = ";
```

```
    A.Disp();
    cout << " B = ";
    B.Disp();
    //两复数相加
    C = A + B;
    cout << " C = A + B = ";
    C.Disp();
    //两复数相减
    C = B - A;
    cout << " C = B -A = ";
    C.Disp();
    //复数加整数
    C = A + 9;
    cout << " C = A + 9 = ";
    C.Disp();
    //复数后置++
    C = A++;
    cout << " C = A ++ , C = ";
    C.Disp();
    //复数前置++
    C = ++A;
    cout << " C = ++ A , C = ";
    C.Disp();
    return 0;
}

// 结果:
// A =  ( 10 , 100 )
// B =  ( 20 , 200 )
// C = A + B =  ( 30 , 300 )
// C = B -A =  ( 10 , 100 )
// C = A + 9 =  ( 19 , 100 )
// C = A ++ , C =  ( 10 , 100 )
// C = ++ A , C =  ( 12 , 100 )
```

本段代码中的 `Complex operator ++(int)` 非常需要记住，括号内带个 int 的重载是后置自加运算符。

重载为类的友元函数

运算符之所以要重载为类的友元函数，是因为这样可以自由地访问该类的任何数据成员。将运算符重载为类的友元函数要在类中使用 friend 关键字来声明该运算符函数为友元函数。在类中定义友元函数的格式如下：

```
friend 型 函数类型 operator 运算符 （ 形参表 ）
{
    函数体;
}
```

输出运算符<< 的重载

C++中的输出通常使用 iostream.h 中定义的流输出，格式为“cout << 待输出变量”。如果想用流输出的代码格式输出自定义类型的数据，就需要在自定义类中重载这个输出运算符。重载输出运算符<<的函数格式为：

```
ostream &operator << (ostream &os, 自定义类名 &对象名)
{
    ..... //自定义输出格式
    return os ;
}
```

例子：

```
#include <iostream>
using namespace std;
//复数类
class Complex
{
private:
    double re; //复数的实部
    double im; //复数的虚部
public:
    Complex(double real = 0.0, double image = 0.0) //构造函数
    {
        re = real;
        im = image;
    }
    friend ostream &operator<<(ostream &os, const Complex &ob);
```

```
};
ostream &operator<<(ostream &os, const Complex &ob)
{
    os << " ( " << ob.re << " + " << ob.im << " i) " << endl;
    return os;
}
int main()
{
    Complex obj1(10.0, 100.0);
    Complex obj2(20.0, 200.0);
    cout << obj1 << endl
         << obj2 << endl;
    return 0;
}

// 结果:
// ( 10 + 100 i)

// ( 20 + 200 i)
```

= 运算符的重载

与拷贝构造函数一节相同，当函数中含有指针成员变量的时候，使用赋值运算符“=”就会使得程序中的两对象指向同一内存地址。这是我们不希望发生的事情。

在这种情况下，需要使用重载=运算符来规避此类情况的发生。

错误❌示范:

```
#include <iostream>
#include <cstring>
using namespace std;
//类A
class A
{
private:
    int nlen;
    char *pbuf;

public:
    //构造函数
    A(int n)
```

```
    {
        nlen = n;
        pbuf = new char[n]; //开辟数组空间赋给指针，
        //指针指向一个字符数组
        cout << "A( )开辟空间" << nlen << endl;
    }
    A(A &obj)
    {
        nlen = obj.nlen;
        pbuf = new char[nlen];
        strcpy(pbuf, obj.pbuf);
        cout << "A(A& obj)开辟空间" << nlen << endl;
    }
    ~A()
    {
        delete[] pbuf; //清理开辟的空间
        cout << "~A( )释放空间" << endl;
    }
};
void func()
{
    A obj1(16), obj2(8); //定义两个对象obj1，obj2
    A obj3 = obj1;       //调用拷贝构造函数用obj1为obj3赋值
    obj2 = obj3;         //用默认赋值运算符函数用obj3为obj2赋值
}
int main()
{
    func(); //调用func函数
    return 0;
}

// 结果:
// A( )开辟空间16
// A( )开辟空间8
// A(A& obj)开辟空间16
// ~A( )释放空间
// free(): double free detected in tcache 2  （报错啦啊啊啊啊啊啊啊）
// Aborted
```

修正错误只需要添加运算符重载代码段：

```
A operator=(A &sr)
    {
        if (this != &sr)
        {
            //释放当前自己所指向的空间
            char *tmp = pbuf;
            delete[] tmp;
            pbuf = NULL;
            cout << " = 释放旧空间" << endl;
            //重新开辟空间
            nlen = sr.nlen; //成员变量赋值
            pbuf = new char[nlen];
            cout << " = 开辟新空间" << nlen << endl;
        }
        return *this;
    }
// 添加后结果:
// A( )开辟空间16
// A( )开辟空间8
// A(A& obj)开辟空间16
//  = 释放旧空间
//  = 开辟新空间16
// A(A& obj)开辟空间16
// ~A( )释放空间
// ~A( )释放空间
// ~A( )释放空间
// ~A( )释放空间
```

虚函数

静态联编与动态联编

联编（binding）又称为绑定，绑定的是调用对象与调用的函数体。

静态多态性的演示示例

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

```
private:
    int na;

public:
    A(int na = 0)
    {
        this->na = na;
    }
    void Show()
    {
        cout << "0" << endl;
    }
};
class B : public A
{
private:
    int nb;

public:
    B(int na, int nb) : A(na)
    {
        this->nb = nb;
    }
    void Show()
    {
        cout << nb << endl;
    }
};
int main()
{
    A a(10);
    cout << " a. Show ( ) = ";
    a.Show();
    cout << endl;
    B b(10, 200);
    cout << " b. Show ( ) = ";
    b.Show();
    cout << endl;
    A *pa;
    pa = &b;
    cout << " pa -> Show ( ) = ";
    pa->Show();
```

```
    cout << endl;
    A &ra = b;
    cout << " ra. Show ( ) = ";
    ra.Show();
    cout << endl;
    return 0;
}

// 结果:
//  a. Show ( ) = 0

//  b. Show ( ) = 200

//  pa -> Show ( ) = 0

//  ra. Show ( ) = 0
```

- 静态联编的特点：静态联编只根据指针和引用的类型去确定调用对象，而不管实际情况中指针和引用到底指向了哪个对象。
- 静态联编的这种特点对采用类型兼容、同名覆盖等方法编写的程序达不到预期的运行效果。

虚函数的定义与使用

虚函数定义的一般语法形式如下：

```
virtual 型 函数类型 函数表 （ 形参表 ）
{
    函数体;
}
```

- 虚函数不能是静态成员函数，也不能是友元函数。因为静态成员函数和友元函数不属于某个对象
- 只有类的成员函数才能说明为虚函数，虚函数的声明只能出现在类的定义中。因为虚函数仅适用于有继承关系的类对象，普通函数不能说明为虚函数。
- 构造函数不能是虚函数，析构函数可以是虚函数，而且通常声明为虚函数

```
#include <iostream>
using namespace std;
class A
{
private:
    int na;

public:
    A(int na = 0)
    {
        this->na = na;
    }
    virtual void Show() //基类成员函数改为虚函数
    {
        cout << "0" << endl;
    }
};
class B : public A
{
private:
    int nb;

public:
    B(int na, int nb) : A(na)
    {
        this->nb = nb;
    }
    void Show() //对虚函数的再定义
    {
        cout << nb << endl;
    }
};
int main()
{
    A a(10);
    cout << " a. Show ( ) = ";
    a.Show();
    cout << endl;
    B b(10, 200);
    cout << " b. Show ( ) = ";
    b.Show();
```

```
    cout << endl;
    A *pa;
    pa = &b;
    cout << " pa -> Show ( ) = ";
    pa->Show();
    cout << endl;
    A &ra = b;
    cout << " ra. Show ( ) = ";
    ra.Show();
    cout << endl;
}

// 结果:
//  a. Show ( ) = 0

//  b. Show ( ) = 200

//  pa -> Show ( ) = 200

//  ra. Show ( ) = 200
```

虚析构函数

虚析构函数定义形式如下:

`virtual ~类名();`

➢ 当基类的析构函数被声明为虚函数，则派生类的析构函数，无论是否使用 virtual 关键字进行声明，都自动成为虚函数。

➢ 析构函数声明为虚函数后，程序运行时采用动态联编，因此可以确保使用基类类型的指针就能够自动调用适当的析构函数对不同对象进行清理工作。

➢ 当使用 delete 运算符删除一个对象时，隐含着对析构函数的一次调用，如果析构函数为虚函数，则这个调用采用动态联编，保证析构函数被正确执行。

纯虚函数与抽象类

抽象类是一种特殊的类，是为了抽象的目的而建立的，建立抽象类，就是为了通过它多态地使用其中的成员函数，为一个类族提供统一的操作界面。抽象类处于类层次的上层，一个抽象类自身无法实例化，也就是说我们无法声明一个抽象类的对象，而只能通过继承机制，生成抽象类的非抽象派生类，然后再实例化。

既然抽象类所描述的共性无法具体化，在其派生之后才有具体的内容，那么在抽象类中也就没有必要去具体描述它的行为，但上节虚函数的特性告诉我们如果是公有的特性，应该用虚函数的形式将这个接口保留下来，抽象类中这种以保留接口为目的无需实现的函数就称为纯虚函数。带有纯虚函数的类即可称为一个抽象类，即一个抽象类至少有一个纯虚函数。

纯虚函数与抽象类

纯虚函数(pure virtual function)是一个在基类中说明的虚函数，它在该基类中没有定义具体实现，要求各派生类根据实际需要定义函数实现。纯虚函数的作用是为派生类提供一个一致的接口。

`virtual 函数类型 函数名( 参数表)=0;`

实际上，它与一般虚函数成员的原型在书写格式上的不同就在于后面加了=0。

C++中，有一种函数体为空的空虚函数，它与纯虚函数的区别为：

① 纯虚函数根本就没有函数体，而空虚函数的函数体为空。

② 纯虚函数所在的类是抽象类，不能直接进行实例化，空虚函数所在的类是可以实例化的。

③ 它们共同的特点是都可以派生出新的类，然后在新类中给出新的虚函数的实现，而且这种新的实现可以具有多态特征。

抽象类

(1) 抽象类只能作为其它类的基类使用，抽象类不能定义对象，纯虚函数的实现由派生类给出；

(2) 派生类仍可不给出所有基类中纯虚函数的定义，继续作为抽象类；如果派生类给出所有纯虚函数的实现，派生类就不再是抽象类而是一个具体类，就可以定义对象。

(3) 抽象类不能用作参数类型、函数返回值或强制类型转换。

(4) 可以定义一个抽象类的指针和引用。通过抽象类的指针和引用，可以指向并访问各派生类成员，这种访问是具有多态特征的。

例子：

```
#include <iostream>
using namespace std;
const double PI = 3.14;
//抽象类
class Shape
{
public:
```

```
    virtual void Show() = 0;  //显示函数，派生类共有的接口定义为纯虚函数
    virtual double Area() = 0; //返回图形面积值
};
//点类
class Point : public Shape
{
protected:
    double X, Y; //横纵坐标值
public:
    Point(double x = 0.0, double y = 0.0)
    {
        X = x;
        Y = y;
    }
    void Show()
    {
        cout << " ( " << X << " , " << Y << " ) " << endl;
    }
    double Area()
    {
        return 0.0;
    }
};
//圆
class Circle : public Point
{
protected:
    double R;

public:
    Circle(double x, double y, double r) : Point(x, y)
    {
        R = r;
    }
    void Show()
    {
        cout << " 原点: ";
        Point ::Show();
        cout << " 半径: " << R << endl;
    }
    double Area()
    {
```

```
        return PI * R * R;
    }
};
//圆柱
class Cylinder : public Circle
{
protected:
    double H;

public:
    Cylinder(double x, double y, double r, double h) : Circle(x, y,
r)
    {
        H = h;
    }
    void Show()
    {
        Circle ::Show();
        cout << " 高度: " << H << endl;
    }
    double Area()
    {
        return 2 * Circle ::Area() + 2 * PI * R * H;
    }
};
int main()
{
    Circle CR(1.0, 1.0, 10.0);
    Shape *pS;  //抽象类指针
    pS = &CR;   //抽象类指针指向对象
    pS->Show(); //抽象类指针调用函数
    Cylinder CY(0.0, 0.0, 10.0, 55.0);
    Shape &rS = CY; //抽象类引用指向派生类对象
    rS.Show();      //抽象类引用调用函数
    cout << " 圆面积: " << pS->Area() << ", 圆柱面积: " << rS.Area()
<< endl;
    return 0;
}

// 结果:
//  原点:  ( 1 , 1 )
//  半径: 10
```

```
//  原点： ( 0 , 0 )
//  半径：10
//  高度：55
//  圆面积：314，圆柱面积：4082
```