CHAPTER 3

多 态 性 第3章

多态性是面向对象程序设计的重要特性之一。简单地说，多态性就是“一个接口，多种实现”，就是同一种事物表现出的多种形态。

3.1 多态概述

3.1.1 多态性的基本概念

1. 什么是多态性

所谓多态性，是指同一个接口可以通过多种方法调用。通俗地说，多态性是指用一个相同的名字定义不同的函数，这些函数的执行过程不同，但是有相似的操作，即用同样的接口访问不同的函数。在实际的系统设计阶段，当设计人员决定把某一类型的活动用于一个给定的对象时，并不关心这个对象如何解释这个活动以及这个方法如何实现，而只关心这个活动对这个对象所产生的作用。

2. 多态性的作用

利用多态性，程序员可以向一个对象发送消息来完成一系列操作，而不用关心软件系统是如何实现这些操作的。因此，多态性的作用是：

(1) 精简代码。

(2) 提高程序的可扩充性和可维护性。

3. 多态的分类

从实现的角度来讲，面向对象的多态性可以分为静态多态和动态多态两种。静态多态是在编译的过程中确定了同名操作的具体操作对象；而动态多态则是在程序运行过程中才动态地确定操作所针对的具体对象。这种确定操作具体对象的过程就是联编，也称为绑定(binding)。

3.1.2 联编与多态的实现方式

联编是指计算机程序自身彼此关联的过程，也就是把一个标识符和一个

存储地址联系在一起的过程。

按照联编进行阶段的不同，联编方法可以分为两种：静态联编和动态联编，这两种联编过程分别对应着多态的两种实现方式。

1. 静态联编

联编工作在编译连接阶段完成的情况称为静态联编。在编译、连接过程中，系统就可以根据类型匹配等特征确定程序中操作调用与执行该操作的代码的关系，即确定某一个同名标识到底是要调用哪一段程序代码。例如，函数重载和运算符重载就属于静态联编。

2. 动态联编

和静态多态相对应，联编工作在程序运行阶段完成的情况称为动态多态。在编译、连接过程中无法解决的联编问题，要等到程序开始运行之后再来确定。例如，虚函数就是通过动态联编完成的。

3.1.3 多态的实现原理

依据实现的方式，多态性可以分为静态多态和动态多态两种，它们之间有一定的相似性，但是应用范围和效率都有所不同。

1. 静态多态的实现原理

C++中的静态多态是通过静态联编实现的，具体表现就是重载。重载包括函数重载和运算符重载，而运算符重载本质上就是函数重载。

函数重载也称多态函数。C++编译系统允许为两个或两个以上的函数取相同的函数名，但是形参的个数或者形参的类型至少有一个不同，编译系统会根据实参和形参的类型及个数的最佳匹配，自动确定调用哪一个函数。

2. 动态多态的实现原理

C++中的动态多态是通过在基类的函数前加上 virtual 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

3.2 运算符重载

运算符重载是多态性的重要表现。C++中预定义的运算符的操作对象只能是基本数据类型，实际上，对于很多用户自定义的类型(如类)，也需要有类似的运算操作。例如，声明了一个复数类 CComplex：

```
class CComplex
{
private:
    int real,imag;
public:
    …
};
```

于是可以这样声明复数类 CComplex 的对象：

```
CComplex k1(1,1),k2(3,3)
```

如果需要对 k1 和 k2 进行加法运算，该如何实现呢？我们当然希望能使用“＋”运算符，写出表达式“k1＋k2”，但是编译的时候却会出错，因为编译器不知道该如何完成这个加法。这时候，我们就需要自己编写程序来说明“＋”在作用于 CComplex 类对象时，该实现什么样的功能，这就是运算符重载。运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据，执行不同类型的行为。

在运算符重载的实现过程中，首先把指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的实参，然后根据实参的类型来确定需要调用的函数，这个过程是在编译过程中完成的。

3.2.1　运算符重载的规则

1. 运算符重载的规则

运算符是在 C++系统内部定义的，它们具有特定的语法规则，如参数说明、运算顺序、优先级别等。因此，运算符重载时必须要遵守一定的规则。

（1）C++中的运算符除了少数几个（类属关系运算符“.”、作用域分辨符“::”、成员指针运算符“*”、sizeof 运算符和三目运算符“?:”）之外，全部可以重载，而且只能重载 C++中已有的运算符，不能臆造新的运算符。

（2）重载之后运算符的优先级和结合性都不能改变，也不能改变运算符的语法结构，即单目运算符只能重载为单目运算符，双目运算符只能重载为双目运算符。

（3）运算符重载后的功能应当与原有功能相类似。

（4）重载运算符含义必须清楚，不能有二义性。

2. 运算符重载的形式

运算符的重载形式有两种：重载为类的成员函数和重载为类的友元函数。

运算符重载为类的成员函数的一般语法形式为：

```
<函数类型>  operator  <运算符>(形参表)
{
    函数体;
}
```

运算符重载为类的友元函数的一般语法形式为：

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

其中：

（1）函数类型——指定了重载运算符的返回值类型，也就是运算结果类型。

（2）operator——是定义运算符重载函数的关键字。

（3）运算符——是要重载的运算符名称。

(4) 形参表——给出重载运算符所需要的参数和类型。

(5) friend——是对于运算符重载为友元函数时,在函数类型说明之前使用的关键字。

特别需要注意:当运算符重载为类的成员函数时,函数的参数个数比原来的操作数个数要少一个(后置"++"、"--"除外);当重载为类的友元函数时,参数个数与原操作数个数相同。原因是重载为类的成员函数时,如果某个对象使用了重载的成员函数,可以直接访问自身的数据,就不需要再放在参数表中进行传递,被省略的操作数就是该对象本身。而重载为友元函数时,友元函数对某个对象的数据进行操作,就必须通过该对象的名称来进行,因此,使用到的参数都要进行传递,操作数的个数就不会有变化。

3.2.2 运算符重载为成员函数

运算符重载实质上就是函数重载,当运算符重载为成员函数之后,它就可以自由地访问本类的数据成员了。实际使用时,总是通过该类的某个对象来访问重载的运算符。如果是双目运算符,一个操作数是对象本身的数据,由 this 指针指出,另一个操作数则需要通过运算符重载函数的参数表来传递;如果是单目运算符,操作数由对象的 this 指针给出,就不再需要任何参数。下面分别介绍这两种情况。

1. 双目运算

```
oprd1 B oprd2
```

对于双目运算符 B,如果要重载 B 为类的成员函数,使之能够实现表达式 oprd1 B oprd2。假设 oprd1 和 oprd2 为 A 类的对象,则应当把 B 重载为 A 类的成员函数,该函数只有一个形参,形参的类型是 A 类。经过重载之后,表达式 oprd1 B oprd2 就相当于函数调用 oprd1.operator B(oprd2)。

2. 单目运算

1) 前置单目运算

```
U oprd
```

对于前置单目运算符 U,如"-"(负号)、"++"等,如果要重载 U 为类的成员函数,用来实现表达式 U oprd。假设 oprd 为 A 类的对象,则 U 应当重载为 A 类的成员函数,函数没有形参。经过重载之后,表达式 U oprd 相当于函数调用 oprd.operator U( )。

例如,前置单目运算符"++"重载的语法形式为:

```
<函数类型>  operator  ++( );
```

使用前置单目运算符"++"的语法形式为:

```
++<对象>;
```

2) 后置单目运算

```
oprd V
```

再来看后置运算符 V,如"++"和"--",如果要将它们重载为类的成员函数,用来实现表达式 oprd ++或 oprd --。假设 oprd 为 A 类的对象,那么运算符就应当重载为 A

类的成员函数，这时函数要带有一个整型(int)形参。重载之后，表达式 oprd ++和 oprd --就相当于函数调用 oprd. operator ++(0)和 oprd. operator --(0)。

例如，后置单目运算符“++”重载的语法形式为：

```
<函数类型>  operator  ++(int);
```

使用后置单目运算符“++”的语法形式为：

```
<对象>++;
```

【例 3-1】 使用运算符重载为成员函数的方法，实现复数加法、前置自增和后置自减运算。

问题分析：复数的加减法是实部和虚部分别相加减，运算符的两个操作数都是 CComplex 类的对象，因此，可以把双目运算符“+”重载为 CComplex 类的成员函数，重载函数只有一个形参，类型同样也是 CComplex 类对象。前置单目运算符“++”重载为 CComplex 类的成员函数，重载函数没有形参。后置单目运算符“--”重载为 CComplex 类的成员函数，重载函数有一个形参。

程序代码如下：

```
#include "stdafx.h"
#include <iostream>
class CComplex
{
private:
    int real,imag;
public:
    CComplex(int r = 0, int i = 0) {real = r;imag = i;}
    int get_real( ){return real;}
    int get_imag( ){return imag;}

    CComplex operator + (CComplex);          //成员函数重载运算符"+"
    CComplex operator ++( );                 //成员函数重载运算符"++"为前置
    CComplex operator -- (int);              //成员函数重载运算符"--"为后置
};
CComplex CComplex::operator + (CComplex q)
{
    return  CComplex (real + q.real, imag + q.imag);
}
CComplex CComplex::operator ++( )
{
    return  CComplex (++real, ++imag);
}
CComplex CComplex::operator -- (int)
{
    return  CComplex (real--, imag--);
}
int main( )
{
    CComplex k1(3,3),k2(2,2),k3,k4,k5(1,1);  //声明 CComplex 类的对象
```

```
    k3 = k1 + k2;                                    //两复数相加
    ++k4;
    k5 -- ;
    cout <<"k1 + k2 =  "<< k3.get_real( )<<" +  i"<< k3.get_imag( )<< endl;
    cout <<"++k4 = "<< k4.get_real( )<<" +  i"<< k4.get_imag( )<< endl;
    cout <<"k5 --= "<< k5.get_real( )<<" +  i"<< k5.get_imag( )<< endl;
    return 0;
}
```

程序运行结果为：

```
k1 + k2 = 5 +  i5
++k4 = 1 +  i1
k5 --= 0 +  i0
```

可以看出，除了在函数声明及实现的时候使用了关键字 operator 之外，运算符重载成员函数与类的普通成员函数没有什么区别。在使用的时候，可以直接通过运算符、操作数的方式来完成函数调用。

3.2.3 运算符重载为友元函数

运算符也可以重载为类的友元函数，这样它就可以自由地访问该类的任何数据成员。这时，运算所需要的操作数都需要通过函数的形参表来传递，在参数表中形参从左到右的顺序就是运算符操作数的顺序。但是，有些运算符不能重载为友元，如“=”、“( )”、“[]”和“->”。

1. 双目运算

```
oprdl B oprd2
```

对于双目运算符 B，如果 oprd1 和 oprd2 为 A 类的对象，则应当把 B 重载为 A 类的友元函数，该函数有两个形参，且两个形参的类型都是 A 类，经过重载之后，表达式 oprd1 B oprd2 就相当于函数调用 operator B(oprd1,oprd2)。

2. 单目运算

1) 前置单目运算

```
U oprd
```

对于前置单目运算符 U，如“－”(负号)等，如果要实现表达式 U oprd，假设 oprd 为 A 类的对象，则 U 可以重载为 A 类的友元函数，函数的形参类型为 A 类，经过重载之后，表达式 U oprd 相当于函数调用 operator U(oprd)。

2) 后置单目运算

```
oprd V
```

对于后置运算符 V，如“＋＋”和“－－”，如果要实现表达式 oprd ＋＋或 oprd －－，假设 oprd 为 A 类的对象，那么运算符就可以重载为 A 类的友元函数，这时函数的形参有两个，一个形参的类型是 A 类，另一个形参的类型是整型(int)。重载之后，表达式 oprd ＋＋

和 oprd －－就相当于函数调用 operator ＋＋(oprd,0)和 operator －－(oprd,0)。

【例 3-2】 使用运算符重载为友元函数的方法,实现两个复数的相减运算、前置自增和后置自减运算。

程序代码如下:

```
#include "stdafx.h"
#include <iostream>
class CComplex
{
private:
    int real,imag;
public:
    CComplex(int r = 0, int i = 0) {real = r;imag = i;}
    int get_real( ){return real;}
    int get_imag( ){return imag;}

    friend CComplex operator - (CComplex,CComplex);   //友元函数重载运算符"-"
    friend CComplex operator ++(CComplex &);          //友元函数重载运算符"++"为前置
    friend CComplex operator -- (CComplex &);         //友元函数重载运算符"--"为后置,
                                                      //参数必须用引用
};
CComplex operator ++(CComplex &q)
{
    return  CComplex(++q.real, ++q.imag);
}
CComplex operator - (CComplex q1,CComplex q2)
{
    return  CComplex(q1.real - q2.real, q1.imag - q2.imag);
}

CComplex operator -- (CComplex &q)
{
    return  CComplex(q.real -- , q.imag -- );
}

int main( )
{
    CComplex k1(3,3),k2(2,2),k3,k4,k5(1,1);
    k3 = k1 - k2;                                     //两复数相减
    ++k4;
    k5 -- ;
    cout <<"k1 - k2 = "<< k3.get_real( )<<" + i"<< k3.get_imag( )<< endl;
    cout <<"++k4 = "<< k4.get_real( )<<" + i"<< k4.get_imag( )<< endl;
    cout <<"k5 --= "<< k5.get_real( )<<" + i"<< k5.get_imag( )<< endl;
    return 0;
}
```

程序运行结果为:

```
k1 - k2 = 1 + i1
```

```
++k4 = 1 + i1
k5 --= 0 + i0
```

从上述程序可以看到，将运算符重载为类的友元函数时，必须把操作数全部通过形参的方式传递给运算符重载函数。和【例 3-1】相比，本例题的主函数只是将两个复数相加改为相减，其余没有做任何改动，程序运行的结果除相减外，其他的完全相同。

3.2.4 运算符重载实例

前面介绍了一些简单运算符的重载，除此之外，还有以下运算符也常被重载。

1. 比较运算符重载

如：＜，＞，＜＝，＞＝，＝ ＝，！＝。

2. 赋值运算符重载

如：＝，＋＝，－＝，＊＝，/＝，这些运算符的重载比较简单。

【例 3-3】 比较运算符和赋值运算符重载。

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CComplex
{
private:
    float real,imag;
public:
    CComplex(float r = 0, float i = 0) {real = r;imag = i;}
    CComplex(CComplex &);
    ~CComplex( ){}
    bool operator == (CComplex);
    bool operator != (CComplex);
    CComplex operator += (CComplex);
    CComplex operator -= (CComplex);
    float get_real( ){return real;}
    float get_imag( ){return imag;}
};
CComplex::CComplex(CComplex &q)
{
    real = q.real;
    imag = q.imag;
}
bool CComplex::operator == (CComplex q)
{
    if( real == q.get_real( ) && imag == q.get_imag( ) )
        return 1;
    else
        return 0;
}
bool CComplex::operator != (CComplex q)
{
```

```
    if(real!= q.get_real( )&&imag!= q.get_imag( ))
        return 1;
    else
        return 0;
}
CComplex CComplex::operator += (CComplex q)
{
    this->real += q.get_real( );
    this->imag += q.get_imag( );
    return *this;
}
CComplex CComplex::operator -= (CComplex q)
{
    this->real -= q.get_real( );
    this->imag -= q.get_imag( );
    return *this;
}
int main( )
{
    CComplex k1(1,2),k2(3,4),k3(5,6);
    cout<<"k1 == k2?  "<<(k1 == k2)<< endl;
    cout<<"k1!= k2?  "<<(k1!= k2)<< endl;
    k3 += k1;
    cout<<"k3 += k1,k3:  "<< k3.get_real( )<<","<< k3.get_imag( )<< endl;
    k3 -= k1;
    cout<<"k3 -= k1,k3:  "<< k3.get_real( )<<","<< k3.get_imag( )<< endl;
    return 0;
}
```

程序运行结果为：

```
k1 == k2?  0
k1!= k2?  1
k3 += k1,k3:  6,8
k3 -= k1,k3:  5,6
```

3. 下标运算符重载

下标运算符“[ ]”通常用于取数组中的某个元素，通过下标运算符重载，可以实现数组下标的越界检测等。下标运算符“[ ]”的重载关键是将下标值作为一个操作数，它的实现其实非常简单，就是用字符指针的首地址加下标值，然后将相加后的地址返回，这样就实现了读取数组中某个元素的目的。

【例 3-4】 重载下标运算符。

程序代码如下：

```
#include "stdafx.h"
#include <iostream>
#include <string.h>
using namespace std;
class CWord
```

```
{
private:
    char * str;
public:
    CWord(char * s)
    {
        str = new char[strlen(s) + 1];
        strcpy(str,s);
    }
    char &operator [] (int k)
    {
        return * (str + k);
    }
    void disp( )
    {
        cout << str << endl;
    }
};

int main( )
{
    char * s = "china";
    CWord w(s);
    w.disp( );
    int n = strlen(s);
    while(n >= 0)
    {
        w[n - 1] = w[n - 1] - 32;
        n -- ;
    }
    w.disp( );
    return 0;
}
```

程序运行结果为：

```
china
CHINA
```

4. 运算符 new 和 delete 重载

通过重载 new 和 delete，可以克服 new 和 delete 的不足，使其按要求完成对内存的管理。

【例 3-5】 重载 new 和 delete 运算符。

问题分析：这是一个很简单的重载 new 和 delete 的程序，其中 new 通过 malloc( )函数实现，new 的操作数是申请内存单元的字节个数。delete 通过 free( )函数实现，它的操作数是一个指针，即告诉系统释放哪里的单元。

程序代码如下：

```
#include "stdafx.h"
```

```
#include <iostream>
#include <malloc.h>
using namespace std;
class  CRectangle
{
private:
    int length,width;
public:
    CRectangle(int l,int w)
    {
        length = l;
        width = w;
    }
    void * operator new(size_t size)
    {
        return  malloc(size);
    }
    void operator delete(void * p)
    {
        free(p);
    }
    void get_area( )
    {
        cout <<"area:"<< length * width << endl;
    }
};
int main( )
{
    CRectangle * pRect;
    pRect = new CRectangle(5,9);
    pRect -> get_area( );
    delete pRect;
    return 0;
}
```

程序运行结果为：

```
area:45
```

5. 逗号运算符重载

逗号运算符“,”是一个双目运算符，和其他运算符一样，也可以通过重载逗号运算符来达到期望的结果。逗号运算符构成的表达式为“左操作数，右操作数”，该表达式返回右操作数的值。

【例 3-6】 重载逗号运算符“,”。

程序代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class  CRectangle
```

```
{
private:
    int length,width;
public:
    CRectangle( ){};
    CRectangle(int l,int w)
    {
        length = l;
        width = w;
    }

    CRectangle operator , (CRectangle r)
    {
        CRectangle t;
        t.length = r.length;
        t.width = r.width;
        return  t;
    }
    CRectangle operator + (CRectangle r)
    {
        CRectangle t;
        t.length = r.length;
        t.width = r.width;
        return  t;
    }
    void get_area( )
    {
        cout <<"area:"<< length * width << endl;
    }
};
int main( )
{
    CRectangle rect1(1,2),rect2(3,4),rect3(5,6);
    rect1.get_area( );
    rect2.get_area( );
    rect3.get_area( );
    rect1 = (rect1,rect2 + rect3,rect3);
    rect1.get_area( );
    return 0;
}
```

程序运行结果为：

```
area:2
area:12
area:30
area:30
```

3.3 虚函数

虚函数是在引入了派生概念以后，用来表现基类和派生类的成员函数之间的一种关系的，它是动态多态的基础。

3.3.1　虚函数的定义

1. 虚函数的定义

虚函数的定义是在基类中进行的，它是在基类中需要定义为虚函数的成员函数的声明中冠以关键字 virtual，从而提供了一种接口界面。当基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义，在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型以及参数的顺序都必须与基类中的原型完全相同。

虚函数的定义形式为：

```
virtual  <函数类型>  <函数名>(形参表)
{
    函数体
}
```

其中，被关键字 virtual 说明的函数为虚函数。

特别要注意：虚函数的声明只能出现在类声明中的函数原型声明中，而不能出现在成员的函数体实现的时候。

动态多态只能通过成员函数来调用或者通过指针、引用来访问虚函数。如果采用对象名的形式访问虚函数，则将采用静态多态方式调用虚函数，而无须在运行过程中进行调用。

【例 3-7】 通过指针访问虚函数。

程序代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CBase                                          //定义基类 CBase
{
public:
    virtual void who( )                              //虚函数声明
    {  cout<<"this is CBase !"<<endl;  }
};
class CDerive1:public CBase                          //定义基类派生类 CDerive1
{
public:
    void who( )                                      //重新定义虚函数
    {  cout<<"this is CDerive1 !"<<endl;  }
};
class CDerive2:public CBase                          //定义派生类 CDerive2
{
public:
    void who( )                                      //重新定义虚函数
    {  cout<<"this is CDerive2 !"<<endl;  }
};
int main( )
{
```

```
    CBase obj, * ptr;
    CDerive1 obj1;
    CDerive2 obj2;
    ptr = &obj;
    ptr -> who( );
    ptr = &obj1;
    ptr -> who( );
    ptr = &obj2;
    ptr -> who( );
    return 0;
}
```

程序运行结果为：

```
this is CBase!
this is Cderive1!
this is CDerive2!
```

程序说明：

在基类中对 who( )进行了虚函数声明 virtual，在其派生类中就可以重新定义它。在派生类 CDerive1 和 CDerive2 中分别重新定义函数 who( )，此虚函数在派生类中重新定义时不再需要 virtual 声明，此声明只在其基类中出现一次。当函数 who( )被重新定义时，其函数的原型与基类中的函数原型必须完全相同。在 main( )函数中，定义了一个指向基类类型的指针，它也被允许指向其派生类，在执行过程中，不断改变它所指向的对象，ptr->who( )就能调用不同的版本，虽然都是 ptr->who( )语句，但是，当 ptr 指向不同的对象时，所对应的执行函数就不同。由此可见，用虚函数充分体现了多态性，并且，因为 ptr 指针指向哪个对象是在执行过程中确定的，所以体现的又是一种动态的多态性。

2. 虚函数与重载的关系

在一个派生类中重新定义基类的虚函数是函数重载的另一种特殊形式，但它不同于一般的函数重载。

一般的函数重载，只要函数名相同即可，函数的返回类型及所带的参数可以不同。但当重载一个虚函数时，也就是说，在派生类中重新定义此虚函数时，则要求函数名、返回类型、参数个数、参数类型以及参数的顺序都与基类中原型完全相同，不能有任何不同。

另外，在多继承中由于派生类是由多个基类派生而来的，因此，虚函数的使用就不像单继承那样简单。若一个派生类，它的多个基类中有公共的基类，在公共基类中定义一个虚函数，则多重派生以后仍可以重新定义虚函数，也就是说，虚特性是可以传递的。

3.3.2 虚函数的限制

如果将所有的成员函数都设置为虚函数，这当然是很有益的。它除了会增加一些额外的资源开销，没有什么坏处。但设置虚函数时须注意：

(1) 只有成员函数才能声明为虚函数。因为虚函数仅适用于有继承关系的类对象，所以普通函数不能声明为虚函数。

(2) 虚函数必须是非静态成员函数。这是因为静态成员函数不受限于某个对象。

(3) 内联函数不能声明为虚函数。因为内联函数不能在运行中动态确定其位置。

(4) 构造函数不能声明为虚函数。多态是指不同的对象对同一消息有不同的行为特性。虚函数作为运行过程中多态的基础,主要是针对对象的,而构造函数是在对象产生之前运行的,因此,虚构造函数是没有意义的。

(5) 析构函数可以声明为虚函数。析构函数的功能是在该类对象消亡之前进行一些必要的清理工作,析构函数没有类型,也没有参数,和普通成员函数相比,虚析构函数情况略为简单。

3.3.3　虚析构函数

虚析构函数的定义形式为:

```
virtual ~类名
```

例如:

```
class B
{
public:
    //…
    virtual ~B( );
};
```

如果一个类的析构函数是虚函数,那么由它派生而来的所有子类的析构函数也是虚函数。析构函数置为虚函数之后,在使用指针引用时可以实现动态多态,即通过使用基类的指针可以调用相应的析构函数对不同的对象进行清理工作。

如果某个类不包含虚函数,那一般是表示它将不作为一个基类来使用。当一个类不准备作为基类使用时,使析构函数为虚一般不是一个好主意,因为它会为类增加一个虚函数表,使得对象的体积翻倍,还有可能降低其可移植性。

无故地声明虚析构函数和永远不去声明一样是错误的。实际上,当且仅当类里包含至少一个虚函数的时候才去声明虚析构函数。

3.3.4　纯虚函数和抽象类

1. 纯虚函数

一个抽象类至少带有一个纯虚函数。纯虚函数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的操作内容,要求各派生类根据实际需要定义自己的实现内容。纯虚函数的声明形式为:

```
virtual  <函数类型>  <函数名>(参数表) = 0
```

纯虚函数与一般虚函数成员的原型在形式上的不同就在于后面加了“=0”,表明在基类中不用定义该函数,它的实现部分——函数体留给派生类去做。

2. 抽象类

含有一个以上纯虚函数的类称为抽象类。抽象类的主要作用是通过它为一个类族建立一个公共的接口,使它们能够更有效地发挥多态特性。使用抽象类时需注意以下几点:

(1) 抽象类只能用作其他类的基类,不能建立抽象类对象。抽象类处于继承层次结构的较上层,一个抽象类自身无法实例化,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。

(2) 抽象类不能用作参数类型、函数返回值或显式转换的类型。

(3) 可以声明一个抽象类的指针和引用。通过指针或引用,就可以指向并访问派生类对象,以实现访问派生类的成员。

(4) 抽象类的类中需要声明纯虚析构函数。

抽象类派生出新的类之后,如果派生类给出所有纯虚函数的函数实现,这个派生类就可以声明自己的对象,因而不再是抽象类;反之如果派生类没有给出全部纯虚函数的实现,这时的派生类仍然是一个抽象类。我们来看下面的例题。

【例 3-8】 使用形状抽象类,通过重新定义求面积的虚函数求长方形和圆的面积。在基类 CShapes 中将成员 get_area( )声明为纯虚函数,这样,基类 CShapes 就是一个抽象类,这时无法声明 CShapes 类的对象,但是可以声明 CShapes 类的指针和引用。CShapes 类经过公有派生产生了 CRectangle 类和 CCircle 类。使用抽象类 CShapes 类型的指针,当它指向某个派生类的对象时,就可以通过它访问该对象的虚成员函数。

程序代码如下:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
const double PI = 3.14159;
class CShapes                                          //抽象基类 CShapes 声明
{
protected:
    int x,y;
public:
    void setvalue(int xx,int yy = 0){x = xx;y = yy;}
    virtual void get_area( ) = 0;                      //纯虚函数成员
};

class CRectangle:public CShapes                        //派生类 CRectangle 声明
{
public:                                                //虚成员函数
    void get_area( ){cout <<"The area of rectangle is :"<< x * y << endl;}
};

class CCircle:public CShapes                           //派生类 CCircle 声明
{
public:                                                //虚成员函数
    void get_area( ){cout <<"The area of circle is :"<< PI * x * x << endl;}
};
int main( )
{
    CShapes * ptr[2];                                  //声明抽象基类指针
    CRectangle rect1;
```

```
    CCircle cir1;
    ptr[0] = &rect1;                                    //指针指向 CRectangle 类对象
    ptr[0] -> setvalue(5,8);
    ptr[0] -> get_area( );
    ptr[1] = &cir1;                                     //指针指向 CCircle 类对象
    ptr[1] -> setvalue(10);
    ptr[1] -> get_area( );
    return 0;
}
```

程序说明：

程序中类 CShapes、CRectangle 和 CCircle 属于同一个类族，抽象类 CShapes 通过纯虚函数为整个类族提供了通用的外部接口语义。通过公有派生而来的子类给出了纯虚函数的具体函数体实现，因此是非抽象类。这时可以定义非抽象类的对象，同时根据赋值兼容规则，抽象类 CShapes 类型的指针也可以指向任何一个派生类的对象，通过基类 CShapes 的指针可以访问到正在指向的派生类 CRectangle 和 CCircle 类对象的成员，这样就实现了对同一类族中的对象进行统一的多态处理。

程序运行结果为：

```
The area of rectangle is :40
The area of circle is :314.159
```

另外，程序中派生类的虚成员函数 get_area( )并没有用关键字 virtual 显式说明，因为它们与基类的纯虚函数具有相同的名称及参数和返回值，由系统自动判断确定其为虚成员函数。

习题

3-1　填空题

（1）C++中的运算符除了_____之外，全部可以重载，而且只能重载 C++中已有的运算符，不能臆造新的运算符。

（2）如果用普通函数重载双目运算符，需要_____个操作数；重载单目运算符，需要_____个操作数。如果用友员函数重载双目运算符，需要_____个操作数；重载单目运算符，需要_____个操作数。

（3）当基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义，在派生类中重新定义时，其函数原型，包括_____、_____和_____，以及_____和_____都必须与基类中的原型完全相同。

3-2　简答题

（1）什么叫做多态性？C++中是如何实现多态的？

（2）函数重载与虚函数有哪些相同点与不同点？

（3）什么叫做抽象类？抽象类有何作用？

3-3　编程题

（1）声明一个 CShape 抽象类，在此基础上派生出 CRectangle 和 CCircle 类，二者都有

GetPerim( )函数计算相应对象的周长。

(2) 编写一个程序声明一个矩阵类,通过重载“+”、“-”和“*”,实现矩阵的相加、相减和相乘。

(3) 编写一个程序,先设计一个链表 list 类,再从链表类派生出一个集合类 set,在集合类中添加一个记录元素个数的数据项。要求可以实现对集合的插入、删除、查找和显示。