

The C++ Programming Language

多态性与虚函数

陈辰

chenc@fudan.edu.cn

复旦大学软件学院

内容要点

- 多态性
- 虚函数
- 纯虚函数
- 抽象类

5.1. 多态性概述

所谓多态性就是不同对象收到相同的消息时，产生不同的动作

多态性的两种描述：

(1) 指具有不同功能的函数可以用同一个函数名；

(2) 指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；

从实现角度看，分为静态多态性和动态多态性

静态多态性——函数重载和运算符重载；

运行时的多态——继承和虚函数；

多态的实现：

- ◆ 函数重载；
- ◆ 运算符重载；
- ◆ 虚函数；

多态的分类

C++中的多态性可以分为四类:

通用多态 { 参数多态
包含多态

专用多态 { 重载多态
强制多态

- **参数多态**与类属函数和类属类相关联，**函数模板**和**类模板**就是这种多态。
- **包含多态**是研究类族中定义于不同类中的同名成员函数的多态行为，主要是通过**虚函数**来实现的。
- **重载多态**如**函数重载**、**运算符重载**等。普通函数及类的成员函数的重载多属于重载多态。
- **强制多态**是指将一个变元的类型加以变化，以符合一个函数或操作的要求，例如加法运算符在进行浮点数与整型数相加时，首先进行类型强制转换，把整型数变为浮点数再相加的情况，就是强制多态的实例。

多态的实现

多态从实现的角度来讲可以划分为两类：编译时多态和运行时多态。在C++中，多态的实现和**联编**这一概念有关。

所谓**联编**就是把函数名与函数体的程序代码连接(联系)在一起的过程。

静态联编就是在**编译**阶段完成的联编。

动态联编是**运行**阶段完成的联编。

多态从实现的角度来讲可以划分为两类：

编译时的多态：编译时的多态是通过**静态联编**来实现的。

静态联编时，系统用实参和形参进行匹配，对于同名的重载函数便根据参数上的差异进行区分，然后进行联编。从而实现了多态性。

运行时的多态：运行时的多态是用**动态联编**实现的。

当程序调用到某一函数名时，才去寻找和连接其程序代码，面向对象的设计而言，就是当对象接收到某一消息时，才去寻找和连接相应的方法。

一般而言，**编译型语言**都采用静态联编，而**解释性语言**都采用动态联编。

纯粹的面向对象程序设计语言由于其执行机制是消息传递，所以只能采用动态联编。这就给基于C语言的C++带来了麻烦。因为为了保持C语言的高效性，**C++**仍是**编译型**的，仍采用**静态联编**。好在C++的设计者想出了“**虚函数**”的机制，解决了这个问题。利用虚函数机制，C++可部分地采用动态联编。这就是说，C++实际上是采用了静态联编和动态联编相结合的联编方法。

在C++中，**编译时多态性**主要是通过**函数重载**和**运算符重载**实现的。**运行时多态性**主要是通过**虚函数**来实现的。

一个典型的例子

例5.0 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。

(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;    };
```

```
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }    //定义构造函数，对圆心坐标和半径初始化
```

```
void Circle::setRadius(float r)    //设置半径值
```

```
{radius=r;}
```

```
float Circle::getRadius( ) const {return radius;} //读取半径值
```

```
float Circle::area( ) const    //计算圆面积
```

```

{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。

5.2. 虚函数

虚函数允许函数调用与函数体之间的联系在运行时才建立，也就是在运行时才决定如何动作，即所谓的动态联编。

- 虚函数是动态联编的基础。虚函数是成员函数，而且是非static的成员函数。说明虚函数的方法如下：
- `virtual<类型说明符><函数名>(<参数表>)`
- 其中，被关键字virtual说明的函数称为虚函数。

- 如果某类中的一个成员函数被说明为虚函数，这就意味着该成员函数在派生类中可能有不同的实现。当使用这个成员函数操作指针或引用所标识对象时，对该成员函数调用采取动态联编方式，即在运行时进行关联或绑定。
- 动态联编只能通过指针或引用标识对象来操作虚函数。如果采用一般类型的标识对象来操作虚函数，则将采用静态联编方式调用虚函数。

- 派生类中对基类的虚函数进行替换时，要求派生类中说明的虚函数与基类中的被替换的虚函数之间满足如下条件：
 - (1) 与基类的虚函数有相同的参数个数；
 - (2) 其参数的类型与基类的虚函数的对应参数类型相同；
 - (3) 其返回值或者与基类虚函数的相同，或者都返回指针或引用，并且派生类虚函数所返回的指针或引用的基类型是基类中被替换的虚函数所返回的指针或引用的基类型的子类型。

- 总结动态联编的实现需要如下三个条件：
 - (1) 要有说明的虚函数；
 - (2) 调用虚函数操作的是指向对象的指针或者对象引用；或者是由成员函数调用虚函数；
 - (3) 子类型关系的建立

构造函数中调用虚函数时，采用静态联编即构造函数调用的虚函数是自己类中实现的虚函数，如果自己类中没有实现这个虚函数，则调用基类中的虚函数，而不是任何派生类中实现的虚函数。

虚函数的作用和定义

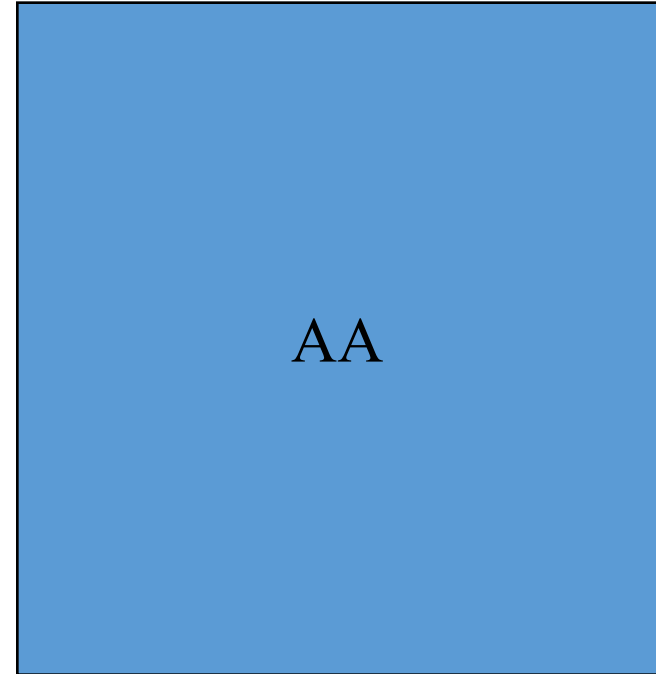
1. 虚函数的作用

虚函数首先是基类中的成员函数,在其前面加上virtual关键字,并在派生类中被重载。

虚函数同派生类的结合可使C++支持运行时的多态性,实现了在基类定义派生类所拥有的通用接口,而在派生类定义具体的实现方法,即常说的“同一接口,多种方法”,它帮助程序员处理越来越复杂的程序。

例5.1 虚函数例子

```
#include<iostream.h>
class A {
public:
    void show() { cout<<"A"; }
};
class B:public A {
public:
    void show() { cout<<"B"; }
};
main()
{
    A a,*pc;
    B b;
    pc=&a; pc->show();
    pc=&b; pc->show();
    return 0;
}
```



【例5.2】 虚函数的例子

```
#include<iostream.h>
class Base{
public:
    Base(int x,int y)
    {
        a=x;
        b=y;
    }
    void show()
    {
        cout<<"Base-----\n";
        cout<<a<<" "<<b<<endl;
    }
private:
    int a , b;
};
```

```

class Derived : public Base
{
public:
    Derived(int x, int y, int z) : Base(x, y)
    {
        c = z;
    }
    void show()
    {
        cout << "Derived-----\n";
        cout << "c=" << c << endl;
    }
private:
    int c;
};

void main()
{
    Base mb(60, 60), *pc;
    Derived mc(10, 20, 30);
    pc = &mb;
    pc->show();
    pc = &mc;
    pc->show();
}

```

```

Base-----
60      60
Base-----
10     20

```

执行语句：

```
pc=&mc;
```

后, 指针pc已经指向了对象mc, 但它所调用的函数show(), 仍然是基类对象的show(), 显然这不是我们所期望的。

这个程序的错误是由C++**静态联编机制**造成的。对于上面的程序, 静态联编机制首先将基类对象的指针pc与基类的成员函数show()连接在一起, 这样, 不管指针pc指向哪个对象, pc->show()调用的总是基类中的成员函数show()。为解决这一问题, C++引入了虚函数的概念。

例5.3 虚函数的作用

```
#include<iostream.h>
class Base {
public:
    Base(int x,int y)
    { a=x; b=y; }
    virtual void show()    //定义虚函数show()
    { cout<<"Base-----\n"; cout<<a<<" "<<b<<endl;}
private:
    int a,b; };
class Derived : public Base    {
public:
    Derived(int x,int y,int z):Base(x,y){c=z; }
    void show()                //重新定义虚函数show()
    { cout<<"Derived-----\n"; cout<<c<<endl;}
private:
    int c;
};
```

```
void main()
{
    Base mb(60,60),*pc;
    Derived mc(10,20,30);
    pc=&mb;
    pc->show();    //调用基类Base的show()版本
    pc=&mc;
    pc->show();    //调用派生类Derived的show()版本
}
```

程序运行结果如下:

Base-----

60 60

Derived-----

30

2. 虚函数的定义

在基类中定义虚函数的方法如下:

```
virtual 函数类型 函数名(形参表)
```

```
{
```

```
    // 函数体
```

```
}
```

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、参数类型的顺序，都必须与其基类中的原型完全相同。

例5.4 虚函数的定义举例

```
#include<iostream.h>
```

```
class Grandma {
```

```
    public:
```

```
        virtual void introduce_self() // 定义虚函数introduce_self()
```

```
        { cout<<"I am grandma."<<endl; }
```

```
};
```

```
class Mother:public Grandma {
```

```
    public:
```

```
        void introduce_self() // 重新定义虚函数introduce_self()
```

```
        { cout<<"I am mother."<<endl;}
```

```
};
```

```
class Daughter:public Mother {
```

```
    public:
```

```
        void introduce_self() // 重新定义虚函数introduce_self()
```

```
        { cout<<"I am daughter."<<endl;}
```

```
};
```

```
void main()
{
    Grandma *ptr;
    Grandma g;
    Mother m;
    Daughter d;
    ptr=&g;
    ptr->introduce_self();//调用基类Grandma的introduce_self()
    ptr=&m;
    ptr->introduce_self();// 调用派生类Mother的introduce_self()
    ptr=&d;
    ptr->introduce_self();    //调用派生类 Daughter的introduce_self()
}
```

I am grandma.
I am mother.
I am daughter.

程序只在基类Grandma中显式定义了introduce_self()为虚函数。C++规定，如果在派生类中，没有用virtual显示地给出虚函数声明，这时系统就会遵循以下的规则来判断一个成员函数是不是虚函数：

- 该函数与基类的虚函数有相同的名称。
- 该函数与基类的虚函数有相同的参数个数及相同的对应参数类型。
- 该函数与基类的虚函数有相同的返回类型或者满足赋值兼容规则的指针、引用型的返回类型。

派生类的函数满足了上述条件，就被自动确定为虚函数。

- **//例5.5: 演示定义和访问虚函数**

```
#include<iostream.h>
```

```
class base                                //定义基类base
```

```
{ public:
```

```
    virtual void who( )                //虚函数声明
```

```
    { cout<<"this is the class of base !"<<endl ; }
```

```
};
```

```
class derive1: public base              //定义派生类derive1
```

```
{ public:
```

```
    void who( )                        //重新定义虚函数
```

```
    { cout<<"this is the class of derive1 !"<<endl ; }
```

```
};
```

```

class derive2: public base//定义派生类derive2
{
    public:
        void who( )           //重新定义虚函数
            { cout<<"this is the class of derive2 !"<<endl; }
};

main( )
{
    base obj, *ptr;           //声明基类对象obj、指针ptr
    derive1 obj1;              //声明派生类1的对象obj1
    derive2 obj2;              //声明派生类2的对象obj2
    ptr = &obj;                //基类指针指向基类对象
    ptr->who( );                //调用基类成员函数
    ptr = &obj1;               //基类指针指向派生类1对象
    ptr->who( );                //调用派生类1成员函数
    ptr = &obj2;               //基类指针指向派生类2对象
    ptr->who( );                //调用派生类2成员函数
    return 1;
}

```


程序运行结果为：

this is the class of base!

this is the class of derive1!

this is the class of derive2!

如果在基类中去掉关键词virtual， 则

程序运行结果为：

this is the class of base!

this is the class of base!

this is the class of base!

例5.6 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。

虚函数定义的几点说明

- (1) . 通过定义虚函数来使用C++提供的多态机制时，派生类应该从它的基类公有派生。赋值兼容规则成立的前提条件是派生类从其基类公有派生。
- (2) . 必须首先在基类中定义虚函数。在实际应用中，应该在类等级内需要具有动态多态性的几个层次中的最高层类内首先声明虚函数。
- (3) . 在派生类对基类中声明的虚函数进行重新定义时，关键字virtual可以写也可以不写。
- (4) . 使用对象名和点运算符的方式也可以调用虚函数，但是这种调用在编译时进行的是静态联编，它没有充分利用虚函数的特性。只有通过基类指针访问虚函数时才能获得运行时的多态性。

(5) 一个虚函数无论被**公有继承多少次**，它仍然保持其虚函数的特性。

(6) 虚函数必须是其所在类的**成员函数**，而不能是**友元函数**，也不能是**静态成员函数**，因为虚函数调用要靠特定的对象来决定该激活哪个函数。但是虚函数可以在另一个类中被声明为友元函数。

(7) **内联函数不能是虚函数**，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时仍将其看作是非内联的。

(8) **构造函数不能是虚函数**。因为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。

(9) **析构函数可以是虚函数**，而且通常说明为虚函数。

虚析构函数

- 问题：

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。

- 解决方法：

将基类的析构函数声明为虚函数。

- 析构函数设置为虚函数后，在使用指针引用时可以动态联编，实现运行时的多态，保证使用基类类型的指针能够调用适当的析构函数针对不同的对象进行清理工作。
- 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派生类的析构函数也是虚析构函数，不管它是否使用了关键字virtual进行说明。
- 虚析构函数的定义格式：
Virtual ~类名（）；

【例5.7】 使用虚析构函数举例

```
#include<iostream.h>
class Grandma{
public:
    Grandma()
    { }
    virtual ~Grandma()
    {
        cout<<"This is Grandma::~~Grandma()."<<endl;
    }
};

class Mother:public Grandma{
public:
    Mother()
    { }
    ~Mother()
    {
        cout<<"This is Mother::~~Mother()."<<endl;
    }
};
```

```
void main()  
{  
    Grandma *f;  
    f=new Mother;  
    delete f;  
}
```

运行结果为：

This is Mother::~~Mother() .

This is Grandma::~~Grandma().

虚函数与重载函数的关系

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形式，但它不同于一般的函数重载。

◆ 普通的函数重载时，其函数的参数个数或参数类型必须有所不同，函数的返回类型也可以不同。

◆ 当重载一个虚函数时，也就是说在派生类中重新定义虚函数时，要求函数名、返回类型、参数个数、参数的类型和顺序与基类中的虚函数原型完全相同。

◆ 如果仅仅返回类型不同，其余均相同，系统会给出错误信息；

◆ 若仅仅函数名相同，而参数的个数、类型或顺序不同，系统将它作为普通的函数重载，这时将丢失虚函数的特性。

虚函数的限制

- (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局限于某个对象。
- (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些。

【例5.8】虚函数与重载函数的比较

```
#include<iostream.h>
class Base{
public:
    virtual void func1();
    virtual void func2();
    virtual void func3();
    void func4();
};
class Derived:public Base {
public:
    virtual void func1();
    void func2(int x);
    char func3();
    void func4();
};
```

```

void Base::func1()
{    cout<<"--Base func1--\n"; }
void Base::func2()
{    cout<<"--Base func2--\n"; }
void Base::func3()
{    cout<<"--Base func3--\n"; }
void Base::func4()
{    cout<<"--Base func4--\n"; }
void Derived::func1()
{    cout<<"--Derived func1--\n"; }
void Derived::func2(int x)
{    cout<<"--Derived func2--\n"; }
void Derived::func4()
{    cout<<"--Derived func4--\n"; }
void main()
{    Base d1 , *bp;
    Derived d2;
    bp=&d2;
    bp->func1();
    bp->func2();
    bp->func4();
}

```

```

--Derived func1--
--Base func2--
--Base func4--

```

```

//调用Derived::func1()
//调用Base::func2()
//调用Base::func4()


```

多继承与虚函数

【例5.9】多继承情况下的虚函数调用

```
#include<iostream.h>
class Base1{
public:
    virtual void fun()          //定义fun()是虚函数
    {    cout<<"--Base1--\n";    }
};
class Base2{
public:
    void fun()                  //定义fun()为普通的成员函数
    {    cout<<"--Base2--\n";    }
};
class Derived:public Base1,public Base2{
public:
    void fun()
    {    cout<<"--Derived--\n";    }
};
```

```
void main()  
{  
    Base1 obj1,*ptr1;  
    Base2 obj2,*ptr2;  
    Derived obj3;  
    ptr1=&obj1;  
    ptr1->fun();  
    ptr2=&obj2;  
    ptr2->fun();  
    ptr1=&obj3;  
    ptr1->fun();  
    ptr2=&obj3;  
    ptr2->fun();  
}
```



--Base1--
--Base2--
--Derived--
--Base2--

虚函数举例

【例5.10】应用C++的多态性，计算三角形、矩形和圆的面积

```
#include<iostream.h>
class Figure {                                // 定义一个公共基类
public:
    Figure(double a,double b)
    {
        x=a;
        y=b;
    }
    virtual void show_area()
    {
        cout<<"No area computation defined";
        cout<<"for this class.\n";
    }
protected:
    double x,y;
};
```

```
class Triangle:public Figure // 定义三角形派生类
{
public:
    Triangle(double a,double b):Figure(a,b)
    { };
    void show_area() // 虚函数重定义,用作求三角形的面积
    {
        cout<<"Triangle with height "<<x;
        cout<<" and base "<<y<<" has an area of ";
        cout<<x*y*0.5<<endl;}
    };
class Square:public Figure { // 定义矩形派生类
public:
    Square(double a,double b):Figure(a,b)
    { };
    void show_area() // 虚函数重定义,用作求矩形的面积
    {
        cout<<"Square with dimension "<<x;
        cout<<" * "<<y<<" has an area of ";
        cout<<x*y<<endl;
    }
};
```

```
class Circle:public Figure { // 定义圆派生类
public:
    Circle(double a):Figure(a,a)
    {   };
    void show_area()           // 虚函数重定义，用作求圆的面积
    {
        cout<<"Circle with radius "<<x;
        cout<<" has an area of ";
        cout<<x*x*3.1416<<endl;
    }
};
```

```

main()
{
    Figure *p;                // 定义基类指针p
    Triangle t(10.0,6.0);     // 定义三角形类对象t
    Square s(10.0,6.0);       // 定义矩形类对象s
    Circle c(10.0);           // 定义圆类对象c
    p=&t;
    p->show_area();            // 计算三角形面积
    p=&s;
    p->show_area();            // 计算矩形面积
    p=&c;
    p->show_area();            // 计算圆面积
    return 0;
}

```

Triangle with height 10 and base 6 has an area of 30
 Square with dimension 10*6 has an area of 60
 Circle with radius 10 has an area of 314.16

【例5.11】 Virtual

members

```
// virtual members
#include <iostream>
using namespace std;
class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area () { return 0; }
};

class Rectangle: public Polygon {
public:
    int area () { return width * height; }
};
```

【例5.11】 **Virtual members**

```
class Triangle: public Polygon {
public:
    int area () { return (width * height / 2); }
};

int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon poly;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    Polygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << '\n';
    cout << ppoly2->area() << '\n';
    cout << ppoly3->area() << '\n';
    return 0;
}
```

5.3. 纯虚函数和抽象类

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，但要求在它的派生类中必须定义自己的版本，或重新说明为纯虚函数。

纯虚函数的定义形式如下：

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书写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。

纯虚函数

纯虚函数没有函数体；

最后面的 “=0” 并不表示函数返回值为0；

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。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。

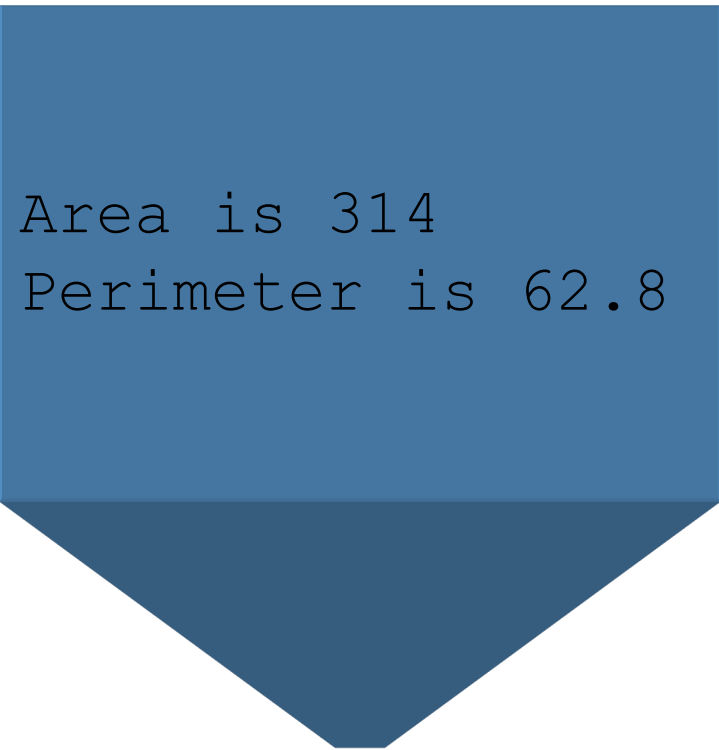
如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数。

一个具有纯虚函数的类称为抽象类。

例5.12纯虚函数的使用

```
#include<iostream.h>
class Circle {
public:
    void setr(int x){ r=x; }
    virtual void show()=0; // 纯虚函数
protected:
    int r;
};
class Area:public Circle{
public:
    void show(){ cout<<"Area is "<<3.14*r*r<<endl;}
};          // 重定义虚函数show( )
class Perimeter:public Circle{
public:
    void show(){cout<<"Perimeter is "<<2*3.14*r<<endl;}
};          // 重定义虚函数show( )
```

```
void main()
{
    Circle *ptr;
    Area ob1;
    Perimeter ob2;
    ob1.setr(10);
    ob2.setr(10);
    ptr=&ob1;
    ptr->show();
    ptr=&ob2;
    ptr->show();
}
```



Area is 314
Perimeter is 62.8

完整的例子

```
#include <iostream>
using namespace std; // Base class
class Shape
{
public: // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
protected:
    int width;
    int height;
};
```

完整的例子

// Derived classes

```
class Rectangle: public Shape
```

```
{
```

```
public:
```

```
    int getArea() { return (width * height); }
```

```
};
```

```
class Triangle: public Shape {
```

```
public:
```

```
    int getArea() { return (width * height)/2; }
```

```
};
```

完整的例子

```
int main(void)
{
    Rectangle Rect;
    Triangle Tri;
    Rect.setWidth(5);
    Rect.setHeight(7); // Print the area of the object.
    cout << "Total Rectangle area: " << Rect.getArea() << endl;
    Tri.setWidth(5);
    Tri.setHeight(7); // Print the area of the object.
    cout << "Total Triangle area: " << Tri.getArea() << endl;
    return 0;
}
```

完整的例子

When the above code is compiled and executed, it produces the following result –

Total Rectangle area: 35

Total Triangle area: 17

5.4 抽象类

如果一个类至少有一个**纯虚函数**，那么就称该类为**抽象类**。

抽象类只能作为其他类的基类来使用，**不能建立抽象类对象**，其纯虚函数的实现由派生类给出。派生类中必须重载基类中的纯虚函数，否则它仍将被看作一个抽象类。

使用抽象类的几点规定

- (1) 由于抽象类中至少包含一个没有定义功能的纯虚函数。因此，抽象类只能作为其他类的基类来使用，**不能建立抽象类的对象**，纯虚函数的实现由派生类给出。
- (2) 不允许从具体类派生出抽象类。
- (3) 抽象类**不能用作参数类型、函数返回类型或显式转换的类型**。
- (4) **可以**声明指向抽象类的**指针或引用**，此指针可以指向它的派生类，进而实现多态性。
- (5) 抽象类的析构函数可以被声明为纯虚函数，这时，应该至少提供该析构函数的一个实现。
- (6) 如果派生类中没有重定义纯虚函数，而派生类只是继承基类的纯虚函数，则这个派生类仍然是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不是抽象类了，它是一个可以建立对象的具体类。
- (7) 在抽象类中也可以定义普通成员函数或虚函数，虽然不能为抽象类声明对象，但仍然可以通过派生类对象来调用这些不是纯虚函数的函数。


```
class Shape {  
public :  
    virtual void rotateshape(int)=0;  
    virtual void drawshape()=0;  
    virtual void hiliteshape()=0;  
    //...  
};
```

Shape s1;	//错误,不能建立抽象类的对象
Shape* ptr;	//正确,可以声明指向抽象类的指针
Shape f();	//错误,抽象类不能作为函数的返回类型
Shape g(shape s);	//错误,抽象类不能作为函数的参数类型
Shape& h(shape&);	//正确,可以声明抽象类的引用

Abstract base classes

// abstract base class

```
#include <iostream>
using namespace std;
class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
public:
    int area (void) { return (width * height); }
};
```

Abstract base classes

```
class Triangle: public Polygon {
public:
    int area (void) { return (width * height / 2); }
};

int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << ppoly1->area() << '\n';
    cout << ppoly2->area() << '\n';
    return 0;
}
```

Abstract base classes

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;
class Polygon {
protected:
    int width, height; public:
    void set_values (int a, int b) { width=a; height=b;
}
    virtual int area() =0;
    void printarea() { cout << this->area() << '\n'; }
};
class Rectangle: public Polygon {
public:
    int area (void) { return (width * height); }
};
```

Abstract base classes

```
class Triangle: public Polygon {
public:
    int area (void) { return (width * height / 2); }
};

int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    return 0;
}
```

- // 【例5.13】 演示抽象类和纯虚函数

```
#include <iostream.h>
```

```
const double PI = 3.14159 ;
```

```
class Shapes //抽象基类Shapes声明
```

```
{
```

```
protected:
```

```
    int x, y ;
```

```
public:
```

```
    void setvalue( int xx, int yy=0 ) { x=xx ; y=yy ; }
```

```
    virtual void display( ) = 0 ; //纯虚函数成员
```

```
};
```

```
class Rectangle:public Shapes //派生类Rectangle声明
```

```
{
```

```
public:
```

```
    void display( ) //虚成员函数
```

```
    { cout<<"The area of rectangle is : "<<x*y<<endl ; }
```

```
};
```

```

class Circle:public Shapes//派生类Circle声明
{ public:
    void display( )           //虚成员函数
    { cout<<"The area of circle is : "<<PI*x*x<<endl ; }
};

void main( )
{ Shapes *ptr[2] ;           //声明抽象基类指针
  Rectangle rect ;           //声明派生类对象rect
  Circle cir ;               //声明派生类对象cir
  ptr[0] = &rect ;           //抽象基类指针指向Rectangle对象
  ptr[0]->setvalue(5, 8) ;    //设置矩形边长
  ptr[0]->display( ) ;        //调用rect虚成员函数显示矩形面积
  ptr[1] = &cir ;            //抽象基类指针指向Circle类对象
  ptr[1]->setvalue(10) ;      //设置圆形半径
  ptr[1]->display( ) ;        //调用cir虚成员函数显示圆形面积
}

```

程序中类Shapes、Rectangle和Circle属于同一个类族，抽象类Shapes通过纯虚函数为整个类族提供了通用的外部接口。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。我们可以定义非抽象类的对象，同时根据赋值兼容规则，抽象类Shapes类型的指针也可以指向任何一个派生类的对象，通过基类Shapes的指针可以访问到正在指向的派生类Rectangle和Circle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


```

#include <iostream.h>
class A
{public:
    virtual void f( )
    { cout<<"A f"<<endl;}
};
class B:public A
{public:
    void f( )
    { cout<<"B f"<<endl;}
};
class C:public B
{public:
    void f( )
    { cout<<"C f"<<endl;}
};

```

```

void main( )
{
    A a;
    B b;
    C c;
    A *p;
    a.f( );
    b.f( );
    c.f( );
    p = &a;
    p->f( );
    p = &b;
    p->f( );
    p = &c;
    p->f( );
}

```

运行结果:

A f

B f

C f

A f

B f

C f

【例5.14】虚函数的作用

```
#include <iostream.h>
#include <string.h>
class person
{public:
    void setinfo(int n, char* strname, char s, char* strfrom)
    {
        id = n; strcpy(name, strname);
        sex = s; strcpy(from, strfrom);
    }
    person( )
    {
        setinfo(1, "jetty", 'M', "china");
    }
    virtual void showinfo( )
    {
        cout<<endl<<"ID: "<<id
            <<endl<<"name: "<<name
            <<endl<<"sex: "<<sex
            <<endl<<"from: "<<from;
    }
protected:
    int id;      char name[10];      char sex;      char from[20];
};
```

```
class undergraduate:virtual public person           // 虚继承
{public:
    void setinfo(int n, char* strname,
                  char s, char* strfrom, float sc)
    {    person::setinfo(n, strname, s, strfrom);
        score = sc;    }
    undergraduate( )
    {
        setinfo(1, "jetty", 'M', "china", 90); }
    void showinfo( )
    {    person::showinfo( );
        cout<<endl<<"score: "<<score;    }
protected:    float score;
};
```

```

class graduate:public undergraduate, public teacher // 从两个类继承
{public:
    void setinfo(int n, char* strname, char s,
                char* strfrom, float sc, float sa)
    {
        id = n;  strcpy(name, strname);
        sex = s; strcpy(from, strfrom);
        score = sc;    salary = sa;    }
graduate( )
{
    setinfo(1, "jetty", 'M', "china", 90, 1000); }
void showinfo()
{
    cout<<endl<<"ID: "<<id<<endl<<"name: "<<name
        <<endl<<"sex: "<<sex<<endl<<"from: "<<from
        <<endl<<"score: "<<score
        <<endl<<"salary: "<<salary;
    }
};

```

```
void main( )  
{ person* pp;  
  person p;  
  pp = &p;  
  pp->showinfo( );  
  undergraduate u;  
  pp = &u;  
  pp->showinfo( );  
  teacher t;  
  pp = &t;  
  pp->showinfo( );  
  graduate g;  
  pp = &g;  
  pp->showinfo( );  
}
```

ID: 1
name: jetty
sex: M
from: china
ID: 1
name: jetty
sex: M
from: china
score: 90
ID: 2
name: rose
sex: F
from: American
salary: 2000
ID: 1
name: jetty
sex: M
from: china
score: 90
salary: 1000

【例5.15】虚析构函数示例

```
#include <iostream.h>
class base
{
public:
    virtual ~base()
    {cout<<endl
        <<"base destructor"; }
};
class derived:public base
{
public:
    ~derived()
    {cout<<endl
        <<"derived destructor"; }
};
```

```
void main()
{ base *pb = new derived;
  delete pb; }
```

再次编译运行结果如下：
derived destructor
base destructor



【例5.16】纯虚函数和抽象类

```
#include <iostream.h>
class a
{public:
    virtual void f( )=0;
    virtual void g( )=0;
};
class b:public a
{public:
    void f( )
    {cout<<endl<<"b f"; }
};
class c:public b
{public:
    void f( )
    {cout<<endl<<"c f"; }
    void g( )
    {cout<<endl<<"c g"; }
};
```

```
void main( )
{
    a* pa;
    b x;
    c y;
    pa = &y;
    pa->f( );
    pa->g( );
}
```

// error;
由于b类没有实现
纯虚函数g，仍为
抽象类，故不能
生成对象实例

【例5.17】综合举例

```
// person.h
#ifndef _PERSON
#define _PERSON
class person
{
public:
    void setinfo(int n, char* strname, char s, char* strfrom);
    person();
    virtual ~person();           // 虚析构函数
    virtual void showinfo()=0;   // 纯虚函数
protected:
    int id;
    char name[10];
    char sex;
    char from[20];
};
#endif
```


【例5.17】综合举例

```
// person.cpp
#include "person.h"
#include <iostream.h>
#include <string.h>
person::person()
{ setinfo(1, "jetty", 'M', "china"); }
person::~~person()
{
    cout<<endl<<"destruct a person";
}
void person::setinfo(int n, char* strname, char s, char* strfrom)
{
    id = n;
    strcpy(name, strname);
    sex = s;
    strcpy(from, strfrom);
}
```

【例5.17】综合举例

```
// undergraduate.h
#include "person.h"
#ifndef _UNDERGRADUATE
#define _UNDERGRADUATE
class undergraduate:virtual public person           // 虚继承
{
public:
    void setinfo(int n, char* strname, char s, char* strfrom, float sc);
    undergraduate();
    ~undergraduate();
    void showinfo();
protected:
    float score;
};
#endif
```

【例5.17】综合举例

```
// undergraduate.cpp
#include "undergraduate.h"
#include <iostream.h>
undergraduate::undergraduate()
{ cout<<endl<<"construct a undergraduate";
  setinfo(1, "jetty", 'M', "china", 90);}
undergraduate::~~undergraduate()
{ cout<<endl<<"destruct a undergraduate";}
void undergraduate::setinfo(int n, char* strname, char s, char*
    strfrom, float sc)
{ person::setinfo(n, strname, s, strfrom);
  score = sc;}
void undergraduate::showinfo()
{ cout<<endl<<"a undergraduate info"
  <<endl<<"ID: "<<id<<endl<<"name: "<<name
  <<endl<<"sex: "<<sex<<endl<<"from: "<<from
  <<endl<<"score: "<<score;}
```

【例5.17】综合举例

```
// teacher.h
#include "person.h"
#ifndef _TEACHER
#define _TEACHER
class teacher:virtual public person           // 虚继承
{
public:
    void setinfo(int n, char* strname, char s, char* strfrom, float sa);
    teacher();
    ~teacher();
    void showinfo();
protected:
    float salary;
};
#endif
```

【例5.17】综合举例

```
// teacher.cpp
#include "teacher.h"
#include <iostream.h>
teacher::teacher()
{ cout<<endl<<"construct a teacher";
  setinfo(2, "rose", 'F', "American", 2000);}
teacher::~~teacher()
{ cout<<endl<<"destruct a teacher";}
void teacher::setinfo(int n, char* strname, char s, char* strfrom, float
    sa)
{ person::setinfo(n, strname, s, strfrom);
  salary = sa;}
void teacher::showinfo()
{ cout<<endl<<"a teacher info"
    <<endl<<"ID: "<<id<<endl<<"name: "<<name
    <<endl<<"sex: "<<sex<<endl<<"from: "<<from
    <<endl<<"salary: "<<salary;
}
```

【例5.17】综合举例

```
// graduate.h
#include "undergraduate.h"
#include "teacher.h"
#ifndef _GRADUATE
#define _GRADUATE
class graduate:public undergraduate, public teacher
    // 从两个类继承，这两个类都通过虚继承而来
{
public:
    graduate();
    ~graduate();
    void setinfo(int n, char* strname, char s, char* strfrom, float sc,
float sa);
    void showinfo();
};
#endif
```

【例5.17】综合举例

```
// graduate.cpp
#include "graduate.h"
#include <iostream.h>
graduate::graduate()
{ cout<<endl<<"construct a graduate";
  setinfo(1, "jetty", 'M', "china", 90, 1000);}
graduate::~~graduate()
{ cout<<endl<<"destruct a graduate";}
void graduate::setinfo(int n, char* strname, char s, char* strfrom, float
  sc, float sa)
{ person::setinfo(n, strname, s, strfrom);
  score = sc;  salary = sa;}
void graduate::showinfo()
{ cout<<endl<<"a graduate info"
  <<endl<<"ID: "<<id<<endl<<"name: "<<name
  <<endl<<"sex: "<<sex<<endl<<"from: "<<from
  <<endl<<"score: "<<score<<endl<<"salary: "<<salary;
}
```

【例5.17】综合举例

```
#include "person.h"
#include "undergraduate.h"
#include "teacher.h"
#include "graduate.h"
#include <iostream.h>
void main()
{
    person* pp;           // 基类指针
    cout<<"new a undergraduate";
    pp = new undergraduate;    pp->showinfo();// 指向本科生对象
    cout<<endl<<"delete a undergraduate";delete pp;// 释放本科生对象
    cout<<endl<<endl<<"new a teacher";
    pp = new teacher;         pp->showinfo();// 指向教师对象
    cout<<endl<<"delete a teacher";        delete pp;// 释放教师对象
    cout<<endl<<endl<<"new a graduate";
    pp = new graduate;        pp->showinfo();// 指向研究生对象
    cout<<endl<<"delete a graduate";        delete pp;// 释放研究生对象
}
```


【例5.18】综合举例

以Point为基类的点—圆—圆柱体类的层次结构。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。

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。

课后练习

判断题：

1. 一旦将函数声明为虚函数，即使类在重写它时没有将其声明为虚函数，它从该点之后的继承层次结构中仍然是虚函数
2. 派生类不重写其基类虚函数时，可直接继承其基类的虚函数
3. 只要将基类指针指向一个派生类对象，我们就可以用该基类指针访问派生类对象的特有成员
4. 派生类对象可以看成是基类对象，反过来，基类对象也可以看成是派生类对象

5. 试图实例化一个抽象类对象会导致运行期错误
6. 抽象类必须至少有一个纯虚函数
7. 抽象类也可以有数据成员和具体函数，包括构造函数和析构函数
8. 构造函数和析构函数都可以被声明为虚函数
9. 如果基类为抽象类，则不能用基类指针指向其派生类对象
10. 抽象类中所有虚函数都必须声明为纯虚函数
11. 用派生类句柄引用基类对象是安全的。

谢谢大家！