第 6 章 多态性和虚函数

多态性（polymorphism）是面向对象程序设计的重要特征之一。多态性是指发出同样的消息被多个不同类型的对象接收时导致多个不同的行为。利用多态性可以设计和实现一个易于扩展的系统，同时又简便、高效。多态性的实现有三种形式：函数重载、运算符重载、虚函数。关于函数重载，我们在前面第二章已经学习过，本章重点讨论运算符重载和虚函数。

6.1 运算符重载

尽管 C++语言有丰富的数据类型和运算符，但仍然不能满足应用编程的一些需要。复数及其操作就是这样的一个例子。虽然用户可以定义一个复数，然后利用成员函数实现数据之间的运算操作，但没有运算符操作来得更为简便、直接。

运算符重载就是赋予已有的运算符多重含义。C++语言通过重新定义运算符，使它能够用于特定类的对象，执行特定的功能，从而增强了 C++语言的扩充能力。

6.1.1 运算符重载的原则

C++中预定义的运算符的操作对象只能是基本数据类型。但实际上，对于许多用户自定义类型（例如类），也需要类似的运算操作。这时就必须在 C++中重新定义这些运算符，赋予已有运算符新的功能，使它能够用于特定类型执行特定的操作。运算符重载的实质是函数重载，它提供了 C++的可扩展性，形成所谓多态性，是 C++最吸引人的特性之一。

运算符重载是通过创建运算符函数实现的，运算符函数定义了重载的运算符将要进行的操作。运算符函数的定义与其它函数的定义类似，唯一的区别是运算符函数的函数名是由关键字 operator 和其后要重载的运算符符号构成的。运算符函数定义的一般格式如下：

<返回类型说明符> operator <运算符符号>(<参数表>)

{

<函数体>

}

运算符重载时要遵循以下规则：

(1)C++有 5 个运算符不能重载，它们是：类属关系运算符“.”、成员指针运算符“.*”、作用域运算符“::”、测试运算符“sizeof”和三目运算符“?:”。除此之外，C++中的其它运算符都可以重载。

(2)重载运算符限定在 C++语言中已有的运算符范围内并且允许重载的运算符之中，不能创建新的运算符。

(3)运算符重载实质上是函数重载，因此编译程序对运算符重载的选择，遵循函数重载的选择原则。

(4)重载之后的运算符不能改变运算符的优先级和结合性，也不能改变运算符操作数的个数及语法结构。

(5)运算符重载只能和用户自定义类型的对象一起使用，或者用于用户自定义类型的对

象和内部类型的对象混合使用。也就是说，经重载的运算符，其操作数中至少应该有一个是自定义类型。这样，重载的运算符才有意义。

(6)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适当的改造，重载的功能应当与原有功能相类似，避免盲目使用和随意使用重载运算符。

运算符重载一般有两种形式：重载为类的成员函数和重载为类的非成员函数。

6.1.2 用成员函数重载运算符

运算符重载为类的成员函数的一般格式为：

```
<函数类型> operator <运算符> (<参数表>)
{
  <函数体>
}
```

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要少一个（后置单目运算符除外），这是因为成员函数用 this 指针隐式地访问了类的一个对象，它充当了运算符函数最左边的操作数。因此

(1)双目运算符重载为类的成员函数时，函数只显式说明一个参数，该形参是运算符的右操作数。

(2)前置单目运算符重载为类的成员函数时，不需要显式说明参数，即函数没有形参。

(3)后置单目运算符重载为类的成员函数时，函数要带有一个整型形参,以便与同一单目运算符的前置单目运算符相区别。

调用成员函数运算符的格式如下：

```
<对象名>.operator <运算符>(<参数>)
```

它等价于

```
<对象名><运算符><参数>
```

例如：a+b 等价于 a.operator +(b)。一般情况下，我们采用运算符的习惯表达方式。

例 6.1 将“+”和“-”运算符重载为复数类的成员函数。

这是一个双目运算符重载为类的成员函数的例子，这里两个操作数都是复数类的对象。复数加、减法的运算规则是:实部和实部相加、减，虚部和虚部相加、减。根据用成员函数重载运算符的原则，重载的运算符函数只有一个参数。本例的源程序如下：

```
//6_1.cpp
#include<iostream>
using namespace std;
class complex //复数类声明
{
  public: //外部接口
    complex(double r=0.0,double i=0.0) {real=r; imag=i;} // 构造函数
    complex operator + (complex c2); //+运算符重载为成员函数
    complex operator - (complex c2); //-运算符重载为成员函数
    void display(); //输出复数
  private:      //私有数据成员
    double real;    //复数实部
```

```
    double imag;   //复数虚部
};

complex complex::operator +(complex c2) //+运算符重载函数实现
{
    complex c;
    c.real=real+c2.real;
    c.imag=imag+c2.imag;
    return c;
}

complex complex::operator -(complex c2)   //-运算符重载函数实现
{
    complex c;
    c.real=real-c2.real;
    c.imag=imag-c2.imag;
    return c;
}

void complex::display(){
   cout<<"("<<real<<","<<imag<<")"<<endl;
}

void main()        //主函数
{   complex c1(5,6),c2(1,2),c3;   //生成复数类的对象
    cout<<"c1="; c1.display(); //显示 c1
    cout<<"c2="; c2.display(); //显示 c2
    c3=c1+c2;      //使用重载运算符完成复数加法
    cout<<"c3=c1+c2=";
    c3.display(); //显示复数相加运算结果
    c3=c1-c2;      //使用重载运算符完成复数减法
    cout<<"c3=c1-c2=";
    c3.display(); //显示复数相减运算结果
    return 0;
}
```

程序输出结果为：

```
c1=(5,6)
c2=(1,2)
c3=c1+c2=(6,8)
c3=c1-c2=(4,4)
```

例 6.2 将单目运算符++重载为时钟类的成员函数。

这是一个将运算符前置++和后置++重载为时钟类的成员函数的例子。根据前述原则，对于前置单目运算符，重载函数没有形参；对于后置单目运算符，重载函数需要有一个整型形参以示区别。操作数是时钟类的对象，实现时间增加 1 秒钟。该实例源程序如下：

```
//6_2.cpp
#include<iostream>
using namespace std;
class Clock     //时钟类声明
{
  public: //公有函数成员
        Clock(int NewH=0, int NewM=0, int NewS=0);
        void ShowTime();   //显示时间函数
        Clock&   operator ++();   //前置单目运算符重载
        Clock operator ++(int);   //后置单目运算符重载
  private:      //私有数据成员
        int Hour,Minute,Second;
};

Clock& Clock::operator ++() //前置单目运算符重载函数
{     Second++;
      if (Second>=60)
      {     Second=Second-60;
            Minute++;
            if (Minute>=60)
            {
                  Minute=Minute-60;
                  Hour++;
                  Hour=Hour%24;
            }
      }
      return *this;//通过 this 指针引用 Clock 对象并返回对象值。
}

//后置单目运算符重载
Clock Clock::operator ++(int)       //形参表中有一整型参数
{   Clock old=*this;
    ++(*this);
    return old;返回值与 Clock 对象值不同步，非引用。
}

//其它成员函数的实现请读者自己完成

void main()
```

```
{
    Clock myClock(23,59,59);
    cout<<"First time output:";
    myClock.ShowTime();
    cout<<"Show myClock++:";
    (myClock++).ShowTime();
    cout<<"Show ++myClock:";
    (++myClock).ShowTime();
}
```

程序运行结果为：

```
First time output: 23:59:59
Show myClock++: 23:59:59   //先引用并显示：23:59:59；后加变为：0:0:0
Show ++myClock: 0:0:1   //先加变为：0:0:1，同时显示出来
```

6.1.3 用友元函数重载运算符

也可以将运算符重载为类的非成员函数。如果需要访问运算符参数对象的私有成员，则可以通过将运算符重载为类的友元函数来达到目的，否则，不必将运算符重载为类的友元函数，只需将运算符重载为类的非成员函数即可。运算符成员函数用于类内访问私有数据，而友元函数用于类外的私有访问。运算符重载为类的友元函数的一般格式为：

```
friend <函数类型> operator <运算符>(<参数表>)
{
<函数体>
}
```

当运算符重载为类的友元函数时，由于没有隐含的 this 指针，因此操作数的个数没有变化，所有的操作数都必须通过函数的形参进行传递。在传递顺序上，函数的参数与运算符操作数自左至右一一对应。

调用友元函数运算符的格式如下：

```
operator <运算符>(<参数 1>,<参数 2>)
```

它等价于

```
<参数 1><运算符><参数 2>
```

例如：a+b 等价于 operator +(a,b)。

例 6.3 以友元函数形式重载 Complex 类的+、-运算符。

请注意，友元函数因为没有 this 指针，所以形参会多一个。

```
//6_3.cpp
#include<iostream>
using namespace std;
class complex //复数类声明
{ public:  //外部接口
    complex(double r=0.0,double i=0.0) {real=r; imag=i;} //构造函数
    friend complex operator +(complex c1,complex c2);//+运算符重载为友元函数
```

```
    friend complex operator -(complex c1,complex c2);//-运算符重载为友元函数
    void display(); //输出复数
  private:      //私有数据成员
    double real;    //复数实部
    double imag;   //复数虚部
};

complex complex::operator +(complex c1,complex c2) //+运算符重载函数实现
{
    complex c;
    c.real=c1.real +c2.real; //不同于运算符重载为类的成员函数
    c.imag=c1.imag +c2.imag; //不同于运算符重载为类的成员函数
    return c;
}

complex complex::operator -(complex c1,complex c2) //-运算符重载函数实现
{
    complex c;
    c.real=c1.real-c2.real; //不同于运算符重载为类的成员函数
    c.imag=c1.imag-c2.imag; //不同于运算符重载为类的成员函数
    return c;
}

void complex::display()
{
   cout<<"("<<real<<","<<imag<<")"<<endl;
}

void main()        //主函数
{    complex c1(5,6),c2(1,2),c3;   //生成复数类的对象
    cout<<"c1="; c1.display(); //显示 c1
    cout<<"c2="; c2.display(); //显示 c2
    c3=c1+c2;      //使用重载运算符完成复数加法
    cout<<"c3=c1+c2=";
    c3.display(); //显示复数相加运算结果
    c3=c1-c2;      //使用重载运算符完成复数减法
    cout<<"c3=c1-c2=";
    c3.display(); //显示复数相减运算结果
    return 0;
}
```

6.1.4 几个常用运算符的重载

c++的一大特性就是重载(overload)，通过重载可以把功能相似的几个函数合为一个，取同一个函数名，使得程序更加简明、高效。下面，我们列出一些常用运算符的重载，以便对C++中常用运算符的重载有一个概貌性的了解。

1．一般运算符重载

```
class A
{
  public:
      A(int d):data(d){}
      A operator+(A&);//成员函数
      A operator-(A&);
      A operator*(A&);
      A operator/(A&);
      A operator%(A&);
      friend A operator+(A&,A&);//友元函数
      friend A operator-(A&,A&);
      friend A operator*(A&,A&);
      friend A operator/(A&,A&);
      friend A operator%(A&,A&);
  private:
      int data;
};
//成员函数的形式
A A::operator + (A &a)
{
   return A(data+a.data);
}

A A::operator - (A &a)
{
   return A(data-a.data);
}

A A::operator * (A &a)
{
   return A(data*a.data);
}

A A::operator / (A &a)
{
   return A(data/a.data);
}
```

```
A A::operator % (A &a)
{
    return A(data%a.data);
}
//友元函数的形式
A operator + (A &a1,A &a2)
{
    return A(a1.data+a2.data);
}

A operator - (A &a1,A &a2)
{
    return A(a1.data-a2.data);
}

A operator * (A &a1,A &a2)
{
    return A(a1.data*a2.data);
}

A operator / (A &a1,A &a2)
{
    return A(a1.data/a2.data);
}

A operator % (A &a1,A &a2)
{
    return A(a1.data%a2.data);
}

//然后就可以对类的对象进行+、-、*、/、%运算了。
void main(void)
{
    A a1(66),a2(55),a3;
    a3=a1+a2; //或者 a3=a1.operator+(a2);
}
```

注意：在进行 a1+a2 的时候会出错，因为上面对+运算符定义了两种方法，实际使用中选择其中之一即可。

2.关系运算符重载

关系运算符有==,!=,<,>,<=,>=。由于函数体比较简单，后面我们只给出成员函数形式的函数声明。

```
bool operator == (const A& );
bool operator != (const A& );
```

```
bool operator <= (const A& );
bool operator >= (const A& );
bool operator < (const A& );
bool operator > (const A& );
```

3.逻辑运算符重载

```
bool operator || (const A& );
bool operator && (const A& );
bool operator ! ();
```

4.单目运算符重载

这里的+、-是正负的意思，放在对象前面。

```
A& operator + ();
A& operator - ();
A* operator & ();
A& operator * ();
```

5.自增、自减运算符重载

++和--根据位置的不同有四种情况，都可以重载。

```
A& operator ++ ();    //前置++
A operator ++ (int);//后置++
A& operator --();     //前置--
A operator -- (int);//后置--
```

6.位运算符重载

按位操作。

```
A operator | (const A& );
A operator & (const A& );
A operator ^ (const A& );
A operator ~ ();
A operator << (int i);
A operator >> (int i);
```

7.赋值运算符重载

```
A& operator += (const A& );
A& operator -= (const A& );
A& operator *= (const A& );
A& operator /= (const A& );
A& operator %= (const A& );
A& operator &= (const A& );
A& operator |= (const A& );
A& operator ^= (const A& );
A& operator <<= (int i);
A& operator >>= (int i);
```

8.特殊运算符重载

上面的运算符重载都有两种方式，而下面的运算符只能用一种，这些运算符的重载只能是成员函数。

```
A& operator = (const A&);
char operator [] (int i);//返回值不能作为左值
const char* operator () ();
T operator -> ();

//类型转换符
operator char* () const;
operator int ();
operator const char () const;
operator short int () const;
operator long long () const

而<<和>>运算符只能以友元函数的形式重载
friend inline ostream & operator << (ostream&, A&);
friend inline istream & operator >> (istream&, A&);
```

应该指出，我们完全可以把一个运算符作为一个非成员、非友元函数重载。但是，这样的运算符函数访问类的私有和保护成员时，必须使用类的公有接口中提供的设置数据和读取数据的函数，调用这些函数时会降低性能。可以内联这些函数以提高性能。

例 6.4 点、圆、圆柱体类的继承与派生举例。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。

对于一个比较大的程序，应当分成若干步骤进行。先声明基类，再声明派生类，逐级进行，分步调试。

(1) 声明基类 Point 类

```
//6_4.cpp
#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

```
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;
};
//定义构造函数，对圆心坐标和半径初始化
Circle::Circle(float a, float b, float r):Point(a,b),radius(r){ }
void Circle::setRadius(float r)     //设置半径值
{radius=r;}
float Circle::getRadius( ) const    //读取半径值
```

```
{return radius;}
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: \\n x=″<<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){}
void Cylinder::setHeight(float h){height=h;}              //设置圆柱高
float Cylinder::getHeight( ) const {return height;}   //读取圆柱高
float Cylinder::area( ) const
{ return 2*Circle::area( )+2*3.14159*radius*height;} //计算圆柱表面积
float Cylinder::volume() const                              //计算圆柱体积
{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     <<"\\n
     area="<< 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<<"\\n original cylinder:\\n x="<<cy1.getX( )<<", y="  <<cy1.getY()<<",
  r="<<cy1.getRadius()<<",                h="<<cy1.getHeight()                <<"\\n
  area="<<cy1.area()<<",volume="<<cy1.volume()<<endl;
  //以上是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<<"\\n new 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<<"\\n pRef 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<<"\\n cRef 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。

6.2 虚函数

6.2.1 为什么要引入虚函数

众所周知，在同一类中是不能定义两个名字相同、参数个数和类型都相同的函数的，否则就是“重复定义”。但是在类的继承层次结构中，在不同的层次中可以出现名字相同、参数个数和类型都相同而功能不同的函数。

在继承关系中，从 this 指针角度出发，可以这样考虑：指向基类和子类的指针，在运行时类型是不同的（分别指向不同的类）。 但是在编译时，他们都要指向基类的类型（从函数执行顺序可以看出，在子类初始化时，首先要进入基类的构造函数），去继承基类的 public 方法，在遇到“virtual”关键字时，“编译指针”会自动跳到对应子类中的函数实现处，而不是直接跳到基类中的函数实现。所以，子类中的方法就覆盖了基类中的方法。基类中的方法对子类形同虚设，就是虚函数。也可以理解为：“虚函数”即“发函数”。通过把函数“发虚”、“发胖”，原来作用单一、功能较少、功能不全的“基本函数”，就变成了作用更强、功能较多较全的“函数族”。

引入虚函数的目的就是为了让指向派生类对象的基类指针可以调用覆盖了基类虚方法的对应子类方法。运用抽象手法，把不同的子类对象的共同方法抽取出来，形成虚函数，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的编程，以适应需求的不断变化。赋值之后，父类指针就可以根据当前赋值给它的子对象的特性以不同的方式运作，从而形成动态多态性。

6.2.2 虚函数的定义与使用

1．虚函数的定义

虚函数就是在类定义中的函数原型声明前加一个关键字 virtual。在派生类中重新定义时，虚函数的函数原型（包括返回类型、函数名、参数个数和参数类型）必须与基类中的虚函数完全相同，否则编译器会认为派生类有两个重载函数。

即申明虚函数的方法是：只要在函数声明的原型前面加上 “virtual”，基类中该函数的定义可以保留 virtual，也可以删掉它。

2．虚函数的使用方法

(1) 在基类中用 virtual 声明成员函数为虚函数。这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必在定义中重复使用 virtual。

(2) 在派生类中重新定义此函数，要求函数名，函数类型，函数参数的个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。c++规定，当一个成员函数被声明为虚函数后，其派生类的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加 virtual，也可以不加，但习惯上一般在每层声明该函数时都加上 virtual，使程序更加清晰。如果在派生类中没有对基类的虚函数重新定义，则派生类简单的继承其基类的虚函数。

(3) 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。

(4) 通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。

下面看一个例子。

例 6.5 虚函数举例

```
//6_5.cpp
#include<iostream>
#include<string>
using namespace std;
class Student
{
    public:
        Student(int, string, float);
        void display();
    protected:
        int num;
        string name;
        float score;
};
Student::Student(int n, string na, float s)
{
    num=n;
    name=na;
    score=s;
}
void Student::display()
{
        cout<<"num:"<<num<<"\n name:"<<name<<"\n score:"<<score<<endl;
}
class Graduate:public Student
{
    public:
            Graduate(int, string, float, float);
            void display();
    private:
            float pay;
};
Graduate::Graduate(int n, string na, float s, float p):Student(n,na,s),pay(p)
{
}
void Graduate::display()
{
    cout<<"num:"<<num<<"\n name:"<<name<<"\n score:"<<score<<"\n pay:"<<pay
    <<endl;
}

int main()
{
    Student stud1(110,"SDL",99.8);
```

```
    Graduate grad1(120,"CCT",88.9, 250.5);
    Student *p;
    p=&stud1;
    cout<<"student:"<<endl;
    p->display();
    p=&grad1;
    cout<<"graduate:"<<endl;
    p->display();
    return 0;
}
```

输出结果如图 6.1 所示：

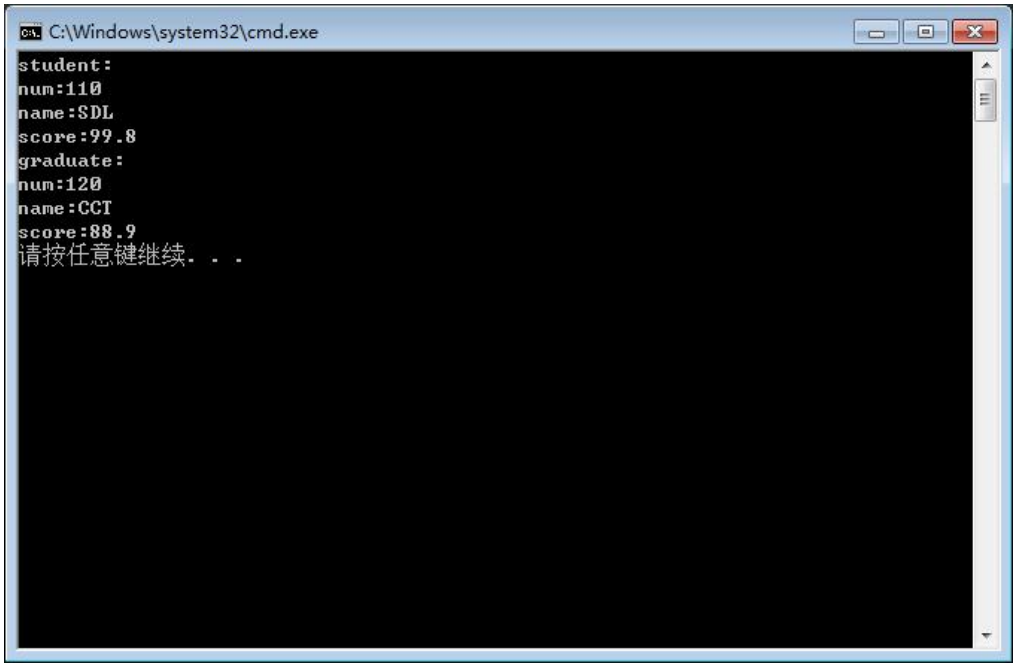

图 6.1 未声明虚函数时的程序输出结果

此时没有将任何函数声明为虚函数，所以在 Graduate 类中 diaplay（）函数输出时没有 pay 的值，因为 p 指针是 Student 类的， Student 本身不包括 pay 数据。

但如果我们想输出 pay 的值，只要把 Student 类中的 display 函数声明为虚函数即可，只要把它改为 virtual void display()声明为虚函数就行，输出的结果如图 6.2 所示：

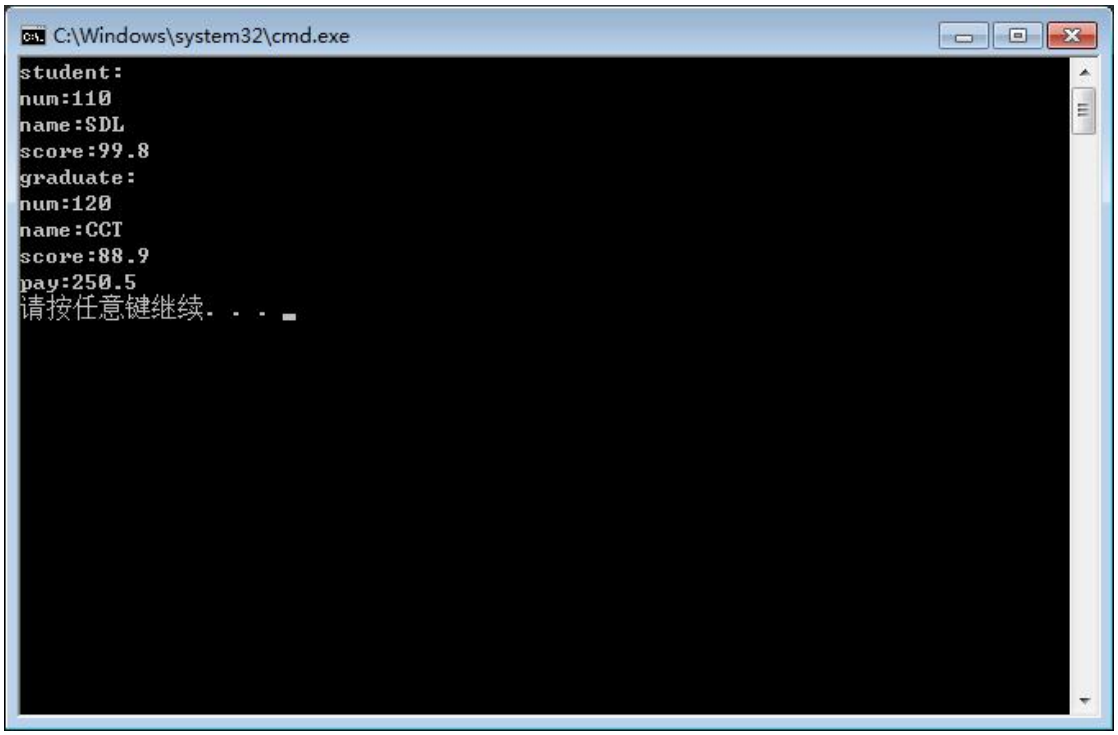

图 6.2 声明了虚函数时的程序输出结果

在使用虚函数时，必须注意以下两个问题。

⑴在派生类中重新定义虚函数时它的原型必须与基类中的虚函数完全相同，否则编译器会把它认为是重载函数，而不是虚函数的重定义。

⑵派生类在对基类的虚函数重定义时，关键字 virtual 可以写也可以不写。不管 virtual 写还是不写，该函数都被认为是虚函数，但最好是在重定义时写上 virtual。

6.3 纯虚函数和抽象类

有时我们在基类中将某一成员函数设定为虚函数，并非基类本身的要求，而是考虑到派生类的需要，在基类中预留一个函数名，具体功能留给派生类根据自己的实际需要去定义。例如，因为“点”（Point）没有面积，因此基类 Point 中没有求面积的 area 函数，即基类本身不需要这个函数，Point 类中可以不定义这个函数。但在其直接派生类圆 Circle 和间接派生类圆柱体 Cylinder 中都需要有 area 函数，一个求圆面积，一个求圆柱体表面积，两个 area 函数的功能有所不同。在这种情况之下，就会用到纯虚函数(pure virtual function)。

6.3.1 纯虚函数的概念

纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
class <类名>
{
    virtual <类型><函数名>(<参数表>)=0;
    …
```

```
};
```

纯虚函数是在声明虚函数时被“初始化”为 0 的函数。纯虚函数没有函数体，只有函数声明，在虚函数声明结尾加上=0，表明此函数为纯虚函数。

下面举一个例子。

例 6.6 纯虚函数举例

```
//6_6.cpp
#include
class point
{
   public:
      point(int i=0, int j=0) { x0=i; y0=j; }
      virtual void set() = 0;
      virtual void draw() = 0;
   protected:
      int x0, y0;
};
class line : public point
{
   public:
      line(int i=0, int j=0, int m=0, int n=0):point(i, j)
      {
            x1=m; y1=n;
      }
      void set() { cout<<"line::set() called.\n"; }
      void draw() { cout<<"line::draw() called.\n"; }
   protected:
      int x1, y1;
};
class ellipse : public point
{
   public:
      ellipse(int i=0, int j=0, int p=0, int q=0):point(i, j)
      {
            x2=p; y2=q;
      }
      void set() { cout<<"ellipse::set() called.\n"; }
      void draw() { cout<<"ellipse::draw() called.\n"; }
   protected:
      int x2, y2;
};
void drawobj(point *p)
{
      p->draw();
```

```
}
void setobj(point *p)
{
    p->set();
}
void main()
{
    line *lineobj =new line;
    ellipse *elliobj =new ellipse;
    drawobj(lineobj);
    drawobj(elliobj);
    setobj(lineobj);
    setobj(elliobj);
    cout<<"\n Redraw the object...\n";
    drawobj(lineobj);
    drawobj(elliobj);
}
```

运行结果如下（见图 6.3）：

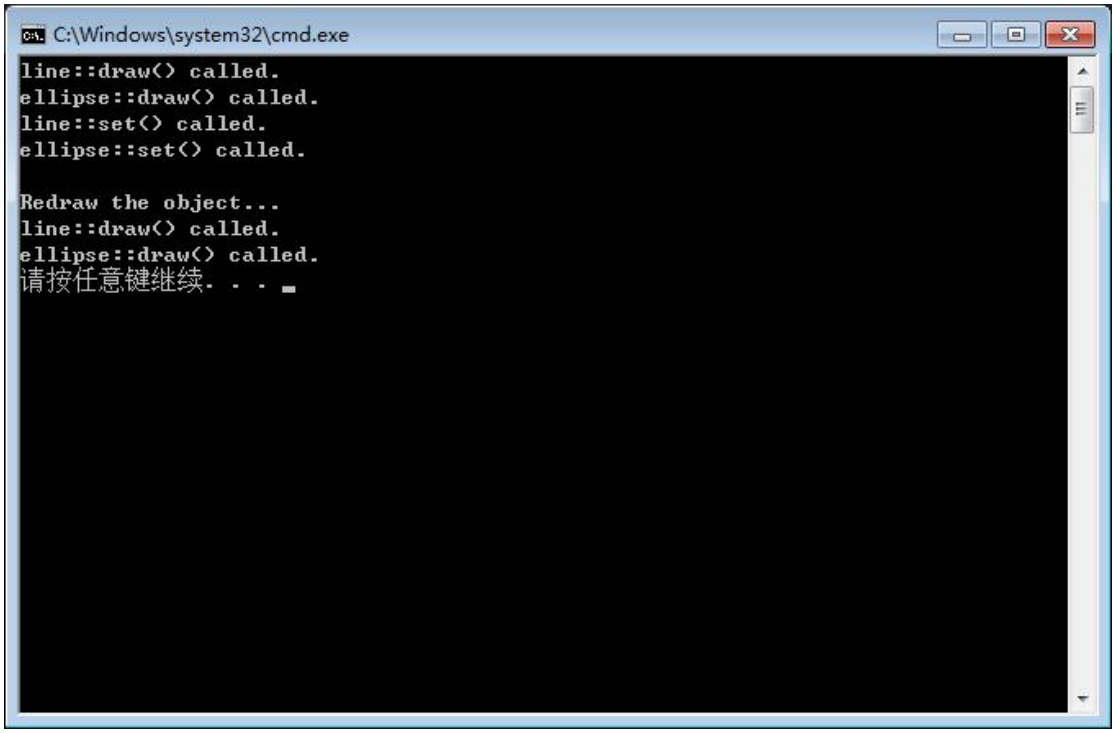

图 6.3 例 6.6 的程序输出结果

关于纯虚函数，我们强调以下几点:

(1) 纯虚函数没有函数体。

(2) 纯虚函数声明语句最后面的“=0”并不表示函数返回值为 0，它只起形式上的作用，告诉编译系统，这是一个纯虚函数。

(3) 纯虚函数声明语句的最后应有分号。

(4) 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。它仅仅是通知编译系统:“此处声明一个虚函数，将在派生类中定义”。只有在派生类中对此函数定义后，它才能具备函数功能，可被调用。

(5) 纯虚函数是实现事物多态性的又一种形式，它的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。如果在基类中没有保留函数名字，则无法实现多态性。

(6) 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。

(7) 只有类中的虚函数才能被声明为纯虚成员函数，普通成员函数和顶层函数均不能声明为纯虚函数。

6.3.2 抽象类的概念

一般来说，声明一个类，就是为了使用它，用它定义具体的对象，然后通过对象对数据、函数等进行实际操作。但是在 C++中，也常常有一些类，它们不用来生成对象。但这并不意味着这些类无用，相反，这种类的作用也是非常重要的，声明它们的目的也非常明确。声明这种类目的就是用它作为基类去建立派生类。它们作为一种基本类型提供给程序员，然后由程序员在这个基础上根据自己的需要定义出其它功能各异的派生类，再用这些派生类去建立各种实际对象。

一个优秀的软件工程师在开发一个大型软件时，一般不会从头至尾逐句逐行由自己编写程序代码，他会充分利用类库、开源代码、实例等已有资源作为自己工作的基础。

上述那种不用来定义对象而只作为一种基本类型用作传承的类，称为抽象类(abstract class)，也称为抽象基类(abstract base class)。

凡是包含纯虚函数的类都是抽象类。纯虚函数不能被调用，包含纯虚函数的类无法建立对象。抽象类的作用是作为一个类族的共同基类，或者说，为一个类族提供一个公共接口。

如果在抽象类所派生出的新类中对基类的所有纯虚函数进行了定义，那么这些函数就有了确定的功能，可以被调用。这个派生类就不再是抽象类，而是变为可以用来定义对象的具体类(concrete class)。如果在派生类中没有对所有纯虚函数进行定义，那么该派生类仍然是抽象类，不能用来定义对象。

虽然抽象类不能定义对象，不能实例化，但是可以定义指向抽象类数据的指针变量。当派生类成为具体类之后，就可以用这种指针指向派生类对象，然后通过该指针调用虚函数，实现多态性操作。

应该指出，虽然一个类层次结构中可以不包含任何抽象类，每一层次的类都是实际可用的，可以用来建立对象的，但这并不表明抽象类是不重要的、可有可无的。相反，我们必须强调，许多好的面向对象的系统，尤其是中、大型 OO 系统，其层次结构的顶部都是一个抽象类，甚至顶部有好几层都是抽象类。

例 6.7 纯虚函数和抽象类举例一

```
//6_7.cpp
#include <iostream>
using namespace std;
class B                                          //声明抽象基类 B
{ public:                                        //外部接口
    virtual void display( )=0;                   //纯虚函数
```

```
};
class C: public B                                    //公有派生
{ public:
    void display(){cout<<"C::display()"<<endl;}   //虚成员函数
};
class D: public C     //公有派生
{ public:
    void display(){cout<<"D::display()"<<endl;}   //虚成员函数
};
void fun(B *ptr)                                     //普通函数
{    ptr->display();    }
void main()                                          //主函数
{    B *p;                                           //声明抽象基类指针
    C c;                                             //声明派生类对象
    D d;                                             //声明派生类对象
    p=&c;
    fun(p);                                          //调用派生类 C 的函数成员
    p=&d;
    fun(p);                                          //调用派生类 D 的函数成员
}
```

运行结果：

```
C::display()
D::display()
```

例 6.8 虚函数和抽象类举例二。

现在我们再举一个典型例子，说明虚函数和抽象类的应用。在这个例子中，由 Shape(形状)、Point(点), Circle(圆), Cylinder(圆柱体)组成类的层次结构。该层次结构的顶层是抽象类 Shape，而 Point 类, Circle 类, Cylinder 类都是 Shape 类的直接派生类或间接派生类。在此基础上，可以求出这几个基本几何形状的面积、体积。

下面是完整的程序。为了层次清楚起见，程序分为 3 个部分。

```
//6_8.cpp
//PART ONE
#include <iostream>
using namespace std;
//声明抽象类 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
};
// PART TWO
//声明 Point 类
```

```
class Point:public Shape   //Point 是 Shape 的公有派生类
{public:
   Point(float=0,float=0);     //声明带默认形参值的构造函数
   void setPoint(float,float); //声明成员函数 setPoint()
   float getX( ) const {return x;} //定义成员函数 getX( )
   float getY( ) const {return y;} //定义成员函数 getY( )
   virtual void shapeName( ) const {cout<<″Point:″;} //对虚函数进行再定义
   friend ostream & operator<<(ostream &, const Point &); //声明友元函数
protected:
   float x,y;
};
//定义 Point 类成员函数
Point::Point(float a,float b) //定义构造函数
{x=a;y=b;}
void Point::setPoint(float a,float b) //定义成员函数 setPoint()
{x=a;y=b;}
ostream & operator<<(ostream &output,const Point &p)//定义友元函数—
//该函数是一个重载运算符函数（插入运算符“<<”一次重载）
{output<<″[″<<p.x<<″,″<<p.y<<″]″;
return output;
}
//PART THREE
//声明 Circle 类
class Circle:public Point   //声明派生类 Circle
{public:
   Circle(float x=0,float y=0,float r=0);
   void setRadius(float);
   float getRadius( ) const;
   virtual float area( ) const;   //继承得来的虚函数
   virtual void shapeName( ) const {cout<<″Circle:″;}//对纯虚函数进行再定义
   friend ostream &operator<<(ostream &,const Circle &);//声明友元函数—
//该函数是一个重载运算符函数（插入运算符“<<”二次重载）
protected:
   float radius;
};
//定义 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) //定义友元函数—
//该函数是一个重载运算符函数（插入运算符“<<”二次重载）
{output<<″[″<<c.x<<″,″<<c.y<<″], r=″<<c.radius;
 return output;
```

```
}
//PART FOUR
//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&);//声明友元函数—
//重载运算符函数（插入运算符"<<"三次重载，注意参数类型不同）
 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) //定义友元函数—
//重载的运算符函数<<（插入运算符"<<"三次重载）
{output<<"["<<cy.x<<","<<cy.y<<"], r="<<cy.radius<<", h="<<cy.height;
return output;
}
//PART FIVE
//main 函数
int main( )
{Point point(3.2,4.5);                    //建立 Point 类对象
 Circle circle(2.4,1.2,5.6);              //建立 Circle 类对象
 Cylinder cylinder(3.5,6.4,5.2,10.5); //建立 Cylinder 类对象
 point.shapeName();                       //静态绑定,显示对象名
 cout<<point<<endl;                       //显示对象数据
 circle.shapeName();                      //静态绑定,显示对象名
 cout<<circle<<endl;                      //显示对象数据
 cylinder.shapeName();                    //静态绑定,显示对象名
 cout<<cylinder<<endl<<endl;              //显示对象数据
 Shape *pt;                               //定义基类指针
 pt=&point;                               //指针指向 Point 类对象
 pt->shapeName( );                        //动态绑定
 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( );                          //动态绑定
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( );                        //动态绑定
  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]
Circle:[2.4,1.2], r=5.6
Cylinder:[3.5,6.4], r=5.2, h=10.5
Point:x=3.2,y=4.5
area=0                    //点的面积
volume=0                  //点的体积
Circle:x=2.4,y=1.2        //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 // cylinder 的数据: 圆心坐标)
area=512.959              //圆的面积
volume=891.96             //圆柱的体积
```

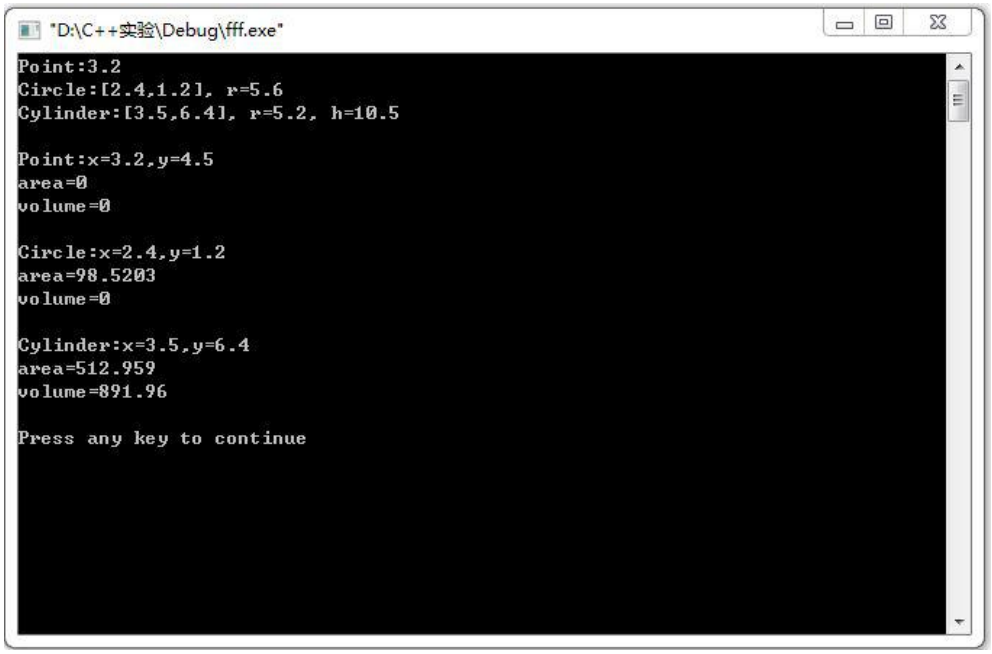

图 6.4 例 6.8 的程序运行结果

最后，我们对抽象类再做一个简单的概括。

(1) 带有一个或多个纯虚函数的类称为抽象类。抽象类是一种特殊的类，它是为了抽象和设计的目的而建立的，它处于继承层次结构的上层。

(2) 抽象类是不能定义对象的，在实际中为了强调一个类是抽象类，可将该类的构造函数说明为保护的访问控制权限。

(3) 抽象类的主要作用是将有关的子类抽象地组织在一个继承层次结构中，由它来为类族中的各子类提供一个公共的根，相关的子类是从这个根派生出来的。

(4) 抽象类刻画了一组子类的操作接口的通用语义，这些语义也传给子类，使得从同一基类派生出的多个类有同一接口。一般而言，抽象类只描述这组子类共同的操作接口，而完整的实现留给子类。

(5) 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类只是继承基类的纯虚函数，没有重新定义纯虚函数，那么这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类，而是一个可以建立对象的具体类了。

(6) 抽象基类与普通基类不同，它并不一定是现实存在的对象的抽象，它可以没有任何物理上的或其它实际意义方面的含义，可以是从设计角度预留的一个“空基类”。

(7) 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。

(8) 使用抽象类提高了程序的可扩充性。

6.4 虚析构函数

在 C++中，不能声明虚构造函数，但是可以声明虚析构函数。虚析构函数没有类型，也没有参数。

虚析构函数的声明语法为：

virtual ~类名( );

如果一个类的析构函数是虚函数,那么由它派生出来的所有子类的析构函数也是虚函数。析构函数设置为虚函数之后，在使用指针引用时可以动态绑定，实现运行时的多态，保证使用基类类型的指针就能够调用适当的析构函数针对不同的对象进行清理工作。

简单地说，如果有可能通过基类指针调用对象的析构函数（通过 delete），就需要让基类的析构函数成为虚函数，否则会产生不确定的后果。

虚析构函数可以满足让基类的指针指向派生类对象，并用基类的指针删除派生类对象的实际需要。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，将析构函数设为虚函数一般是不好的。因为它会为类增加一个虚函数表，使得对象的体积翻倍，还有可能降低程序代码的可移植性。所以无故声明虚析构函数和永远不去声明虚析构函数同样是错误的。实际上，很多人这样总结:当且仅当类里包含至少一个虚函数时才去声明虚析构函数。抽象类是准备用作基类的，基类必须要有一个虚析构函数，纯虚函数会产生抽象类，所以方法很简单:在想要成为抽象类的类里声明一个纯虚析构函数。

下面是一个使用虚析构函数的例子:

例 6.9 虚析构函数举例。

```
//6_9.cpp
#include<iostream>
```

```
using namespace std;
class Base        //声明基类
{
public:
    ~Base();   //析构函数声明
};
Base::~Base()   //析构函数定义
{
    cout<<"Base destructor"<<endl;
}
class Derived:public Base //声明派生类
{
  public:
    Derived();            //声明构造函数
    ~Derived();           //声明析构函数
  private:
    int *p;               //派生类私有数据成员
};
Derived::Derived()        //定义构造函数
{
    p=new int(0);
}
Derived::~Derived()       //定义析构函数
{
    cout<<"Derived destructor"<<endl;
    delete p;
}
void fun(Base *b)         //定义普通函数 fun
{
    delete b;
}
int main()
{
    Base*b=new Derived(); //定义基类指针 b 并赋值
    fun(b);               //调用函数 fun
    return 0;
}
```

运行输出结果为：

```
Base destructor
```

结果表明，通过基类指针删除派生类对象，调用的是基类的析构函数，派生类的析构函数没有被执行，因此派生类对象中动态分布的内存空间将不会得到释放，从而造成“内存泄漏”。也就是说派生类对象成员 p 所指向的内存空间，在对象消失后既不能被本程序继续使

用也没有被释放。对于内存需求量较大、长期连续运行的程序来说，如果持续发生这样的错误是很危险的，由此可能导致内存不足而引起程序非正常终止。

避免上述错误的有效方法就是将析构函数声明为虚函数：

```
class Base{
public :
virtual ~Base();
};
```

运行时的输出信息变为：

```
Derived destructor
Base destructor
```

这说明由于使用了虚析构函数，实现了多态性，使得派生类的析构函数被调用了，派生类对象中动态申请的内存空间正确地得到了释放。

6.5 综合实例

在实际工作中，有时我们会遇到求函数的定积分问题。当被积函数只有离散值、不便用数学公式表达或无法用初等函数精确表达时，通常采用某种近似方法求解该函数的定积分。其中，变步长梯形积分算法就是一种利用计算机求解函数定积分的简便方法。下面介绍这种方法，并结合一个综合实例，进行较为详细的讨论。

6.5.1 变步长梯形积分算法

设被积函数 $f(x)$ 是一个一元函数，其定积分 I 表示公式如下：

$$I = \int_a^b f(x)dx \tag{6.1}$$

函数 $f(x)$ 的积分的几何意义是 $f(x)$ 在区间[a, b]之间与 x 轴所夹的面积，如图 6.5 所示。

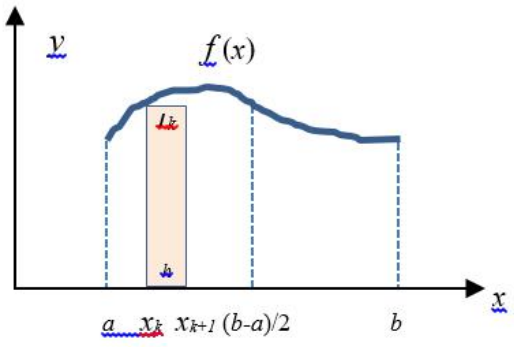

图 6.5 梯形积分图示

对于任意正整数 $n(n\geq 0)$，将区间$[a, b]$等分为 2^n 个小区间，每个小区间的长度 $h=(b-a)/2^n$，则任一分点可表示为：

$$x_k = a + kh,\ x_k \in [a,b], k = 0,1,\cdots,2^n \tag{6.2}$$

函数 $f(x)$在区间$[x_k, x_{k+1}]$的积分为：

$$I_k = \int_{x_k}^{x_{k+1}} f(x)dx \tag{6.3}$$

函数 $f(x)$在区间$[a, b]$的积分为：

$$I = \sum_{k=1}^{2^n} I_k \tag{6.4}$$

如果在每一个小区间$[x_k, x_{k+1}]$上，取 I_k 的某种近似值，则可求得 I 的某种近似值。

现在我们采用梯形近似积分法，即将每一个小区间看作一个梯形小区间，每个小区间的面积为：

$$I_k = \frac{[f(x_k) + f(x_{k+1})]h}{2} \tag{6.5}$$

当 n 足够大，小区间足够小时，我们就得到了符合一定精度要求的函数 $f(x)$在区间$[a, b]$的积分近似值 T_n，可表示为：

$$T_n = \sum_{k=1}^{2^n} [f(x_k) + f(x_{k+1})]h/2 \tag{6.6}$$

为了兼顾计算精度和计算时间两方面的需要，通常采取迭代计算的方法，即“走一步，看一步”，“边走边看”，当不满足精度要求时，采用二分法继续将区间分小，计算二分之后每个小区间的积分值，同时进行新的精度测试，如仍不满足要求，则继续迭代计算下去。一旦某次迭代计算并测试之后，达到了预设的精度要求，则算法终止，此时即得到了所要求的计算结果。

原区间$[x_k, x_{k+1}]$经过二分之后，增加了一个分点 $x_{k+0.5}=[x_k+x_{k+1}]/2$。二分之后，原区间长度 h 变小，新的区间长度 $h_1=h/2$，区间$[x_k, x_{k+1}]$上的积分值变为

$$\frac{[f(x_k) + 2f(x_{k+0.5}) + f(x_{k+1})]h}{4} \tag{6.7}$$

此时，函数 $f(x)$在区间$[a, b]$的积分变成为

$$T_{2n} = \frac{h}{4}\sum_{k=1}^{2^n}[f(x_k) + f(x_{k+1})] + \frac{h}{2}\sum_{k=1}^{2^n} f(x_{k+0.5}) \tag{6.8}$$

进一步利用前面的公式，可得

$$T_{2n} = \frac{1}{2}T_n + \frac{h}{2}\sum_{k=1}^{2^n} f(x_{k+0.5}) \tag{6.9}$$

上述公式的几何意义是明显的：每次迭代之后，函数 $f(x)$在区间$[a, b]$的积分值变为两部分组成。一部分按原来的参数值计算，另一部分按照二分之后新的参数值进行计算；一部分取上次计算的积分值的一半（$T_n/2$），另一部分取二分之后新插入点 $x_{k+0.5}$ 的函数值 $f(x_{k+0.5})$和新的区间长度（$h/2$）形成的矩形来计算其积分值，将两部分相加，得到新的一次计算的积分值。

变步长梯形算法求解函数定积分的流程图如图 6.6 所示。

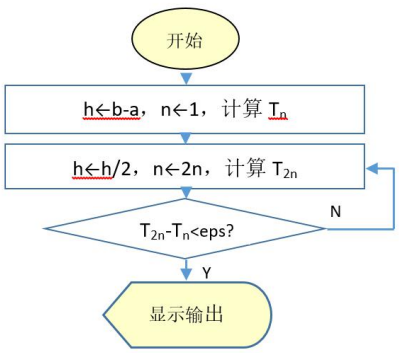

图 6.6 变步长梯形算法求积分流程图

流程图所表示的变步长梯形算法求解函数定积分的步骤如下：

(1) $n\leftarrow 1$, $h\leftarrow b-a$，利用公式计算积分值 T_n。

(2) 区间二分，$h\leftarrow h/2$，利用迭代公式 T_{2n} 计算新的积分值。

(3) 查看积分误差 $T_{2n}-T_n$，并与预设的积分控制误差 eps 比较，当 $T_{2n}-T_n<$eps 时，算法结束，否则，转第二步继续迭代计算。

6.5.2 变步长梯形算法求定积分程序设计

通过上面的分析以及得出的变步长梯形算法求定积分计算公式可以看出，用这种方法计算函数的定积分关键的计算步骤有两个：第一，计算被积函数 $f(x)$的值，因为每次迭代计算函数的积分值，都要在小区间内新插入的点上计算函数 $f(x)$的值；第二，变步长梯形算法计算函数积分的迭代计算。

据此分析，以及考虑到程序的可扩展性，我们可以定义两个抽象类，函数类 Function 和积分类 Integration。它们各自有一个纯虚函数，都重载了运算符“( )”，一个用来计算 $f(x)$在点 x 处的函数值，另一个用来计算 $f(x)$在区间$[a, b]$上的积分值。具体的函数实现由派生类定义。为了给出具体的计算实例，假设被积函数为：

$$f(x)=[\log(1+x)]/(1+x^2) \tag{6.10}$$

积分区间为[1, 2]，即积分公式为：

$$I=\int_1^2 \frac{\log(1+x)}{1+x^2}dx \tag{6.11}$$

下面给出变步长梯形算法求函数 $f(x)=[\log(1+x)]/(1+x^2)$的定积分的完整程序，积分误差设为 eps=10^{-7}。

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
class Function
{                                                    //抽象类Function的定义
public:
    virtual double operator () (double x) const = 0;    //纯虚函数重载运算符()
    virtual ~Function() { }
};

class MyFunction: public Function
{                                                    //公有派生类MyFunction定义
public:
    virtual double operator()(double x) const;          //覆盖虚函数
};

class Integration
```

```
{                                                    //抽象类Integration定义
public:
    virtual double operator () (double a, double b, double eps) const = 0;
    virtual ~Integration() { }
};

class Trapz: public Integration
{                                                    //公有派生类Trapz定义
public:
    Trapz(const Function &f) : f(f) {}               //构造函数
    virtual double operator ()(double a, double b, double eps) const;
private:
    const Function &f;                   //私有成员，Function类对象的指针
};

//被积函数
double MyFunction::operator () (double x) const
{
    return log(1.0 + x) / (1.0 + x * x);
}

//积分运算过程，重载为运算符()
double Trapz::operator () (double a, double b, double eps) const
{                                        //是Trapz类的虚函数成员
    bool done = false;
    int n = 1;
    double h = b - a;
    double tn = h * (f(a) + f(b)) / 2;     //计算n = 1时的积分值
    double t2n;
    do {
        double sum = 0;
        for    (int k = 0; k < n; k++)
    {
            double x=a+(k+0.5)*h;
            sum += f(x);
        }
        t2n = (tn + h * sum) / 2.0;      //变步长梯形法计算
        if (fabs(t2n - tn) < eps)
            done = true;                 //判断积分误差
        else {                           //进行下一步计算
            tn = t2n;
            n *= 2;
            h /= 2;
        }
```

```
    } while (!done);
    return t2n;
}

int main()                          //主函数
{
    MyFunction f;                   //定义MyFunction类的对象
    Trapz trapz(f);                 //定义Trapz类的对象
                                    //计算并输出积分结果
    cout << "TRAPZ Int: " << setprecision(7) << trapz(0, 2, 1e-7) << endl;
    return 0;
}
```

程序运行结果如下：

```
TRAPZ Int: 0.5548952
```

应该指出，上述算法在某些特殊情况下可能失效或误差很大，下面是一个典型的例子，如图 6.7 所示。

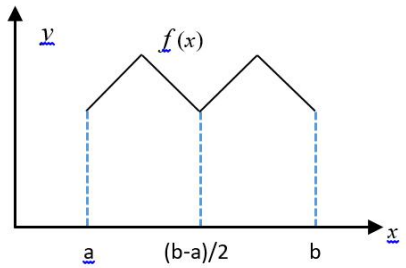

图 6.7 引起算法误差较大的一个例子

图中，函数 $f(x)$是左右完全对称的函数，且 $f(a)=f(b)=f((b-a)/2)$。当 $k=1$ 时，$T_n=T_{2n}$，误差计算为零，算法终止，但实际上误差非常大，因此，变步长梯形函数积分方法对于类似这种情况的例子是不适用的，在实际工作中，应该注意避免此类问题的出现。

习题

1. 什么叫做多态性？在 C++中是如何实现多态性的？

2. 简述虚函数与纯虚函数的作用和区别。

3. 什么叫做抽象类？抽象类有何作用？抽象类的派生类是否一定要给出纯虚函数的实现？

4. 在 C++中，能否声明虚构造函数？为什么？能否声明虚析构函数？为什么？有何用途？

5. 请编写一个哺乳动物类 Mammal，再由此派生出猫类 Cat，二者都声明 speak()成员函数，该函数在基类中被声明为虚函数。声明一个 Cat 类的对象，通过此对象调用 speak 函数，观察运行结果。

6. 将单目运算符--重载为时钟类的成员函数。

7. 编写一个抽象类 Shape，在此基础上派生出类 Rectangle 和 Circle，二者都有计算对象面积的函数 getArea（），计算对象周长的函数 getPerim（）。

8. 定义一个基类 BaseClass，从它派生出类 DerivedClass。BaseClass 有成员函数 fn1(),fn2(),其中 fn1()是虚函数；DerivedClass 也有成员函数 fn1(),fn2()。在主函数中声明一个 DerivedClass 的对象，分别用 BaseClass 和 DerivedClass 的指针指向 DerivedClass 的对象，并通过指针调用 fn1(),fn2(),观察运行结果。

9. 编写程序定义一个基类 BaseClass，从它派生出类 DerivedClass。在 BaseClass 中声明虚析构函数，在主函数中将一个动态分配的 DerivedClass 的对象地址赋给一个 BaseClass 的指针，然后通过指针释放对象空间。观察程序运行过程。

10.编写程序定义类 Point，有数据成员 x、y，为其定义友元函数实现重载“+”。

11.编写一个程序实现图书和杂志销售管理。当输入一系列图书和杂志销售记录后，将销售良好（图书销售：每月 500 本以上，杂志销售：每月 2500 本以上）的图书和杂志名称显示出来。

12.应用抽象类，求圆、园内接正方形和圆外切正方形的面积和周长。