第5章 多态性

多态性是C++面向对象技术中又一个重要的基本特征。客观事物之间的联系和作用也体现了多态性，即对同一条消息，不同的对象有不同的反应。多态性的应用可以使编程显得更为简捷、便利，它为程序的模块化设计提供了又一手段。本章围绕类层次中同名函数的不同实现，首先介绍类型兼容规则，进而引出多态的实现类型、多态性实现的相关技术，然后介绍虚函数的声明和使用、纯虚函数与抽象类等，最后介绍运算符重载与友元。

5.1 类型兼容规则

在类层次中，派生类对象之间有一个共同点，那就是来自于同一个基类，而派生类对象与基类有什么关系呢？事实上，它们遵循类型兼容规则。

类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象替代。通过公有继承，派生类得到了基类中除构造函数、析构函数之外的所有成员。这样，公有派生类实际具备了基类的所有功能，凡是基类能解决的问题，公有派生类都可以解决。类型兼容规则中“替代”包括以下情况。

(1) 派生类的对象可以赋值给基类的对象。

(2) 派生类的对象可以初始化基类的引用。

(3) 派生类的对象的地址可以赋值给基类的指针变量。

例如：

```
class A
{ … }
class B:public A                    //公有继承
{ … }
A a1, * pa1;
B b1;
a1 = b1;                            //第(1)种情况
A &bb = b1;                         //第(2)种情况
pa1 = &b1;                          //第(3)种情况
```

思考题：如果基类指针要访问派生类的成员，怎么办？

C++提供了多态机制解决这个问题，而类型兼容规则是C++多态的重要基础。下面介绍多态。

5.2 多态的实现类型

继承性反映的是类与类之间的层次关系，多态性则是考虑这种层次关系中特定成员函数之间的关系问题，是解决行为的再抽象问题。多态性是指类中同一函数名对应多个具有相似功能的不同函数，可以使用相同的调用方式调用这些具有不同功能的同名函数的特性。在C++程序中，表现为用同一种调用方式完成不同的处理。

从实现的角度划分，多态可以分为编译时多态和运行时多态。编译时多态是指在编译阶段由编译系统根据操作数据确定调用哪个同名的函数；运行时多态是指在运行阶段才根据产生的信息确定需要调用哪个同名的函数。调用不同的函数意味着执行不同的处理。C++采用联编技术支持多态性。

5.3 联编

多态性的实现过程中，确定调用哪个同名函数的过程就是联编(binding)，又称绑定。联编是指计算机程序自身彼此关联的过程，也就是把一个函数名和一个函数体联系在一起的过程。用面向对象的术语讲，就是把一条消息和一个对象的行为相结合的过程。按照进行的阶段的不同，联编可以分为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。

5.3.1 静态联编

在编译阶段完成的联编称为静态联编(static binding)。在编译过程中，编译系统可以根据参数不同确定哪一个是同名函数。函数重载和运算符重载就是通过静态联编方式实现的编译时多态的体现。静态联编的优点是函数调用速度快、效率较高，缺点是编程不够灵活。

例 5-1 示例静态联编。

```
//example 5_1.cpp
#include <iostream>
using namespace std;
class Student
{
public:
    void print()
    {
        cout <<"A student"<< endl;
    }
};
class GStudent:public Student
{
public:
    void print()
    {
```

```
    cout <<"A graduate student"<< endl;
    }
};
int main()
{
    Student s1, * ps;
    GStudent s2;
    s1.print();
    s2.print();
    s2.Student::print();
    ps = &s1;
    ps -> print();
    ps = &s2;                    //类型兼容规则(3)
    ps -> print();               //希望调用对象 s2 的输出函数,但调用的却是对象 s1 的输出函数
    return 0;
}
```

该程序的运行结果为：

```
A student
A graduate student
A student
A student
A student
```

说明：基类指针 ps 指向派生类对象 s2 时并没有调用派生类的 print(),而仍然调用基类的 print(),这是静态联编的结果。在程序编译阶段,基类指针 ps 对 print()的操作只能绑定到基类的 print()。

5.3.2 动态联编

对例 5-1 的分析可知,有些联编工作无法在编译阶段准确完成,只有在运行程序时才能确定将要调用的函数。这种在运行阶段进行的联编称为动态联编(dynamic binding)。动态联编的优点是提供了更好的编程灵活性、问题抽象性和程序易维护性；缺点是与静态联编相比,函数调用速度慢。

在例 5-1 中,静态联编把基类指针 ps 指向的对象绑定到基类上,而在运行时进行动态联编将把 ps 指向的对象绑定到派生类上。可见,同一个指针在不同阶段被绑定的类对象将是不同的,进而被关联的类成员函数也是不同的。那么如何确定是用静态联编还是用动态联编呢？C++规定,动态联编通过继承和虚函数实现。

从上述分析可以看出,静态联编和动态联编都是属于多态性的表现,它们是在不同阶段对不同实现进行不同的选择。

5.4 虚函数

虚函数是动态联编的基础。虚函数是非静态的成员函数,经过派生之后,虚函数在类族中可以实现运行时多态。

5.4.1 虚函数的声明

虚函数是一个在某基类中声明为 virtual，并在一个或多个派生类中被重新定义的成员函数。声明虚函数的格式如下：

```
virtual <返回值类型> <函数名>(<参数表>);
```

一个函数一旦被声明为虚函数，则无论声明它的类被继承了多少层，在每一层派生类中该函数都保持虚函数特性。因此，在派生类中重新定义该函数时，可以省略关键字 virtual。但是，为了提高程序的可读性，往往不省略。在程序运行时，不同类的对象调用各自的虚函数，这就是运行时多态。

5.4.2 虚函数的使用

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函数实现。当使用对象指针或对象引用调用虚函数时，采用动态联编方式，即在运行时进行关联或绑定。

例 5-2 示例动态联编。采用对象指针调用虚函数。

```
//example 5_2.cpp
#include <iostream>
using namespace std;
class Student
{
public:
    virtual void print()                          //定义虚函数
    {
        cout <<"A student"<< endl;
    }
};
class GStudent:public Student
{
public:
    virtual void print()                          //派生类的关键字 virtual 可以省略
    {
        cout <<"A graduate student"<< endl;
    }
};
int main()
{
    Student s1, * ps;
    GStudent s2;
    s1.print();
    s2.print();
    s2.Student::print();
    ps = &s1;
```

```
    ps->print();
    ps=&s2;                                    //类型兼容规则(3)
    ps->print();                               //对象指针调用虚函数,采用动态联编方式
    return 0;
}
```

该程序的运行结果为:

```
A student
A graduate student
A student
A student
A graduate student
```

说明:该程序将例 5-1 中基类的 print()声明为虚函数,使结果不同,即定义一个基类的对象指针,就可以指向不同派生类的对象,同时调用不同派生类的虚函数。这就是动态联编的结果。

值得注意的是:只有在类型兼容规则基础上,通过对象指针或对象引用调用虚函数,才能实现动态联编。如果采用对象调用虚函数,则采用的是静态联编方式。

例 5-3 示例动态联编。采用对象引用调用虚函数。

```
//example 5_3.cpp
#include<iostream>
using namespace std;
class Student
{
public:
    virtual void print()
    {
        cout<<"A student"<<endl;
    }
};
class GStudent:public Student
{
public:
    virtual void print()
    {
        cout<<"A graduate student"<<endl;
    }
};
void fun(Student &s)
{
    s.print();                                 //采用对象引用调用虚函数
}
int main()
{
    Student s1;
    GStudent s2;
    fun(s1);
    fun(s2);                                   //类型兼容规则(2)
```

```
    return 0;
}
```

该程序的运行结果为：

```
A student
A graduate student
```

说明：运行结果表明，只要定义一个基类的对象指针或对象引用，就可以调用期望的虚函数。在实际应用中，就可以使编程人员不必过多地考虑类的层次关系，无须显式地写出虚函数的路径，只需将对象指针指向相应的派生类或引用相应的对象，通过动态联编就可以对消息做出正确的反应。

思考题：将虚函数改为普通成员函数，其结果如何？或将 fun()的参数改为一般对象，其结果如何？

使用虚函数时应注意：

(1) 在派生类中重新定义虚函数时，必须保证函数的返回值类型和参数与基类中的声明完全一致。在类的成员函数被声明为虚函数后，派生类就具有多态性。但是，如果仅仅是基类和派生类成员函数的名字相同，而参数的类型不同，或者函数的返回值不同，即使被声明为虚函数，派生类的函数也不具备多态性。

(2) 如果在派生类中没有重新定义虚函数，则派生类的对象将使用基类的虚函数代码。

将一个类的成员函数定义为虚函数有利于编程，尽管它会引起一些额外的开销。是不是任何成员都可以声明为虚函数呢？一般来说，可将类族中的具有共性的成员函数声明为虚函数，而具有个性的函数没有必要声明为虚函数。但是，下面的情况例外。

① 静态成员函数不能声明为虚函数。因为静态函数不属于某一个对象，没有多态性的特征。

② 内联成员函数不能声明为虚函数。因为内联函数的执行代码是明确的，没有多态性的特征。如果将那些在类声明时就定义内容的成员函数声明为虚函数，此时函数不是内联函数，而以多态性出现。

③ 构造函数不能是虚函数。构造函数是在定义对象时被调用，完成对象的初始化，此时对象还没有完全建立。虚函数作为运行时的多态性的基础，主要是针对对象的，而构造函数是在对象产生之前运行的。所以，将构造函数声明为虚函数是没有意义的。

④ 析构函数可以是虚函数，且往往被定义为虚函数。一般来说，若某类中有虚函数，则其析构函数也应当定义为虚函数。特别是需要析构函数完成一些有意义的操作，如释放内存时，尤其应当如此。由于实施多态性时是通过将基类的指针指向派生类的对象完成的，如果删除该指针，就会调用该指针指向的派生类的析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象才被完全释放。因此，析构函数常被声明为虚函数。如果一个类的析构函数是虚函数，那么，由它派生的所有子类的析构函数也是虚函数。

例 5-4 示例虚析构函数。

```
//example 5_4.cpp
#include <iostream>
```

```
using namespace std;
class A
{
public:
    virtual ~A()
    {
        cout <<"call A::~A()"<< endl;
    }
};
class B:public A
{
    char * buf;
public:
    B(int i)
    {
        buf = new char[i];
    }
    virtual ~B()
    {
        delete[ ] buf;
        cout <<"call B::~B()"<< endl;
    }
};
void fun(A * a)
{
    delete a;
}
int main()
{
    A * a = new B(10);
    fun(a);
    return 0;
}
```

该程序的运行结果为：

```
call B::~B()
call A::~A()
```

如果类 A 中的析构函数不定义为虚函数，则程序的运行结果为：

```
call A::~A()
```

说明：第一个结果是因为基类的析构函数说明为虚函数时，调用 fun(a)函数，执行“delete a;”语句时采用动态联编，a 被关联到派生类对象，先调用派生类的析构函数，再调用基类的析构函数；如果基类的析构函数不定义为虚函数，调用 fun(a)函数，执行“delete a;”语句时采用静态联编，a 被关联到基类对象，只调用基类的析构函数，所以输出第二个结果。

利用虚函数可以使所设计的软件系统变得灵活，提高了代码的可重用性。虚函数为一个类族中所有派生类的同一行为提供了统一的接口，使得程序员在使用一个类族时只需记住一个接口即可。这种接口与实现分离的机制也提供了对 MFC 的支持，如果能正确地实

现这些类库，则它们将操作一个公共接口，可以用来派生自己的类以满足特定的需要。有时在声明一个基类时无法为虚函数定义其具体实现，这时可以将其声明为纯虚函数。包含纯虚函数的类称为抽象类。

5.5 纯虚函数与抽象类

抽象类是一种特殊的类，专门作为基类派生新类，自身无法实例化，也就是无法定义抽象类的对象，主要为一类族提供统一的操作接口。抽象类的主要作用是将有关的派生类组织在一个继承层次结构中，由抽象类为它们提供一个公共的根，相关的派生类就从这个根派生出来。通过抽象类为一个类族建立一个公共的接口，这个公共接口就是纯虚函数。

5.5.1 纯虚函数的定义

纯虚函数是一个在抽象类中声明的虚函数，只给出了函数声明而没有具体实现内容，要求各派生类根据实际需要定义自己的内容，纯虚函数在声明时要在函数原型的后面赋 0。声明纯虚函数的一般格式如下：

```
virtual <返回值类型> <函数名>(<参数表>) = 0;
```

说明：声明为纯虚函数之后，抽象类中就不再给出函数的实现部分。

5.5.2 抽象类的使用

抽象类只能用作其他类的基类，不能建立抽象类对象。因此，抽象类不能用作参数类型、函数类型或显式转换的类型，但可以说明指向抽象类的指针或引用，该指针或引用可以指向抽象类的派生类，进而实现多态性。例如，保护的构造函数类和保护的析构函数类都不能声明对象，因此，都是抽象类。含有纯虚函数的类也是抽象类。

抽象类派生出新类之后，如果派生类给出所有纯虚函数的函数实现，这个派生类就可以定义自己的对象，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，继承了部分纯虚函数，这时的派生类仍然是一个抽象类。

例 5-5 示例纯虚函数及抽象类。计算图形面积。

```
//example 5_5.cpp
#include <iostream>
using namespace std;
const double PI = 3.14159;
class Shapes                                    //抽象类
{
protected:
    int x,y;
public:
    void setvalue(int d,int w = 0){x = d;y = w;}
    virtual void disp() = 0;                    //纯虚函数
```

```
};
class Square:public Shapes
{
public:
    void disp()                          //计算矩形面积
    {
        cout<<"area of rectangle:"<<x*y<<endl;
    }
};
class Circle:public Shapes
{
public:
    void disp()                          //计算圆面积
    {
        cout<<"area of circle:"<<PI*x*x<<endl;
    }
};
int main()
{
    Shapes* ptr[2];                      //定义抽象类指针
    Square s1;
    Circle c1;
    ptr[0]=&s1;                          //抽象类指针指向派生类对象,类型兼容规则(3)
    ptr[0]->setvalue(10,5);
    ptr[0]->disp();                      //抽象类指针调用派生类成员函数
    ptr[1]=&c1;                          //抽象类指针指向派生类对象,类型兼容规则(3)
    ptr[1]->setvalue(10);
    ptr[1]->disp();                      //抽象类指针调用派生类成员函数
    return 0;
}
```

该程序的运行结果为:

```
area of rectangle:50
area of circle:314.159
```

说明:本程序定义了一个抽象类 Shapes 和它的两个派生类 Square 与 Circle。在 main()中定义了抽象类的指针数组 ptr[2],分别指向对象 s1 和 c1,从而通过这两个对象指针分别调用两个派生类中的虚函数,实现动态联编。同时,派生类的虚函数并没有显式声明,因为它们与基类的纯虚函数具有相同的名称、参数及返回值,由编译系统自动判断确定其为虚函数。

5.6 运算符重载与友元

C++预定义的运算符只能对基本类型数据进行操作,实际上,很多用户自定义类型数据也需要有类似的运算,这就提出了对运算符进行重新定义,赋予预定义的运算符新功能的要求。当然也可以通过普通成员函数实现(例 5-6),但是,有对象参加运算时,将运算符重载使程序代码更直观、易读。

运算符重载的实质就是函数重载。在实现过程中，首先把指定的表达式转化为对运算符重载函数的调用，将操作数转化为运算符重载函数的实参，然后根据参数匹配的原则确定需要调用的函数，这个过程是在编译过程中完成的，采用静态联编方式。与函数重载不同的是，运算符重载函数的参数一定含有对象且参数个数有限制，并保持优先级、结合性以及语法结构不变等特性。

例 5-6 示例运算符重载，并与成员函数实现的方式进行比较，计算应付给的人民币。

```
//example 5_6.cpp
#include <iostream>
using namespace std;
class RMB                                          //人民币类
{
public:
    RMB(double d)
    {
        yuan = (int)d;
        jf = (int)((d - yuan) * 100);
    }
    RMB interest(double rate);                     //计算利息
    RMB add(RMB d);                                //人民币相加
    void display( )
    {
        cout <<(yuan + jf/100.0)<< endl;
    }
    RMB operator + (RMB d)                         //运算符"+"重载函数
    {
        return RMB(yuan + d.yuan + (jf + d.jf)/100.0);
    }
    RMB operator * (double rate)                   //运算符"*"重载函数
    {
        return RMB((yuan + jf/100.0) * rate);
    }
private:
    unsigned int yuan;                             //元
    unsigned int jf;                               //角分
};
RMB RMB::interest(double rate)
{
    return RMB((yuan + jf/100.0) * rate);
}
RMB RMB::add(RMB d)
{
    return RMB (yuan + d.yuan + jf/100.0 + d.jf/100.0);
}
//以下是计算应付人民币的两种方法
RMB expense1(RMB principle,double rate)            //采用普通成员函数求本利和
{
    RMB interests = principle.interest(rate);
    return principle.add(interests);
```

```
}
RMB expense2(RMB principle,double rate)                    //采用运算符重载函数求本利和
{
    RMB interests = principle * rate;                      //本金乘利率
    return principle + interests;                          //本利和
}
int main( )
{
    RMB x = 10256.50;
    double yrate = 0.035;
    expense1(x,yrate).display();
    expense2(x,yrate).display();
    return 0;
}
```

该程序的运行结果为：

```
10615.5
10615.5
```

说明：两种计算应付人民币的函数得到的结果相同，但 expense2()更直观，可读性更好，它符合人们用“+”“*”运算符计算的习惯。如果不定义运算符重载，则 expense2()中 principle * rate 和 principle+interests 是非法的，因为参加运算的是类对象而不是基本类型的数据。

运算符重载的目的仅仅是为了语法上的方便，增强程序的易读性。因此为使用户自定义的类型更易写，尤其是更易读的时候，就有理由重载运算符。但是必须明白一点，运算符重载并非一个程序必须有的功能。

5.6.1 运算符重载的定义

运算符重载就是赋予系统预定义的运算符多重含义，使同一个运算符既可以作用于预定义的数据类型，也可以作用于用户自定义的数据类型。同一运算符作用于不同数据类型时，其行为是不同的。运算符重载的一般格式如下：

```
<返回值类型> operator <运算符>(<参数表>)
{
    <函数体>;
}
```

其中，operator 是定义运算符重载函数的关键字。<参数表>中最多有一个形参。

以复数类 Complex 为例，说明运算符重载的具体方式。如果要完成与复数相关的运算，用户可以自定义一个复数类 Complex 完成复数运算。源代码如下：

```
class Complex                                              //复数类
{
public:
    Complex(double r = 0.0,double i = 0.0){m_fReal = r;m_fImag = i;}  //构造函数
```

```
    double Real(){return m_fReal;}                    //返回复数的实部
    double Imag(){return m_fImag;}                    //返回复数的虚部
    Complex operator + (Complex &c);                  //重载运算符" + ",复数加复数
    Complex operator + (double d);                    //重载运算符" + ",复数加实数
    Complex operator - (Complex &c);                  //重载运算符" - ",复数减复数
    Complex operator = (Complex x);                   //重载运算符" = ",复数赋值
private:
    double m_fReal,m_fImag;                           //私有数据成员
};
```

说明：进行运算符重载，不过是将原函数名替换为关键字 operator 和相应运算符。除此之外，在该方法的具体实现代码中没有任何不同之处。从本质上讲，运算符重载就是函数重载，是另一种形式的函数调用而已。但是运算符重载也有别于函数重载，运算符重载的函数参数就是该运算符涉及的操作数，因此运算符重载在参数个数上是有限制的，这是它与函数重载的不同之处。

5.6.2 运算符重载规则

重载运算符应遵循如下规则。

(1) C++的运算符除了少数几个之外，其他的可以重载，而且只能重载已有的运算符，不可臆造新的运算符。因为基本数据类型之间的关系是确定的，如果允许定义新运算符，那么，基本数据类型的内在关系将混乱。

不能重载的运算符只有 6 个，它们是成员访问运算符“.”、成员指针运算符“*”和“->”、作用域运算符“::”、sizeof 运算符以及三目运算符“?:”。前面 3 个运算符保证了 C++中访问成员功能的含义不被改变。作用域运算符和 sizeof 运算符的操作数是数据类型，而不是普通的表达式，也不具备重载的特征。

(2) 重载之后运算符的优先级和结合性都不会改变，并且要保持原运算符的语法结构。参数和返回值类型可以重新说明。

(3) 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。一般来讲，重载的功能应当与原有功能类似，不能改变原运算符的操作数个数，同时至少要有一个操作数的类型是自定义类型。

(4) 运算符重载有两种方式：重载为类的成员函数和重载为类的友元函数。当运算符重载为类的成员函数时，函数的参数个数应比原来的操作数个数少一个(后缀“++”和后缀“--”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，如果某个对象调用重载的成员函数，自身的数据可以直接访问，就无须再放在参数表中进行传递，少了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进行，因此使用到的参数都要进行传递，参数个数与运算符原操作数个数相同。一般将单目运算符重载为成员函数，而双目运算符则重载为友元函数。

(5) 当运算符重载为类的成员函数时，由于单目运算除了对象以外没有其他参数，因此重载“++”和“--”运算符，不能区分是前缀操作还是后缀操作。C++约定，在参数表中放上一个整型参数，表示后缀运算符。

5.6.3 运算符重载为成员函数

运算符重载为成员函数后,它就可以自由地访问类的所有成员。实际使用时,总是通过该类的某个对象访问重载的运算符。此时,运算符重载函数的参数最多一个。如果是双目运算符,左操作数一定是对象本身,由 this 指针给出,另一个操作数则需要通过运算符重载函数的参数表传递;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。

例 5-7 示例运算符重载为成员函数形式。复数类加法、减法和赋值运算符重载。

```
//example 5_7.cpp
#include <iostream>
using namespace std;
class Complex                                   //复数类
{
public:
    Complex(double r = 0.0,double i = 0.0){m_fReal = r;m_fImag = i;}   //构造函数
    double Real(){return m_fReal;}              //返回复数的实部
    double Imag(){return m_fImag;}              //返回复数的虚部
    Complex operator + (Complex &c);            //复数加复数
    Complex operator + (double d);              //复数加实数
    Complex operator - (Complex &c);            //复数减复数
    Complex operator = (Complex x);             //复数赋值
private:
    double m_fReal,m_fImag;                     //私有数据成员
};
Complex Complex::operator + (Complex &c)        //重载运算符" + ",两个复数相加
{
    Complex temp;
    temp.m_fReal = m_fReal + c.m_fReal;         //实部相加
    temp.m_fImag = m_fImag + c.m_fImag;         //虚部相加
    return temp;
}
Complex Complex::operator + (double d)          //重载运算符" + ",一个复数加一个实数
{
    Complex temp;
    temp.m_fReal = m_fReal + d;
    temp.m_fImag = m_fImag;
    return temp;
}
Complex Complex::operator - (Complex &c)        //重载运算符" - ",两个复数相减
{
    Complex temp;
    temp.m_fReal = m_fReal - c.m_fReal;         //实部相减
    temp.m_fImag = m_fImag - c.m_fImag;         //虚部相减
    return temp;
}
Complex Complex::operator = (Complex c)         //重载运算符" = "
{
```

```
    m_fReal = c.m_fReal;
    m_fImag = c.m_fImag;
    return * this;                          // * this 表示当前对象
}
int main()
{
    Complex c1(3,4),c2(5,6),c3,c4;          //定义复数类的对象
    cout <<"c1 = "<< c1.Real()<<" + j"<< c1.Imag()<< endl;
    cout <<"c2 = "<< c2.Real()<<" + j"<< c2.Imag()<< endl;
    c3 = c1 + c2;                           //调用运算符" + "" = "重载函数,完成复数加复数
    cout <<"c3 = c1 + c2 = "<< c3.Real()<<" + j"<< c3.Imag()<< endl;
    c3 = c3 + 6.5;                          //调用运算符" + "" = "重载函数,完成复数加实数
    cout <<"c3 + 6.5 = "<< c3.Real()<<" + j"<< c3.Imag()<< endl;
    c4 = c2 - c1;                           //调用运算符" - "" = "重载函数,完成复数减复数
    cout <<"c4 = c2 - c1 = "<< c4.Real()<<" + j"<< c4.Imag()<< endl;
    return 0;
}
```

该程序的运行结果为：

```
c1 = 3 + j4
c2 = 5 + j6
c3 = c1 + c2 = 8 + j10
c3 + 6.5 = 14.5 + j10
c4 = c2 - c1 = 2 + j2
```

说明：在本例中把复数的加法、减法和赋值运算符重载为复数类的成员函数。可以看出，除了在函数声明和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区别，在使用的时候，可以直接通过运算符对操作数操作的方式完成函数调用。这时，运算符原有的功能都不改变，对整型数、浮点数等基本类型数据的运算仍然遵循 C++ 预定义的规则。由此可见，运算符作用于不同的对象上，就会导致不同的操作行为，具有了更广泛的多态特征。

例 5-8 示例单目运算符“++”重载为成员函数形式。

```
//example 5_8.cpp
#include <iostream>
using namespace std;
class Clock                                 //时钟类
{
public:
    Clock(int NewH = 0,int NewM = 0,int NewS = 0);
    void ShowTime();
    void operator++();                      //前缀单目运算符重载函数的声明
    void operator++(int);                   //后缀单目运算符重载函数,加 int 参数以示区分
private:
    int Hour,Minute,Second;
};
Clock::Clock(int NewH,int NewM,int NewS)    //构造函数
{
    if(0 <= NewH&&NewH < 24&&0 <= NewM&&NewM < 60&&0 <= NewS&&NewS < 60)
```

```
    {
        Hour = NewH;
        Minute = NewM;
        Second = NewS;
    }
    else
        cout <<"Time error!"<< endl;
}
void Clock::ShowTime()                          //显示时间函数的实现
{
    cout << Hour <<":"<< Minute <<":"<< Second << endl;
}
void Clock::operator++()                        //前缀单目运算符重载函数的实现
{
    Second++;
    if(Second >= 60)
    {
        Second = Second - 60;
        Minute++;
        if(Minute >= 60)
        {
                Minute = Minute - 60;
            Hour++;
            Hour = Hour % 24;
        }
    }
        cout <<"++Clock:";
}
void Clock::operator++(int)                     //后缀单目运算符重载函数的实现
{
        Second++;
        if(Second >= 60)
        {
            Second = Second - 60;
            Minute++;
            if(Minute >= 60)
            {
                Minute = Minute - 60;
                Hour++;
                Hour = Hour % 24;
            }
        }
        cout <<"Clock++:";
}
int main()
{
    Clock myClock(11,59,59);
    cout <<"First time output:";
    myClock.ShowTime();
    myClock++;
    myClock.ShowTime();
```

```
    ++myClock;
    myClock.ShowTime();
    return 0;
}
```

该程序的运行结果为：

```
First time output:11:59:59
Clock++:12:0:0
++Clock:12:0:1
```

说明：本例中，作为成员函数的前缀单目运算符重载函数没有参数，而后缀单目运算符重载函数有一个整型参数。这个整型参数在函数体中并不使用，仅用于区别前缀与后缀，因此参数表中只给出了类型名，没有参数名。

思考题：如果要进行实数加复数的运算，应该怎么办？

我们知道，运算符重载为成员函数后，如果是双目运算符，左操作数一定是对象本身，由this指针给出，而现在左操作数是一个实数，这时需要用友元来实现。

5.6.4 友元及运算符重载函数

根据类的封装性，一般将数据成员声明为私有成员，外部不能直接访问，只能通过类的公有成员函数对私有成员进行访问。有时，需要频繁地调用成员函数访问私有成员，这就存在一定的系统开销。C++从高效的角度出发，提供友元机制，使被声明为友元的全局函数或者其他类可以直接访问当前类中的私有成员，又不改变其私有成员的访问权限。

1. 友元的作用

友元不是类的成员，但能直接访问类的所有成员，避免了频繁调用类的成员函数。使用友元可以节约开销，提高程序的效率。但友元破坏了类的封装性，这也从侧面说明了C++不是完全的面向对象语言，它的设计目的是实用，增加友元机制是为了解决一些实际问题。

2. 友元的定义

如果友元是普通函数，且是另一个类的成员函数，称为友元函数；如果友元是一个类，则称为友元类。友元类的所有成员函数都称为友元函数。友元函数和友元类在被访问的类中声明，其格式分别如下：

```
friend <返回值类型> <函数名>(<参数表>);
```

```
friend <类名>;
```

例 5-9 示例友元。计算屏幕上两点之间的距离。

```
//example 5_9.cpp
#include <iostream>
#include <cmath>
```

```
using namespace std;
class TPoint
{
public:
    TPoint(double a,double b)
    {
        x = a;
        y = b;
        cout <<"point:("<< x <<","<< y <<")"<< endl;
    }
    friend double distance1(TPoint &a,TPoint &b);                    //友元函数的声明
private:
    double x,y;
};
double distance1(TPoint &a,TPoint &b)                                //被定义为友元的普通函数
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y)); //访问私有成员
}
int main()
{
    TPoint p1(2,3),p2(4,5);
    cout <<"the distance between two point is:"<< distance1(p1,p2)<< endl;
    return 0;
}
```

该程序的运行结果为：

```
point:(2,3)
point:(4,5)
the distance between two point is:2.82843
```

说明：在 TPoint 类中声明友元函数时，只给出了友元函数原型，友元函数 distance()的实现是在类外。在友元函数中通过对象名直接访问了 TPoint 类中的私有数据成员 x 和 y。

注意：友元函数一定不是本类的成员函数。即使将友元函数的实现放在类中，仍是友元函数。

思考题：将友元的函数体放在类中，其结果如何？

上例中的友元函数是一个普通函数，同样，另外一个类的成员函数也可以是友元函数，其使用与普通函数作为友元函数的使用基本相同，只是在使用该友元时要通过相应的类或对象名来访问。

另外，类也可以声明为另一个类的友元。若 A 类为 B 类的友元类，则 A 类的所有成员函数都是 B 类的友元函数，都可以访问 B 类的私有成员和保护成员。

例 5-10 示例友元类。

```
//example 5_10.cpp
#include <iostream>
#include <cmath>
using namespace std;
class A
{
```

```
public:
    A(){x = 5;}
    friend class B;                                    //友元类的声明
private:
    int x;
};
class B
{
public:
    void disp1(A tmp){tmp.x++;cout <<"disp1:x = "<< tmp.x << endl;}//访问私有成员
    void disp2(A tmp){tmp.x -- ;cout <<"disp2:x = "<< tmp.x << endl;}
};
int main()
{
    A obj1;
    B obj2;
    obj2.disp1(obj1);
    obj2.disp2(obj1);
    return 0;
}
```

该程序的运行结果为：

```
disp1:x = 6
disp2:x = 4
```

说明：

(1) 友元关系是不能传递的。B 类是 A 类的友元，C 类是 B 类的友元，C 类和 A 类之间，如果没有声明，就没有任何友元关系，不能进行数据共享。

(2) 友元关系是单向的。如果声明 B 类是 A 类的友元，B 类的成员函数就可以访问 A 类的私有成员和保护成员，但 A 类的成员函数不能访问 B 类的私有成员和保护成员，除非声明 A 类是 B 类的友元。

下面对友元的相关知识进行小结。

(1) 友元的出现主要是为了解决一些实际问题，友元本身不是面向对象的内容。

(2) 通过友元机制，一个类或函数可以直接访问另一类中的非公有成员。

(3) 可以将全局函数、类、类的成员函数声明为友元。

(4) 友元关系是不能传递的。B 类是 A 类的友元，C 类是 B 类的友元，C 类和 A 类之间，如果没有声明，就没有任何友元关系，不能进行数据共享。

(5) 友元关系是单向的，如果声明 B 类是 A 类的友元，B 类的成员函数就可以访问 A 类的私有成员和保护成员。但 A 类的成员不能访问 B 类的私有成员和保护成员，除非声明 A 类是 B 类的友元。

(6) 友元关系是不能继承的。B 类是 A 类的友元，C 类是 B 类的派生类，则 C 类和 A 类之间没有任何友元关系，除非 C 类声明 A 类是友元。

3. 运算符重载为友元函数

运算符重载为类的友元函数，它也可以自由地访问类的所有成员。不同的是，运算符所

需要的操作数都需要通过函数的形参传递,在参数表中参数从左至右的顺序就是运算符操作数的顺序。

运算符重载为类的友元函数的一般格式如下:

```
friend <返回值类型> operator <运算符>(<参数表>)
{
    <函数体>;
}
```

其中,<参数表>最多有两个形参。

例 5-11 示例实数加复数运算符重载为友元函数形式。

```
//example 5_11.cpp
#include <iostream>
using namespace std;
class Complex
{
    double m_fReal,m_fImag;
public:
    Complex(double r = 0,double i = 0):m_fReal(r),m_fImag(i){}
    double Real(){return m_fReal;}
    double Imag(){return m_fImag;}
    Complex operator + (double);
    Complex operator = (Complex);
    friend Complex operator + (double,Complex&);  //友元函数
};
Complex Complex::operator + (double d)
{
    Complex temp;
    temp.m_fReal = m_fReal + d;
    temp.m_fImag = m_fImag;
    return temp;
}
Complex Complex::operator = (Complex c)
{
    m_fReal = c.m_fReal;
    m_fImag = c.m_fImag;
    return *this;
}
Complex operator + (double d,Complex &c)          //普通函数
{
    Complex temp;
    temp.m_fReal = d + c.m_fReal;
    temp.m_fImag = c.m_fImag;
    return temp;
}
int main()
{
    Complex c1(3,4),c2,c3;
```

```
    cout <<"c1 = "<< c1.Real()<<" + j"<< c1.Imag()<< endl;
    c2 = c1 + 6.5;                                    //先调用成员函数" + ",再调用成员函数" = "
    cout <<"c2 = c1 + 6.5 = "<< c2.Real()<<" + j"<< c2.Imag()<< endl;
    c3 = 6.5 + c1;                                    //先调用友元函数" + ",再调用成员函数" = "
    cout <<"c3 = 6.5 + c1 = "<< c3.Real()<<" + j"<< c3.Imag()<< endl;
    return 0;
}
```

该程序的运行结果为：

```
c1 = 3 + j4
c2 = c1 + 6.5 = 9.5 + j4
c3 = 6.5 + c1 = 9.5 + j4
```

说明：如果把运算符“+”重载为成员函数，其左操作数一定是当前对象，因此不能完成实数加复数的运算，将运算符“+”重载为友元函数就可以完成。事实上，在C++标准库中，已经为用户提供了与复数有关的库函数，它们包含在< complex >头文件中。在实际应用中，用户只需要将此头文件包含到源程序文件中即可。

5.7 类型转换

C++是一种强类型语言。编译时，对不同类型的值进行复制通常需要显示转换(或强制转换)。

5.7.1 显示转换

通用的显式转换有再构造和类C两种方式。

```
double x = 10.3;
int y;
y = int (x);                                          //再构造方式
y = (int) x;                                          //类C方式
```

对于基本数据类型的大多数需求，通用显式转换是足够的。但是，通用显式转换可以不加区分地应用于指针，这样即使编译通过了也会导致运行时错误。例如，下面的代码编译没有错误。

例 5-12 示例显示转换。

```
//example 5_12.cpp
#include < iostream >
using namespace std;
class Dummy
{
    double i,j;
};
class Addition
{
public:
```

```
    Addition (int a, int b) { x = a; y = b; }
    int result() { return x + y;}
private:
    int x,y;
};
int main()
{
    Dummy d;
    Addition * padd;
    padd = (Addition * ) &d;
    cout << padd -> result();
    return 0;
}
```

程序声明一个指向 Addition 的指针,但是它使用通用显式转换为其赋值,指向了另一个不相关类型的对象:

```
padd = (Addition * ) &d;
```

由此看出,通用显式转换是无限制的。这种无限制显式转换允许将指针转换为任何其他指针,随后对成员结果的调用将产生运行时错误或其他意外的结果。为了避免这种错误,也为了规范和控制指针之间的显式转换,C++引入了 4 个特定的转换运算符。

5.7.2 特定的 4 个转换运算符

C++引入的 4 个特定转换运算符是:dynamic_cast、static_cast、reinterpret_cast 和 const_cast。它们的格式如下:

```
dynamic_cast <新类型> (表达式)
static_cast <新类型> (表达式)
reinterpret_cast <新类型> (表达式)
const_cast <新类型> (表达式)
```

它们能代替通用显式转换的功能,且每一个都有其特殊的意义,是 C++的类型转换多态。

1. 动态转换 dynamic_cast

dynamic_cast 只能用于类的指针或引用(或 void *)。其目的是确保转换的结果指向目标类型的有效完整对象。

动态转换既可以向上转换(派生类到基类),也可以向下转换(基类到派生类)。向上转换是为了实现泛化,其方式与隐式转换所允许的方式相同;而向下转换是为了实现多态,当且仅当基类指针指向一个派生类的有效完整对象时,向下转换才能成功,否则转换的结果为空指针。

例 5-13 示例动态转换。

```
//example 5_13.cpp
#include <iostream>
#include <exception>
```

```
using namespace std;
class Base { virtual void dummy() {} };
class Derived: public Base { int a; };
int main ()
{
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;
    pd = dynamic_cast < Derived * >(pba);
    if (pd == 0)
        cout << "Null pointer on first type - cast.\n";
    pd = dynamic_cast < Derived * >(pbb);
    if (pd == 0)
        cout << "Null pointer on second type - cast.\n";
    return 0;
}
```

该程序的运行结果为：

```
Null pointer on second type - cast.
```

上面的代码尝试从 Base * (pba 和 pbb)类型的指针对象向 Derived * 类型的指针对象执行两个动态转换，但只有第一个成功。注意它们各自的初始化：

```
Base * pba = new Derived;
Base * pbb = new Base;
```

尽管两者都是 Base * 类型的指针，但是 pba 实际上指向了 Derived 类型的对象，而 pbb 指向了 Base 类型的对象。因此，当使用 dynamic_cast 执行各自的类型转换时，pba 指向 Derived 类的完整对象，而 pbb 指向 Base 类的对象，这是 Derived 类的不完整对象。

当动态转换失败时，会返回一个空指针指示失败，如果转换的不是指针而是引用，则会抛出类型为 bad_cast 的异常。

2. 静态转换 static_cast

static_cast 的作用和动态转换相似，区别在于静态转换不会进行完整性检查。如果将基类对象的基类指针转换为派生类指针，则转换的结果实际指向一个不完整的派生类对象，例如：

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast < Derived * >(a);
```

以上代码是有效的，尽管 b 指向了一个不完整的 Derived 对象。后续对 b 进行的成员访问操作如果超出了 a 的范围，会导致严重的运行时错误和内存错误。

因此在实际应用中，静态转换应谨慎使用。只有对类型转换的上下文有十足的把握，才可以使用静态转换节约完整性检查的开销。特别地，把派生类指针转换成基类指针是 static_cast 最常见的用法，因为派生类包含了基类的所有成员，是绝对安全的：

```
class Base {};
class Derived: public Base {};
void f(Base * ){/ * … * /}
Derived * b = new Derived;
f(static_cast < Base * >(b));
```

以上代码中由于 b 所指向的对象包含了 Base 类型的所有成员,所以提倡使用 static_cast 节省完整性检查的开销。

3. 重释转换 reinterpret_cast

reinterpret_cast 用于对位的简单重新解释,可以将指针转换为任意其他指针类型,也可以将指针转换为长度足够长的整数类型或将长整数转换为任意指针。重释转换是简单二进制复制,不进行任何安全性检查。

```
class A { / * … * / };
class B { / * … * / };
A * a = new A;
B * b = reinterpret_cast < B * >(a);
auto c = reinterpret_cast < long long >(a);
a = reinterprete_cast < A * >(c);
```

以上代码中将 A * a 转换成 B * b 同样是非常不安全的行为,几乎无任何意义,因此不建议在应用中使用重释转换将指针转换为其他指针,除非对上下文有足够的把握。

4. 弃常转换 const_cast

const_cast 将任意具有 const 特性的指针或引用转换为一个指向同一对象的指针或引用,并且该指针或引用的 const 特性被移除。

例 5-14 示例弃常转换。

```
//example 5_14.cpp
# include < iostream >
using namespace std;
void print(char * str)
{
    cout << str << '\n';
}
int main ()
{
    const char * c = "sample text";
    print( const_cast < char * > (c) );
    return 0;
}
```

该程序的运行结果为:

```
sample text
```

5.8 本章小结

多态性是指同样的消息被不同类型的对象接收时导致不同行为的特征，是对行为的再抽象。这里同样的消息是指用同一函数名对函数的调用，不同行为是指不同的函数实现，也就是调用了不同的函数。

多态从实现的角度可以分为两类：编译时多态和运行时多态，分别通过静态联编和动态联编方式实现。函数重载和运算符重载是静态联编的体现，而动态联编通过继承和虚函数实现。

虚函数是用 virtual 关键字声明的非静态成员函数。虚函数是实现动态联编的基础，可以通过基类指针指向派生类对象，访问派生类的同名函数。这样，通过基类指针，就可以导致不同派生类的不同对象对同样的消息产生不同的行为，从而实现运行时多态。如果在基类不定义虚函数，那么通过基类指针只能访问基类的同名成员，尽管将基类指针指向派生类对象也是如此，是静态联编的结果。另外，要实现多态性在派生类中的延伸，派生类中的同名函数必须与基类的虚函数形式（即返回值类型和参数）一致，否则派生类中的同名函数将失去多态性。

纯虚函数是只给出了函数声明，而未给出具体实现且值为 0 的虚函数。包含纯虚函数的类称为抽象类。抽象类主要为派生类的多态提供共同的基类。抽象类不能生成对象，不能作为参数类型、函数返回值类型或显式转换的类型，其中的纯虚函数就是公共接口，在不同派生类中再给出虚函数的不同实现。可以声明指针或引用，作用于派生类对象实现多态性。

运算符重载是赋予系统预定义的运算符多重含义，使得预定义运算符能够对类对象进行运算。运算符重载实质上就是函数重载，但不同的是，运算符重载函数对参数个数有限制，并保持优先级、结合性以及语法结构不变等特性。在实现过程中，首先把指定的运算表达式转化为对运算符重载函数的调用，运算对象转化为运算符重载函数的实参，然后根据实参的类型确定需要调用的函数，这个过程是在编译阶段完成的。

运算符可以重载为成员函数或者友元。类的友元可以访问该类的所有成员。友元可以是普通函数、其他类的成员函数，也可以是其他类。友元在类之间、类与普通函数之间共享了内部封装的数据，对类的封装性有一定的破坏。友员关系既不能传递，也不可逆。

C++引入了 4 个特定转换运算符：dynamic_cast、static_cast、reinterpret_cast 和 const_cast。它们能代替通用显式转换的功能，且每一个都有其特殊的意义，是 C++的类型转换多态。

5.9 习题

1. 什么是多态性？在 C++中是如何实现多态的？
2. 虚函数与重载在设计方法上有何异同？
3. 编写一个时间类，实现时间的加、减、读和输出。
4. 定义一个哺乳动物 Mammal 类，再由此派生出狗 Dog 类，两者都定义 Speak()成员

函数,基类中定义为虚函数,定义一个 Dog 类的对象,调用 Speak()函数,观察运行结果。

5. 写出下面程序的运行结果,并回答问题。

```
#include <iostream>
using namespace std;
class Point
{
public:
    Point(int x1,int y1){x=x1;y=y1;}
    int area() const {return 0;}
private:
    int x,y;
};
class Rect:public Point
{
public:
    Rect(int x1,int y1,int u1,int w1): Point(x1,y1)
    {
        u=u1;w=w1;
    }
    int area() const {return u*w;}
private:
    int u,w;
};
void fun(Point &p)
{
    cout<<p.area()<<endl;
}
int main()
{
    Rect rec(2,4,10,6);
    fun(rec);
    return 0;
}
```

如果将 Point 类的 area()函数定义为虚函数,其运行结果是什么? 为什么?

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

7. 什么是抽象类? 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?

8. 定义一个 Shape 抽象类,在此基础上派生出 Rectangle 和 Circle 类,二者都由 GetArea()函数计算对象的面积,GetPerim()函数计算对象的周长。使用 Rectangle 类派生一个新类 Square。

9. 写出下面程序的运行结果,并回答问题。

```
#include <iostream>
using namespace std;
class A
{
public:
    A(int i):k(i){}
```

```
    virtual void operator!()
    {
        cout <<"A: K = "<< k << endl;
    }
protected:
    int k;
};
class B:public A
{
public:
    B(int n = 0):A(0),j(n){k++;}
    virtual void operator!()
    {
        cout <<"B: K = "<< k <<",J = "<< j << endl;
    }
protected:
    int j;
};
class C:public B
{
public:
    C(int n = 0):B(0),m(n){k++;j++;}
    virtual void operator!()
    {
        cout <<"C: K = "<< k <<",J = "<< j <<",M = "<< m << endl;
    }
private:
    int m;
};
int main()
{
    B b(5);
    C c(3);
    A a(2);
    A * ab = &a;
    ! * ab;
    !b;
    !c;
    A &ba = (A)b;
    !ba;
    A &ca = (B)c;
    !ca;
    B &cb = c;
    !cb;
    return 0;
}
```

如果将 A 类的虚函数定义为普通成员函数，其结果如何？为什么？如果将 C 类改为 A 类的公有派生类，应做如何修改才能使程序正常运行？

10. 前缀自加和后缀自加运算符重载时如何区别？

11. 为什么要定义友元？友元有哪几种类型？

12. 改正下面代码的错误。

```
#include <iostream>
using namespace std;
class Animal;
void SetValue(Animal&,int);
void SetValue(Animal&,int,int);
class Animal
{
public:
    friend void setValue(Animal&,int);
protected:
    int itsWeight;
    int itsAge;
};
void SetValue(Animal& ta,int tw)
{
    ta.itsWeight = tw;
}
void SetValue(Animal& ta,int tw,int tn)
{
    ta.itsWeught = tw;
    ta.itsAge = tn;
}
int main()
{
    Animal peppy;
    SetValue(peppy,5);
    SetValue(peppy,7,9);
    return 0;
}
```

13. 将第12题程序中的友元改成普通函数，为此增加访问类中保护数据的成员函数。

实验5.1 虚函数的使用

一、实验目的

1. 理解多态的概念。

2. 理解函数的静态联编和动态联编。

3. 掌握虚函数的定义。

4. 理解虚函数在类的继承层次中的作用、虚函数的引入对程序运行时的影响，掌握其使用。

二、实验内容

虚函数是在类中被声明为 virtual 的成员函数，当编译器看到通过指针或引用调用此类

函数时，对其执行动态联编，即通过指针（或引用）指向的类的类型信息决定该函数是哪个类的。通常此类指针或引用都是声明为基类的，它可以指向基类或派生类的对象。多态指同一个方法根据其所属的不同对象可以有不同的行为。

虚函数是 C++ 中用于实现多态（polymorphism）的机制。核心理念就是通过基类访问派生类定义的函数。

1. 输入下面程序，并分析结果。

```
#include <iostream>
#include <complex>
using namespace std;

class Base
{
public:
    Base() {cout <<"Base - ctor"<< endl;}
    ~Base() {cout <<"Base - dtor"<< endl;}
    virtual void f(int){cout <<"Base::f(int)"<< endl;}
    virtual void f(double){cout <<"Base::f(double)"<< endl;}
    virtual void g(int i = 10){cout <<"Base::g()"<< i << endl;}
};
class Derived : public Base
{
public:
    Derived() {cout <<"Derived - ctor" << endl;}
    ~Derived(){cout <<"Derived - dtor"<< endl;}
    void f(complex < double >) {
    cout <<"Derived::f(complex)"<< endl;
}
void g(int i = 20){
    cout <<"Derived::g()"<< i << endl;
}
};
int main()
{
cout << sizeof(Base)<< endl;
cout << sizeof(Derived)<< endl;

Base b;
Derived d;
Base * pb = new Derived;
b.f(1.0);
d.f(1.0);
pb -> f(1.0);
b.g();
d.g();
pb -> g();
delete pb;
return 0;
}
```

2. 输入下面程序,分析运行结果。

```
#include <iostream>
using namespace std;
class Base
{
public:
Base():data(count)
{
    cout <<"Base - ctor"<< endl;
    ++count;
}
~Base()
{
    cout <<"Base - dtor"<< endl;
    --count;
}
static int count;
int data;
};
int Base::count;
class Derived : public Base
{
public:
Derived():data(count),data1(data)
{
    cout <<"Derived - ctor"<< endl;
    ++count;
}
~Derived()
{
    cout <<"Derived - dtor"<< endl;
    --count;
}
static int count;
int data1;
int data;
};
int Derived::count = 10;
int main()
{
cout << sizeof(Base)<< endl;
cout << sizeof(Derived)<< endl;

Base * pb = new Derived[3];
cout << pb[2].data << endl;
cout <<((static_cast<Derived *>(pb)) + 2) -> data1 << endl;
delete[] pb;

cout << Base::count << endl;
cout << Derived::count << endl;
```

```
return 0;
}
```

三、实验要求

1. 写出程序，并调试程序，给出测试数据和实验结果。
2. 整理上机步骤，总结经验和体会。
3. 完成实验报告和上交程序。

实验 5.2 抽象类的使用

一、实验目的

1. 了解抽象类的概念。
2. 灵活应用抽象类。

二、实验内容

1. 输入下面程序，分析编译错误信息。

```
#include <iostream>
#include <new>
#include <assert.h>
using namespace std;
class Abstract
{
public:
    Abstract()
    {
        cout << "in Abstract()\n";
    }
    virtual void f() = 0;
};
int main()
{
    Abstract * p = new Abstract;
    p->f();
    return 0;
}
```

2. 基类 Shape 类是一个表示形状的抽象类，area()为求图形面积的函数。请从 Shape 类派生三角形类(Triangle)、圆类(Circles)，并给出具体的求面积函数。

```
#include <iostream.h>
class shape
{
public:
virtual float area( ) = 0;
};
```

3. 定义一个抽象类Base,在该类中定义一个纯虚函数“virtual void abstractMethod()=0;”,派生一个基于Base的派生类Derived,在派生类Derived的abstractMethod()方法中输出“Derived::abstractMethod is called”,最后编写主函数,其内容如下:

```
int main()
{
    Base * pBase = new Derived;
    pBase->abstractMethod();
    delete pBase;
    return 0;
}
```

分析运行结果。

三、实验要求

1. 写出程序,并调试程序,给出测试数据和实验结果。
2. 整理上机步骤,总结经验和体会。
3. 完成实验报告和上交程序。

实验5.3 运算符重载和友元

一、实验目的

1. 掌握运算符重载和友元的概念。
2. 掌握使用友元重载运算符的方法。

二、实验内容

1. 设计一个类,用自己的成员函数重载运算符,使对整型的运算符=、+、-、*、/适用于分数运算。

要求:

(1) 输出结果是最简分数(可以是带分数)。

(2) 分母为1,只输出分子。

2. 用友元函数重载运算符,使对整型的运算符=、+、-、*、/适用于分数运算。

三、实验要求

1. 写出程序,并调试程序,给出测试数据和实验结果。
2. 整理上机步骤,总结经验和体会。
3. 完成实验报告和上交程序。