The C++ Programming Language

运算符重载

陈辰
chenc@fudan.edu.cn
复旦大学软件学院

运算符重载

- 在C++中有这样的情况，同一个类型或运算符在不同的类中代表不同的意思或者实施不同的运算，这就是面向对象的三大特点之一的多态。
- 运算符重载是使同一个运算符作用于不同类型的数据时具有不同的行为。例如，我们声明了一个点类point和它的对象point p1(1，1)，p2(3，3)，并希望能使用“+”运算符实现表达式“p1+p2”，就需要重载“+”运算符。
- 运算符重载是实质上将运算对象转化为运算函数的实参，并根据实参的类型来确定重载的运算函数。
- 运算符重载和类型重载是多态的另外两种表现形式。

例1 通过函数来实现复数相加。

```
#include <iostream>
using namespace std;
class Complex                                   //定义Complex类
{public:
Complex( ){real=0;imag=0;}                      //定义构造函数
Complex(double r,double i){real=r;imag=i;}     //构造函数重载
Complex complex_add(Complex &c2);               //声明复数相加函数
void display( );                                //声明输出函数
 private:
double real;                                    //实部
double imag;                                    //虚部
};

Complex Complex::complex_add(Complex &c2)
{Complex c;
c.real=real+c2.real;
```

```
c.imag=imag+c2.imag;
return c;}

void Complex :: display( )                          //定义输出函数
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}

int main( )
{Complex c1(3,4),c2(5,-10),c3;                      //定义3个复数对象
c3=c1.complex_add(c2);                              //调用复数相加函数
cout<<"c1="; c1.display( );                         //输出c1的值
cout<<"c2="; c2.display( );                         //输出c2的值
cout<<"c1+c2="; c3.display( );                      //输出c3的值
return 0;
}
```

运行结果如下：

```
c1=(3+4i)
c2=(5-10i)
c1+c2=(8,-6i)
```

结果无疑是正确的，但调用方式不直观、太烦琐，使人感到很不方便。能否也和整数的加法运算一样，直接用加号“+”来实现复数运算呢？如

c3=c1+c2;

编译系统就会自动完成c1和c2两个复数相加的运算。如果能做到，就为对象的运算提供了很大的方便。这就需要对运算符“+”进行重载。

运算符重载的方法

运算符重载的方法是定义一个重载运算符的函数，在需要执行被重载的运算符时，系统就自动调用该函数，以实现相应的运算。也就是说，运算符重载是通过定义函数实现的。运算符重载实质上是函数的重载。

重载运算符的函数一般格式如下：

函数类型 operator 运算符名称 (形参表列)

{ 对运算符的重载处理 }

例如，想将“+”用于Complex类(复数)的加法运算，函数的原型可以是这样的：

Complex operator+ (Complex& c1,Complex& c2);

在定义了重载运算符的函数后，可以说： 函数operator+重载了运算符+。

为了说明在运算符重载后，执行表达式就是调用函数的过程，可以把两个整数相加也想像为调用下面的函数：

```
int operator + (int a,int b)
{return (a+b);}
```

如果有表达式5+8，就调用此函数，将5和8作为调用函数时的实参，函数的返回值为13。这就是用函数的方法理解运算符。

可以在例子程序的基础上重载运算符“+”，使之用于复数相加。

运算符重载的规则

1. 只能重载C++中已有的运算符，不能臆造新的运算符；

2. 类属关系运算符“.”、作用域分辨符“::”、成员指针运算符“*”、sizeof运算符和三目运算符“？：”不能重载。

3. 重载之后运算符的优先级和结合性都不能改变；单目运算符只能重载为单目运算符，双目运算符只能重载为双目运算符；重载运算符的函数不能有默认的参数；

4. 运算符重载后的功能应当与原有功能相类似。必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象(或类对象的引用)。

5. 重载运算符含义必须清楚，不能有二义性。用于类对象的运算符一般必须重载，但运算符“=”和“&”不必用户重载。

运算符的重载形式有两种：

(1) 重载为类的成员函数；

(2) 重载为类的友元函数。

重载运算符有哪些限制？

- （1）不可臆造新的运算符。必须把重载运算符限制在C++语言中已有的运算符范围内的允许重载的运算符之中。
- （2）重载运算符坚持4个“不能改变”：
  - 不能改变运算符操作数的个数；
  - 不能改变运算符原有的优先级；
  - 不能改变运算符原有的结合性；
  - 不能改变运算符原有的语法结构；

将运算符重载为类的成员函数

将运算符重载为类的成员函数就是在类中用关键字operator定义一个成员函数，函数名就是重载的运算符。运算符如果重载为类的成员函数，它就可以自由地访问该类的数据成员。

运算符重载为类的成员函数的一般格式为：

```
〈类型〉〈类名〉:: operator 〈要重载的运算符〉（形参表）
{
    函数体
}
```

其中，类型为运算符重载函数的返回类型。类名为成员函数所属类的类名，〈operator〉〈重载运算符〉即为重载函数名。形参为参加运算的对象或数据。

运算符重载为类的友元函数的一般语法形式为：

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

其中：

(1) 函数类型：重载运算符的返回值类型，即运算结果 类型；

(2) operator：定义运算符重载函数的关键字。

(3) 运算符：要重载的运算符名称。

(4) 形参表：给出对象和类型。

(5) friend：运算符重载为友元函数时的关键字。

运算符重载为成员函数

1. 双目运算：oprd1 B oprd2

- 把B重载为oprd1所属类的成员函数，只有一个形参，形参的类型是oprd2所属类。例如，经过重载之后，表达式oprd1 + oprd2就相当于函数调用

oprd1.operator +(oprd2)。

2. 单目运算

1）前置单目运算：U oprd

把U重载为oprd所属类的成员函数，没有形参。例如，“++”重载的语法形式为：

<函数类型> operator ++( );

++ oprd 就相当于函数调用oprd.operator ++( );

2) 后置单目运算 oprd V

运算符V重载为oprd所属类的成员函数，带有一个整型(int)形参。例如，后置单目运算符“- -”重载的语法形式为：

<函数类型> operator - - (int);

oprd-- 就相当于函数调用oprd.operator - - (0);

【例1改】 复数的加减运算符重载

```
#include <iostream>
using namespace std;
class Complex
{
    private:
        float real,image;
    public:
        Complex(float r=0,float i=0);
        Complex Add(const Complex &c);  //定义一个Add函数
        Complex operator+(const Complex &c); //重载+运算符
        Complex operator-(const Complex &c); //重载-运算符
        Complex& operator+=(const Complex &c);
//重载+=运算符，复合赋值操作符必须返回左操作数的引用，必须定义为成员函数
        Complex& operator=(const Complex &other);
    //重载=运算符，赋值运算符必须返回对*this的引用，
        void Show(int i);
};
```

```
Complex::Complex(float r,float i)
{
    real=r;  image=i;
}
void Complex::Show(int i)
{  //一般情况下，这里不应该有参数i，本例的目的是为了区分不同的复数，便于观看结果
    cout<<"复数：c"<<i<<"="<<real;
    if(image>0)
        cout<<"+"<<image<<"i"<<endl;
    if(image<0)
        cout<<image<<"i"<<endl;
}
```

```
Complex  Complex::Add(const Complex &c)
{
    Complex t;
    t.real=this->real+c.real;
    t.image=this->image+c.image;
    return t;
}
Complex  Complex::operator+(const Complex
   &c)
{
    Complex t;
    t.real=this->real+c.real;
    t.image=this->image+c.image;
    return t;
}
```

```
Complex  Complex::operator-(const Complex &c)
{
    Complex t;
    t.real=this->real-c.real;
    t.image=this->image-c.image;
    return t;
}
Complex&  Complex::operator+=(const Complex &c)
{
    real=real+c.real;
    image=image+c.image;
    return *this;
}
Complex& Complex::operator=(const Complex & other)
{
    if(this == &other)
        return *this;
    this->real=other.real;
    this->image=other.image;
    return *this;
}
```

```
int main()
{
    Complex c1(12,35),c2(20,46),c3,c4,c5,c6;
    c1.Show(1);
    c2.Show(2);
    c3=c1.Add(c2);
    c3.Show(3);
    c4=c1+c2;
    c4.Show(4);
    c2+=c1;
    c2.Show(2);
    c5=c1-c2;
    c5.Show(5);
    return 0;
}
```

如果我们把“operator+”看成函数名，可以在main()函数中写出如下语句：

c3=c1.Add(c2);

c3=c1.operator+(c2);

这时，operator+就完全是一个函数了（它本质上就是函数），.Add()和operator+的作用和功能完全相同，只是表现形式有些区别。

【例2】一元运算符重载，在Time类（描述时间的类，用三个数据成员分别存放时、分和秒）中重载自加运算符，即一个时间加上n秒后形成一个新的的时间。

```
class Time
{
    private:
        int hour,minute,second;
    public:
        Time(int h=0,int m=0,int s=0);
        //其他构造函数省略 如Time(Time& other);
        void Show();//显示时：分：秒的成员函数
        Time& operator++(); //定义前置++运算符重载成员函数
        Time operator++(int);//定义后置++运算符重载成员函数
        Time& operator=(const Time& other); //重载=运算符
};
```

```
Time::Time(int h,int m,int s)
{
    hour=h;
    minute=m;
    second=s;
}
void Time::Show()
{

   cout<<hour<<":"<<minu
   te<<":"<<second<<endl;
}
```

```
Time& Time::operator++()
{  //先加，然后再返回
   second++;
   if(second==60)
   {
      second=0;
      minute++;
      if(minute==60)
      {
         minute=0;
         hour++;
         if(hour==24) hour=0;
      }
   }
   return *this;
}
```

```
Time Time::operator++(int)
{
    //返回原来的值，再加
    Time temp=*this;
    second++;
    if(second==60)
    {
        second=0;
        minute++;
        if(minute==60)
        {
            minute=0;
            hour++;
            if(hour==24) hour=0;
        }
    }
    return temp;
}
```

```
Time& Time::operator=(const Time& other)
{
    if(this == &other)
        return *this;
    this->hour=other.hour;
    this-> minute =other. minute;
    this-> second =other. second;
    return *this;
}
```

```
int main()
{
    Time t1(10,25,52),t2,t3;
    t1.Show();
    t2=++t1;
    t1.Show();
    t2.Show();
    t3=t1++;
    t3.Show();
    t1.Show();
}
```

对于++（或--）运算符的重载，因为编译器不能区分出++（或--）是前置的还是后置的，所以要加上(int)来区分。

```
operator++();   //重载前置++
operator++(int); //重载后置++
operator--();   //重载前置--
operator--(int); //重载后置--
```

运算符重载的一些特点如下：

(1) 运算符重载函数名必须为：operator<运算符>。

(2) 运算符的重载是通过运算符重载函数来实现的。对于二元运算符重载函数，函数的参数通常为一个即右操作数，运算符的左操作数为调用重载函数的对象。对于一元运算符重载函数，运算符的左操作数或右操作数为调用重载函数的对象。

(3) 运算符重载函数的返回类型：若对象进行运算后的结果类型仍为原类型，则运算符重载函数的返回类型应为原类型。

```
//例3：演示重载二维点point的“+”、“-”运算符
#include <iostream.h>

class point
{
    private:
        float x, y ;
    public:
        point( float xx=0,  float yy=0 ) { x=xx;  y=yy; }
        float get_x( ){ return x ; }
        float get_y( ){ return y ; }
        point operator + (point p1);          //重载运算符“+”
        point operator - (point p1);          //和“-”为成员函数
};
```

```
point point::operator+( point q )
{ return  point ( x+q.x,  y+q.y ) ; }
point point::operator - ( point q )
{ return  point ( x-q.x,  y-q.y ) ; }

void main( )
{
   point  p1(3, 3), p2(2, 2), p3, p4 ;        //声明point类的对象
   p3=p1+p2 ;                                 //两点相加
   p4=p1-p2 ;                                 //两点相减
   cout<<"p1+p2: x="<<p3.get_x( )<<", y="<<p3.get_y(
)<<endl;
   cout<<"p1- p2: x="<<p4.get_x( )<<", y="<<p4.get_y(
)<<endl;
}
```

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

程序运行结果为：

p1+p2: x=5，y=5

p1- p2: x=1，y=1

例4 定义一个字符串类String，用来存放不定长的字符串，重载运算符 “==”，“<”和 “>”，用于两个字符串的等于、小于和大于的比较运算。

为了使读者便于理解程序，同时也使读者了解建立程序的步骤，下面分几步来介绍编程过程。

(1) 先建立一个String类：

```
#include <iostream>
using namespace std;
class String
{public:
String( ){p=NULL;}                    //默认构造函数
String(char *str);                    //构造函数
void display( );
private:
char *p;                              //字符型指针，用于指向字符串
};
```

```
String :: String(char *str)                    //定义构造函数
{p=str;}                                       //使p指向实参字符串

void String :: display( )                      //输出p所指向的字符串
{cout<<p;}

int main( )
{String string1("Hello"),string2("Book");
string1.display( );
cout<<endl;
string2.display( );
return 0;
}
```

运行结果为

```
Hello
Book
```

(2) 有了这个基础后，再增加其他必要的内容。现在增加对运算符重载的部分。为便于编写和调试，先重载一个运算符“>”。程序如下：

```
#include <iostream>
#include <string>
using namespace std;
class String
{public:
String( ){p=NULL;}
String(char *str);
friend bool operator>(String &string1,String &string2);//声明运算符函数为友元函数
void display( );
private:
char *p;                                    //字符型指针，用于指向字符串
};
String :: String(char *str)
{p=str;}
```

```
void String :: display( )                          //输出p所指向的字符串
{cout<<p;}
bool operator>(String &string1,String &string2)     //定义运算符重载函数
{if(strcmp(string1.p,string2.p)>0)
return true;
else return false;
}

int main( )
{String string1("Hello"),string2("Book");
cout<<(string1>string2)<<endl;
}
```

程序运行结果为1。

这只是一个并不很完善的程序，但是，已经完成了实质性的工作了，运算符重载成功了。其他两个运算符的重载如法炮制即可。

(3) 扩展到对3个运算符重载。

在String类体中声明3个成员函数：

```
friend bool operator> (String &string1, String &string2);
friend bool operator< (String &string1, String &string2);
friend bool operator==(String &string1, String& string2);
```

在类外分别定义3个运算符重载函数：

```
bool operator>(String &string1,String &string2)          //对运算符“>”重载
{if(strcmp(string1.p,string2.p)>0)
return true;
else
return false;
}

bool operator<(String &string1,String &string2)          //对运算符“<”重载
{if(strcmp(string1.p,string2.p)<0)
return true;
else
```

```
return false;
}

bool operator==(String &string1,String &string2)      //对运算符“==”重载
{if(strcmp(string1.p,string2.p)==0)
return true;
else
return false;
}
```

再修改主函数：

```
int main( )
{String string1("Hello"),string2("Book"),string3("Computer");
cout<<(string1>string2)<<endl;            //比较结果应该为true
cout<<(string1<string3)<<endl;            //比较结果应该为false
cout<<(string1==string2)<<endl;           //比较结果应该为false
return 0;
}
```

运行结果为

1

0

0

结果显然是对的。到此为止，主要任务基本完成。

(4) 再进一步修饰完善，使输出结果更直观。下面给出最后的程序。

```
#include <iostream>
using namespace std;
class String
{public:
String( ){p=NULL;}
String(char *str);
friend bool operator>(String &string1,String &string2);
friend bool operator<(String &string1,String &string2);
friend bool operator==(String &string1,String &string2);
```

```
void display( );
private:
char *p;
};
String :: String(char *str)
{p=str;}

void String :: display( )                    //输出p所指向的字符串
{cout<<p;}

bool operator>(String &string1,String &string2)
{if(strcmp(string1.p,string2.p)>0)
return true;
else
return false;
}

bool operator<(String &string1,String &string2)
```

```
{if(strcmp(string1.p,string2.p)<0)
return true;
else
return false;
}

bool operator==(String &string1,String &string2)
{if(strcmp(string1.p,string2.p)==0)
return true;
else
return false;
}

void compare(String &string1,String &string2)
{if(operator>(string1,string2)==1)
{string1.display( );cout<<">";string2.display( );}
else
if(operator<(string1,string2)==1)
```

```
{string1.display( );cout<<"<";string2.display( );}
else
if(operator==(string1,string2)==1)
{string1.display( );cout<<"=";string2.display( );}
cout<<endl;
}
int main( )
{String string1("Hello"),string2("Book"),string3("Computer"),string4("Hello");
compare(string1,string2);
compare(string2,string3);
compare(string1,string4);
return 0;
}
```

运行结果为

```
Hello>Book
Book<Computer
Hello==Hello
```

增加了一个compare函数，用来对两个字符串进行比较，并输出相应的信息。这样可以减轻主函数的负担，使主函数简明易读。

通过这个例子，不仅可以学习到有关双目运算符重载的知识，而且还可以学习怎样去编写C++程序。

这种方法的指导思想是： 先搭框架，逐步扩充，由简到繁，最后完善。边编程，边调试，边扩充。千万不要企图在一开始时就解决所有的细节。类是可扩充的，可以一步一步地扩充它的功能。最好直接在计算机上写程序，每一步都要上机调试，调试通过了前面一步再做下一步，步步为营。这样编程和调试的效率是比较高的。读者可以试验一下。

赋值运算符重载一般包括以上几个步骤，首先要检查是否自赋值，如果是要立即返回，如果不返回，后面的语句会把自己所指空间删掉，从而导致错误；第二步要释放原有的内存资源；第三步要分配新的内存资源，并复制内容；第四步是返回本对象的引用。如果没有指针操作，则没有第二步操作。

赋值运算符与拷贝构造函数在功能上有些类似，都是用一个对象去填另一个对象，但拷贝构造函数是在对象建立的时候执行，赋值运算符是在对象建立之后执行。

运算符重载为友元函数

1. 双目运算：oprd1 B oprd2

双目运算符B重载为oprd1所属类的友元函数，该函数有两个形参，表达式oprd1 B oprd2就相当于函数调用operator B(oprd1，oprd2)。

2. 单目运算

1）前置单目运算U oprd

前置单目运算符U重载为oprd所属类的友元函数，表达式U oprd相当于函数调用operator U(oprd)。

2）后置单目运算oprd V

后置单目运算符V重载为oprd所属类的友元函数，表达式oprd V相当于函数调用operator V(oprd，int)。

运算符重载为友元函数

二元运算符重载为友元函数的一般格式为：

```
friend <函数返回类型> operator <重载运算符>(<形参1>, <形参2>);

<函数返回类型> operator <重载运算符>(<形参1>, <形参2>)
{
  函数体;
}
```

一元运算符重载为友元函数的一般格式为：

```
<函数返回类型> operator<一元运算符> (类名 &对象)
{
  函数体
}
```

运算符重载为友元函数

其中，函数返回类型为运算符重载函数的返回类型。operator<重载运算符>为重载函数名。当重载函数作为友元普通函数时，重载函数不能用对象调用，所以参加运算的对象必须以形参方式传送到重载函数体内，在二元运算符重载函数为友元函数时，形参通常为二个参加运算的对象。

```
//例5:
//演示使用运算符重载为友元函数的方法重做两点加减法运算
#include <iostream.h>
class point
{
   private:
      float x, y ;
   public:
      point ( float xx=0, float yy=0 ) { x=xx; y=yy; }
      float get_x( ) { return x ; }
      float get_y( ) { return y ; }
      friend point operator + (point p1, point p2) ; //重载 “+” 和
      friend point operator-(point p1, point p2) ; // “-” 为友元函
数
};
```

```
point operator + ( point p1, point p2 )
{ return  point (p1.x+p2.x,  p1.y+p2.y) ; }

point operator - ( point p1, point p2 )
{ return  point (p1.x- p2.x,  p1.y-p2.y) ; }
void main( )
{ point  p1(3, 3),  p2(2, 2),  p3,  p4 ; //声明point类的对象
   p3=p1+p2;                                    //两点相加
   p4=p1- p2;                                   //两点相减
   cout<<"p1+p2: x="<<p3.get_x( )<<", y="<<p3.get_y( )<<endl
;
   cout<<"p1- p2: x="<<p4.get_x( )<<", y="<<p4.get_y( )<<endl
;
}
```

程序运行结果：p1+p2: x=5, y=5

【例6】用友元运算符重载函数实现复数的加减运算。

```
#include <iostream>
using namespace std;
class Complex
{
    private:
        float real,image;
    public:
        Complex(float r=0,float i=0);
        friend Complex operator+( const Complex &, const
    Complex &);
        friend Complex operator-( const Complex &, const
    Complex &);
        void Show(int i);
};
```

```
Complex::Complex(float r,float i)
{
    real=r;  image=i;
}
void Complex::Show(int i)
{
    cout<<"复数
   :c"<<i<<"="<<real;
    if(image>0)

   cout<<"+"<<image<<"i"<<en
   dl;
    if(image<0)
        cout<<image<<"i"<<endl;
}
```

```
Complex operator+( const Complex &c1, const
   Complex &c2)
{
    Complex t;
    t.real=c1.real+c2.real;
    t.image=c1.image+c2.image;
    return t;
}
Complex operator-( const Complex &c1, const
   Complex &c2)
{
    Complex t;
    t.real=c1.real-c2.real;
    t.image=c1.image-c2.image;
    return t;
}
```

```
int main()
{
    Complex c1(12,35),c2(20,46),c3,c4;
    c1.Show(1);
    c2.Show(2);
    c3=c1+c2;
    c3.Show(3);
    c4=c1-c2;
    c4.Show(4);
    return 0;
}
```

【例7】用友元运算符重载函数实现时间类的“++”运算符。

```
class Time
{
    public:
        Time(int h=0,int m=0,int s=0);
        //其他构造函数省略
        void Show();//显示时：分：秒的成员函数
        friend Time& operator++(Time &t);
        //定义前置++运算符重载成员函数
        friend Time operator++(Time &t,int);
        //定义后置++运算符重载成员函数
    private:
        int hour,minute,second;
};
```

其它运算符重载

- 1．比较运算符重载

  例如，<、>、<=、>=、==和!=。

- 2．赋值运算符重载

  例如，=、+=、-=、*=和/=。

这些运算符的重载比较简单。

参数和返回值

- 1）对于任何函数参数，如果仅需要从参数中读而不改变它，缺省地应当按const引用来传递它。普通算术运算符（像+和-号等）和布尔运算符不会改变参数，所以以const引用传递是使用的主要方式。当函数是一个类成员的时候，就转换为const成员函数。只是对于会改变左侧参数的赋值运算符（像+ =）和运算符‘=’，左侧参数才不是常量(constant)，但因为参数将被改变，所以参数仍然按地址传递。
- 2）应该选择的返回值取决于运算符所期望的类型。如果运算符的效果是产生一个新值，将需要产生一个作为返回值的新对象。

- 3）所有赋值运算符改变左值。为了使得赋值结果用于链式表达式（像A = B = C），应该能够返回一个刚刚改变了的左值的引用。但这个引用应该是const还是nonconst呢？虽然我们是从左向右读表达式A = B = C，但编译器是从右向左分析这个表达式，所以并非一定要返回一个nonconst值来支持链式赋值。然而人们有时希望能够对刚刚赋值的对象进行运算，例如(A = B).foo( )，这是B赋值给A后调用foo( )。因此所有赋值运算符的返回值对于左值应该是nonconst引用。
- 4）对于逻辑运算符，人们希望至少得到一个int返回值，最好是bool返回值。（在大多数编译器支持C + +内置bool类型之前开发的库函数使用int或typedef等价物）。
- 5）当有不同类型的参数时，要注意参数的自动类型转换（称为隐式类型转换），隐式类型转换在很多地方可以简化程序的书写，但是也可能留下隐患。

【例8】在Complex类中重载运算符+，使该类对象能加上一个复数形成一个新的复数，能加上一个整数或加上一个实数形成一个新的复数。

```
#include <iostream>
using namespace std;
class Complex
{
    private:
        float real,   image;
    public:
        Complex(float r=0,float i=0);
        Complex operator+(const Complex
    &c);
        Complex operator+(const int k);
        Complex operator+(const float f);
        void Show(int i);
};
```

```
int main()
{
    Complex c1(12,35),c2(20,46),c3,c4,c5,c6;
    int i=5;
    float f=7.7;
    c1.Show(1);
    c2.Show(2);
    c3=c1+i;    //编译正确
    c3.Show(3);
    c4=c1+f;    //编译正确
    c4.Show(4);
    c5=c1+1.5;  //编译错误，可用c5=c1+(int)1.5;或者
    c5=c1+(float)1.5;
    c5.Show(5);
    return 0;
}
```

在进行运算符重载时，我们要注意：

1）重载不能改变该运算符用于内置类型时的含义，正如程序员不能改变运算符+用于两个int型相加时的含义。

2）运算符函数的参数至少有一个必须是类的对象或者类的对象的引用，这种规定可以防止程序员运用运算符改变内置类型的函义。

3）重载不能改变运算符的优先级。

4）重载不能改变运算符的结合律。

5）重载不能改变运算符操作数的个数。比如+需要两个操作数，则重载的+也必须要有两个操作数。

6）不存在用户定义的运算符，即不能编写目前运算符集合中没有的运算符。不能这样做的部分原因是难以决定其优先级，另一部分原因是没有必要增加麻烦。

- 当一个重载操作符的含义不明显时，给操作取一个名字更好。对于很少用的操作，使用命名函数通常比用操作符更好。如果不是普通操作，没必要为简洁而使用操作符。
- 重载逗号、取地址、逻辑与、逻辑或等操作符通常不是好做法。这些操作符具有有用的内置含义，如果我们定义了自己的版本，就不能再使用这些内置含义。
- 如果类定义了相等操作符，也应该定义不等操作符!=。同样，如果定义了<，则可能应该定义四个关系操作符（>，>=，<，<=）。

- 为类设计重载操作符的时候，必须选择是将操作符设置为类成员函数还是友元函数。在某些情况下，程序员没得选择，操作符必须是成员函数。有些指导原则有助于决定将操作符设置为类成员还是友元函数：
  - 1）赋值（=）、下标（[]）、调用（( )）和成员访问箭头（->）等操作符必须定义为成员，将这些操作符定义为非成员函数在编译时标记为错误
  - 2）像赋值一样，复合赋值操作符通常应定义为类的成员函数。定义成非成员函数不会出现编译错误
  - 3）改变对象状态或与给定类型紧密联系的其他一些操作符，如自增、自减等通常定义为类成员函数
  - 4）对称的操作符，如算术操作符、相等操作符、关系操作符和位操作符，最好定义为非成员函数。

重载流插入和流提取运算符

istream 和 ostream 是 C++ 的预定义流类，cin 是 istream 的对象，cout 是 ostream 的对象。运算符 << 由ostream 重载为插入操作，运算符 >> 由 istream 重载为提取操作， 用于输出和输入基本类型数据。可用重载 << 和 >>运算符 ，用于输出和输入用户自定义的数据类型，必须定义为类的友元函数。

1）输出操作符<<的重载

输出操作符重载函数形式为：

ostream & operator << (ostream &,const 自定义类 &);

第一个参数和函数的类型都必须是ostream&类型，第二个参数是对要进行输出的类类型的引用，它可以是const，因为一般而言输出一个对象不应该改变对象。返回类型是一个ostream引用，通常是输出操作符所操作的ostream对象。

【例9】为Date类重载提取运算符

```
#ifndef DATE_H_INCLUDED
#define DATE_H_INCLUDED
#include <iostream>
using namespace std;
class Date
{
    public:
        Date(int y,int m,int d);
        bool isLeapYear();
        void print();
        friend ostream &operator<<(ostream &output,Date &d);
    private:
        int  year;
        int  month;
        int  day;
};
#endif // DATE_H_INCLUDED
```

```
#include <iostream>
#include "Date.h"
using namespace std;

Date::Date(int y,int m,int d)
{
    year=y;
    month=m;
    day=d;
}
bool Date::isLeapYear()
{
    return (year%4==0 && year%100!=0)||(year%400==0);
}
```

```
void Date::print()
{
    cout<< year<<"-"<<
   month<<"-"<< day<<endl;
 }
ostream &operator<<(ostream
   &output,Date &d)
{
    output<<d.year<<  “-
   ”  <<d.month<<  “-”  <<d.day ;
    return output;
}
```

```
#include <iostream>
#include "Date.h"
using namespace std;
int main()
{
    Date d1(2013,4,1);
    cout<<d1;
    return 0;
}
```

在重载函数中，由于operator<<()函数是Date类的友元函数，因此在使用Date类的数据成员和成员函数时必须指定对象。一般而言，输出操作符应该输出对象的内容，进行最小限度的格式化，不应该输出换行符。在主函数（main）中， cout的值被传递给output。由于函数返回的是ostream对象的引用，所以在主函数中可以将”<<”连接起来使用。

2. 输入操作符>>的重载

输入操作符重载函数形式为：

```
istream & operator >> (istream &, 自定义类 &);
```

与输出操作符类似，输入操作符的第一个形参是一个引用，指向要读的流，并且返回的也是同一个流的引用。第二个形参是对要读入的对象的非const 引用，该形参必须为非const，因为输入操作符的目的是将数据读到这个对象中。和输出操作符不同的是输入操作符必须处理错误和文件结束的可能性。

【例10】重载输入操作符，在上例的基础上加上Date类对象能用“>>”输入数据。

```
class Date
{
    public:
        Date(){year=0;month=0;day=0;}
        Date(int y,int m,int d);
        Date(string s);
        bool isLeapYear();
    private:
        int  year;
        int  month;
        int  day;
    friend ostream& operator<<(ostream &output,const Date
    &d);
    friend istream& operator>>(istream &input,Date &d);
};
```

```
istream& operator>>(istream &input,Date &d)
{
    input>>d.year>>d.month>>d.day;
    if(!input)
    {
        d=Date();
    }
    return input;
}
```

```
int main()
{
    Date d1(2013,3,20);
    cout<<d1<<endl;
    Date d2("2013-04-01");
    cout<<d2<<endl;
    cout<<"Input date(year
   month day):";
    cin>>d2;
    cout<<d2<<endl;
    return 0;
}
```

类型重载

C++中提供了类型转换函数，可以将一种类类型对象转换成另一种类类型的对象，这就是类型重载。类型转换函数必须由用户在类中定义为成员函数，其一般格式为：

```
class <类名1>
{
   public:
      operator <类名2>( );
      ……
 };
<类名1>::operator <类名2>()
{      函数体;  }
```

其中operator <类名2>为转换函数的函数名，转换函数的作用是将类型1的对象转换成类型2的对象。类中类型转换函数必须是非静态的成员函数，不能定义成友元函数，无返回值类型且不带参数。

【例11】定义一个时间类，类中数据成员为时、分、秒。编写类型转换函数，将时、分、秒变成一个以秒为单位的等价实数。

```
#include <iostream>
using namespace std;
class Time
{
    private:
        int hour,minute,second;
    public:
        Time(int h=0,int m=0,int s=0);
        void Show();//显示时：分：秒的成员函数
        operator float();
};
Time::Time(int h,int m,int s)
{
    hour=h;
    minute=m;
    second=s;   }
```

```
void Time::Show()
{

   cout<<hour<<":"<<minute<<":"<<second<
   <endl;
}
Time::operator float()
{
    float sec;
    sec=hour*3600+minute*60+second;
    //cout<<"second="<<sec<<endl;
    return sec;
}
```

```
int main()
{
    float s1,s2,s3;
    Time t(10,15,20);
    s1=t;
    s2=float(t);
    t.Show();
    s3=(float)t;

   cout<<"s1="<<s1<<'\t'<<"s2="<<s2<<
   '\t'<<"s3= "<<s3<<endl;
}
```

- //例12：演示比较运算符和赋值运算符重载的程序

```
#include <iostream.h>
class point
{  private:
       float  x, y ;
    public:
       point( float  xx=0,  float  yy=0 ) { x=xx; y=yy; }
      point( point &) ;
       ~point( ){ }
       bool operator == ( point ) ;
       bool operator !=  ( point ) ;
       point operator += ( point ) ;
       point operator -=  ( point) ;
       float get_x( ) { return x ; }
       float get_y( ) { return y ; }
};
```

```
point::point ( point &p )
{  x=p.x;  y=p.y; }

bool point::operator == ( point p )
{
   if( x == p.get_x( ) && y == p.get_y( ) )  return 1 ;
   else  return 0 ;
}

bool point::operator != ( point p )
{
   if( x != p.get_x( ) && y != p.get_y( ) )  return 1 ;
   else  return 0 ;
}

point point::operator += (point p)
{
   this->x += p.get_x( ) ;
   this->y += p.get_y( ) ;
   return *this ;
}
```

```
point point::operator -= (point p)
{  this->x -= p.get_x( ) ;
   this->y -= p.get_y( ) ;
   return *this ;
}

void main( )
{  point  p1(1, 2), p2(3, 4), p3(5, 6) ;
   cout<<"p1 == p2?  "<<( p1 ==p2 )<<endl ;
   cout<<"p1 !=  p2?  "<<( p1 != p2 )<<endl ;
   p3 += p1;
   cout<<"p3+=p1, p3: "<<p3.get_x( )<<","<<p3.get_y( )<<endl ;
   p3 -= p1;
   cout<<"p -= p1, p3:  "<<p3.get_x( )<<","<<p3.get_y( )<<endl ;
}
```

程序运行结果为：

p1==p2? 0

p1!= p2? 1

p3 += p1, p3: 6, 8

p3 -= p1, p3: 5, 6

下标运算符[ ]重载

- //例13：演示下标运算符[ ]重载例题

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

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

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

程序运行结果为： china

CHINA

4．运算符new和delete重载

```
//例14: 演示重载new和delete的程序。其中new通过
//malloc( )函数实现，new的操作数是申请内存单元的字节个数。
//delete通过free( )函数实现，它的操作数是一个指针，即告诉
//系统释放哪里的单元。

#include <iostream.h>
#include<malloc.h>

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

```
    void *operator new ( size_t  size)//size_t  unsigned integer
          { return  malloc( size ) ; }

    void operator delete( void *p )
    { free( p ) ; }

    void disp( )
    { cout<<"area: "<<length*width<<endl ; }
};

void main( )
{   rect  *p ;
    p = new rect ( 5, 9 ) ; //调用构造函数rect( 5, 9 )
    p->disp( ) ;
    delete  p ;
}
//程序运行结果为：area: 45
```

5．逗号运算符重载

```
//例15：演示重载逗号“,”和加号“+”运算符的程序

#include <iostream.h>
#include<malloc.h>

class point
{
   private:
      int x, y ;
   public:
      point( ){ } ;
      point( int xx, int yy )
     {  x=xx ;  y=yy ;  }
```

```
    point operator , (point r)
          {
              return r ;
          }
    point operator + (point r)
          {
        point t ;
        t.x=x + r.x ;
        t.y=y + r.y ;
              return  t ;
          }
          void disp( )
           { cout<<"area: "<<x*y<<endl ; }
} ;
```

```
void main( )
{
    point  p1(1, 2), p2(3, 4), p3(5, 6) ;
    p1.disp( ) ;
    p2.disp( ) ;
    p3.disp( ) ;
    p1=(p1, p2+p3, p3) ;     //返回右操作数p3的坐标
    p1.disp( ) ;
}
```

程序运行结果为：

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

6. 重载转换运算符

在C++中，数据类型转换对于基本数据类型有两种方式：

1、隐式数据类转换

2、显式数据类型转换，也叫强制类型转换。

对于自定义类型和类类型，类型转换操作是没有定义的。

强制类型转换使用“( )”运算符完成，在C++中我们可以将“( )”运算符进行重载，达到数据转换的目的。

- 转换运算符声明形式

  operator 类型名 () ;

- 特点

  1、没有返回值

  2、功能类似强制转换

我们以RMB类为例说明如何重载转换运算符

```
class RMB
{
 public:
        RMB(double value=0.0)
        {
             yuan =value;
             fen = (value-yuan)*100+0.5;
        }
        void ShowRMB()
        {
             cout<<yuan<< “元”
                  <<fen<< "分" <<endl;
        }
        operator double ()
        {
            return yuan+fen/100.0;
        }
private:
            int yuan, fen;
};
```

```
void main()
{
        RMB r1(1.01),r2(2.20);
        RMB r3;
        //显式转换类型
        r3 = RMB((double)r1+(double)r2);
        r3.ShowRMB();
        //自动转换类型
        r3=r1+2.40;
        r3.ShowRMB();
        //自动转换类型
        r3 =2.0-r1;
        r3.ShowRMB();
}
```

6.重载转换运算符

对于r3=r1+2.40;的系统工作

1、寻找重载的成员函数+运算符

2、寻找重载的友元函数+运算符

3、寻找转换运算符

4、验证转换后的类型是否支持+运算。

转换运算符重载一般建议尽量少使用。

小结

- 注意运算符重载的规则和限制；
- 重载运算符的时候要注意函数的返回类型；
- 前增量和后增量运算符的重载区别；
- 赋值运算符重载要注意内存空间的释放和重新申请；
- 转换运算符重载与构造函数、析构函数一样没有返回值，通过转换运算符重载可以在表达式中使用不同类型的对象，但要注意转换运算符重载不可滥用。

谢谢大家！