

C++高级语言程序设计

王晨宇

北京邮电大学网络空间安全学院

第6章 运算符重载

- **运算符重载**：重定义已有运算符，完成特定操作，即同一运算符根据不同的运算对象完成不同的操作。
- 运算符重载是面向对象程序设计的多态性之一。
- C++允许运算符重载。

[6.1 运算符重载的基本方法](#)

[6.2 运算符重载为成员函数](#)

[6.3 运算符重载为友元函数](#)

[6.4 几个特殊运算符的重载](#)

[6.5 运算符重载的规则](#)

[6.6 字符串类](#)

6.1 运算符重载的基本方法

- 重载运算符的目的：使运算符可以直接操作自定义类型的数据。

```
class Complex{  
    float real,imag;  
public:  
    Complex(float r=0,float i=0)  
    { real=r;imag=i; }  
    Complex add(const Complex &c)  
    { return Complex(real+c.real,image+c.image);}  
};  
void main( )  
{ Complex c1(3.2,4.6),c2(6.2,8.7),c3;  
  c3=c1.add(c2);  
};
```

问：能否简写成 $c3=c1+c2$;

答：运算符虽能直接操作基本数据类型的数据，但对于自定义类型的数据，若未重载有关运算符，则不能用这些运算符操作自定义类型的数据。

- 如何重载运算符

- 运算符重载函数：为自定义类型添加一个运算符重载函数，简称**运算符函数**。这样当该运算符操作指定的自定义类型数据时，系统自动调用该函数完成指定操作。

- 运算符函数分为：

- 成员运算符函数
 - 友元运算符函数。

- 说明运算符函数的语法格式：

返回值类型 operator @(参数表);

关键字

重载运算符，如：+、-等

- 举例

- 为Complex类添加一个完成两个Complex类型对象相加的成员运算符函数

```
Complex operator+(Complex &c)
```

```
{return Complex(real+c.real,image+c.image);}
```

- 设c1和c2为Complex类型的对象，则当遇到表达式c1+c2时，系统自动调用上面的运算符重载函数完成加法操作。

6.2 运算符重载为成员函数

- 定义格式:

```
class X{
```

```
...
```

```
public:
```

```
<type> operator @(<Arg>);
```

```
...
```

```
};
```

- 说明:

- @函数应说明为公有的，否则外界无法使用。
- 系统约定，当@函数为类成员时，其第一操作数为当前对象。因此，若@为一元运算符，则参数表为空；若@为二元运算符，则参数表的参数为第二操作数。
- 成员运算符函数的定义方法与普通成员函数相同。

- 例6.1 定义复数类，重载加(+)、减(-)、乘(*)、除(/)运算符，实现复数的四则运算；重载取负(-)运算符，实现复数的取负操作。

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

```
class Complex{                //复数类
    double real,image;
public:
    Complex(double r=0,double i=0){real=r;image=i;}
    void print( );
    Complex operator+(Complex&);//重载+: 复数+复数
    Complex operator-(Complex&);//重载-: 复数-复数
    Complex operator*(Complex&);//重载*: 复数*复数
    Complex operator/(Complex&);//重载/: 复数/复数
    Complex operator-( );      //重载-: -复数
};
```

```
void Complex::print( )  
{ if(image<0) cout<<real<<image<<"i\n";  
  else if(image>0) cout<<real<<'+'<<image<<"i\n";  
  else cout<<real<<endl;  
}
```

第一操作数为当前对象

```
Complex Complex::operator+(Complex &c)  
{ return Complex(real+c.real, image+c.image); }
```

```
Complex Complex::operator-(Complex &c)  
{ return Complex(real-c.real, image-c.image); }
```

```
Complex Complex::operator*(Complex &c)  
{ Complex temp;  
  temp.real=real*c.real-image*c.image;  
  temp.image=real*c.image+image*c.real;  
  return temp;  
}
```



```
Complex Complex::operator/(Complex &c)
{ Complex temp;
  double r=c.real*c.real+c.image*c.image;
  temp.real=(real*c.real+image*c.image)/r;
  temp.image=(image*c.real-real*c.image)/r;
  return temp;
}
```

```
Complex Complex::operator-( )
{ return Complex(-real,-image); }
```

```
void main(void)
{ Complex c1(5,2),c2(3,4),c3;
  c3= c1+c2; c3.print( );
  c3= c1+c2+c3; c3.print( );
  c3=c1-c2; c3.print( );
  c3=c1*c2; c3.print( );
  c3=c1/c2; c3.print( );
  c3= -c1; c3.print( );
}
```

实际通过调用 “+”运算符重载函数来完成，即：

c1.operator+(c2)

+、-、*和/运算符重载函数的返回值类型均为Complex，使它们可继续参加Complex类型数据的运算，如c1+c2+c3。

实际通过调用取负运算符重载函数来完成，即：

c1.operator-()

- 问题：例6.1将+运算符重载为类的成员函数实现了“复数+复数”操作，能否通过类似的方法实现“复数+实数”和“实数+复数”操作？
- 答：要实现“复数+实数”操作，可在复数类Complex中重载下列+运算符：

```
Complex Complex::operator+(double d)
```

```
{ return Complex(real+d, image); }
```

但要实现“实数+复数”操作，无法在复数类Complex中通过重载+运算符为类的成员函数来实现。原因是，将运算符重载为类的成员函数时，其第一操作数只能是当前对象，而不可能是其它类型的数据。

- 对于“实数+复数”这样的操作只能通过重载+运算符为类的友员函数来实现。

6.3 运算符重载为友元函数

- 定义格式:

```
class X{  
    ...  
public:  
    friend <type> operator@(<Arg>);  
    ...  
};
```

- 说明:

- 在函数原型前加关键字friend。
- 类的友元函数可置于类的任何位置。
- 友元函数无this指针，应在参数表中列出每个操作数。若@为一元运算符，则参数表应有一个参数；若@为二元运算符，则参数表应有两个参数，第一个参数为左操作数，第二个参数为右操作数。

- 例6.2 重载加(+)、减(-)、乘(*)、除(/)运算符为复数类的友元函数实现复数的四则运算；重载取负(-)运算符为复数类的友元函数实现复数的取负操作。

```
#include<iostream>
Using namespace std;
class Complex{
    double real,image;
public:
    Complex(double r=0,double i=0){real=r;image=i;}
    void print( );
    friend Complex operator+(const Complex&, const Complex&);
    friend Complex operator+(const Complex&, const double&);
    friend Complex operator+(const double&, const Complex&);
    friend Complex operator-(const Complex&, const Complex&);
    friend Complex operator*(const Complex&, const Complex&);
    friend Complex operator/(const Complex&, const Complex&);
    friend Complex operator-(const Complex&);
};
```

```
Complex operator+(const Complex& c1, const Complex& c2) //复数+复数
{ return Complex(c1.real + c2.real, c1.image + c2.image);}
```

```
Complex operator+(const Complex& c, const double& d) //复数+实数
{ return Complex(c.real + d, c.image);}
```

```
Complex operator+(const double& d, const Complex& c) //实数+复数
{ return Complex(c.real + d, c.image);}
```

```
Complex operator-(const Complex& c1, const Complex& c2) //复数-复数
{ return Complex(c1.real - c2.real, c1.image - c2.image);}
```

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

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

```
Complex operator-(const Complex& c1) //复数取负
{
    return Complex(-c1.real, -c1.image);
}
```

```
void Complex::print()
{
    if (image < 0) cout << real << image << "i\n";
    else if (image > 0) cout << real << '+' << image << "i\n";
    else cout << real << endl;
}
```

```
int main(void)
{ Complex c1(5,2),c2(3,4),c3;
  c3= c1+c2; c3.print( );
  c3=c1-c2; c3.print( );
  c3=c1*c2; c3.print( );
  c3=c1/c2; c3.print( );
  c3= -c1; c3.print( );
  c3=c1+8.2; c3.print( );
  c3=3.5+c2; c3.print( );
  return 0;
}
```

实际通过调用 “+”运算符重载函数来完成，即：
operator+(c1,c2)

实际通过调用取负运算符重载函数来完成，即：
operator-(c1)

- 从例6.1和例6.2可见，不管运算符的重载是通过成员函数实现还是通过友元函数实现，运算符的用法是相同的，但编译器所做的解释是不同的。

- 大多运算符既可重载为成员函数又可重载为友元函数，给编程带来了灵活性。但从类的封装性和成员数据的安全性角度考虑，重载运算符时应优先重载为成员函数，无法用成员函数重载时才考虑用友元函数。
- 举例：对于例6.1和例6.2来说只有“实数+复数”这种情况无法用成员函数重载加(+)运算符，其余情况均可。因此复数类定义成下列形式更好。

```
class Complex{                                //复数类
    double real,image;
public:
    Complex(double r=0,double i=0){real=r;image=i;}
    Complex operator+(Complex&);              //复数+复数
    Complex operator+(double);                //复数+实数
    friend Complex operator+(double,Complex&); //实数+复数
    ...
};
```


6.4 几个特殊运算符的重载

6.4.1 赋值运算符重载

6.4.2 ++和--运算符的重载

6.4.3 下标运算符的重载

6.4.4 函数调用运算符的重载

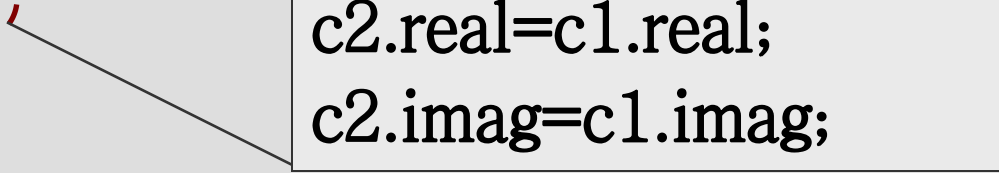
6.4.5 转换函数

6.4.1 赋值运算符重载

- 若没有为类重载赋值运算符，则编译系统自动为该类型生成一个缺省的赋值运算符，以实现同类对象之间对应数据成员的逐一赋值。

例：Complex c1(4,5),c2;

c2 = c1;



```
c2.real=c1.real;  
c2.imag=c1.imag;
```

- 何时需要为类重载赋值运算符：类中含有指向动态内存的指针。

- 例6.3 用缺省的赋值运算符产生的问题。

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

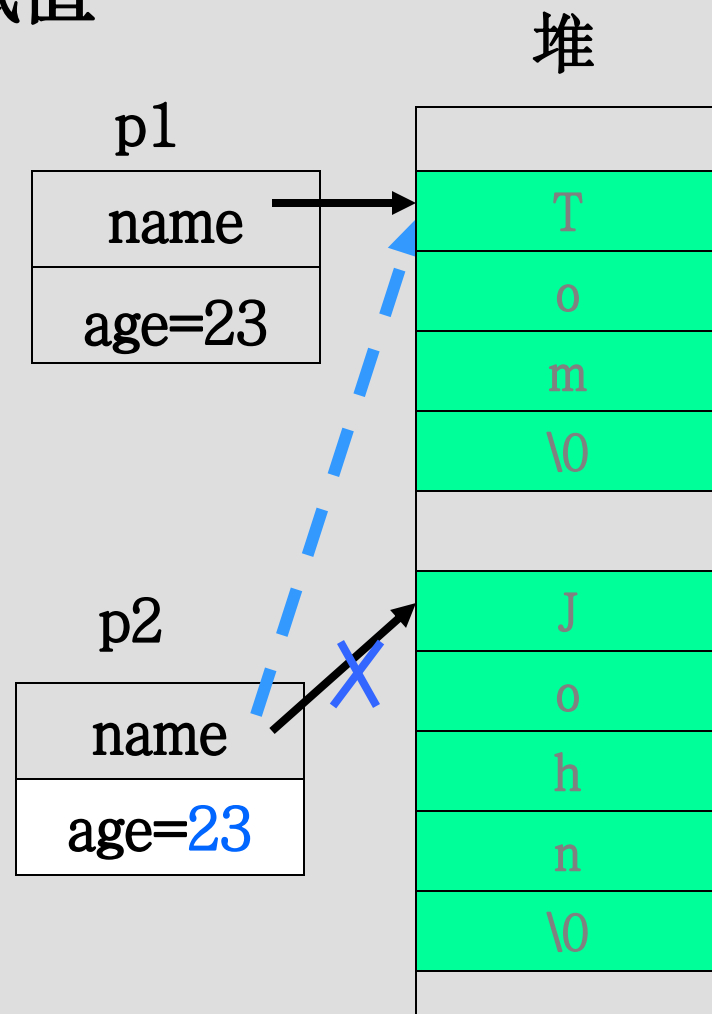
```
class Person{  
    char*name;  
    int age;  
public:  
    Person(char*n,int a)  
    { name=new char[strlen(n)+1];  
      strcpy(name,n);  
      age=a;  
    }  
    ~Person( ){ delete []name; }  
};
```

```

void main( )
{ Person p1("Tom",23),p2("John",18);
  p2=p1; //用缺省的赋值运算符赋值
}

```

- p2.name=p1.name;p2.age=p1.age;
- 存在问题:
 - p2对象的成员数据name所指动态内存丢失，无法释放。
 - p1和p2对象的成员数据name共用同一块动态内存，使两者的数据不独立，产生逻辑错误。
 - p1对象被撤销时，其成员数据name所指动态内存已被p2对象释放，导致一个运行错误。



解决方法:

- 为该类重载赋值运算符。C++语言规定:
 - 赋值运算符必须重载为类的成员函数;
 - 重载的赋值运算符不能被派生类继承。
- 例6.3定义的Person类应重载赋值运算符:

```
Person& Person::operator=(Person&p)
{ if(this==&p)    //避免对象自我赋值
    return *this;
  delete []name;
  name=new char[strlen(p.name)+1];
  strcpy(name,p.name);
  age=p.age;
  return *this;
}
```

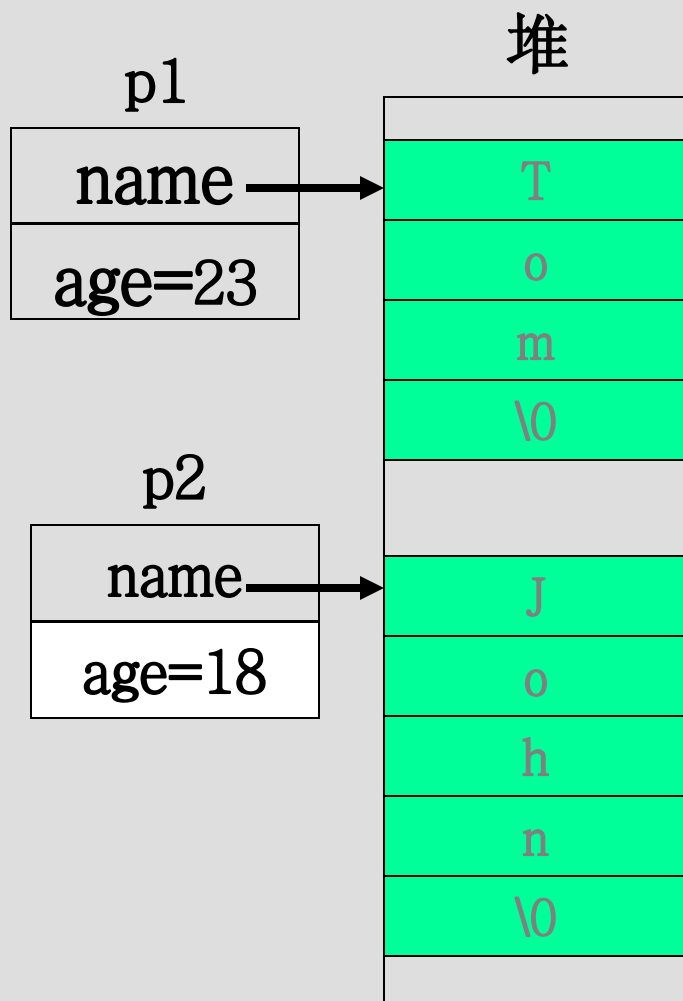


图. $p2=p1$ 前

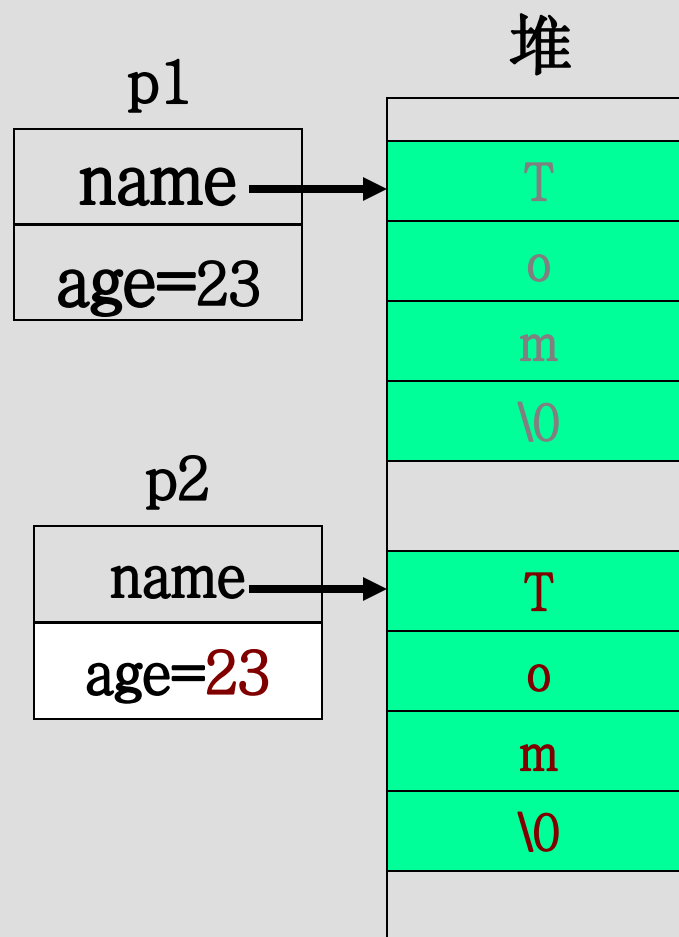


图. $p2=p1$ 后

6.4.2 ++和--运算符的重载

- “++”和 “--”运算符的重载相类似，下面以 “++”为例。
- “++”有前置和后置之分，为前置运算时，运算符重载为成员函数的格式为：

```
<type> operator++( );
```

为后置运算时，运算符重载函数的格式为：

```
<type> operator++(int);
```

其中，type是函数返回值的类型；参数int仅作后置标志，并无其它意义，可以给一个变量名，也可以不给出变量名。

- 例6.4 定义一个类描述时间，能分别存放时、分、秒。重载“++”运算符，实现时间对象的加1秒运算。

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

```
class Time{
protected:
    int hour,minute,second;
    void AddOne( )
    { second++;
      if(second==60)
      { second-=60; minute++;
        if(minute==60)
        { minute-=60; hour++;
          if(hour==24) hour-=24;
        }
      }
    }
}
```


public:

Time(int h=0,int m=0,int s=0)

{ second=s%60;

minute=(m+s/60)%60;

hour=(h+(m+s/60)/60)%24;

}

Time& operator++() //前置++

{ AddOne();

return *this;

}

Time operator++(int) //后置++

{ Time t=*this; //记录修改前的对象

AddOne();

return t; //返回修改前的对象

}

```
void show( )  
{ cout<<hour<<':'<<minute<<':'<<second<<endl;}  
};
```

```
void main( )  
{ Time t1(8,56,57),t2(23,59,59),t3;  
  t1.show();  
  ++t1; t1.show();  
  t1++; t1.show();  
  t3=++t1; t1.show(); t3.show();  
  t3=t2++; t2.show(); t3.show();  
}
```

- 上例中的“++”运算符重载为成员函数，若改用友元函数重载，可定义为：

```
Time& operator++(Time& t) //前置++
```

```
{ t.AddOne( );
```

```
    return t;
```

```
}
```

```
Time operator++(Time& t,int) //后  
置++
```

```
{
```

```
    Time tmp(t);
```

```
    t.AddOne( );
```

```
    return tmp;
```

```
}
```

参数类型应是Time类的引用。若用Time类的对象做形参，则在调用函数时是用实参对象的值初始化形参，在函数中对形参的修改并不能影响到实参对象。

6.4.3 下标运算符的重载

- 在C++中访问数组元素时，系统并不对下标做越界检查，但C++允许用户通过重载下标运算符来实现这种检查或完成更广义的操作。

- 重载下标运算符的格式：

`<type> ClassName::operator[](<arg>);`

其中<type>是返回值的类型； <arg>指定下标值。

- 重载下标运算符应注意：

- 下标运算符只能由类的成员函数来实现。
- 左操作数必须是对象。
- 下标运算符重载函数有且仅有一个参数。

- 例6.5 定义一维数组类，并重载下标运算符实现一维数组下标的越界检查。

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

```
#include<stdlib.h>
```

```
class Array{
```

```
    int*p;
```

```
    int size;
```

```
public:
```

```
    Array(int i=10){ p=new int[i]; size=i; }
```

```
    ~Array( ){ delete []p; }
```

```
    int getSize( ) const{ return size; }
```

```
    int& operator[](int index);
```

```
};
```

```
int& Array::operator[](int index)
{ if(index>=0&&index<size) return p[index];
  else{
    cout<<"\n出错: 下标"<<index<<"越界!\n";
    exit(2);
  }
}
```

```
void main( )
{ Array a(10);
  for(int i=0;i<10;i++) a[i]=i;
  for(i=1;i<11;i++)
    cout<<a[i]<<' ';
}
```

- 下标越界检查: 若不越界, 则返回数组中相应元素的引用; 否则输出出错信息, 并终止程序的执行。
- 当要输出a[10]的值时, 因下标越界而终止程序的执行。

程序运行结果:

1 2 3 4 5 6 7 8 9

出错: 下标10出界!

*6.4.4 函数调用运算符的重载

- 函数名后的括号()称为函数调用运算符。
- 函数调用运算符也可重载，但只能重载为非静态的成员函数。
- 重载函数调用运算符的格式：
`<type> ClassName::operator()(<Arg>);`

- 例6.6 重载函数调用运算符，计算下列函数的值：

$$f(x,y)=x^2+3xy+y^2。$$

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

```
class Fun{
```

```
public:
```

```
    double operator( )(double x,double y)
```

```
    { return x*x+3*x*y+y*y; }
```

```
};
```

```
void main( )
```

```
{ Fun f1,f2;
```

```
    cout<<f1(3.5,2.7)<<endl;
```

```
    cout<<f2(1.6,9.4)<<endl;
```

```
}
```

f1.operator()(3.5,2.7)

f2.operator()(1.6,9.4)

程序运行结果：

47.89

136.04

- 将例6.5定义的一维数组类Array的下标运算符重载函数改为函数调用运算符重载函数，实现一维数组下标的越界检查。

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

```
class Array{
    int*p;
    int size;
public:
    Array(int i=10){ p=new int[i]; size=i; }
    ~Array( ){ delete []p; }
    int getSize( ) const{ return size; }
    int& operator( )(int index);
};
```

```
int& Array::operator( )(int index)
{ if(index>=0&&index<size) return p[index];
  else{
    cout<<"\n出错: 下标"<<index<<"越界!\n";
    exit(2);
  }
}
```

```
void main( )
{ Array a(10);
  for(int i=0;i<10;i++) a(i)=i;
  for(i=1;i<11;i++)
    cout<<a(i)<<' ';
}
```

- 下标越界检查: 若不越界, 则返回数组中相应元素的引用; 否则输出出错信息, 并终止程序的执行。
- 当要输出a(10)的值时, 因下标越界而终止程序的执行。

思考题: 定义二维数组类DArray, 并重载函数调用运算符, 实现二维数组下标的越界检查。

6.4.5 转换函数

- 转换函数：类型转换函数的简称。
- 转换函数的作用：将该类的对象转换成type类型的数据。
- 转换函数只能是类中定义的成员函数，格式为：
 ClassName::operator <type>();
其中，ClassName是类名；type是要转换后的一种数据类型；operator与type一起构成转换函数名。该函数不能带有参数，也不能指定返回值类型，它的返回值的类型是type。
- 转换函数的操作数是该类的对象。

- 例6.7 定义一个人民币类，类中包含存储元、角、分的数据成员。请为该类定义类型转换函数，把人民币类的对象转换为等价的实数。

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

```
class RMB{  
    int yuan,jiao,fen;//分别为元、角、分  
public:  
    RMB(int y=0,int j=0,int f=0)  
    { yuan=y;jiao=j;fen=f; }  
    operator float( ) //将元、角、分转换成实数返回  
    { return yuan+jiao/10.0f+fen/100.0f; }  
};
```

```
void main( )  
{ RMB m(25,50,70);
```

```
    cout<< m << ", " //隐式类型转换  
    << float(m) << ', ' //显式类型转换  
    << (float)m << endl; //显式类型转换  
}
```

m.operator float()

程序运行结果:
30.7,30.7,30.7

- 一个类可定义多个转换函数，但使用时必须采用显式类型转换，以明确所调用的转换函数。例如：

```
#include<iostream.h>
class C{
    int x;
    float y;
public:
    C(int a=0,float b=0){ x=a;y=b; }
    operator int( ){ return x; }
    operator float( ){ return y; }
};
void main(void)
{ C c(3,4.5);
  cout<<"x="<<int(c)<<" ,y="<<float(c)<<"\n";
}
```

若类中定义多个转换函数，则使用时只能用显式类型转换。

6.5 运算符重载的规则

- 对可重载的运算符的限制
 - 不可臆造新的运算符。
 - 不允许重载运算符: `..`、`*`、`::`、`?:`、`->`。
- 对运算符函数形式的限制
 - 只能重载为成员函数的运算符: `=`、`()`、`[]`。
 - 只能重载为友元函数的运算符:
提取运算符(`>>`)和插入运算符(`<<`)。
- 不能改变运算符的操作数个数、优先级和结合性。

- 不能改变运算符对预定义类型数据的操作方式。
 - 例如，“+”运算符操作预定义类型的数据时，表示对数据做加法，无法通过重载来改变。
 - 可见，重载运算符时至少应有一个自定义类型的数据做操作数。
- 重载的运算符应尽可能保持其原有的基本语义。
 - 运算符重载的目的，是模仿该运算符操作预定义类型数据时的习惯用法，来操作自定义类型的对象，使算法的表达更自然流畅，使程序更易理解，否则，不如用一个普通函数来实现。
 - 例如，对某自定义类型重载了运算符“+”，则它的含义应是“相加”、“连接”等，而不应是“相减”。

6.6 字符串类

- 在定义的字符串类中
 - 重载 “=”：实现字符串的**赋值**；
 - 重载 “+”：实现两个字符串的**拼接**；
 - 重载 “+=”：实现字符串的**附加**；
 - 重载 “<”、“>”、“==”：实现两个字符串之间的**比较**；
 - 重载 “[]”：访问字符串中的**每个字符**；
 - 重载 “int”：计算字符串的**长度**；
 - 重载**提取运算符** “>>”：实现字符串的**输入**；
 - 重载**插入运算符** “<<”：实现字符串的**输出**。

问题

- `cout << 5 << "this" ;`

为什么能够成立？

- `cout`是什么？

“`<<`”为什么能用在 `cout`上？

流插入运算符的重载

➤ `cout` 是在 `iostream` 中定义的, `ostream` 类的对象。

➤ “`<<`” 能用在 `cout` 上是因为, 在 `iostream` 里对 “`<<`” 进行了重载。

➤ 考虑,怎么重载才能使得

`cout << 5;` 和 `cout << "this"` 都能成立?

流插入运算符的重载

- 有可能按以下方式重载成 ostream 类的成员函数:

```
void ostream::operator<<(int n)
{
..... //输出n的代码
return;
}
```

流插入运算符的重载

`cout << 5 ;` 即 `cout.operator<<(5);`

`cout << "this";` 即 `cout.operator<<("this");`

- 怎么重载才能使得

`cout << 5 << "this" ;`

成立?

- 例6.8 实现字符串直接操作的字符串类。

```
#include<iostream>
#include<string>
#include<stdlib.h>
Using namespace std;
class String{
protected:
    char *ptr;
public:
    String( ){ ptr=0; }
    String(const char*s)
    { ptr=new char[strlen(s)+1];
      strcpy_s(ptr,strlen(s),s);
    }
}
```

```
String(const String&s)
{ if(s.ptr)
    { ptr=new char[strlen(s.ptr)+1];
      strcpy(ptr,s.ptr);
    }else ptr=0;
}
~String( ){ if(ptr) delete []ptr; }
String& operator=(const String&);
String& operator=(const char*p);
String operator+(const String&);
String operator+(const char*p);
friend String operator+(const char*p,
                        const String&s);
```

```
String& operator+=(const String&);  
String& operator+=(const char*p);  
int operator<(const String&s) const  
{ return strcmp(ptr,s.ptr)<0; }  
int operator>(const String&s) const  
{ return strcmp(ptr,s.ptr)>0; }  
int operator==(const String&s) const  
{ return strcmp(ptr,s.ptr)==0; }  
char& operator[](int i);  
operator int( ) const{ return strlen(ptr); }  
friend istream& operator>>(istream&,String&);  
friend ostream& operator<<(ostream&,String&);  
};
```



```
String& String::operator=(const String&s)
{ if(this==&s) return *this;
  if(ptr) delete []ptr;
  if(s.ptr){
    ptr=new char[strlen(s.ptr)+1];
    strcpy(ptr,s.ptr);
  }else ptr=0;
  return *this;
}
```

```
String& String::operator=(const char*p)
{ *this=String(p);
  return *this;
}
```

```
String String::operator+(const String&s)
```

```
{ String t;  
  int len=strlen(ptr)+strlen(s.ptr);  
  if(len>0)  
  { t.ptr=new char[len+1];  
    strcpy(t.ptr,ptr); strcat(t.ptr,s.ptr);  
  }else t.ptr=0;  
  return t;  
}
```

```
String String::operator+(const char*p)
```

```
{ String t(p);  
  t=*this+t;  
  return t;  
}
```

```
String operator+(const char*p,const String&s)
```

```
{ String t(p);
```

```
    t+=s;
```

```
    return t;
```

```
}
```

```
String& String::operator+=(const String& s)
```

```
{ *this=*this+s;
```

```
    return *this;
```

```
}
```

```
String& String::operator+=(const char*p)
```

```
{ String t(p);
```

```
    *this+=t;
```

```
    return *this;
```

```
}
```

```

char& String::operator[](int i)
{ if(i<0||i>=strlen(ptr))
  { cout<<"out of range.\n"; exit(1); }
  return *(ptr+i);
}

istream& operator>>(istream& in,String& s)
{ char temp[4096];
  in>>temp;
  s.ptr=new char[strlen(temp)+1];
  strcpy(s.ptr,temp);
  return in;
}

ostream& operator<<(ostream& out,String& s)
{ out<<s.ptr; return out; }

```

```
void main( )
{ String s1("study "),s2("C++ "),s3,s4;
  s1[0]=s1[0]-'a'+'A';
  s3=s1+s2;
  cout<<s3<<endl;
  s4=s3+"programming";
  cout<<s4<<endl;
  s1+="hard";
  cout<<s1<<",串长 = "<<int(s1)<<endl;
  s2=s1;
  if(s1==s2) cout<<"s1==s2\n";
  s2="abc";
  if(s1<s2) cout<<"s1<s2\n";
}
```

程序运行结果:

Study C++

Study C++ programming

Study hard,串长 = 10

s1==s2

s1<s2

作业 (6)

编写一个点（包含坐标x、y和z）类，重载加法、减法、赋值、++和--共7个运算符，同时重载下标运算符（下标0表示x，下标1表示y，下标2表示z）