

运算符重载 – 基本概念

郭 炜 刘家瑛



北京大學



运算符

■ C++预定义表示对数据的运算

- +, -, *, /, %, ^, &, ~, !, |, =, <<, >>, !=
- 只能用于基本的数据类型
 - 整型, 实型, 字符型, 逻辑型.....



自定义数据类型与运算符重载

- C++提供了数据抽象的手段:

用户自己定义数据类型 -- 类

- 调用类的成员函数 → 操作它的对象

- 类的成员函数 → 操作对象 时, 很不方便

- 在数学上, 两个复数可以直接进行+/-等运算

Vs. 在C++中, 直接将+或-用于复数是不允许的



运算符重载

- 对抽象数据类型也能够直接使用C++提供的运算符
 - 程序更简洁
 - 代码更容易理解
- 例如:
 - `complex_a`和`complex_b`是两个复数对象
 - 求两个复数的和, 希望能直接写:

`complex_a + complex_b`



运算符重载

运算符重载

- 对已有的运算符赋予多重的含义
- 使同一运算符作用于不同类型的数据时 → 不同类型的行为

目的

- 扩展C++中提供的运算符的适用范围, 以用于类所表示的抽象数据类型

同一个运算符, 对不同类型的操作数, 所发生的行为不同

- $(5, 10i) + (4, 8i) = (9, 18i)$
- $5 + 4 = 9$



运算符重载

- 运算符重载的实质是函数重载
返回值类型 operator 运算符（形参表）

```
{  
    .....  
}
```



运算符重载

在程序编译时:

- 把含 运算符的表达式 → 对 运算符函数 的调用
- 把 运算符的操作数 → 运算符函数的 参数
- 运算符被多次重载时, 根据 实参的类型 决定调用哪个运算符函数
- 运算符可以被重载成普通函数
- 也可以被重载成类的成员函数



运算符重载为普通函数

```
class Complex {  
    public:  
        Complex( double r = 0.0, double i= 0.0 ){  
            real = r;  
            imaginary = i;  
        }  
        double real;           // real part  
        double imaginary;      // imaginary part  
};
```



```
Complex operator+ (const Complex & a, const Complex & b)
{
    return Complex( a.real+b.real, a.imaginary+b.imaginary);
} // “类名(参数表)” 就代表一个对象
```

```
Complex a(1,2), b(2,3), c;
```

```
c = a + b; // 相当于什么?
```

- 重载为普通函数时, 参数个数为运算符目数



运算符重载为成员函数

```
class Complex {  
    public:  
        Complex( double r= 0.0, double m = 0.0 ):  
            real(r), imaginary(m) { }    // constructor  
        Complex operator+ ( const Complex & ); // addition  
        Complex operator- ( const Complex & ); // subtraction  
    private:  
        double real;    // real part  
        double imaginary; // imaginary part  
};
```

- 重载为成员函数时, 参数个数为运算符目数减一



// Overloaded addition operator

```
Complex Complex::operator+(const Complex & operand2) {  
    return Complex( real + operand2.real,  
                    imaginary + operand2.imaginary );  
}
```

// Overloaded subtraction operator

```
Complex Complex::operator- (const Complex & operand2){  
    return Complex( real - operand2.real,  
                    imaginary - operand2.imaginary );  
}
```

```
int main(){  
    Complex x, y(4.3, 8.2), z(3.3, 1.1);  
    x = y + z; // 相当于什么?  
    x = y - z; // 相当于什么?  
    return 0;  
}
```

赋值运算符的重载

郭 炜 刘家瑛

北京大学



赋值运算符 ‘=’ 重载

- ▲ 赋值运算符 两边的类型 可以 不匹配
 - 把一个 int类型变量 赋值给一个 Complex对象
 - 把一个 char * 类型的字符串 赋值给一个 字符串对象
- ▲ 需要 重载赋值运算符 ‘=’
- ▲ 赋值运算符 “=” 只能重载为 成员函数



■ 编写一个长度可变的字符串类String

- 包含一个char * 类型的成员变量
→ 指向动态分配的存储空间
- 该存储空间用于存放 '\0' 结尾的字符串

```
class String {  
    private:  
        char * str;  
    public:  
        String () : str(NULL) { }    //构造函数, 初始化str为NULL  
        const char * c_str() { return str; }  
        char * operator = (const char * s);  
        ~String( );  
};
```



//重载 '=' → obj = "hello"能够成立

```
char * String::operator = (const char * s){  
    if(str) delete [] str;  
    if(s) { //s不为NULL才会执行拷贝  
        str = new char[strlen(s)+1];  
        strcpy(str, s);  
    }  
    else  
        str = NULL;  
    return str;  
}
```



```
String::~~String( ) {  
    if(str) delete [] str;  
};  
int main(){  
    String s;  
    s = "Good Luck," ;  
    cout << s.c_str() << endl;  
    // String s2 = "hello!"; //这条语句要是不注释掉就会出错  
    s = "Shenzhen 8!";  
    cout << s.c_str() << endl;  
    return 0;  
}
```

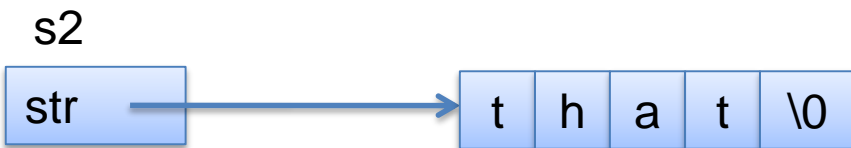
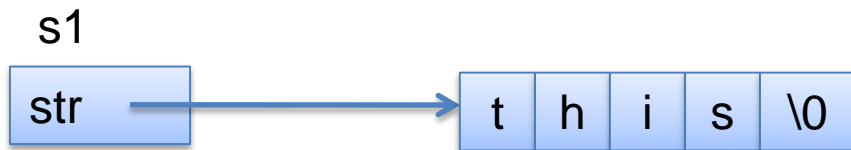
程序输出结果：
Good Luck,
Shenzhen 8!



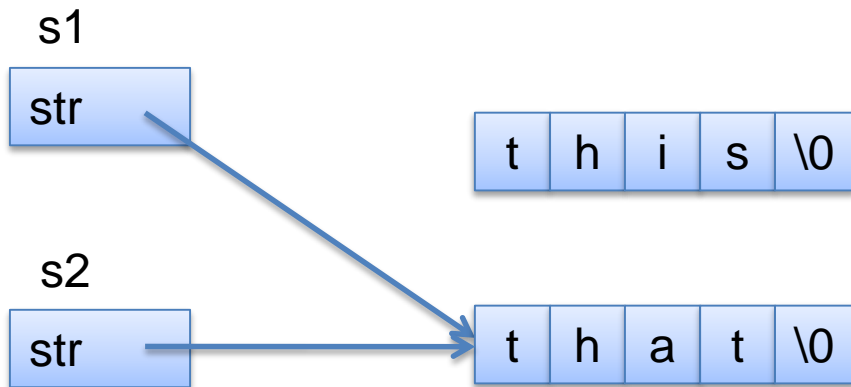
重载赋值运算符的意义 – 浅复制和深复制

- ▀ **S1 = S2;**
- ▀ 浅复制/浅拷贝
 - 执行逐个字节的复制工作

```
MyString S1, S2;  
S1 = "this";  
S2 = "that";  
S1 = S2;
```



String S1, S2;
S1 = "this";
S2 = "that";



S1 = S2;

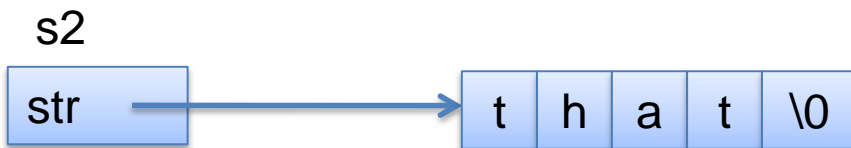
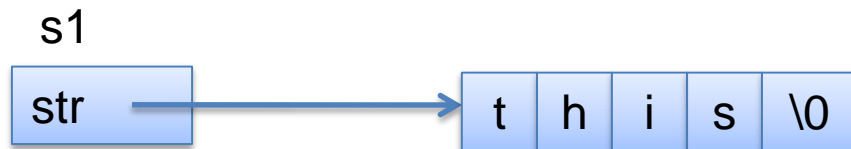


重载赋值运算符的意义 – 浅复制和深复制

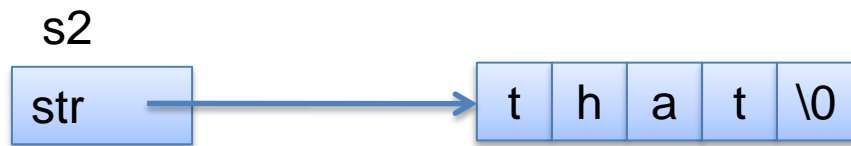
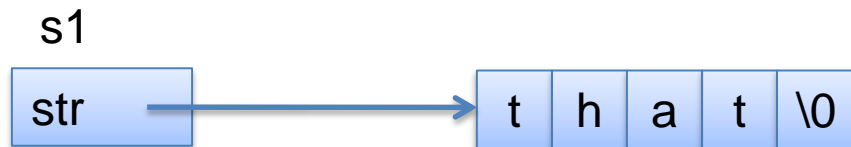
■ 深复制/深拷贝

- 将一个对象中指针变量指向的内容,
→ 复制到另一个对象中指针成员对象指向的地方


```
MyString S1, S2;  
S1 = "this";  
S2 = "that";  
S1 = S2;
```



```
String S1, S2;  
S1 = "this";  
S2 = "that";
```



```
S1 = S2;
```



```
MyString S1, S2;  
S1 = "this";  
S2 = "that";  
S1 = S2;
```

- 在 class MyString 里添加成员函数:
String & operator = (const String & s) {
 if(str) delete [] str;
 str = new char[strlen(s.str)+1];
 strcpy(str, s.str);
 return * this;
}



思考

考虑下面语句:

```
MyString s;  
s = "Hello";  
S = S;
```

是否会有问题?



思考

正确写法:

```
String & String::operator = (const String & s){  
    if(str == s.str) return * this;  
    if(str) delete [] str;  
    if(s.str) {    //s.str不为NULL才会执行拷贝  
        str = new char[strlen(s.str)+1];  
        strcpy( str,s.str);  
    }  
    else  
        str = NULL;  
    return * this;  
}
```



对 operator = 返回值类型的讨论

void 好不好?

考虑: `a = b = c;`

`//等价于a.operator=(b.operator=(c));`

String 好不好?

为什么是 String &

- 运算符重载时, 好的风格 -- 尽量保留运算符原本的特性

考虑: `(a=b)=c;` `//会修改a的值`

- 分别等价于:

`(a.operator=(b)).operator=(c);`



上面的String类是否就没有问题了？

- 为 String类编写 **复制构造函数** 时
- 会面临和 ‘=’ 同样的问题, 用同样的方法处理

```
String::String(String & s)
```

```
{  
    if(s.str) {  
        str = new char[strlen(s.str)+1];  
        strcpy(str, s.str);  
    }  
    else  
        str = NULL;  
}
```

运算符重载为友元函数

郭 炜 刘家瑛

北京大学



运算符重载为友元

- 通常, 将运算符重载为**类的成员函数**
- 重载为友元函数的情况:
 - 成员函数不能满足使用要求
 - 普通函数, 又不能访问类的私有成员



运算符重载为友元

```
class Complex{  
    double real, imag;  
    public:  
        Complex(double r, double i):real(r), imag(i){ };  
        Complex operator+(double r);  
};  
Complex Complex::operator+(double r){ //能解释 c+5  
    return Complex(real + r, imag);  
}
```



- 经过上述重载后:

Complex c ;

c = c + 5; //有定义, 相当于 c = c.operator +(5);

- 但是:

c = 5 + c; //编译出错

- 为了使得上述表达式能成立, 需要将+重载为普通函数

```
Complex operator+ (double r, const Complex & c) {
```

```
//能解释 5+c
```

```
    return Complex( c.real + r, c.imag);
```

```
}
```



普通函数不能访问私有成员

→ 将运算符+重载为友元函数

```
class Complex {  
    double real, imag;  
public:  
    Complex( double r, double i):real(r),imag(i){ };  
    Complex operator+( double r );  
    friend Complex operator + (double r, const Complex & c);  
};
```



程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



运算符重载实例：可变长整型数组

(教材P215)

```
int main() { //要编写可变长整型数组类，使之能如下使用：
```

```
    CArray a; //开始里的数组是空的
```

```
    for( int i = 0; i < 5; ++i)
```

```
        a.push_back(i);
```

要用动态分配的内存来
存放数组元素，需要一
个指针成员变量

```
    CArray a2, a3;
```

```
    a2 = a; //要重载 “=”
```

```
    for( int i = 0; i < a.length(); ++i )
```

```
        cout << a2[i] << " ";
```

要重载 “[]”

```
    a2 = a3; //a2是空的
```

```
    for( int i = 0; i < a2.length(); ++i ) //a2.length()返回0
```

```
        cout << a2[i] << " ";
```

```
    cout << endl;
```

```
    a[3] = 100;
```

```
    CArray a4(a); //要自己写复制构造函数
```

```
    for( int i = 0; i < a4.length(); ++i )
```

```
        cout << a4[i] << " ";
```

```
    return 0;
```

```
}
```

程序输出结果是：

0 1 2 3 4

0 1 2 100 4

要做哪些事情？

```
class CArray {
    int size; //数组元素的个数
    int *ptr; //指向动态分配的数组
public:
    CArray(int s = 0); //s代表数组元素的个数
    CArray(CArray & a);
    ~CArray();
    void push_back(int v); //用于在数组尾部添加一个元素v
    CArray & operator=( const CArray & a);
    //用于数组对象间的赋值
    int length() { return size; } //返回数组元素个数
    _____ CArray::operator[](int i) //返回值是什么类型?
    { //用以支持根据下标访问数组元素,
      // 如n = a[i] 和a[i] = 4; 这样的语句
        return ptr[i];
    }
};
```

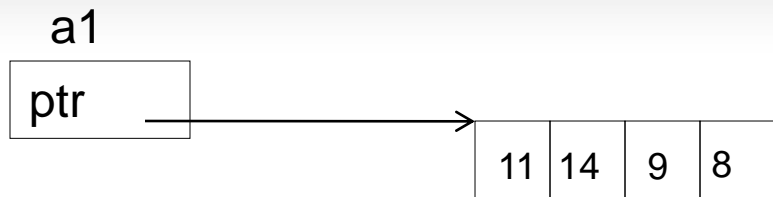


```

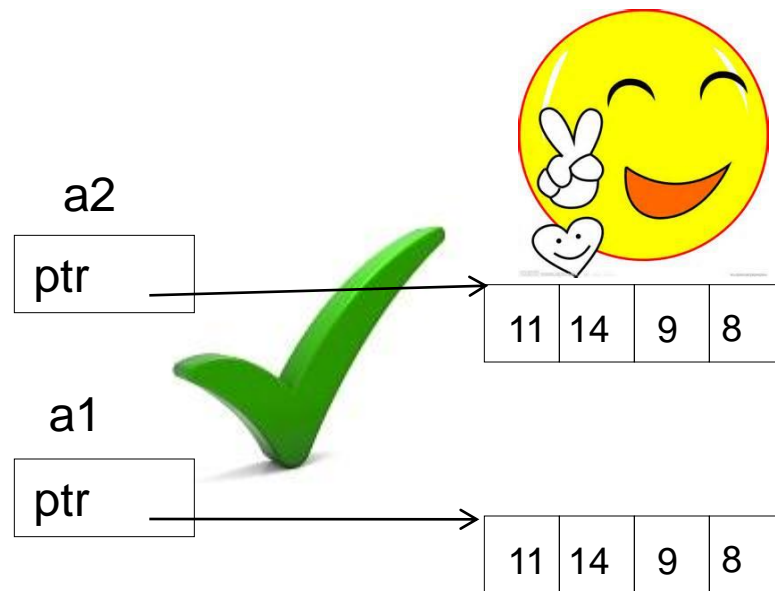
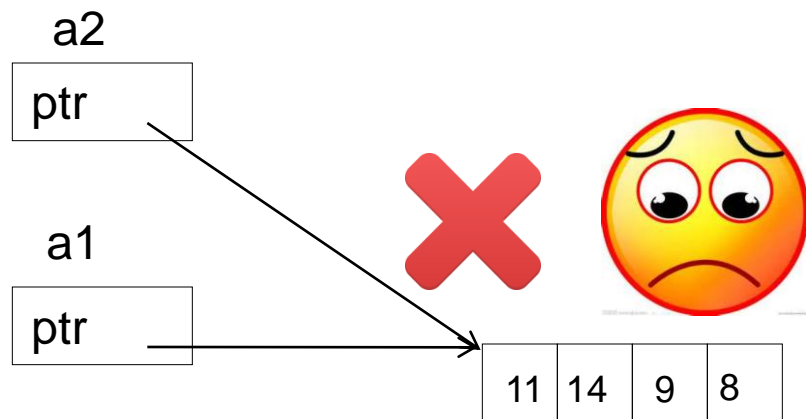
class CArray {
    int size; //数组元素的个数
    int *ptr; //指向动态分配的数组
public:
    CArray(int s = 0);    //s代表数组元素的个数
    CArray(CArray & a);
    ~CArray();
    void push_back(int v); //用于在数组尾部添加一个元素v
    CArray & operator=( const CArray & a);
    //用于数组对象间的赋值
    int length() { return size; } //返回数组元素个数
    int & CArray::operator[](int i) //返回值为 int 不行!不支持 a[i] = 4
    { //用以支持根据下标访问数组元素,
        // 如n = a[i] 和a[i] = 4; 这样的语句
        return ptr[i];
    }
};

```

```
CArray::CArray(int s):size(s)
{
    if( s == 0)
        ptr = NULL;
    else
        ptr = new int[s];
}
CArray::CArray(CArray & a) {
    if( !a.ptr) {
        ptr = NULL;
        size = 0;
        return;
    }
    ptr = new int[a.size];
    memcpy( ptr, a.ptr, sizeof(int ) * a.size);
    size = a.size;
}
```



CArray a2(a1);



```
CArray::~~CArray()
```

```
{
```

```
    if( ptr) delete [] ptr;
```

```
}
```

```
CArray & CArray::operator=( const CArray & a)
```

```
{ //赋值号的作用是使“=”左边对象里存放的数组，大小和内容都和右边的对象一样
```

```
    if( ptr == a.ptr) //防止a=a这样的赋值导致出错
```

```
        return * this;
```

```
    if( a.ptr == NULL) { //如果a里面的数组是空的
```

```
        if( ptr ) delete [] ptr;
```

```
        ptr = NULL;
```

```
        size = 0;
```

```
        return * this;
```

```
}
```

```
if( size < a.size) { //如果原有空间够大，就不用分配新的空间
    if(ptr)
        delete [] ptr;
    ptr = new int[a.size];
}
memcpy( ptr,a.ptr,sizeof(int)*a.size);
size = a.size;
return * this;
} // CArray & CArray::operator=( const CArray & a)
```

```
void CArray::push_back(int v)
{ //在数组尾部添加一个元素
    if( ptr) {
        int * tmpPtr = new int[size+1]; //重新分配空间
        memcpy(tmpPtr,ptr,sizeof(int)*size); //拷贝原数组内容
        delete [] ptr;
        ptr = tmpPtr;
    }
    else //数组本来是空的
        ptr = new int[1];
    ptr[size++] = v; //加入新的数组元素
}
```



北京大学
PEKING UNIVERSITY

信息科学技术学院

程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



北京大学
PEKING UNIVERSITY

信息科学技术学院《程序设计实习》 郭炜 刘家瑛

流插入运算符和流提取运算符的重载

(教材P218)

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？

流插入运算符的重载

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

```
ostream & ostream::operator<<( const char * s )
{
    ..... //输出s的代码
    return * this;
}
```

流插入运算符的重载

cout << 5 << “this”;

本质上的函数调用的形式是什么？

cout.operator<<(5).operator<<(“this”);

流插入运算符的重载

- 假定下面程序输出为 5hello, 该补写些什么

```
class CStudent{  
    public: int nAge;  
};  
  
int main(){  
    CStudent s ;  
    s.nAge = 5;  
    cout << s << "hello";  
    return 0;  
}
```

流插入运算符的重载

```
ostream & operator<<( ostream & o,const CStudent & s){  
    o << s.nAge ;  
    return o;  
}
```

例题(教材P218)

假定c是Complex复数类的对象，现在希望写“`cout << c;`”，就能以“`a+bi`”的形式输出c的值，写“`cin>>c;`”，就能从键盘接受“`a+bi`”形式的输入，并且使得`c.real = a, c.imag = b`。

例题

```
int main() {  
    Complex c;  
    int n;  
    cin >> c >> n;  
    cout << c << ", " << n;  
    return 0;  
}
```

程序运行结果可以如下：

13.2+133i 87 ✓
13.2+133i, 87

```
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

class Complex {
    double real,imag;
public:
    Complex( double r=0, double i=0):real(r),imag(i){ };
    friend ostream & operator<<( ostream & os,
                                const Complex & c);
    friend istream & operator>>( istream & is,Complex & c);
};

ostream & operator<<( ostream & os,const Complex & c)
{
    os << c.real << "+" << c.imag << "i"; //以"a+bi"的形式输出
    return os;
}
```

```
istream & operator>>( istream & is,Complex & c)
{
    string s;
    is >> s; //将"a+bi"作为字符串读入,“a+bi”中间不能有空格
    int pos = s.find("+",0);
    string sTmp = s.substr(0,pos); //分离出代表实部的字符串
    c.real = atof(sTmp.c_str()); //atof库函数能将const char*指针指向的内容转换成
    float
    sTmp = s.substr(pos+1, s.length()-pos-2); //分离出代表虚部的字符串
    c.imag = atof(sTmp.c_str());
    return is;
}
```

```
int main()
{
    Complex c;
    int n;
    cin >> c >> n;
    cout << c << ", " << n;
    return 0;
}
```

程序运行结果可以如下：

$$\frac{13.2+133i \ 87 \checkmark}{13.2+133i, \ 87}$$

自加/自减运算符的重载

郭 炜 刘家瑛

北京大学



自加/自减运算符的重载

- ▲ 自加 ++/自减 -- 运算符有 前置/后置 之分
- ▲ 前置运算符作为一元运算符重载
 - 重载为成员函数:
`T & operator++();`
`T & operator--();`
 - 重载为全局函数 :
`T & operator++(T &);`
`T & operator--(T &);`
- ▲ `++obj`, `obj.operator++()`, `operator++(obj)` 都调用上述函数



自加/自减运算符的重载

■ 后置运算符作为二元运算符重载

- 多写一个参数, 具体无意义
- 重载为成员函数:

`T operator++(int);`

`T operator--(int);`

- 重载为全局函数:

`T operator++(T &, int);`

`T operator--(T &, int);`

■ `obj++`, `obj.operator++(0)`, `operator++(obj,0)` 都调用上函数



```
int main(){
    CDemo d(5);
    cout << (d++) << ","; //等价于 d.operator++(0);
    cout << d << ",";
    cout << (++d) << ","; //等价于 d.operator++();
    cout << d << endl;
    cout << (d--) << ","; //等价于 operator--(d,0);
    cout << d << ",";
    cout << (--d) << ","; //等价于 operator--(d);
    cout << d << endl;
    return 0;
}
```

程序输出结果:

5,6,7,7

7,6,5,5

如何编写 CDemo?



```
class CDemo {  
    private :  
        int n;  
    public:  
        CDemo(int i=0):n(i) { }  
        CDemo & operator++();    //用于前置++形式  
        CDemo operator++(int);  //用于后置++形式  
        operator int ( ) { return n; }  
        friend CDemo & operator--(CDemo &);    //用于前置--形式  
        friend CDemo operator--(CDemo &, int); //用于后置--形式  
};  
CDemo & CDemo::operator++() { //前置 ++  
    n++;  
    return * this;  
}
```



```
CDemo CDemo::operator++(int k) {    //后置 ++
    CDemo tmp(*this);    //记录修改前的对象
    n++;
    return tmp;    //返回修改前的对象
}
CDemo & operator--(CDemo & d) {    //前置--
    d.n--;
    return d;
}
CDemo operator--(CDemo & d, int) {    //后置--
    CDemo tmp(d);
    d.n --;
    return tmp;
}
```



```
operator int ( ) { return n; }
```

- int 作为一个类型强制转换运算符被重载,

```
Demo s;
```

```
(int) s ; //等效于 s.int();
```

- 类型强制转换运算符重载时,
 - 不能写返回值类型
 - 实际上其返回值类型 -- 类型强制转换运算符代表的类型



运算符重载的注意事项

- ✦ C++不允许定义新的运算符
- ✦ 重载后运算符的含义应该符合日常习惯
 - `complex_a + complex_b`
 - `word_a > word_b`
 - `date_b = date_a + n`
- ✦ 运算符重载不改变运算符的优先级
- ✦ 以下运算符不能被重载: “.”, “.*”, “::”, “?:”, `sizeof`
- ✦ 重载运算符(), [], ->或者赋值运算符=时, 重载函数必须声明为类的成员函数