

新标准C++程序设计

北京大学信息学院 郭 炜

GWPL@PKU.EDU.CN



运算符重载



内容提要

- 两种运算符重载的实现方式
- 常见的运算符重载
 - 流运算符：>>、<<
 - 自增运算符++、自减运算符--



自定义数据类型与运算符重载

- C++预定义了一组运算符，用来表示对数据的运算
 - +、-、*、/、%、^、&、~、!、|、=、<<、>>、!=、.....
 - 只能用于基本的数据类型：整型、实型、字符型、逻辑型、.....



- C++提供了数据抽象的手段，允许用户自己定义数据类型：**类**。
 - 通过调用类的成员函数，对它的对象进行操作。
- 但是，在有些时候，用类的成员函数来操作对象时，很不方便。例如：
 - 在数学上，两个复数可以直接进行+、-等运算。
 - 但在C++中，直接将+或-用于复数是不允许的。



运算符重载

- 我们希望：对一些抽象数据类型（即自定义数据类型），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，就是对已有的运算符(C++中预定义的运算符)赋予多重的含义，使同一运算符作用于不同类型的数据时导致不同类型的行为。
- 运算符重载的目的是：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。
- 在程序编译时：
 - 把含运算符的表达式转换成对运算符函数的调用。
 - 把运算符的操作数转换成运算符函数的参数。
 - 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; // 等效于 c = operator+(a,b);
```

- 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。



```
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;
}
```



Complex x, y(4.3, 8.2), z(3.3, 1.1);


x = y + z; // x = y.operator+(z)

x = y - z; // x = y.operator-(z)



赋值运算符 ‘=’ 重载

```
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  
        const Complex &operator=( const Complex & );  
        // assignment  
    private:  
        double real;    // real part  
        double imaginary; // imaginary part  
};
```



// Overloaded = operator

```
const Complex& Complex::operator=( const Complex  
    &right )
```

```
{  real = right.real;  
    imaginary = right.imaginary;  
    return *this;  
}
```

```
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?
```



Complex x, y(4.3, 8.2), z(3.3, 1.1);

x = y + z; // x.operator=(y.operator+(z))

x = y - z; // x.operator= (y.operator-(z))

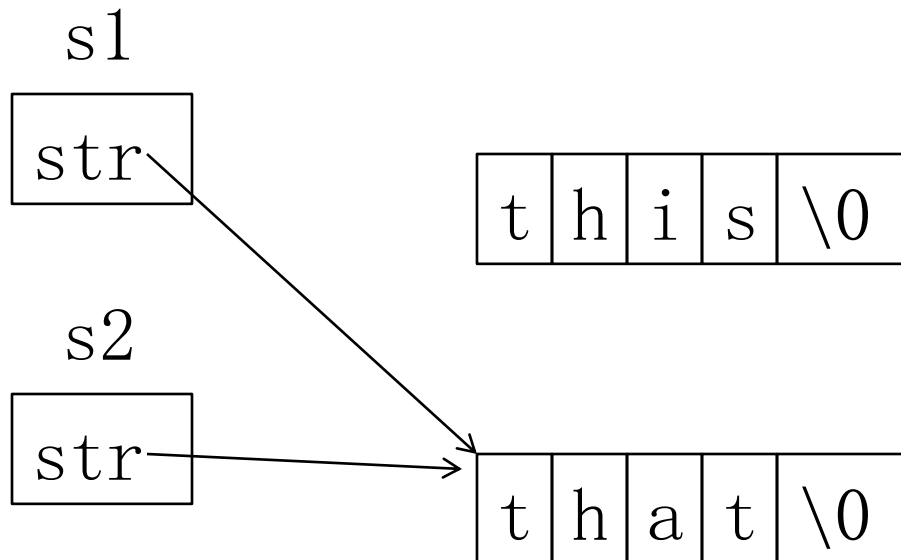
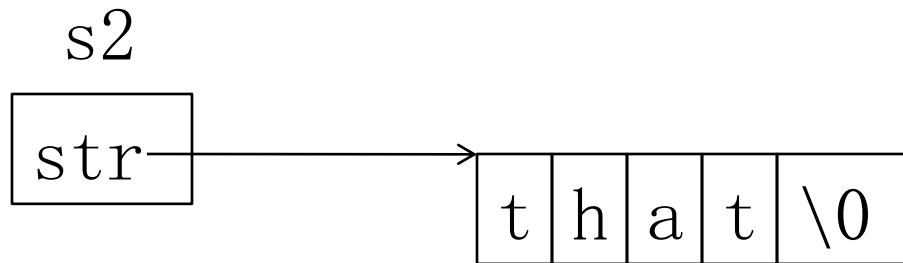
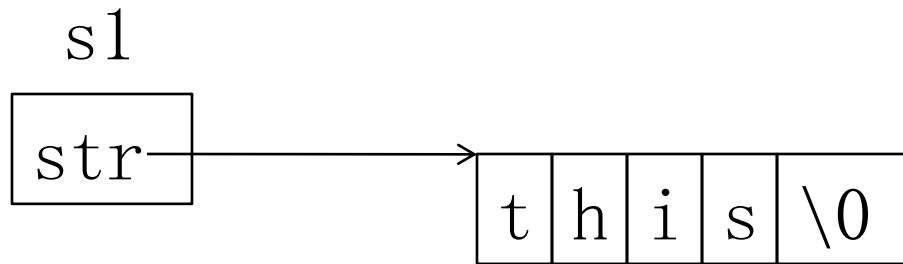


重载赋值运算符的意义

```
class MyString {  
    private: char *str;  
    public:  
        MyString () { str = NULL;}  
        MyString& operator = (const char * s) {  
            if( str) delete [] str;  
            str = new char[strlen( s)+1];  
            strcpy( str, s);  
            return * this;  
        }  
        ~MyString( ) { if( str) delete [] str; };  
};
```

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





MyString S1, S2;

S1 = “this”;

S2 = “that”;

S1 = S2;

- 如不定义自己的赋值运算符，那么S1=S2实际上导致 S1.str 和 S2.str 指向同一地方。
- 如果S1对象消亡，析构函数将释放 S1.str指向的空间，则S2消亡时还要释放一次，不妥。

因此要在 class MyString里添加成员函数：

```
MyString & operator = (const MyString & s) {  
    if( str)  
        delete [] str;  
    str = new char[strlen( s.str)+1];  
    strcpy( str, s.str);  
    return * this;  
}
```

这么做就够了吗？还有什么需要改进的地方？



考虑下面语句：

```
MyString s;
```

```
s = "Hello";
```

```
s = s;
```

是否会有问题？

正确写法：

```
MyString & operator = (const MyString & s) {  
    if( str == s.str)  
        return * this;  
    if( str)  
        delete [] str;  
    str = new char[strlen( s.str)+1];  
    strcpy( str, s.str);  
    return * this;  
}
```



为 `MyString` 类编写复制构造函数的时候，会面临和 `=` 同样的问题，用同样的方法处理。



运算符重载的注意事项

1. C++不允许定义新的运算符；
2. 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`
3. 运算符重载不改变运算符的优先级；
4. 以下运算符不能被重载：“.”、“.*”、“::”、“?:”、`sizeof`；
5. 重载运算符()`[]`、`->`或者赋值运算符`=`时，运算符重载函数必须声明为类的成员函数。



运算符重载为友元

- 一般情况下，将运算符重载为类的成员函数，是较好的选择。
- 但有时，重载为成员函数不能满足使用要求，重载为普通函数，又不能访问类的私有成员，所以需要将运算符重载为友元。

```
class Complex {  
    public:  
        Complex( double r= 0.0, double i=  
            0.0 ):real(r),imaginary(i){ }; //constructor  
        Complex operator+( int r ){  
            return Complex(real + r,imaginary);  
        }  
    private:  
        double real;    // real part  
        double imaginary; // imaginary part  
};
```



- 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

- 所以，为了使得上述的表达式能成立，需要将 + 重载为普通函数。

```
Complex operator + ( int n, const Complex & c) {  
    return Complex( c.real + n, c.imaginary);  
}
```



- 但是普通函数又不能访问私有成员，所以，需要将运算符 + 重载为友元。

```
class Complex {
```

```
public:
```

```
    Complex( double r= 0.0, double i= 0.0 ):
```

```
        real(r),imaginary(i){ }; //constructor
```

```
    Complex operator+( int r ){
```

```
        return Complex(real + r,imaginary);
```

```
    }
```

```
    friend Complex operator+( int r, const Complex & C );
```

```
private:
```

```
    double real;    // real part
```

```
    double imaginary; // imaginary part
```

```
};
```



- 在VC 6.0中，如果使用了
`using namespace std;`

而且又将运算符重载为友元，编译时会出错，这个是vc6的bug,打上补丁后可以解决。 Vs 2008无此问题



流插入运算符的重载

- `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？



- 有可能按以下方式重载：

```
void operator<<(ostream & o, int n) {  
    Output( n);  
}
```

- 假定 **Output** 是一个能将整数 **n** 输出到屏幕上的函数，至于其内部怎么实现，不必深究。



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？



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

- 假定 **Output** 是一个能将整数 n 输出到屏幕上的函数，至于其内部怎么实现，不必深究。

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



```
cout << 5 << “this” ;
```

本质上的函数调用的形式是什么？

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



- 假定下面程序输出为 5hello, 请问该补写些什么?

```
#include <iostream>
using namespace std;
class CStudent{
    public:
        int nAge;
};
int main(){
    CStudent s ;
    s.nAge = 5;
    cout << s <<"hello";
    return 0;
}
```




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



例题13.6.1: 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`。



```
#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：

$$\begin{array}{l} 13.2+133i \quad 87 \checkmark \\ \hline 13.2+133i, \quad 87 \end{array}$$


考虑编写一个整型数组类

```
class Array{
public:
    Array( int n = 10 ):size(n) {
        ptr = new int[n];
    }
    ~Array() {
        delete [] ptr;
    }
private:
    int size; // size of the array
    int *ptr; // pointer to first element of array
};
```



考虑编写一个整型数组类

- 该类的对象就代表一个数组，希望能象普通数组一样使用该类的对象。例如：

```
int main()
{
    Array a(20);
    a[18] = 62;
    int n = a[18];
    cout << a[18] << "," << n;
    return 0;
}
```

输出 62,62

该做些什么？



✓ 当然是重载 [] ！

```
class Array{
```

```
public:
```

```
    Array( int n = 10 ) :size(n) { ptr = new int[n]; }
```

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

```
    int & operator[]( int subscript ){
```

```
        return ptr[ subscript ];
```

```
    }
```

```
private:
```

```
    int size;
```

```
    int *ptr;
```

```
};
```

如果 “int operator[](int)” 是否可以？



当然不行! 因为:

```
a[18] = 62;
```

这样的语句就无法实现我们习惯的功能, 即对数组元素赋值。

✓ 如果我们希望两个Array对象可以互相赋值, 比如:

```
int main() {  
    Array a(20),b(30);  
    a[18] = 62;  
    b[18] = 100;  
    b[25] = 200;  
    a = b;  
    cout << a[18] << "," << a[25];  
    return 0;  
}
```

希望输出 100,200 , 该做些什么?



添加重载等号的成员函数

```
const 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;  
}
```

//返回const array & 类型是为了高效实现

// a = b = c; 形式

✓memcpy是内存拷贝函数，要

#include <memory>

它将从a.ptr起的sizeof(int) * a.size 个字节拷贝到地址 ptr。



Array 类还有没有什么需要补充的地方？

还需要编写复制构造函数

```
Array(const Array & 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;
}
```



重载类型转换运算符

```
#include <iostream>
using namespace std;
class Complex
{
```

```
    double real,imag;
```

```
public:
```

```
    Complex(double r=0,double i=0):real(r),imag(i) { };
```

```
    operator double () { return real; } //重载强制类型转换运算符
```

```
double
```

```
};
```

```
int main()
```

```
{
```

```
    Complex c(1.2,3.4);
```

```
    cout << (double)c << endl; //输出 1.2
```

```
    double n = 2 + c; //等价于 double n=2+c.operator double()
```

```
    cout << n; //输出 3.2
```

```
}
```

输出结果：

1.2

3.2



自增，自减运算符的重载

- 自增运算符++、自减运算符--有前置/后置之分，为了区分所重载的是前置运算符还是后置运算符，C++规定：

- 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：

```
T1 operator++(T2 );
```

```
T1 operator--( T2);
```



自增，自减运算符的重载

- 后置运算符作为二元运算符重载，多写一个没用的参数：

重载为成员函数：

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

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

重载为全局函数：

```
T1 operator++(T2,int );
```

```
T1 operator--( T2,int);
```

但是在没有后置运算符重载而有前置重载的情况下，在vc,vs2008中，obj++ 也调用前置重载，而dev则令obj ++ 编译出错




```
class Sample {  
private :  
    int n;  
public:  
    Sample(int i):n(i) { }  
  
};  
int main()  
{  
    Sample s(5);  
    cout << ( s++ ) << endl;  
    cout << (int) s << endl;  
    cout << (int) (++s) << endl;  
    cout << (int) s << endl;  
    return 0;  
}
```

希望输出:

5

6

7

7

该怎么写Sample?



```
class Sample {  
private :  
    int n;  
public:  
    Sample(int i):n(i) { }  
    Sample operator++();           // 前置  
    Sample operator++( int );      // 后置  
    operator int ( ) { return n; }  
};
```

✓这里，**int** 作为一个类型强制转换运算符被重载，此后

Sample s;

(int) s ; 等效于 s.int();

✓类型强制转换运算符被重载时不能写返回值类型，实际上其返回值类型就是该**类型强制转换运算符**代表的类型



```
Sample Sample::operator++()
```

```
{
```

```
    n ++;
```

```
    return * this;
```

```
}
```

```
Sample Sample::operator++( int k )
```

```
{
```

```
    Sample tmp(* this);
```

```
    n ++;
```

```
    return tmp;
```

```
}
```

++s 即为: s.operator++();

s++即为: s.operator++(0);



上面例子如果去掉后置重载，则在dev 中编译不能通过

`cout << (s++) << endl;` 出错

但在 vc 6和vs2008中，编译没有问题，输出结果是：

6
6
7
7

