

第六章(b) 操作符 重载



本章内容

- 操作符重载的必要性
- 作为成员函数重载
- 作为全局（友元）函数重载
- 一些特殊操作符的重载



操作符重载的必要性

- C++本身没有提供复数类型，可定义一个类来实现：

```
class Complex //复数类定义
```

```
{public:
```

```
    Complex(double r=0.0, double i=0.0)  
    { real=r; imag=i; }
```

```
    void display() const
```

```
    { cout << real << '+' << imag << 'i'; }
```

```
    .....
```

```
private:
```

```
    double real;
```

```
    double imag;
```

```
};
```

- 方案一：定义Complex类成员函数add：

```
class Complex
```

```
{ public:
```

```
    Complex add(const Complex& x) const
```

```
{    Complex temp;
```

```
    temp.real = real + x.real;
```

```
    temp.imag = imag + x.imag;
```

```
    return temp;
```

```
}
```

```
.....
```

```
};
```

```
.....
```

```
Complex a(1.0, 2.0), b(3.0, 4.0), c;
```

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

- 方案二：定义一个全局函数：

```
class Complex //复数类定义
```

```
{ .....
```

```
    friend Complex complex_add(const Complex& x1,  
    const Complex& x2);
```

```
};
```

```
Complex complex_add(const Complex& x1, const  
    Complex& x2)
```

```
{ Complex temp;
```

```
    temp.real = x1.real+x2.real;
```

```
    temp.imag = x1.imag+x2.imag;
```

```
    return temp;
```

```
}
```

```
Complex a(1.0,2.0), b(3.0,4.0), c;
```

```
c = complex_add(a,b);
```

两种实现均不符合数学上的习惯 $c = a + b$;

- C++允许对已有的操作符进行重载，使得它们能对自定义类型（类）的对象进行操作。操作符可以以两种方式重载：
 - 以成员函数形式重载
 - 以全局函数形式重载

操作符重载的基本原则

- 只能重载C++语言中已有的操作符，不可臆造新的操作符。
- 可以重载C++中除下列操作符外的所有操作符：
“.” , “*” , “?:” , “::” , “sizeof”
- 重载不改变原操作符的优先级和结合性。
- 重载不能改变操作数个数。
- 尽量遵循已有操作符的语义（不是必需的）。

作为成员函数重载操作符

◉ 双目操作符重载

● 定义格式

```
class <类名>
```

```
{ .....
```

```
    <返回值类型> operator # (<类型>); // #代表操作符
```

```
};
```

```
    <返回值类型> <类名>::operator # (<类型> <参数>) { .....
```

● 使用格式

```
<类名> a;
```

```
<类名> b;
```

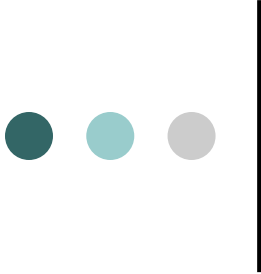
```
a # b 或 a.operator#(b)
```




复数的加法操作

```
class Complex
{   double real, imag;
    public:
        .....
        Complex operator + (const Complex& x) const
        {   Complex temp;
            temp.real = real+x.real;
            temp.imag = imag+x.imag;
            return temp;
        }
};

Complex a(1.0,2.0), b(3.0,4.0), c;
c = a + b;
```



复数的 “等于” 和 “不等于” 操作

```
class Complex
{   double real, imag;
    public:
        .....
        bool operator ==(const Complex& x) const
        {   return (real == x.real) && (imag == x.imag);
        }
        bool operator !=(const Complex& x) const
        {   return !(*this == x);
        }
};

Complex c1,c2;

.....
if (c1 == c2) //或 if (c1 != c2)
```



○ 单目操作符重载

- 定义格式

```
class <类名>
```

```
{ .....
```

```
    <返回值类型> operator # ();
```

```
};
```

```
    <返回值类型> <类名>::operator # () { ..... }
```

- 使用格式

```
<类名> a;
```

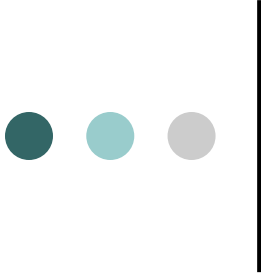
```
#a 或 a.operator#()
```



复数的取负操作

```
class Complex
{
    .....
    public:
        Complex operator -() const
        {
            Complex temp;
            temp.real = -real;
            temp.imag = -imag;
            return temp;
        }
};

.....
Complex a(1,2), b;
b = -a; //把b修改成a的负数。
```



作为全局（友元）函数重载操作符

- 操作符也可以作为全局函数来重载，要求操作符重载函数的参数类型至少有一个为类、结构、枚举或它们的引用类型。
- 如果在操作符重载函数中需要访问参数类的私有成员（提高访问效率），则需要把它说明成相应类的友元。



● 双目操作符重载

- 定义格式:

<返回值类型> operator #(<类型1> <参数1>,
 <类型2> <参数2>)

$$\{ \dots \}$$

- 使用格式：

```
<类型1> a;
```

<类型2> b;

$a \# b$ 或 $\text{operator}\#(a,b)$



○ 单目操作符重载

- 定义格式

`<返回值类型> operator #(<类型> <参数>) { }`

- 使用格式

`<类型> a;`

`#a` 或 `operator#(a)`

- 也可以专门定义一个后置用法的重载函数:

`<返回值类型> operator #(<类型> <参数>, int) { ... }`



复数的加法运算

```
class Complex
```

```
{ .....
```

```
    friend Complex operator + (const Complex& c1, const  
        Complex& c2);  
};
```

```
Complex operator + (const Complex& c1, const  
    Complex& c2)
```

```
{ Complex temp;  
    temp.real = c1.real + c2.real;  
    temp.imag = c1.imag + c2.imag;  
    return temp;  
}
```

```
.....
```

```
Complex a(1.0,2.0),b(3.0,4.0),c;  
c = a + b;
```


[illegible]

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

```
{ return Complex(c1.real+c2.real, c1.imag+c2.imag);  
}
```

Complex operator + (const Complex& c, double d)

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

Complex operator + (double d, const Complex& c)

//只能作为友元函数重载。为什么?

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

.....

Complex a(1,2), b(3,4), c1, c2, c3;

c1 = a + b;

c2 = b + 21.5;

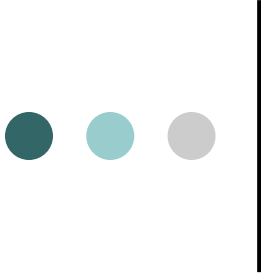
c3 = 10.2 + a;

操作符++和-- 的重载

- 操作符++（--）有前置和后置两种用法，如果没有特殊说明，它们的前置用法与后置用法均使用同一个重载函数：operator ++();
- 为了能够区分++（--）的前置与后置用法，可为它们再写一个重载函数用于实现它们的后置用法，该重载函数应有一个形式上的int型参数：operator ++(int);

```
class Counter
{   int value;
    public:
        Counter() { value = 0; }
        Counter& operator ++() //前置的++重载函数
        { value++;
          return *this;
        }
        const Counter operator ++(int) //后置的++重载函数
        { Counter temp=*this; //保存原来的对象
          ++(*this); //调用前置的++重载函数，或直接写成value++;
          return temp; //返回原来的对象
        }
};

Counter a,b,c;
b = ++a; //使用的是上述类定义中不带参数的操作符++重载函数
c = a++; //使用的是上述类定义中带int型参数的操作符++重载函数
```



赋值操作符 “=” 的重载

- C++编译程序会为每个类定义一个隐式的赋值操作符重载函数，其行为是：逐个成员进行赋值操作（member-wise assignment）。
 - 对于普通成员，它采用常规的赋值操作。
 - 对于成员对象，则调用该成员对象的赋值操作符重载函数进行赋值操作，该定义对成员对象是递归的。

- 对下面类A对象进行赋值操作时，存在以下问题：

```
class A
```

```
{    int x,y;  
    char *p;
```

```
public:
```

```
    A() { x = y = 0; p = NULL; }
```

```
    A(const char *str)
```

```
{    p = new char[strlen(str)+1];  
    strcpy(p,str);  
    x = y = 0; }
```

```
    ~A()
```

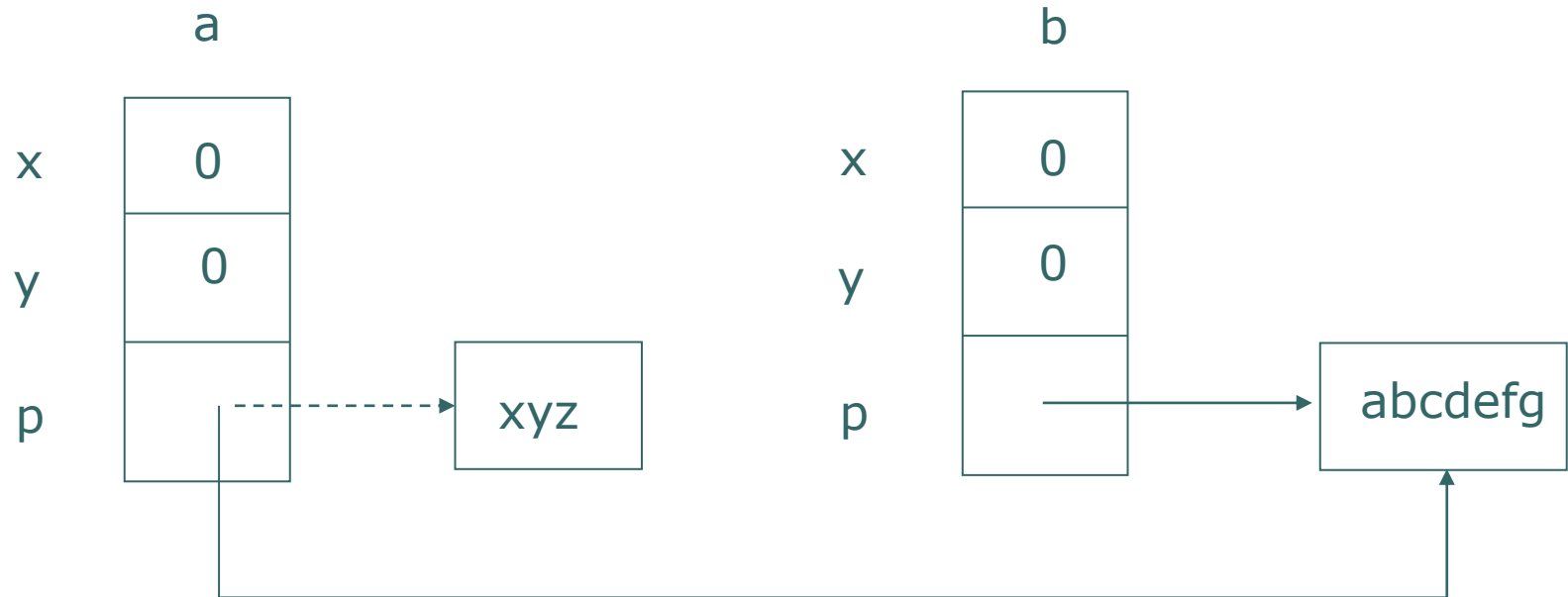
```
{    delete []p;  
    p = NULL; }
```

```
};
```

```
A a("xyz"), b("abcdefg");
```

.....

```
a = b; //赋值后, a.p原来所指向的空间成了“孤儿”。
```



- 
- 解决办法：自己定义赋值操作符重载函数。

```
A& A::operator = (const A& b)
```

```
{ if (&b == this)
```

```
    return *this; //防止自身赋值。
```

```
    delete []p;
```

```
    p = new char[strlen(b.p)+1];
```

```
    strcpy(p,b.p);
```

```
    x = b.x; y = b.y;
```

```
    return *this;
```

```
}
```


- 自定义的赋值操作符重载函数不会自动地去进行成员对象的赋值操作，必须要在自定义的赋值操作符重载函数中显式地指出

```
class A { .....
```

```
class B
```

```
{    A a;
```

```
    int x,y;
```

```
public:
```

```
.....
```

```
B& operator = (const B& b)
```

```
{    a = b.a;//调用A类的赋值操符重载函数来实现成员对象的赋值。
```

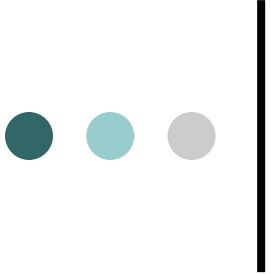
```
    x = b.x;
```

```
    y = b.y;
```

```
    return *this;
```

```
}
```

```
};
```

- 
- 赋值操作符只能作为非静态的成员函数来重载。
 - 一般来讲，需要自定义拷贝构造函数的类通常也需要自定义赋值操作符重载函数。
 - 注意：要区别何时调用拷贝构造函数和赋值操作符重载函数。

A a;

A b = a; //调用拷贝构造函数，它等价于：A b(a);。

.....

b = a; //调用赋值操作符重载函数。

数组元素访问操作符 “[]” 的重载

- 由具有线性关系的元素所构成的对象，可重载 “[]”，实现对其元素的访问。

```
class String
{   char *p;
    public:
        char& operator [](int i) //操作符[]的重载函数
        {   if (i >= strlen(p) || i < 0)
            {   cerr << "下标越界错误\n";
                exit(-1);
            }
            return p[i]; }
};

String s("abcdefg");
...s[i]... //访问s表示的字符串中第i个字符。
```

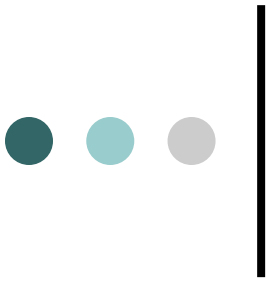
函数调用操作符 ()

- 在C++中，把函数调用也作为一种操作符来看待，并且可以针对类来对其进行重载。函数调用操作符只能作为非静态的成员函数来实现

```
class A
{   int value;
public:
    A() { real = 0;  imag = 0; }

    A(int i)
    { value = i; }

    int operator() (int x) // 函数调用操作符 () 的重载函数
    { return x + value; }
};
```



.....

```
A a(3);
```

```
cout << a(10) << endl; // a(10)将会去调用A中的函数调用  
                        // 操作符函数，10作为实参
```

类成员访问操作符 “->” 的重载

- “->” 为一个双目操作符，其第一个操作数为一个指向类对象或结构变量的指针，第二个操作数为第一个操作数所指向的类对象或结构变量的成员。
- “->” 只能作为非静态成员函数重载，重载时需要按单目操作符重载形式来实现。
- 通过对 “->” 进行重载，可以实现一种智能指针（smart pointers）用该 “指针” 访问另一个对象的成员时，能在访问前做一些额外的事情。

class B //智能指针类

```
{  A * p_a;  
    int count;
```

```
public:
```

```
    B(A * p)
```

```
{    p_a = p;  
    count = 0; }
```

```
A * operator ->() //操作符 “->” 的重载函数
```

```
{    count++;  
    return p_a; }
```

```
int num_of_a_access() const  
{    return count; }
```

```
};
```

```
A a;
```

```
B b(&a); //b为一个智能指针对象，它指向了a。
```

```
b->f(); //等价于： b.operator->()->f(); 即访问的是a.f()
```

```
b->g(); //等价于： b.operator->()->g(); 即访问的是a.g()
```

```
cout << b.num_of_a_access(); // 显示对象a的访问次数
```

class A

```
{    int x,y;
```

```
    public:
```

```
        void f();
```

```
        void g();
```

```
};
```

- 

{

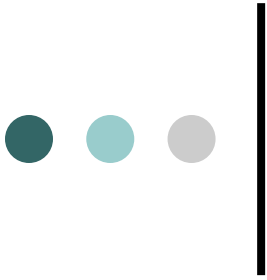
public:

Complex(double r) //一个参数的构造函数可兼作类型转换

}

```
friend Complex operator + (const Complex& x,
                           const Complex& y);
```

}:



**Complex operator + (const Complex& x,
const Complex& y)**

```
{ Complex temp;  
  temp.real = x.real + y.real;  
  temp.imag = x.imag + y.imag;  
  return temp;  
}
```

.....

Complex c1(1,2), c2, c3;

c2 = c1 + 1.7; //1.7隐式转换成一个复数对象Complex(1.7)

c3 = 2.5 + c2; //2.5隐式转换成一个复数对象Complex(2.5)



○ 自定义类型转换函数

```
class A
```

```
{   int x,y;
```

```
    public:
```

```
    .....
```

```
        operator int()
```

```
        { return x + y; } //类型转换操作符int的重载函数
```

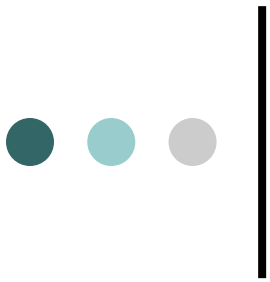
```
};
```

```
.....
```

```
A a;
```

```
int i = 1;
```

```
int z = i + a; //将调用类型转换操作符int的重载函数  
               //把对象a隐式转换成int型数据。
```



```
class A
{   int x,y;
    public:
        A() { x =0;  y = 0; }
        A(int i) { x = i; y = 0; }
        operator int() { return x + y; }
        friend A operator +(const A &a1, const A &a2);
};

.....
A a;
int i = 1, z;
z = a + i; //是a转换成int呢, 还是i转换成A?
```

- 
- 对上面的问题，可以用显式类型转换来解决：

`z = (int)a + i;` 或 `z = a + (A)i;`

- 也可以通过给A类的构造函数A(int i)加上一个修饰符**explicit**来解决

```
class A
```

```
{ .....
```

```
    explicit A(int i) //禁止把它当作隐式类型转换符来用。
```

```
    { x = i; y = 0;
```

```
}
```

```
.....
```

```
};
```