

2018年春

# 程序设计实习 (I): C++程序设计

## 第五讲 运算符重载

刘家瑛

*liujiaying@pku.edu.cn*



# 课前多吼歪

## □ 本周开始上机

- 时间: 周六晚 18点30 – 20点30, 可延伸至21点
- 地点: 院机房1235

## □ 关于时间调整

## □ 第一次上机任务

- 收集大家的联系方式
- 认识各位助教大大
- TA小班课: `const`, 数据对齐, 各种调试技巧



# 课前多吼歪

## □ 助 教

- 看清分组, 各找各TA
- POJ小组也请选择相应TA, 方便统计提交情况

杨 帅



胡越予



魏 晨



张孝帅



夏思烽



厉扬豪



# 上节知识点回顾

- 内联函数 / 内联成员函数(2种方式) **inline** 关键字
- 函数重载 / 缺省值
- 静态成员变量和静态成员函数 **static**
  - 所有对象共享/没有对象生成也能直接访问 / 本质是全局变量
  - 静态成员函数中, 不能访问非静态成员变量 / 不能调用非静态成员函数
- **const**
  - 不能通过const指针修改其指向的地方的内容
  - 不能把常量指针赋值给非常量指针
  - 常量对象 / 常量成员函数重载
- 成员对象和封闭类
  - 封闭类的构造函数时, 添加初始化列表
- 友元: 友元函数和友元类



# 上节知识点回顾

## □析构函数

- 只有一个
- 没有参数和返回值
- 如果不定义, 自动生成, 什么也不做
- 如果定义, 没有缺省的
- 完成对象消亡前的收尾工作
- `~类名(){ ... }`





# 上节课知识点复习

□ 下面说法哪个不正确?

- A) 静态成员函数内部不能访问同类的非静态成员变量, 也不能调用同类的非静态成员函数
- B) 非静态成员函数不能访问静态成员变量
- C) 静态成员变量被所有对象所共享
- D) 在没有任何对象存在的情况下, 也可以访问类的静态成员



# 上节课知识点复习

□ 以下关于友元的说法哪个是不正确的？

A) 一个类的友元函数中可以访问该类对象的私有成员

B) 友元类关系是相互的, 即若类A是类B的友元, 则类B也是类A的友元

C) 在一个类中可以将另一个类的成员函数声明为友元

D) 类之间的友元关系不能传递



# 上节课知识点复习

□ 以下说法正确的是：

A) 成员对象都是用无参构造函数初始化的

B) 封闭类中成员对象的构造函数先于封闭类的构造函数被调用

C) 封闭类中成员对象的析构函数先于封闭类的析构函数被调用

D) 若封闭类有多个成员对象, 则它们的初始化顺序取决于封闭类构造函数中的成员初始化列表





# this 指针

□ C++程序到C程序的翻译:

```
class CCar{  
    public:  
        int price;  
        void SetPrice(int p);  
};  
void CCar::SetPrice(int p){  
    price = p;  
}  
int main(){  
    CCar car;  
    car.SetPrice(20000);  
    return 0;  
}
```

```
struct CCar{  
    int price;  
};  
void SetPrice(CCar * this, int p){  
    this->price = p;  
}  
int main(){  
    struct CCar car;  
    SetPrice(& car, 20000);  
    return 0;  
}
```



# this 指针

□ 非静态成员函数中可以直接使用 this

→ 代表指向该函数作用的对象的指针

```
class Complex {  
    public:  
        double real, imag;  
        Complex(double r, double i):real(r), imag(i) { }  
        Complex AddOne() {  
            this->real ++;  
            return * this;  
        }  
};  
int main() {  
    Complex c1(1, 1), c2(0, 0);  
    c2 = c1.AddOne();  
    cout << c2.real << ", " << c2.imag << endl; //输出 2, 1  
    return 0;  
}
```

# 上节课知识点复习

□ 以下说法不正确的是：

A) 静态成员函数中不能使用this指针

B) this指针就是指向成员函数所作用的对象指针

C) 每个对象的空间中都存放着一个this指针

D) 类的非静态成员函数, 真实的参数比所写的参数多1



# 主要内容

- 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
  - $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $^$ ,  $\&$ ,  $\sim$ ,  $!$ ,  $|$ ,  $=$ ,  $<<$ ,  $>>$ ,  $!=$ , ...
  - 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

□ 有时, 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

□ 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

- 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

## □ 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:

- 把运算符的表达式 → 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





# 运算符重载示例

```
class Complex {  
    public:  
        double real, imag;  
        Complex( double r = 0.0, double i= 0.0 ):real(r), imag(i) { }  
        Complex operator-(const Complex & c);  
};  
  
Complex operator+( const Complex & a, const Complex & b){  
    return Complex( a.real+b.real, a.imag+b.imag);  
    //返回一个临时对象  
}  
  
Complex Complex::operator-(const Complex & c){  
    return Complex(real - c.real, imag - c.imag);  
    //返回一个临时对象  
}
```

重载为成员函数时, 参数个数为运算符目数减 1

重载为普通函数时, 参数个数为运算符目数

```
int main()
{
    Complex a(4, 4), b(1, 1), c;
    c = a + b; //等价于c=operator+(a, b);
    cout << c.real << ", " << c.imag << endl;
    cout << (a-b).real << ", " << (a-b).imag << endl;
    //a-b等价于a.operator-(b)
    return 0;
}
```

程序输出结果:

5, 5

3, 3

$a + b \leftrightarrow \text{operator}+(a, b);$   
 $a - b \leftrightarrow a.\text{operator}-(b);$



# 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, " ; //等价于 s.operator = ("Good Luck, ");
    cout << s.c_str() << endl;
    // String s2 = "hello!"; //不注释掉就会出错, 没定义构造函数
    s = "Shenzhou 8!"; //等价于 s.operator = ("Shenzhou 8!");
    cout << s.c_str() << endl;
    return 0;
}

```

程序输出结果:  
*Good Luck,*  
*Shenzhou 8!*



# 重载赋值运算符的意义

## — 浅复制和深复制

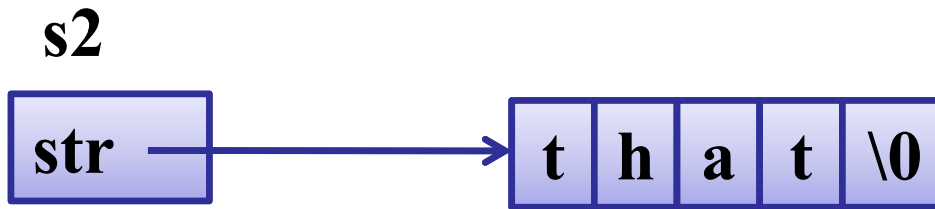
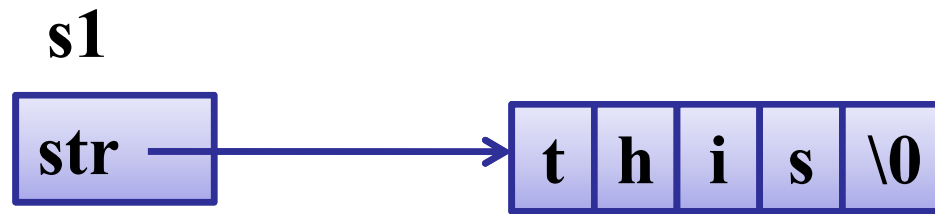
□ **S1 = S2;**

- 浅复制/浅拷贝

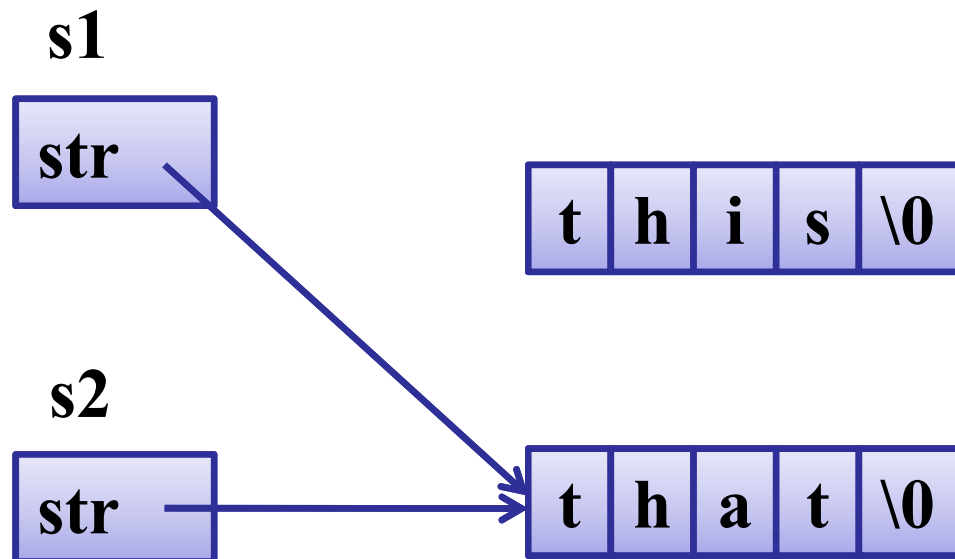
- 执行逐个字节的复制工作

```
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

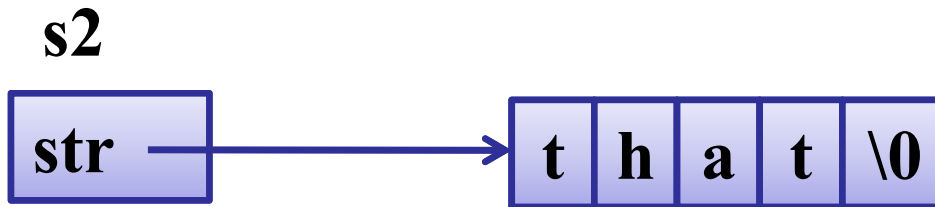
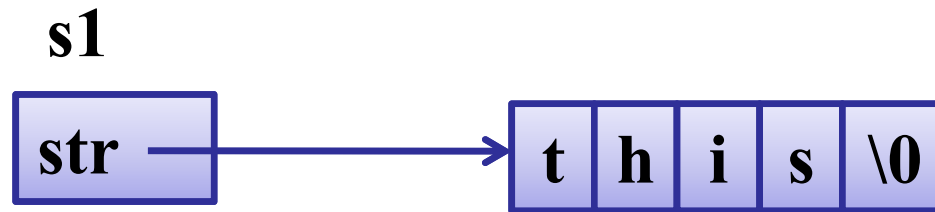
- 深复制/深拷贝

- 将一个对象中指针变量指向的内容,

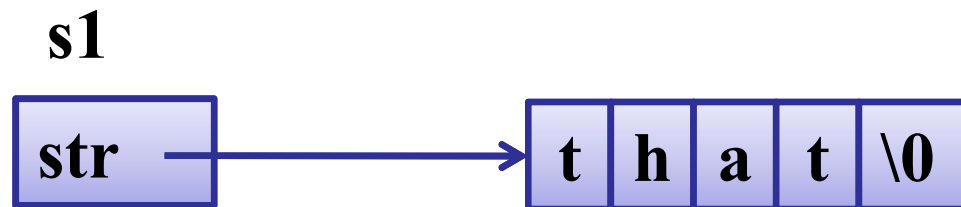
- 复制到另一个对象中指针成员对象指向的地方

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

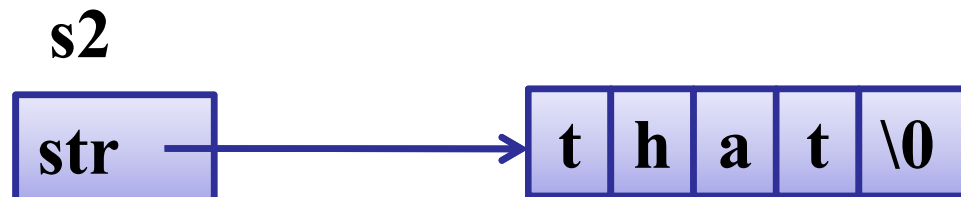




String S1, S2;  
S1 = "this";  
S2 = "that";



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?

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;` 和 `(a=b)=c;` `//(a=b)` 表达式是 `a` 的引用
- 分别等价于:

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

`(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, 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
- 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**

□ 为了使得上述表达式能成立, 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  
};
```



# 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

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

```
void operator<<( ostream & o, const char * s ){  
    Output(s);  
}
```



# 流插入运算符的重载

`cout << 5 ;` 即 `operator<<(cout, 5);`

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

□ 怎么重载才能使得

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

成立?



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

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 & operator<<( ostream & o, const char * s){  
    Output(s);  
    return o;  
} //用引用作为返回值是为了提高效率
```

当然,也可能是ostream类将 << 重载为成员函数





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

本质上的函数调用的形式是什么？

```
operator<<(operator <<(cout, 5) , "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;  
}
```



事实上在 `iostream` 里是将 `<<` 重载成 成员函数

```
class ostream {  
    ostream & operator<< (int n) {  
        Output(n);  
        return * this;  
    }  
};
```

那么,

```
cout << n ++ << n;
```

的函数调用形式是什么呢?



```
cout.operator << (n++).operator << (n);
```

□ 实际上, 上面这条语句可以直接写在程序里, 其效果和

```
cout << n++ << n;
```

完全一样



# 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  
};
```



- 该类的对象就代表一个数组
- 希望能像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! (引用作为函数返回值)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?
```



## 添加重载赋值号的成员函数

```
const Array & operator=( const Array & a)
{
    if( ptr == a.ptr ) return * this;
    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 类还有什么需要补充的地方?

还需要编写复制构造函数

```
Array(Array & a) {  
    ptr = new int[ a.size ];  
    memcpy( ptr, a.ptr, sizeof(int) * a.size);  
    size = a.size;  
}
```

完成形如 **Array b(a);** 方式的初始化

而缺省的复制构造函数不能完成数组元素空间的分配



# 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
  - `class_a &operator++()`
  - `class_a &operator--()`
  - `class_a &operator++(class_a &)`
  - `class_a &operator--(class_a &)`

`++obj`, `obj.operator++()` 或者 `operator++(obj)` 都调用上述函数



# 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:

- `class_a operator++(int)`
- `class_a operator--(int)`
- `class_a operator++(class_a &, int)`
- `class_a operator--(class_a &, int)`

`obj++`, `obj.operator++(0)`或者`operator++(obj, 0)`都调用上函数  
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; }
};

```

□ **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;**

} //前置形式叫做 "增加然后取回值"

//(对象值和取回值均为增加后的值), 允许++++s

**Sample Sample::operator++( int k )**{

**Sample tmp(\* this);**

**n ++;**

**return tmp;**

} //后置形式叫做 "取回然后增加"

(取回值为原值, 对象值为增加后的值, 故取回值需要用复制构造函数方式来实现赋值), 不允许s++++

**++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



# 重载类型转换运算符

**operator type();**

- 必须为成员函数, 不指定返回类型, 形参为空;
- 一般不改变被转换对象, 因此常定义为 **const**
- 类型转换自动调用



# 重载类型转换运算符

```
class Sample {  
private :  
    int n;  
public:  
    Sample(int i){  
        n=i;  
        cout<<"constructor called"<<endl;  
    }  
    Sample operator+(int k){  
        Sample tmp(n + k);  
        return tmp;  
    }  
    operator int ( ) { //重载类型强制转换运算符  
        cout<<"int convertor called"<<endl;  
        return n;  
    }  
};
```



```
int main()  
{  
    Sample s(5);  
    s = s + 4;  
    cout << s << endl;  
    cout << 3 + s << endl;  
    s = 3 + s;  
    return 0;  
}
```



```
int main()
{
    Sample s(5);
    s = s + 4;
    cout << (int)s << endl;
    cout << 3 + (int)s << endl;
    s = 3 + (int)s;
    return 0;
}
```

输出：

constructor called  
constructor called  
int convertor called  
9  
int convertor called  
12  
int convertor called  
constructor called



春天啦~~  
快出去玩耍哟!

