



北京大学
PEKING UNIVERSITY

信息科学技术学院

程序设计实习

C++面向对象程序设计

郭炜 微博 <http://weibo.com/guoweiofpku>
<http://blog.sina.com.cn/u/3266490431>



北京大学
PEKING UNIVERSITY

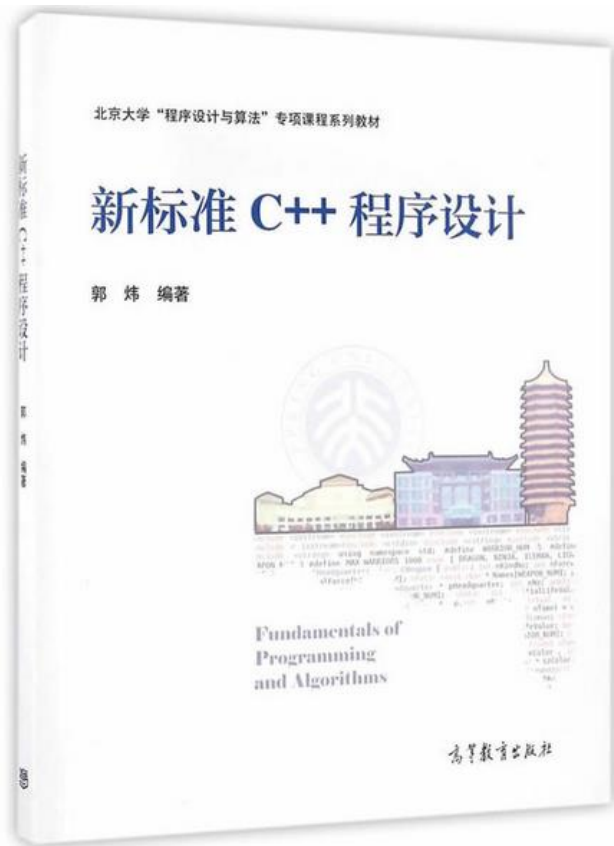
信息科学技术学院

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著





北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

运算符重载 基本概念



威尼斯

运算符重载的需求

- C++预定义的运算符，只能用于基本数据类型的运算：整型、实型、字符型、逻辑型

.....

+, -, *, /, %, ^, &, ~, !, |, =, <<

>>, !=,

运算符重载的需求

- 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-用于复数对象是不允许的。
- 有时希望，让对象也能通过运算符进行运算。这样代码更简洁，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，使之能作用于对象。
- 同一个运算符，对不同类型的操作数，所发生的行为不同。
 - `complex_a + complex_b` 生成新的复数对象
 - $5 + 4 = 9$

运算符重载的形式

- 运算符重载的实质是函数重载

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 把含运算符的表达式转换成对运算符函数的调用。

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 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。

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 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。

运算符重载的形式

返回值类型 operator 运算符 (形参表)

{

.....

}

运算符重载示例 (P209)

```
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一。

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;
}
```

输出:

5,5

3,3

c = a + b; 等价于c=operator+(a,b);
a-b 等价于a.operator-(b)

如果将 `[]` 运算符重载成一个类的成员函数，则该重载函数有几个参数？

☐ A 0

☒ B 1

☐ C 2

☐ D 3

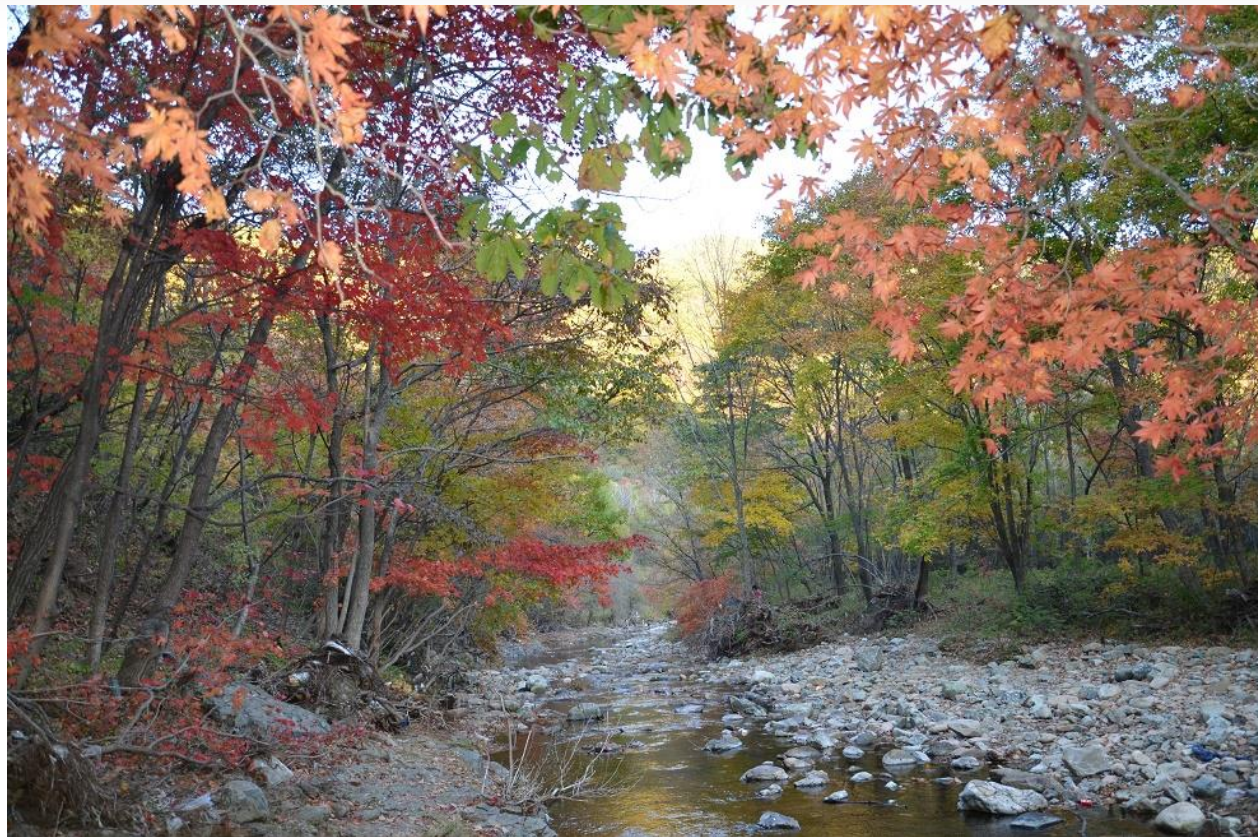
提交



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

赋值运算符 的重载



本溪洋湖沟

赋值运算符 '=' 重载(P210)

有时候希望赋值运算符两边的类型可以不匹配，比如，把一个int类型变量赋值给一个Complex对象，或把一个 char * 类型的字符串赋值给一个字符串对象，此时就需要重载赋值运算符 “=”。

赋值运算符 “=” 只能重载为成员函数

```
class String {
    private:
        char * str;
    public:
        String ():str(new char[1]) { str[0] = 0;}
        const char * c_str() { return str; };
        String & operator = (const char * s);
        String::~~String( ) { delete [] str; }
};

String & String::operator = (const char * s)
{
    //重载"="以使得 obj = "hello"能够成立
    delete [] str;
    str = new char[strlen(s)+1];
    strcpy( str, s);
    return * this;
}
```

```
int main()
{
    String s;
    s = "Good Luck," ; //等价于 s.operator=("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 = "Shenzhou 8!"; //等价于 s.operator=("Shenzhou 8!");
    cout << s.c_str() << endl;
    return 0;
}
```

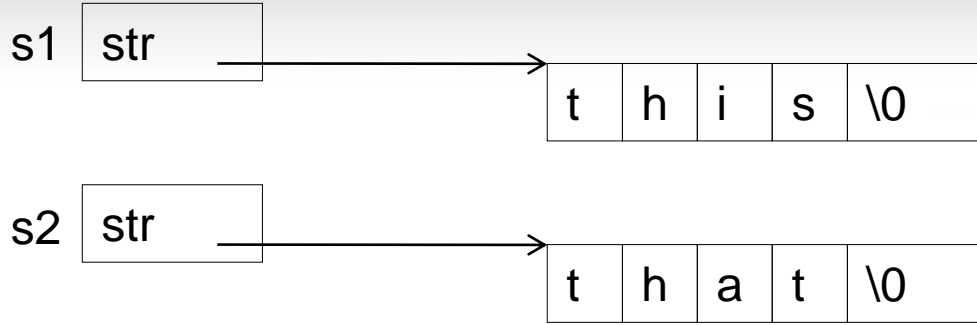
输出:

Good Luck,
Shenzhou 8!

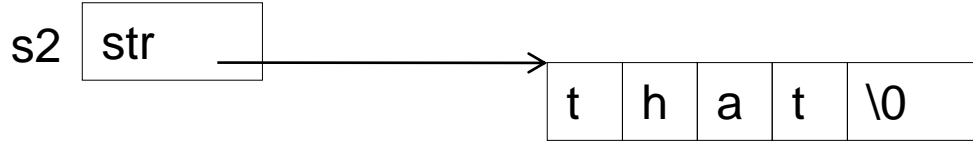
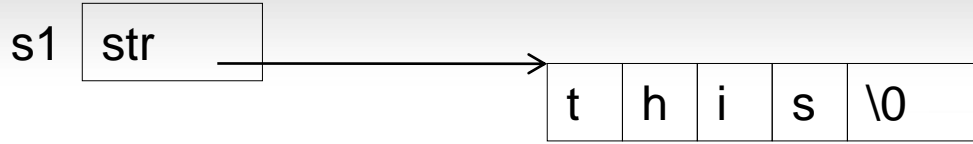
浅拷贝和深拷贝(P213)

```
class String {  
    private:  
        char * str;  
    public:  
        String ():str(new char[1]) { str[0] = 0;}  
        const char * c_str() { return str; };  
        String & operator = (const char * s){  
            delete [] str;  
            str = new char[strlen(s)+1];  
            strcpy( str, s);  
            return * this;  
        };  
        ~String( ) { delete [] str; }  
};
```

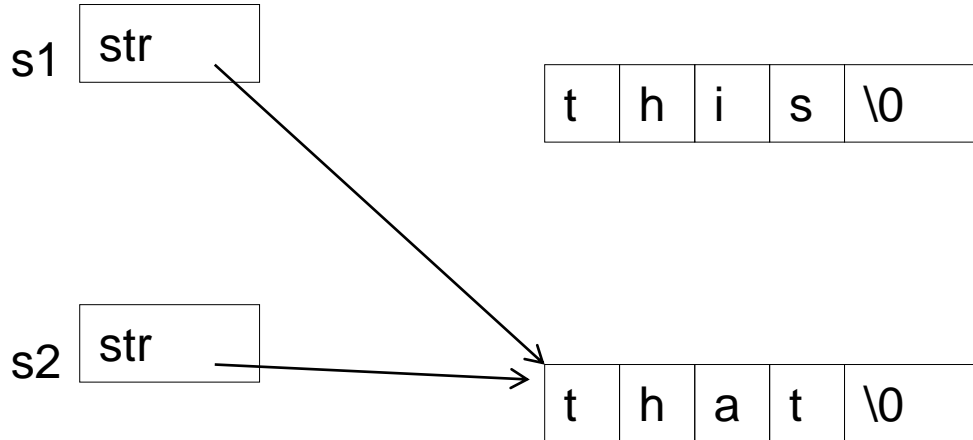
String S1, S2;
S1 = “this”;
S2 = “that”;
S1 = S2;



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



```
String 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。

- 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。
- 另外，如果执行 $S1 = "other";$ 会导致 $S2.str$ 指向的地方被delete

- 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。
- 另外，如果执行 S1 = "other"; 会导致S2.str指向的地方被delete
- 因此要在 class String里添加成员函数:

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

- 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。
- 另外，如果执行 S1 = "other"; 会导致S2.str指向的地方被delete
- 因此要在 class String里添加成员函数:

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

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

是否会有问题？

考虑下面语句：

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

是否会有问题？

解决办法：

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

对 operator = 返回值类型的讨论

void 好不好?

String 好不好?

为什么是 String &

对运算符进行重载的时候，好的风格是应该尽量保留运算符原本的特性

考虑： a = b = c;

和 (a=b)=c; //会修改a的值

对 operator = 返回值类型的讨论

void 好不好?

String 好不好?

为什么是 String &

对运算符进行重载的时候，好的风格是应该尽量保留运算符原本的特性

考虑： a = b = c;

和 (a=b)=c; //会修改a的值

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？



上面的String类是否就没有问题了？

为 String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。

```
String( String & s)
{
    str = new char[strlen(s.str)+1];
    strcpy(str,s.str);
}
```

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析构函数先于封闭类的析构函数被调用
- ☐ C 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
- ☒ D 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

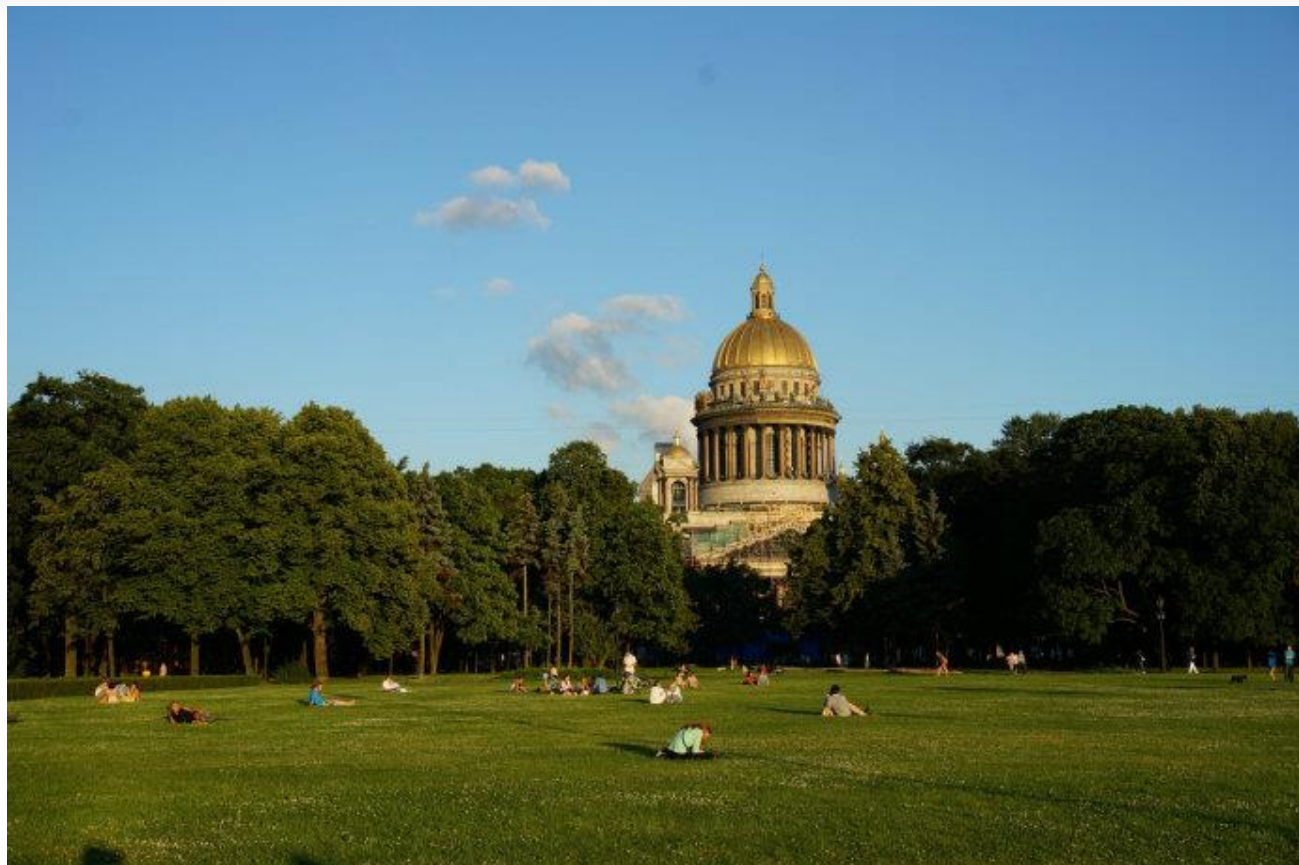
提交



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

运算符重载 为友元函数



俄罗斯圣彼得堡圣以撒教堂

运算符重载为友元函数(P215)

- 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
{
    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);
}
```

运算符重载为友元函数(P215)

- 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+ (double r,const Complex & c)
{
    //能解释 5+c
    return Complex( c.real + r, c.imag);
}
```

运算符重载为友元函数(P215)

- 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
{
    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);
};
```



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

运算符重载实例： 可变长整型数组 (教材P215)



美国拱门国家公园


```
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);
    CArray a2, a3;
    a2 = a;
    for( int i = 0; i < a.length(); ++i )
        cout << a2[i] << " ";
    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);
    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？

```
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;
```

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

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

```
    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);
```

```
    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？

```
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] << " ";
```

```
    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);
```

```
    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？

```
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);
```

```
    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？

```
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];
    }
};
```

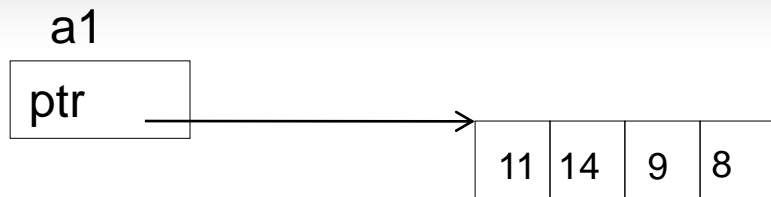


```

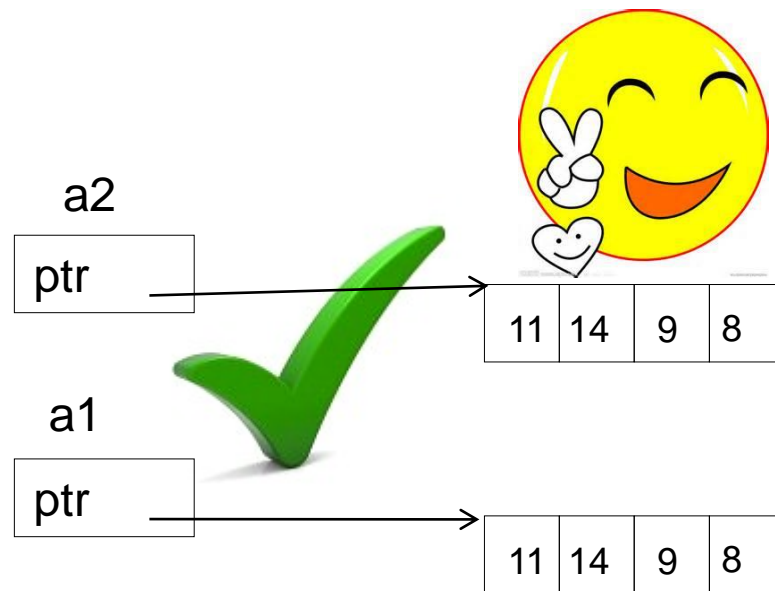
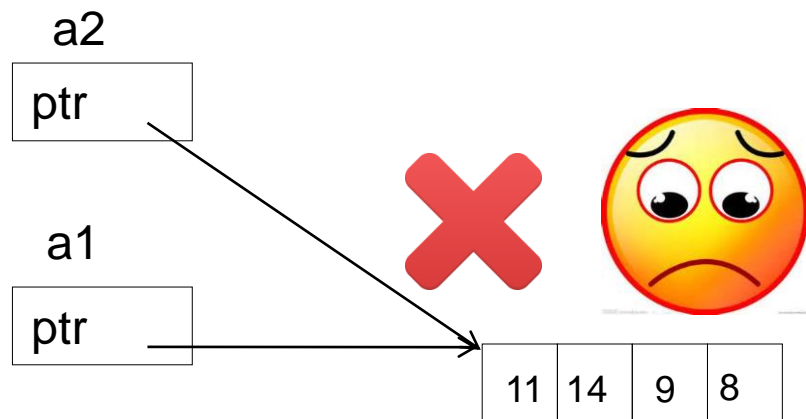
class CArray {
    int size; //数组元素的个数
    int *ptr; //指向动态分配的数组
public:
    CArray(int s = 0);    //s代表数组元素的个数
    CArray(CArray & a);
    ~CArray();
    void push_back(int v); //用于在数组尾部添加一个元素v
    CArray & operator=( const CArray & a);
    //用于数组对象间的赋值
    int length() { return size; } //返回数组元素个数
    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

        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
}
```

内容

在本例的可变长数组类中，重载了哪些运算符或编写了哪些成员函数？

- ☐ A = , [], ++
- ☒ B = , [], 复制构造函数
- ☐ C = , [], ++, 复制构造函数
- ☐ D = , [], &, 复制构造函数

提交



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

流插入运算符和 流提取运算符的重载



荷兰阿姆斯特丹库肯霍夫公园

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;
}
```

运行结果可以如下：

13.2+133i 87 ✓
13.2+133i, 87

重载 "<<" 用于将自定义的对象通过cout输出时，以下说法哪个是正确的？

- ☐ A 可以将"<<"重载为 ostream 类的成员函数，返回值类型是 ostream &
- ☐ B 可以将"<<"重载为全局函数，第一个参数以及返回值，类型都是 ostream
- ☒ C 可以将"<<"重载为全局函数，第一个参数以及返回值，类型都是 ostream &
- ☐ D 可以将"<<"重载为 ostream 类的成员函数，返回值类型是 ostream

提交



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

类型转换运算符和 自增、自减运算符 的重载



旧金山九曲花街

重载类型转换运算符(P220)

```
#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
}
```

自增，自减运算符的重载 (P221)

- 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vs中，obj++ 也调用前置重载，而dev则令 obj ++ 编译出错


```
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;
}
```

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 & );
        friend CDemo operator--(CDemo & ,int);
};

CDemo & CDemo::operator++()
{ //前置 ++
    n ++;
    return * this;
} // ++s即为: s.operator++();
```

```
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
} // s++即为: s.operator++(0);
CDemo & operator--(CDemo & d)
{ //前置--
    d.n--;
    return d;
} //--s即为: operator--(s);
CDemo operator--(CDemo & d,int)
{ //后置--
    CDemo tmp(d);
    d.n --;
    return tmp;
} //s--即为: operator--(s, 0);
```

```
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

```
Demo 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

如何区分自增运算符重载的前置形式和后置形式？

- ☒ A 后置形式比前置形式多一个 int 类型的参数
- ☐ B 重载时，前置形式的函数名是 ++ operator，后置形式的函数名是 operator ++
- ☐ C 无法区分，使用时不管前置形式还是后置形式，都调用相同的重载函数
- ☐ D 前置形式比后置形式多了一个int类型的参数

提交

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。