



華中科技大学

第十一章 运算符重载

许向阳

xuxy@hust.edu.cn





大纲

11.1 运算符概述

11.2 运算符参数

11.3 赋值与调用

11.4 强制类型转换

11.5 重载new和delete

11.6 运算符重载实例



难点

- 运算符重载函数的正确定义
- []、() 运算符重载
- 强制类型转换



11.1 运算符概述

由函数到运算符

设有类 A, A类的对象 a, b, c; 要完成将 b 赋值给 a.

a. assign(b);

Q: assign 的定义形式是什么?

void assign(A &t); ? 如果被赋值的是一个临时对象怎么办?

void assign(const A &t); ? 如果有连续赋值, 怎么办?

a.assign(b.assign(c));

A assign(const A &t); ? 如果有 (a=b) =c; 怎么办?

(a.assign(b)) .assign(c));

←正确的 赋值函数定义形式

A & assign(const A &t);





11.1 运算符概述

由函数到运算符

设有类 A, A类的对象 a, b, c; 要完成将 b 赋值给 a.

定义成员函数: **A & assign(const A &t);**

Q : a=b 更简洁, 能否有 a=b 代替 a. assign(b) ?

定义运算符函数: **A& operator = (const A &t);**

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

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

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



11.1 运算符概述

赋值运算符函数： A& **operator =** (const A &t);

- 赋值表达式（例如 $a = b$ ）的结果是一个左值。
- 它返回的是赋值操作符左侧的对象（即被赋值的对象）的引用。
- 赋值表达式可以出现在赋值操作的左侧（即左值）或右侧（即右值）。



11.1 运算符概述

- 运算符重载是C++中的一种特性；
- 允许为自定义类型（如类或结构体）定义运算符的行为；
- 通过运算符重载，可以使用自然的方式操作自定义类型的对象，就像操作内置类型（如int、float）一样。



11.1 运算符概述

- ◆ 纯单目运算符，只能有一个操作数
!、~、**sizeof**、**new**、**delete**、++、--等
- ◆ 纯双目运算符，只能有两个操作数
[]、->、%、=
- ◆ 三目运算符，有三个操作数，如“?:”
- ◆ 既是单目又是双目的运算符
+、-、&、*
- ◆ 多目运算符，如函数参数表 “()”。



11.1 运算符概述

- ◆ **左值运算符：** 运算结果为左值
其表达式可出现在等号左边
如前置 $++/-$ 、赋值运算 $=$ 、 $+ =$ 、 $* =$ 和 $\& =$ 等。
- ◆ **右值运算符：** 运算结果为右值
如 $+$ 、 $-$ 、 $\%$ 、后置 $++/-$ 等。
- ◆ 某些运算符要求第一个操作数为左值
如 前置 $++ / -$ 、 $=$ 、 $+ =$ 、 $\& =$ 等。



11.1 运算符概述

- ◆ C++预定义了简单类型的运算符重载
如 $3+5$ 、 $3.2+5.3$ 分别表示整数和浮点加法。
- ◆ 运算符重载必须针对类的对象
重载时至少有一个参数代表对象(类型如A、const A&)。
- ◆ C++用“operator 运算符”进行运算符重载。
- ◆ 对于运算符实例函数成员，隐含参数this代表第一个操作数对象。



11.1 运算符概述

- 有些运算符不能重载

sizeof . .* :: ? :

- 不能重载为静态函数成员

+ - * / += *= 等

可以重载为普通函数、实例函数成员

- 不能重载为普通函数、静态函数成员

= -> () [] 可重载为实例函数成员

- 不能重载为类中的实例成员函数

new delete 可重载为普通函数、静态函数成员



11.1 运算符概述

普通函数	实例函数成员	静态函数成员
+、 -、 *、 / 等 +=、 *=、 /=等 new、 delete	+、 -、 *、 / 等 =、 -> () [] +=、 *=、 /=等	new delete

算术运算: +、 -、 *、 /、 %

关系运算: ==、 !=、 <、 >、 <=、 >=

逻辑运算: ||、 &&、 !

单目运算: +/正、 -/负、 */指针、 &/取地址

自增自减: ++、 --

位运 算: |、 &、 ~、 ^、 <<、 >>

赋值运算: =、 +=、 -=、

空间申请与释放: new, delete、 new[], delete []

其他运算: ()(函数调用)、 [](下标)、 ->、 , (逗号)





重载 赋值运算=

赋值运算=、 +=、 -=、 *= 等等，是左值运算符

```
class MAT {  
    int* e;           //指向所有整型矩阵元素的指针  
    int r, c;  
public:  
    MAT(int r, int c); //矩阵定义  
    MAT(const MAT& a); //深拷贝构造  
    MAT(MAT&& a);  
    MAT& operator=(const MAT& a); //深拷贝赋值运算  
    MAT& operator=(MAT&& a); //移动赋值运算  
    MAT& operator+=(const MAT& a); //“+”运算  
    MAT& operator-=(const MAT& a); //“-”运算  
    MAT& operator*=(const MAT& a);  
};
```



重载 赋值运算=

```
MAT::MAT(const MAT& a) // 深拷贝构造
{
    cout << "construct using object" << endl;
    r = a.r;
    c = a.c;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
}
```

```
MAT::~MAT()
{
    cout << "deconstruct object " << endl;
    if (e) delete e;
    e = NULL;
}
```



华中科技大学

重载 赋值运算=

赋值函数的正确写法:

ClassName & operator =(const ClassName &obj);

有问题的一些写法:

void operator =(ClassName obj);

void operator =(ClassName &obj);

void operator =(const ClassName &obj);

ClassName operator =(const ClassName &obj);

Q: 各自的执行过程是什么?

错误的写法:

ClassName & operator =(const ClassName &obj) const;





重载 赋值运算=

ClassName & operator =(const ClassName &obj);

```
MAT& MAT::operator=(const MAT& a) //深拷贝赋值运算
{
    cout << "assign operation ..." << endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
    return *this;
}
```



重载 赋值运算=

```
void MAT ::operator=(MAT a) //效率低
{
    cout<<"assign operation ..."<<endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
}
```

```
MAT a1(10, 20);
MAT a2(3, 4);
cout << " = begin " << endl;
a1 = a2;
cout << " = over " << endl;
```

Q : void operator =(ClassName obj); 的执行流程?

= begin

construct using object

assign operation ...

deconstruct object

= over



重载 赋值运算=

```
void MAT ::operator=(MAT &a)
{
    cout<<"assign operation ..."<<endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
}
```

```
MAT a1(10, 20);
MAT a2(3, 4);
cout << " = begin " << endl;
a1 = a2;
cout << " = over " << endl;
```

若赋值函数写成: void operator=(ClassName &obj);

= begin

assign operation ...

= over



重载 赋值运算=

```
void MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ... "<<endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
}
```

```
MAT a1(10, 20);
MAT a2(3, 4);
cout << " = begin " << endl;
a1 = a2;
cout << " = over " << endl;
```

若赋值函数写成: **void operator=(const ClassName &obj);**

= begin
assign operation ...
= over

```
MAT a1(10, 20);
const MAT a2(3, 4);

a1 = a2;      a1=MAT(5, 6);
只有=运算参数上带了 const,
才支持 该语句
```





重载 赋值运算=

```
void MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ..."<<endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
}
```

```
MAT a1(10, 20);
MAT a2(3, 4);
MAT a3(6, 5);
cout << " = begin " << endl;
a1 = a2 = a3;
cout << " = over " << endl;
```

若赋值函数写成: **void operator=(const ClassName &obj);**

不支持 `a1 = a2 = a3;`
`a1.operator=(a2.operator=(a3));`
`a1.operator=(void);`



重载 赋值运算=

```
MAT MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ..."<<endl;
.....
    return *this;
}
```

```
MAT a1(10, 20);
MAT a2(3, 4);
MAT a3(6, 5);
cout << " = begin " << endl;
a1 = a2;
cout << " = over " << endl;
```

若赋值函数写成： **MAT operator =(const ClassName &obj);**

= begin
assign operation ...
construct using object
deconstruct object
= over

返回对象的构造与析构

类比：

a1. operator =(a2);

fassign(a1, a2);

虽然没有接收 fassign 的返回对象，但有临时对象的构造与析构；

类比： **MAT fassign(MAT *const this, const MAT &obj);**





重载 赋值运算=

```
MAT MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ..."<<endl;
    .....
    return *this;
}
```

```
= begin
assign operation ...
construct using object
assign operation ...
construct using object
deconstruct object
deconstruct object
= over
deconstruct object
deconstruct object
deconstruct object
```

```
MAT a1(10, 20);
MAT a2(3, 4);
MAT a3(6, 5);
cout <<" = begin " << endl;
(a1 = a2) = a3;
cout <<" = over " << endl;
```

名称	值
a1	{e=0x014b5a38 {-842150451} r=3 c=4 }
e	0x014b5a38 {-842150451}
r	3
c	4
a2	{e=0x014b6d88 {-842150451} r=3 c=4 }
e	0x014b6d88 {-842150451}
r	3
c	4
a3	{e=0x014beb60 {-842150451} r=6 c=5 }
e	0x014beb60 {-842150451}
r	6
c	5

a1 与 a2 相同； a3 赋给临时对象
有两次临时对象的构造和析构





重载 赋值运算=

```
MAT MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ... "<<endl;
    .....
    return *this;
}
```

```
MAT a1(10, 20);
MAT a2(3, 4);
MAT a3(6, 5);
cout <<" = begin " << endl;
a1 = ( a2 = a3 );
cout <<" = over " << endl;
```

执行： a2 = a3；
并产生 临时对象；
临时对象时 a2 的复制品

执行 : a1 = 临时对象
也会产生另一个临时对象，
是 a1 的复制品

有两次临时对象的构造和析构



重载 赋值运算=

```
MAT & MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ... "<<endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
    return *this;
}
```

(a1=a2) = a3; 执行后 a1 与 a3 相同

但对于： MAT MAT ::operator=(const MAT &a) ;
执行后， a1 与 a2 相同



重载 赋值运算=

Q: 假设有 `a=a` ; 运行结果如何?

```
MAT & MAT ::operator=(const MAT &a)
{
    cout<<"assign operation ... "<<endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
    return *this;
}
```

可以运行，也不会出现异常提示。

但结果不正确！

`This.e == a.e`

加语句: `if (this == &a) return *this;`





移动 赋值运算=

设有矩阵 $a, b, c; c = a+b;$ $a+b$ 的返回是一个临时对象

```
MAT & MAT ::operator=(const MAT &a)
{
    cout<< "assign operation ..." << endl;
    if (this == &a) return *this;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = new int[r * c];
    memcpy(e, a.e, r * c * sizeof(int));
    return *this;
}
```

Q: 赋值语句执行完后，临时对象要析构，
上述方法有何不足？（不是错误，而是不足）
如何改进？





移动 赋值运算 =

```
MAT& MAT::operator=(MAT && a) //移动赋值运算
{
    if (this == &a) return *this;
    cout << "moving assign operation ..." << endl;
    r = a.r;
    c = a.c;
    if (e) delete e;
    e = a.e;
    a.e = nullptr;
    return *this;
}
```

复制构造 VS 移动构造
深拷贝赋值 VS 移动赋值





重载 赋值运算=

Q: 在函数返回引用时，增加const修饰，结果如何？

const CName & operator =(const CName &obj);

a = b ; 正常

a = b =c; 正常 a = const CName Object;

(a=b) =c; 编译有错误。

const Cname Object = c;



重载 赋值运算 =

编译器提供默认的赋值运算

以浅拷贝的形式实现。

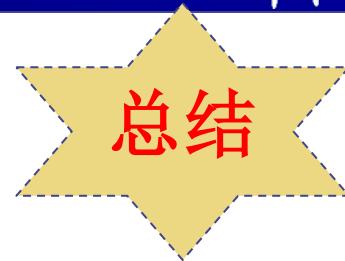
在对象中无指针类型的成员，即无对象体外空间，则无需自己编写 赋值 (=) 的重载函数。

赋值函数的恰当写法：

ClassName & operator =(const ClassName &obj);

为提高效率，提供移动赋值函数

ClassName & operator =(ClassName &&obj);





重载 +=、 -=、 *=

运算符为左值运算符

ClassName & operator +=(const ClassName &obj);

返回非只读
引用类型

不能加 const

当运算符第一参数为左值时，
隐含this指向的对象可以修改

(1) 加const，函数体中不修改第二参数；

(2) 可适配三种类型的实参

普通对象、常量对象、临时对象



重载 加法运算 +

Q: 加法运算符 + 函数的正确写法是什么？

例如： MAT m1,m2,m3,m4; m4=m1+m2+m3;

ClassName operator +(const ClassName &obj);

ClassName operator +(const ClassName &obj) const;

const ClassName operator +(const ClassName &obj) const;

~~const ClassName operator +(const ClassName &obj);~~

Q: 不同位置的 const 分别是什么意思？



重载 加法运算 +

ClassName operator +(const ClassName &obj);

➤ 参数有 & 及 const 的原因

引用参数 比 对象参数有更高的效率；

+号 之右：有名对象、常量对象、临时对象，const 通配

➤ 返回是对象，而不是引用

相加的结果放在局部对象中，不应返回局部对象引用。

若在函数中申请存放结果的“永久”空间，不合逻辑。

➤ 返回的类型前不加 const （除非在最后也加 const）

返回结果是一个临时对象 temp，若之后有

temp + m3； 隐含参数为 const ClassName * const this;

与 + 运算的隐含参数 ClassName * const this 不匹配





重载 加法运算 +

```
MAT operator+(const MAT & a, const MAT &b)
{   MAT      temp(a);
    .....
    return temp;
}
```

```
MAT MAT::operator+(const MAT & a)
{   MAT      temp(a);
    .....
    return temp;
}
```

```
MAT a(3, 4), b(3, 4), c(3, 4);
c=a+b; // 优先使用实例函数成员
c = a. operator +(b); // 使用实例函数成员
c = operator +(a,b); // 使用普通重载函数
```

重载 + 为普通函数

VS

重载+ 为函数成员

注意成员访问权限



重载 运算 []

数组元素的访问；对象中某个元素

```
class STRING {  
private:    char *s;  
public:  
    STRING(const char *s) {  
        s=new char[strlen(str)+1];  
        strcpy(s,str);  
    }  
    char operator[](int i) {  
        return *(s+i); }  
};
```

Q: 能不能 `s1[5] = 'A';` ?

编译报错：=的左操作数必须为 左值。

[] 返回的是一个 `char`，在一个临时空间中，
不能修改该临时空间中的内容。即 非左值！

```
STRING s1("S1 hello");  
char t;  
t=s1[5];  
cout<<s1[0]<<s1[1]<<endl;
```

`s1[5]` 相等于
`s1.operator [] (5)`

Q: `s1[5]` 与 `s[5]` 会不会混淆？

不会！类型不同





重载 运算 []

Q: 如何修改函数，使 s1[5] 可以出现在 = 的左边？

```
class STRING {  
private:    char *s;  
public:  
char operator [](int i) {  
    return *(s+i); }  
};
```

```
STRING s1("S1 hello");  
s1[5]='A';  
char & operator [](int i) {  
    return *(s+i); }  
}
```

char & t = s[i];

引用对象 t，不可改 \dashrightarrow 被引用对象 s[i]，可以修改

t: 在临时空间中，该临时空间中的内容不可改！

对于一个引用变量而言，它本身就是无法修改的！

$t = 'A'$; 不是修改 t, 而是修改被引用的对象





理解函数的返回：传统左值 和右值

对于一个函数 $T f(\dots)$ ；可以看成是变量 $T t$ ，
若 T 是简单类型，如 `char`, `int` 等等，返回结果先存放在
寄存器中，不允许修改寄存器中的值！等同 $T \text{ const } t$.

`char f(\dots); char const t; t = 'A'` 错误

`char const` 等同 `const char` ; $f(\dots) = 'A'$ 错误

`char *f(\dots); char * const t;`

$*t = 'A'$ 正确，可以有 $*f(\dots) = 'A'$;

设 p 是一个 `char` 指针， $t = p$; 错误，即不能有 $f(\dots) = p$;

`char &f(\dots); char & (const t);`

类比指针 `char * (const q);`

$t = 'A'$ ，不是改 t ，而是改被引用的对象！等同 $*q = 'A'$ 。

$f(\dots) = 'A'$; 正确





理解函数的返回：传统左值 和右值

对于一个函数 $T f(\dots)$ ；可以看成是变量 $T t$ ，
若 T 是自己定义的类，返回结果是在一片临时空间中，而不是在一个寄存器中。这片临时空间中的内容可改！

```
MAT MAT::operator+(const MAT & a);  
MAT & MAT ::operator=(const MAT &a);
```

```
MAT a1(..), a2(..), a3(..);  
(a1+a2) = a3; 是可以的!
```

```
MAT MAT ::operator=(const MAT &a);  
(a1 = a2) = a3; 是可以的!
```

注：不同的编译器做法可能不同！本PPT 是用 VS 2019.





华中科技大学

理解函数的返回：传统左值 和 右值

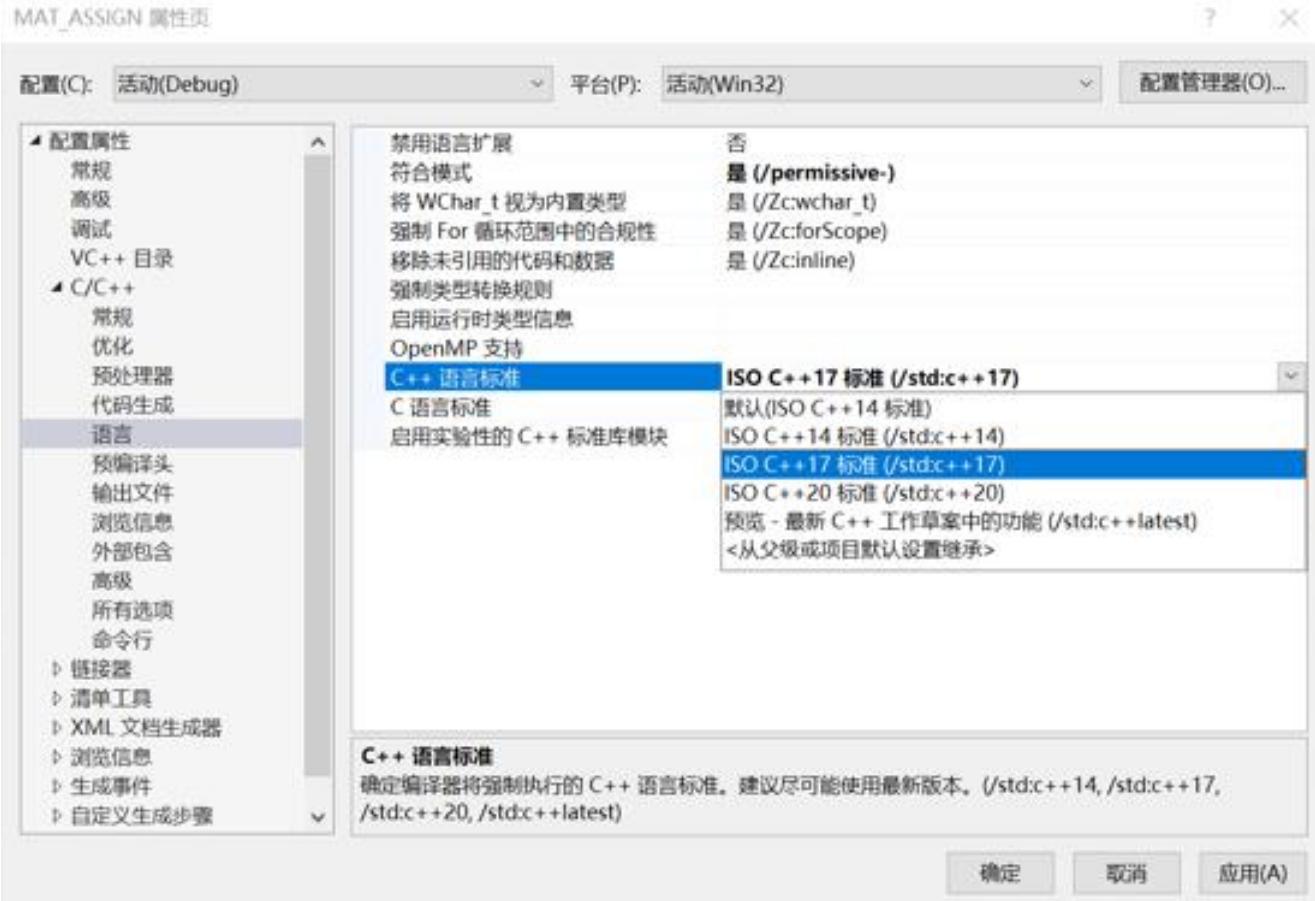
MAT_ASSIGN 属性页

配置(C): 活动(Debug) 平台(P): 活动(Win32) 配置管理器(O)...

禁用语言扩展 否
符合模式 是 (/permissive-)
将 WChar_t 视为内置类型 是 (/Zcwchar_t)
强制 For 循环范围中的合规性 是 (/Zc:forScope)
移除未引用的代码和数据 是 (/Zc:inline)
强制类型转换规则
启用运行时类型信息
OpenMP 支持

C++ 语言标准 ISO C++17 标准 (/std:c++17)
C 语言标准 默认 (ISO C++14 标准)
启用实验性的 C++ 标准库模块 ISO C++14 标准 (/std:c++14)
ISO C++17 标准 (/std:c++17)
ISO C++20 标准 (/std:c++20)
预览 - 最新 C++ 工作草案中的功能 (/std:c++latest)
<从父级或项目默认设置继承>

C++ 语言标准
确定 取消 应用(A)



VS2019 : 属性 → C/C++ → 语言 → C++ 语言标准





重载 运算 []

二维数组的访问

```
class MAT {  
public:  
    int *e;  
    int r, c;  
        //取矩阵 i行的第一个元素地址  
    int* const operator[ ] (int i) {  
        return e+i*c;  
    }  
};  
MAT a (3,4);  
a[0][1]=1;
```





重载 运算 []

```
MAT a (3, 4);  
a[0][1]=1;
```

Q:两个[][]的处理方式为何不同?

它的解析顺序是 (a[0])[1], 在执行 a[0] 时, 隐含的参数类型是 a 的类型, 即 MAT 类型, 此时使用类的运算符函数[], 但其返回类型是 int*, 相当于 int *p;

之后的[], 相当于 p[], 类型不再是 MAT, 故不会调用 MAT类中的运算符[]。



重载 运算()

```
class STRING {  
private:  
    char *s;  
public:  
    STRING(const char *s) {  
        s=new char[strlen(str)+1];  
        strcpy(s,str);  
    }  
    int operator () (int i) {  
        s[0]=s[0]+i;  
        return i+10;  
    }  
};
```

```
STRING s1("S1 hello");  
int x;  
t=s1(1);  
cout<<s1(0)<<s1(1)<<endl;
```

s1(1) 相等于
s1.operator () (1)

定义对象时构造函数的(),
不受()重载影响。

注：本例只是说明()的用法，并无实际意义





重载普通函数，申明为友元

```
class A{  
    int x, y;  
public:  
    A(int x, int y) { A::x=x; A::y=y; }  
    A &operator=(const A&m)// 返回类型为左值，即返回值可再被修改赋值  
    { x=m.x; y=m.y; return *this; } // 左值引用当前对象，赋值后还可赋值  
    friend A operator-(const A&); // 返回右值，参数也为右值，不可修改  
    friend A operator+(const A&, const A&); // const 表示不能修改两个加数  
} a(2,3), b(4,5), c(1, 9);  
A operator -(const A&a){ return A(-a.x, -a.y); } // 普通函数返回右值  
A operator +(const A&x, const A&y){ // 返回右值，(被)加数、结果均不能修改  
    return A(x.x+y.x, x.y+y.y); // A(x.x+y.x, x.y+y.y)为类A的常量  
}  
void main(void){ (c=a+b)=b+b; /*c=a+b, c=b+b*/  
    c= -b; }
```



重载规则

- ◆ 若运算符为左值运算符，则应返回非只读引用类型；
- ◆ 当运算符第一个参数为左值时，不能使用const说明；
 隐含的this指向的对象可以修改；
- ◆ 重载不改变运算符的优先级和结合性；
- ◆ 重载一般也不改变运算符的操作数个数，
 特殊的运算符->、++、--除外。
- ◆ 重载运算符函数一般不能缺省参数，
 只有任意目的运算符()省略参数才有意义；
- ◆ 重载运算符函数可以声明为类的友元；
 重载的普通运算符成员函数可定义为虚函数。



11.2 运算符参数

- ◆ 重载函数种类不同，参数表列出的参数个数也不同。
 - 重载为普通函数：参数个数 = 运算符目数
 - 重载为普通成员：参数个数 = 运算符目数 - 1 (即 **this** 指针)
 - 重载为静态成员：参数个数 = 运算符目数 (没有 **this** 指针)
- ◆ 有的运算符既可以是单目，也可是双目，如 *, +, - 等。
- ◆ 特殊运算符不满足上述关系： \rightarrow 双目重载为单目，前置 $++$ 和 $--$ 重载为单目，后置 $++$ 和 $--$ 重载为双目、函数()可重载为任意目。
- ◆ () 表示强制类型转换时为单参数；
表示函数时可为任意个参数。



11.2 运算符参数

```
class SYMTAB;
struct SYMBOL{
    char *name; int value; SYMBOL *next;
    friend SYMTAB;
private:
    SYMBOL(char*s,int v, SYMBOL *n){ ... };
    ~SYMBOL( ) { ... }
} *s;
class SYMTAB{
    SYMBOL *head;
public:
    SYMTAB( ) { head=0; };
    ~SYMTAB( ){ ... }
    SYMBOL *operator( )(const char *s, int v, int w){ /*...*/};
};
SYTAB tab;
void main(void){ s=tab("a", 1, 2); } // 包括this，有四个参数
```





11.2 运算符参数

- ◆ 运算符++和--都会改变当前对象的值，重载时最好将参数定义为**非只读引用类型(左值)**，左值形参在函数返回时能使实参带出执行结果。
- ◆ 前置运算是先运算再取值，后置运算是先取值再运算。
- ◆ **后置运算应重载为返回右值的双目运算符函数：**
 - 如果重载为类的普通函数成员，要定义一个int类型的参数（已包含一个不用const修饰的this参数）；
 - 如果重载为普通函数，则要定义两个参数非const引用类型参数、int类型参数。
- ◆ **前置运算应重载为返回左值的单目运算符函数：**
 - 前置运算结果应为**左值**，其返回类型应该定义为**非只读类型的引用类型**；左值运算结果可继续++或--运算。
 - 如果重载为普通函数(C函数)，则最好声明非const引用类型一个参数(无this参数)。



11.2 运算符参数

◆ 函数返回

对象的引用 → 左值

对象 → 右值

`const` 对象的引用 → 右值

◆ 函数参数

如何区分 前置 ++、后置 ++? `t++`, `++t`?



11.2 运算符参数

```
class A{  
    int a;  
    friend A &operator--(A&x){x.a--; return x;} //自动内联，返回左值  
    friend A operator--(A&, int); //后置运算，返回右值  
public:
```

```
    A &operator++(){ a++; return *this;} //单目，前置运算  
    A operator++(int){ return A(a++); } //双目，后置运算  
    A(int x) { a=x; }  
}; //A m(3); (m--)-- ;
```

可以，因为--m左值，其后--要求左值操作数

```
A operator--(A&x, int){  
    // x左值引用，实参被修改  
    return A(x.a--); // 先取x.a返回A(x.a)右值，再x.a--  
} //A m(3); (m--)-- ;  
// 不可，因为m--右值，其后--要求左值操作数
```



11.2 运算符参数

重载 ++，时钟，时间增加1秒

```
#include <iostream>
using namespace std;
class Clock {
private:
    int hour;
    int minute;
    int second;
public:
    Clock() {
        hour=minute=second=0;
    }
}
```

```
Clock(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
}
void display() {
    cout << hour << ":" << minute <<
    ":" << second << endl;
}
Clock & operator++(); // 前置++
};
```



11.2 运算符参数

Clock & Clock::operator++()

```
{  
    second++;  
    if (second == 60) {  
        second = 0;  
        minute++;  
        if (minute == 60) {  
            minute = 0;  
            hour++;  
            if (hour == 24) hour = 0;  
        }  
    }  
    return *this;  
}
```



11.2 运算符参数

```
int main()
{
    Clock t(10, 59, 50);
    for (int i = 0;i < 100;i++ ) {
        ++t;
        t.display();
    }
    return 0;
}
```

从例子中看，operator++ 的返回值未使用，
能否用void operator++(); 代替 Clock& operator++();?

单从例子中看，是可以用void operator++(); 来代替。
但是对于 t = ++t; 或者 another = ++t; 就会报错。



理解后置++ 为何返回一个对象

```
int xx = 5;  
int yy;  
yy = xx++; // yy=5; xx=6;
```

重载后置++

```
Clock Clock::operator++(int )  
{ Clock temp(*this);  
    second++;  
    if (second == 60) {  
        second = 0;  
        minute++; .....  
    }  
    return temp;  
}
```

先建立了一个局部对象 temp,
存放了 t1 的副本；再修改 t1；
最后返回 temp.

Clock t1(10, 59, 50);

Clock t2;

t2 = t1++;

// 执行后

// t1 = (10, 59, 51)

// t2 = (10, 59, 50)

Q: 执行过程是什么？

= 运算的优先级别很低

++ 运算级别高

t2 = t1.operator++(0);

类比：t2 = fplus(t1, 0);





重载 ->

//重载双目->，使其只有一个参数(单目)，返回指针类型

```
struct A{  
    int a;  
    A(int x) { a=x; }  
};  
class B{  
    A x;  
public:  
    A *operator ->() { return &x; } // 只有一个参数this，故重载为单目  
    B(int v):x(v) {}  
}b(5);  
void main(void){  
    int i=b->a; // 等价于下一条语句， i=b.x.a=5  
    i=b.operator ->()->a; //i=b.x.a=5  
    i =( b.operator ->() ) ->a;  
    i=*(b.operator->()).a; //i = b.operator ->()->a  
}
```





11.3 赋值与调用

➤ 区别 赋值与定义对象



Array a=b;

// 定义对象 a, 即为 a 分配空间

// 调用以对象为参数的构造函数, 初始化对象 a;

a=b;

// a 的空间已存在

// 要考虑释放 a 已有的体外空间

// 考虑为 a 中的指针分配新的体外空间

// 将 b 中的数据 深拷贝到 a 中



11.3 赋值与调用

➤ 区别 深拷贝赋值与移动赋值



`a=b; // b 是一个命名的对象
// 在赋值后, b 应该保持不变`

`a=临时对象;`

例： `a = f(...);` `f(...)` 返回对象

例： `a = u + v;` `u+v` 的结果是一个临时对象

`// 临时对象在赋值语句执行后会被析构`

`// 考虑 a 中的指针直接指向临时对象中指针指向的
体外空间, 然后将临时对象中的指针置空`



11.3 赋值与调用

- $=$ 、 $+=$ 、 $*=$ 、 $\&=$ 、 $|=$ 等是左值运算符
- 编译程序为每个类都提供了缺省赋值运算符函数
- 缺省赋值运算是实现数据成员的复制或浅拷贝赋值，对指针类型的数据成员，不复制指针所指存储单元的内容。
若类不包含指针，浅拷贝赋值不存在问题。



11.3 赋值与调用

赋值运算符重载和复制对象构造函数的比较

- 两者功能相似
- 实现方式相似
- 赋值运算符显式调用，有 =
- 复制对象构造函数自动调用
以一个对象为参数，生成另一个对象
- 两个都有默认的函数，都是浅拷贝

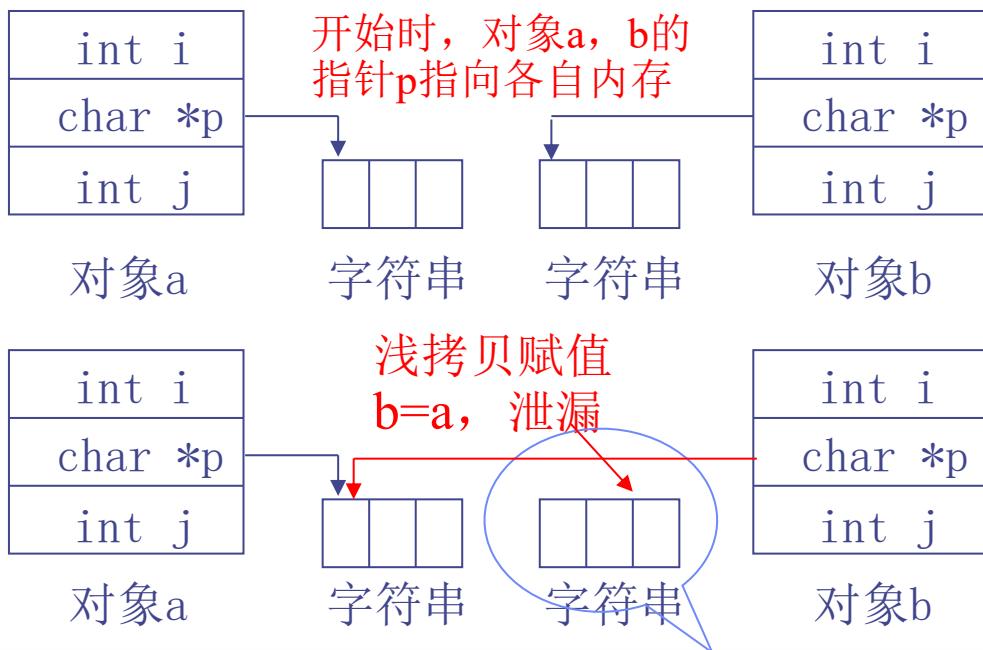


11.3 赋值与调用

- 编译程序为每个类提供了缺省赋值运算符函数
对类A而言，其成员函数原型为 $A& operator=(const A&)$ 。
- 缺省赋值运算是实现数据成员的复制或浅拷贝赋值，
对指针类型的数据成员，不复制指针所指存储单元的内容。
若类不包含指针，浅拷贝赋值不存在问题。
- 如果类自定义或重载了赋值运算函数，则优先调用类自定义或重载的赋值运算函数(不管是否取代型定义)。
- 如果函数参数要值参传递一个对象，当实参传值给形参时，
若类A没有定义 $A(const A&)$ 形式的构造函数，则值参传递
也通过浅拷贝赋值实现。

11.3 赋值与调用

当类包含指针时，浅拷贝赋值可造成内存泄漏，并可导致页面保护错误或产生副作用。





11.3 赋值与调用

```
#include <string.h>
class STRING{
    char *s;
public:
    virtual char &operator[ ](int x){ return s[x]; }
    STRING(const char *c){strcpy(s=new char[strlen(c)+1], c); }
    STRING(const STRING &c){strcpy(s=new char[strlen(c.s)+1], c.s); }
    virtual STRING operator+(const STRING &)const;
    //加数、被加数及和(即函数返回值)都不能被修改或赋值，返回右值
    virtual STRING&operator=(const STRING &); //返回左值可连续赋值
    virtual STRING&operator+=(const STRING&s){return *this=*this+s;}
    virtual ~STRING( ) { if(s){ delete [ ]s; s=0; }}
} s1("S1"), s2="S2", s3("S3");//s2="S2"等价于是s2("S2")
```



11.3 赋值与调用

```
STRING STRING::operator+(const STRING &c) const{
    char *t=new char[strlen(s)+strlen(c.s)+1];
    STRING r(strcat(strcpy(t,s),c.s)); //strcpy、strcat返回t
    delete []t;   return r;
}

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

void main(void){
    (s1=s1+s2)=s2; //重载“=” 返回左值，可连续赋值否则不可
                     //等价于s1=s1+s2; s1=s2;s1被连续赋值
    s1+=s3;
    s3[0]='T';// s3[0]=调用char &operator[](int x)返回左值
}
```



11.3 赋值与调用

对于类T，防止内存泄露要注意以下几点：

- 不要随便使用exit和abort退出程序；
- 定义T(const T &)等形式的深拷贝构造函数；
- 定义virtual T &operator=(const T &)等形式的深拷贝赋值运算符函数；
- 定义virtual ~T()形式的虚析构函数；
- 在定义引用T &p=*new T()后，使用delete &p析构并释放对象占用的内存；
- 在定义指针T *p=new T()后，使用delete p析构并释放对象占用的内存。



11.4 强制类型转换

- C++是强类型的语言，运算时要求类型相容或匹配。
- 定义合适的类型转换函数，可完成操作数的类型转换；

```
int x = 3;    double y=4.6;
```

```
x = y; // 警告：从double 转换到 int，可能丢失数据
```

```
x = (int)y;
```

```
x = int(y); // 等价写法
```

```
cvttsd2si eax, mmword ptr [y]
```

```
mov     dword ptr [x], eax
```

讨论：数据类型转换 与 地址类型转换有何差别？

`x=int(y); // x=4`

`x =*(int *)&y; // x =1717986918`





11.4 强制类型转换

设有复数类 class COMPLEX {double r, v;};

要求完成如下功能，如何实现？

复数 C3 = 复数 C1 + 复数 C2

复数 C3 = 复数C1 + 浮点数 r2

复数 C3 += 复数 C1

复数 C3 += 浮点数 r2



11.4 强制类型转换

```
class COMPLEX{  
    double r, v;  
public:  
    COMPLEX(double r1, double v1);  
    COMPLEX operator+ (const COMPLEX &c) const;  
    COMPLEX operator+ (double d) const;  
    COMPLEX operator+ (int d) const;  
    COMPLEX operator- (const COMPLEX &c) const;  
    COMPLEX operator- (double d) const;  
    COMPLEX operator- (int d) const;  
};
```

方法1：定义合适的构造函数，可以构造符合类型要求的对象，构造函数可以起到类型转换的作用。

方法2：定义类型转换函数，实现类型转换





11.4 强制类型转换

◆ 单参数的构造函数具备类型转换作用

能自动将参数类型的值转换为要构造的类型。

◆ C++会自动将 int 转为 double

```
class COMPLEX{
```

```
    double r, v;
```

```
public:
```

```
    COMPLEX(double r1);
```

```
    COMPLEX(double r1, double v1){ r=r1; v=v1; }
```

```
    COMPLEX operator+(const COMPLEX &c) const;
```

```
    COMPLEX operator-(const COMPLEX &c) const;
```

```
};
```

```
COMPLEX m(3);
```

m + 2 转换为 m + 2.0; 再转换为 m + COMPLEX(2.0)





11.4 强制类型转换

- 表面上是多参数，但是有缺省参数时，
等同单参数的构造函数具备类型转换作用

- C++会自动将 int 转为 double

```
class COMPLEX{  
    double r, v;  
public:  
    COMPLEX(double r1, double v1=0){ r=r1; v=v1; }  
    COMPLEX operator+(const COMPLEX &c) const;  
    COMPLEX operator-(const COMPLEX &c) const;
```

};

COMPLEX m(3);

m + 2 转换为 m + 2.0; 再转换为 m + COMPLEX(2.0)



11.4 强制类型转换 - explicit

```
class COMPLEX {  
    double r, v;  
public:  
    explicit COMPLEX(double r1=0, double v1 = 0)  { r = r1; v = v1; }  
    COMPLEX operator+(const COMPLEX &c) const  
    {      return COMPLEX(r + c.r, v + c.v);    };  
    explicit operator double() { return r; }  
}m(2,3);  
  
double d=m; //error无法从COMPLEX 转换为 double  
z=m + 2.0;   // error :没有与操作数匹配的运算符  
  
double d=m.operator double();  
    d = double(m);  
z=m + COMPLEX(2.0);
```





11.4 强制类型转换

- ◆ 单参数的构造函数相当于类型转换函数

T::T(A)

T::T(const A)

T::T(const A &)

相当于A类到T类的强制转换函数。

COMPLEX(double r1); // 由浮点数类型转换为COMPLEX类型



11.4 强制类型转换

- ◆ 用operator定义强制类型转换函数。 operator 类型(...)
转换后的类型就是函数的返回类型， 不需要定义返回类型。
- ◆ 不能同时定义T::T(A) 和 T::T(const A&)
表面上看， 两者是不同的。
但若有语句 T(A)， 编译报错： 对重载的调用不明确。
- ◆ 类型转换的结果通常为右值， 故最好不要将类型转换函数
的返回值定义为左值， 也不应该修改当前被转换的对象
(参数表后用const说明this)。



11.4 强制类型转换

```
struct A{  
    int i;          A(int v) { i=v; }  
    virtual operator int( ) const{ return i; } //类型转换返回右值  
}a(5);  
struct B{  
    int i, j;      B(int x, int y) { i=x; j=y; }  
    operator int( ) const{ return i+j; }      //类型转换返回右值  
    operator A( ) const{ return A(i+j); }     //类型转换返回右值  
}b(7, 9), c(a, b);  
  
void main(void){  
    int i=1+(int)a; //强制转换，调用A::operator int( )转换a, i=6  
    i = a;           // i = a. operator int( );  
    i=b+3;          //自动转换，调用B::operator int( )转换b, i=19  
    i=a=b;          //调用B::operator A( )和A::operator int( ), i=16  
                      // i = a = b.operator A();  
}
```



11.5 重载new和delete

- ◆ 运算符函数new和delete定义在头文件new.h中

```
extern void * operator new(unsigned bytes);  
extern void operator delete(void *ptr);
```

- ◆ 运算符new分配内存的大小

类型表达式而不是值表达式作为实参

```
new long[20]      // sizeof(long)*20
```

- ◆ new和delete可重载为普通函数，

也可重载为静态函数成员。



总结

- 赋值运算符的重载

ClassName & operator =(const ClassName &obj);

- 加法运算符的重载

ClassName operator +(const ClassName &obj);

- [] 运算符的重载

ReturnType & operator [](int i);

- () 运算符的重载

ReturnType [?] operator ()(0个或者多个参数);

- 前置 ++ 的重载

ClassName & operator ++();

- 后置 ++ 的重载

ClassName operator ++(int);





总结

- 单参数的构造函数具备类型转换作用
- 多参数，除首参外，其他参数是缺省参数时，等同单参数的构造函数，具备类型转换作用
- 强制类型转换函数
operator 类型(...);