



C++程序设计精要教程

华中科技大学

第11章 运算符重载

◆11.1 运算符概述

- 纯单目运算符，只能有一个操作数，包括：!、~、sizeof、new、delete 等
- 纯双目运算符，只能有两个操作数，包括：[]、->、%、= 等
- 三目运算符，有三个操作数，如“?:”
- 既是单目又是双目的运算符，包括：+、-、&、* 等
- 多目运算符，如函数参数表“()”。
- 左值运算符是运算结果为左值的运算符，其表达式可出现在等号左边，如前置++、-- 以及赋值运算=、+=、*= 和 &= 等。右值运算符是运算结果为右值的运算符，如+、-、>>、%、后置++、-- 等。
- 某些运算符要求第一个操作数为左值，如++、--、=、+=、&=等。

第11章 运算符重载

【例11.1】传统左值运算符的用法

```
#include <stdio.h>
```

```
void main(int argc, char *argv[ ])
```

```
{   int x=0;
```

```
    ++x;           //++x 的结果为左值（可出现在等号左边）
```

```
    ++ ++x;        //++x 仍为左值，故可连续运算，x=3
```

```
    --x = 10;       //--x 仍为左值，故可再次赋值，x=10
```

```
    (x = 5) = 12;    //x=5 仍为左值，故可再次赋值，x=12
```

```
    (x += 5) = 7;    //x+=5 仍为左值，故可再次赋值，x=7
```

```
    printf("%d %d", x, x++); //( )可看作任意目运算符
```

```
} //(x--)+是错的：x--的结果为右值，而++要求一个左值
```


第11章 运算符重载

◆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。故C++规定运算符重载必须针对类的对象，即**重载时至少有一个参数代表对象**（类型如A、const A、A &、const A &、volatile A等）。
- C++用 **operator加运算符** 进行运算符重载。对于普通运算符成员函数，this 隐含参数代表第一个操作数对象。
- 根据能否重载及重载函数的类型，运算符分为：
 - 不能重载：**sizeof、.、.*、::、?:**
 - 只能重载为类的普通成员函数：**=、->、()、[]**
 - 只能重载为普通函数：**new、delete**
 - 其他运算符：**可以重载为类的普通成员函数和普通函数，但不能重载为类的静态成员函数。**

第11章 运算符重载

◆11.1 运算符概述

- 若运算符为左值运算符，则重载后运算符函数最好返回非只读引用类型(左值)。当运算符要求第一个参数为左值时，不能使用 `const` 说明第一个参数(含 `this`)，例如 `++`、`--`、`=`、`+=` 等的第一个参数。
- 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也可定义为虚函数；重载的非成员函数被视为普通函数。
- 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。

第11章 运算符重载

```
class A;
```

```
int operator = (int, A &);    //错误, 不能重载为普通函数
```

```
A &operator +=(A &, A &);    //正确, 重载 += 为普通函数
```

```
A &operator ==(A *a, A b[ ]); //A * 和 A[ ] 参数不代表对象
```

```
class A {
```

```
    friend int operator=(int, A &);    //错误, 不存在普通函数operator=( )
```

```
    static int operator( )(A &, int);    //错误, 不能为静态成员
```

```
    static int operator+(A &, int);    //错误, 不能为静态成员
```

```
    friend A & operator += (A &, A &); //正确, operator += ( ) 是什么函数?
```

```
    A &operator ++( );    //隐含参数this代表一个对象
```

```
};
```

第11章 运算符重载

```
#include <iostream>
using namespace std;

class A {
    int x;
public:
    int getx ( ) const { return x; } //const A *const this, 代表对象
    A(int x) { A::x = x; }          //A *const this
};

int operator+(const A &x, int y) //参数const A &x 代表一个对象
{ return x.getx( ) + y; }        //能否写成: return x.x + y ?

int operator+(int y, A x)        //参数A x 代表一个对象
{ return x.getx( ) + y; }
```

Problem: (1) 2个operator+()是什么函数？ 类的成员函数？ 全局普通函数？
(2) 2个operator+()函数可以放到类A里面吗？ Why？

第11章 运算符重载

//不能声明 `int operator+(A[6], int);` //A[6]不是单个对象

//不能声明 `int operator+(A*, int);` //A * 是对象指针, 属于简单类型, 不代表对象

`void main(void)`

{

`A a(6);` //调用A(int)时, 实参&a传递给隐含形参this

`cout << "a+7=" << a+7;` //调用`int operator+(const A &, int)`

`cout << "a+7=" << operator+(a, 7);` //调用`int operator+(const A &, int)`

`cout << "8+a=" << operator+(8, a);` //调用`int operator+(int, A)`

`cout << "8+a=" << 8 + a;` //调用`int operator+(int, A)`

}

第11章 运算符重载

```
#include <iostream>
using namespace std;
class A {
    int x;
public:
    int getx ( ) const { return x; }    //const A *const this, 代表对象
    A(int x) { A::x = x; }              //A *const this
    int operator+(const A &x, int y) //error
    int operator+(const A &x)
    { return x.getx( ) + y; }           //能否写成: return x.x + y ? YES
    int operator+(int y, A x) { ... } //error
};
```

Problem: (1) 2个operator+()是什么函数？ 类的成员函数？ 全局普通函数？
(2) 2个operator+()函数可以放到类A里面吗？ Why？

第11章 运算符重载

◆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)
- 有的运算符既为单目又为双目，如 *, +, - 等。
- 特殊运算符不满足上述关系：->双目重载为单目；前置++、--重载为单目，**后置++、-- 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。

第11章 运算符重载

- `()` 运算符可以重载为任意目的运算符函数：`operator (...)`;
- 实现了`operator()`的类称为**仿函数**，这个类的对象就是**函数对象**，这样的类具有类似函数的行为（将类名当函数名一样使用）;
- 调用`operator(...)`的简洁方式为：**对象名 (...)**
(把对象名看成函数名)

第11章 运算符重载

```
struct A {  
    int a;  
    A( ) { a = 0; }  
    A(int x) { a = x; }  
    int f( ) { return a; }  
    int operator( ) (int x, int y) {  
        return a + x + y;  
    }  
    ~A( ) { cout << a; }  
} a(0);  
  
int main( ) {  
    cout << a.operator( )(1, 2);  
    cout << a(1, 2);  
    cout << A(1)(1, 2); //创建临时对象A(1) => operator()(1,2) => 析构  
    cout << A( ) (1, 2); //创建临时对象A( ) => operator()(1,2) => 析构  
}
```


第11章 运算符重载

◆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)。
- 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，则该函数只需定义一个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(C函数)，则最好声明非const引用类型和int类型的两个参数(无this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)。

第11章 运算符重载

```
class A {  
    int a;  
    friend A &operator--(A &x) {x.a--; return x; } //前置运算 (普通函数), 返回左值  
    friend A operator--(A &, int); //后置运算(普通函数), 返回右值  
public:  
    A &operator++() { a++; return *this; } //前置运算 (成员函数)  
    A operator++(int) { return A(a++); } //后置运算 (成员函数)  
    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
```

第11章 运算符重载

//重载双目->, 使其只有一个参数(单目), 重载->必须返回指针类型.

```
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; //被解释为: i = (b.operator->( )) -> a, i = 5
```

```
    i = b.operator->( ) ->a; //i = b.x.a = 5
```

```
    i = (*b.operator->( )).a; //i = (&*b.operator->( ))->a = b.operator ->( )->a
```

```
}
```

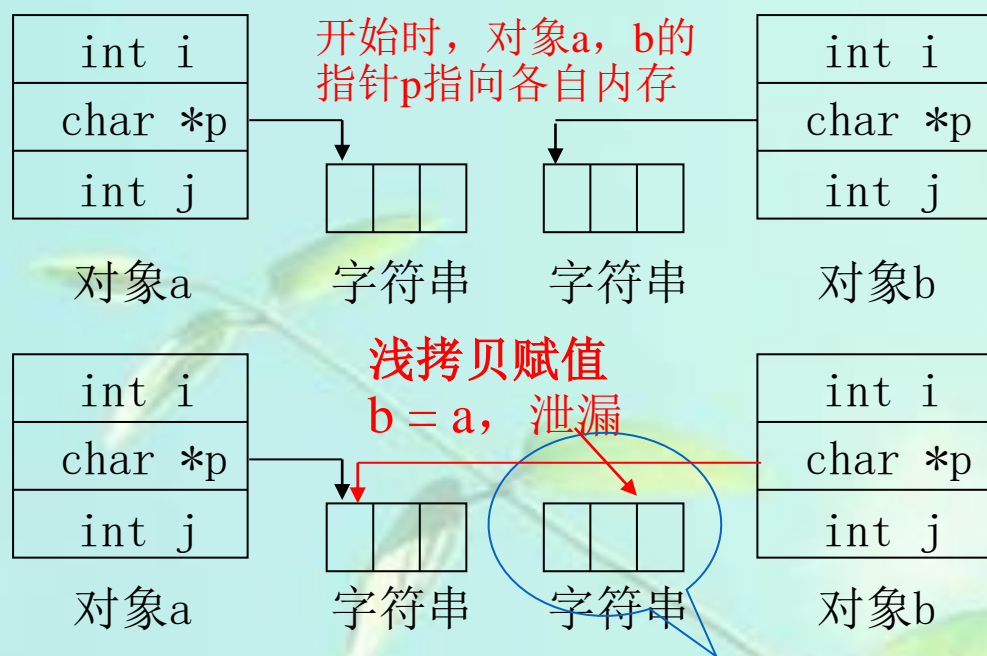
第11章 运算符重载

◆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（浅拷贝），对类A而言，其成员函数原型为 `A &operator=(const A &)`。
- 如果类自定义或重载了赋值运算符函数，则优先调用类自定义或重载的赋值运算符函数（不管是否取代型定义）。
- 缺省赋值运算符实现数据成员的浅拷贝复制，如果数据成员为指针类型，则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。

第11章 运算符重载

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变量产生副作用。



第11章 运算符重载

```
#include <string.h>
#include <iostream>
using namespace std;
class STRING {
    char *s;
public:
    STRING(const char *c) { strcpy(s = new char[strlen(c)+1], c); }
    STRING(const STRING &s); //深拷贝构造函数
    STRING(STRING &&s) noexcept; //移动构造函数
    virtual STRING &operator=(const STRING &s); //深拷贝赋值函数
    virtual STRING &operator=(STRING &&s) noexcept; //移动赋值函数
    virtual char &operator[ ](int x) { return s[x]; }
    virtual STRING operator+(const STRING &) const;
    virtual STRING &operator+=(const STRING &s) { return *this = *this + s; };
    virtual ~STRING( ) noexcept { if (s) { delete[ ]s; s = 0; }; };
};
```

第11章 运算符重载

```
STRING:: STRING(STRING &cs) noexcept {  
    strcpy(s=new char[strlen(cs.s)+1], cs.s);  
}  
STRING:: STRING(STRING &&cs) noexcept {  
    s = cs.s; cs.s = 0;  
}  
STRING &STRING:: operator=(STRING &&s) noexcept  
{  
    s = cs.s; cs.s = 0;  
    return *this;  
}  
STRING &STRING::operator=(const STRING &cs) {  
    delete [ ] s; //this->~STRING()  
    strcpy(s=new char[strlen(cs.s)+1], cs.s); //new (this) STRING(cs); ???  
    return *this;  
}
```

第11章 运算符重载

```
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;  
}
```

```
void main(void) {  
    STRING s1("123"), s2("abc"); s3("hello");  
    (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  
}
```


第11章 运算符重载

对于类T，防止内存泄露要注意以下几点：

- (1) 应定义 `T(const T &)` 形式的深拷贝构造函数；
- (2) 应定义 `T(T &&) noexcept` 形式的移动构造函数；
- (3) 应定义 `virtual T &operator=(const T &)` 形式的深拷贝赋值运算符；
- (4) 应定义 `virtual T &operator=(T &&) noexcept` 形式的移动赋值运算符；
- (5) 应定义 `virtual ~T()` 形式的虚析构函数；
- (6) 在定义引用 `T &p = *new T()` 后，要用 `delete &p` 删除对象；
- (7) 在定义指针 `T *p = new T()` 后，要用 `delete p` 删除对象；

第11章 运算符重载

◆11.4 强制类型转换

- 如定义了合适的**类型转换函数**，就可以完成操作数的类型转换；如定义了合适的构造函数，就可以构造符合类型要求的对象，**构造函数也可以起到类型转换的作用**。
- 对象与不同类型的数据进行运算，可能出现在双目运算符的左边和右边，为此，可能需要定义多种运算符重载函数。
- 只定义几种运算符重载函数是可能的，即限定操作数的类型为少数几种乃至一种。如果运算时对象类型不符合操作数的类型，则可以通过类型转换函数转换对象类型，或者通过构造函数构造出符合类型要求的对象。

第11章 运算符重载

定义“复数+复数”、“复数+实数”、“复数+整数”、“复数-复数”、“复数-实数”、“复数-整数”几种运算(还有复数同实数乘除运算,等等):

```
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;  
    ... .. //更多的 +、- 运算法重载函数  
};
```

第11章 运算符重载

- 单参数的构造函数具备类型转换作用，必要时能自动将参数类型的值转换为要构造的类型。以下通过定义单参数构造函数简化重载（同时注意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;  
};
```

调用单
参数构
造函数

- 定义COMPLEX m(3)，m+2转换为m+2.0转换为 **m+COMPLEX(2.0)**。

第11章 运算符重载

单参数的构造函数、
强制类型转换函数、
作用相反 !!!

◆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，单参数的 `T::T(const A)`、`T::T(A &&)`、`T::T(const A &)` 等相当于A类到T类的强制转换函数。
- 强制类型转换函数：`operator 类型表达式 ()`。由于转换后的类型就是函数的返回类型，所以强制类型转换函数不需要定义返回类型，也不带输入参数（表示将 `this` 对象转换为其他类型）。
- 同时定义 `A::operator T()` 和 `T::T(const A &)` 容易出现二义性错误。
- 按照C++约定，强制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`）。
- C++规定转换的类型表达式不包含 `()` 和 `[]`，只能使用引用、指针。如 `operator int A::**const&()` 正确，而 `operator int(*)*()` 是错误的。

第11章 运算符重载

```
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) { //分析下面各条语句的执行过程
```

```
    int i = 1 + a; //a.operator int() => 1 + int
```

```
    A x = b; //x(b): b.operator A() => A(const &A) (编译器提供)
```

```
    A y(a); //A(const A &) (编译器提供)
```

```
    x = b; //b.operator A() => A(int) => x.operator=(int) (编译器提供) => ~A()
```

```
    x = b + a; //b.operator int() + a.operator int() => A(int) => x.operator=(int) => ~A()
```

```
    printf(“%d %d”, a, (int)a); //VFT地址 5
```

```
} //若去掉A:: operator int() 中的virtual, printf() 结果? 5 5
```


第11章 运算符重载

```
class A
{
    int a;
public:
    A(int x) { a = x; }
    A(const A &x) { a = x.a; }
    A &operator=(const A &x) { a = x.a; return *this; };
    operator int() const { return a + a; }
    ~A() {}
};

void main()
{
    A a(1), b(a);    //A(int), A(const A &)
    A c = 1;         //A(1)
    A d = a;         //A(a),不调用operator=()
    c = 1;           //A(1), operator=(), ~A()
    d = a;           //operator=()
    d = 1 + a;       //a.operator int(), A(int), operator=(), ~A()
}
```

Problem:

(1) 若没有构造函数

`A::A(const A &x) ?`

(2) 可以定义构造函数

`A::A(const A x) 吗?`

第11章 运算符重载

```
class A
{
    int a;
public:
    A(int x) { a = x; }
    A &operator=(const A &x) { a = x.a; return *this; };
    operator int() const { return a + a; }
    ~A() { }
};
```

```
void main()
{
    A a(1), b(a), x(a+1);    //A(int), 浅拷贝a到b, { int(), +, A(int) }
    A c = 1;                //A(1), 不调用~A()
    A d = a;                //A(a), 不调用operator=(), 浅拷贝a到d, 等价d(a)
    c = 1;                  //A(1), operator=(), ~A()
    d = a;                  //operator=()
    d = 1 + a;              //operator int(), ADD, A(int), operator=(), ~A()
}
```


第11章 运算符重载

◆11.5重载new和delete

- 运算符函数new和delete定义在头文件new.h中，new的参数就是要分配的内存的字节数。其函数原型为：

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

```
extern void operator delete(void *ptr);
```

- 在使用运算符new分配内存时，使用类型表达式而不是值表达式作为实参，编译程序会根据类型表达式计算内存大小并调用上述new函数。例如：new long[20]。
- 按上述函数原型重载，new和delete可重载为普通函数，也可重载为类的静态成员函数。
- OS的最小内存分配单位为节(16字节，即使new char)，故重载new可先分得OS一大块内存，然后再分给需要单个字符的指针变量。