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（作为第二个++的操作数），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
```

运算符重载工作机制

运算符重载就是函数重载，编译器在编译时确定调用哪个运算符重载函数

```
class A {
public:
   int i;
   A (int x): i(x) {}
   //重载双目运算符+，使”+”能支持二个A类型对象相加
   //二个A类型对象相加的语义由operator+函数负责实现
   //由于双目运算符+要求2个操作数，因此operator+也要求2个操作数
   //如果用A的普通成员函数（实例函数）实现，则只需要一个显式参数（因为隐含的//this指针指//向了第一个操作数，即出现在+左边的操作数）
   int operator+( A a) {  return this->i + a.i;}
};
//用全局函数重载双目“-”，这时要显式指定二个参数（a1为第一个操作数）
int operator-(A a1, A a2) { return a1.i – a2.i;}
```

运算符重载工作机制

```
A a1(10), a2(20);
int i = a1.i + a2.i;
int j = a1 + a2;
int k = a1- a2;
```

int i = a1.i + a2.i; 当编译器检查到这个语句时，会检查二个操作数的类型。检查结果是int + int，编译器会调用内置的简单类型的operator+。

int i = a1 + a2; 当编译器检查到这个语句时，会检查二个操作数的类型。检查结果是A + A，内置的简单类型的operator+不支持。编译器会在Class A里寻找operator+（A，A）函数，如果找到，将int j = a1 + a2;语句编译成int j = a1.operator+(a2)

int k = a1 - a2; 编译器类型检查结果是A - A。编译器会在Class A里寻找operator-（A，A）函数，如果没找到，将在全局函数里寻找。最后int k = a1 - a2;语句编译成int k = operator-(a1, a2)

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（复杂类型），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)。（注意A*，A[]不是复杂类型，是简单类型：指针）
- C++用operator加运算符进行运算符重载。如果用类的普通成员函数重载运算符，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：

- 不能重载的：sizeof　.　.*　::　?:
- 只能重载为类的普通成员函数的：=、->、()、[]
- 不能重载为类的普通成员函数的：new、delete
- 其他运算符：不能重载为类的静态成员函数，但可以重载为类的普通成员函数和全局函数。

	类的普通成员函数	类的静态成员函数	全局函数
=、->、()、[]	√	×	×
new、delete	×	√	√
其他	√	×	√

	类的普通 成员函数	类的静态 成员函数	全局函数
=、->、()、[]	√	×	×
new、delete	×	√	√
其他	√	×	√

以上规则只是语法上的规定。但是对于具体的运算符，重载时要根据具体运算符的操作语义来权衡。例如>>、<<重载时，就不应该用类的普通成员函数来重载。例如若用A类的普通成员函数去重载<<，第一个操作数就必须是A类对象（例如a），第二个操作数是输出流对象，就会出现如下奇怪的语法：a << cout;

又例如，若想实现一个整数（作为第一个操作数）和A类对象相加，这时显然只能用全局函数去重载A operator+(int, const A &);

```
class A;
int operator= (int, A&);        //错误，=不能重载为全局函数
A& operator += (A&s, A&r) //注意，+= 可以重载为全局函数，
{ return  s; }
//注意这时参数不能写A s，因此时不知A的字节数。而A *r可以。
int operator+(A[6],int); //不允许，A[6]和A *为普通类型
int operator+(A *, int);  //不允许，A[6]和A *为普通类型
class A{
      friend int operator=(int, A&);  //错误，=不能为全局函数重载
      static int operator ( ) (A&, int); //错误，不能为静态成员重载函数调用（）
      static int operator+ (A&, int);    //错误，不能为静态成员
      friend A& operator+=(A&, A&);  //正确，+=可以用全局函数重载
      A& operator ++( );    //隐含参数this代表一个对象
};
//注意重载左值运算符如=、+=、…时最好返回非const左值引用，凡是左值参数最好都用引用(非const)，否则改变了运算符的性质
```

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左值引用类型(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；重载运算符的普通成员函数也可定义为虚函数；重载运算符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->、++、--（区分前置和后置）除外。

第11章 运算符重载

```
#include <iostream>
using namespace std;
class A{
   int x;
public:
   int getx ( )const{ return x; }      //隐含参数this的类型为const A*const this，可代表对象
   A(int x) { A::x=x; }                //隐含参数this的类型为A*const this
};
int operator+(const A&x, int y)   //定义非成员函数(全局函数重载+)：参数const A&x代表一个对象
{  return x.getx( )+y; }
int operator+(int y, const A& x)  //定义非成员函数(全局函数重载+)：参数const A&x代表一个对象
{  return x.getx( )+y; }
```

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，不能用于代表对象
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<<"\n";        //调用int operator+(const A&, int)
   cout<<"a+7="<< operator+(a,7)<<"\n";  //调用int operator+(const A&, int)
   cout<<"8+a="<< operator+(8,a)<<"\n";  //调用int operator+(int, const A& )
   cout<<"8+a="<<8+a<<"\n";        //调用int operator+(int, const A&,)
}
```

```
class A{
   int  x, y;
public:
   A (int x, int y) { A::x=x; A::y=y; }
   A &operator=(const A&m)        //取m送给this指向的可修改对象
   { x=m.x; y=m.y; return *this; } //返回非const引用，赋值后还可赋值
   friend A operator-(const A&);  //全局函数重载,返回右值,单目-
   friend A operator+(const A&, const A&); //const不能修改两个加数
} a(2, 3), b(4, 5), c(1, 9);
A operator-(const A&a)                    //重载为单目运算符
{ return A(-a.x, -a.y); }                 //返回右值（临时变量）
A operator+(const A&x, const A&y) {       //返回右值
   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; /*等价于c=a+b, c=b+b, (c=a+b)返回的是c的引用*/
   c= -b;
}
```

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(全局函数）：参数个数＝运算符目数
  - 重载为类的普通成员函数（实例函数）：参数个数＝运算符目数-1(即this指针)
  - 重载为类的静态成员函数：参数个数 ＝ 运算符目数(没有this指针)
- 注意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++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。

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const左值引用类型(左值)，左值形参在函数返回时能使实参带出执行结果，非const保证可以改变操作数。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（返回类型为值类型）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)；
  - 如果重载为全局函数，则最好声明非const左值引用类型（原因同上）和int类型的两个参数(无this参数)。
- 前置运算应重载为返回非const左值引用的单目运算符函数：
  - 前置运算结果应为非const左值，其返回类型应该定义为非只读类型的左值引用类型；左值运算结果可继续++或--运算。
  - 如果重载为全局函数，则最好声明非const左值引用类型一个参数(无this参数)。

重载++，--

- 前置++，--要返回非const左值引用，否则不能出现在＝号左边，改变了运算符的性质
- 前置++，--和后置++，--参数要为非const左值引用，否则无法将结果带回，也改变了运算符的性质。因为++，--（不管是前置还是后置）都要求改变操作数自身的值。如果参数为值参，会导致传值调用，函数里改变的是形参，实参不会改变。
- 上述二个规则不是语法规定，而是语义上的规定。

第11章 运算符重载

```
class A{
   int  a;
   //参数和返回值都是非const左值引用，全局函数重载前置--，先运算，后返回
   friend A &operator--(A&x){x.a--; return x; }
   friend A operator--(A&, int); //全局函数重载后置--，返回右值
public:
   A &operator++( ){ a++;  return *this; }//实例函数重载，为单目，前置++，先运算，后返回
   A operator++(int){ return  A(a++); }//实例函数重载，为双目，后置++
   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--);
} //A m(3); (m--)--不可；因为m--右值，其后--要求左值操作数
```

后置++，--重载的语义

```
struct A{
    static int  x;
    int  y;
public:
    const A operator ++(int){
            return A(x++, y++);
    }
    A(int i, int j){ x=i; y=j;}
};
int A::x = 23;
int main(){
    A a(4,3);
    A b = a++; //b.x, b.y ,a.x, a.y=?
    return 0;
}
```

a(4,4), b(4,3)

return A(x++,y++)等价于
1：先取x，y的值
temp1 = x; //4
temp2 = y; //3
2: x加1,y加1,对象a变为a(5,4)
3:构造临时对象tempo
tempo = A(temp1,temp2); //A(4,3)
x为静态成员，使得a变为a(4,4)
4:返回tempo给b，b为b(4,3)

第11章 运算符重载

重载双目->，使其只有一个参数（单目），返回指针类型

双目->,是唯一可以重载为单目运算的纯双目运算符

下面例子说明->运算符重载后左操作数不是对象指针，右操作数也不是对象成员

```
struct A{ int a;  A (int x){ a=x;  } };
class  B{
          A  x;
public：
          A *operator-> ( ) { return &x; };  //只有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
}
```

当返回指针后，则内置的成员访问操作符->的语义被应用在返回值上

重载纯单目和纯双目运算符

```
class A{
        int x;
public:
        A(int y):x(y){}
        A operator%(A m) { return A(x%m.x); }  //双目,返回的是右值
        A operator!() { return A(!x); }        //单目,返回的是右值
};

void main(int argc, char* argv[])
{
        A a(3),b(3);
        b = a%b;        //b.x = 0
        b= !a;          //b.x = 0
}
```

调用运算符函数的二种形式：表达式，函数调用

```
class POINT{
        int x,y;
public:
        POINT(int x,int y):x(x),y(y){}
        POINT operator-() { return POINT(-x,-y); } //单目-
        POINT operator-(POINT p) { return POINT(x-p.x,y-p.y);}//双目-
        POINT operator+(POINT p) { return POINT(x+p.x,y+p.y); }//双目+
        friend POINT operator+(POINT p) { return POINT(p.x,p.y);}//单目+
        friend POINT operator+(POINT p1,POINT p2) {//双目+
                return POINT(p1.x+p2.x,p1.y+p2.y);
        }
};
void main(int argc, char* argv[]){
    POINT p1(1,2),p2(3,4);
    POINT p3=-p1;                   //调 POINT operator-()
    POINT p4=p1-p2;                 //调 POINT operator-(POINT p)
    POINT p5=p1.operator -(p2);     //调 POINT operator-(POINT p)
    POINT p6 = +p1;                 //调 friend POINT operator+(POINT p)
    POINT p7=operator+(p1,p2);      // friend POINT operator+(POINT p1,POINT p2)
    POINT p8 = p1.operator +(p2);   //调 POINT operator+(POINT p)
    POINT p9 = p1+p2;               //错误，无法确定是调用友元还是函数成员
}
```

重载函数调用操作符（ ）

为多目运算符

```
int sum(int x, int y) { return x + y;}
int s = sum (1,2); //三个操作数sum, 1, 2
//第一个操作数为函数名，这里把函数名看做函数对象
```

()只能通过类的普通成员函数重载，这意味着()第一个操作数必须是this指针指向的类对象，这样的对象称为**函数对象（也称为仿函数（Functor））**。

```
class AbsInt{
public:
   //语法：returnType operator( ) (参数列表)
   //调用：objectofAbsInt(实参)，类AbsInt的对象称为函数对象
   int operator()( int val) { return val > 0? val:-val; }
} absInt;
int i = absInt(-1); //函数对象可以作为函数的参数，这意味着我们可以将函数当做对象传递. 在模板编程里广泛使用。在C+11里，还可以用lamda表达式
```

函数指针也可以作为函数的参数，但函数指针主要的缺点是：
被函数指针指向的函数是无法内联的。而函数对象则没这个问题。

第11章 运算符重载

◆11.3 赋值与调用

默认情况下，编译器为类A默认生成6个实例成员函数：A()、A(A&&)、A(const A&)、~A()、A&operator=(const A&)、A& operator=(A&&)

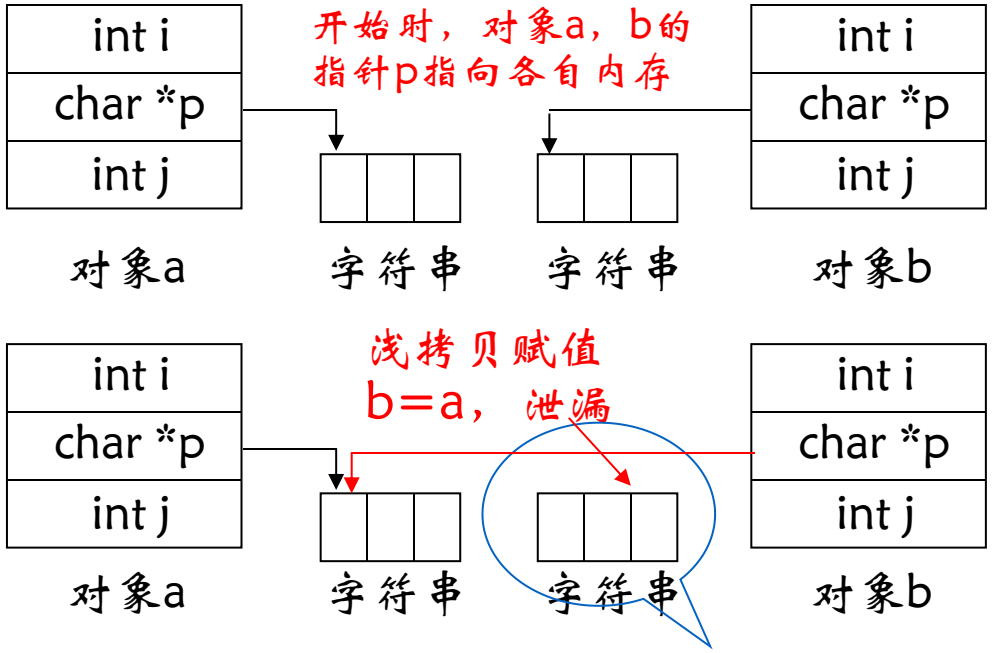
如果类自定义上述函数，则优先调用自定义的函数；若函数体用=default取代，表示启用上述函数；若用=delete取代表示删除上述函数，且不得再定义同型函数，即禁止使用。

默认赋值运算函数实现数据成员的浅拷贝赋值，如果普通数据成员为指针或引用类型且指向了一块动态分配的内存，则不复制指针所指存储单元的内容。若类不包含上述普通数据成员，浅拷贝赋值不存在问题。

如果函数参数为值参对象，当实参传值给形参时，若类A没有自定义拷贝构造函数，则值参传递也通过浅拷贝构造实现（调用编译器提供的默认拷贝构造函数）。

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。



赋值与调用

```
#include <string.h>
class STRING{
   char *s;
public:
   virtual  char  &operator[ ] (int x) { return s[x]; } //[]重载要返回非const引用
   STRING (const char*c) {strcpy (s=new char[strlen(c)+1], c) ; }
   STRING (const STRING &c)  //拷贝构造函数
   {strcpy (s=new char[strlen (c.s) +1], c.s); }
   virtual STRING operator+(const STRING&)const; //不改参数
   virtual STRING &operator= (const STRING &);
   //<<必须用全局函数重载，声明为友元是为了方便访问私有成员
    friend ostream& operator<<(ostream &os, const STRING &s);
   virtual STRING &operator+=(const STRING&s)
   {return *this=*this+s; }   //调用重载的operator+, =
   virtual ~STRING ( ) { if (s)  { delete [ ]s;  s=0;  }}
} s1( “S1” ),  s2= “S2” , s3( “S3” );
```

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 & rhs){
    char *t = new char[strlen(rhs.s) + 1]; strcpy(t,rhs.s);
    delete[] this->s; //一定要在char *t指向的内存分配成功后再delete[] this->s
    this->s = t;
    return *this;
}
ostream& operator<<(ostream &os, const STRING &s){
    os << s.s;  return os;
}
void main (void){
    s1=s1+s2;
    s1+=s3;
    s3[0]='T';          //调用char &operator[ ] (int x)
}
```

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；
(8) 对于形如“T a; T&& f( );”的定义，不要使用“T && b=f( );”之类的声明和“a=f( );”
(9) 不要随便使用exit和abort退出程序。
(10) 最好使用异常处理机制。
(11) 注意delete p和delete [ ]p

对于形如“T a; T&& f( );”的定义，不要使用“T && b=f( );和“a=f( );”之类的声明

```
class A {
private:
   int size;  int *p;
public:
   A():size(1),p(new int[size]){cout << "Default Constructor, size = " << size << endl;}
   A(int s):size(s > 0?s:1),p(new int[size]){cout << "Constructor,size = " << size << endl;}
   virtual ~A() {
      cout << "Deconstructor,size = " << size << endl; if (p){delete[] p; p = 0;size = 0;}
   }
   int get_size() { return size; };
   A(const A &old):size(old.size),p(new int[size]) { //拷贝构造
      for (int i = 0; i < size; i++) {p[i] = old.p[i];} cout << "Copy constructor,size = " << size << endl;
   }
   A(A &&old):size(old.size),p(old.p) { //移动构造
      old.p = 0; old.size = 0; cout << "Move constructor,size = " << size << endl;
   }
   A &operator=(A &&rhs){
      if(this == &rhs) return *this;
      int *t = this->p; this->p = rhs.p;  this->size = rhs.size; rhs.p = 0;  rhs.size = 0;
      if(t) delete t;
      cout << "Move =,size = " << size << endl;
   }
};
```

对于形如“T a; T&&f( );”的定义，不要使用“T &&b=f( );和“a=f( );”之类的声明

```
//返回右值引用的函数要非常小心，特别是引用了即将出栈的临时对象
A&& f() { return A(100); }
A&& g() { return A(200); }
int main() {
   A a;
//   A && rr1 = f(); //右值引用引用了一个已经出栈的对象
//   cout << rr1.get_size() << endl; //会输出不确定值，因为局部对象出栈
//   A && rr2 = g();
//   cout << rr2.get_size() << endl;      //会输出不确定值，因为局部对象出栈

   A c = A(100);      //在C++11下，调用移动构造；但是在C++17下，被编译器优化为调用构造函数A(int)，等价于A c(100)；
   A d = a;           //调用拷贝构造

   //f()返回右值引用，然后用该右值引用引用的对象去构造对象b
   // 应该调用移动构造，但是该右值引用引用的对象已经在f()返回后出栈，因此出错
//   A b = f();

   //A &r = f(); //编译错误，f()返回的右值

   //和A b = f();存在同样的问题： f()返回右值引用，然后将该右值引用引用的对象移动赋值给a
   //应该调用移动赋值，但是但是该右值引用引用的对象已经在f()返回后出栈，因此出错
   a = f(); return 0;
}
```

第11章 运算符重载

◆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。隐含参数this匹配调用当前函数的对象，若用const、volatile说明this指向的对象，则匹配的是const、volatile对象。
- 如定义了合适的**类型转换函数重载**，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，实在太多）：

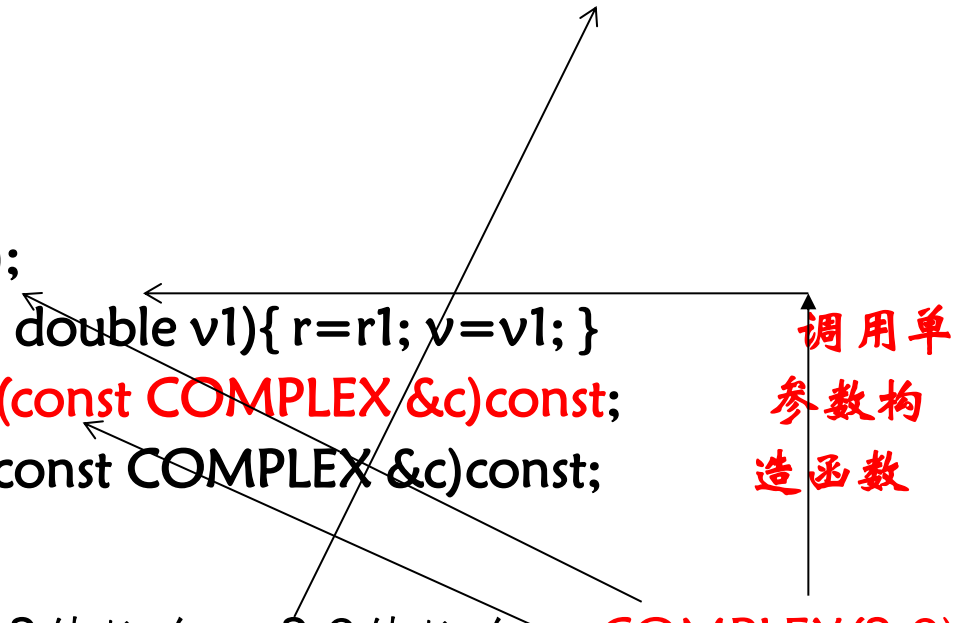
```
class COMPLEX{
    double r, v;
public:
    COMPLEX(double r1, double v1);
    COMPLEX operator+(const COMPLEX &c)const;
    COMPLEX operator+(double d)const;
    COMPLEX operator+(int d)const;
    COMPLEX operator-(const COMPLEX &c)const;
    COMPLEX operator-(double d)const;
    COMPLEX operator-(int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

如果为了满足交换律，还得定义更多版本的重载函数

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;
};
```



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

强制类型转换运算符重载

```
struct A{
   int  i;
   A (int v)   { i=v; }
   //C style casting:  double d; int i = (int) d;
   //C++ style casting:  double d; int i = int(d);
   //重载类型强制转换函数：operator targetType()，不需要定义返回类型
   //( ) 只能用普通成员函数重载
   virtual operator int ( ) const{ return  i; } //类型转换返回右值(int)
}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) ; //a强制转换int，b强制转换int，再调B的构造函数
void  main (void)  {
   int  i=1+ int (a); //强制转换，调用A::operator int ( )转换a, i=6
   i=b+3;          //自动转换，调用B::operator int ( )转换b, i=19
   i=a=b;          //调用1:B::operator A ( )；2：调用默认=重载，3：A::operator int ( ), i=16
}
```

第11章 运算符重载

◆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。
- 也可以用operator定义强制类型转换函数。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。
- 不应该同时定义T::operator A()和A::A (const T&)，否则容易出现二义性错误。
- 按照C++约定，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)。
- 转换的目标类型只要是能作为函数的返回类型即可。函数不能返回数组和函数。因此C++规定转换的类型表达式不能是数组和函数，但可以是指向数组和函数的指针

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，也可重载为静态函数成员。