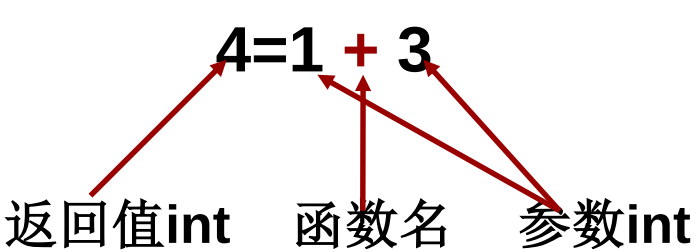
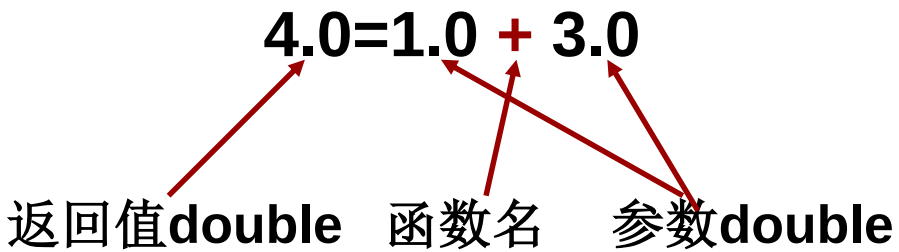
第4章：运算符重载

计算机系 江志英 jiangzy@mail.buct.edu.cn

引言

运算符和函数





int + (int, int);

double + (double, double);

可以将+代表的操作看成一个函数：+为函数的名字，+操作的两个操作数的类型即为函数形参的类型，+操作的结果的类型即为函数的返回值类型。

运算符重载的提出

C/C++中，每种基本数据类型的变量都可通过几种运算符来做相关计算，如int型变量，它们可以同“+”，“-”，“*”，“/”等运算符来做加法，减法，乘法，除法等。如：

```
int a, b;
a+b;   a-b;   a*b;   a/b;
```

我们能否将自定义的类的对象也通过“+”，“-”，“*”，“/”等运算符来进行两个对象的加，减，乘，除呢？如：

```
class Vector{
    int *  data;
    int     num;
    public: … …
}
```

定义一个向量类

```
Vector   vec1,vec2
vec1+vec2;   vec1-vec2;   vec1*vec2;   vec1/vec2;
```

两个向量类对象的加减乘除

C++提供了运算符重载机制来帮助我们实现上述目的。

计算机系 江志英 jiangzy@mail.buct.edu.cn

本章内容

1. 运算符重载的基本语法
2. 常用运算符的重载
3. 重载赋值运算符operator=
4. 重载输入/输出运算符 <<｜>>
5. 重载下标运算符operator[]
6. 重载类型转换函数

计算机系 江志英 jiangzy@mail.buct.edu.cn

运算符重载的基本语法

运算符的使用和普通函数在语法上有所不同，但是可以将运算符看作是一种特殊的函数，操作数是函数的参数，运算结果是函数的返回值。

其实在我们知道“运算符重载”的概念之前，已经在重载运算符了，例如：

- int a = 1
- double d = 2.3;
- a + d;
- a + 1;
- d +1.5;

在C++ 中，为一个类类型定义运算符和普通函数的定义形式很相似，只是函数的名字是关键字operator后紧跟要重载的运算符。例如，重载的“+”运算符函数名字为operator+ 。定义了运算符函数之后，就可以对类类型的操作数使用该运算符。

(1) 运算符函数

定义重载运算符和定义普通函数类似，只是该函数的名字是opeartor@，@表示要重载的运算符。

函数的参数个数取决于两个因素：

- 运算符的操作数个数：
  - 一元运算符
  - 二元运算符
- 运算符函数：
  - 成员函数
  - 全局函数

将运算符函数定义为成员函数时，调用成员函数的对象（this指向的对象）被作为运算符的第一个操作数，所以如果是一元运算符，无需提供参数。使用成员函数重载二元运算符时，将当前对象（this指向的对象）作为左操作数，需要提供一个参数作为右操作数。

如果将运算符函数定义为全局函数，则通常要将其声明为类的友元函数。重载一元运算符时需要提供一个类类型的参数，重载二元运算符时需要提供两个参数，分别作为左右操作数，其中至少一个参数必须是类类型的。


```
#include <iostream.h>
class MinInt {
 char b;
 public:
      MinInt(char ch = 0) : b(ch) {}
      MinInt operator-( )const{  //一元运算"-"
          cout<<"MinInt::operator-"<<endl;
          return MinInt(-b);
      }
      MinInt operator+(const MinInt& rv) const {//二元运算 +
          cout << "MinInt::operator+" << endl;
          return MinInt(b + rv.b);
      }
      MinInt&  operator+=(const MinInt& rv){//复合赋值运算 +=
          cout << "MinInt::operator+=" << endl;
          b += rv.b;
          return *this;
      }
}; //end of class MinInt
```

```
int main() {
    cout << "内置数据类型:" << endl;
    int v1 = 1, v2 = 2, v3 = 3;
    v3 += v1 + v2;
    cout << "用户自定义类型:" << endl;
    MinInt b1(10), b2(20), b3(30);
    b3 += -b1 + b2; //调用重载的运算符
}
```

程序的输出结果为：

```
内置数据类型:
用户自定义类型:
MinInt::operator-
MinInt::operator+
MinInt::operator+=
```

(2) 运算符重载的限制

- 只有C++预定义运算符集合中的运算符才可以重载
- C++中有些运算符不能被重载：::（作用域解析符）、.（成员选择符）、.*（成员指针间接引用符）及?:（条件运算符）。
- 不能定义C++中没有的运算符，如operator**会产生编译错误。
- 内置类型的运算符的预定义意义不能改变，也不能为内置数据类型定义其他运算符。例如，不能定义内置数组的operator+。
- 重载运算符不能改变运算符的优先级和结合性。
- 重载运算符不能改变操作数的个数。

可以重载的运算符

+	—	*	/	%	^	&
|	~	!	,	=	<	>
<=	>=	++	——	<<	>>	==
!=	&&	||	+=	—=	/=	*=
%=	^=	&=	|=	>>=	<<=	[]
( )	—>	—>*	new	delete	new[]	delete[]

计算机系 江志英 jiangzy@mail.buct.edu.cn

2 常用运算符的重载

本节给出一些常规运算符以成员函数和友元函数两种方式重载的例子。在下面的代码中，我们分别以两个类Integer和Byte 来举例说明如何使用成员函数和友元函数重载常用的运算符。

(1) 一元运算符

```
class Byte {
    unsigned char b;
public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // 无副作用的运算符定义为 const成员函数
    const Byte& operator+( ) const { //正号
        return *this;}
    const Byte operator-( ) const { //负号
        return Byte(-b);}
    const Byte operator~( ) const { //按位取反
        return Byte(~b);}
    Byte operator!( ) const {//逻辑非
        return Byte(!b);}
    Byte* operator&( ) {//取地址
        return this;}
```

一元运算符函数是成员函数时，形式参数列表里没有参数。

计算机系 江志英 jiangzy@mail.buct.edu.cn

```
// 有副作用的运算符定义为非const成员函数
    const Byte& operator++() { // 前缀++
        b++;
        return *this;}
    const Byte operator++(int) { // 后缀++
        Byte before(b);
        b++;
        return before;}
    const Byte& operator--() { // 前缀--
        --b;
        return *this;}
    const Byte operator--(int) { // 后缀--
        Byte before(b);
        --b;
        return before;}
};
```

int参数是用来区分前缀和后缀的；调用时系统传0；去掉就变成前缀形式

前缀形式返回改变后的对象，返回*this。

后缀形式返回改变之前的值，所以必须创建一个代表这个值的独立对象并返回它,是通过传值方式返回的。


```
// 重载运算符的使用示例
int main() {
        Byte a, b, *bp;
        +b;
        -b;
        ~b;
        bp = &b;
        !b;
        ++b;
        b++;
        --b;
        b--;
} //end of main()
```

用全局友元函数重载一元运算符

```
class Integer {
    long i;
    Integer* This( ) { return this; }
public:
    Integer(long ll = 0) : i(ll) {}
    friend const Integer&  operator+(const Integer& a);
    friend const Integer   operator-(const Integer& a);
    friend const Integer   operator~(const Integer& a);
    friend Integer*   operator&(const Integer& a);
    friend bool operator!(const Integer& a);
    friend const Integer&   operator++(Integer& a);  // 前缀++
    friend const Integer   operator++(Integer& a, int);//后缀++
    friend const Integer&  operator--(Integer& a);  // 前缀++
    friend const Integer   operator--(Integer& a, int);//后缀++ };
```


计算机系 江志英 jiangzy@mail.buct.edu.cn

```
// 全局运算符函数的定义
const Integer& operator+(const Integer& a) {
    return a;
}
const Integer operator-(const Integer& a) {
    return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
    return Integer(~a.i);
}
Integer* operator&(Integer& a) {
    return a.This( ); // 不能用&a，会导致递归调用本函数
}
bool operator!(const Integer& a) {
    return !a.i;
}
```


```
const Integer& operator++(Integer& a) { // 前缀
    a.i++;
    return a;
}
const Integer operator++(Integer& a, int) { // 后缀
    Integer before(a.i);
    a.i++;
    return before;
}
const Integer& operator--(Integer& a) {
    a.i--;
    return a;
}
const Integer operator--(Integer& a, int) {
    Integer before(a.i);
    a.i--;
    return before;
}
```


计算机系 江志英 jiangzy@mail.buct.edu.cn


```
// 重载运算符的使用
int main() {
    Integer a,  * ip ;
    +a;
    -a;
    ~a;
    ip = &a;
    !a;
    ++a;
    a++;
    - -a;
    a- -;
}//end of main
```

自增和自减运算符

自增和自减运算符有前缀和后缀两种形式，都会改变对象，所以不能对常量对象操作。

- 前缀形式返回改变后的对象，返回*this。
- 后缀形式返回改变之前的值，所以必须创建一个代表这个值的独立对象并返回它,是通过传值方式返回的。

如何区分前缀和后缀形式呢?

- 后缀形式的自增和自减比前缀形式多一个int参数，这个参数在函数中并不使用，只是作为重载函数的标记来区分前缀和后缀运算。例如，对Byte类对象b，编译器看到++b，会调用Byte::opeartor++()， 而b++会调用Byte::operator++(int)。

如果要重载自增和自减运算符，一般应同时定义自增、自减运算符的前缀式和后缀式。重载运算符的行为应尽量与内置运算符保持一致。

计算机系 江志英 jiangzy@mail.buct.edu.cn

(2) 二元运算符

用成员函数重载二元运算符

```
class Byte {
  unsigned char b;
public:
    Byte(unsigned char bb = 0) : b(bb) {}
     const Byte operator+(const Byte& right) const { return Byte(b +
    right.b);}
    const Byte operator-(const Byte& right) const { return Byte(b - right.b);}
    const Byte operator*(const Byte& right) const { return Byte(b * right.b);}
    const Byte operator/(const Byte& right) const {
        assert(right.b != 0);
        return Byte(b / right.b);}
    const Byte operator%(const Byte& right) const {
        assert(right.b != 0);
        return Byte(b % right.b);}
```


```
//位运算符 ^,&,|,<<,>>
const Byte operator^(const Byte& right) const {
        return Byte(b ^ right.b);
}
const Byte operator&(const Byte& right) const {
        return Byte(b & right.b);
}
const Byte operator|(const Byte& right) const {
        return Byte(b | right.b);
}
const Byte operator<<(const Byte& right) const {
        return Byte(b << right.b);
}
const Byte operator>>(const Byte& right) const {
        return Byte(b >> right.b);
}
```

```
Byte& operator=(const Byte& right) { // 只能用成员函数重载
    if(this == &right) return *this; // 自赋值检测
    b = right.b;
    return *this; }
//复合赋值运算符有：+=,-=,*=,/=,%=,^=,&=,|=,<<=,>>=
 Byte& operator+=(const Byte& right) {
  b += right.b;
  return *this;
 }
 Byte& operator/=(const Byte& right) {
  assert(right.b != 0);
  b /= right.b;
  return *this;
 }
 Byte& operator^=(const Byte& right) {
  b ^= right.b;
  return *this;
 }
```


```
//关系运算符有 ==, !=, <,<=,>,>=
 bool operator==(const Byte& right) const {
     return b == right.b;
 }
 bool operator!=(const Byte& right) const {
     return b != right.b;
 }
 bool operator<(const Byte& right) const {
     return b < right.b;
 }
//二元逻辑运算符&&和||
 bool operator&&(const Byte& right) const {
     return b && right.b;
 }
 bool operator||(const Byte& right) const {
     return b || right.b;
 }
}; //end of class Byte
```

用全局函数重载二元运算符

全局函数重载二元运算符时，要带两个参数，其中至少有一个是类类型的。

第一个参数作为左操作数；第二个参数是右操作数。

注意，赋值运算符operator= 只能用成员函数重载。

计算机系 江志英 jiangzy@mail.buct.edu.cn


```
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  friend const Integer operator+(const Integer& left, const Integer& right);
  friend const Integer operator-(const Integer& left, const Integer& right);
  friend const Integer operator*(const Integer& left, const Integer& right);
  friend const Integer operator/(const Integer& left, const Integer& right);
  friend const Integer operator%(const Integer& left, const Integer& right);

  friend const Integer operator^(const Integer& left, const Integer& right);
  friend const Integer operator&(const Integer& left, const Integer& right);
  friend const Integer operator|(const Integer& left, const Integer& right);
  friend const Integer operator<<(const Integer& left,const Integer& right);
  friend const Integer operator>>(const Integer& left,const Integer& right);
```

```
// 修改并返回左值的复合赋值运算符，第一个参数是非const引用，即左值
    friend Integer& operator+=(Integer& left, const Integer& right);
    friend Integer& operator-=(Integer& left, const Integer& right);
    friend Integer& operator*=(Integer& left, const Integer& right);
    friend Integer& operator/=(Integer& left, const Integer& right);
    friend Integer& operator%=(Integer& left, const Integer& right);
    friend Integer& operator^=(Integer& left, const Integer& right);
    friend Integer& operator&=(Integer& left, const Integer& right);
    friend Integer& operator|=(Integer& left, const Integer& right);
    friend Integer& operator>>=(Integer& left, const Integer& right);
    friend Integer& operator<<=(Integer& left, const Integer& right);
    // 逻辑运算符和关系运算符返回bool值，不改变操作数
    friend bool operator==(const Integer& left, const Integer& right);
    friend bool operator!=(const Integer& left, const Integer& right);
    friend bool operator<(const Integer& left, const Integer& right);
    friend bool operator>(const Integer& left, const Integer& right);
    friend bool operator<=(const Integer& left, const Integer& right);
    friend bool operator>=(const Integer& left, const Integer& right);
    friend bool operator&&(const Integer& left, const Integer& right);
    friend bool operator||(const Integer& left, const Integer& right);
};
```

```
//算术运算符，只给出+和/的定义，其余实现方式类似，略
const Integer operator+(const Integer& left,  const Integer& right) {
    return Integer(left.i + right.i);
}
const Integer operator/(const Integer& left,  const Integer& right) {
    assert(right.i != 0);
    return Integer(left.i / right.i);
}
//位运算符，只给出了&和<<的实现，其余实现方式类似，略
const Integer operator&(const Integer& left,  const Integer& right) {
    return Integer(left.i & right.i);
}
const Integer operator<<(const Integer& left,  const Integer& right) {
    return Integer(left.i << right.i);
}
```


```
// 复合赋值运算符，此处只给出了+=, /=和&=的实现，其余实现类似，略
Integer& operator+=(Integer& left, const Integer& right) {
    if(&left == &right) {/* self-assignment */}
    left.i += right.i;
    return left;
}
Integer& operator/=(Integer& left, const Integer& right) {
    assert(right.i != 0);
    left.i /= right.i;
    return left;
}
Integer& operator&=(Integer& left, const Integer& right) {
    left.i &= right.i;
    return left;
}
```

// 关系运算符和逻辑运算符，此处给出了==和&&的实现，其余实现类似，略

```
bool operator==(const Integer& left,  const Integer& right) {
    return left.i == right.i;
}

bool operator&&(const Integer& left,  const Integer& right) {
    return left.i && right.i;
}
```

(3) 运算符函数参数/返回类型

各种不同的参数传递和返回方法，选择要合 乎逻辑：

- 对于任何参数类型，如果仅仅只是读参数的值，而不改变参数，应该作为const引用来传递。普通算术运算符、关系运算符、逻辑运算符都不会改变参数，所以以const引用作为主要的参数传递方式。当运算符函数是类的成员函数时，就将其定义为const成员函数。
- 返回值的类型取决于运算符的具体含义。如果使用运算符的结果是产生一个新值，就需要产生一个作为返回值的新对象，这个对象作为一个常量通过传值方式返回。如果函数返回的是原有对象，则通常以引用方式返回，根据是否希望对返回的值进行运算来决定是否返回const引用。
- 所有赋值运算符均改变左值。为了使赋值结果能用于链式表达式，如a=b=c，应该返回一个改变了的左值的引用。一般赋值运算符的返回值是非const引用，以便能够对刚刚赋值的对象进行运算。
- 逻辑运算符和关系运算符最好返回bool值，也可返回int值或者由typedef定义的等价类型。

计算机系 江志英 jiangzy@mail.buct.edu.cn

返回值优化

通过传值方式返回创建的新对象时，使用一种特殊的语法，例如在operator+中：

```
return Integer(left.i + right.i);
```

这种形式看起来像是一个构造函数的调用，这称为临时对象语法，含义是创建一个临时的Integer对象并返回它。

这种方式和创建一个有名字的对象并返回它是否相同呢？例如：

```
Integer temp(left.i + right.i);
return temp;
```

这两条语句先创建temp对象，会调用构造函数。然后拷贝构造函数把temp拷贝到外部返回值的存储单元，最后在temp作用域的结尾时调用析构函数。

返回临时对象的方式不同与此。当编译器看到这种语法时，会明白创建这个对象的目的只是返回它，所以编译器直接把这个对象创建在外部返回值的存储单元中，所以只需要调用一次构造函数，不需要拷贝构造函数和析构函数的调用。因此，使用临时对象语法的效率非常高，这被称为返回值优化。

(4) 全局运算符和成员运算符

大部分运算符可以使用成员函数和全局函数两种方式重载。在这两种方式之间如何选择呢?

总的来说，如果没有什么差异，应该是成员运算符，因为这样强调了运算符和类的密切关系。但有时我们会因为另外一个原因选择使用全局函数重载运算符,如下例：

计算机系 江志英 jiangzy@mail.buct.edu.cn


```
class Number {
        int i;
public:
        Number(int ii = 0) : i(ii) {}
        const Number operator+(const Number& n) const { //成员
          return Number(i + n.i);}
       friend const Number operator-(const Number&, const Number&);
};
const Number operator-(const Number& n1, const Number& n2){ //友元
   return Number(n1.i - n2.i);
}
int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 1; // 右操作数转换为Number
  1 + a; // 错误：左操作数不是Number类型
  a - b; // OK
  a - 1; //右操作数转换为Number
  1 - a; //左操作数转换为Number
}
```

可以看到，使用成员运算符的限制是左操作数必须是当前类的对象，左操作数不能进行自动类型转换，而全局运算符为两个操作数都提供了转换的可能性。因此，如果左操作数是其他类的对象，或是希望运算符的两个操作数都能进行类型转换，则使用全局函数重载运算符。

类似的情况还出现在为I/O流重载运算符operator<<及operator>>时。

(5) 重载输入/输出运算符

operator>> 带两个操作数，左操作数cin是istream 类型的对象，而右操作数是接收输入数据的变量。输入操作会引起两个操作数的改变，因而，这两个参数需要传递非const引用。

operator>> 可以用于链式表达式，如 cin>>b>>c。这等价于： cin>>b; cin>>c;

由此看到这个运算应该返回istream对象cin本身，并且可以继续用于输入。

重载operator>>的基本形式如下：

istream& operator>> (istream&, type&);

重载operator<<的基本形式如下：

ostream& operator<< (ostream&, const type&);

重载operator<<和operator>>

```
#include <iostream.h>
class complex{
private:
    double real, image;
public:
    complex(double r = 0, double i = 0){
        real = r; image = i; }
    //成员函数重载运算符+
    const complex operator+(const complex& right)  const{
        return complex (real+right.real,image+right.image);  }
    //全局函数重载输入输出运算符
    friend ostream& operator<<(ostream& os, const complex& c);
    friend istream&  operator>>(istream& is,complex& c);
};
```

```
//输入输出运算符函数的定义
ostream& operator<<(ostream& os, const complex& c){
    if(c.real==0 && c.image==0){ os << "0"; }
    if(c.real!=0){ os << c.real; }
    if(c.image!=0){
        if(c.image>0 && c.real!=0)
            os << "+";
        os << c.image << "i" ;
    }
    return os; //返回ostream对象便于链式表达式
}
istream& operator>>(istream& is, complex& c){
    cout<<"please input a complex:";
    return is>>c.real>>c.image;
}
int main() {
    complex c1,c2;
    cin>>c1;
    cin>>c2;
    cout<<c1+c2<<endl; //调用全局函数operator<<
}
```

3 重载赋值运算符operator=

- 赋值运算符用来覆盖对象原有的值，是最常用的运算符之一，也是定义类时经常要重载的运算符。
- 与其它运算符相比，赋值运算符（=）的特别之处是：程序员没有定义赋值运算符函数，编译器会为生成一个默认的赋值运算符函数。
- 赋值和初始化不同。初始化是在创建新对象时进行，只能有一次；而赋值可以对已存在的左值多次使用。类类型的对象在初始化时调用构造函数，而赋值时调用operator=。
- 赋值运算符“=”可能用在初始化对象的地方，但是这种情况并不会引起operator=的调用。赋值运算符的左侧操作数是已经存在的对象时，才会调用operator=。


```
void f(){
    int m = 10;//初始化
    int n;  //定义变量，但没有初始化
    n = 5;//赋值，虽是首次赋值，但不是初始化
    n = m;  //赋值

    MyType b;  //调用缺省构造函数
    MyType a = b;//创建并初始化a，等价MyTypea(b);
    //调用拷贝构造函数，而不是operator=

    a = b;//a是已经存在的对象，调用operator=
}
```

赋值运算符必须作为成员函数重载，不能使用全局函数重载operator=。因为如果允许定义全局的operator=，那么可能会导致重定义内置的“=”，例如：

```
int operator= (int, MyType);//Error
```

赋值检测

operator= 的基本行为：

将右操作数中的信息拷贝到左操作数中。

对于简单的对象，这种行为是很显然的。

简单对象的赋值

```
#include<iostream.h>
class Value {
  int a, b;
  float c;
public:
      Value(int aa = 0, int bb = 0, float cc = 0.0) : a(aa), b(bb), c(cc) {}
      Value& operator=(const Value& rv) {
            a = rv.a;b = rv.b;c = rv.c;
            return *this; }
      friend ostream&  operator<<(ostream& os, const Value& rv) {
      //重载operator<<，将对象写到输出流
            os << "a = " << rv.a << ", b = " << rv.b << ", c = " << rv.c;
            return os; }
}; //end of Value
```


计算机系 江志英 jiangzy@mail.buct.edu.cn

```
int main() {
    Value a, b(1, 2, 3.3);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    a = b; //调用operator=
    cout << "a after assignment: " << a << endl;
}
```

这个例子中定义了赋值运算符，但是隐含了一个常见的错误。在对象赋值之前应该进行自赋值检测：检验对象是否在给自身赋值。在简单的情况下，即使对象给自身赋值也没有什么危害，例如本例。但是有些情况下，对象给自身赋值会导致严重后果。


```
//一个简单的字符串类
class my_string{
    char* str;
    int len;
public:
    my_string(const char* s = ""){
        len = strlen(s);
        str = new char[len + 1];
        strcpy(str, s);
    }
    ~my_string(){delete[]str;}
    //…其他构造函数和成员函数略
    my_string& operator=(const my_string& s);
};

my_string a("abcde"), b("hijk");
a = b;   //如何赋值?
```

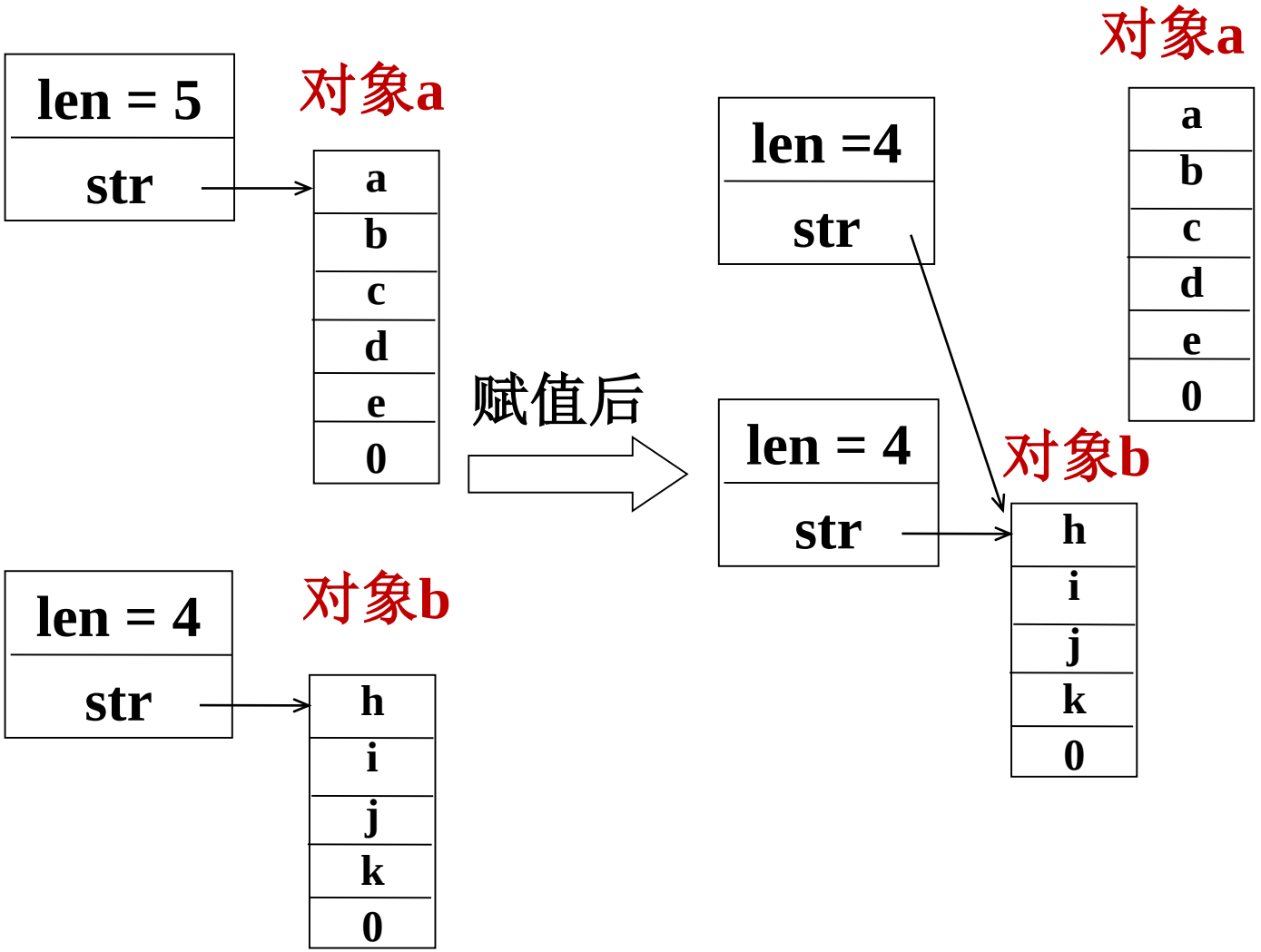
这个类中operator=如何实现呢?

第一种方式是进行成员之间的直接赋值，代码如下:

```
my_string& my_string::operator= (const my_string& s) {
        len = s.len;
        str = s.str;
        return *this;
}
```

这段代码中的直接赋值会导致什么后果呢?

直接赋值会导致什么后果?



这样的赋值行为导致了两个问题：

① 原来的a.str指向的动态存储空间没有释放，造成内存泄漏。

② a和b的str指向了同一段存储空间，破坏了对象的完整性，即对a和b中一个字符串操作，另一个将受到影响。

那么下面修改后的代码是否正确呢?

```
my_string& my_string::operator= (const my_string& s) {
    //先释放当前对象中的动态存储空间
    delete[] str;
    //再重新分配空间
    len = s.len;
    str = new char[len + 1];
    //最后进行字符串的拷贝
    strcpy(str, s.str);
    return *this;
}
```

修改后的代码是否正确呢?

可以看到，对上面例子中的a和b，这个operator= 能够产生正确的后果。但是对下面的赋值语句会产生什么后果呢?

```
a=a;
```

这个赋值操作先释放左操作数（a）的str指针，但右操作数（a）的指针同时也被释放了。这当然不是我们希望的结果。

造成这种后果的原因是没有在operator=中进行自赋值检测。通过比较两个对象的内存地址是否相同可以判断它们是否为同一个对象。

赋值之前先进行自赋值检测

```
my_string& my_string::operator= (const my_string& s) {
    //赋值之前先进行自赋值检测
    if(this == &s)  return *this;
    delete[] str;
    len = s.len;
    str = new char[len + 1];
    strcpy(str, s.str);
    return *this;
}
```

所有的赋值运算符在重载时都要进行自赋值检测。虽然在某些情况下，这并不是必需的，但是我们最好养成好习惯，以防止在操作复杂对象时可能引起的错误。

4 重载下标运算符operator[]

有些类的对象可以像数组一样操作，可以为这样的类提供下标运算符：

- 下标运算符operator[]必须是成员函数，它只接收一个参数，通常是整值类型。
- 下标运算符作用的对象应该能像数组一样操作，所以经常用该运算符返回一个元素的引用，以便用作左值。

为动态数组类vect重载下标运算符。

```
class vect {
public:
    //构造函数和析构函数
    explicit vect(int n = 10);
    vect(const vect& v);
    vect(const int a[], int n);
    ~vect() { delete []p; }
    //其他成员函数
    int& operator[](int i); // 重载下标运算
    int ub() const { return (size - 1); }
    vect& operator=(const vect& v);
private:
    int* p;
    int  size;
};
```


```
//成员函数类外定义
vect::vect(int n) : size(n){
      assert(size > 0);
      p = new int[size];
 }
vect::vect(const int a[], int n) : size(n){
      assert(size > 0);
      p = new int[size];
      for(int i=0; i<size; ++i)
            p[i] = a[i];
}
vect::vect(const vect& v) : size(v.size){
      p = new int[size];
      for(int i=0; i<size; ++i)
            p[i] = v.p[i];
}
```

```
int& vect::operator[](int i){
    assert( i>=0 && i<size);
    return p[i];//返回的是左值
}
vect& vect::operator=(const vect& v){
    if (this != &v) {
        assert(v.size == size);//只允许相同大小的数组赋值
        for(int i =0; i<size; ++i)
            p[i] = v.p[i];
    }
    return *this;
}
int main() { //测试程序
    int a[5]={1,2,3,4,5};
    vect v1(a,5);
    v1[2]=9;//调用operator[]
    for(int i=0; i<=v1.ub(); ++i)
        cout<<v1[i]<<"\t";
}
```

5 用户定义的类型转换

如果在表达式中使用了类型不合适的操作数，编译器会尝试执行自动类型转换，从当前类型转换到要求的类型。除了内置类型之间的标准提升和转换之外，用户也可以定义自己的类型转换函数。

带单个参数的构造函数提供了参数类型的对象到本类型对象的转换。那么如何将本类型的对象转换为其他类型呢?

(1) 类型转换运算符

- 重载operator type运算符可以将当前类型转换为type指定的类型。
- 这个运算符只能用成员函数重载，而且不带参数，它对当前对象实施类型转换操作，产生type类型的新对象。
- 不必指定operator type函数的返回类型，返回类型就是type。

例如，要定义一个MinInt类型，表示100以内的非负整数，并且要求能够对它像int类型一样进行操作。见下页

```
#include <iostream.h>
#include <cassert>
class MinInt{
    char m;
public:
    MinInt(int val = 0){ //int类型转换为MinInt
        assert(val>=0 && val<=100); //要求取值范围在0～100之间
        m = static_cast<char>(val);
    }
    operator int( ){//MinInt对象转换为int 类型
        return static_cast<int>(m); }
};
int main( )  {
    MinInt mi(10), num;
    num = mi + 20; /*首先将mi转换为int类型，再执行加法运算;
        再将int类型的计算结果30 转换为赋值左边的MinInt 类型 */
    int val = num;  //将num 自动转换为int，并赋值给val
    cout<< mi << '\t' << num << '\t' << val;
    //num 和mi 转换为int输出
}
```


```
// 用户自定义的类型转换
class One {
public:
    One() {}
};

class Two {
public:
    Two(const One&) {} //类型转换构造函数 One => Two
};

class Three {
    int i;
public:
    Three(int ii = 0, int = 0) : i(ii) {}
    //int => Three
};
```


```
class Four {
    int x;
public:
    Four(int xx) : x(xx) {} // int => Four
    operator Three() const  //Four => Three
    { return Three(x); }
};
void f(Two) {}
void g(Three) {}
int main() {
    One one;
    f(one); // OK:自动类型转换Two(const One&)
    Four four(1);
    g(four);//OK: 自动类型转换operator Three()
    g(1);  // Three(1,0)
}
```

(2) 可能引起的二义性问题

自动类型转换应小心使用，它在减少代码方面有很大作用，但不当使用也会引起一些麻烦。

如果程序中定义了多种用于从X类到Y类自动转换的方法，那么在实际需要进行类型转换时将会产生二义性错误。


```
class Y; // 类声明
class X {
public:
    operator Y() const; // X到Y的转换
};
class Y {
public:
    Y(X); // X到Y的转换
};
void f(Y) {}
int main() {
    X a;
    f(a); // Error: 二义性
}
```

如果程序中定义了从一种类型到其他多种类型的自动转换方法，那么在实际需要进行类型转换时也可能导致二义性。

```
class Y{ };
class Z{ };
class X {
public:
    operator Y() const; // X到Y的转换
    operator Z() const; // X到Z的转换
};
void f(Y) {}
void f(Z) {}
int main() {
    X a;
    f(a); // Error: 二义性
}
```

因此应该在确保不引起二义性，并能够优化代码的情况下谨慎使用自动类型转换。要避免上面的二义性问题，最好的办法是保证最多只有一种途径将一个类型转换为另一类型。

另外还有一些特殊运算符的重载，如operator,、operator->、operator->*、operator new及operator delete等，由于这些运算符的重载较少用到，并且使用不当容易造成混淆，因此不再介绍。

6 小结

- 运算符重载可以像基本数据类型一样，用简洁明确的运算符操作自定义的类对象。
- 重载运算符函数可以对运算符作出新的解释，但重原有的基本语义不变。
- 运算符函数既可以重载为成员函数，也可以重载义为友元函数或普通函数。
- 当单目运算符的操作数，或者双目运算符的左操作数是该类的一个对象时，以成员函数重载；当一个运算符的操作需要修改类对象状态时，应该以成员函数重载。如果以成友元函数重载，可以使用引用参数修改对象。
- 当运算符的操作数（尤其是第一个操作数）希望有隐式转换，则重载算符时必须用友元函数。
- 构造函数和类型转换函数可以实现基本类型与类类型，以及类类型之间的类型转换。.

运算符函数可以是成员函数，也可以是全局函数，一般的使用原则：

运算符	建议重载方式
一元运算符	成员函数
= [] () -> ->* 类型转换	必须是成员函数
复合赋值运算符	成员函数
其他二元运算符	非成员函数
输入输出运算符<<和>>	非成员函数