C++编程(9)

Tang Xiaosheng
北京邮电大学电信工程学院

第11章 运算符重载

- 引言
- 运算符函数
- 一个复数类型
- 转换运算符
- 友元
- 大型对象
- 基本运算符
- 下标
- 函数调用
- 间接
- 增量和减量
- 一个字符串类
- 忠告

11.1 引言

- 每个技术领域都发展了一套习惯性的简化记述形式，以便使涉及到常用概念的说明和讨论更加方便，例如：x+y*z
- 对于C++，已经为内部类型提供了一组运算符
- C++的用户希望也能在自己创建的新的类型(类)上也能使用这类运算符
- C++中解决类似问题的技术叫做运算符重载

示例

```
class complex {
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) {}
    complex operator+(complex);
    complex operator*(complex);
};
void f() {
    complex a = complex(1,3.1);
    complex b = complex(1.2,2);
    complex c = b;
    a = b+c;
    b = b+c*a; // 运算符优先级仍然有效
    c = a*complex(1,2);
}
```

11.2 运算符函数

以下的运算符可以被重载(注意，不允许定义新的运算符)						
+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]
以下操作符不能被重载						
::	.	.*	这些操作符以名字(不是值)来做为第二个参数			
? :	sizeof	typeid				

11.2.1 二元和一元运算符

- 二元运算符可以定义为取一个参数的非静态成员函数，也可以定义为取两个参数的非成员函数
- 对于任何二元运算符@，aa@bb可以解释为aa.operator@(bb)，或者解释为operator@(aa,bb)，如果两者都有定义，那么按照重载解析来确定究竟应该用哪个定义

二元和一元运算符

- 对于一元运算符，无论它是前缀的还是后缀的，都可以定义为无参数的非静态成员函数，或者定义为取一个参数的非成员函数
- 对任何前缀一元运算符@，@aa可以解释为aa.operator@()或者operator@(aa)
- 对任何后缀一元运算符@，aa@可以解释为aa.operator@(int)或者operator@(aa,int)

二元和一元运算符示例

```
class X{
public:
   void operator+(int);
   X(int);
};

void operator+(X,X);
void operator+(X,double);

void f(X a)
{
   a+1;  // a.operator+(1)
   1+a;  // ::operator+(X(1),a)
   a+1.0;// ::operator+(a,1.0)
}
```

```
class X{ // 隐含有this指针
   X* operator&(); // 前缀一元&取地址
   X operator&(X); // 二元& 与
   X operator++(int);// 后缀增量
   X operator&(X,X);// 错误：三元
   X operator/();  // 错误：一元
};

X operator-(X); // 前缀一元减
X operator-(X,X); // 二元减
X operator--(X&,int); // 后缀减量
X operator-(); // 错误：无操作数
X operator-(X,X,X); // 错误：三元
X operator%(X); // 错误：一元%
```

11.2.2 运算符的预定义意义

- 对于用户定义运算符只做了不多的几个假定，特别是operator=、operator[]和operator->只能做为非静态的成员函数，这就保证了它们的第一个运算对象一定是一个左值
- 有些内部运算符在意义上等价于针对同样参数的另外一些运算符的组合，但是对于用户定义运算符而言就没有这类关系，除非用户正好将它们定义成这样，例如：+=

11.2.3 运算符和用户定义类型

- 一个运算符函数必须或者是一个成员函数，或者至少有一个用户定义类型的参数(重新定义运算符new和delete的函数则没有此要求)，这一规则保证了用户不能改变原有表达式的意义，除非表达式中包含有用户定义类型的对象
- 如果某个运算符函数想接受某个内部类型做为第一参数，那么它自然就不可能是成员函数了，例如：2+aa，可以用非成员函数方便地处理这类问题
- 为枚举类型也可以定义运算符

运算符和用户定义类型示例

```
enum Day { sun, mon, tue, wed, thu, fri, sat };
Day operator++(Day& d)
{
   return d = (sat==d) ? sun: Day(d+1);
}
```

11.2.4 名字空间里的运算符

- 一个运算符或者是一个成员函数，或者是定义在某个名字空间里(也可以是全局名字空间)
- 注意：运算符查寻机制并不认为成员函数比非成员函数更应优先选取
- 此外，也不存在运算符屏蔽的问题，这保证了内部运算符绝对不会变得无法使用，也保证了用户可以为运算符提供新的意义，而不必去修改现存的类声明

名字空间里的运算符示例

```
namespace std{
    class ostream
    {
        ostream& operator<<(const
char*);
    };
    extern ostream cout;
    class string
    {
        // ...
    };
    ostream& operator<<(ostream&,
const string&);
}
```

```
int main()
{
    char* p = "Hello";
    std::string s = "world";
    std::cout << p << ", "
<< s << "!\n";
}
// 本例没有使用
// using namespace std;
// 而是使用了std::string和
// std::cout
// 按照最好的方式行事，没
// 有污染全局名字空间，也
// 没有引进不必要的依赖性
```

二元运算符的解析方式

- 考虑二元运算符@，如果x的类型为X而y的类型为Y，x@y将按照如下方式解析
  - 若X是类，查询作为X的成员函数或者X的某个基类的成员函数的operator@
  - 在围绕x@y的环境中查询operator@的声明
  - 若X在名字空间N里定义，在N里查询operator@的声明
  - 若Y在名字空间M里定义，在M里查询operator@的声明
- 当查询到operator@重载时，按照重载规则处理
- 一元运算符的解析方式与之类似

11.3 一个复数类型

- 我们希望实现一个复数类，可以进行如下操作
- 此外，还希望提供一些运算符，例如以==做比较，<<做输出等

```
void f()
{
    complex a=complex(1,2);
    complex b=3;
    complex c=a+2.3;
    complex d=2+b;
    complex e=-b-c;
    b=c*2*c;
}
```

11.3.1 成员运算符和非成员符

- 作者喜欢让尽量少的函数能直接去操作对象的表示，因此，作者只在类本身中定义那些本质上需要修改第一个参数值的运算符，如+=，而像+这样基于参数的值简单产生新值的运算符，则在类之外利用基本运算符来定义

成员运算符和非成员符示例

```
class complex
{
    double re, im;
public:
    complex&
    operator+=(complex a);
};

complex operator+
  (complex a, complex b)
{
    complex r=a;
    return r+=b;
}
```

```
complex& complex::operator+=
(complex a) {
    re += a.re; im += a.im;
    return *this;
}
void f(complex x, complex y,
       complex z) {
    complex r1=x+y+z;
    // r1 = operator+
    //      (operator+(x,y), z)
    complex r2=x;
    r2+=y;
    // r2.operator+=(y)
    r2+=z;}
```

11.3.2 混合模式算术

- 为了处理complex d=2+b; 我们需要能够接受不同类型的运算对象的+运算符，为此，可以简单地增加运算符的版本来解决这个问题

```
class complex
{
    double re, im;
public:
    complex& operator+=
        (complex a)
    {
        re += a.re; im += a.im;
        return *this;
    }
    complex& operator+=
        (double a);
    {
        re += a;
        return *this;
    }
};
```

混合模式算术示例

```
complex operator+(complex a, complex b)
{
    complex r=a;
    return r+=b; // 调用complex::operator+=(complex)
}
complex operator+(complex a, double b)
{
    complex r=a;
    return r+=b; // 调用complex::operator+=(double)
}
complex operator+(double a, complex b)
{
    complex r=b;
    return r+=a; // 调用complex::operator+=(double)
}
```

11.3.3 初始化

- 我们可以使用标量对complex变量进行初始化和赋值，为此，可以定义一个只有一个参数的构造函数来解决这个问题

```
class complex{
    double re, im;
public:
    complex(double r) : re(r), im(0) {}
    complex(): re(0), im(0){}
};

complex b = 3; // b.re = 3, b.im = 0 即complex(3)
complex c;
```

11.3.4 复制

- 除了显式的构造函数之外，complex还按照默认规定，得到了一个定义好的复制构造函数，默认的构造函数就是简单地复制成员，这对于comple类来说已经足够好

```
class complex{
    double re, im;
public:
    complex(const complex& c) : re(c.re), im(c.im) {}
};
```

11.3.5 构造函数和转换

- 前面的例子中，我们为每个标准的算术运算定义了三个版本

  complex operator+(complex, complex);
  complex operator+(complex, double);
  complex operator+(double, complex);

- 若函数中需要处理的每个参数有三种类型，那么一个参数的函数需要3个版本，两个参数的需要9个版本，三个参数就需要27个版本，而且这些不同版本的函数的代码实现会非常相似
- 依靠类型转换可以大大简化此类问题

构造函数和转换示例

- 例如：complex类提供了一个构造函数，能够将double转换到complex，因此，我们就只需要为comple的==运算符定义一个版本

```
bool operator==(complex,complex);
void f(complex x, complex y)
{
    x == y; // operator==(x,y)
    x == 3; // operator==(x, complex(3))
    3 == y; // operator==(complex(3), x)
}
```

11.3.6 文字量

- 不能为类类型定义文字量(例如：double类型的1.2或者12e3等)
- 但是，我们常常可以利用内部类型的文字量，只需要通过类的成员函数对其提供一种解释
- 当构造函数很简单并且被内联时，把这种以文字量为参数的构造函数就看成是文字量也是相当合理的
- 例如：可以把complex(3)看成是complex类型的文字量，虽然从技术上说它并不是

11.3.7 另一些成员函数

- 为了使得complex更好的工作，还可以定义其他函数，比如说：检查实部和虚部的值，然后利用这些函数进行各种有用的运算

```
class complex{
    double re, im;
public:
    double real() const { return re; }
    double imag() const { return im; }
};
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

11.3.8 协助函数

- 将所有的东西集中到一起，complex就变成了

```
class complex
{
    double re, im;
public:
    complex(double r=0, double i=0) : re(r), im(i) {}
    double real() const { return re; }
    double imag() const { return im; }
    complex& operator+=(complex);
    complex& operator+=(double);
    // -=, *=, /= ...
};
```

协助函数

- 除此之外，还必须提供一批协助函数

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);
// -, *, / ...

complex operator+(complex); //一元正号
complex operator-(complex); //一元负号
bool operator==(complex, complex);
bool operator!=(complex, complex);
istream& operator>>(istream&, complex&);
ostream& operator<<(ostream&, complex);
// 还可能提供一些以极坐标方式处理复数的函数、数学函数
// 可以参阅标准库的<complex>
```

11.4 转换运算符

- 虽然通过构造函数来进行类型转换确实很方便，但是构造函数无法刻画
  - 从用户类型到一个内部类型的转换(因为内部类型不是类)
  - 从新类型到某个已有类型的转换(而不去修改那个已有类的声明)
- 这些问题可以通过为源类型定义转换运算符的方式解决
- 成员函数X::operator T()，其中T是一个类型名，定义了一个从X到T的转换

转换运算符示例

```
class Tiny
{
    char v;
    void assign(int i)
        { if(i&~077) throw Bad_range(); v = i; }
public:
    class Bad_range{};
    Tiny(int i) { assign(i); }
    Tiny& operator=(int i)
        { assign(i); return *this; }
    operator int() const { return v; }
    // 转换到int的函数，不需要声明返回值
    // 注意：int operator int() const { return v; } 是错误的
};
```

转换运算符示例

```
int main()
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2-c1; // c3 = 60
    Tiny c4 = c3; // 不检查值域(没必要)
    int i = c1+c2; // i = 64

    c1 = c1+c2; // 值域错误：c1不能是64
    i = c3-64; // i = -4
    c2 = c3-64; // 值域错误：c2不能是-4
    c3 = c4; // 不需要检查值域
}
```

istream和ostream也依靠类型转换函数，这使得我们可以写下面这类语句

```
while(cin>>x)
    cout << x;

// cin>>x返回istream&
// 这个值被隐式地转换到
// 一个表明cin状态的值
// 然后被while检测
// 但是这类转换丢失了
// istream&信息
```

11.4.1 歧义性

- 如果同时存在用户定义转换和用户定义运算符，那么也可能产生用户定义运算符和内部运算符之间的歧义性问题

```
int operator+(Tiny, Tiny);
voif f(Tiny t, int i)
{
    t+i; // 错误：歧义，operator+(t,Tiny(i)) 或 int(t)+i ?
}
// 因此，对于一个类型而言，最好是或者依靠用户定义转换
// 或者依靠用户定义运算符，但不要两者都用
```

歧义性示例

```
class X{/*...*/X(int);X(char*);};
class Y{/*...*/Y(int);};
class Z{/*...*/Z(X);};

X f(X);  Y f(Y);   Z g(Z);

void k1()  {
    f(1); // 错误：歧义的f(X(1))或f(Y(1))?
    f(X(1));  // ok
    f(Y(1));  // ok
    g("Mack");
    // 错误：需要两次用户定义转换，不会试验g(Z(X("Mack")))
    g(X("Doc")); // ok: g(Z(X("Doc")))
    g(Z("Suzy"));// ok: g(Z(X("Suzy")))
}
```

歧义性示例

- 只有在解析一个调用时有需要，才会考虑用户定义的转换

```
class XX {/*...*/XX(int); };

void h(double);
void h(XX);

void k2()
{
    h (1); // 使用h(double(1))还是h(XX(1))?
    // 结果是调用h(double(1))，因为这一选择只使用了标准转换
};
```

歧义性示例

```
class Quad
{
public:
    Quad(double);
};
Quad operator+(Quad, Quad);
void f(double a1, double a2)
{
    Quad r1 = a1+a2; // 双精度加法
    Quad r2 = Quad(a1) + a2; // 强制要求Quad算术
}
```

11.5 友元

- 一个常规的成员函数声明描述了三件在逻辑上相互不同的事情
  - 该函数能访问类声明的私用部分
  - 该函数位于类的作用域之中
  - 该函数必须经由一个对象去激活(有一个this指针)
- static函数具有前两种性质
- 友元(friend)则只具有第一种性质

友元示例

- 比如说，我们希望定义一个运算符去做Matrix和Vector的乘法，然而，我们所要的这个乘法例程不可能同时成为两者的成员函数，此外，我们也不希望提供一些低级访问函数，使每个用户都能对Matrix和Vector的完整表示进行读写操作

```
class Matrix;
class Vector{
    float v[4];
    friend Vector operator*(const Matrix&, const Vector&);
};
class Matrix{
    Vector v[4];
    friend Vector operator*(const Matrix&, const Vector&);
};
```

友元示例

- friend声明可以放在类声明中的任何部分(私用或者公用)，放在哪里都没有关系
- 一个类的成员函数也可以是另一个类的友元
- 某个类的所有函数可以全都是另一个类的友元

```
class List_iterator{
    int* next();
};
class List{
    firend int* List_iterator::next();
};
```

```
class List{
    friend class List_iterator;
};
```

11.5.1 友元的寻找

- 像成员的声明一样，一个友元声明不会给外围的作用域引进一个名字

```
class Matrix{
    friend class Xform;
    friend Matrix invert(const Matrix&);
};

Xform x; // 错误：作用域里无Xform
Matrix (*p)(const Matrix&) = &invert;
// 错误：作用域里无invert()
```

友元的寻找示例

- 一个友元类必须或者是在外围作用域里先行声明的
- 或者是在它作为友元的那个类的直接外围的非类作用域里定义的
- 在外围的最内层名字空间作用域之外的作用域不在考虑之列

```
class X{/*…*/}; // Y的友元
namespace N{
    class Y{
        friend class X;
        friend class Z;
        friend class AE;
    };
    class Z{/*…*/}; //Y的友元
}
class AE{/*…*/};//不是Y的友元
```

友元的寻找示例

- 友元函数可以像友元类一样明确地声明，此外，还可以通过它的参数找到，即使如果它并没有在外围最近的作用域里声明

```
void f(Matrix& m)
{
    invert(m);
    //Matrix的友元invert()
}
```

11.5.2 友元和成员

- 什么时候选用友元函数，什么时候用成员函数刻画一个运算才更好？
  - 应该尽可能地减少访问类表示的函数的数目
  - 尽可能地将能访问的函数集合做好

11.6 大型对象

- 前面complex的运算符都定义为以类型complex为参数，意味着对于每个运算符，各个运算对象都需要复制，为了避免过度的复制，我们需要定义以引用为参数的函数

```
class Matrix{
    double m[4][4];
public:
    Matrix();
    friend Matrix operator+(const Matrix&, const Matrix&);
    friend Matrix operator*(const Matrix&, const Matrix&);
};
```

大型对象示例(一种避免复制的技术)

- 使用静态对象的缓冲区

```
const int max_matrix_temp=7;
Matrix& get_matrix_temp()
{
    static int nbuf = 0;
    static Matrix
        buf[max_matrix_temp];

    if(nbuf==max_matrix_temp)
            nbuf=0;
    return buf[nbuf++];
}
```

```
Matrix& operator+(const Matrix&
        arg1, const Matrix& arg2)
{
    Matrix& res=get_matrix_temp();
    //...
    return res;
}
```

11.7 基本运算符

- 一般说来，对于类型X，复制构造函数X(const X&)负责完成从同类型X的一个对象出发的初始化
- 如果一个类X有一个完成非平凡工作的析构函数，这个类很可能需要一组完整的互为补充的函数来控制构造、析构和复制
- 对于一个类，如果程序员没有显式地声明复制赋值或者复制构造函数，编译器就会生成所缺少的操作

```
class X{
    X(Sometype);
    X(const X&);
    X& operator=(const X&);
    ~X();
};
```

11.7.1 显式构造函数

- 按照默认规定，只有一个参数的构造函数也定义了一个隐式转换，例如：complex z=2; // 用complex(2)初始化
- 但是有时候我们也不希望进行隐式转换，例如，我们可以用一个int作为大小去初始化一个string，但是，某人或许会写出：string s = ‘a’; // 将s做成一个包含int(‘a’)个元素的string
- 通过将构造函数声明为explicit(显式)的方式可以抑制隐式转换

显式构造函数示例

```
class String
{
    explicit String(int n);
    String(const char* p);
};
String s1='a';  //错误
String s2(10);  //可以
String s3=String(10);//可以
String s4="Brian";//可以
String s5("Fawlty");

void f(String);
```

```
String g()
{
    f(10); //错误
    f(String(10));//可以
    f("Arthur");//可以
    f(s1);
    String* p1=new String("Eric");
    String* p2=new String(10);
    return 10;
    // 错误
}
```

显式构造函数示例

```
class Year
{
    int y;
public:
    explicit Year(int i):y(i) {}
    operator int()const {return y;}
};
class Date
{
public:
    Date(int d, Month m, Year y);
};
Date d3(1978,feb,21);  //错误
Date d4(21,feb,Year(1978));//可以
```

Year只不过是包裹着int的一层布，由于有operator int()，很容易将Year转换成一个int数值

将构造函数声明为explicit，可以保证int到Year的转换只能在明确要求的地方进行

类似的技术也可以用于定义表示范围的类型

11.8 下标

- 函数operator[]可以用于为类的对象定义下标运算的意义
- operator[]的第二个参数(下标)可以具有任何类型，这就使我们可以去定义vector、关联数组等
- operator[]()必须是成员函数

下标示例

```
class Assoc {
    struct Pair {
        string name;
        double val;
        Pair(string n="",double v=0):name(n), val(v) {}
    };
    vector<Pair> vec;
    Assoc(const Assoc&); //私用，防止复制
    Assoc& operator=(const Assoc&);//私用，防止复制
public:
    Assoc(){}
    const double& operator[](const string&);
    double& operator[](string&);
    void print_all() const;
};
```

下标示例

```
double& Assoc::operator[](string& s){
// 检索s；如果找到就返回其值；
// 否则，做一个新的Pair并返回默认值0
    for(vector<Pair>::const_iterator p = vec.begin();
        p!=vec.end(); ++p)
            if(s==p->name) return p->val;
    vec.push_back(Pair(s,0));
    return vec.back().val;
}
void Assoc::print_all() const{
    for(vector<Pair>::const_iterator p=vec.begin();
        p!=vec.end();++p)
        cout << p->name<<":"<<p->val<<'\n';
}
```

下标示例

```
int main()
// 计算每一个单词在输入时出现的次数
{
    string buf;
    Assoc vec;
    while(cin>>buf) vec[buf]++;
    vec.print_all();
}
```

11.9 函数调用

- 函数调用，记法expression(expression-list)，也可以解释为一种二元符号，其中将expression作为左运算对象，而expression-list作为右运算对象，该调用运算符()也可以被重载
- 运算符()最明显、或许也是最重要的应用就是为对象提供常规的函数调用语法形式，使它们具有像函数似的行为方式，一个活动起来像函数的对象常常被称作一个拟函数对象，或简称为函数对象(仿函数对象 functor)
- operator()()的另外一些很流行的应用包括作为子串运算符，以及作为多维数组的下标运算符
- operator()()必须作为成员函数

函数调用示例

```
template<class Iter, class Fct>Fct for_each
    (Iter b, Iter e, Fct f)
{
    while(b!=e) f(*b++);
    return f;
};
// for_each是一个模板，它反复地应用第三个参数的()
void negate(complex& c){ c = -c; }

void f(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), negate);
    for_each(ll.begin(), ll.end(), negate);
}
```

函数调用示例

```
void add23(complex& c)
{
    c+=complex(2,3);
}
void g(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), add23);
    for_each(ll.begin(), ll.end(), add23);
}

// 解决向一个向量或者列表中所有元素加一个固定的值的问题
```

函数调用示例

```
class Add{
    complex val;
public:
    Add(complex c) { val=c; }
    Add(double r, double i){ val = complex(r,i); }
    void operator()(complex& c) const{ c+= val; }
};

void h(vector<complex>& aa, list<complex>& ll, complex z)
{
    for_each(aa.begin(), aa.end(), Add(2,3));
    for_each(ll.begin(), ll.end(), Add(z));
}
// 解决向一个向量或者列表中所有元素加一个任意的值的问题
```

11.10 间接(Dereferencing)

- 间接(提领)运算符->可以被定义为一个一元的后缀运算符，即：给出了类

```
class Ptr{
    /*…*/
    X* operator->();
};
```

就可以用类Ptr的对象访问类X的成员，其方式就像所用的Ptr的对象是一个指针，例如：

```
void f(Ptr p)
{
    p->m=7;//(p.operator->())->m=7
} // 其中m是类型X的成员
```

间接

- 从对象p到指针p.operator->()的转换并不依赖于被指向对象的成员m，这也是->被看作是一元后缀运算符的理由，但是，无论如何这里并没有引进一种新语法形式，所以，在->之后仍然要求写一个成员名字

```
void g(Ptr p)
{
    X* q1 = p->; // 语法错误
    X* q2 = p.operator->(); // ok
}
```

间接

- 间接是一个很重要的概念
- 迭代器是这个说法的一个重要佐证
- 认识运算符->的另一种方式是把它看成C++里提供的一种受限的，但也非常有用的委托(delegation)机制
- 运算符->必须是成员函数，如果使用，它的返回类型必须是一个指针，或者是一个你可以使用->运算符的类对象

间接示例

- 重载->的最有用之处就是创建所谓的“灵巧指针”,也就是一种行为像是指针的对象

```
class Rec_ptr {
    const char* identifier;
    Rec* in_core_address;
public:
    Rec_ptr(const char* p):identifier(p),in_core_address(0){}
    ~Rec_ptr(){ write_to_disk(in_core_adress, identifier);}
    Rec* operator->();
};
Rec* Rec_ptr::operator->() {
    if(in_core_address==0)
            in_core_address=read_from_disk(identifier);
    return in_core_address;}
```

间接示例

```
struct Rec
{
    string name;
};

void update(const char* s)
{
    Rec_ptr p(s); // 对s得到一个Rec_ptr
    p->name = “Roscoe”;
    // 更新s：如果需要，第一次将从磁盘读取
}
// 真正使用时，Rec_ptr可能是个模板，而Rec类型则是它的参数
// 此外，实际程序可能需要包含错误处理代码等
```

间接示例

- 对于常规指针，->的使用和一元*以及[]的某些使用方式意义相同，有了Y* p;则下面的关系成立：p->m==(*p).m==p[0].m(其中m是Y的成员)
- 对于用户自己定义的->()运算符，不保证这些关系，但若必要，用户也可以提供这些等价性

```
class Ptr_to_Y{
    Y* p;
public:
    Y* operator->(){ return p;}
    Y& operator*(){ return *p; }
    Y& operator[](int i) { return p[i]; }
};
```

11.11 增量和减量

- 一旦人们定义了“灵巧指针”，他们也常常会决定需要提供增量运算符++和减量运算符--，去模拟这些运算符对于内部类型的使用

```
class Ptr_to_T
{
    T* p;
    T* array;
    int size;
public:
    Ptr_to_T(T* p, T* v, int s);
    Ptr_to_T(T* p);
    Ptr_to_T& operator++(); // 前缀
    Ptr_to_T operator++(int);// 后缀
    Ptr_to_T& operator--();// 前缀
    Ptr_to_T operator--(int);// 后缀
    T& operator*();// 一元*
}; // 注意后缀操作符声明中虚设的参数
```

增量和减量示例

```
void f1(T a)
{
    T v[200];
    T* p=&v[0];
    p--;
    *p=a;
    // p越界，没有捕捉到
    ++p;
    *p=a; // ok
}
```

```
void f2(T a)
{
    T v[200];
    Ptr_to_T p(&v[0],v,200);
    p--; // p.operator--(0);
    *p=a;
    // p.operator*() = a;
    // 运行错误，p越界
    ++p;
    *p=a; // ok
}
```

11.12 一个字符串类

- 一个更实际的String类版本
- 提供了一种值语义、字符读写操作、检查和不检查的访问、流I/O、用字符串文字量作为文字量、相等和拼接运算符，将字符串表示为C风格的、用0结束的字符数组，并利用引用计数尽可能地减少复制

11.13 忠告

- [1] 定义运算符主要是为了模仿习惯使用方式
- [2] 对于大型运算对象，请使用const 引用参数类型
- [3] 对于大型的结果，请考虑优化返回方式
- [4] 如果默认复制操作对一个类很合适，最好是直接用它
- [5] 如果默认复制操作对一个类不合适，重新定义它，或者禁止它
- [6] 对于需要访问表示的操作，优先考虑作为成员函数而不是作为非成员函数
- [7] 对于不访问表示的操作，优先考虑作为非成员函数而不是作为成员函数

忠告

- [8] 用名字空间将协助函数与“它们的”类关联起来
- [9] 对于对称的运算符采用非成员函数
- [10] 用()作为多维数组的下标
- [11] 将只有一个“大小参数”的构造函数做成explicit
- [12] 对于非特殊的使用，最好是用标准string而不是你自己的练习
- [13] 要注意引进隐式转换的问题
- [14] 用成员函数表达那些需要左值作为其左运算对象的运算符