Class and object
运算符重载

董洪伟
http://hwdong.com

Operator Overloading

- C++对类型类型提供了一组简洁、方便的运算符。如

  x+y*z;　　//clear

  add(x,mul(y,z)); //not good

- 希望这些运算符也能用于用户定义类型，如字符串、矩阵代数等。
- 例如<<、>>运算符，string类重载的运算符+、+=等

Complex (复数)

- 数学中的整数、实数在C/C++语言中都是内在类型，但复数不是。
- 数学中复数由实部和虚部构成，其写法各种各样：a+ib ,(a,b) ,<a,b>等
- 如何在计算机中表示复数？

```
struct Complex{
        double a,b;
};
```

Complex：定义加减乘除运算

Complex add(Complex a, Complex b);

Complex aub(Complex a, Complex b);

Complex mul(Complex a, Complex b);

Complex div(Complex a, Complex b);

```
Complex x(3,4),y(5,6),z(7,8),u,v;
u = add(x,y);
v = sub(div(u,x),z);    //v = u/x-z
```

我们希望直接写成更简洁直观的形式，如

v = u/x-z

Complex

```
class complex{ / / very simplified complex
  double re, im;
public:
  complex(double r, double i) : re(r) , im(i) { }
 friend  complex operator+ (const complex &a，  const
    complex &b) ;
 friend  complex operator* (const complex &a，  const
    complex &b) ;
//…
};
```

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;
   c = a*b+complex(1,2) ;
}
```

普通的优先规则仍然保持

Complex：成员和友元的区别

```
class complex{ // very simplified complex
   double re, im;
public:
   complex(double r, double i) : re(r) , im(i) { }
  friend  complex operator+ (const complex &a,
     const complex &b) ;
   complex& operator+ (const complex &a);
 //…
};
```

Complex：成员和友元的区别

```
complex x(3,5),y(4,7),z(0,0);
z = x+y;
```

```
friend  complex operator+ (const complex
    &a，  const complex &b) ;
complex operator+ (const complex &a);
```

调用哪一个？

可以重载的运算符

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

不能重载的运算符

- :: (作用域限定)
- . (成员选择)
- .* (成员指针选择).
- ? : (条件运算符)
- sizeof　测试大小运算符
- typeid　类型id

运算符函数名

- 关键字operator后跟运算符构成。如

  operator<<

- 调用运算符时可省略关键字operator.如

```
void f (complex a, complex b){
  complex c = a + b;      // 简写
  complex d = a. operator+(b) ;  //显式调用
}
```

重载运算符既可是成员函数也可全局函数

```
class complex{
        ……
    public:
        bool operator==(const complex& c);
}
bool operator==(const complex& c1,
                const complex& c);
```

二元运算符@

- 可解释为aa.operator@(bb),或operator@(aa,bb)
- 可以定义为取一个参数的非静态成员函数，也可以定义为两个参数的非成员函数。

二元运算符aa@bb

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

```
void f( X a){
   a+2;
   2+a;
   a+2.5
}
```

二元运算符aa@bb

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

```
void f(X a){
   a+2;
   2.operator+(a);
   a+2.5
}
```

一元运算符@

- 可解释为aa.operator@(),或operator@(aa)

运算符定义不能违背约定的语法

- 如不能将一元运算符定义为二元的或三元的。

```
class X {
    X* operator&() ;        // 一元取地址
    X operator&(X) ;        // 二元 ‘与’
   X operator++(int) ;     // 后缀增量
   X operator&(X,X) ;      // error: ternary
   X operator/() ;          // error: unary /
};
```

运算符定义不能违背约定的语法

```
// nonmember functions :
X operator-(X) ;       // prefix unary minus
X operator-(X,X) ;       // binary minus
X operator--(X&,int) ;    // postfix decrement
X operator-();     // error: no operand
X operator-(X,X,X) ;   // error: ternary
X operator%(X) ;   // error: unary %
```

其他注意点

- 重载运算符不能有缺省参数
- 除operator=外，重载运算符被派生类所继承
- 重载运算符的第一个参数一定是该类或派生类类型。如

```
point p;
3.+=p;  //错！
```

其他注意点

- =、[]、()、 ->必须是非静态成员函数
- 为防止误用运算符(如=，&，，)，可以通过定义私有运算符函数，将它们设为私有。

```
class X {
private:
    void operator=(const X&) ;
    void operator&() ;
    void operator, (const X&) ;
    // ...
};
```

```
void f(X a, X b){
  a = b;
  &a;
  a, b;
}
```

运算符的重载解析

- 如果有多个同名运算符，需要根据运算对象进行重载解析。
- 如对二元运算符@(x,y)，x的类型是X， y的类型是Y，解析过程：

1)在X或X基类的成员函数中找operator@

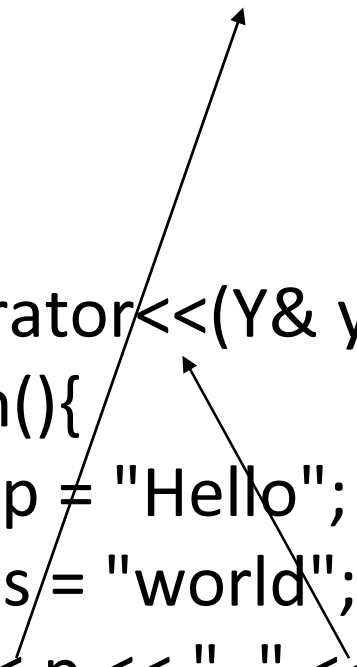
2)在x@y的环境中找operator@

3)在X的名字空间N中找operator@

4)在Y的名字空间M中找operator@

运算符的重载解析

```
class Y {
public:
   Y& operator<<(const char*) { return *this;   }
   //…
};
Y yval;
Y& operator<<(Y& y, const string&) {return y; }
int main(){
  char* p = "Hello";
  string s = "world";
  yval << p << ", " << s << "!\n";
}
```



成员运算符和非成员运算符

- 对于那些修改其第一个参数的运算符如*=可定义为成员运算符，否则，定义为非成员运算符如+。
- C++必须要求作为成员函数的运算符：=、[ ]、( )、->

成员运算符和非成员运算符

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

几种特殊运算符

- 赋值运算符=
- 下标运算符[]
- 类成员访问运算符->
- 类型转换运算符T()
- 函数调用运算符()
- 增量与减量运算符++，--

赋值运算符=

- C++会默认定义一个隐式的赋值运算符，用于同类型对象之间的赋值。默认行为是成员硬拷贝
- 如果隐式赋值运算符不能满足要求，就可以显示定义赋值运算符，但不能被派生类继承

```
class String{
  char *s;
  int sz;
};
```

```
String s1 = “Li”,s2;
s2 = s1;
```

赋值运算符=

```
class String{
   char *s;
   int sz;
 public:
    String() {s = 0; sz = 0;}
    String(const char *str);
};
```

赋值运算符=

```
String::String(const char *str){
   sz = strlen(str);
   s= new char[sz+1];
   strcpy(s,str);
}

void f(){
   String s1 = “Li”,s2;
   s2 = s1;
}
```

赋值运算符=

- 拷贝构造函数与赋值运算符的区别：

  拷贝构造函数在定义一个对象时被调用，而赋值运算符用一个对象对另一个对象赋值。

```
class X{…  X(int);  …};
X  x, y;
X z = x;    //拷贝构造
y  = z;     //赋值运算符
x = 3;      //类型转换+赋值运算符
```

下标运算符[]

- 用于根据下标访问有序集合类型的数据元素

```
class String{
    char *s;    int sz;
  public:
    //…
    char& operator[](int i){
        return s[i];
   }
};
```

```
String s1 = “Li” ,s2;
S1[1] = ‘L’;
S1[3] = ‘T’;
```

类成员访问运算符->

- 二元运算符，其第一操作数是指向类类型的指针。其第二个操作数是该类的成员。

```
class String{
    char *s;    int sz;
  public:
    //…
    int  size(){return sz;}
};
```

```
String s = “Li”,*p;
int n = s.size();
p = &s;
n = p->size();
```

类型转换运算符T()

- 带参数的构造函数相当于从参数类型到该类类型的类型转换。

```
String(const char *str);
```

- 类也可以定义类型转换运算符将该类对象转换为其他类型的对象。如

```
class A{
    …
    operator int() { };
};
```

类型转换运算符T()

- 当一个类同时定义了一个参数(t类型)的构造函数和一个t类型的类型转换函数时，有时会引起歧义。

```
class A{
    …
    A(int);
    operator int();
    friend A operator+(const A& a1,
        const A& a2);
};
```

```
A a;
int i = 1,z;
z= a+i; //bad

z = (int)a +i;
或z= a +(A)i
```

函数调用运算符()

- 函数调用()使得类对象可以当成一个函数来使用。如

```
class Matrix{
  …
  double operator()(int i,int j);
   //double get(int i,int j);
};

Matrix M;

double s = M(1,2) ;

double s = M.get(1,2);
```

间接访问运算符->

- 主要用于创建一种“灵巧指针”
- 可以用类PtrX的对象来访问类X对象的成员

```
class X{
  int m;
};
Class PtrX{
  X* operator->();
};
 void f(PreX p){
   p->m = 7;  // (p.operator->())->m = 7;
}
```

operator->必须为成员函数

间接访问运算符->

```
class ref_counter {
    unsigned int _counter;
public:
    ref_counter(void) : _counter(0) { }
    virtual ~ref_counter() { }

    void increase() {  ++_counter;   }
    void decrease() {
        if (–_counter == 0) { delete this;  }
     }
};
```

间接访问运算符->

```
class Sample : public ref_counter  {
      public:
          void doSomething(void) {
           TRACE("Did something\n");};
};
```

间接访问运算符->

```
typedef Sample  T;
class  Ptr    {      T* p;
public:
    Ptr(T* p_) : p(p_) { p->increase(); }
    virtual ~Ptr( void ) { p->decrease(); }
    operator T*(void) { return p; }
    T& operator*(void) { return *p; }
    T* operator->(void) { return p; }
    Ptr& operator=(Ptr<T> &p_) {   return operator=((T *) p_);
      }
    Ptr& operator=(T* p_) {
        p->decrease(); p = p_; p->increase(); return *this;
    }
};
```

间接访问运算符->

```
int main() {
    Ptr<Sample> p  = new Sample; // sample #1
    Ptr<Sample> p2 = new Sample; // sample #2
    p = p2; // #1 will have 0 crefs, so it is destroyed;
                // #2 will have 2 crefs.
    p->doSomething();
    return 0;
    // As p2 and p go out of scope, their destructors call
    // downcount. The cref variable of #2 goes to 0, so #2 is
    // destroyed
}
```

增量与减量运算符++，--

- 自增++和自减—都可以被重载，但有前置和后置的区分：

```
int x=6;
x++;  //前置增
++x;  //后置增
```

- 前置增和后置增声明：

```
operator++();
operator++( int );
```

增量与减量运算符++，--

```
class Counter {
  int _i;
  public:
      Counter(int i):_i(i){ }
      Counter& operator++(   ); //前缀增
      Counter  operator++( int ); //后缀增
};
   void main(){
        Counter C;
        ++C;   C++;   std::cout<<C<<“\n”;
   }
```

增量与减量运算符++，--

- 当然也可以定义自增和自减运算符为外部友元函数

```
class Counter {
   //….
   friend Counter& operator++(Counter& ); //前缀增
   Counter  operator++(Counter& , int ); //后缀增
   //….
};
```

增量与减量运算符++，--

- 人们可能定义“灵巧指针”来方便指针操作的使用。看一个传统程序：

```
void f (T a) {
    T v[20];
    T *p = &v[0];
    p-- ;
    *p = a;  //超出范围！,没有被捕捉到
    ++p;
    *p  =a; //ok
}
```

增量与减量运算符++，--

- 灵巧指针类 Ptr2T

```
void f (T a) {
    T v[20];
    Ptr2T p(&v[0],v,200);
    p-- ;
    *p = a;  //超出范围！，被捕捉到
    ++p;
    *p  =a; //ok
}
```

增量与减量运算符++，--

```
class Ptr2T{
   T *p; T *array; int size;
public:
  Ptr2T(T* p,T *v=0,int s=0);
  Ptr2T& operator++(); //前缀
  Ptr2T operator++(int); //后缀
  Ptr2T& operator--(); //前缀
  Ptr2T operator--(int); //后缀
  T& operator*();//前缀
};
```

增量与减量运算符++，--

```
Ptr2T Ptr2T::(T* p_,T *v=0,int s=0){
    p =  p_; array = v; size = s;
}

Ptr2T& Ptr2T::operator--(){
   if(p-v<0) throw Range_error()
   --p;
}
```

作业和练习

- 设计和实现下列类：

  复数类Complex，向量类Vector，字符串类String，矩阵类Matrix