运算符重载

Operator Overloading

Why Operator Overloading?

```
class Point {
public:
   double x_, y_;
   Point (double x =0, double y = 0):x_(x),y_(y) { }
};

 int main(){
    Point  a(1., 2) , b(3,4);
    Point  c = a + b;
    return 0;
 }
```

Why Operator Overloading?

```
class Point {
public:
   double x_, y_;
   Point (double x =0, double y = 0):x_(x),y_(y) { }
};

 int main(){
    Point  a(1., 2) , b(3,4);
    Point  c = a + b;
    return 0;
 }
```

error: no match for 'operator+' (operand types are 'Point' and 'Point') Point c = a + b;

What Can I Overload?

- 可重载的运算符

Algorithm	**+ , - , * , / , %, ^ , &, |, ~ , ! , = , +=, -= , *= , /=, %=, ^=, &=, |=, && , ||, ++ , --,**
Comparison	!= , == , < , <= , > , >=
Access	[ ], *,->, ( ) ,
Stream	<<, >>, >>=, <<=
Scary	New, delete , New[], delete[] , ‘,’

Operator Overloading

- 允许我们对新类型重新定义'==’等运算符的新含义. 当且仅当两点的x, y坐标分别相等.
- 如何重载一个运算符?
- 两种方式:
  - Member function syntax
  - Free function syntax

Operator Overloading

- 允许我们对新类型重新定义'==’等运算符的新含义

```
class Point {
public:
   double x_, y_;
   Point (double x =0, double y = 0):x_(x),y_(y) { }
    bool operator== (const Point& rhs){
         return (x == rhs.x && y == rhs.y);
  }
};
```

```
int main(){
    Point p1(3, 2);
    Point p2(3, 2);
    if (p1 == p2)
        cout << “Points are equal!” << endl;
     return  0;
}
```

```
int main(){
    Point p1(3, 2);
    Point p2(3, 2);
    if (p1. operator== p2)
            cout << “Points are equal!” << endl;
     return  0;
}
```

Free Function Syntax

```
bool operator==(Point l, Point r) {
  return l.x == r.x && l.y == r.y;
}

int main(){
    Point p1(3, 2);
    Point p2(3, 2);
    if (p1 == p2)
          cout << “Points are equal!” << endl;
     return  0;
}
```

Free Function Syntax

```
Point operator*(double l, Point r) {
   Point result(l * r.x, l * r.y);
   return result;
}

int main(){
    Point p(1, 1);
    Point result = 5 * p;
    result.print();   // prints (5, 5)
     return  0;
}
```

Free Function Syntax

```
bool operator==(Point l, Point r) {
  return l.x == r.x && l.y == r.y;
}

int main(){
    Point p1(3, 2);
    Point p2(3, 2);
    if (operator== (p1 , p2) )
          cout << “Points are equal!” << endl;
     return  0;
}
```

Operator Overloading

```
class Point {
public:
   double x_, y_;
   Point (double x =0, double y = 0):x_(x),y_(y) { }
   Point  operator+ (Point Q){
      return Point( x_+ Q.x_ , y_+ Q.y_);
  }
};
```

Operator Overloading

```
class Point {
public:
   double x_, y_;
   Point (double x =0, double y = 0):x_(x),y_(y) { }
};

Point  operator+ (Point P, Point Q){
    return Point( P.x_+ Q.x_ , P.y_+ Q.y_);
}
```

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。
- 当左操作数不是类类型时，则必须为非成员函数

```
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)

++i vs i++

```
int i = 1;  // i = 1
int j = i++;  // j = 1, i = 2
int k = ++i;  // k = 3, i = 3
```

++i vs i++

```
int i = 1;  // i = 1
int j = i++;  // j = 1, i = 2
int k = ++i;  // k = 3, i = 3
```

```
class foo {
   // Define ++foo
    foo operator++() {
        …
    }
    // Define foo++
    foo operator++(int) {
      …
    }
}
```

Cautions: 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 /
};
```

Cautions: 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 %
```

Other Cautions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！
```

Other Cautions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;
}
```

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=

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

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

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

拷贝构造函数

赋值运算符

赋值运算符=

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

```
String & String::operator=
(String &T){
   sz = strlen(T.s);
   s= new char[sz+1];
   strcpy(s,T.s);
}

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

拷贝构造函数

赋值运算符

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);
};

void f()  {
    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[]

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

```
void f(){
    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;}
};
```

```
Void f(){
   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()使得类对象可以当场一个函数来使用。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);
```

友元friend

- Complex类

```
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) ;
  //…
};
```

类的友元函数可以访问该类对象的私有数据

友元friend

- Complex类

```
complex operator+ (const complex &a，
          const complex &b) {
    return Complex( a.re + b.re, a.im + b.im ) ;
};
```

类的友元函数可以访问该类对象的私有数据

作业

- 实现相对完整的Complex、String、 Matrix类