操作符重载

基本做法

主要内容

- 操作符重载概述
- 双目操作符重载
- 单目操作符重载
- 操作符++和-- 的重载

操作符重载的需要性

- C++语言本身没有提供复数类型，可通过定义一个类来实现：

```
class Complex     //复数类定义
{public:
    Complex(double r=0.0,double i=0.0)
    { real=r; imag=i;
    }
    void display() const
    { cout << real << '+' << imag << 'i';
    }
    ......
  private:
    double real;
    double imag;
};
Complex a(1.0,2.0),b(3.0,4.0);
```

- 如何实现两个复数（类型为Complex）相加？

用函数实现

- 一种方案：为Complex类定义一个成员函数add，例如：

```
class Complex
{ public:
        ……
        Complex add(const Complex& x) const
        {    Complex temp;
             temp.real = real+x.real;
             temp.imag = imag+x.imag;
             return temp;
        }
  private:
       double real;
       double imag;
};
……
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a.add(b);
```

- 另一种方案：定义一个全局函数（声明成Complex类的友元），例如：

```
class Complex     //复数类定义
{  ……
   friend Complex add(const Complex& x1, const Complex& x2);
 private:
   double real;
   double imag;
};
Complex add(const Complex& x1, const Complex& x2)
{  Complex temp;
   temp.real = x1.real+x2.real;
   temp.imag = x1.imag+x2.imag;
   return temp;
}
……
Complex a(1.0,2.0),b(3.0,4.0),c;
c = add(a,b);
```

用操作符重载实现

- 用函数来实现复数的加法操作不符合数学上的使用习惯：

  c=a+b

- C++允许对已有的操作符进行重载，使得用它们能对自定义类型（类）的对象进行操作。
- 与函数名重载一样，操作符重载也是实现多态性的一种语言机制。

C++操作符重载的实现途径

- 操作符重载可通过定义一个函数名为“operator #”（“#”代表某个可重载的操作符）的函数来实现，该函数可以作为：
  - 一个类的非静态的成员函数（操作符new和delete除外）。
  - 一个全局（友元）函数。

- 例如，以成员函数重载复数的“+”：

```
class Complex
{ public:
    Complex operator + (const Complex& x) const
    {       Complex temp;
            temp.real = real+x.real;
            temp.imag = imag+x.imag;
            return temp;
    }
    ……
};
……
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a + b;
```

- 再例如，以全局函数重载复数的“+”：

```
class Complex
{ ……
   friend Complex operator + (const Complex& c1,
                              const Complex& c2);
};
Complex operator + (const Complex& c1,
                    const Complex& c2)
{ Complex temp;
   temp.real = c1.real + c2.real;
   temp.imag = c1.imag + c2.imag;
   return temp;
}
……
Complex a(1.0,2.0),b(3.0,4.0),c;
c = a + b;
```

- 一般情况下，操作符既可以作为全局函数，也可以作为成员函数来重载。
- 在有些情况下，操作符只能作为全局函数或只能作为成员函数来重载！

操作符重载的基本原则

- 只能重载C++语言中已有的操作符，不可臆造新的操作符。
- 可以重载C++中除下列操作符外的所有操作符：
  “.”，“.*”，“?:”，“::”，“sizeof”
- 需要遵循已有操作符的语法：
  - 不能改变操作数个数。
  - 原操作符的优先级和结合性不变。
- 尽量遵循已有操作符原来的语义：
  - 语言本身没有对此做任何规定，使用者自己把握！

双目操作符重载

- 作为成员函数重载
  - 只需要提供一个参数，其类型为第二个操作数的类型
  - 定义格式

```
class <类名>
{ ……
   <返回值类型> operator # (<类型>); //#代表可重载的操作符
};
<返回值类型> <类名>::operator # (<类型> <参数>) { …… }
```

  - 使用格式

```
<类名> a;
<类型> b;
a # b
或
a.operator#(b)
```

例、实现复数的“等于”和“不等于”操作

```
class Complex
{   double real, imag;
   public:
    ……
    bool operator ==(const Complex& x) const
    {   return (real == x.real) && (imag == x.imag);
    }
    bool operator !=(const Complex& x) const
    {   return (real != x.real) || (imag != x.imag);
    }
};
……
Complex c1,c2;
……
if (c1 == c2) //或 if (c1 != c2)
……
```

```
bool operator !=(const Complex& x) const //更好！
{   return !(*this == x);
}
```

双目操作符重载（续1）

- 作为全局（友元）函数重载
  - 需要提供两个参数，其中至少应该有一个是类、结构、枚举或它们的引用类型。
  - 定义格式

    <返回值类型> operator #(<类型1> <参数1>,
    <类型2> <参数2>)
    { …… }
  - 使用格式

    <类型1> a;
    <类型2> b;
    a # b
    或
    operator#(a,b)

例：重载操作符+，使其能够实现实数与复数的混合运算。

```
class Complex
{    double real, imag;
  public:
     Complex() { real = 0; imag = 0; }
     Complex(double r, double i) { real = r; imag = i; }
     ......
  friend Complex operator + (const Complex& c1,
                             const Complex& c2);
  friend Complex operator + (const Complex& c,
                             double d);
  friend Complex operator + (double d,
                             const Complex& c);
};
```

```
Complex operator + (const Complex& c1,
                    const Complex& c2)
{  return Complex(c1.real+c2.real,c1.imag+c2.imag);
}
Complex operator + (const Complex& c, double d)
{  return Complex(c.real+d,c.imag);
}
Complex operator + (double d, const Complex& c)
//"实数+复数"只能作为全局函数重载。为什么？
{  return Complex(d+c.real,c.imag);
}
......
Complex a(1,2),b(3,4),c1,c2,c3;
c1 = a + b;
c2 = b + 21.5;
c3 = 10.2 + a;
```

单目操作符重载

- 作为成员函数重载
  - 不需要提供参数
  - 定义格式

```
class <类名>
{ ......
  <返回值类型> operator # ();
};
<返回值类型> <类名>::operator # () { ..... }
```

  - 使用格式

```
<类名> a;
#a
或,
a.operator#()
```

例：实现复数的取负操作

```
class Complex
{  ……
   public:
    ……
    Complex operator -() const
    {   return Complex(-real, -imag);
    }
};
……
Complex a(1,2),b;
b = -a;  //把a的负数赋值给b
```

单目操作符重载（续1）

- 作为全局（友元）函数重载
  - 只需要提供一个参数，其类型必须是类、结构、枚举或它们的引用类型
  - 定义格式

    <返回值类型> operator #(<类型> <参数>) { …… }
  - 使用格式为：

    <类型> a;

    #a

    或

    operator#(a)

例：实现判断复数是否为“零”的操作

```
class Complex
{  ……
   public:
     ……
     friend bool operator !(const Complex &c);
};
bool operator !(const Complex &c)
{  return (c.real == 0.0) && (c.imag == 0.0);
}
……
Complex a(1,2);
……
if (!a) //a为0
  ……
……
```

操作符++和-- 的重载

- 单目操作符++（--）：
  - int x,y;
  - ++x;或 x++; //x的值加1
  - x = 0; y = ++x; //x的值为1，y的值为1
  - x = 0; y = x++; //x的值为1，y的值为0
  - ++(++x);或 (++x)++; //
  - ++(x++);或 (x++)++; //

OK，x的值加2。++x为左值表达式

Error。x++为右值表达式

- 重载++（--）时，如果没有特殊处理，它们的后置用法和前置用法共用同一个重载函数。
- 为了能够区分++（--）的前置与后置用法，可以为后置用法再写一个重载函数，该重载函数应有一个额外的int型参数（函数体中可以不使用该参数的值）。

```
class Counter
{    int value;
   public:
     Counter() { value = 0; }
     Counter& operator ++()  //前置的++重载函数
     { value++;
       return *this;
     }
     const Counter operator ++(int)  //后置的++重载函数
     { Counter temp=*this; //保存原来的对象
       value++; //写成：++(*this);更好！调用前置的++重载函数
       return temp; //返回原来的对象
     }
};
……
Counter a,b;
++a;  //使用的是不带参数的操作符++重载函数
a++;  //使用的是带int型参数的操作符++重载函数
b = ++a;  //加一之后的a赋值给b
b = a++;  //加一之前的a赋值给b
++(++a);或 (++a)++; //OK，a加2
++(a++);或 (a++)++; //Error，编译不通过
```