12 Operator Overloading

Hu Sikang
skhu@163.com

Contents

- Introduction :
  - The name of an operator function
  - Why operators need overloading ?
  - Unary, binary, ternary
  - Operators allowing overloaded
  - Operators not allowing overloaded
- Overloading as :
  - Member functions
  - Friend functions
- Special Operators :
  - Conversion, ++/--,=, [ ], ( ), <<

12.0 Function Overloading

- Different functions have the same name (*polymorphism*)
- In C++, a function is identified not only by the *name*, but also by the *number and* the *types of its parameters* and the keyword, *const*, as a member function of a class.

12.0 Function Overloading

```
class complex
{
public:
   complex(double x = 0, double y = 0)
   {     re = x;          im = y;          }

    complex Add(const complex& c)
    {
          double t1 = re+c.re;
          double t2= im+c.im;
          return complex(t1,t2);
    }
private:
    double re, im;
};
```

What's the difference?

complex T(t1, t2);
return T;

```
void main()
{
    complex c,  c1,  c2(5.5, 2);
    c = c1.Add(c2);
```

In fact, it's used to write as follows:

c = c1 + c2;

12.1 Warning & reassurance

- Operator overloading is *only* syntactic sugar, another way of calling a function.
- It is for the code involving your class easier to write and especially easier to *read*.
- All the operators used in expressions that contain only built-in data types cannot be changed. Only an expression containing a user-defined type can have an overloaded operator.

12.2 Syntax

- The name of an operator function is the keyword operator followed by the operator itself.

```
return type   operator @  (argument list)
{
        // code realization
}
```

12.3 Overloadable operators

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

12.3 Operators not Allowing Overloaded

- . member selection
- .* member selection by a pointer
- :: scope resolution
- ?: ternary conditional expression
- sizeof
- typeid

12.3.1 Increment and Decrement

Syntax of increment overloading is as follows:

Prefix: type operator++()
Postfix: type operator++(int)

Prefix: type operator-- ()
Postfix: type operator-- (int)

The *int* argument is used to indicate that the function is to be invoked for postfix application of ++ or --. This *int* is never used; the argument is simply a *dummy* used to distinguish between prefix and postfix application.

12.3.1 Increment and Decrement

```
#include<iostream>
using namespace std;

class  CDate {
public:
   CDate()          {  Year = 2021,  Month = 4,  Day = 6;  }
   void  display()   {  cout << Day << endl;  }
CDate  operator ++()      {  Day++;    return *this; }      // prefix
CDate  operator ++(int)  {  CDate temp;    temp.Day = Day++;    return temp; }  // postfix
private:
   int  Year, Month, Day;
};

void  main( ) {
     CDate   D1,
     D1  =  D++;
     cout << "D

     D2  =  ++D;
     cout << "D = ";   D.display();  cout << "D2 = ";  D2.display();
}
```

It can be written as:
D1 = D.operator ++(int);

It can be written as:
D1 = D.operator ++( );

12.3.2 Assignment

Syntax of assignment overloading is as follows:

```
X &  X :: operator = ( const X & from)
{
        // copy data from from argument
}
```

12.3.2 Assignment

```
#include<iost
using namesp

class  CDate
{
public:
   CDate()
   void  display()    {  cout << Day << endl;  }
CDate   operator ++()      {  Day++;    return *this; }      // prefix
CDate   operator ++(int)  {  CDate temp;    temp.Day = Day++;    return temp; } // postfix
private:
   int  Year, Month, Day;
};

void  main()
{
    CDate   D1,
    D1  =  D++;
    cout << "D = ";   D.display();  cout << "D1 = ";  D1.display(

    D2  =  ++D;
    cout << "D = ";   D.display();  cout << "D2 = ";  D2.display(
}
```

```
void operator = (const CDate& date)
{
      Year = date.Year;
      Month = date.Month;
      Day = date.Day;
}
```

It can be written as:

D1.operator=(D.operator ++(int));

Question: can codes be written as this: *D = D1 = D2;*

12.3.2 Assignment

C++ will give every class a default assignment. So here is a question:

When shall we need define an assignment?

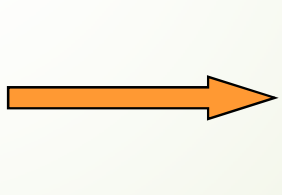
```
#include <iostream>
using namespace std;

class  pointer
{
private:
        int *p;
public:
        pointer(int x)     {   p = new int(x);   }
        ~pointer( )        {   if (p)  delete p;  }
};
```

```
void main()
{
    pointer p(10), q(20);
    q = p;   // Hidden error
}
```

12.3.2 Assignment

Solution:

```
#include <iostream>
using namespace std;
class  pointer
{
private:
        int *p;
public:
        pointer(int x)  {  p = new int(x);   }
        pointer& operator =(const pointer& obj)
        {
            *p = *obj.p;
             return *this;
        }
        ~pointer() {
             if (p)  { if (p)  delete p; }
        }
};
```



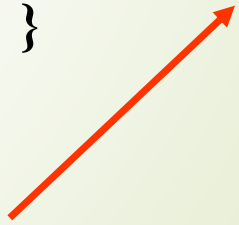
```
void main()
{
        pointer p(10), q(20);
        q = p;       // All right
}
```

```
if  (this != &obj)
        *p = *obj.p;
return *this;
```

12.3 Member and Nonmember Overloading

```
#include <iostream>
using namespace std;
class complex  {
public:
    complex(double x = 0, double y = 0) {
          re = x;     im = y;
    }
    complex  operator +(const complex& a)
  {
          double m = re + a.re;
          double n = im + a.im;
          return complex(m, n);
    }
private:
    double re, im;
};
```

```
void main()
{
    complex x(10, 20);
    complex y(30, 40);
    complex z;

     z = x + y;  //ok
     z = x + 3;  //ok
     z = 3 + x;  //error
}
```



The number, 3, cannot convert to complex.

How can we do?

12.3 Member and Nonmember Overloading

If the constructor is defined as:
explicit complex(double x=0, double y=0)

```
#include <ios
class complex
public:
    complex(double x = 0, double y = 0) {
        re = x;     im = y;
    }
    friend complex operator +(const complex& a, const complex& b);
private:
    double re, im;
};
complex operator +(const complex& a,
                   const complex& b)
{
    double m = a.re + b.re;
    double n = a.im + b.im;
    return complex(m, n);
}
```

```
void main()
{
    complex x(10, 20);
    complex z;

    z = x + 3;  //ok
    z = 3 + x;  //ok
}
```

12.3 Member and Nonmember Overloading

When you define a operator, you do also corresponding operators.

```
#include <iostream.h>
class complex {
public:
   complex(double x = 0, double y = 0) {
         re = x;     im = y;
   }
friend complex operator+(const complex& a, const complex& b) ;
Complex& operator +=(const complex& c);
   void Display() {
         cout << "re = " << re << endl;
         cout << " im = " << im << endl;
   }
private:
   double re, im;
};
complex& complex::operator +=(const complex &c) {
      re += c.re;
      im += c.im;
      return *this;    }
```

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

void main()
{
     complex x(10, 20);
      complex y(30, 40);
      complex z;

       y += x;
       z = x + y;
       z.Display();
}
```

Example 1. Subscripting: [ ]

An *operator [ ]* function can be used to give subscripts a meaning for class objects. The argument (the subscript) of an *operator [ ]* function may be of any type.

Note: **An *operator [ ]* function must be overloaded as member function of class and have only an argument.**

Example 1. Subscripting: [ ]

```
#include <iostream.h>
class vector {
public:
   vector(int size)
        { v = new int[size];  }
   ~vector()
        { if (v != NULL) delete[] v;  }
   int& operator [ ](int i)
        { return v[i];  }
private:
     int *v;
};
```

```
void main()
{
      vector a(5);
      a[2] = 12;
      cout << a[2] << endl;
}
```


a.operator[ ](2) = 12;

Example 2. Function call: ( )

***Function call*, this is, the notation *expression(expression-list)*, can be interpreted as a binary operation with the *expression* as the left-hand operand and the *expression-list* as the right-hand operand.**

Example 2. Function call: ( )

Overloading function call to realize expression:

$$f(x, y) = x * y + 5$$

```
#include <iostream.h>
class F {
public:
        double operator( ) (double x, double y)
        {  return x * y + 5; }
};
void main()
{
        F f;
        cout << f(5.2, 2.5) << endl;
}
```

f.operator() (5.2, 2.5);

Example 2. Function call: ( )

Overloading function call to realize expression:

$$f(x, y) = a * x * y + b$$

```
#include <iostream.h>
class F {
public:
   F(double m, double n)
  {   a = m; b = n;  }
   double operator( ) (double x, double y) const
   {  return a * x * y + b;  }
private:
   double a, b;
};
```

```
void main()
{
      F f(1, 5);
      cout << f(5.2, 2.5);
}
```

Example 3. ostream: <<

The *operator* << can be defined as a binary operator. In general, the *operator* << is defined as a friend member function of class and has two arguments: one is the reference of ostream, the other is an object.

```
class complex {
public:
    complex(double x = 0, double y = 0)
    {  re = x;     im = y;  }
    void Display( )
     { cout << re << "+" << im << "i" << endl;  }
private:
    double re, im;
};
```

Example 3. ostream: <<

```
#include <iostream.h>
class complex {
public:
    complex(double x = 0, double y = 0)
    {   re = x;     im = y;   }
    friend ostream& operator <<(ostream& os, const complex& a);
private:
    double re, im;
};
ostream& operator <<(ostream& os, const complex& a)
{
    os << a.re << " + " << a.im << "i" << endl;
    return os;
}
```

```
void main()
{
        complex obj(10, 20);
        cout << obj;
}
```