12 Operator Overloading

Hu Sikang
skhu@163.com

Contents

- Introduction :
  - Operator overloaded function
  - Unary, binary, ternary
- 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;
};
```

re + im*i*

```
int main()
{
    complex c,  c1,  c2(5.5, 2);
    c = c1.Add(c2);
    retrun 0;
}
```

It’s better to write as follows:
c = c1 + c2;

12.1 Warning & reassurance

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

Prefix:    return type  operator-- ( )
Postfix:   return 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 = 2024,   Month = 3,   Day = 25;  }
    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;
};

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

   D2 = ++D;
   cout << "D = ";   D.display();      cout << "D2 = ";  D2.display();
   return 0;
}
```

It can be written as:
D1 = D.operator ++(0);

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

12.3.2 Assignment

Syntax of assignment overloading is as follows:

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

12.3.2 Assignment

```
#include<iostre
using namespac

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;
};

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

  D2 = ++D;
  cout << "D = ";   D.display();  cout << "D2 = ";  D2.display();
  return 0;
}
```

```
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: ***D2 = D1 = D;***

12.3.2 Assignment

C++ will give every class a default assignment.

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 != nullptr)  delete p;   }
};
```

```
int main()
{
     pointer p(10), q(20);
     q = p;    // Hidden error
     return 0;
}
```

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 != nullptr)   delete p;
            }
};
```

```
pointer(const pointer& obj) {
    p = new int(*obj.p);
}
```

```
int main()
{
       pointer p(10), q(20);
       q = p;       // All right
       return 0;
}
```

```
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;
};
```

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

    z = x + y;  //ok
    z = x + 3;  //ok
    z = 3 + x;  //error
    return 0;
}
```

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 <iostream>
using namespace std;
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);
}
```

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

      z = x + 3;  //ok
      z = 3 + x;  //ok
      return 0;
}
```

12.3 Member and Nonmember Overloading

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

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

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

                         y += x;
                         z = x + y;
                         z.Display();
                         return 0;
                     }
```

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>
using namespace std;
class vector {
public:
   vector(int s)  { v = new int[s]; capacity = s;   size = 0;  }
   ~vector()      {  if (v != nullptr)    delete[] v;    }
   int& operator [ ](int i)     {    return v[i];    }
private:
      int *v;
      int capacity;  // number of elements' storage
      int size;          // number of current element
};
```

```
int main()
{
      vector a(5);
      a[2] = 12;
      cout << a[2] << endl;
      return 0;
}
```

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; }
};
int main()
{
        F f;
        cout << f(5.2, 2.5) << endl;
        return 0;
}
```

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>
using namespace std;
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;
};
```

```
int main()
{
       F f(1, 5);
       cout << f(5.2, 2.5);
       return 0;
}
```

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;
};
```

```
complex C(10, 20);
cout << C << endl;
```

Example 3.   ostream: <<

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

```
int main( )  {
        complex C(10, 20);
        cout << C << endl;
        return 0;
}
```

Example 4. Dereferencing

The dereferencing *operator ->* can be defined as a unary postfix operator. In general, the dereferencing *operator ->* hasn’t argument.

Example 4. Dereferencing

```
#include <iostream>
using namespace std;
class Student
{     public:        int age, ID;     };

class Prt_Rec  //  Define a pointer to Student
{
public:
     Prt_Rec()
     {    S = new Student;    S->age = 0;   S->ID = 0; }
     Student* operator ->( )     {    return S;  }
      ~Prt_Rec( )    {   delete S;   }
public:
     Student *S;
};
```

```
int main()
{
     Prt_Rec  PR;
     PR->age = 20;
     PR->ID = 001;
     return 0;
}
```

```
(PR.operator->( )) ->age = 20;
(PR.operator->( )) ->ID = 001;
```