南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

Lab 10, operator overloading

廖琪梅，王大兴

Operator overloading and friend function

- Operator overloading
- Returning object
- Conversion of class

Operator overloading

To overload an operator, use a special function form called an **operator function**.

return_type operator op(argument-list)

op is the symbol for the operator being overloaded

An operator function must either be a member function of a class or have at least one parameter of class type.

member function, non-member function, friend function

Only a non-member operator overloading function can **implement type conversion** on its **left argument**, so if a function need convert type on its left argument, define the function as non-member function; if the function must get the non-public members of the class, define it as a friend function of the class.

Other cases beyond the above, define the function as a member function.

The assignment (=)operators must be defined as member function. However, IO operators(<< and >>) must be non-member functions.

Returning object

When a member function or standard function returns an object, you have choices. The function could return an **object** , or return a **reference to an object**.

1. Returning an object

```
Complex Complex::operator+(const Complex &rhs) const
{
    Complex result;
    result.real = real + rhs.real;
    result.imag = imag + rhs.imag;

    return result;
}
```

When a function returns an object, a temporary object will be created. There is the added expense of calling the copy constructor to create the temporary object, it is less efficient. It is invisible and does not appear in your source code. The temporary object is automatically destroyed when the function call terminates.

```
Complex& Complex::operator+(const Complex &rhs) const
{
    Complex result;
    result.real = real + rhs.real;
    result.imag = imag + rhs.imag;

    return result;
}
```

Do not return the reference of a local object, because when the function terminates, the reference would be a reference to a non-existent object.

```
complexclass.cpp: In member function ‘Complex& Complex::operator+(const Complex&)’:
complexclass.cpp:20:12: warning: reference to local variable ‘result’ returned [-Wreturn-local-addr]
   20 |     return result;
      |            ^~~~~~
complexclass.cpp:16:13: note: declared here
   16 |     Complex result;
      |             ^~~~~~
```

```
Complex& Complex::operator+(const Complex& rhs)
{
    this->real += rhs.real;
    this->imag += rhs.imag;
    return *this;
}
```

Returning a reference to **this object** works efficiently, but it modifies the values of the data member of **this object**.

```
Complex Complex::operator+(const Complex& rhs)
{
    double re = real + rhs.real;
    double im = imag + rhs.imag;
    return Complex(re, im);
}
```

This return style is known as return constructor argument. By using this style instead of returning an object, the compiler can eliminate the cost of the temporary object. This even has a name: the ***return value optimization***.

2. Returning a reference to an object

```
// version 1
Vector Max(const Vector& v1, const Vector& v2)
{
    if(v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

```
// version 2
const Vector& Max(const Vector& v1, const Vector& v2)
{
    if(v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

- Returning an object invokes the copy constructor, whereas returning a reference doesn't. Thus version 2 does less work and is more efficient.
- The reference should be to an object that exists when the calling function is executing.
- Both v1 and v2 are declared as being const references, so the return type has to be const to match.

Two common examples of returning a non-const object are overloading the **assignment operator** and overloading the **<< operator** for use with **cout**.

Conversion of class

1. Implicit Class-Type Conversions

Every constructor that can be called with **a single argument** defines an implicit conversion to a class type. Such constructors are sometimes referred to as ***converting constructors***.

```
class Circle
{
private:
    double radius;

public:
    Circle() : radius(1) { }

    Circle(double r) : radius(r) { }
```

Converting constructor

```
circle.cpp > ...
1   #include <iostream>
2   #include "circle.h"
3
4   int main()
5   {
6       Circle r1;
7
8       Circle r2 = 3;
9
10      Circle r3(10);
11
12      r3 = 4;
13
14      std::cout << r1 << std::endl;
15      std::cout << r2 << std::endl;
16      std::cout << r3 << std::endl;
17
18      return 0;
19  }
```

Convert int to Circle type

Convert int to Circle type

when we use the copy form of initialization or assignment (with an =), implicit conversions happens.

rational > rational.h > ...

```
1   #pragma once
2   #include <iostream>
3
4   class Rational
5   {
6   private:
7       int numerator;
8       int denominator;
9
10  public:
11      Rational(int n = 0, int d = 1) : numerator(n),denominator(d) { }
12
13      int getN() const  {  return numerator; }
14
15      int getD() const {   return denominator; }
16
17      friend std::ostream& operator <<(std::ostream &os, const Rational &rhs)
18      {
19          os << rhs.numerator << "/" << rhs.denominator;
20          return os;
21      }
22  };
23
24  const Rational operator * (const Rational &lhs, const Rational &rhs)
25  {
26      return Rational(lhs.getN() * rhs.getN(), lhs.getD() * rhs.getD());
27  }
```

Constructor with default arguments works as a converting constructor.

We define the operator * as a normal function not a friend function of the Rational class.

rational > rational.cpp > ...

```
1   #include <iostream>
2   #include "rational.h"
3
4   using namespace std;
5
6   int main()
7   {
8       Rational a = 10;
9       Rational b(1,2);
10
11      Rational c = a * b;
12      cout << "c = " << c << endl;
13
14      Rational d = 2 * a;
15      cout << "d = " << d << endl;
16
17      Rational e = b * 3;
18      cout << "e = " << e << endl;
19
20      Rational f = 2 * 3;
21      cout << "f = " << f << endl;
22
23      return 0;
24  }
```

Convert int to Rational type

```
c = 10/2
d = 20/1
e = 3/2
f = 6/1
```

Use explicit to supper the implicit conversion

We can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor as ***explicit***:

```
class Circle
{
private:
    double radius;

public:
    Circle() : radius(1) { }

    explicit Circle(double r) : radius(r) { }
```

Turn off implicit conversion

circle.cpp > ...

```
1   #include <iostream>
2   #include "circle.h"
3
4   int main()
5   {
6       Circle r1;
7
8       Circle r2 = 3;
9
10      Circle r3(10);
11
12      r3 = 4;
13
14      std::cout << r1 << std::endl;
15      std::cout << r2 << std::endl;
16      std::cout << r3 << std::endl;
17
18      return 0;
19  }
```

Can not do the implicit conversion

```
Circle r2 = (Circle)3;
```

```
r3 = static_cast<Circle>(4);
```

Use these two styles for explicit conversion

2. Conversion function

Conversion function is a member function with the name ***operator*** followed by a type specification, no return type, no arguments.

operator **typeName( );**

```
class Rational
{
private:
    int numerator;
    int denominator;

public:
    Rational(int n = 0, int d = 1) : numerator(n),denominator(d) { }

    int getN() const
    {
        return numerator;
    }

    int getD() const
    {
        return denominator;
    }

    operator double() const
    {
        return numerator/denominator;
    }
};
```

conversion function

```
Rational a(10,2);
double d = 0.5 + a;
```

Convert Rational object **a** to **double** by conversion function

```
Rational a(10,2);
double d = 0.5 + (double)a;
```

Declare a conversion operator as explicit for calling it explicitly

```
explicit operator double() const
{
    return numerator/denominator;
}
```

Caution: You should use implicit conversion functions with care. Often a function that can only be invoked explicitly is the best choice.

Exercises

Continue improving the Complex class and adding more operations for it, such as: **-, *, ~, ==, !=** etc. Make the following program run correctly.

```cpp
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    Complex a(3, 4);
    Complex b (2,6);

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "~b = " << ~b << endl;
    cout << "a + b = " << a+b << endl;
    cout << "a - b = " << a-b << endl;
    cout << "a - 2 = " << a-2 << endl;
    cout << "a * b = " << a*b << endl;
    cout << "2 * a = " << 2*a << endl;
    cout << "============================" << endl;

    Complex c = b;
    cout << "c = " << c << endl;
    cout << "b == c? " << boolalpha << (b==c) << endl;
    cout << "b != c? " << (b!=c) << endl;
    cout << "a == b? " << (a==b) << endl;
    cout << "============================" << endl;
    Complex d;
    cout << "Enter a complex number(real part and imaginary part):";
    cin >> d;
    cout << "Before assignment, d = " << d << endl;
    d = c;
    cout << "After assignment, d = " << d << endl;

    return 0;
}
```

Note that you have to overload the **<< and >> operators**. Use reference to object and const whenever warranted. A sample runs might look like this:

```
a = 3+4i
b = 2+6i
~b = 2-6i
a + b = 5-2i
a - b = 1+10i
a - 2 = 1+4i
a * b = 30-10i
2 * a = 6+8i
============================
c = 2-6i
b == c? true
b != c? false
a == b? false
============================
Enter a complex number(real part and imaginary part):4 6
Before assignment, d = 4+6i
After assignment, d = 2-6i
```