C/C++ Program Design

LAB 11

CONTENTS

- Learn **operator overloading**
- Learn **Friend functions**
- Learn how to overload the **<< operator** for output
- Learn the conversion of class

2 Knowledge Points

2.1 Operator overloading

2.2 Friend functions

2.3 Overloading the <<operator for output

2.4 Conversion of class

2.1 Operator Overloading

In C++, the overloading principle applies **not only to functions**, **but to operator**. Operators can be extended to work **not just with built-in types**, **but also classes**.

Overloaded operators are functions with special names: the keyword ***operator*** followed by the **symbol for the operator being defined**. Like any other function, an overloaded operator has a return type, a parameter list, and a body.

operator op(argument-list)

op is the symbol for the operator being overloaded

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

using the + symbol to add two Complex objects

complex.h

```
#include <iostream>
#ifndef COMPLEX_H
#define COMPLEX_H
class Complex
{
private:
    double real;
    double imag;

public:
    Complex() : real(1), imag(1) { }
    Complex(double re, double im)
    {
        real = re;
        imag = im;
    }

    Complex operator+(const Complex &rhs);

    void Show() const;
};

#endif
```

complexclass.cpp

```
#include <iostream>
#include "complex.h"

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

    return *this;
}

void Complex::Show() const
{
    std::cout <<  real << (imag >= 0? "+":"") << imag << "i";
}
```

Operator overloading works as a function

complexmain.cpp > main()

```
4   int main()
5   {
6       Complex c1;
7       Complex c2(2,-4);
8
9       c1.Show();
10      std::cout << " + ";
11      c2.Show();
12
13      Complex c = c1 + c2;
14
15      std::cout << " = ";
16      c.Show();
17      std::cout << std::endl;
18
19      std::cout << "Done." << std::endl;
20
21      return 0;
22  }
```

Operator overloading

The **left operand** is the invoking object, the **right operand** is the one passed the argument.

Output:

```
1+1i + 2-4i = 3-3i
Done.
```

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

    return result;
}
```

You can return local object to the caller

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.

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

    return result;
}
```

```
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;
      |             ^~~~~~
```

Consider this case: compute the addition of a complex and a numeric number

```
Complex c = c1 + 2;
```

If an operator function is a member function, the first (left-hand) operand is the invoking object. So we can write another overloaded addition operator function with a double parameter as follows:

```
Complex operator+(double n) const;
```

The definition of the function is:

```
Complex Complex::operator+(double n) const
{
    Complex result;
    result.real = this->real + n;
    result.imag = this->imag;

    return result;
}
```

or

```
Complex Complex::operator+(double n) const
{
    double re = this->real + n;
    double im = this->imag;

    return Complex(re,im);
}
```

When a function returns an object, a temporary object will be created. It is invisible and does not appear in your source code. The temporary object is automatically destroyed when the function call terminates.

rational.h > ...

```cpp
#include <iostream>
#pragma once

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;
    }

    void show() const
    {
        std::cout << numerator << "/" << denominator << std::endl;
    }
};

const Rational operator * (const Rational& lhs, const Rational& rhs)
{
    return Rational(lhs.getN() * rhs.getN(), lhs.getD() * rhs.getD());
}
```

Define the overloading operator function as a nonmember function. Use the getter to access the data of Rational class.

rational.cpp > ...

```cpp
#include "rational.h"

int main()
{
    Rational a = 10;
    Rational b(1,2);

    Rational c = a * b;

    c.show();

    return 0;
}
```

The total cost of temporary objects as a result of your calling operator* is zero: no temporaries are created. Instead, you pay for only one constructor call, the one to create c.

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. This even has a name: the ***return value optimization***.

How about the following case?

```
Complex c = 2 + c1;
```

The compiler can not find the correspond member function.

Conceptually, **2 + c1** should be the same as **c1 + 2** , but the first expression can not match any member function because 2 is not a Complex object.

Remember, **the left operand is the invoking object**, but 2 is not an object. So the compiler cannot replace the expression with a member function call.

In this case, only nonmember overloading operator function can be used. A nonmember function is not invoked by an object. But nonmember functions can’t directly access private data in a class. This time we use **friend function** to solve this problem.

2.2 Friend Function

If a function is defined as a **friend function** of a class, it has the same access privileges as a member function of the class. This means a friend function can access all the **private** and **protected** data of that class.

By using the keyword **friend** compiler knows the given function is a friend function.

Friend Function in C++

```
class ClassName
{
    .......
    // friend function declaration

    friend return_type functionName(parameter list);

};

return_type functionName(parameter list)
{
    ....    /* private and protected data of
               ClassName can be accessed form
               this function because it is a
               friend function
            */
}
```

The friend function prototype is preceded by keyword **friend,** and is declared in the class.

The function can be defined anywhere in the program like a normal C++ function. **The function definition does not use either the keyword friend or scope resolution operator.**

The first step toward creating a friend function is to place a prototype in the class declaration and prefix the prototype with the keyword **friend**:

```
friend Complex operator +(double r, const Complex& other);
```

This prototype has two implications:

- Although the **operator +() function** is declared in the class declaration, it is not a member function. So it isn't invoked by using the membership operator.
- Although the **operator +() function** is not a member function, it has same access rights as a member function.

complex.h > ...

```
1   #include <iostream>
2   #ifndef COMPLEX_H
3   #define COMPLEX_H
4   class Complex
5   {
6   private:
7       double real;
8       double imag;
9
10  public:
11      Complex() : real(1), imag(1) { }
12      Complex(double re, double im) : real(re),imag(im) { }
13
14      Complex operator+(const Complex &rhs) const;
15      Complex operator+(const Complex &rhs);
16
17      Complex operator+(double n) const;
18
19      void Show() const;
20
21      friend Complex operator+(double n, Complex &rhs);
22  };
23
24  #endif
```

friend function declaration in Complex class definition

The second step is to write the function definition. Because it is not a member function, don't use the **Complex::** qualifier. Also you need not use the **friend** keyword in the definition.

```
Complex operator+(double n, Complex &rhs)
{
    double re = n + rhs.real;
    double im = rhs.imag;

    return Complex(re, im);
}
```

or

```
Complex operator+(double n, Complex &rhs)
{
    return rhs + n;
}
```

complexmain.cpp > ...

```cpp
#include <iostream>
#include "complex.h"

int main()
{
    Complex c1;

    Complex c = 2 + c1;

    std::cout << 2  ;
    std::cout << " + ";
    c1.Show();

    std::cout << " = ";
    c.Show();
    std::cout << std::endl;

    std::cout << "Done." << std::endl;

    return 0;
}
```

With the nonmember overloaded operator function, the **left operand** of an operator expression corresponds to the first argument of the operator function, and the **right operand** corresponds to the second argument.

```cpp
Complex operator+(double n, Complex &rhs)
{
    double re = n + rhs.real;
    double im = rhs.imag;

    return Complex(re, im);
}
```

2.3 Overloading the << operator for output

One very useful feature of classes is that you can overload the **<< operator**, so that you can use it with **cout** to **display an object's contents**.

Suppose **a** is a **Complex object**, to display Complex values, we've been using:

a.Show();

```
void Complex::Show() const
{
    std::cout <<  real << (imag >= 0? " + ":"") << imag << "i";
}
```

Can we use **cout << a;** to display Complex value?

The First Version of Overloading <<

If you use a **Complex** member function to overload **<<** , the **Complex** object would come first, the display's style is like **c << cout**; not cout << c;. So we choose to overload the operator by using a **friend function**:

```
friend void operator << (std::ostream &os, const Complex &c);
```

friend function declaration

```
void operator << (std::ostream &os, const Complex &c)
{
    os << c.real << (c.imag >= 0? " + ":"") << c.imag << "i" << std::endl;
}
```

friend function definition

But the implementation doesn't allow you to combine the redefined **<< operator** with ones cout normally uses:

cout << a << "\n"; // can't do

The Second Version of Overloading <<

We revise the operator<<( ) function so that it returns a reference to an ostream object:

```
friend std::ostream& operator << (std::ostream &os, const Complex &c);
```

friend function declaration

```
std::ostream& operator << (std::ostream &os, const Complex &c)
{
    os << c.real << (c.imag >= 0? " + ":"") << c.imag << "i";
    return os;
}
```

friend function definition

Ordinarily, the first parameter of an output operator is a reference to a nonconst ostream object. The ostream is nonconst because writing to the stream changes its state. The parameter is a reference because we cannot copy an ostream object.

The second parameter ordinarily should be a reference to const of the class type we want to print. The parameter is a reference to avoid copying the argument. It can be const because (ordinarily) printing an object does not change that object. To be consistent with other output operators, operator<< normally returns its ostream parameter.

2.4 Conversion of class

2.4.1 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***.

```cpp
class Circle
{
private:
    double radius;

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

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

Converting constructor

circle.cpp > ...

```cpp
#include <iostream>
#include "circle.h"

int main()
{
    Circle r1;

    Circle r2 = 3;

    Circle r3(10);

    r3 = 4;

    std::cout << r1 << std::endl;
    std::cout << r2 << std::endl;
    std::cout << r3 << std::endl;

    return 0;
}
```

Convert int to Circle type

Convert int to Circle type

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

```
Radius=1,Area=3.14
Radius=3,Area=28.26
Radius=4,Area=50.24
```

rational.h > ...

```cpp
#include <iostream>
#pragma once

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;
    }

    void show() const
    {
        std::cout << numerator << "/" << denominator << std::endl;
    }
};

const Rational operator * (const Rational& lhs, const Rational& rhs)
{
    return Rational(lhs.getN() * rhs.getN(), lhs.getD() * rhs.getD());
}
```

Constructor with default arguments works as a converting constructor

rational.cpp > ...

```cpp
#include "rational.h"

int main()
{
    Rational a = 10;
    Rational b(1,2);

    Rational c = a * b;

    c.show();

    return 0;

}
```

Convert int to Rational type

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.cpp > ...

```cpp
#include <iostream>
#include "circle.h"

int main()
{
    Circle r1;

    Circle r2 = Circle(3);

    Circle r3(10);

    r3 = (Circle)4;

    std::cout << r1 << std::endl;
    std::cout << r2 << std::endl;
    std::cout << r3 << std::endl;

    return 0;
}
```

Use these two styles for explicit conversion

2.4.2 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 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.

3 Exercises

1. 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( re: 3.0,  im: 4.0);
    Complex b( re: 2.0,  im: 6.0);

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

    Complex c = b;
    cout << "b == c is " << (b == c) << endl;
    cout << "b != c is " << (b != c) << endl;
    cout << "a == c is " << (a == c) << endl << endl;

    Complex d;
    cout << "Enter a complex number: " << endl;
    cin >> d;
    cout << "d is " << d << endl;

    return 0;
}
```

Note that you have to overload the << and >> operators. Use const whenever warranted.

A sample runs might look like this:

```
a is 3 + 4i
b is 2 + 6i
~b is 2 - 6i
a + b is 5 + 10i
a - b is 1 - 2i
a - 2 is 1 + 4i
a * b is -18 + 26i
2 * a is 6 + 4i
b == c is true
b != c is false
a == c is false

Enter a complex number:
real:4
imaginary:-6
d is 4 - 6i
```

2. Design a class named **Number** that stores a number which is integer and private to the class. The default constructor should set the number to 0. Add another constructor that allows the caller to set the number. Finally, overload the prefix and postfix ++ and - - operators to make the following program run correctly.

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

int main()
{
    Number n1( num: 20);
    Number n2 = n1++;
    cout << "n1 = " << n1 << endl;
    cout << "n2 = " << n2 << endl;

    Number n3 = n2--;
    cout << "n2 = " << n2 << endl;
    cout << "n3 = " << n3 << endl;

    Number n4 = ++n3;
    cout << "n3 = " << n3 << endl;
    cout << "n4 = " << n4 << endl;

    Number n5 = --n4;
    cout << "n4 = " << n4 << endl;
    cout << "n5 = " << n5 << endl;

    return 0;
}
```

A sample runs might look like this:

```
n1 = 21
n2 = 20
n2 = 19
n3 = 20
n3 = 21
n4 = 21
n4 = 20
n5 = 20
```

Note that you have to overload the << operators. Use const whenever warranted.

Hips:

Syntax for overloading postfix increment operator is as follows:

```
return-type operator ++(int)
{
    //Body of function
    ...
}
```

Syntax for overloading prefix increment operators is as follows:

```
return-type operator ++( )
{
    //Body of the function
    ...
}
```