C/C++ Program Design

LAB 11

CONTENTS

- Learn operator overloading
- Learn Friend functions
- Learn how to overload the << operator for output

2 Knowledge Points

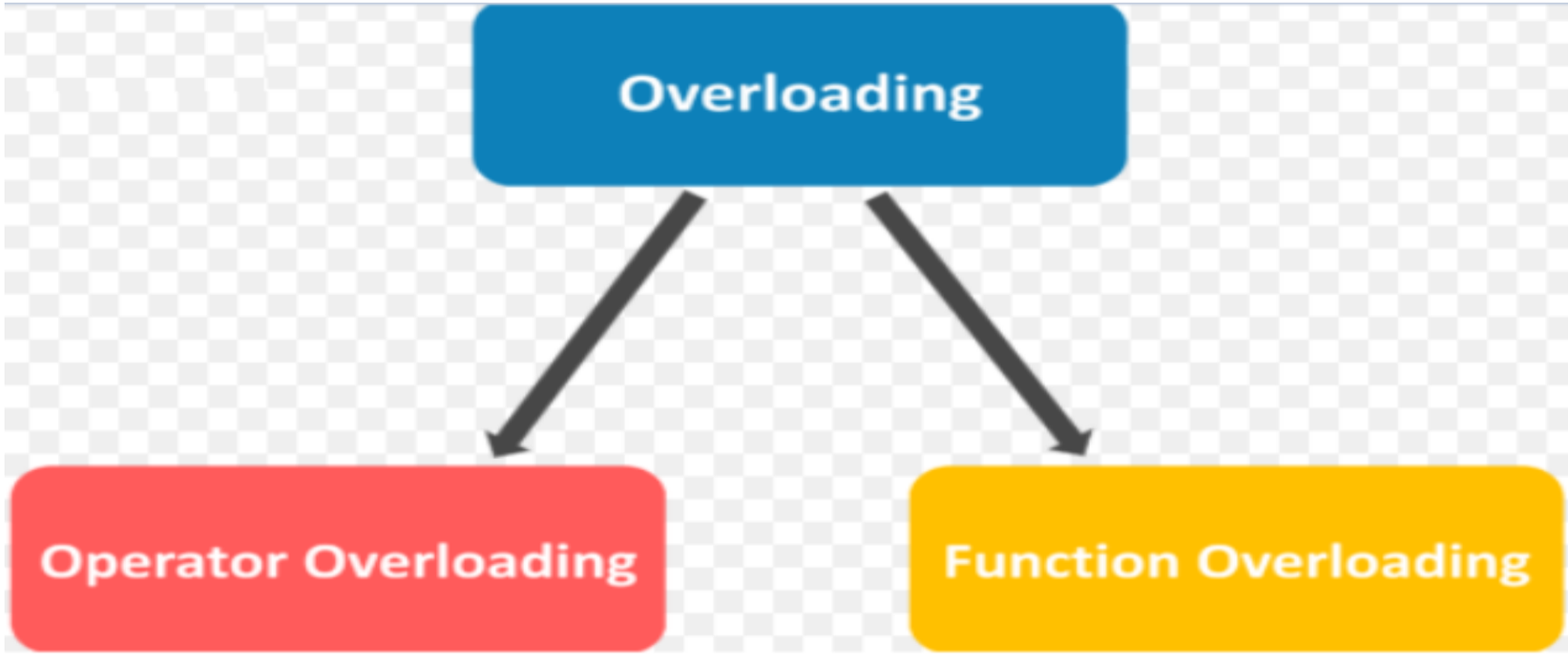
2.1 Operator overloading

2.2 Friend functions

2.3 Overloading the <<operator for output

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**.

Example: Addition of complex numbers

```cpp
// complex.h -- Complex class without operator overloading
#ifndef _MYCOMPLEX_H
#define _MYCOMPLEX_H

class Complex {
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double re, double im);
    Complex Add(const Complex& data);
    void Show() const;
};

#endif //_MYCOMPLEX_H
```

```cpp
//complex.cpp --- implementing Complex methods
#include <iostream>
#include "complex.h"

Complex::Complex() : real(0), imag(0)
{
}
Complex::Complex(double re, double im) : real(re), imag(im)
{
}
Complex Complex::Add(const Complex& data)
{
    Complex sum;
    sum.real = data.real + this->real;
    sum.imag = data.imag + this->imag;
    return sum;
}
void Complex::Show() const
{
    std::cout << real << " + " << imag << "i" << std::endl;
}
```

real = 0; imag = 0;

initialization list

real = re; imag = im;

const indicates that the data of the class would not be modified by the function. Only **member functions** can use **const** after parentheses.

```cpp
//complex_test.cpp --- using the first draft of the Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    // Complex class no operator overloading
    Complex a(3.0, 4.0);
    Complex b(5.0, 6.0);
    Complex sum = a.Add(b);
    sum.Show();

    cout << "Done!" << endl;
    return 0;
}
```

The addition of two complex data by calling function.

Output:

```
8 + 10i
Done!
```

How can we deal with addition of Complex like build-in types?

Complex sum = a + b ?

Operator overloading

Operator overloading extends the overloading concept to operators, letting you assign multiple meanings to C++ operators.

C++ lets you extend operator overloading to user-defined types, permitting you use the **+** symbol to add two objects. The compiler uses the number and type of operands to determine which definition of addition to use.

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

operator op(argument-list)

op is the symbol for the operator being overloaded

Adding an Addition Operator

Convert the Complex class to using an overloaded addition(+) operator.

```cpp
// complex.h -- Complex class with operator overloading
#ifndef _MYCOMPLEX_H
#define _MYCOMPLEX_H

class Complex {
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double re, double im);


    Complex operator +(const Complex & other) const;
    void Show() const;
};

#endif //_MYCOMPLEX_
```

```cpp
//complex.cpp --- implementing Complex methods
#include <iostream>
#include "complex.h"

Complex::Complex() : real(0), imag(0)
{
}
Complex::Complex(double re, double im) : real(re), imag(im)
{
}
Complex Complex::operator +(const Complex & other) const
{
    double result_real = real + other.real;
    double result_imag = imag + other.imag;
    return Complex (result_real, result_imag);
}

void Complex::Show() const
{
    std::cout << real << " + " << imag << "i" << std::endl;
```

Operator overloading works as function

```cpp
//complex_test.cpp --- using the operator overloading of the Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    // Complex class with operator overloading
    Complex a(3.0, 4.0);
    Complex b(5.0, 6.0);
    Complex sum = a + b;

    sum.Show();

    cout << "Done!" << endl;
    return 0;
}
```

Operator overloading

Output:

```
8 + 10i
Done!
```

Operator function

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

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

    return result;
}
```

```
: warning C4172: returning address of local variable or temporary: result
```

Consider this case: Adding a Complex object by a double value

```
//complex_test.cpp --- using the operator overloading of the Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    Complex a(3.0, 4.0);
//    Complex b(5.0, 6.0);
    Complex sum = a + 2;

    sum.Show();

    cout << "Done!" << endl;
    return 0;
}
```

The **left operand is the invoking object**, so we can write another overloaded addition operator function to solve the problem

Complex operator+(double real) const;

```cpp
// complex.h -- Complex class with operator overloading
#ifndef _MYCOMPLEX_H
#define _MYCOMPLEX_H

class Complex {
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double re, double im);


    Complex operator +(const Complex & other) const;
    Complex operator +(double real)const;

    void Show() const;
};

#endif //_MYCOMPLEX_H
```

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

    return result;
}


Complex Complex::operator +(double real) const
{
    double result_real = real + this->real;
    double result_imag = this->imag;
    return Complex(result_real, result_imag);
}


void Complex::Show() const
{
    std::cout << real << " + " << imag << "i" << std::endl;
}
```

But what about the following statement?

```
Complex sum = 2.0 + a;   // compiler can not find the correspond member function
```

Conceptually, **2.0 + a** should be the same as **a+2.0** , but the first expression can not correspond to a member function because 2.0 is not a type Complex object.

Remember, **the left operand is the invoking object**, but 2.0 is not an object. So the compiler cannot replace the expression with a member function call.

We can use **friend function** to solve this problem.

2.2 Friend Function

If a function is defined as a **friend function** of a class, then that function can access all the **private** and **protected** data.

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.

```cpp
//complex.h --- Complex class after adding friend function
#ifndef _FRIEND_OPERATOR_COMPLEX_H
#define _FRIEND_OPERATOR_COMPLEX_H

class Complex {
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double re, double im);


    Complex operator +(const Complex& other) const;
    Complex operator +(double real) const;

    void show() const;

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


};

#endif //_FRIEND_OPERATOR_COMPLEX_H
```

friend function declaration in Complex class definition

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

```cpp
// complex.cpp --- implementing Complex class methods
#include <iostream>
#include "complex.h"

Complex::Complex() : real(0), imag(0)
{
}

Complex::Complex(double re, double im) : real(re), imag(im)
{
}

Complex Complex:: operator+(double real)const
{
    double result_real = real + this->real;
    double result_imag = this->imag;
    return Complex(result_real, result_imag);
}
Complex Complex::operator +(const Complex& other) const
{
    double result_real = this->real + other.real;
    double result_imag = this->imag + other.imag;
    return Complex(result_real, result_imag);
}

void Complex::show() const
{
    std::cout << real << " + " << imag << "i" << std::endl;
}

Complex operator +(double real, const Complex& other)
{
    double result_real = real + other.real;
    double result_imag = other.imag;
    return Complex(result_real, result_imag);
}
```

friend function definition

```cpp
//test.cpp --- using the friend function Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    // Complex class after using friend function
    Complex a(3.0, 4.0);
    Complex sum = 2.0 + a;

    sum.Show();

    cout << "Done!" << endl;
    return 0;
}
```

Output:

```
5 + 4i
Done!
```

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 << "i" << std::endl;
}
```

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, this would be confusing. So **we choose to overload the operator by using a friend function**:

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

friend function declaration

```
void operator<< (std::ostream& os, const Complex& other)
{
    os << other.real << " + " << other.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& other);
```

friend function declaration

```
std::ostream & operator<< (std::ostream& os, const Complex& other)
{
    os << other.real << " + " << other.imag << "i" << std::endl;
    return os;
}
```

friend function definition

```cpp
//complex.h --- Complex class after adding friend function
#ifndef _FRIEND_OPERATOR_COMPLEX_H
#define _FRIEND_OPERATOR_COMPLEX_H

class Complex {
private:
    double real;
    double imag;

public:
    Complex();
    Complex(double re, double im);


    Complex operator +(const Complex& other) const;
    Complex operator +(double real) const;

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

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


};

#endif //_FRIEND_OPERATOR_COMPLEX_H
```

```cpp
// complex.cpp --- implementing Complex class methods
#include <iostream>
#include "complex.h"

Complex::Complex() : real(0), imag(0)
{
}

Complex::Complex(double re, double im) : real(re), imag(im)
{
}

Complex Complex:: operator+(double real)const
{
    double result_real = real + this->real;
    double result_imag = this->imag;
    return Complex(result_real, result_imag);
}
Complex Complex::operator +(const Complex& other) const
{
    double result_real = this->real + other.real;
    double result_imag = this->imag + other.imag;
    return Complex(result_real, result_imag);
}
Complex operator +(double real, const Complex& other)
{
    double result_real = real + other.real;
    double result_imag = other.imag;
    return Complex(result_real, result_imag);
}

std::ostream& operator<< (std::ostream& os, const Complex& other)
{
    os << other.real << " + " << other.imag << "i" << std::endl;
    return os;
}
```

```cpp
//test.cpp --- using the friend function Complex class
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    // Complex class after using friend function
    Complex a(3.0, 4.0);
    Complex sum = 2.0 + a;

    cout << sum << endl;

    cout << "Done!" << endl;
    return 0;
}
```

Output:

```
5 + 4i

Done!
```