Advanced Programming

Lab 10, Class(2): operator overloading

廖琪梅，王大兴，于仕琪，王薇

Topic

- **Operator overloading in C++**
  - **Member** function vs **Non-member** function(e.g. **Friend function**)
  - Returning **object** vs Returning **reference**
  - Conversion of class
    - **implicit conversion** vs **explicit conversion**
    - conversion function

- **‘Operateor overloading’ in Python**
  - x + y : type(x).__add__(x, y)
  - x + y: type(y).__radd__(y, x)
  - __str__()

- **Exercises**

Operator overloading in C++

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

Non-member operator overloads are required to enable type conversion on the left operand. If a function needs such conversion, define it as a non-member. If it also requires access to non-public members, declare it as a friend of the class.

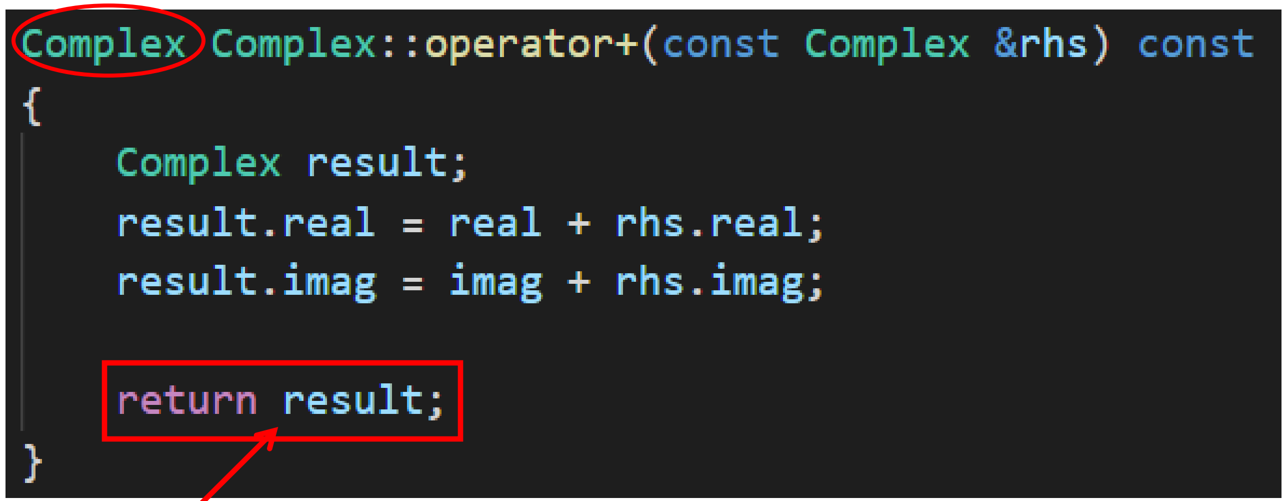
Other cases beyond the above, define the function as a member function.

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

Returning object

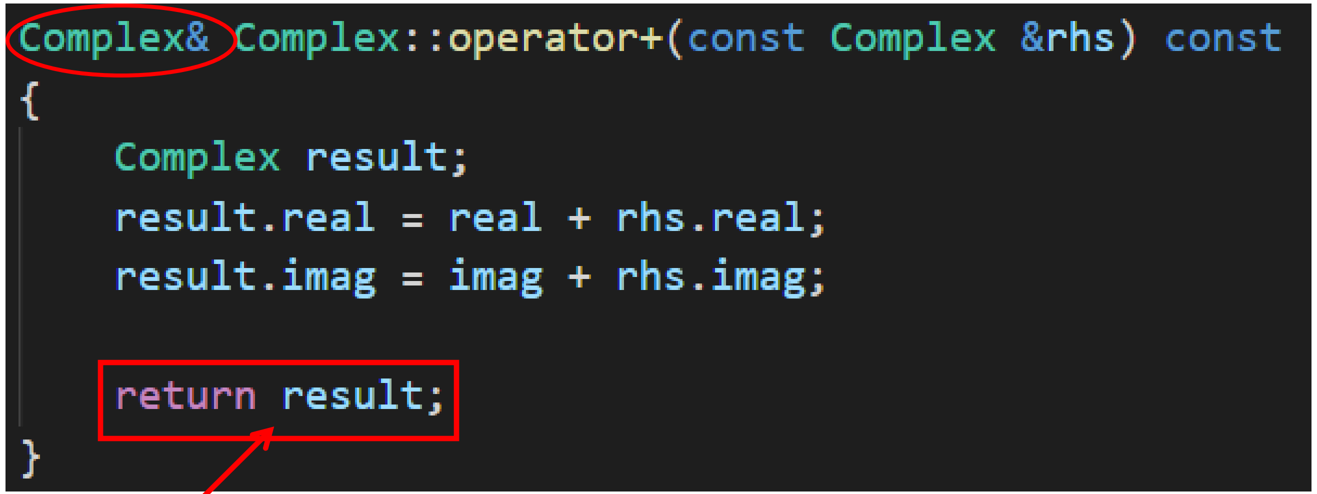
Return the value, or the reference?

1. Returning an object



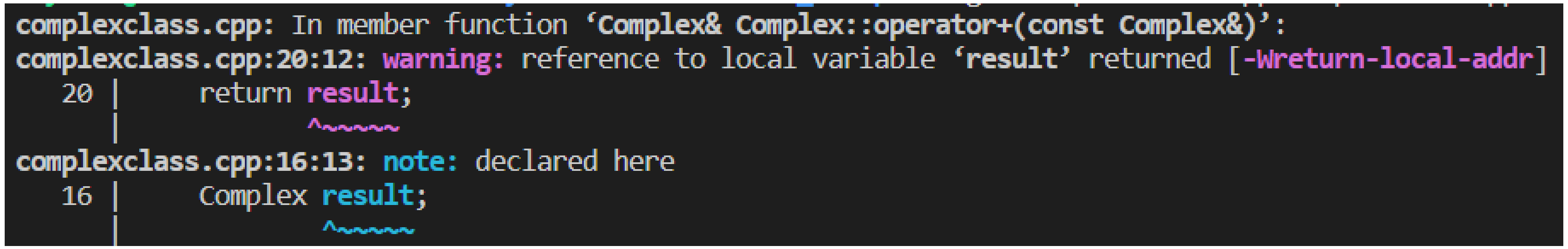
```
Complex Complex::operator+(const Complex &rhs) const
{
    Complex result;
    result.real = real + rhs.real;
    result.imag = imag + rhs.imag;

    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.

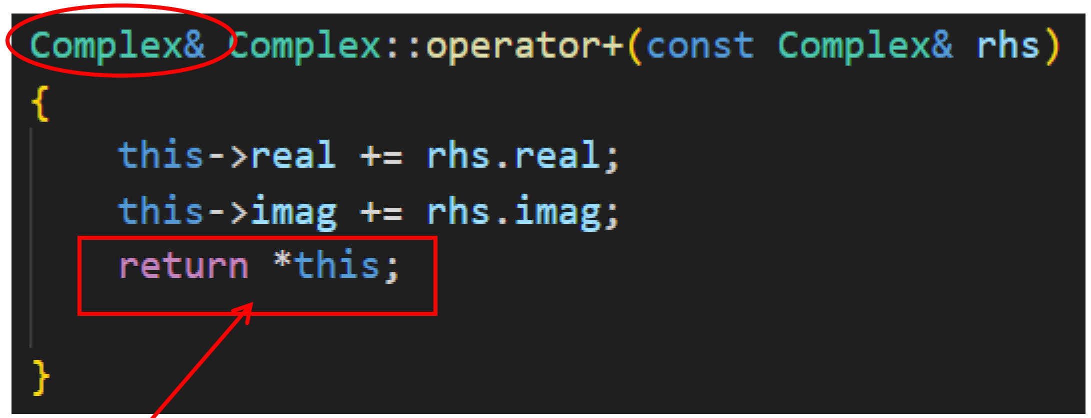


```cpp
Complex& Complex::operator+(const Complex &rhs) const
{
    Complex result;
    result.real = real + rhs.real;
    result.imag = imag + rhs.imag;

    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.

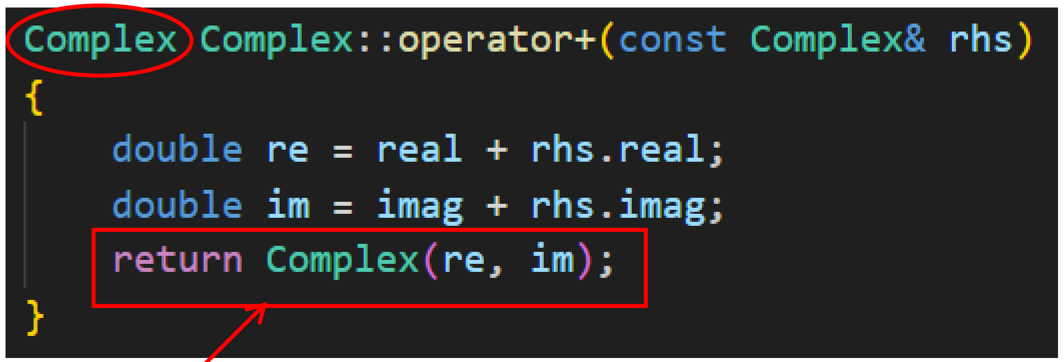


```
complexclass.cpp: In member function ‘Complex& Complex::operator+(const Complex&)’:
complexclass.cpp:20:12: warning: reference to local variable ‘result’ returned [-Wreturn-local-addr]
   20 |     return result;
      |            ^~~~~~
complexclass.cpp:16:13: note: declared here
   16 |     Complex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



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

}
```

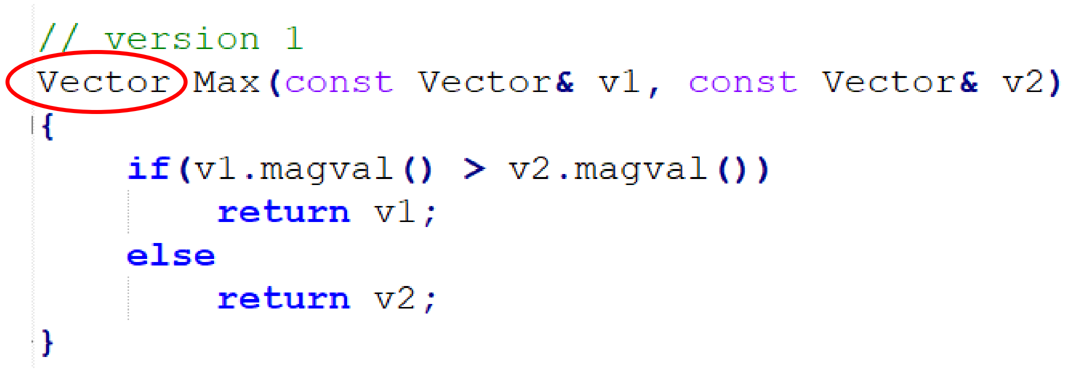
Returning a reference to **this object** works efficiently, but it modifies the values of the data member of **this object**.



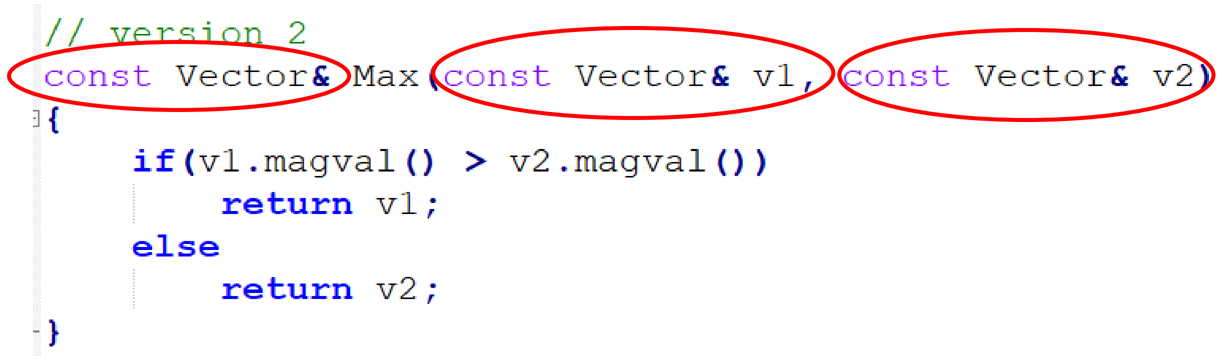
```
Complex Complex::operator+(const Complex& rhs)
{
    double re = real + rhs.real;
    double im = imag + rhs.imag;
    return Complex(re, im);
}
```

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;
}
```

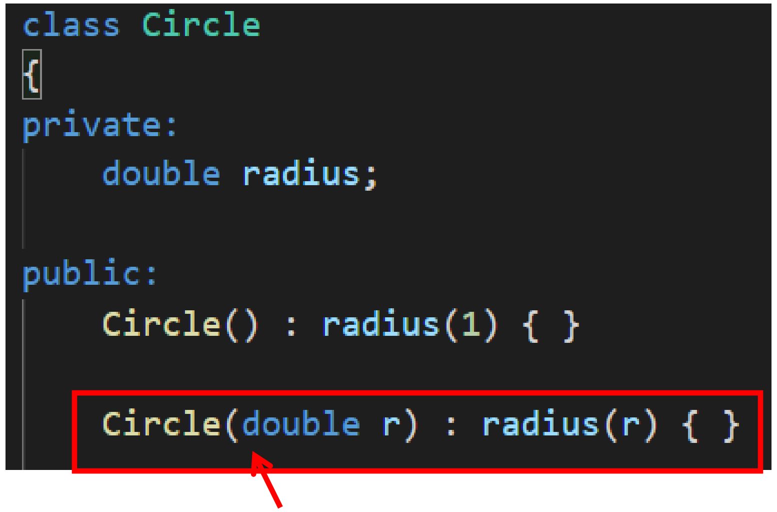
- Returning an object invokes the copy constructor, whereas returning a reference doesn’t. Thus version 2 does less work and is more efficient.
- The reference should be to an object that exists when the calling function is executing.
- Both v1 and v2 are declared as being const references, so the return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

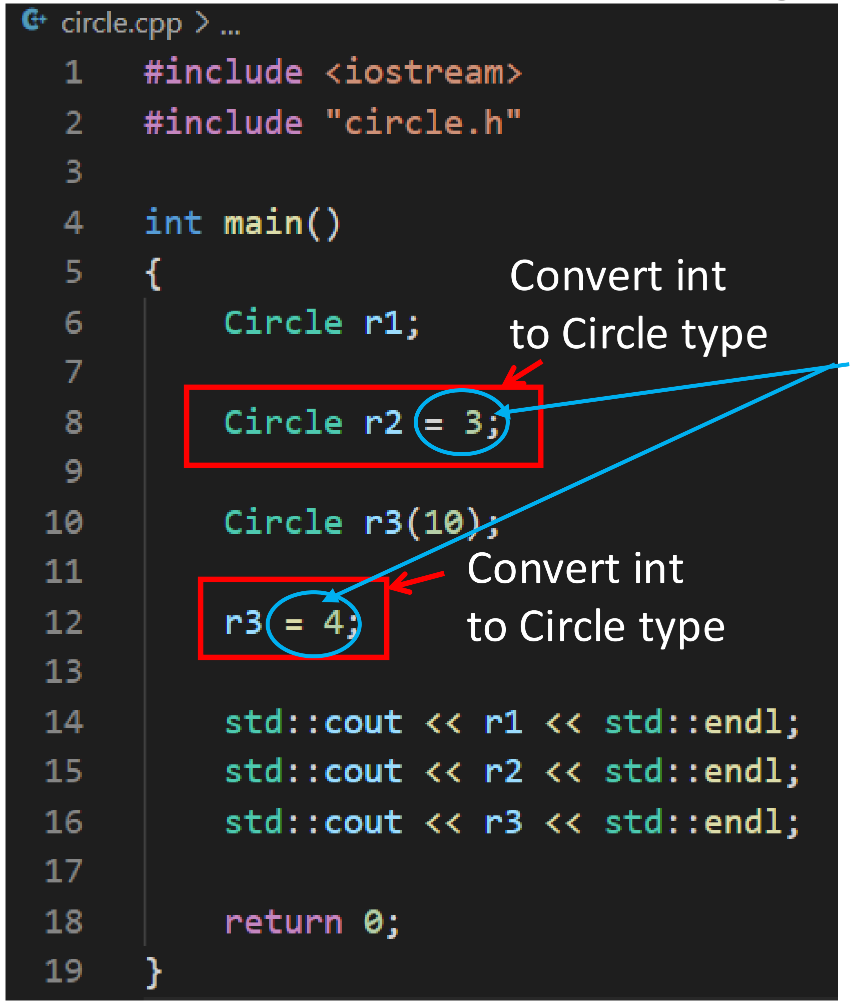
1. Implicit Class-Type Conversions

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***.



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

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

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

Converting constructor



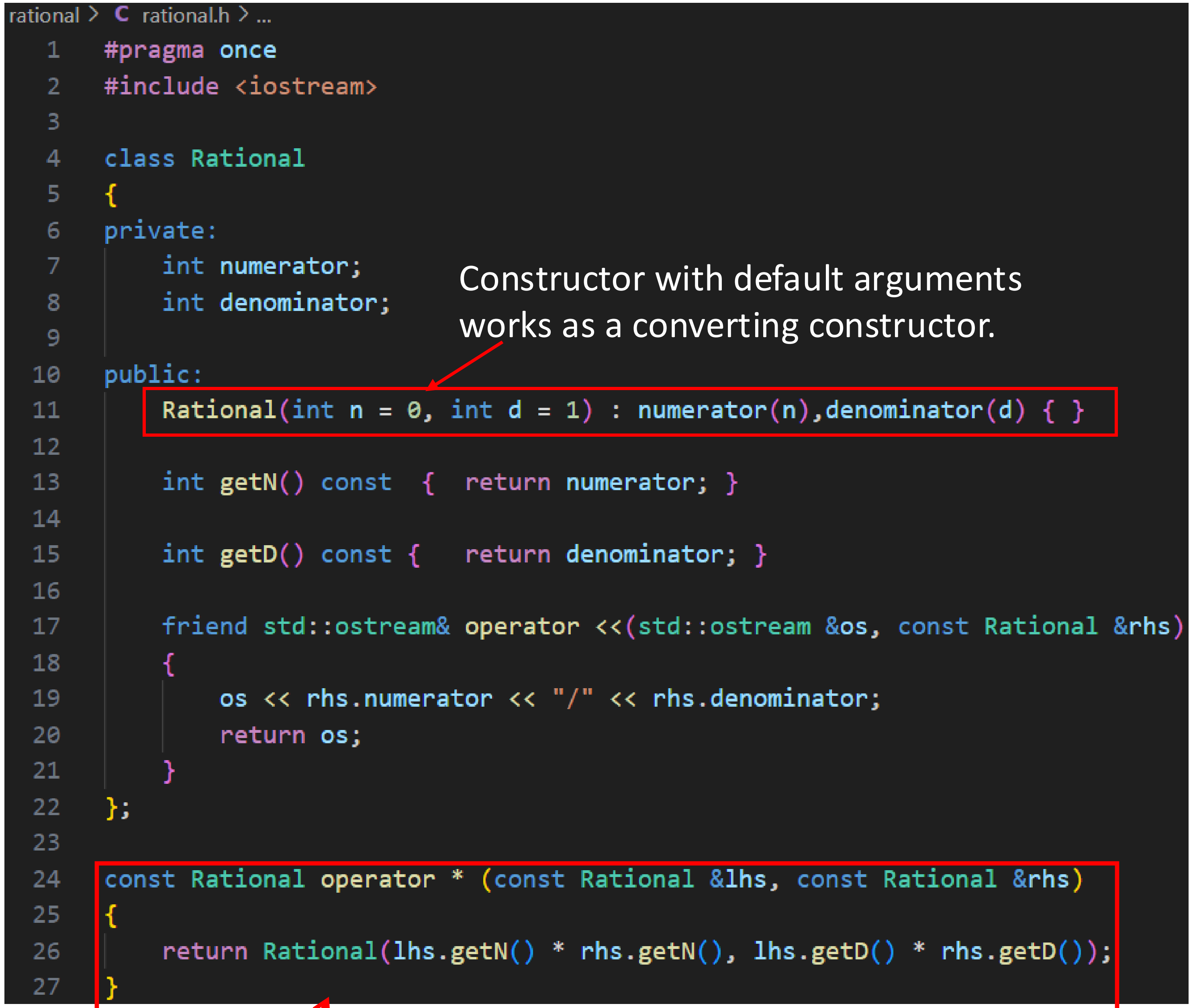
circle.cpp > ...

```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

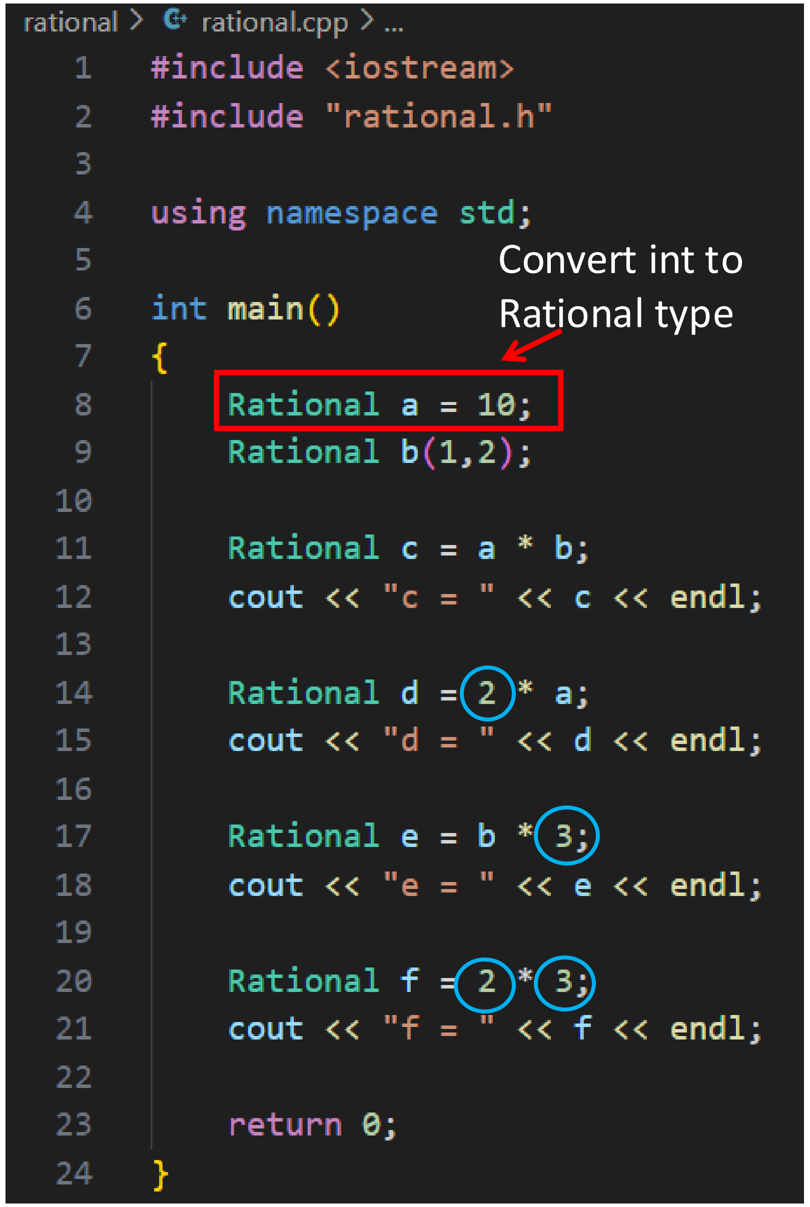
Convert int to Circle type

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


rational > rational.h > ...

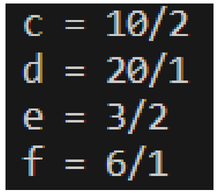
```cpp
#pragma once
#include <iostream>

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; }

    friend std::ostream& operator <<(std::ostream &os, const Rational &rhs)
    {
        os << rhs.numerator << "/" << rhs.denominator;
        return os;
    }
};

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 > rational.cpp > ...

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

using namespace std;

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

    Rational c = a * b;
    cout << "c = " << c << endl;

    Rational d = 2 * a;
    cout << "d = " << d << endl;

    Rational e = b * 3;
    cout << "e = " << e << endl;

    Rational f = 2 * 3;
    cout << "f = " << f << endl;

    return 0;
}
```

Convert int to Rational type

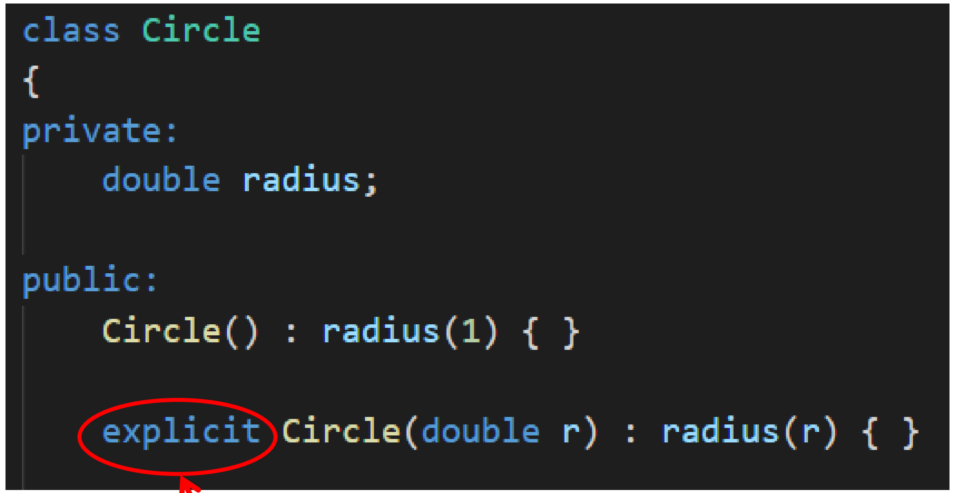


```
c = 10/2
d = 20/1
e = 3/2
f = 6/1
```

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

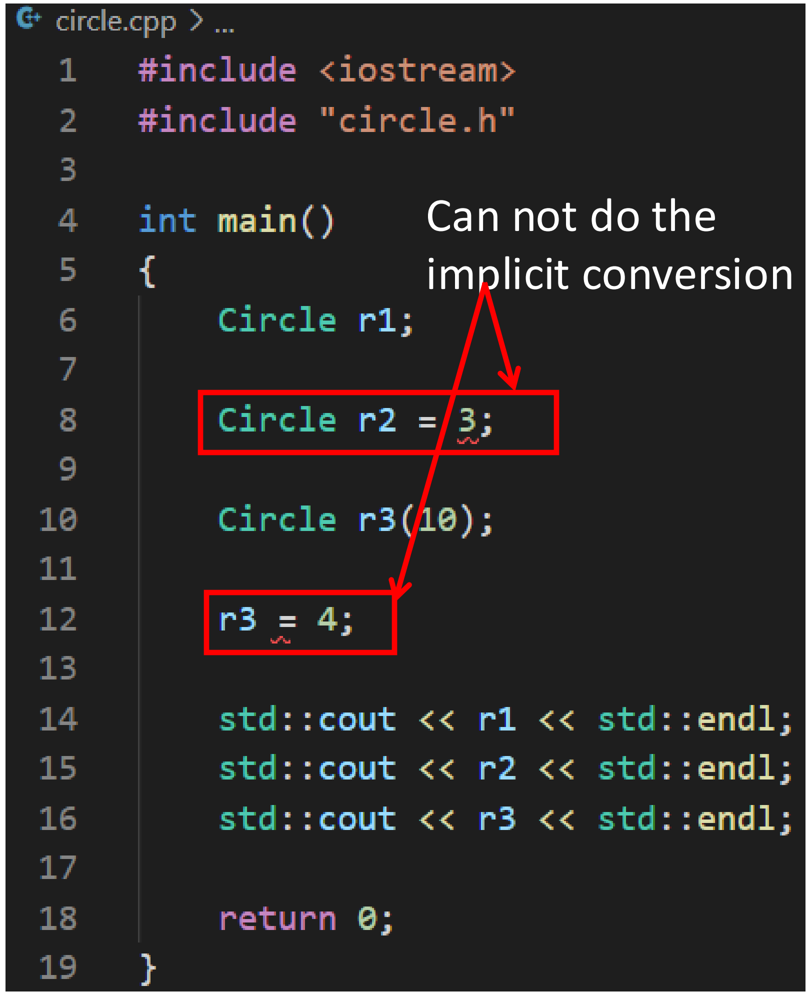
Use explicit to supper the implicit conversion

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



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

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

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

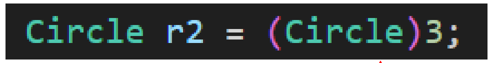
Turn off implicit conversion



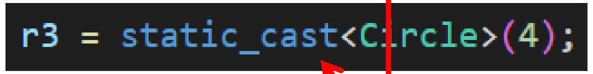
circle.cpp > ...

```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;
}
```

Can not do the implicit conversion



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



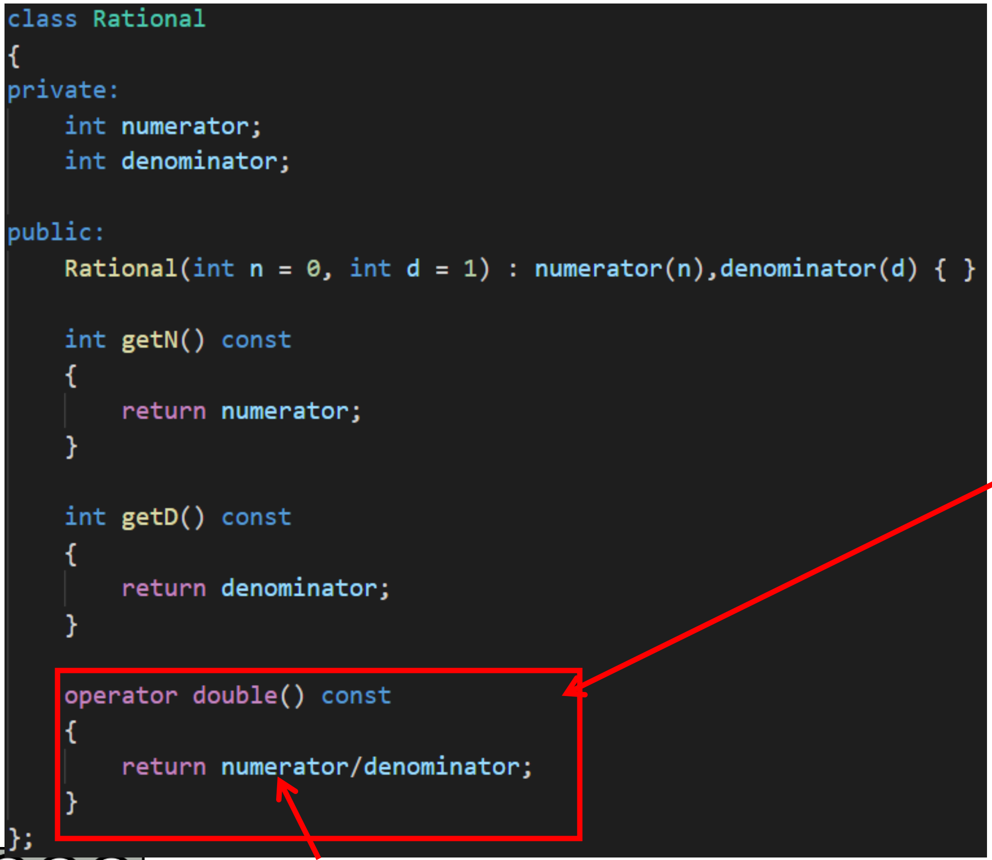
```cpp
r3 = static_cast<Circle>(4);
```

Use these two styles for explicit conversion

2. Conversion function

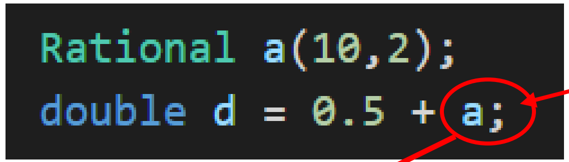
Conversion function is a member function with the name ***operator*** followed by a type specification, no return type, no arguments.

operator **typeName( );**



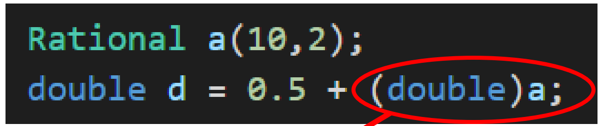
```
class Rational
{
private:
    int numerator;
    int denominator;

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



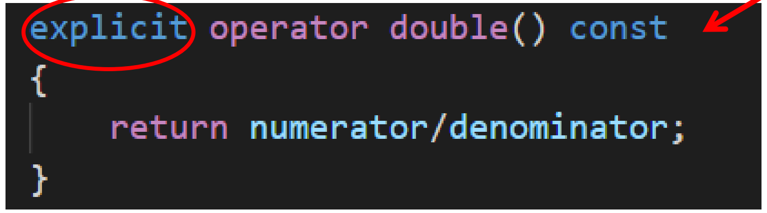
```
Rational a(10,2);
double d = 0.5 + a;
```

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.

Operator overloading in Python

‘Operateor overloading’ in Python

- In Python, operator overloading is accomplished by implementing special methods of classes (also known as magic methods). These methods start and end with double underscores (such as __add__).

implement the binary arithmetic operations
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |).

For instance, to evaluate the expression x + y, where x is an instance of a class that has an __add__() method, type(x).__add__(x, y) is called.

https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

Implement the binary arithmetic operations with reflected (swapped) operands

object.__radd__(self, other)
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rmatmul__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other[, modulo])
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands.

These functions are only called if the left operand does not support the corresponding operation and the operands are of different types.

For instance, to evaluate the expression x - y, where y is an instance of a class that has an __rsub__() method, type(y).__rsub__(y, x) is called

if type(x).__sub__(x, y) returns NotImplemented.

https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

```python
class MyTime:  #MyTime.py
    def __init__(self, h=0, m=0):
        self.hours=h
        self.minutes=m

    def __add__(self, other):
        if isinstance(other, int):
            total_min=self.minutes+other
            new_hours=self.hours+total_min//60
            new_minutes=total_min%60
            return MyTime(new_hours, new_minutes)
        return NotImplemented

    def __radd__(self, other):
        return self.__add__(other)

    def __str__(self):
        return f"{self.hours} hours and {self.minutes}
minutes."

if__name__=="__main__":
    mt=MyTime(1, 59)
    print(mt+2)
    print(1+mt)
```

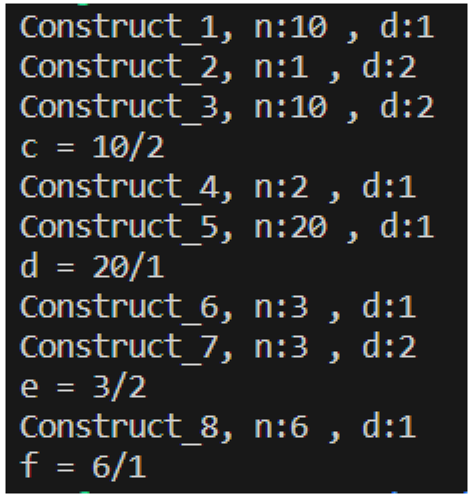
```cpp
#include <iostream>   //MyTime.cpp
class MyTime{
    int hours;
    int minutes;
  public:
    MyTime(): hours(0), minutes(0){}
    MyTime(int h, int m): hours(h), minutes(m){}
    MyTime operator+(int m) const{
        MyTime sum;
        sum.minutes=this->minutes+m;
        sum.hours=this->hours;
        sum.hours+=sum.minutes/60;
        sum.minutes%=60;
        return sum;
    }
    friend MyTimeoperator+(int m, const MyTime&t){
        return t+m;
    }
    friend std::ostream& operator<<(std::ostream&os, const MyTime&t){
        std::string str=std::to_string(t.hours) +" hours and "
                +std::to_string(t.minutes) +" minutes.";
        os<<str;
        return os;
    }
};
int main(){
  MyTime mt(1,59);
  std::cout<<mt+2<<std::endl;
  std::cout<<1+mt<<std::endl;
  return 0;
}
```

Exercise1

Modify the code(rational.h) so that the program runs as shown in the screenshot on the right and explain the output where each constructor is run.

```cpp
// rational.h
#pragma once
#include <iostream>
class Rational{
private:
    staticint id;
    int numerator;
    int denominator;
public:
    Rational(int n=0, int d=1): numerator(n), denominator(d){
        std::cout<<"Construct_"<<id<<", n:"<<numerator<<" ,
d:"<<denominator<<std::endl;
    }

    int getN() const { return numerator; }
    int getD() const { return denominator; }
    friend std::ostream & operator<<(std::ostream& os, const Rational&rhs){
        os<<rhs.numerator<<"/"<<rhs.denominator;
        returnos;
    }
};
int Rational::id=0;
const Rational operator*(constRational&lhs, const Rational&rhs){
        return Rational(lhs.getN()*rhs.getN(), lhs.getD()*rhs.getD() );
}
```

```cpp
#include <iostream>
#include "rational.h"
using namespace std;
int main(){
    Rational a=10;
    Rational b(1,2);
    Rational c=a*b;
    cout<<"c = "<<c<<endl;
    Rational d=2*a;
    cout<<"d = "<<d<<endl;
    Rational e=b*3;
    cout<<"e = "<<e<<endl;
    Rational f=2*3;
    cout<<"f = "<<f<<endl;
    return0;
}
```



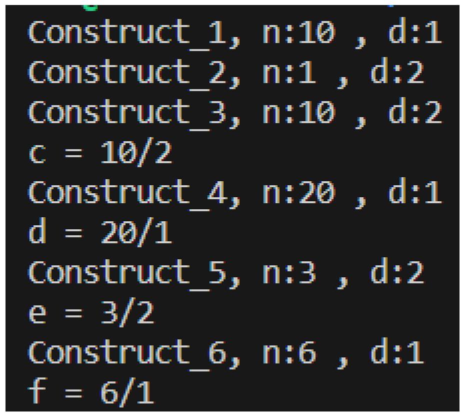
```
Construct_1, n:10 , d:1
Construct_2, n:1 , d:2
Construct_3, n:10 , d:2
c = 10/2
Construct_4, n:2 , d:1
Construct_5, n:20 , d:1
d = 20/1
Construct_6, n:3 , d:1
Construct_7, n:3 , d:2
e = 3/2
Construct_8, n:6 , d:1
f = 6/1
```

Exercise2

Please modify the code(rational.h) in Exercise1 to the program runs as shown in the screenshot. Explain what modifications have been made and wy.

```cpp
// rational.h
#pragma once
#include <iostream>
class Rational{
private:
    staticint id;
    int numerator;
    int denominator;
public:
    Rational(int n=0, int d=1): numerator(n), denominator(d){
        std::cout<<"Construct_"<<id<<", n:"<<numerator<<" ,
d:"<<denominator<<std::endl;
    }

    int getN() const { return numerator; }
    int getD() const { return denominator; }
    friend std::ostream & operator<<(std::ostream& os, const Rational&rhs){
        os<<rhs.numerator<<"/"<<rhs.denominator;
        returnos;
    }
};
int Rational::id=0;
const Rational operator*(constRational&lhs, const Rational&rhs){
        return Rational(lhs.getN()*rhs.getN(), lhs.getD()*rhs.getD() );
}
```

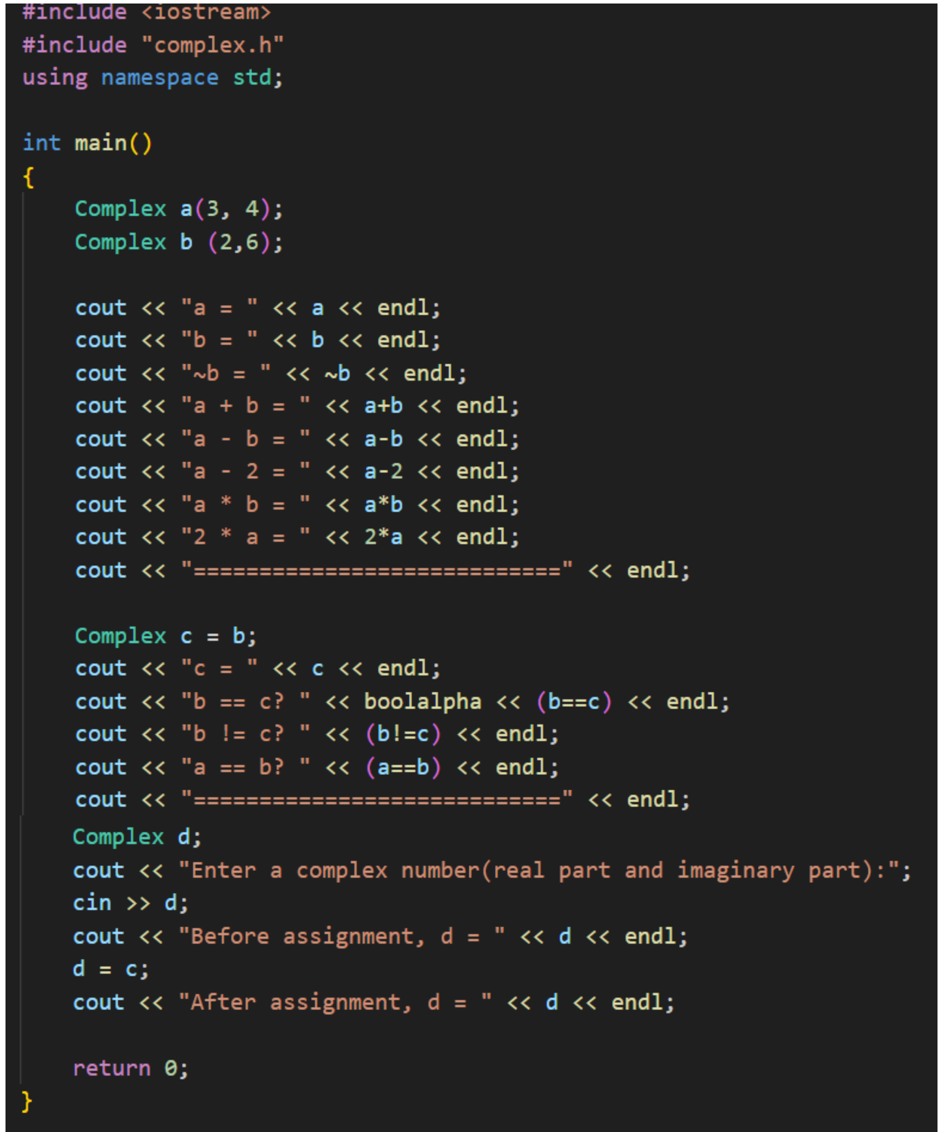
```cpp
#include <iostream>
#include "rational.h"
using namespace std;
int main(){
    Rational a=10;
    Rational b(1,2);
    Rational c=a*b;
    cout<<"c = "<<c<<endl;
    Rational d=2*a;
    cout<<"d = "<<d<<endl;
    Rational e=b*3;
    cout<<"e = "<<e<<endl;
    Rational f=2*3;
    cout<<"f = "<<f<<endl;
    return0;
}
```



```
Construct_1, n:10 , d:1
Construct_2, n:1 , d:2
Construct_3, n:10 , d:2
c = 10/2
Construct_4, n:20 , d:1
d = 20/1
Construct_5, n:3 , d:2
e = 3/2
Construct_6, n:6 , d:1
f = 6/1
```

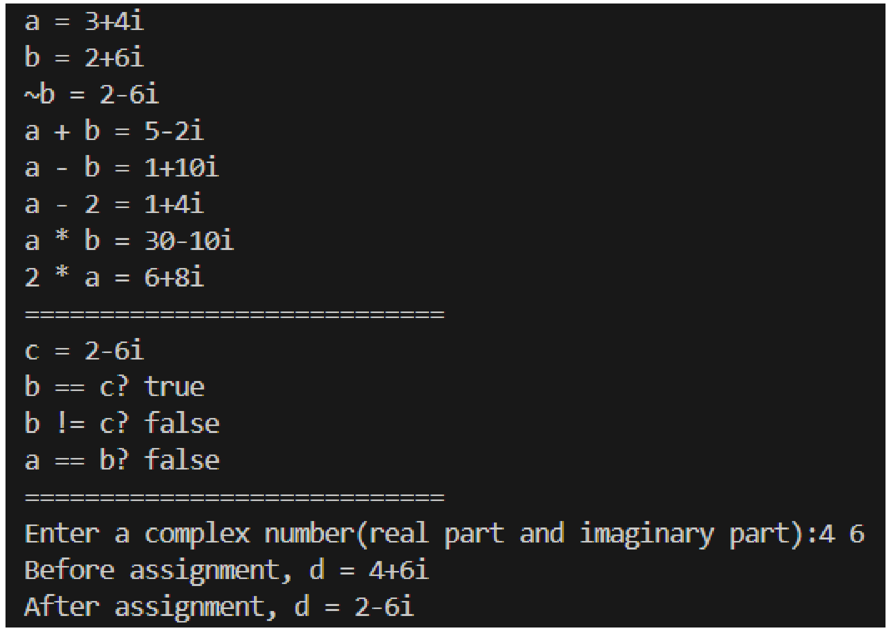
Exercise3

Continue to improve the Complex class and add 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 the references to objects and const whenever warranted. The sample should run as follows:



```
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
```