Operator Overloading

Function operators in string

```
string s1("Washington");
string s2("California");
cout << "The first character in s1 is " << s1[0] << endl;
cout << "s1 + s2 is " << (s1 + s2) << endl;
cout << "s1 == s2? " << (s1 == s2) << endl;

string s1("Washington");
string s2("California");
cout << "The first character in s1 is " << s1.operator[](0) << endl;
cout << "s1 + s2 is " << operator+(s1, s2) << endl;
cout << "s1 == s2? " << operator==(s1, s2) << endl;
```

The Rational Class

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

        Rational(int numerator, int denominator);
        int compareTo(const Rational& secondRational) const;
        Rational add(const Rational& secondRational) const;
        Rational subtract(const Rational& secondRational) const;
        bool operator<(const Rational &secondRational) const;
        Rational operator+(const Rational &secondRational) const;
        string toString() const;
        Rational& operator+=(const Rational &secondRational);
        Rational& operator++();
        Rational operator++(int dummy);
};
```

The Rational Class

```
Rational Rational::add(const Rational& secondRational) const
{
        int n = numerator * secondRational.denominator + denominator *
        secondRational.numerator;
        int d = denominator * secondRational.denominator;
        return Rational(n, d);
}
```

add

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

```
Rational Rational::subtract(const Rational& secondRational) const
{
    int n = numerator * secondRational.denominator
      - denominator * secondRational.numerator;
    int d = denominator * secondRational.denominator;
    return Rational(n, d);
}
```

subtract

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

The Rational Class

```
int Rational::compareTo(const Rational& secondRational) const
{
    Rational temp = subtract(secondRational);
    if (temp.numerator < 0)
      return -1;
    else if (temp.numerator == 0)
      return 0;
    else
      return 1;
}
```

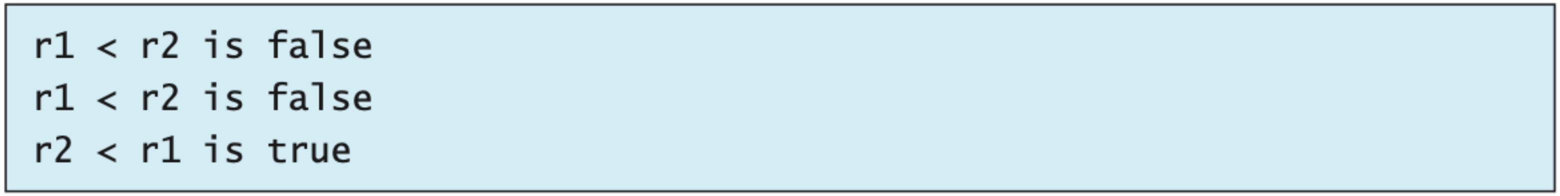
Overloadable Operators

```
+     -     *     /     %     ^     &     |     ~     !     =
<     >     +=    -=    *=    /=    %=    ^=    &=    |=    <<
>>    >>=   <<=   ==    !=    <=    >=    &&    ||    ++    --
->*   ,     ->    []    ()    new   delete
```

< Function Operator

```
bool Rational::operator<(const Rational &secondRational) const
{
  // compareTo is already defined Rational.h
  return compareTo(secondRational) < 0;
}

Rational r1(4, 2);
Rational r2(2, 3);
cout << "r1 < r2 is " << (r1.operator<(r2) ? "true" : "false");
cout << "\nr1 < r2 is " << ((r1 < r2) ? "true" : "false");
cout << "\nr2 < r1 is " << (r2.operator<(r1) ? "true" : "false");
```



```
r1 < r2 is false
r1 < r2 is false
r2 < r1 is true
```

+ Function Operator

```
Rational Rational::operator+(const Rational &secondRational) const
{
  // add is already defined Rational.h
  return add(secondRational);
}
```

```
Rational r1(3, 2);
Rational r2(2, 3);
cout << "r1 + r2 is " << (r1 + r2).toString() << endl;
```

```
r1 + r2 is 13/6
```

+ Function Operator

```
Rational Rational::operator+(int s) const
{
    int n = numerator + s*denominator;
    return Rational(n, denominator);
}
```

```
Rational r1(3, 2);
cout << "r1 + 3 is " << (r1+3).toString() << endl;
```

```
r1 + r2 is 9/2
```

Overloading the Augmented Operators

C++ has augmented assignment operators +=, **-=**, *=, /=, and %= for adding, subtracting, multiplying, dividing, and modulus a value in a variable. You can overload these operators in the Rational class.

```
Rational& Rational::operator+=(const Rational &secondRational)
{
  *this = add(secondRational);
  return *this;
}
```

Overloading the Augmented Operators

```
Rational r1(3, 2);
Rational r2 = r1 += Rational(2, 3);
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

```
r1 is 13/6
r2 is 13/6
```

Overloading the ++ and -- Operators

- The ++ and -- operators may be prefix or postfix.

- The prefix ++var or --var first adds or subtracts 1 from the variable and then return to the new value.

- The postfix var++ or var-- adds or subtracts 1 from the variable, but return to the old value.

Overloading the ++ and -- Operators

```
int i = 0, j = 0;
j = i++;
cout << j << i << endl;
i = j = 0;
j = ++i;
cout << j << i << endl;
```

```
01
11
```

Overloading the ++ and -- Operators

```
// Prefix increment
Rational& Rational::operator++()
{
  numerator += denominator;
  return *this;
}
// Postfix increment
Rational Rational::operator++(int dummy)
{
  Rational temp(numerator, denominator);
  numerator += denominator;
  return temp;
}
```

Overloading the ++ and --

```
Rational r1(2, 3);
Rational r3 = ++r1;
cout << "r1 is " << r1.toString() << endl;
cout << "r3 is " << r3.toString() << endl;
Rational r2(2, 3);
r3 = r2++;
cout << "r2 is " << r2.toString() << endl;
cout << "r3 is " << r3.toString() << endl;
```

```
r1 is 5/3
r3 is 5/3
r2 is 5/3
r3 is 2/3
```

Overloading the = Operator

By default, the = operator performs a memberwise copy from one object to the other. For example, the following code copies r2 to r1.

```
Rational r1(1, 2);
Rational r2(4, 5);
r1 = r2;
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

Overloading the = Operator

```
Rational& Rational::operator=(const Rational &secondRational)
{
        numerator = secondRational.numerator;
        denominator = secondRational.denominator;
        return *this;
}
```

Copy Constructor

```
Rational::Rational(const Rational &secondRational){
    numerator = secondRational.numerator;
    denominator = secondRational.denominator;
}
```

Rule of Three

The **copy constructor**, the = **assignment operator**, and the **destructor** are called the *rule of three*, or *the Big Three*. If they are not defined explicitly, all three are created by the compiler **automatically**. You should customize them if a data field in the class is a **pointer** that points to a dynamic generated array or object. If you have to customize one of the three, you should customize the other two as well.

Operator Overloading Practice

- Implement prefix and postfix -- operators
  - Rational r1(5,3); Rational r2 = r1--;
  - Rational r3(5,3); Rational r4 = --r3;
  - cout << r1.toString() << endl; // 2/3
  - cout << r2.toString() << endl; // 5/3
  - cout << r3.toString() << endl; // 2/3
  - cout << r4.toString() << endl; // 2/3
- Implement == operator
  - cout << r1 == r2; << endl; // 0
- Implement /= operator by an integer number
  - r1 /= 3;
  - cout << r1.toString(); << endl; // 5/9
- If you have any questions, please feel free to ask.

Operator Overloading Solution

- Implement prefix and postfix -- operators

```
Rational& Rational::operator--()
{
 numerator -= denominator;
 return *this;
}
Rational Rational::operator--(int dummy)
{
 Rational temp(numerator, denominator);
 numerator -= denominator;
 return temp;
}
```

Operator Overloading Solution

- Implement == operator

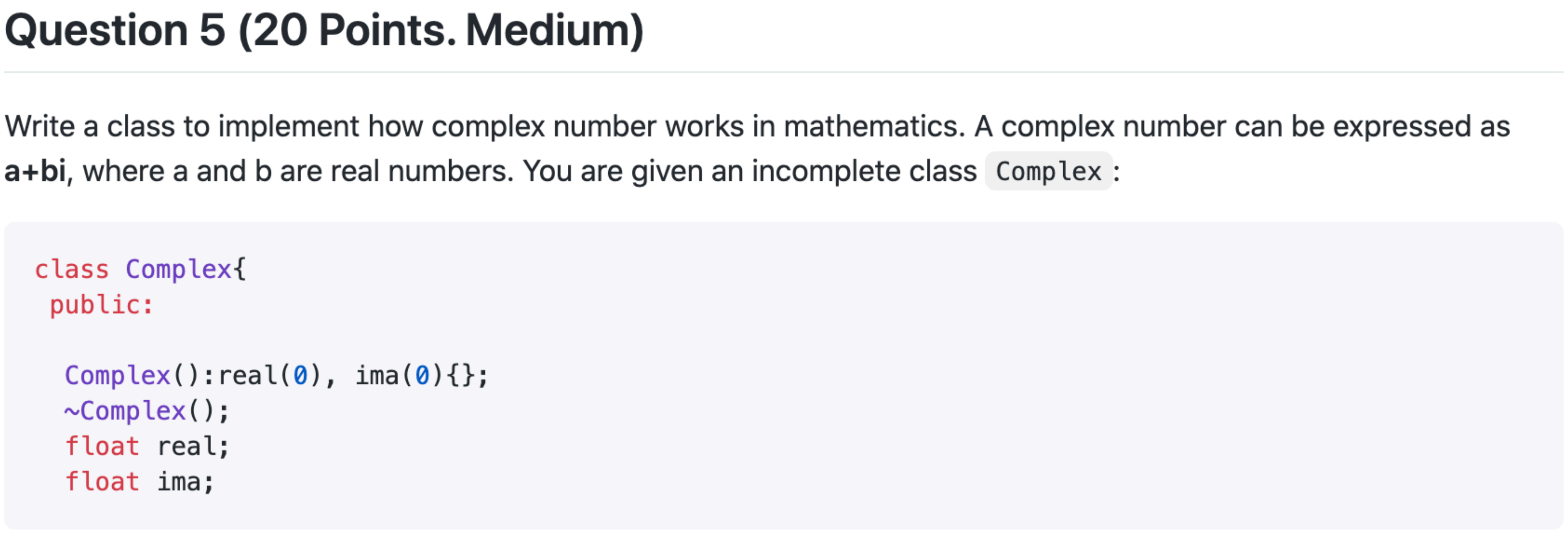
```
Rational& Rational::operator--()
{
 numerator -= denominator;
 return *this;
}
Rational Rational::operator--(int dummy)
{
 Rational temp(numerator, denominator);
 numerator -= denominator;
 return temp;
}
```

Operator Overloading Solution

- Implement /= operator by an integer number

```
Rational& Rational::operator/=(const int s)
{
    denominator*=s;
    return *this;
}
```

HW3 Q5



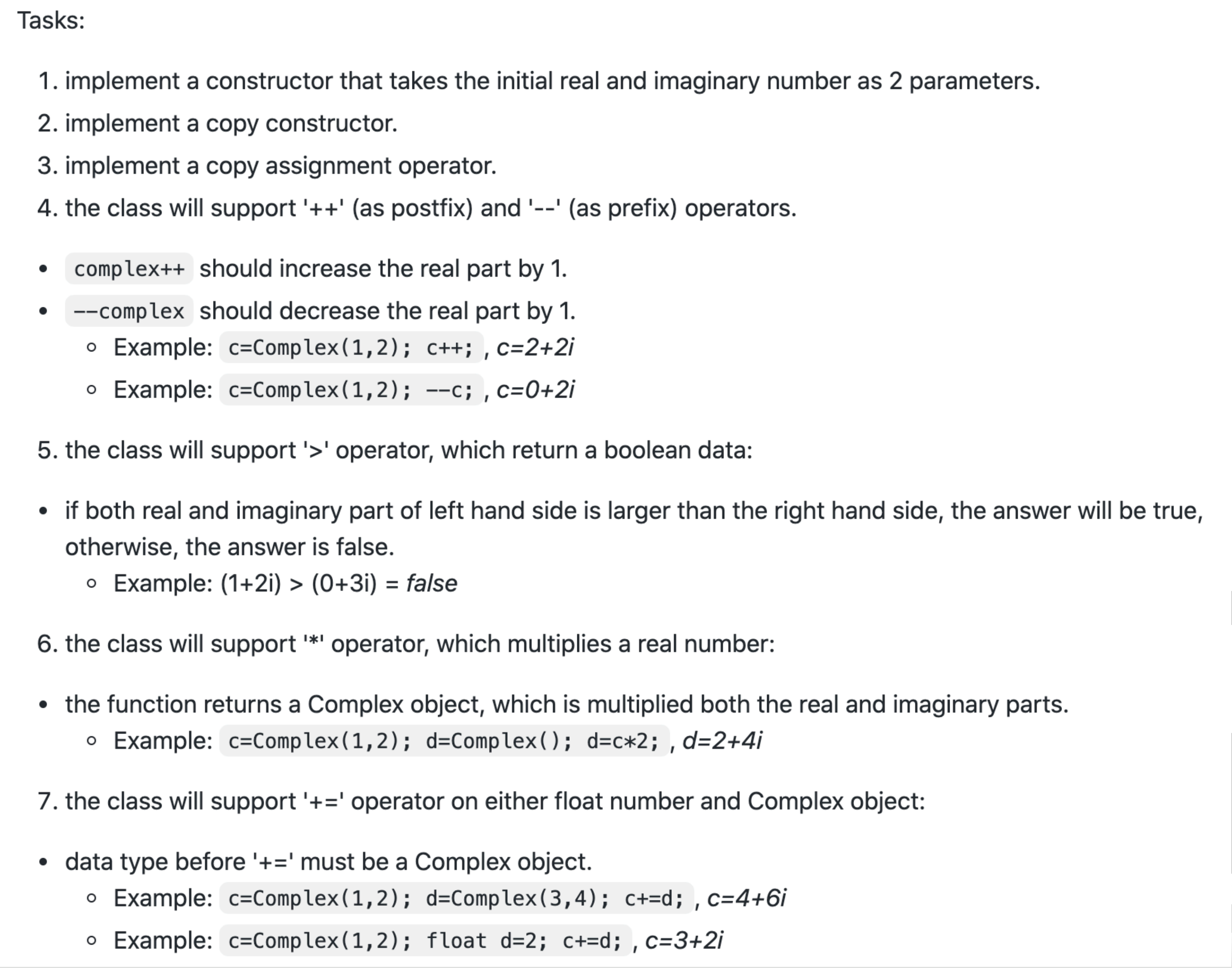
Question 5 (20 Points. Medium)

Write a class to implement how complex number works in mathematics. A complex number can be expressed as **a+bi**, where a and b are real numbers. You are given an incomplete class `Complex`:

```
class Complex{
 public:

  Complex():real(0), ima(0){};
  ~Complex();
  float real;
  float ima;
```

HW3 Q5



Tasks:

1. implement a constructor that takes the initial real and imaginary number as 2 parameters.
2. implement a copy constructor.
3. implement a copy assignment operator.
4. the class will support '++' (as postfix) and '--' (as prefix) operators.

- `complex++` should increase the real part by 1.
- `--complex` should decrease the real part by 1.
  - Example: `c=Complex(1,2); c++;` , *c=2+2i*
  - Example: `c=Complex(1,2); --c;` , *c=0+2i*

5. the class will support '>' operator, which return a boolean data:

- if both real and imaginary part of left hand side is larger than the right hand side, the answer will be true, otherwise, the answer is false.
  - Example: (1+2i) > (0+3i) = *false*

6. the class will support '*' operator, which multiplies a real number:

- the function returns a Complex object, which is multiplied both the real and imaginary parts.
  - Example: `c=Complex(1,2); d=Complex(); d=c*2;` , *d=2+4i*

7. the class will support '+=' operator on either float number and Complex object:

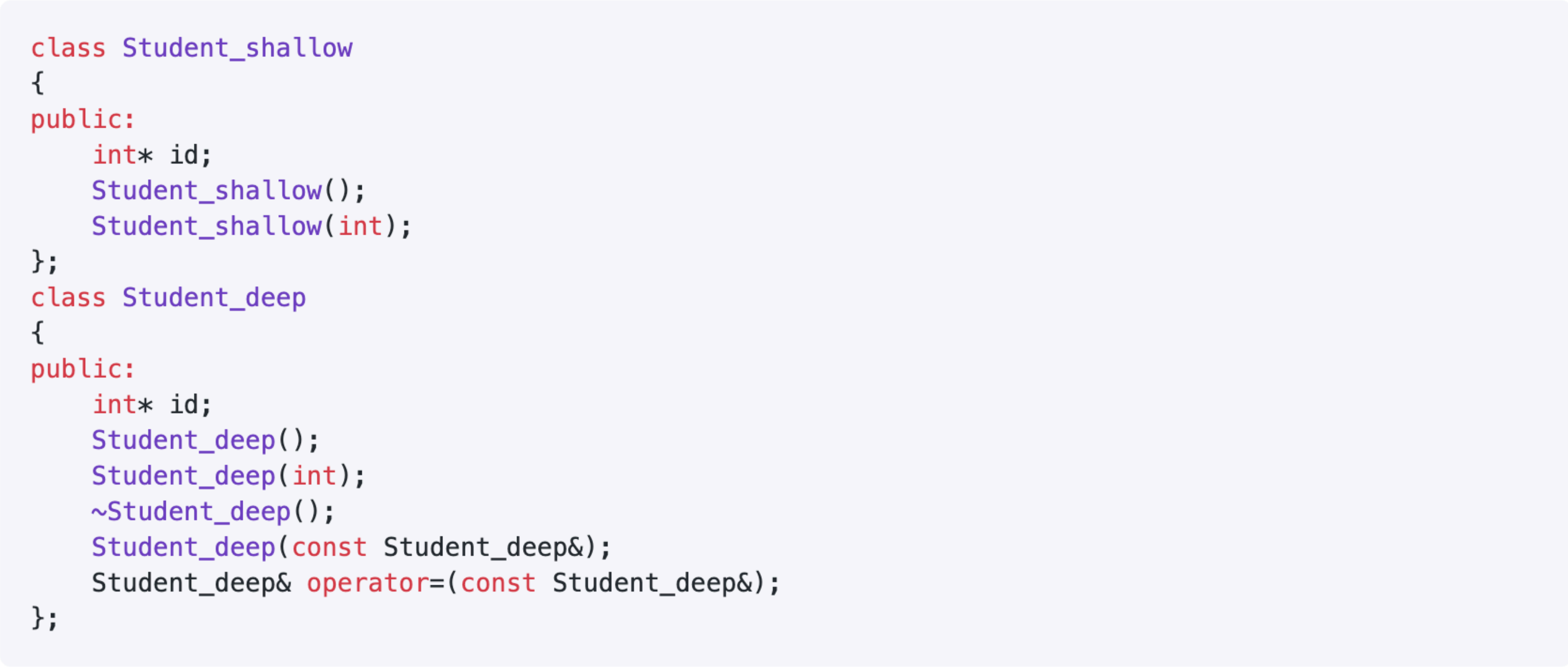
- data type before '+=' must be a Complex object.
  - Example: `c=Complex(1,2); d=Complex(3,4); c+=d;` , *c=4+6i*
  - Example: `c=Complex(1,2); float d=2; c+=d;` , *c=3+2i*

Deep copy vs Shallow copy

- If data fields of a class include a **pointer**, we suggest customize your copy constructor and copy assignment operator.

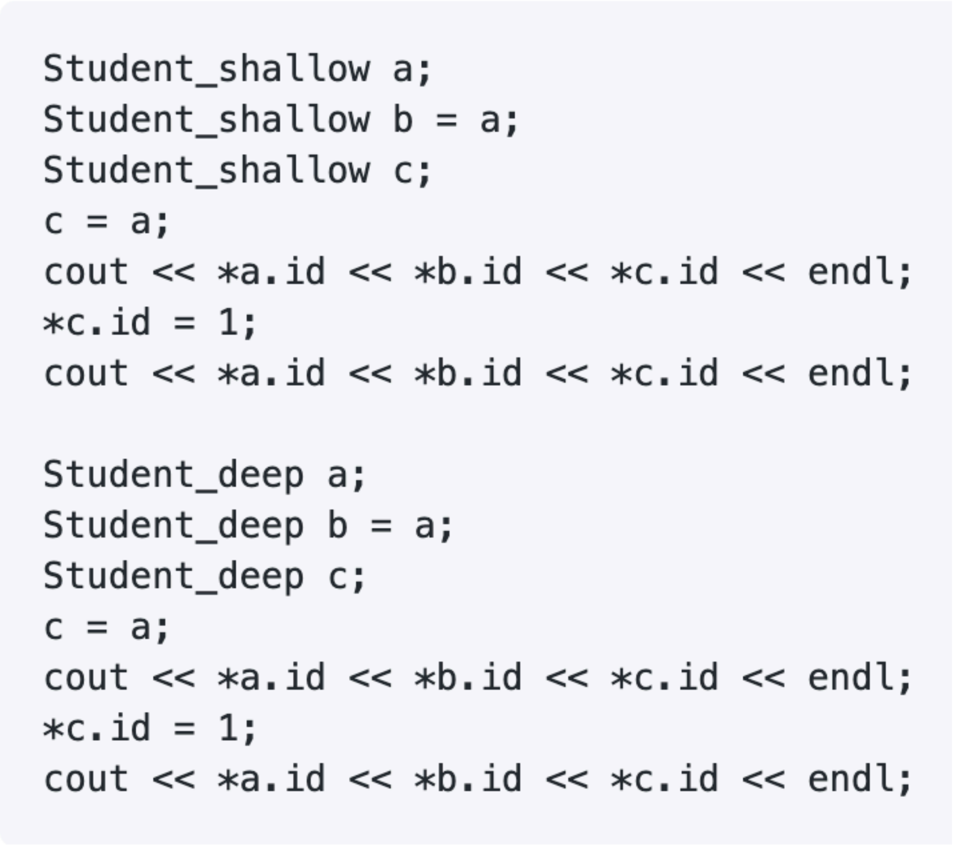
```
Student_shallow(const Student_shallow & s){
    id = s.id;
}
Student_deep(const Student_deep& s){
    id = new int(*s.id);
}
```

HW3 Q4



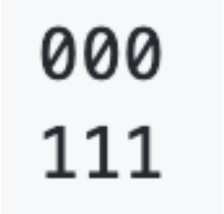
```
class Student_shallow
{
public:
    int* id;
    Student_shallow();
    Student_shallow(int);
};
class Student_deep
{
public:
    int* id;
    Student_deep();
    Student_deep(int);
    ~Student_deep();
    Student_deep(const Student_deep&);
    Student_deep& operator=(const Student_deep&);
};
```

HW3 Q4

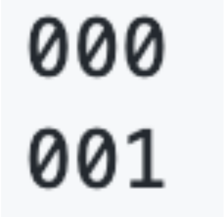


```
Student_shallow a;
Student_shallow b = a;
Student_shallow c;
c = a;
cout << *a.id << *b.id << *c.id << endl;
*c.id = 1;
cout << *a.id << *b.id << *c.id << endl;

Student_deep a;
Student_deep b = a;
Student_deep c;
c = a;
cout << *a.id << *b.id << *c.id << endl;
*c.id = 1;
cout << *a.id << *b.id << *c.id << endl;
```



```
000
111
```



```
000
001
```