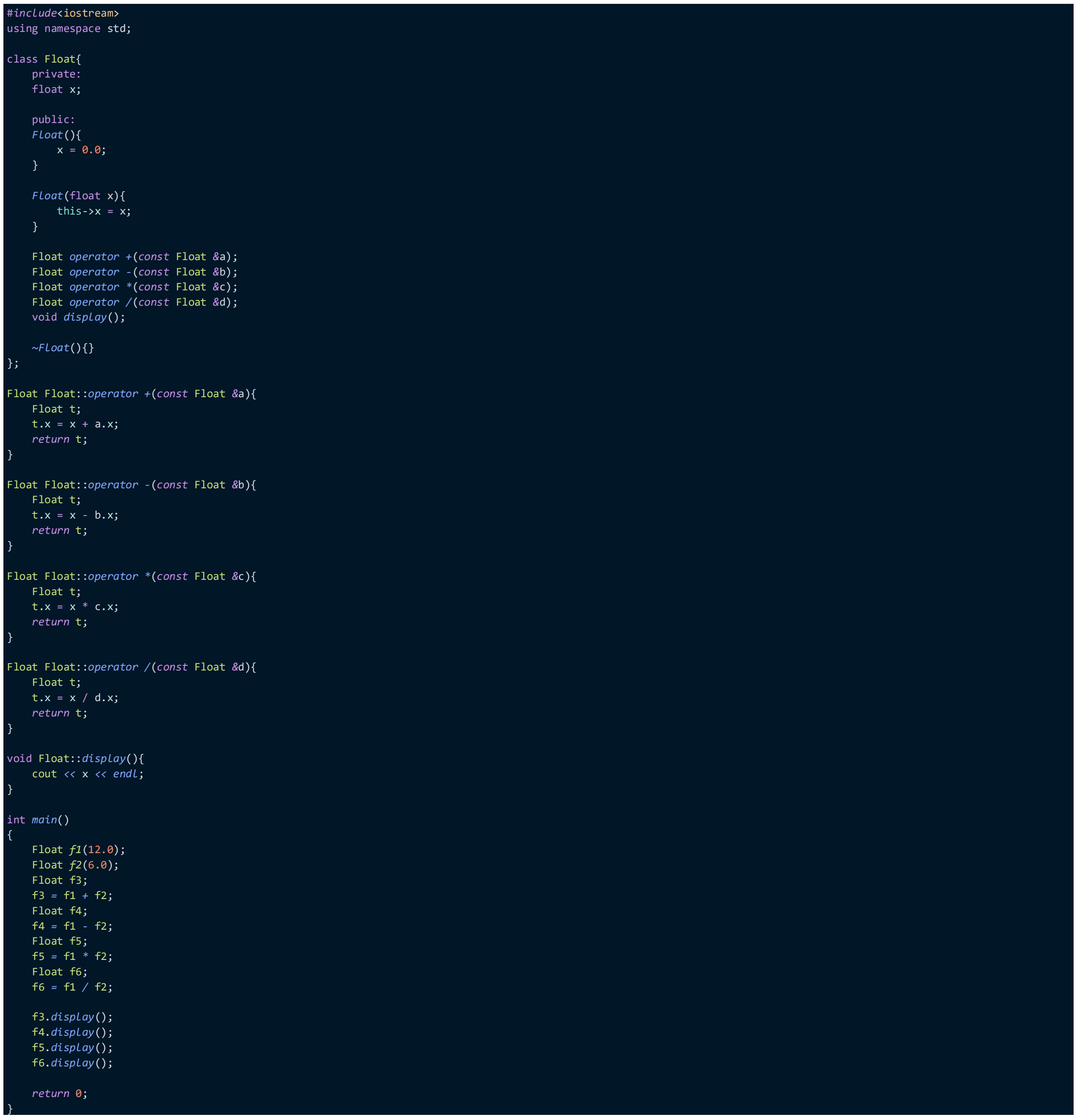
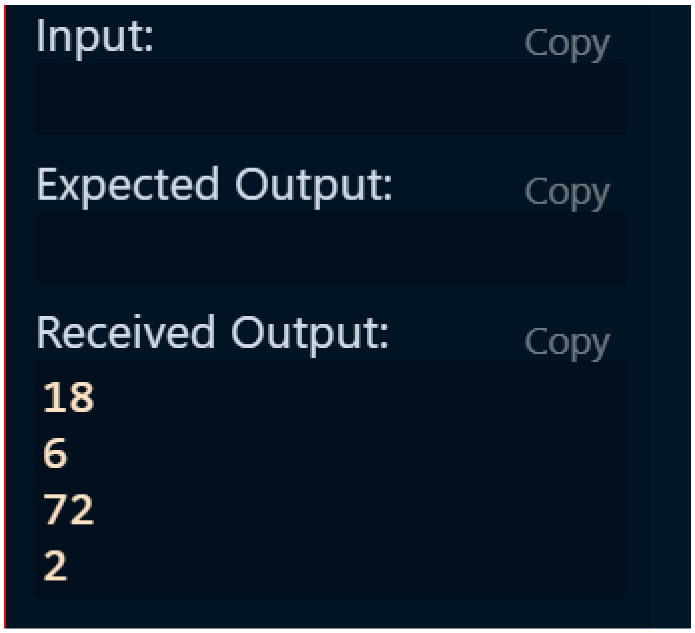
1. Create a class FLOAT that contains one float data member .Overload all the four

arithmetic operators so that they operate on the objects of FLOAT.

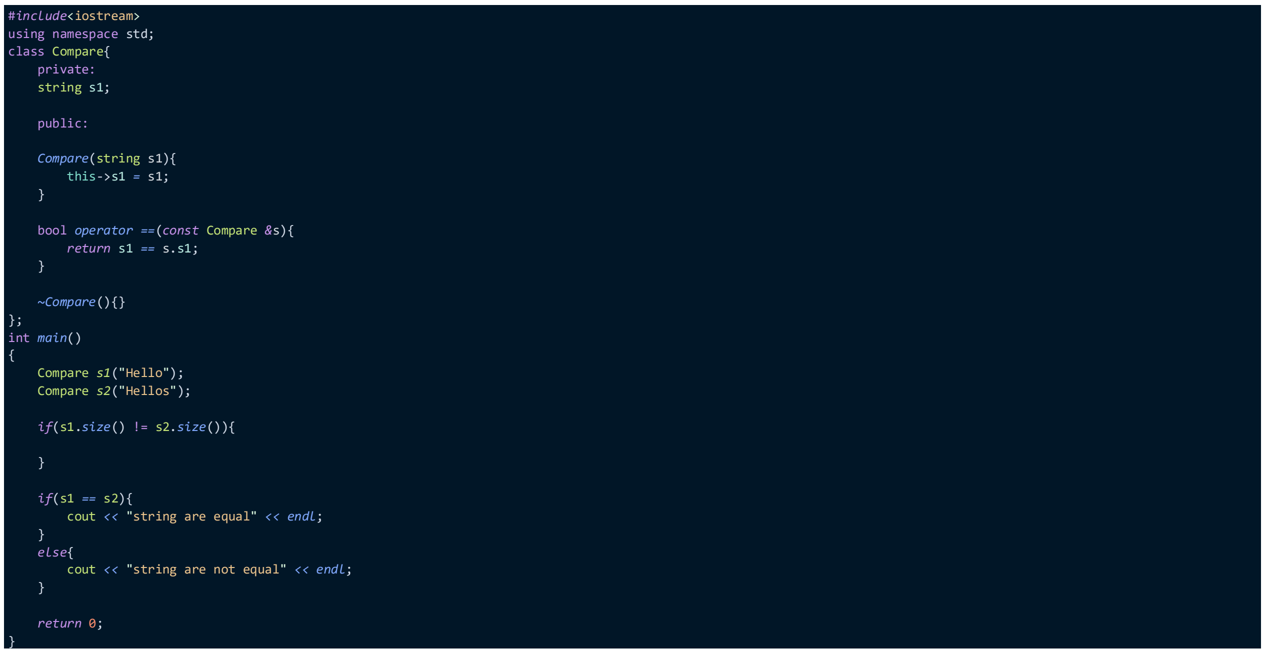


```cpp
#include<iostream>
using namespace std;

class Float{
    private:
    float x;

    public:
    Float(){
        x = 0.0;
    }

    Float(float x){
        this->x = x;
    }

    Float operator +(const Float &a);
    Float operator -(const Float &b);
    Float operator *(const Float &c);
    Float operator /(const Float &d);
    void display();

    ~Float(){}
};

Float Float::operator +(const Float &a){
    Float t;
    t.x = x + a.x;
    return t;
}

Float Float::operator -(const Float &b){
    Float t;
    t.x = x - b.x;
    return t;
}

Float Float::operator *(const Float &c){
    Float t;
    t.x = x * c.x;
    return t;
}

Float Float::operator /(const Float &d){
    Float t;
    t.x = x / d.x;
    return t;
}

void Float::display(){
    cout << x << endl;
}

int main()
{
    Float f1(12.0);
    Float f2(6.0);
    Float f3;
    f3 = f1 + f2;
    Float f4;
    f4 = f1 - f2;
    Float f5;
    f5 = f1 * f2;
    Float f6;
    f6 = f1 / f2;

    f3.display();
    f4.display();
    f5.display();
    f6.display();

    return 0;
}
```

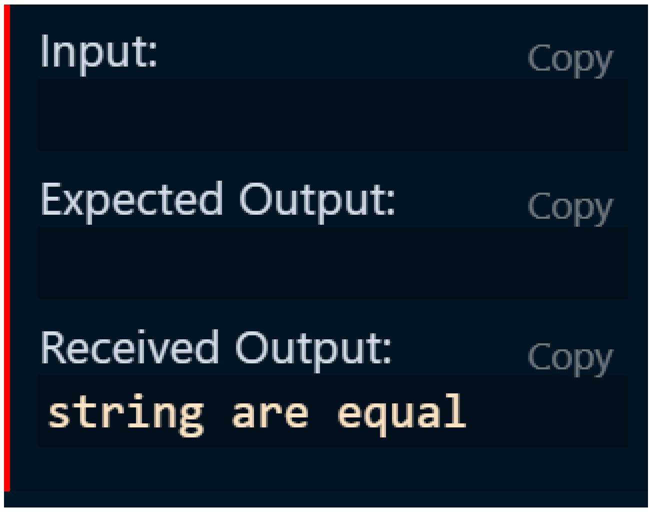


```
18
6
72
2
```

2. Define a class string. Overlaod ==operator to compare 2 strings.



```cpp
#include<iostream>
using namespace std;
class Compare{
    private:
    string s1;

    public:

    Compare(string s1){
        this->s1 = s1;
    }

    bool operator ==(const Compare &s){
        return s1 == s.s1;
    }

    ~Compare(){}
};
int main()
{
    Compare s1("Hello");
    Compare s2("Hellos");

    if(s1.size() != s2.size()){

    }

    if(s1 == s2){
        cout << "string are equal" << endl;
    }
    else{
        cout << "string are not equal" << endl;
    }

    return 0;
}
```

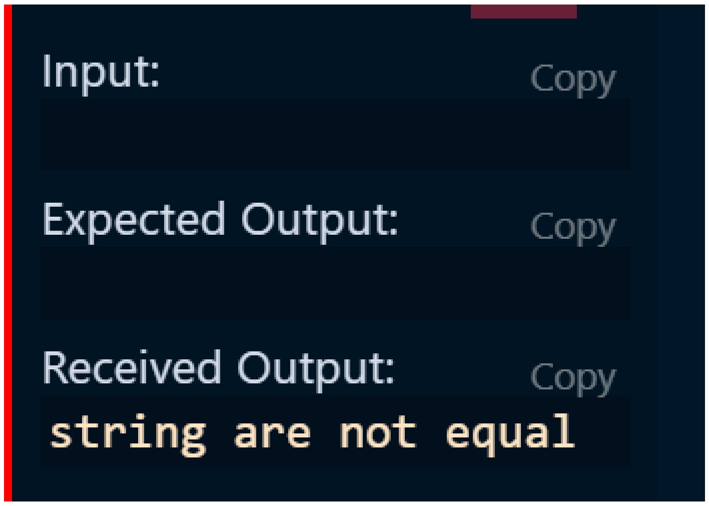


Input: Copy

Expected Output: Copy

Received Output: Copy

```
string are equal
```



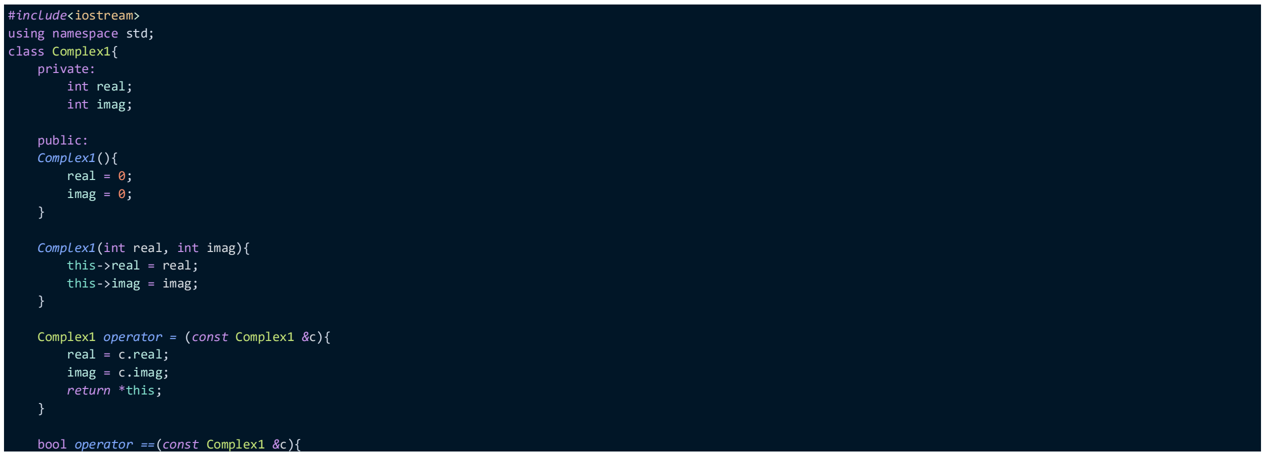
Input: Copy

Expected Output: Copy

Received Output: Copy

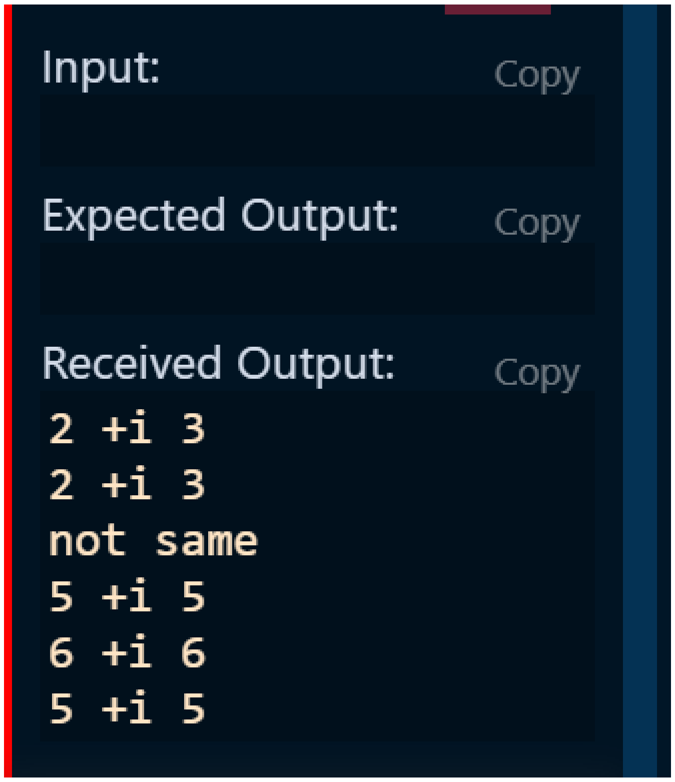
```
string are not equal
```

3. Create a Complex class that has real(int) and img(int) as member data, and has getData and showData functions. Then also overload the following operators for Complex class. =, ==, +, ++, --,



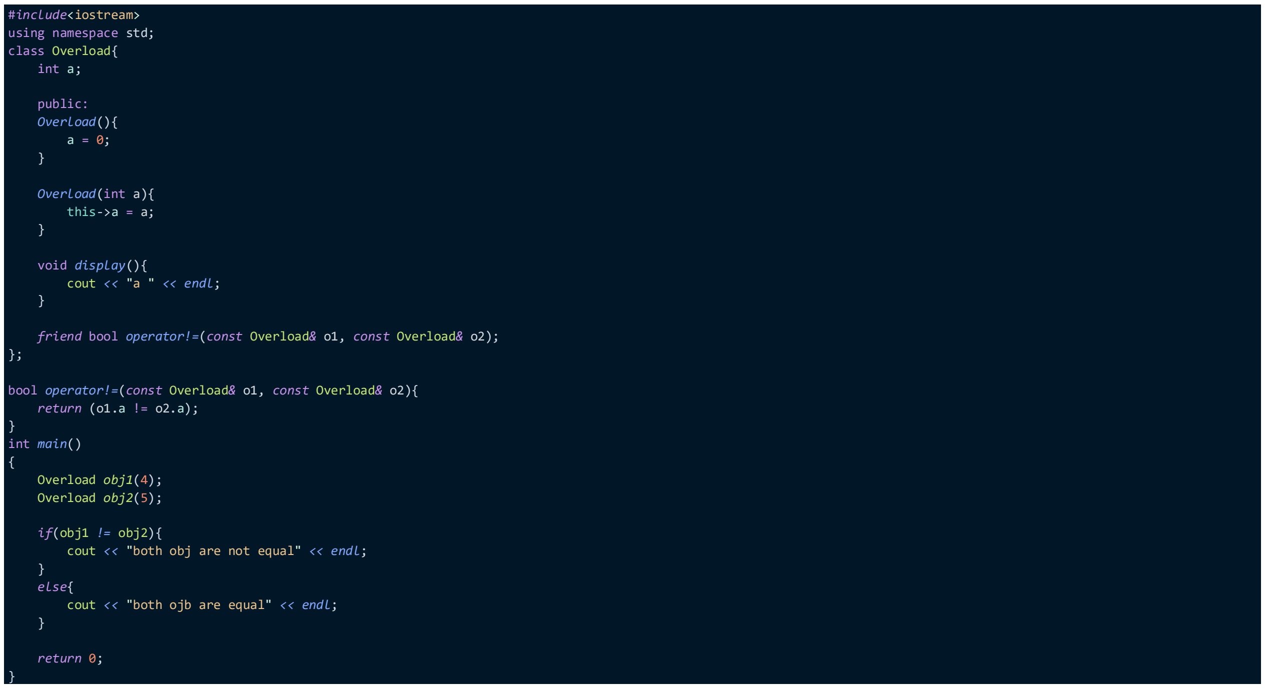
```
#include<iostream>
using namespace std;
class Complex1{
    private:
        int real;
        int imag;

    public:
    Complex1(){
        real = 0;
        imag = 0;
    }

    Complex1(int real, int imag){
        this->real = real;
        this->imag = imag;
    }

    Complex1 operator = (const Complex1 &c){
        real = c.real;
        imag = c.imag;
        return *this;
    }

    bool operator ==(const Complex1 &c){
```

```cpp
        return (real == c.real) && (imag = c.imag);
    }

    Complex1 operator+(const Complex1 &c){
        real += c.real;
        imag += c.imag;
        return *this;
    }

        Complex1
        operator++()
    {
        ++real;
        ++imag;
        return *this;
    }

    Complex1 operator--(){
        --real;
        --imag;
        return *this;
    }

    void display(){
        cout << real << " "<< "+i"<< " " << imag << endl;
    }

    ~Complex1(){}
};
int main()
{
    Complex1 c1(2,3);
    c1.display();

    Complex1 c2;
    c2 = c1;
    c2.display();

    Complex1 c3(3, 2);
    if(c3 == c1){
        cout << "same" << endl;
    }
    else{
        cout << "not same" << endl;
    }

    Complex1 c4;
    c4 = c1 + c3;
    c4.display();

    ++c4;
    c4.display();
    --c4;
    c4.display();

    return 0;
}
```

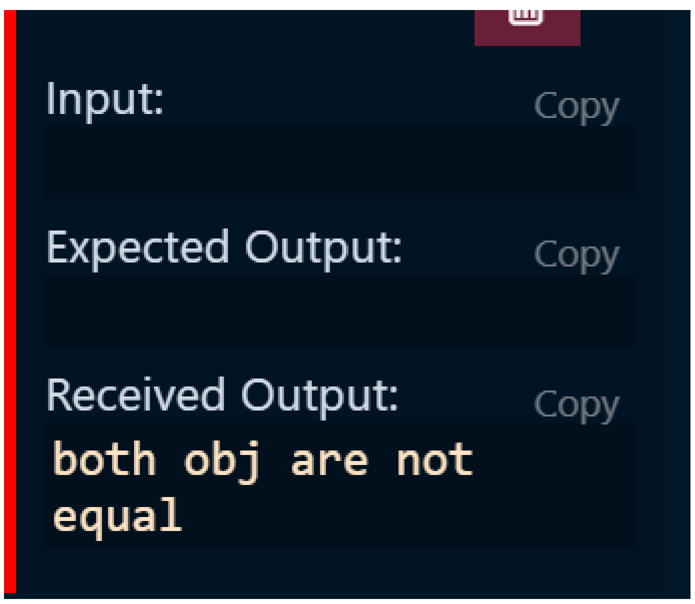


```
2 +i 3
2 +i 3
not same
5 +i 5
6 +i 6
5 +i 5
```

4. Write a C++ program to overload '!' operator using friend function



```cpp
#include<iostream>
using namespace std;
class Overload{
    int a;

    public:
    Overload(){
        a = 0;
    }

    Overload(int a){
        this->a = a;
    }

    void display(){
        cout << "a " << endl;
    }

    friend bool operator!=(const Overload& o1, const Overload& o2);
};

bool operator!=(const Overload& o1, const Overload& o2){
    return (o1.a != o2.a);
}
int main()
{
    Overload obj1(4);
    Overload obj2(5);

    if(obj1 != obj2){
        cout << "both obj are not equal" << endl;
    }
    else{
        cout << "both ojb are equal" << endl;
    }

    return 0;
}
```

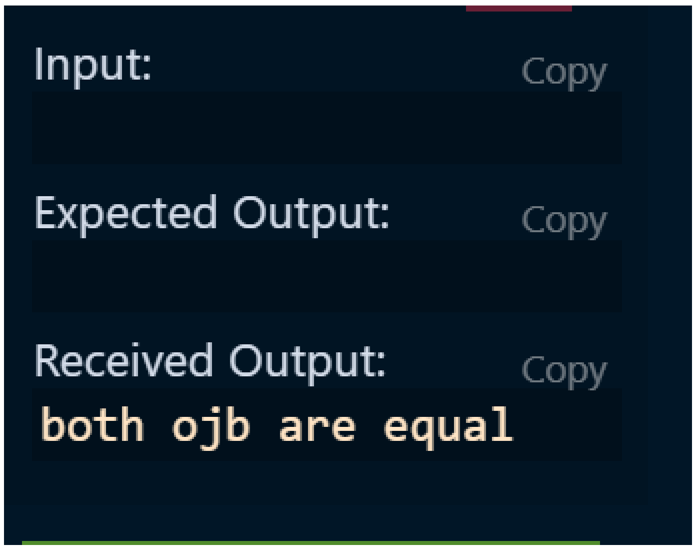


Input: Copy

Expected Output: Copy

Received Output: Copy

```
both obj are not
equal
```



Input: Copy

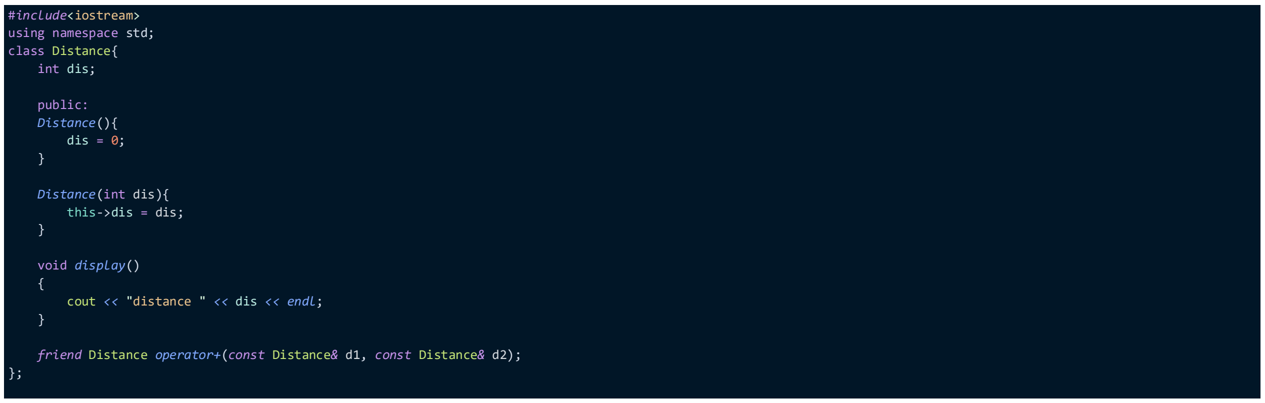
Expected Output: Copy

Received Output: Copy

```
both ojb are equal
```

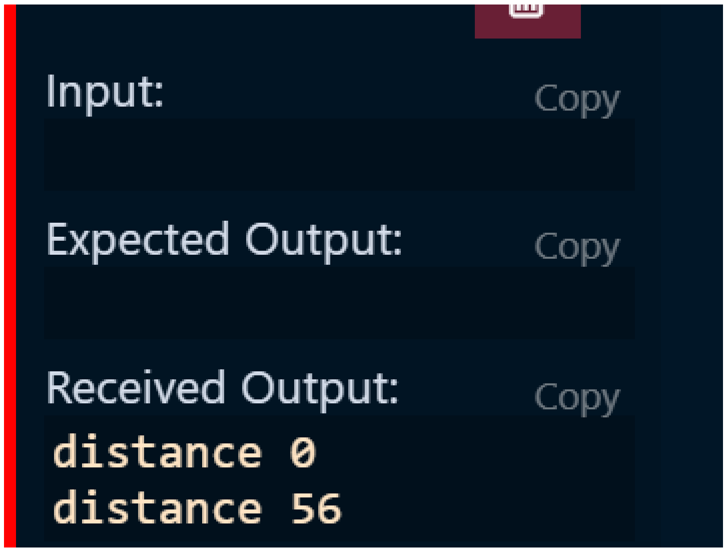
5. Read a value of distance from one object and add with a value in

another object using friend function.



```cpp
#include<iostream>
using namespace std;
class Distance{
    int dis;

    public:
    Distance(){
        dis = 0;
    }

    Distance(int dis){
        this->dis = dis;
    }

    void display()
    {
        cout << "distance " << dis << endl;
    }

    friend Distance operator+(const Distance& d1, const Distance& d2);
};
```

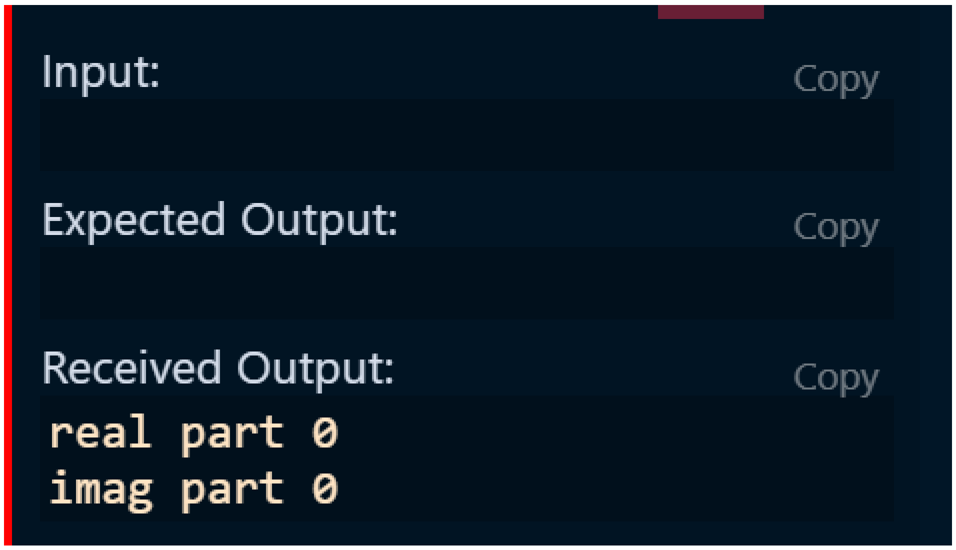
```cpp
Distance operator+(const Distance &d1, const Distance &d2)
{
    Distance result;
    result.dis = d1.dis + d2.dis;
    return result;
}
int main()
{
    Distance d1;
    d1.display();
    Distance d2(56);
    d1 = d2;
    d1.display();

    return 0;
}
```



```
distance 0
distance 56
```

## Insertion Operator

```cpp
#include<iostream>
using namespace std;
class Complex1{
    int real;
    int imag;

    public:
    Complex1(){
        cin >> real;
        cin >> imag;
    }

    Complex1(int real, int imag){
        this->real = real;
        this->imag = imag;
    }

    friend istream& operator >>(istream &i, Complex1 &c);

    void display()
    {
        cout << "real part " << real << endl;
        cout << "imag part " << imag << endl;
    }
};

istream& operator >>(istream &i, Complex1 &c){
    i >> c.real;
    i >> c.imag;
    return i;
}

int main()
{
    Complex1 c1;
    cin >> c1;
    c1.display();

    return 0;
}
```



Input:

Copy

Expected Output:

Copy

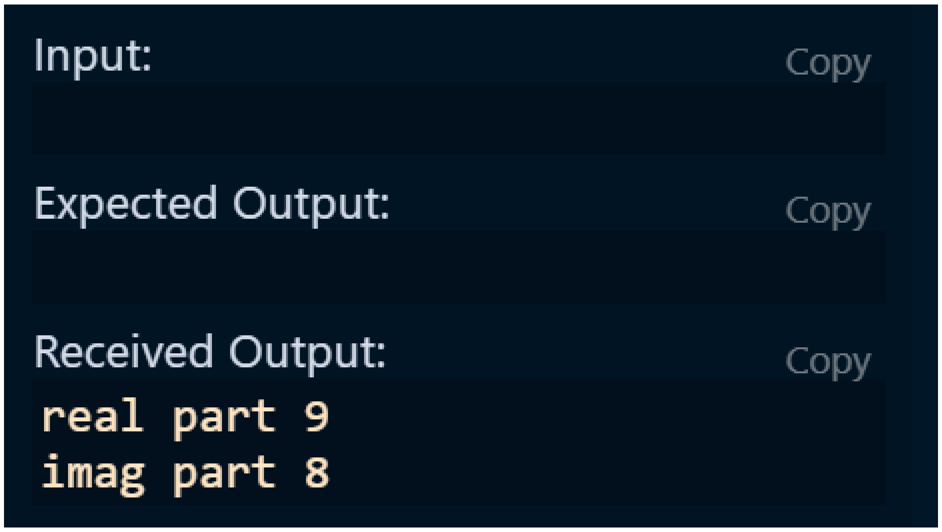
Received Output:

Copy

```
real part 0
imag part 0
```

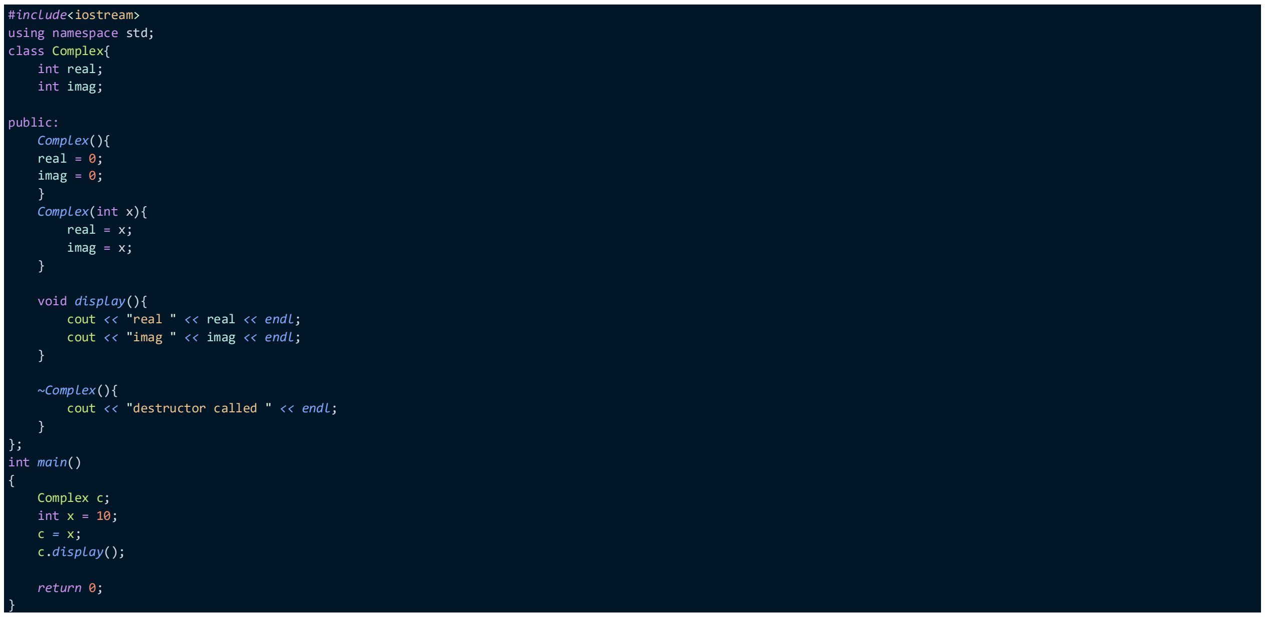
Extration Operator

```
#include<iostream>
using namespace std;
class Complex1{
    int real;
    int imag;

    public:
    Complex1(){
        real = 0;
        imag = 0;
    }

    Complex1(int real, int imag){
        this->real = real;
        this->imag = imag;
    }

    friend ostream &operator<<(ostream &o, const Complex1 &c);
};

ostream& operator <<(ostream &o, const Complex1 &c){
    o << "real part " << c.real << endl;
    o << "imag part " << c.imag << endl;
    return o;
}
int main()
{
    Complex1 c(9, 8);
    cout << c;

    return 0;
}
```

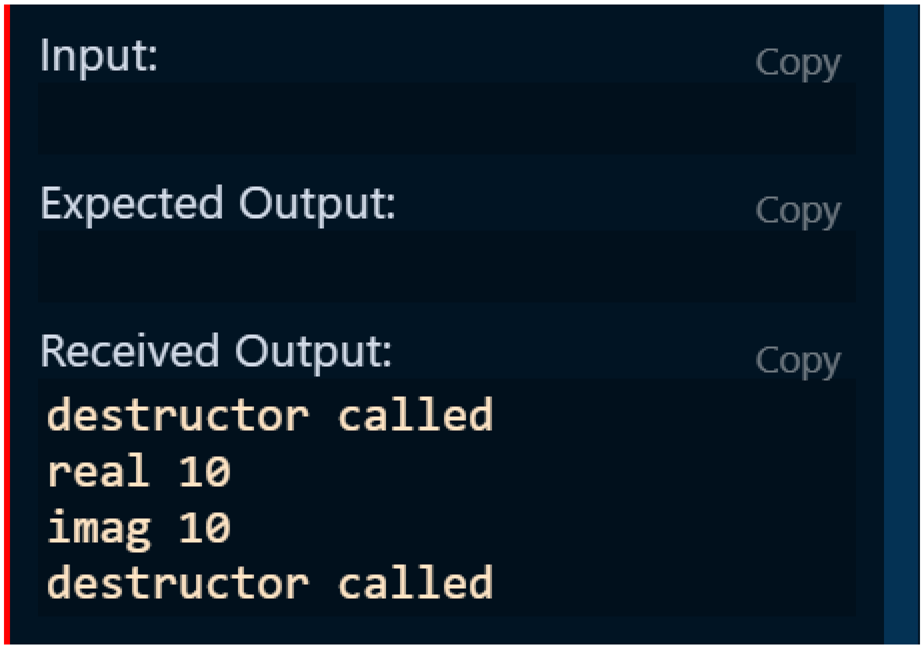


```
real part 9
imag part 8
```

Built in to Class type

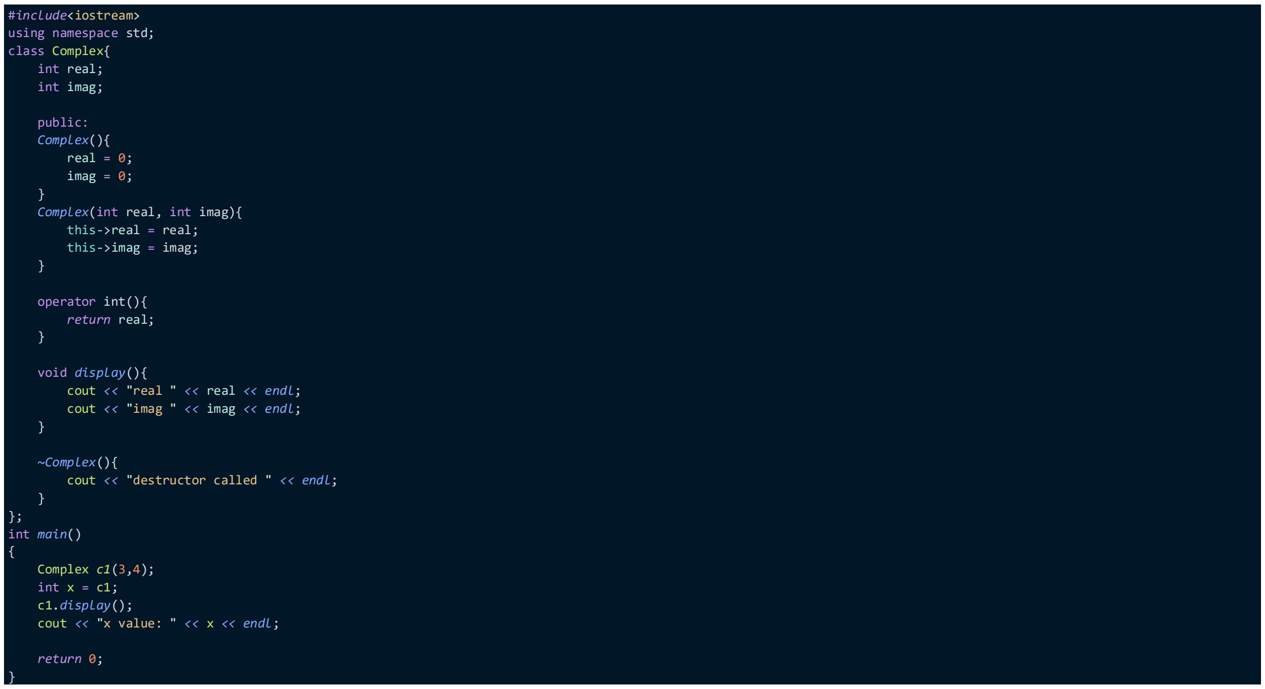


```
#include<iostream>
using namespace std;
class Complex{
    int real;
    int imag;

public:
    Complex(){
    real = 0;
    imag = 0;
    }
    Complex(int x){
        real = x;
        imag = x;
    }

    void display(){
        cout << "real " << real << endl;
        cout << "imag " << imag << endl;
    }

    ~Complex(){
        cout << "destructor called " << endl;
    }
};
int main()
{
    Complex c;
    int x = 10;
    c = x;
    c.display();

    return 0;
}
```

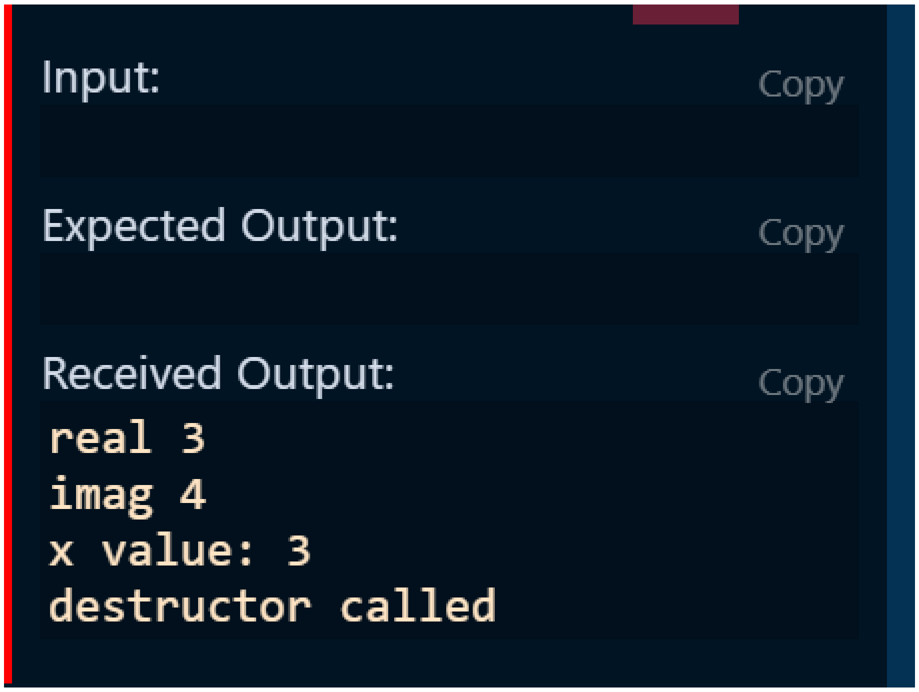


```
destructor called
real 10
imag 10
destructor called
```

Class type to Built in type

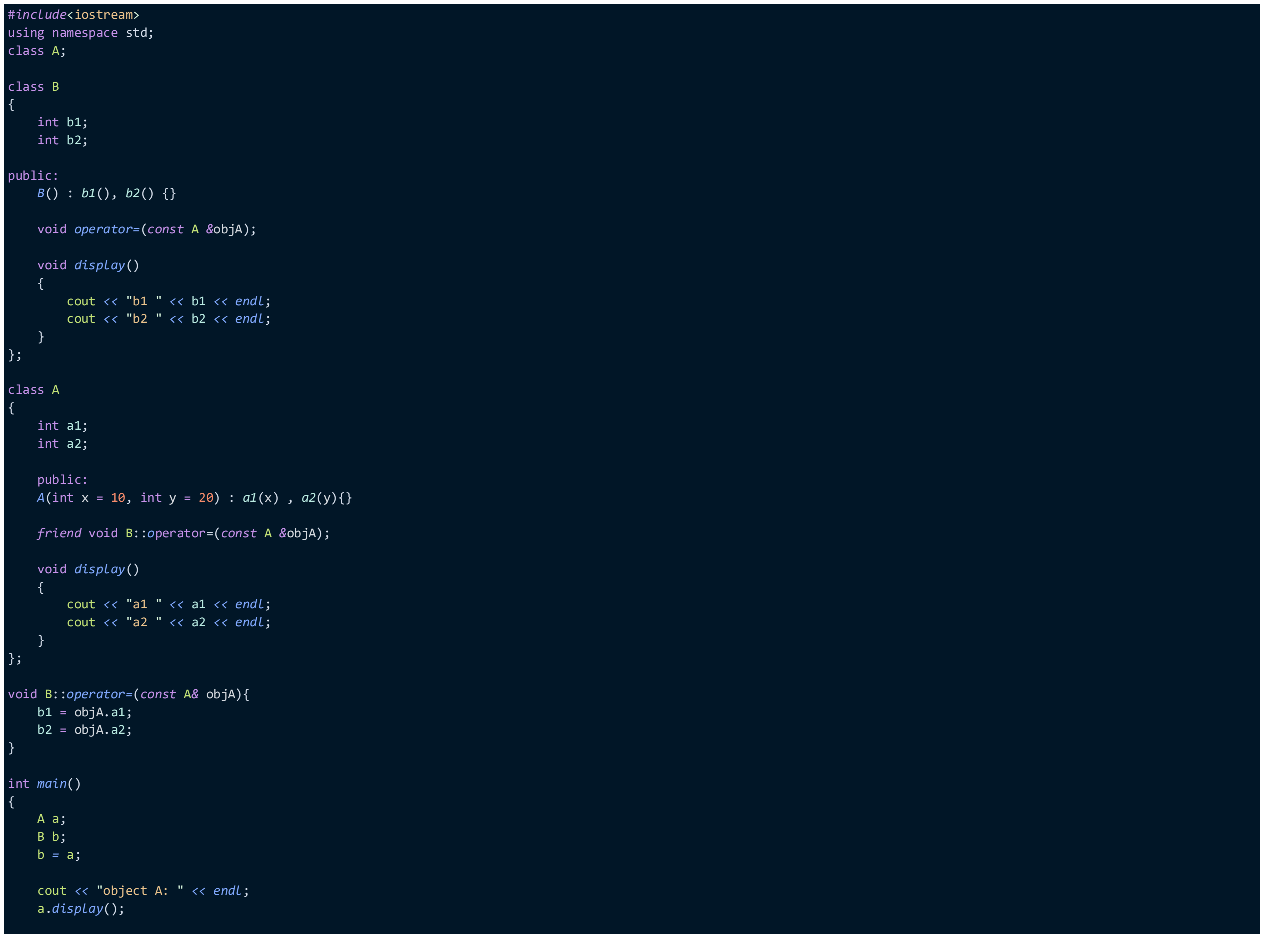


```cpp
#include<iostream>
using namespace std;
class Complex{
    int real;
    int imag;

    public:
    Complex(){
        real = 0;
        imag = 0;
    }
    Complex(int real, int imag){
        this->real = real;
        this->imag = imag;
    }

    operator int(){
        return real;
    }

    void display(){
        cout << "real " << real << endl;
        cout << "imag " << imag << endl;
    }

    ~Complex(){
        cout << "destructor called " << endl;
    }
};
int main()
{
    Complex c1(3,4);
    int x = c1;
    c1.display();
    cout << "x value: " << x << endl;

    return 0;
}
```

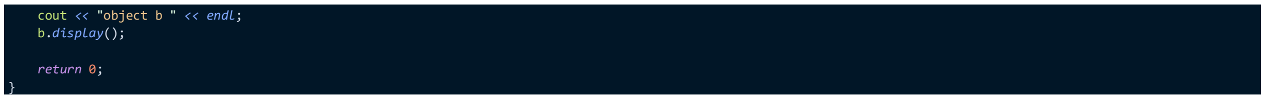


```
real 3
imag 4
x value: 3
destructor called
```

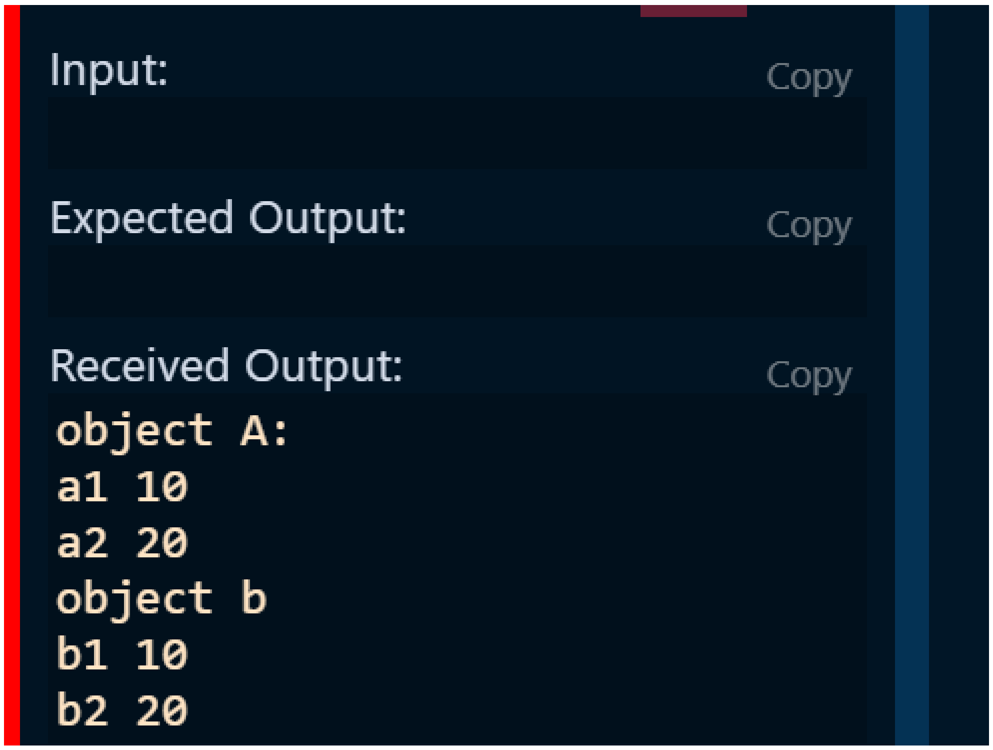
Class type to Another Class type



```cpp
#include<iostream>
using namespace std;
class A;

class B
{
    int b1;
    int b2;

public:
    B() : b1(), b2() {}

    void operator=(const A &objA);

    void display()
    {
        cout << "b1 " << b1 << endl;
        cout << "b2 " << b2 << endl;
    }
};

class A
{
    int a1;
    int a2;

    public:
    A(int x = 10, int y = 20) : a1(x) , a2(y){}

    friend void B::operator=(const A &objA);

    void display()
    {
        cout << "a1 " << a1 << endl;
        cout << "a2 " << a2 << endl;
    }
};

void B::operator=(const A& objA){
    b1 = objA.a1;
    b2 = objA.a2;
}

int main()
{
    A a;
    B b;
    b = a;

    cout << "object A: " << endl;
    a.display();
```



```
    cout << "object b " << endl;
    b.display();

    return 0;
}
```



Input:

Expected Output:

Received Output:

```
object A:
a1 10
a2 20
object b
b1 10
b2 20
```