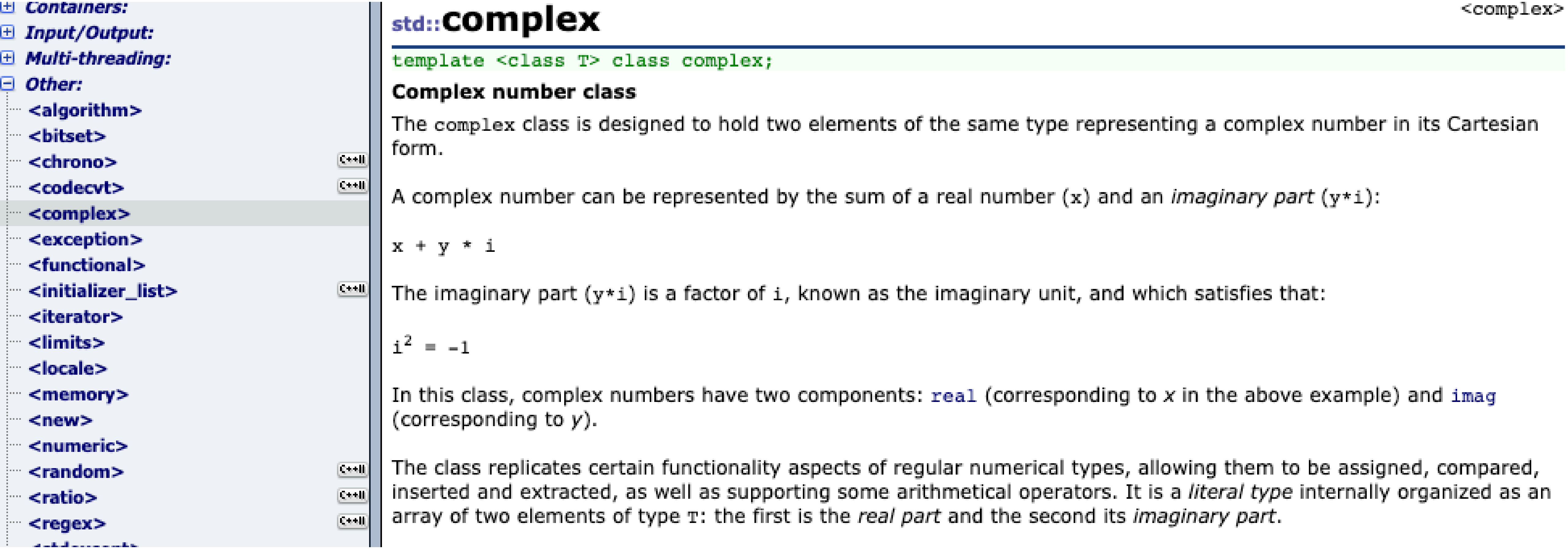
연산자 중복과 friend 선언

2023

국민대학교 소프트웨어학부

복소수 클래스

복소수: $a + bi$



std::complex

```
template <class T> class complex;
```

Complex number class

The `complex` class is designed to hold two elements of the same type representing a complex number in its Cartesian form.

A complex number can be represented by the sum of a real number (`x`) and an *imaginary part* (`y*i`):

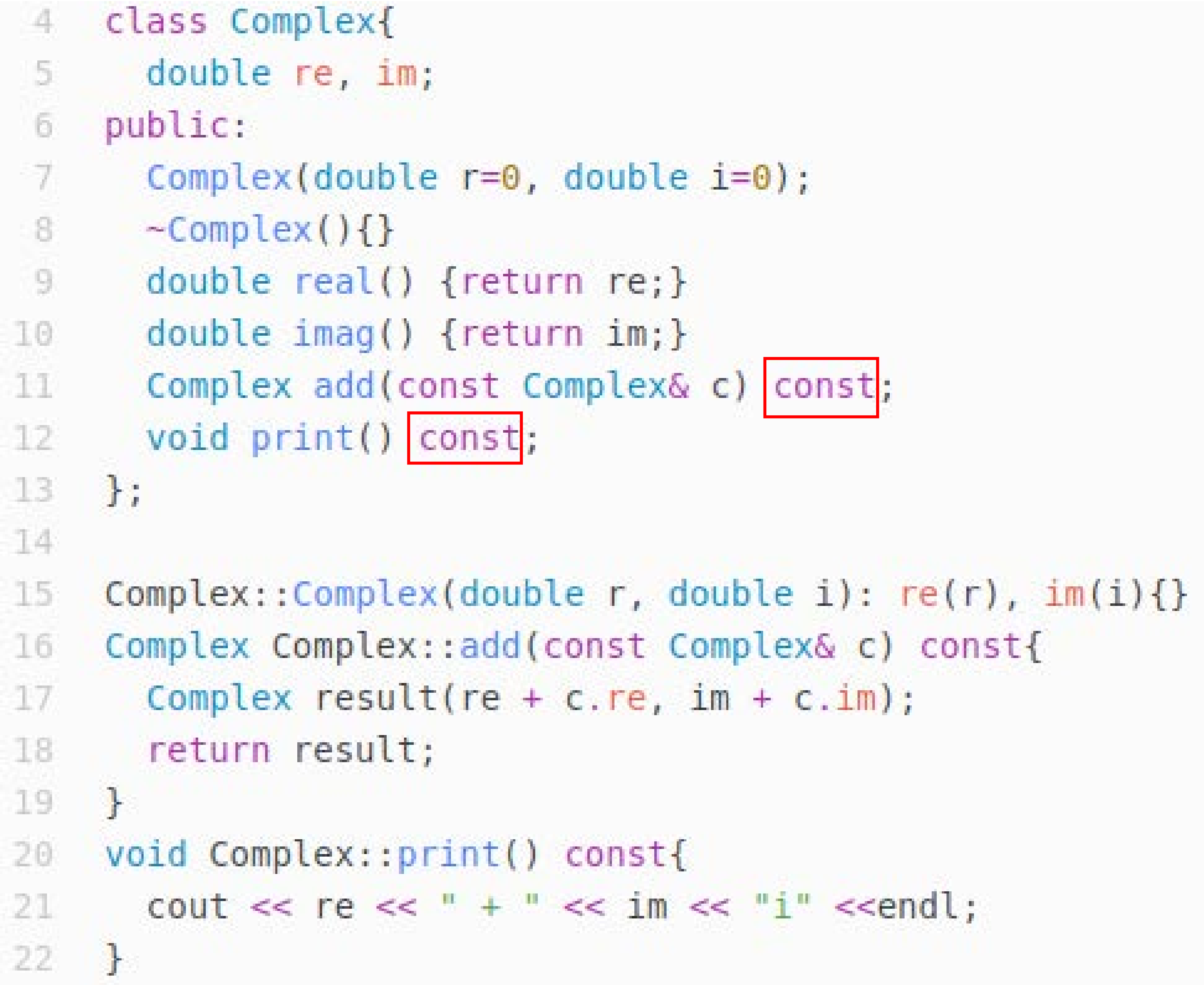
```
x + y * i
```

The imaginary part (`y*i`) is a factor of `i`, known as the imaginary unit, and which satisfies that:

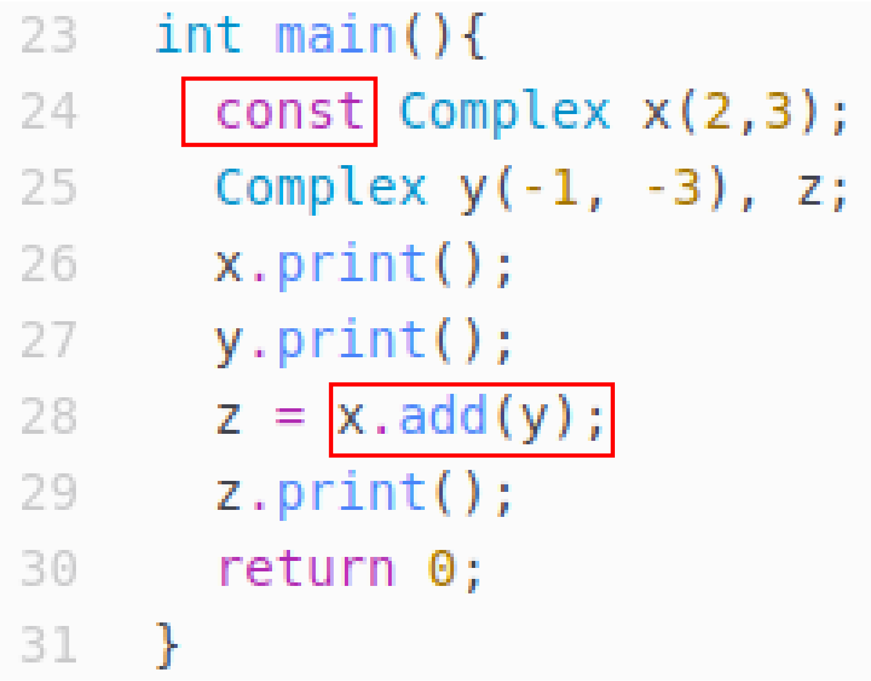
$$i^2 = -1$$

In this class, complex numbers have two components: `real` (corresponding to *x* in the above example) and `imag` (corresponding to *y*).

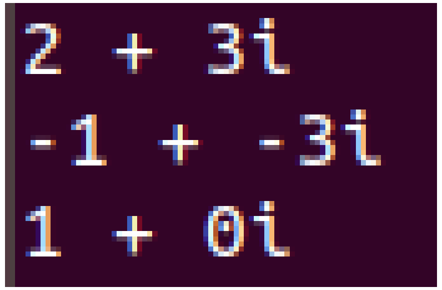
The class replicates certain functionality aspects of regular numerical types, allowing them to be assigned, compared, inserted and extracted, as well as supporting some arithmetical operators. It is a *literal type* internally organized as an array of two elements of type `T`: the first is the *real part* and the second its *imaginary part*.



```cpp
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0);
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex add(const Complex& c) const;
  void print() const;
};

Complex::Complex(double r, double i): re(r), im(i){}
Complex Complex::add(const Complex& c) const{
  Complex result(re + c.re, im + c.im);
  return result;
}
void Complex::print() const{
  cout << re << " + " << im << "i" <<endl;
}
```

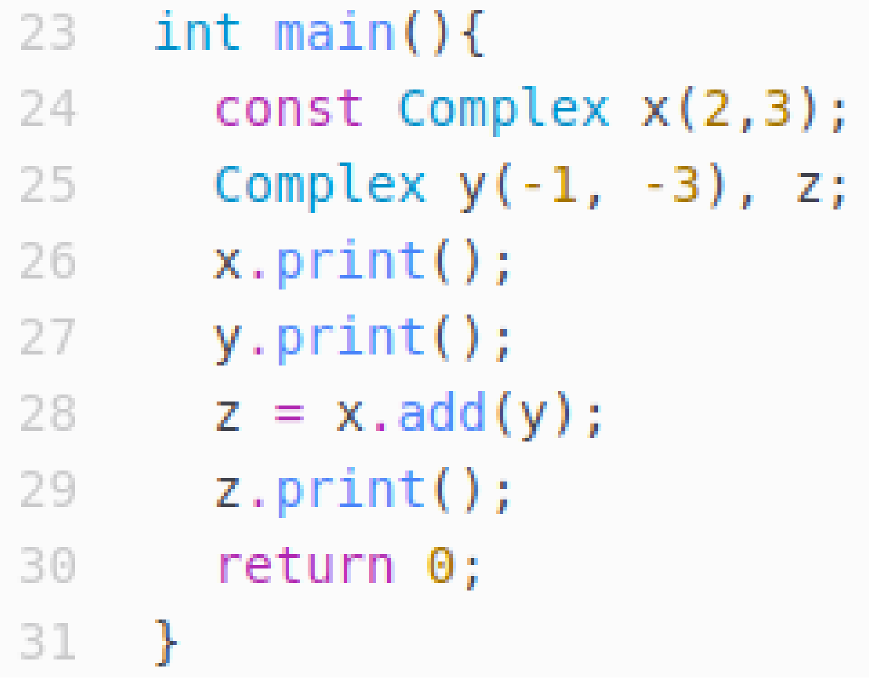


```cpp
int main(){
  const Complex x(2,3);
  Complex y(-1, -3), z;
  x.print();
  y.print();
  z = x.add(y);
  z.print();
  return 0;
}
```



```
2 + 3i
-1 + -3i
1 + 0i
```

operator overloading : 연산자를 함수로 구현



```
int main(){
  const Complex x(2,3);
  Complex y(-1, -3), z;
  x.print();
  y.print();
  z = x.add(y);
  z.print();
  return 0;
}
```

z = x + y;

```
반환형 operator연산자(매개 변수 목록)
{
....// 연산 수행
}
```

연산자 중복

- 일반적으로는 연산자 기호를 사용하는 편이 함수를 사용하는 것보다 이해하기가 쉽다.

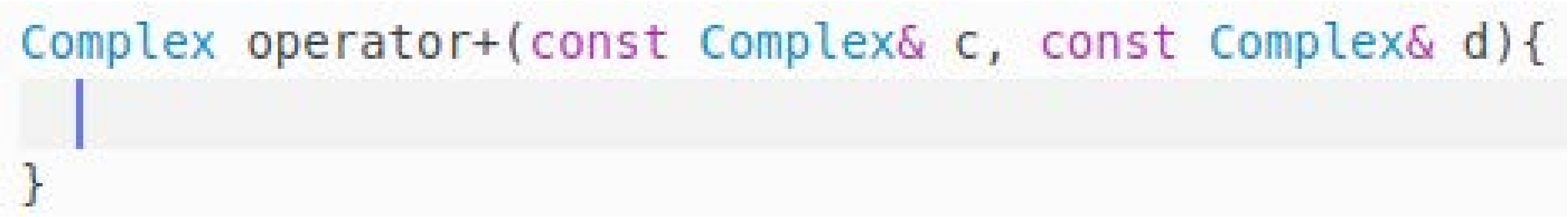
- 다음의 두 가지 문장 중에서 어떤 것이 더 이해하기 쉬운가?

1. sum = x + y + z;
2. sum = add(x, add(y, z));

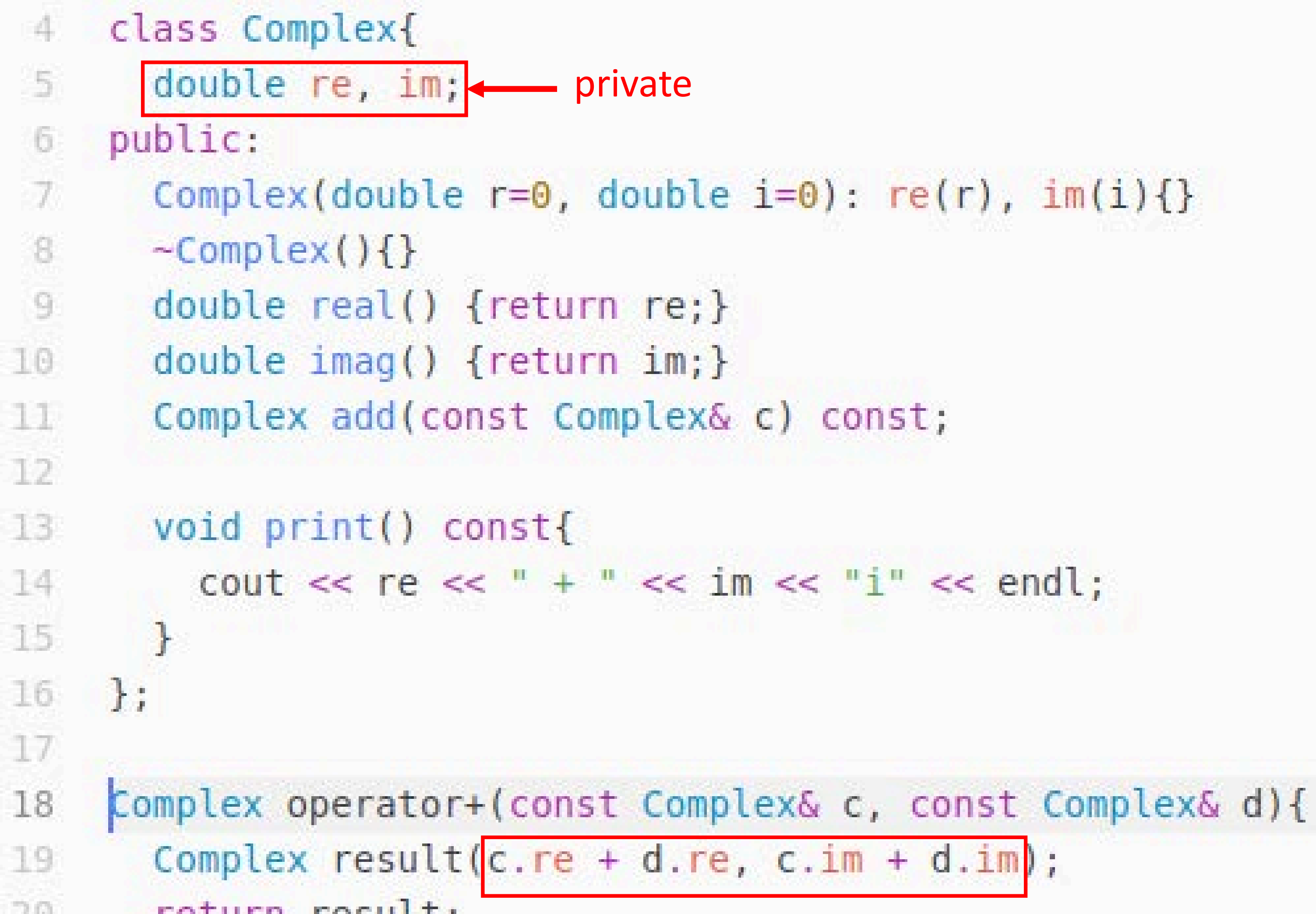
연산자 중복

- 연산자 중복(operator overloading): 여러 가지 연산자들을 클래스 객체에 대해서도 적용하는 것
- C++에서 연산자는 함수로 정의

```
반환형 operator연산자(매개 변수 목록)
{
....// 연산 수행
}
```

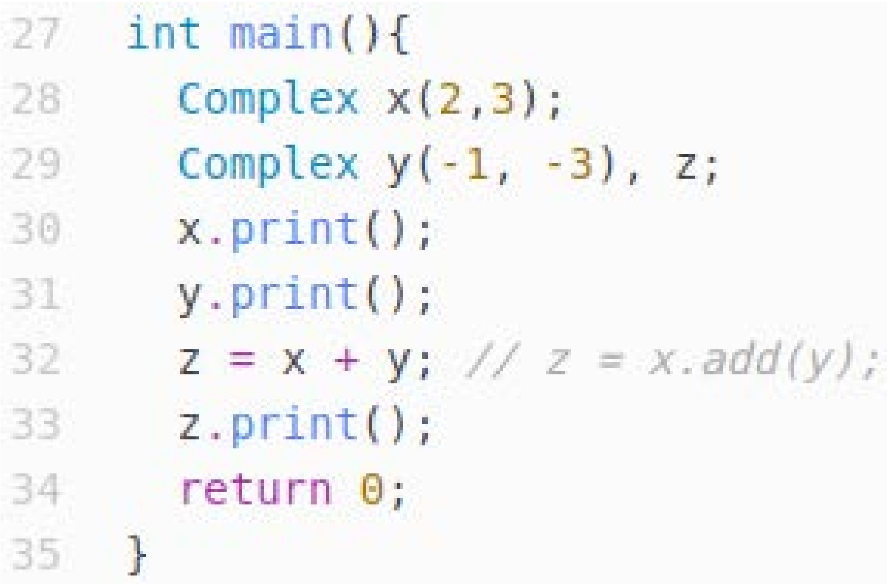


```
Complex operator+(const Complex& c, const Complex& d){
    
}
```



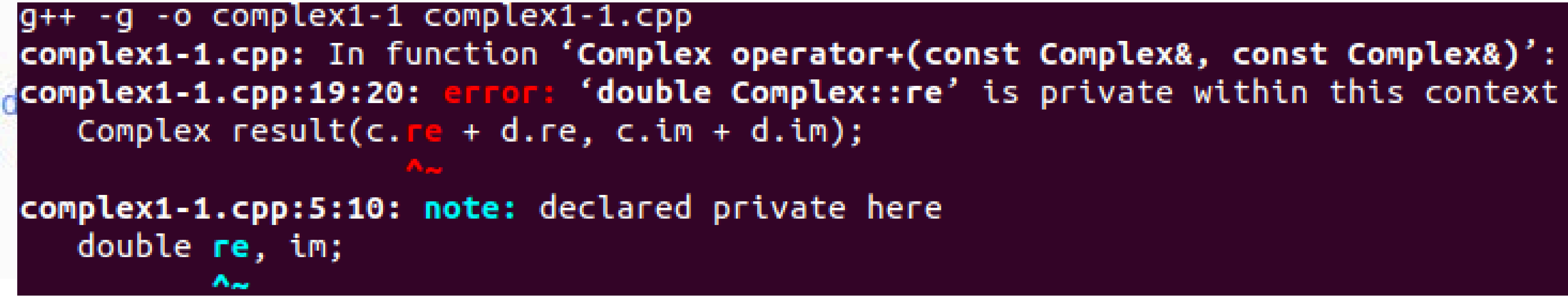
```
4   class Complex{
5     double re, im;
6   public:
7     Complex(double r=0, double i=0): re(r), im(i){}
8     ~Complex(){}
9     double real() {return re;}
10    double imag() {return im;}
11    Complex add(const Complex& c) const;
12
13    void print() const{
14      cout << re << " + " << im << "i" << endl;
15    }
16  };
17
18  Complex operator+(const Complex& c, const Complex& d){
19    Complex result(c.re + d.re, c.im + d.im);
20    return result;
21  }
22  Complex Complex::add
23    Complex result(re
24    return result;
25  }
```

private



```
27  int main(){
28    Complex x(2,3);
29    Complex y(-1, -3), z;
30    x.print();
31    y.print();
32    z = x + y; // z = x.add(y);
33    z.print();
34    return 0;
35  }
```

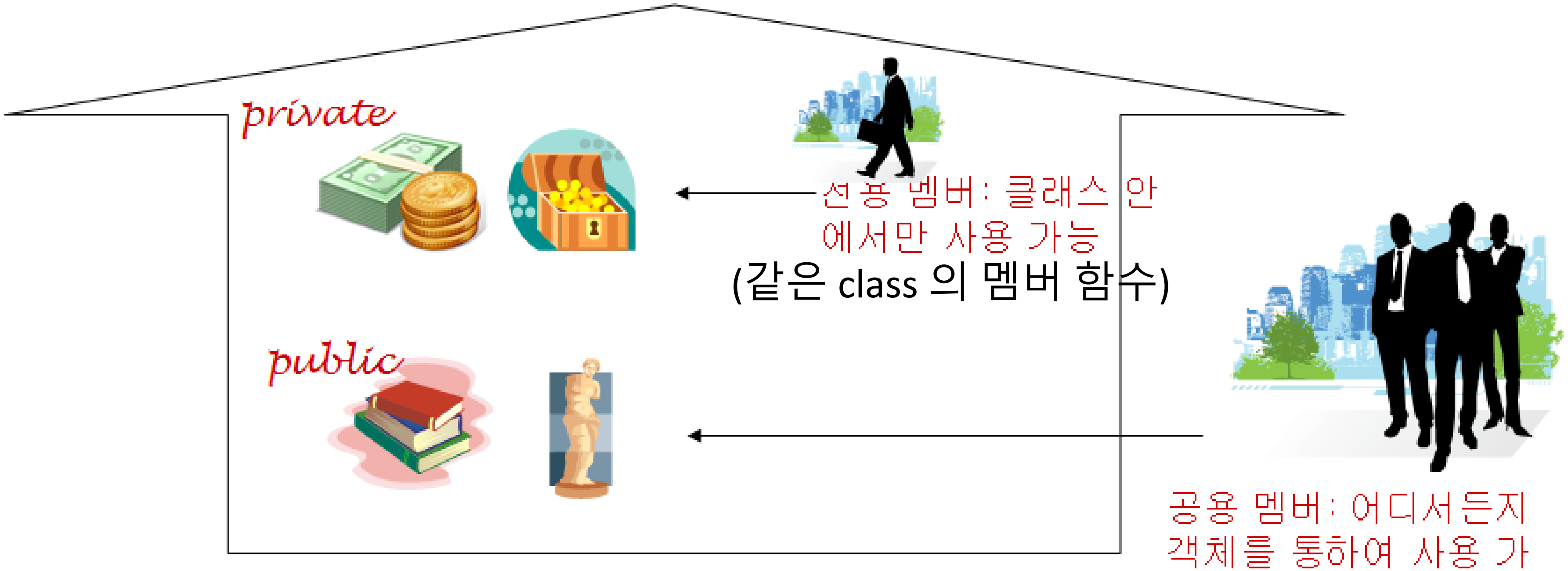
global 함수에서 Complex class 의 private member 를 접근 불가



```
g++ -g -o complex1-1 complex1-1.cpp
complex1-1.cpp: In function ‘Complex operator+(const Complex&, const Complex&)’:
complex1-1.cpp:19:20: error: ‘double Complex::re’ is private within this context
   Complex result(c.re + d.re, c.im + d.im);
                    ^~
complex1-1.cpp:5:10: note: declared private here
   double re, im;
          ^~
```

접근 제어자 private vs. public

클래스



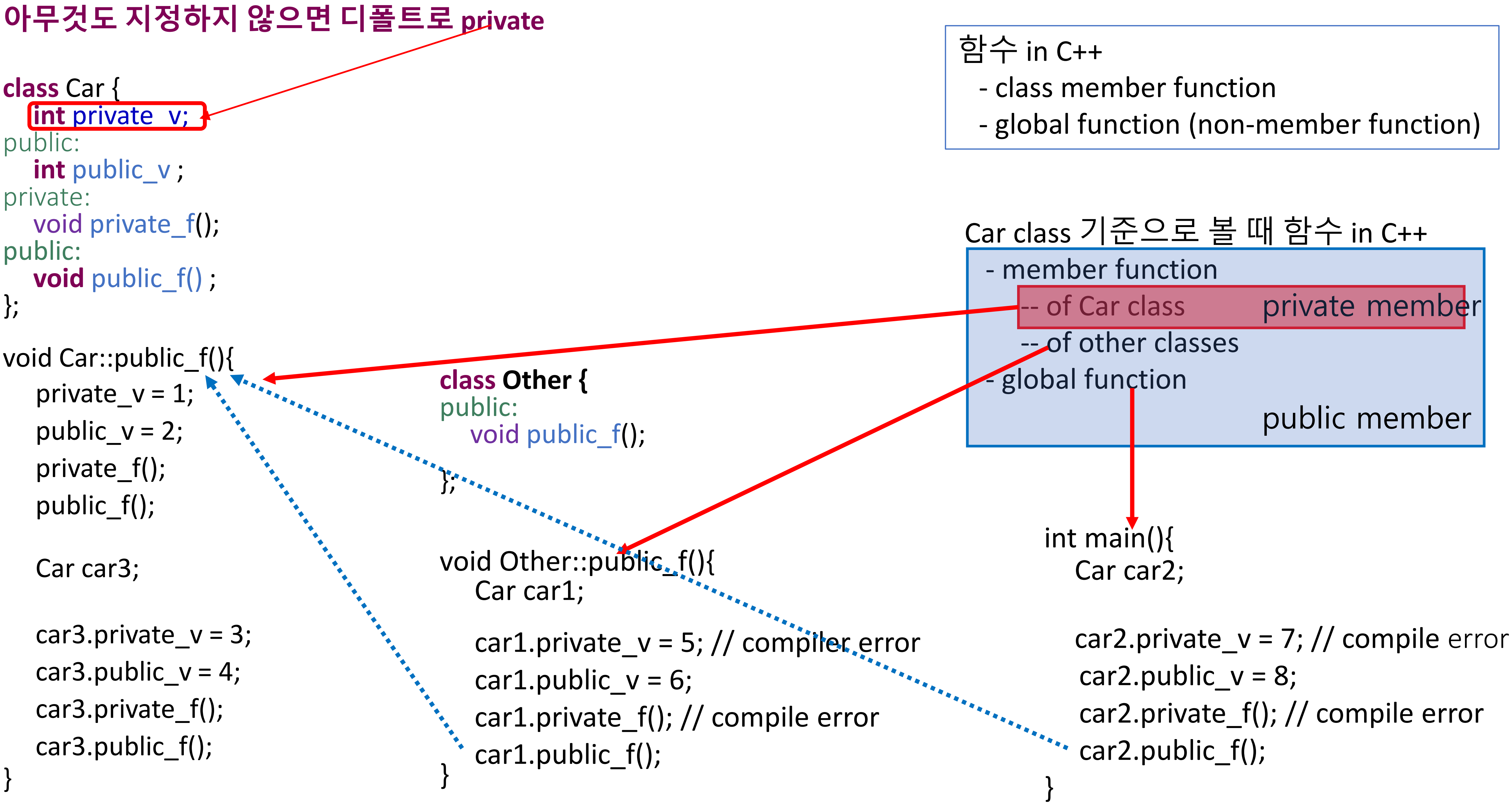
전용 멤버: 클래스 안에서만 사용 가능

(같은 class 의 멤버 함수)

공용 멤버: 어디서든지 객체를 통하여 사용 가능

(외부 함수, 다른 class의 멤버 함수)

private/public member variable/function



아무것도 지정하지 않으면 디폴트로 private

```
class Car {
   int private_v;
public:
   int public_v ;
private:
   void private_f();
public:
   void public_f() ;
};

void Car::public_f(){
    private_v = 1;
    public_v = 2;
    private_f();
    public_f();

    Car car3;

    car3.private_v = 3;
    car3.public_v = 4;
    car3.private_f();
    car3.public_f();
}
```

```
class Other {
public:
   void public_f();

};

void Other::public_f(){
    Car car1;

    car1.private_v = 5; // compiler error
    car1.public_v = 6;
    car1.private_f(); // compile error
    car1.public_f();
}
```

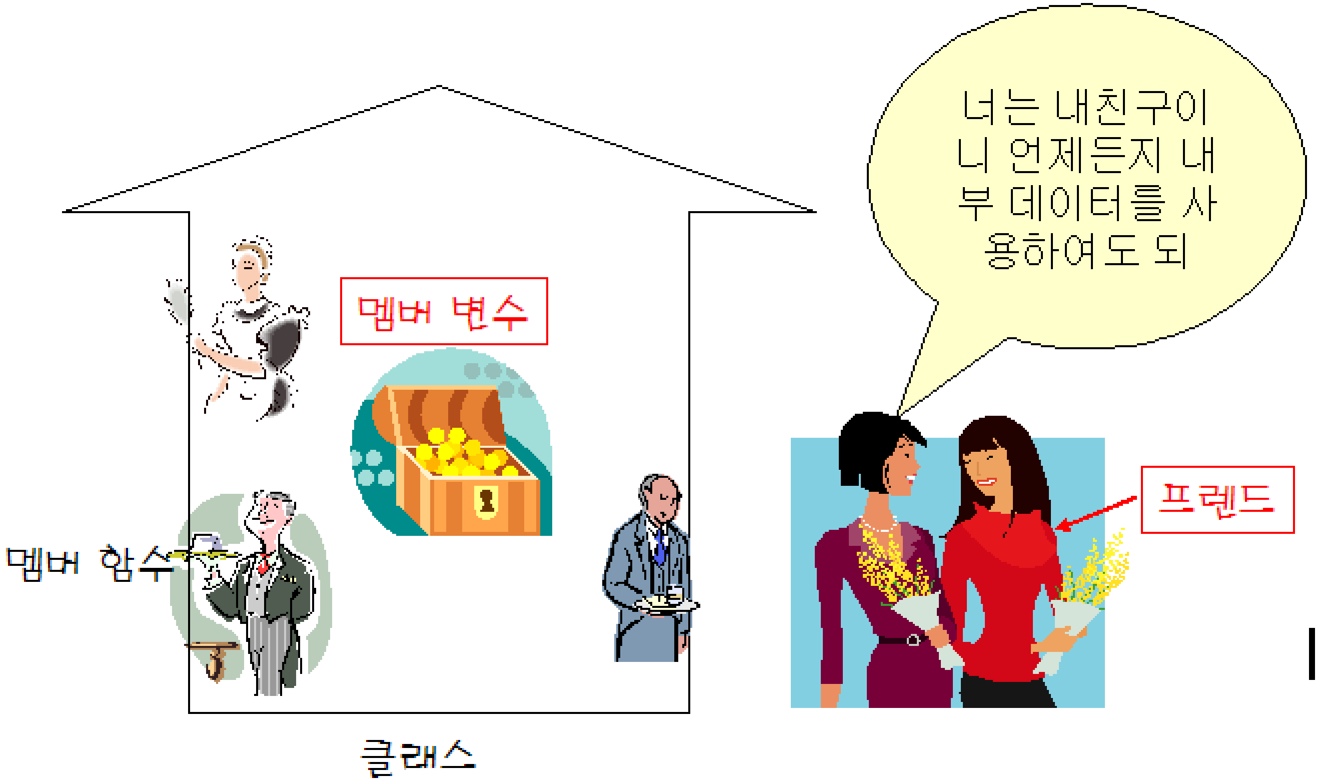
```
int main(){
    Car car2;

    car2.private_v = 7; // compile error
    car2.public_v = 8;
    car2.private_f(); // compile error
    car2.public_f();
}
```

함수 in C++
- class member function
- global function (non-member function)

Car class 기준으로 볼 때 함수 in C++
- member function
  -- of Car class   private member
  -- of other classes
- global function
    public member

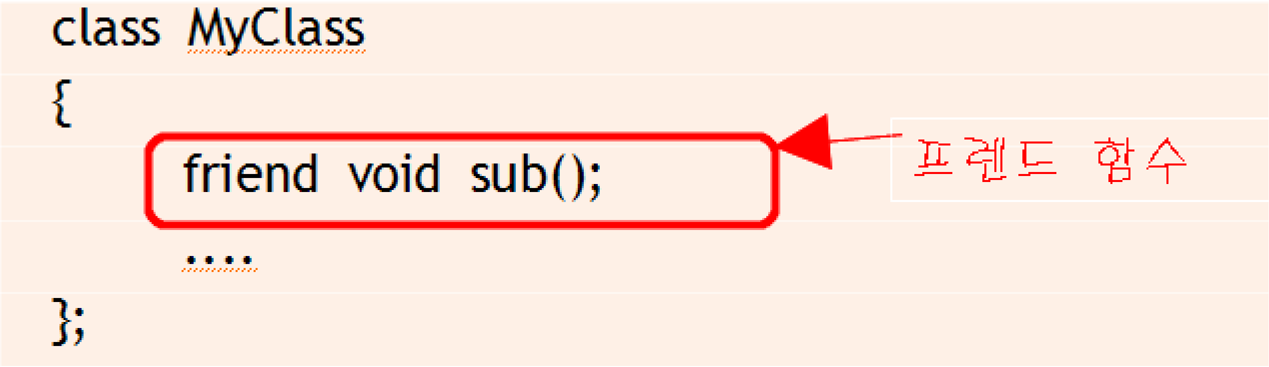
프렌드 함수

- *프렌드 함수(friend function):* 클래스의 내부 데이터에 접근할 수 있는 특수한 함수



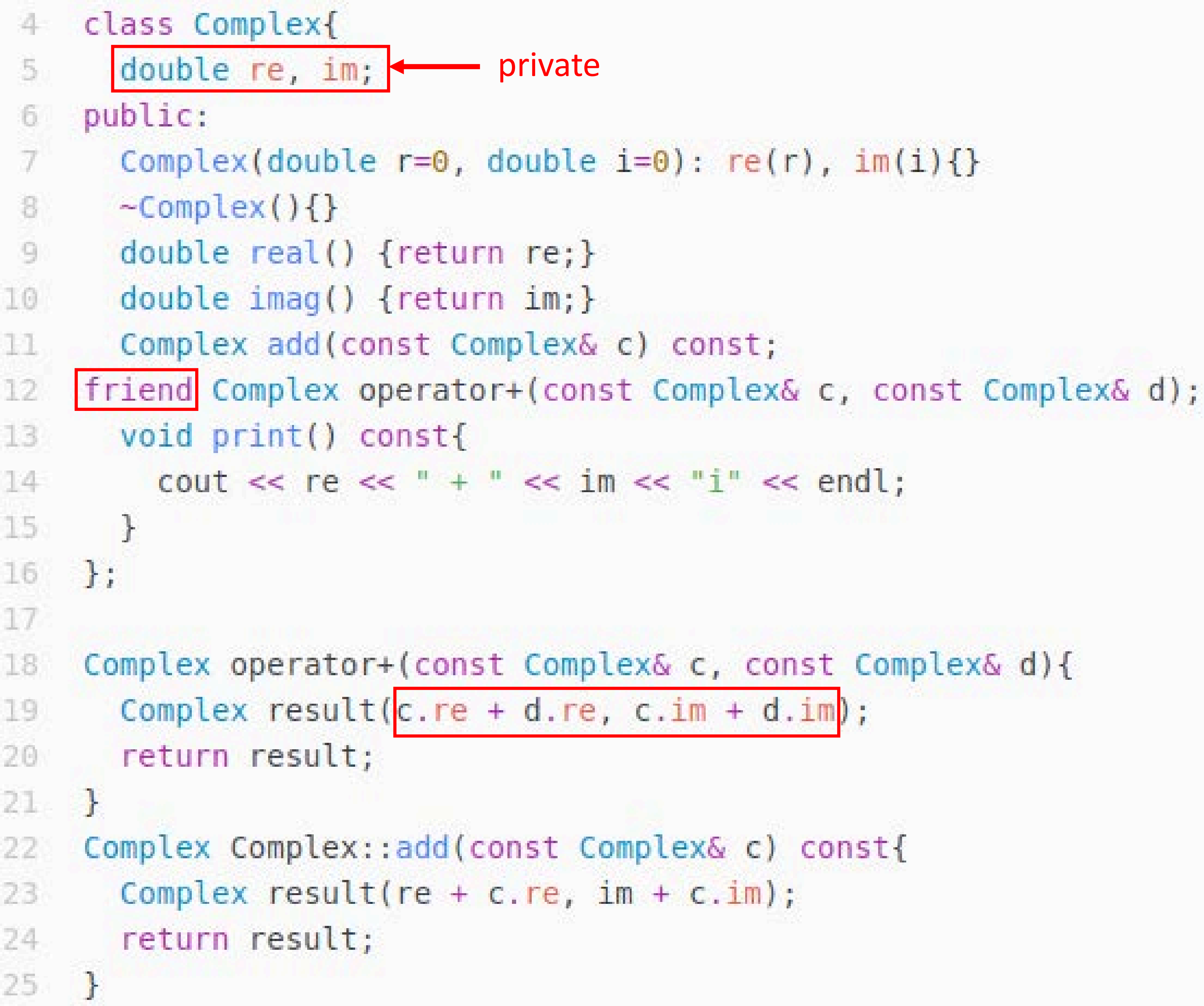
프렌드 함수 선언 방법

- 프렌드 함수의 원형은 비록 클래스 안에 포함하지만 멤버 함수는 아니다.
- 프렌드 함수의 본체는 외부에서 따로 정의
- 프렌드 함수는 클래스 내부의 모든 멤버 변수를 사용 가능



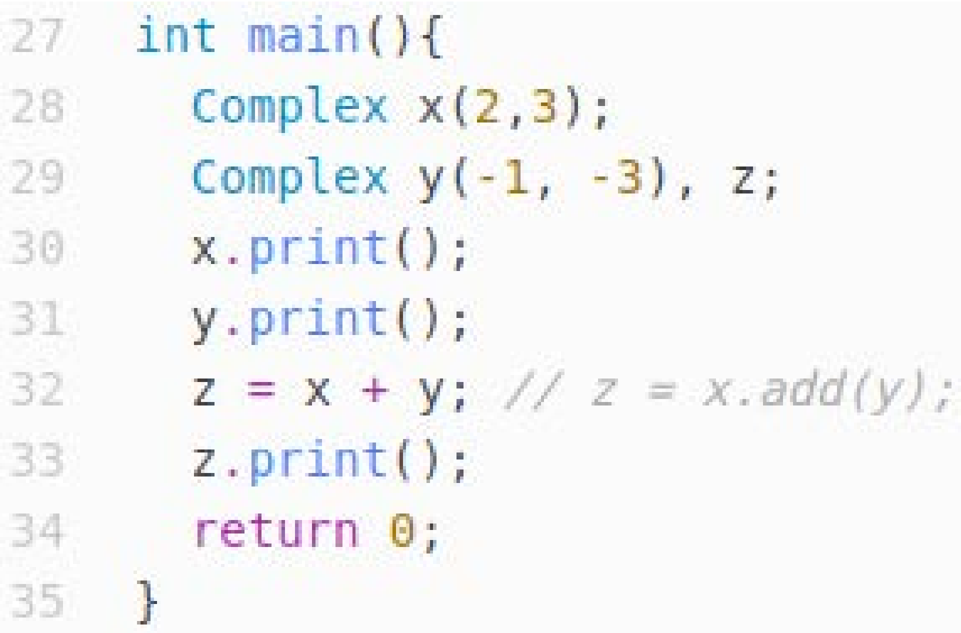
```
class MyClass
{
        friend void sub();
        ....
};
```

friend function operator+

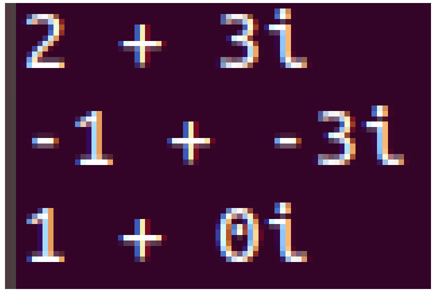


```
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex add(const Complex& c) const;
friend Complex operator+(const Complex& c, const Complex& d);
  void print() const{
    cout << re << " + " << im << "i" << endl;
  }
};

Complex operator+(const Complex& c, const Complex& d){
  Complex result(c.re + d.re, c.im + d.im);
  return result;
}
Complex Complex::add(const Complex& c) const{
  Complex result(re + c.re, im + c.im);
  return result;
}
```

private



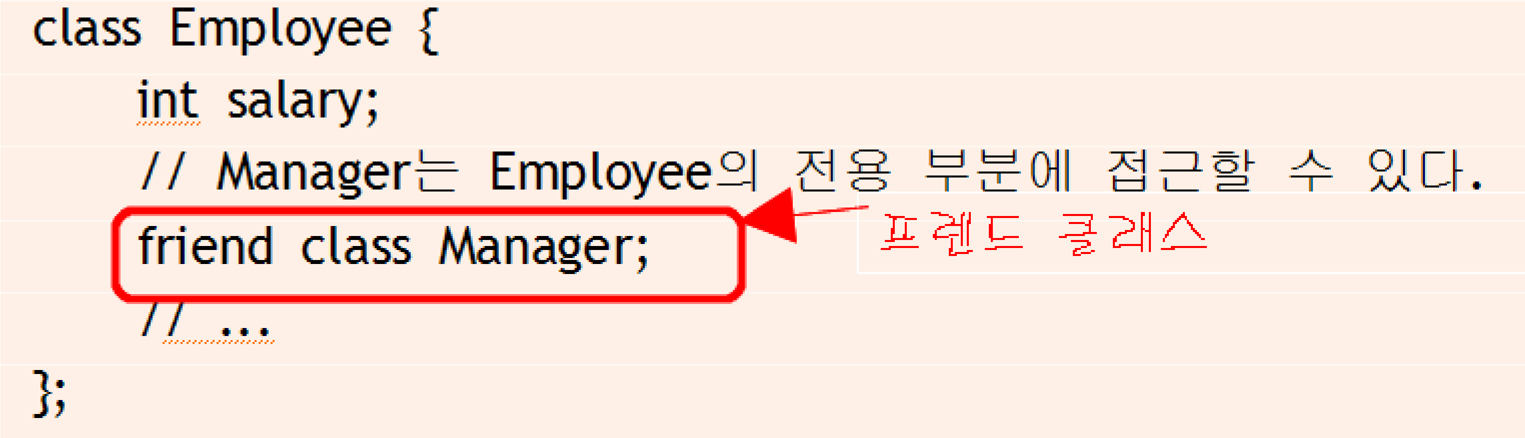
```
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  x.print();
  y.print();
  z = x + y; // z = x.add(y);
  z.print();
  return 0;
}
```



```
2 + 3i
-1 + -3i
1 + 0i
```

프렌드 클래스

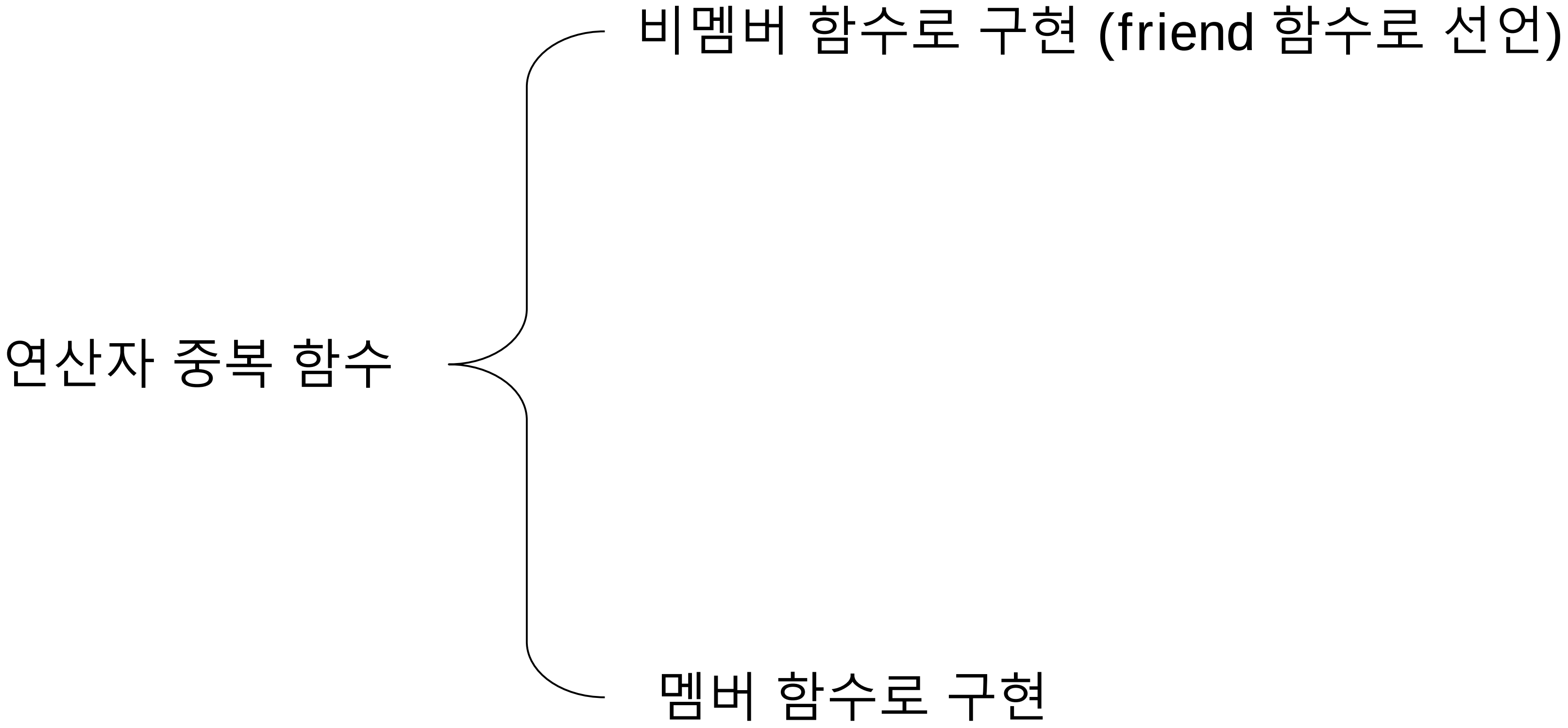
- 클래스도 프렌드로 선언할 수 있다.
- (예) Manager 객체의 멤버 함수들은 Employee 객체의 전용 멤버를 직접 참조할 수 있다.
- 교환 법칙은 성립하지 않는다. (class A 가 class B 의 friend 로 선언된다고 해서 class B 가 class A 의 friend 가 되는 것이 아니다.)



```
class Employee {
    int salary;
    // Manager는 Employee의 전용 부분에 접근할 수 있다.
    friend class Manager;
    // ...
};
```

프렌드 클래스

연산자 중복 구현의 방법

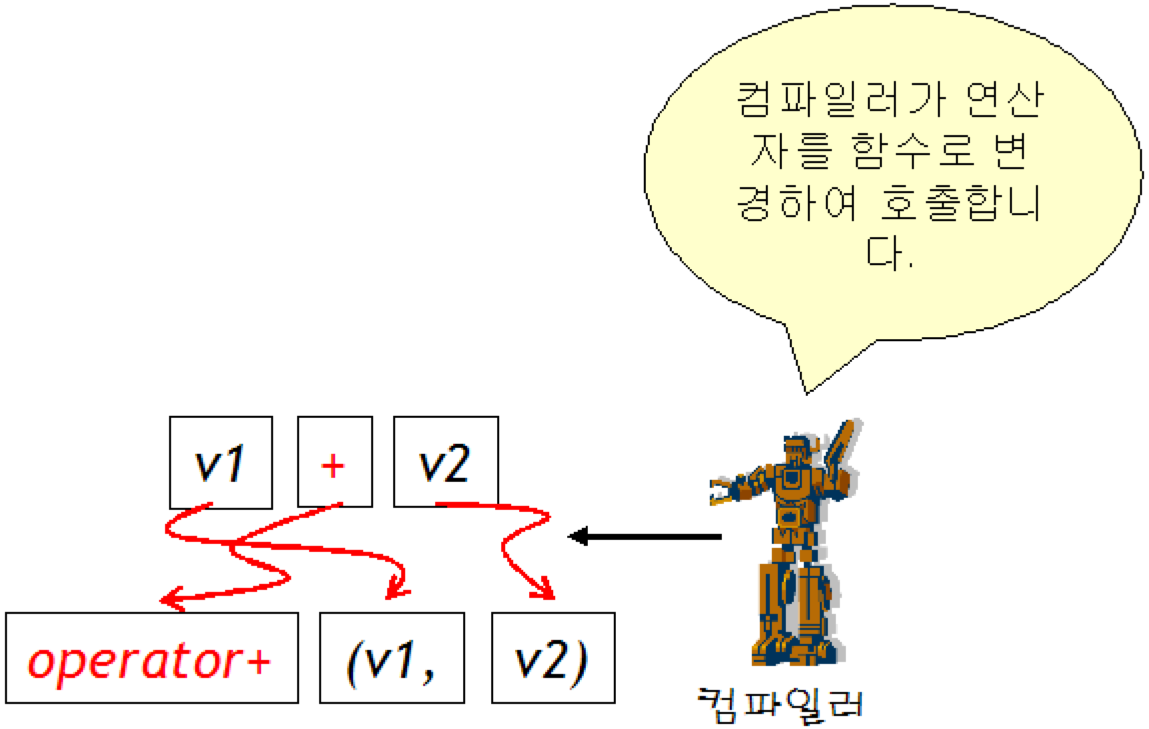


연산자 중복 함수

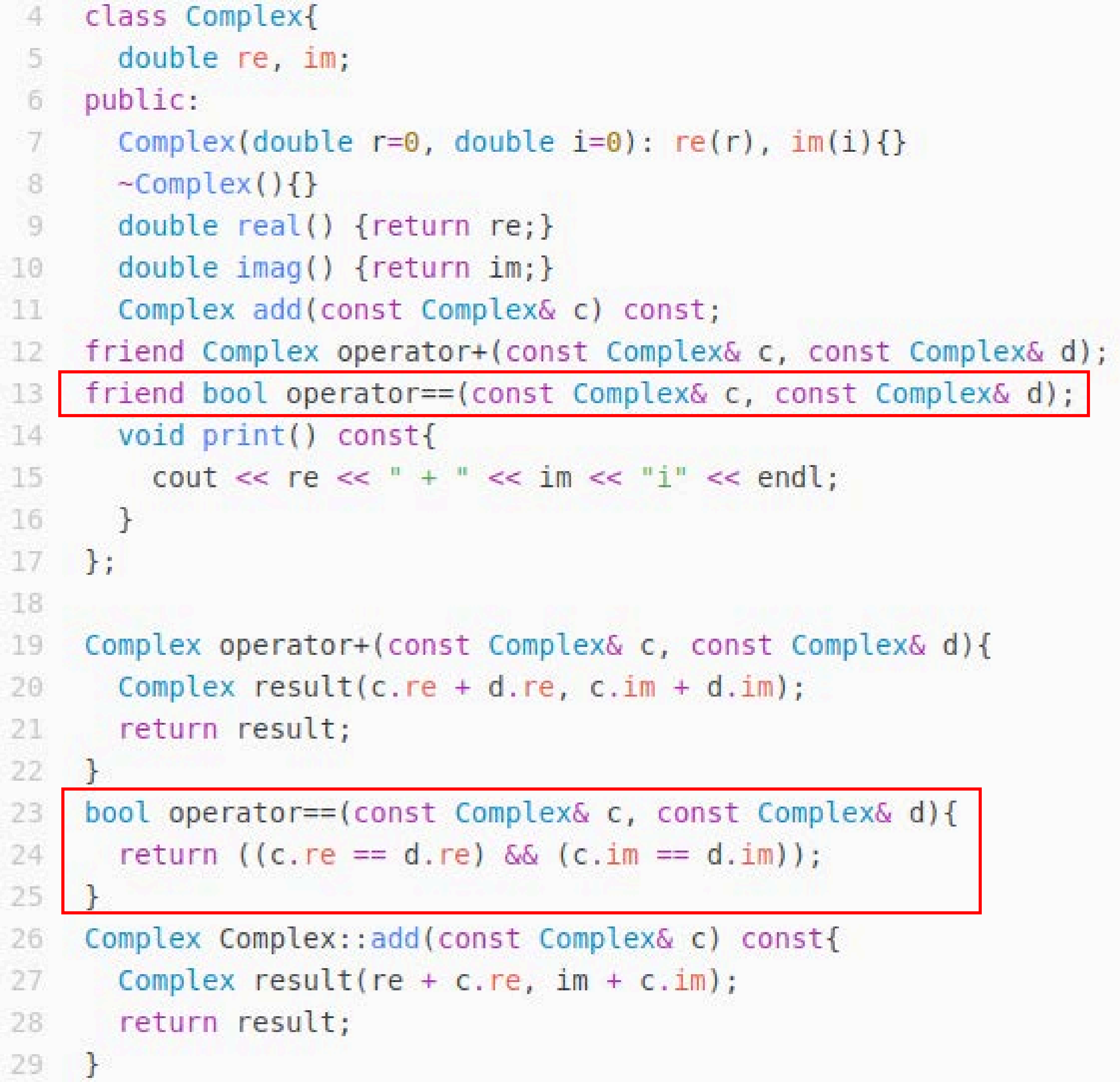
비멤버 함수로 구현 (friend 함수로 선언)

멤버 함수로 구현

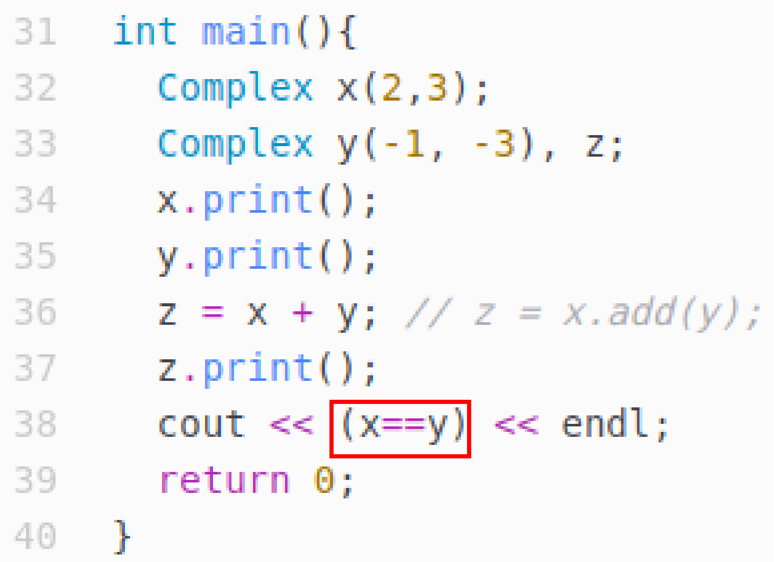
비멤버 함수로 연산자 함수 구현



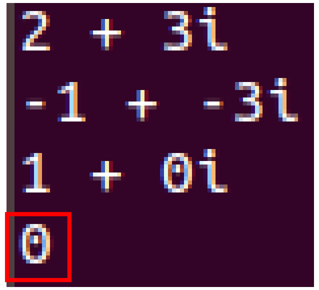
friend function operator==



```cpp
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex add(const Complex& c) const;
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
  void print() const{
    cout << re << " + " << im << "i" << endl;
  }
};

Complex operator+(const Complex& c, const Complex& d){
  Complex result(c.re + d.re, c.im + d.im);
  return result;
}
bool operator==(const Complex& c, const Complex& d){
  return ((c.re == d.re) && (c.im == d.im));
}
Complex Complex::add(const Complex& c) const{
  Complex result(re + c.re, im + c.im);
  return result;
}
```



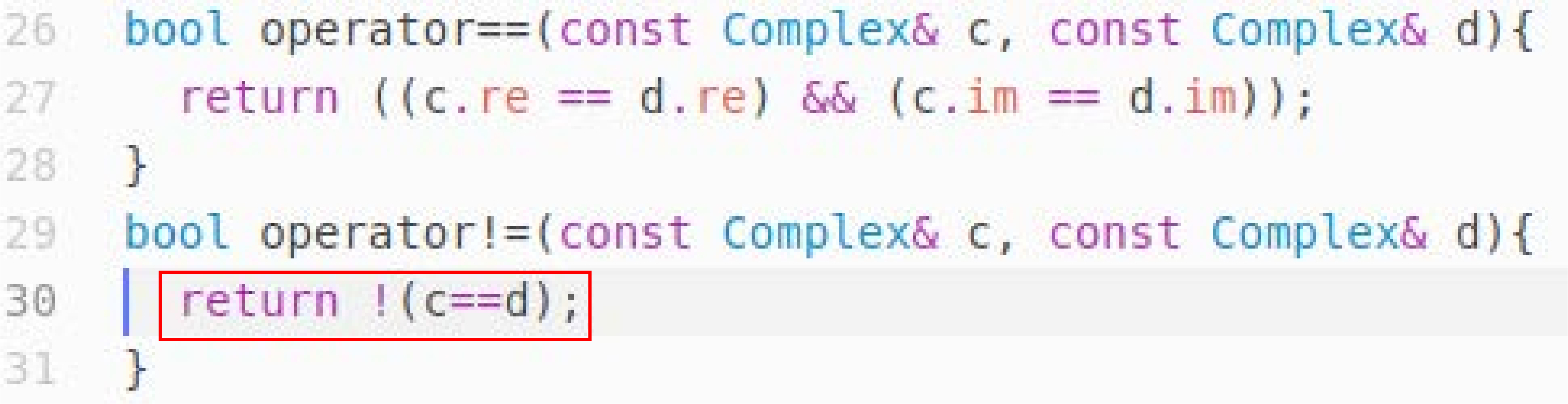
```cpp
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  x.print();
  y.print();
  z = x + y; // z = x.add(y);
  z.print();
  cout << (x==y) << endl;
  return 0;
}
```



```
2 + 3i
-1 + -3i
1 + 0i
0
```

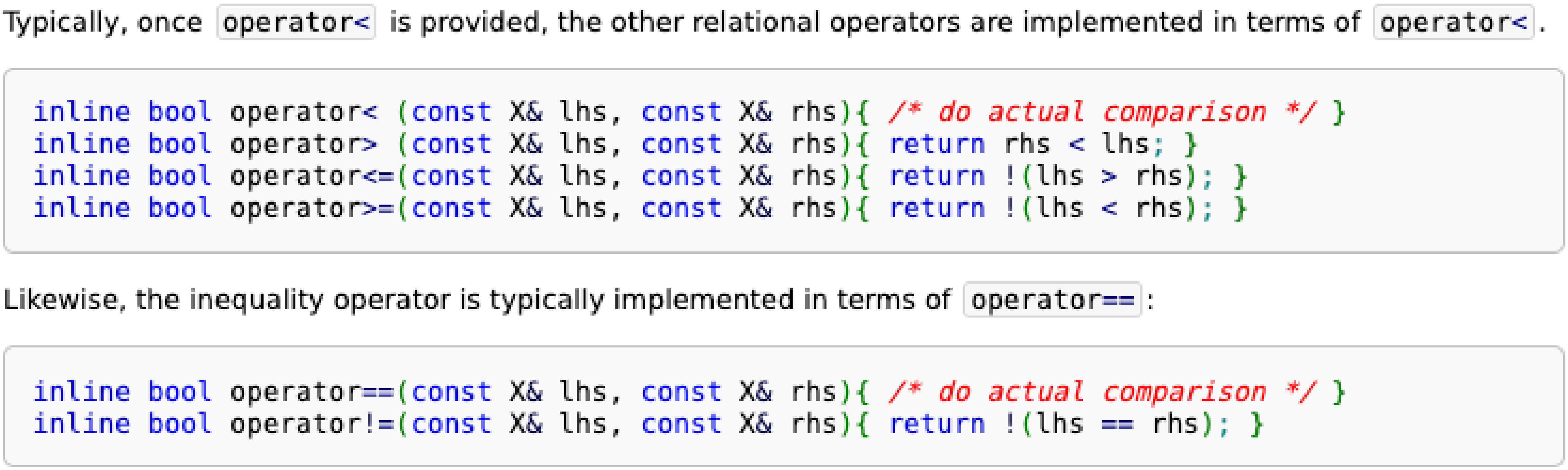
!= 연산자 중복

이 경우에 operator!= 는 friend 로 선언하지 않아도 된다.



```
bool operator==(const Complex& c, const Complex& d){
  return ((c.re == d.re) && (c.im == d.im));
}
bool operator!=(const Complex& c, const Complex& d){
  return !(c==d);
}
```

typical relational operators



Typically, once `operator<` is provided, the other relational operators are implemented in terms of `operator<`.

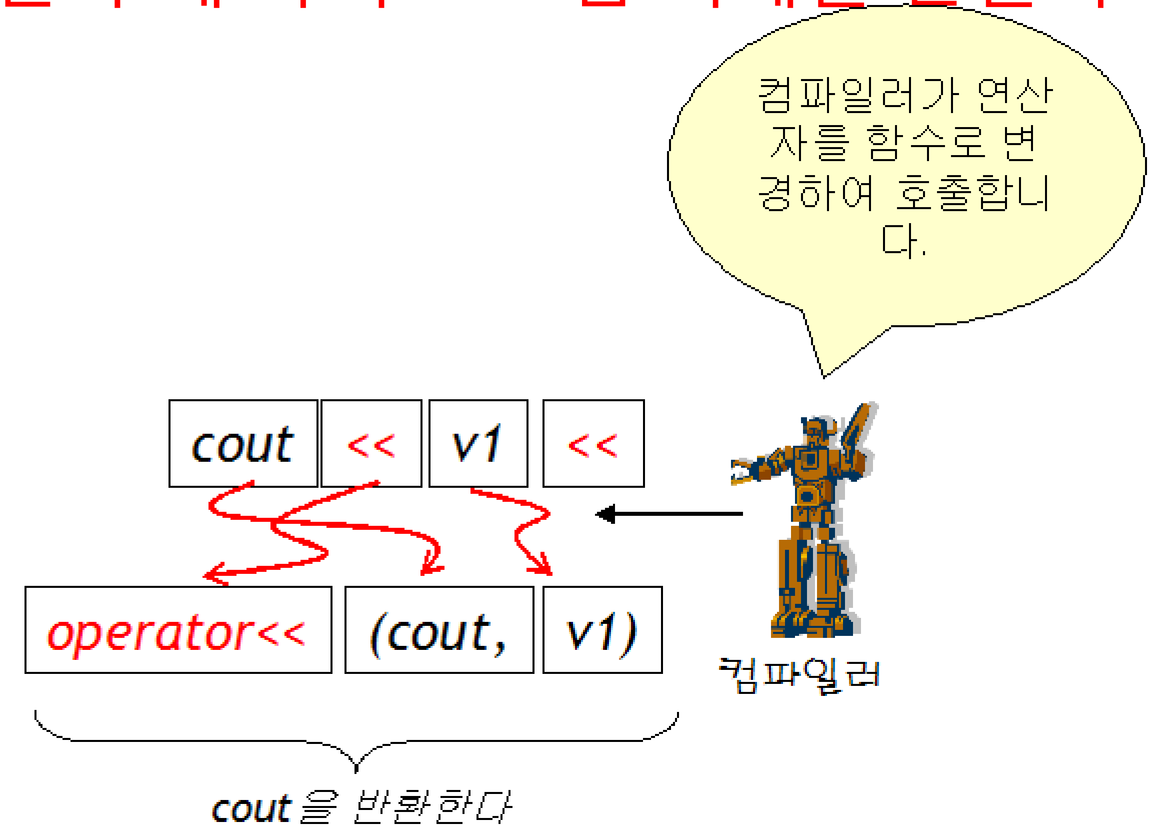
```
inline bool operator< (const X& lhs, const X& rhs){ /* do actual comparison */ }
inline bool operator> (const X& lhs, const X& rhs){ return rhs < lhs; }
inline bool operator<=(const X& lhs, const X& rhs){ return !(lhs > rhs); }
inline bool operator>=(const X& lhs, const X& rhs){ return !(lhs < rhs); }
```

Likewise, the inequality operator is typically implemented in terms of `operator==`:

```
inline bool operator==(const X& lhs, const X& rhs){ /* do actual comparison */ }
inline bool operator!=(const X& lhs, const X& rhs){ return !(lhs == rhs); }
```

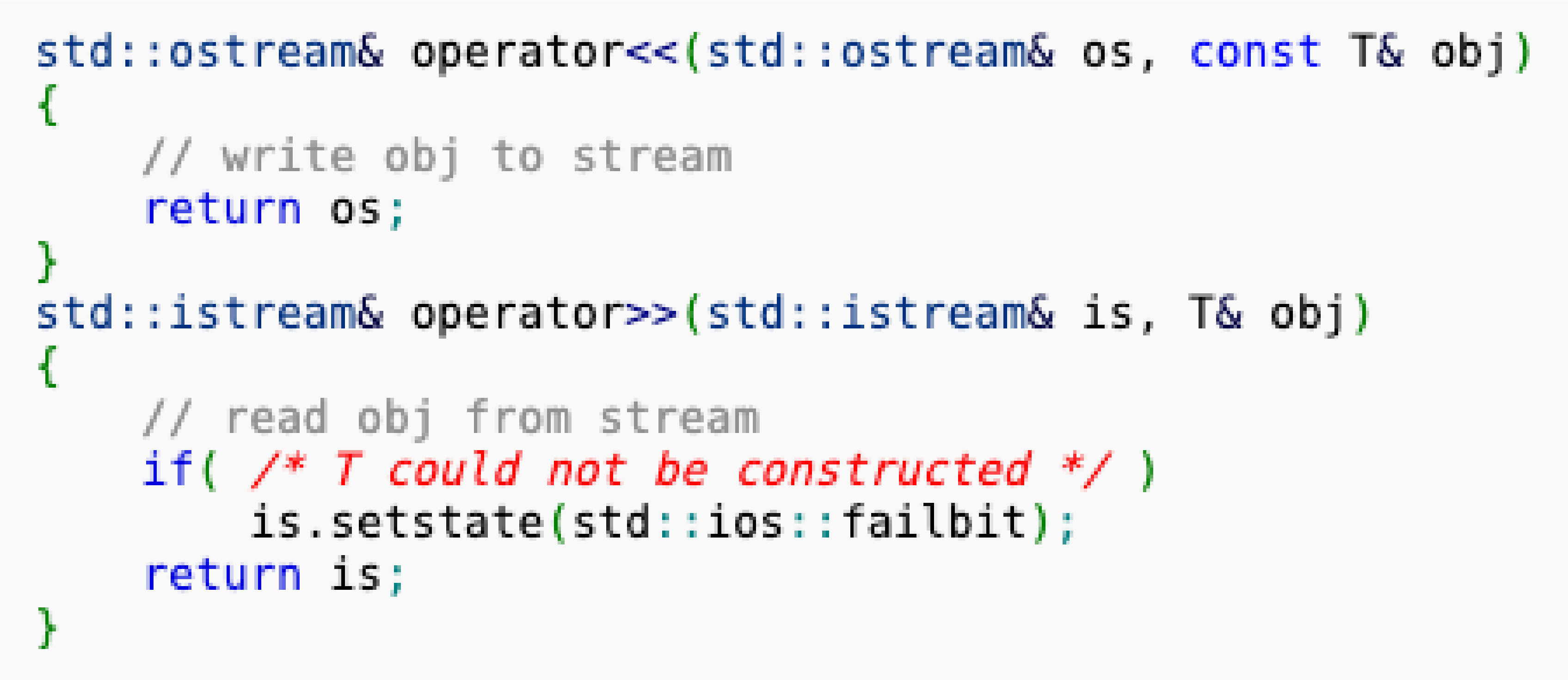
<<과 >> 연산자 중복

- 연산을 수행한 후에 다시 스트림 객체를 반환하여야 함



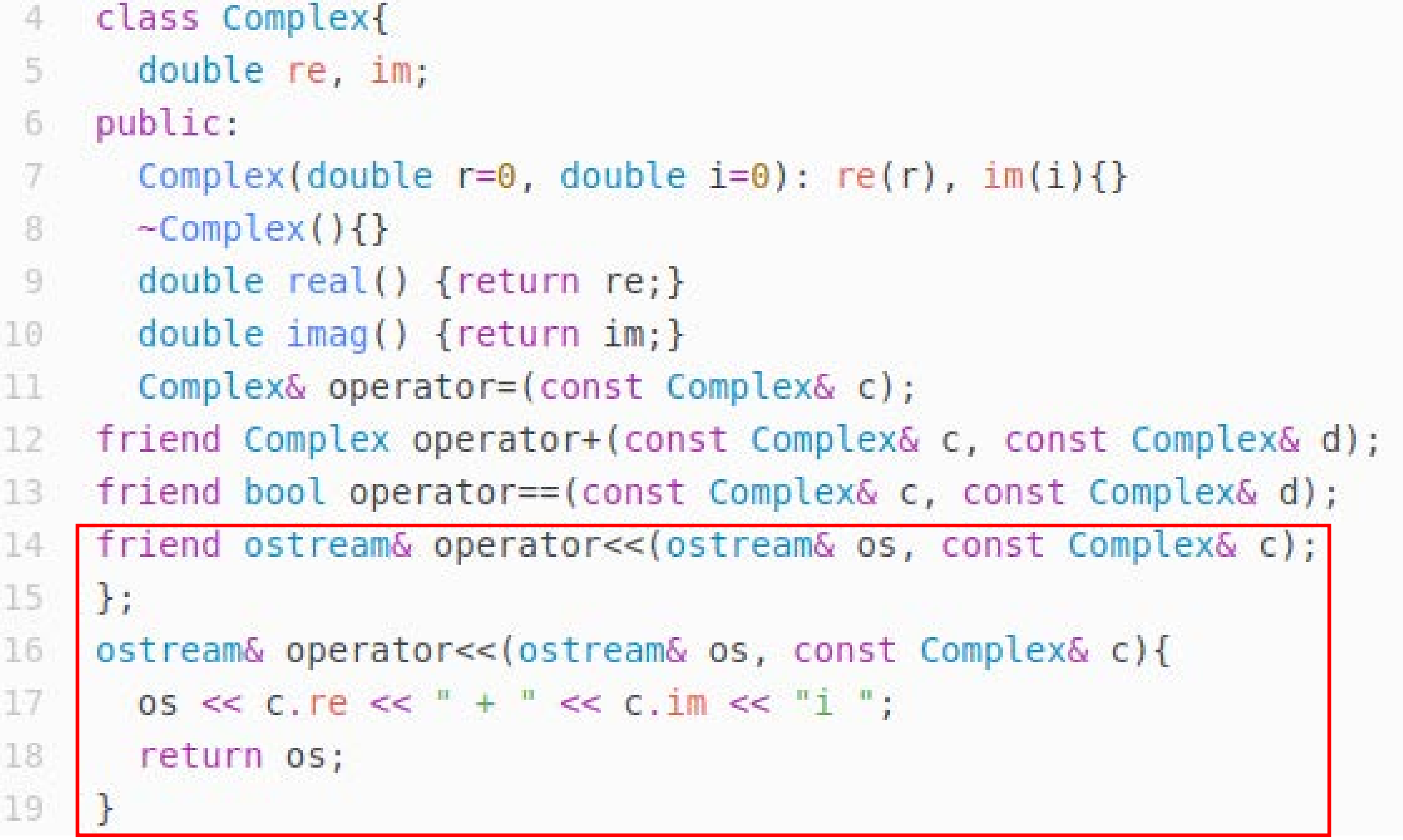
cout : standard output stream 을 나타내는 ostream class 의 객체, stdout in C 에 해당

typical << and >> operators

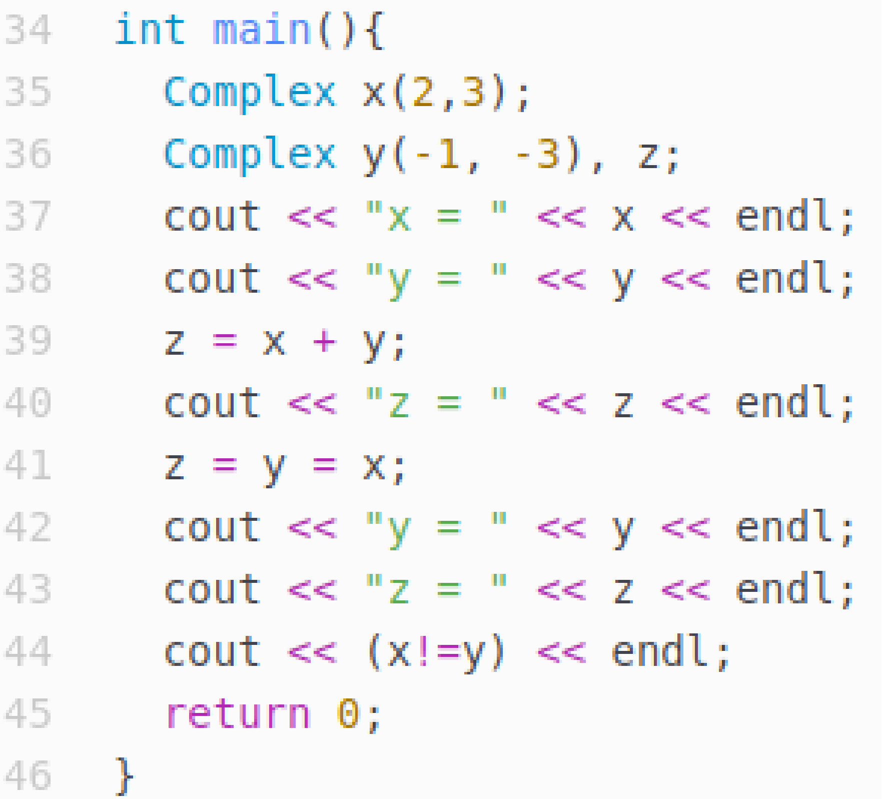


```
std::ostream& operator<<(std::ostream& os, const T& obj)
{
    // write obj to stream
    return os;
}
std::istream& operator>>(std::istream& is, T& obj)
{
    // read obj from stream
    if( /* T could not be constructed */ )
        is.setstate(std::ios::failbit);
    return is;
}
```

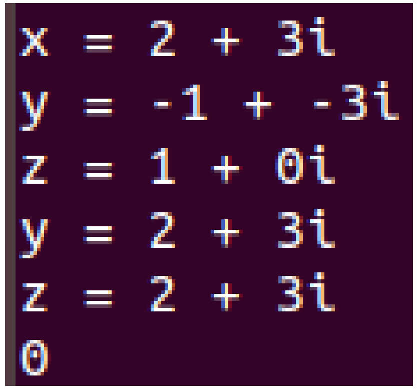
<< operator



```
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex& operator=(const Complex& c);
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
friend ostream& operator<<(ostream& os, const Complex& c);
};
ostream& operator<<(ostream& os, const Complex& c){
  os << c.re << " + " << c.im << "i ";
  return os;
}
```

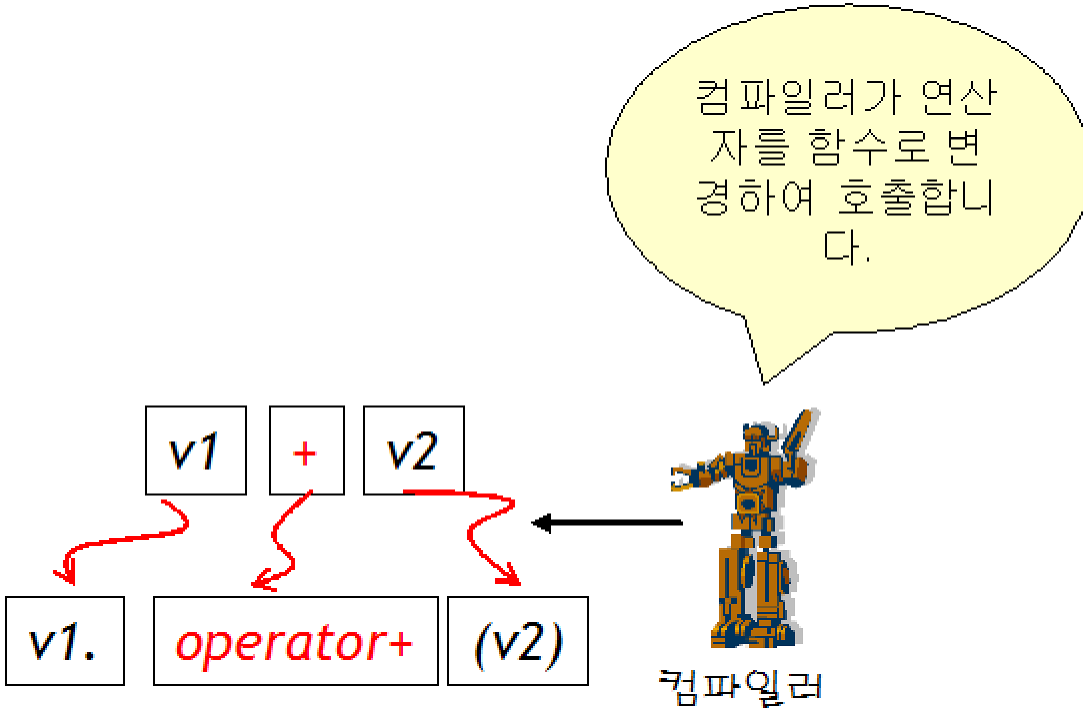


```
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = x + y;
  cout << "z = " << z << endl;
  z = y = x;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;
  cout << (x!=y) << endl;
  return 0;
}
```



```
x = 2 + 3i
y = -1 + -3i
z = 1 + 0i
y = 2 + 3i
z = 2 + 3i
0
```

멤버 함수로 연산자 함수 구현



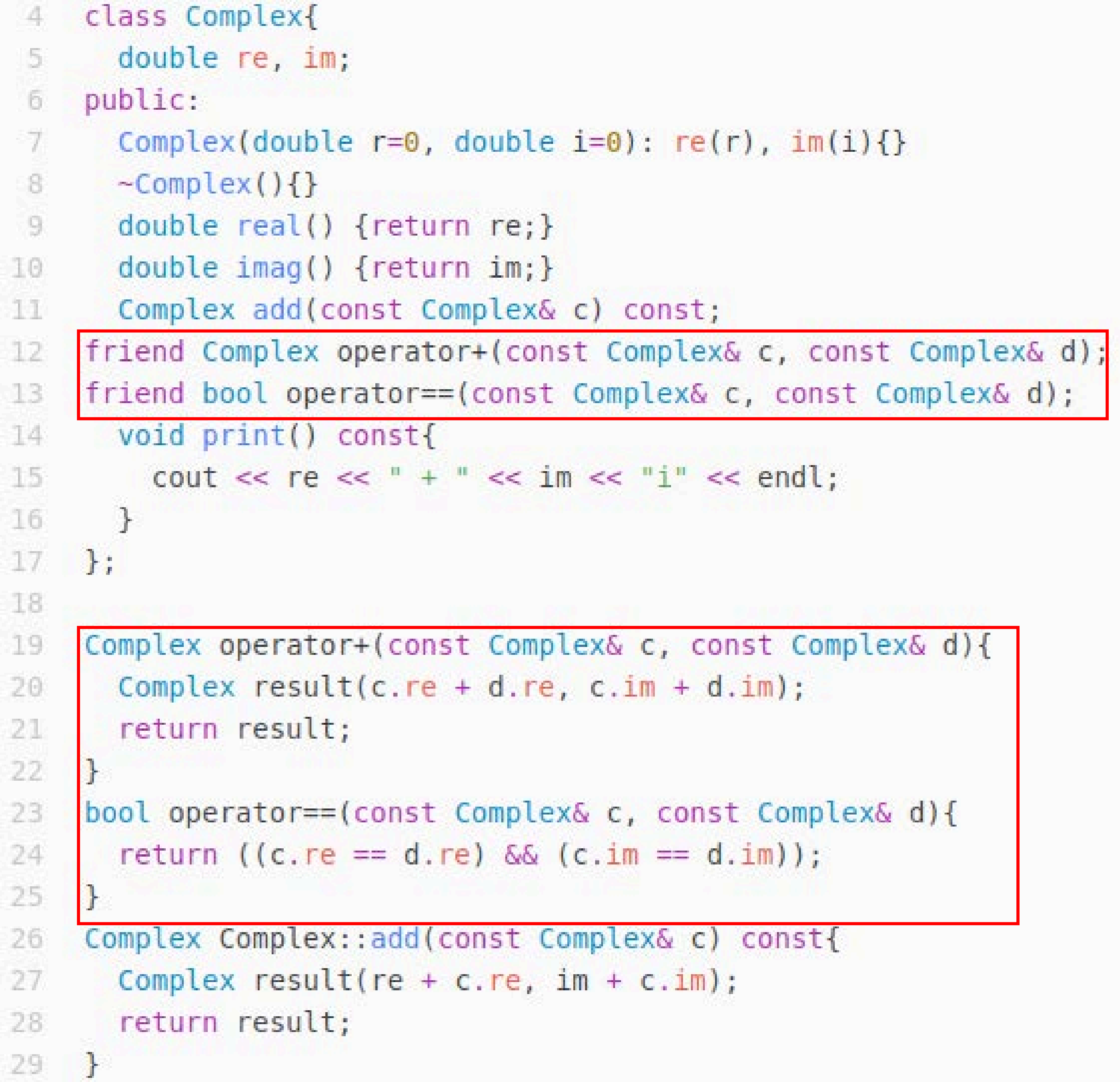
컴파일러가 연산자를 함수로 변경하여 호출합니다.

v1 + v2

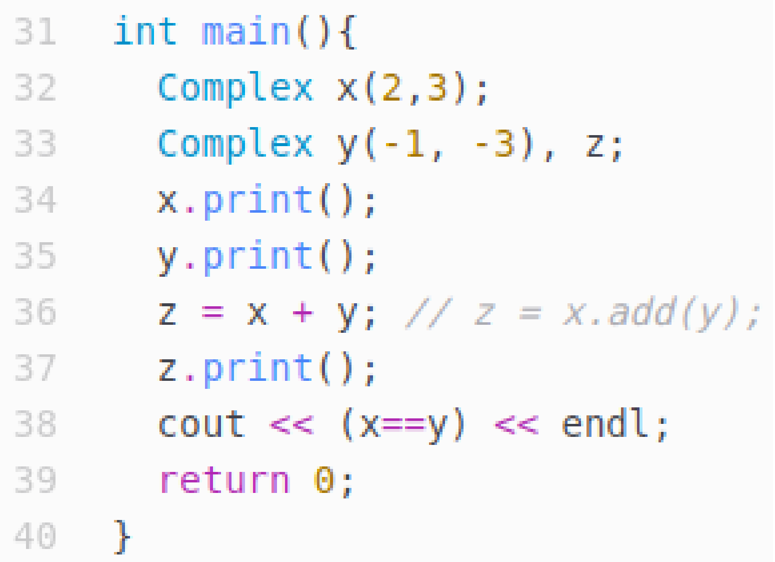
v1. operator+ (v2)

컴파일러

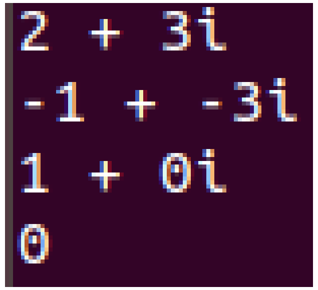
operators + and == as friend functions



```cpp
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex add(const Complex& c) const;
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
  void print() const{
    cout << re << " + " << im << "i" << endl;
  }
};

Complex operator+(const Complex& c, const Complex& d){
  Complex result(c.re + d.re, c.im + d.im);
  return result;
}
bool operator==(const Complex& c, const Complex& d){
  return ((c.re == d.re) && (c.im == d.im));
}
Complex Complex::add(const Complex& c) const{
  Complex result(re + c.re, im + c.im);
  return result;
}
```

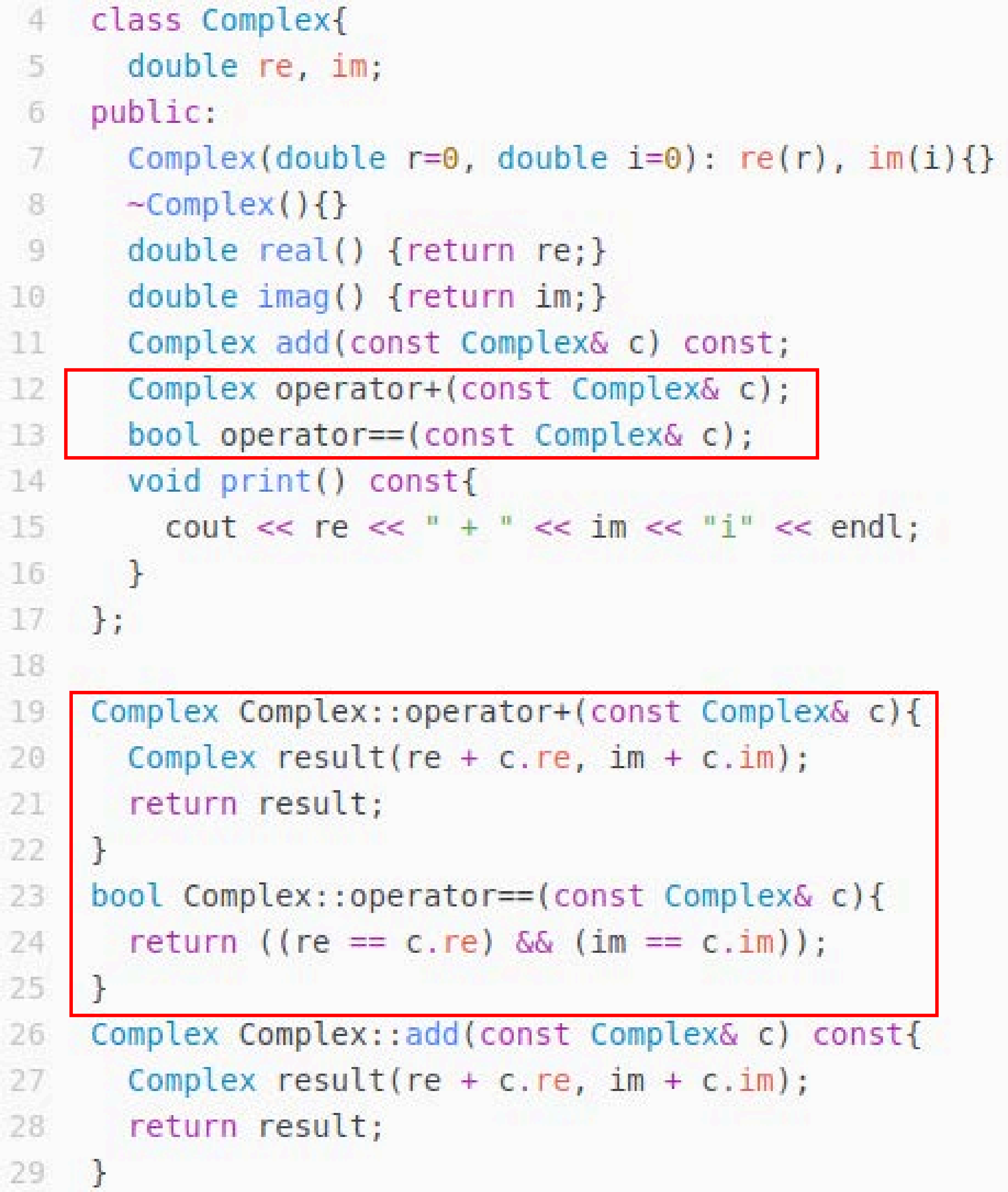


```cpp
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  x.print();
  y.print();
  z = x + y; // z = x.add(y);
  z.print();
  cout << (x==y) << endl;
  return 0;
}
```

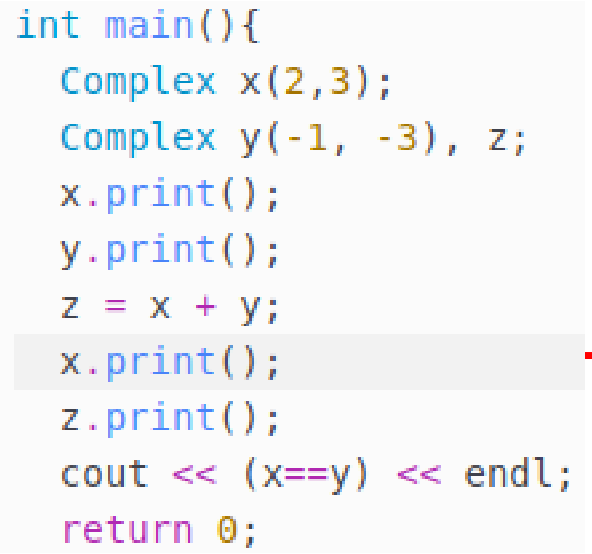


```
2 + 3i
-1 + -3i
1 + 0i
0
```

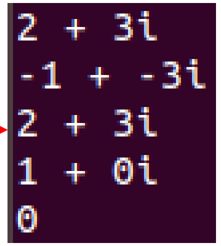
member function operators + and ==



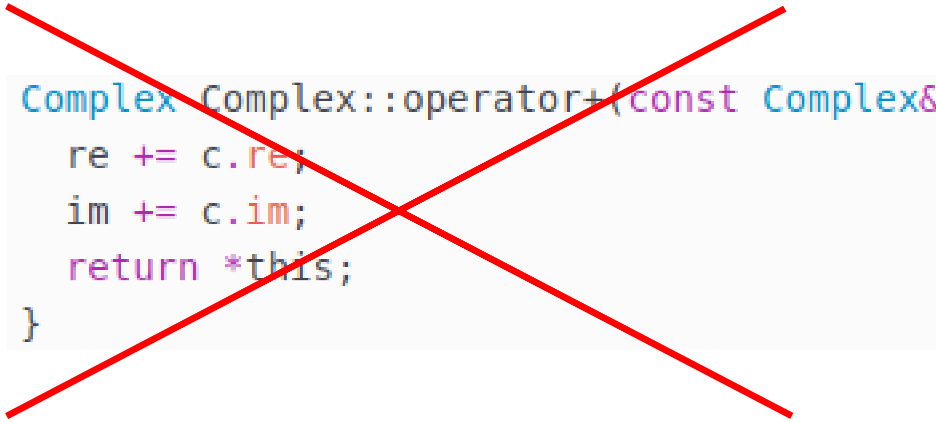
```cpp
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex add(const Complex& c) const;
  Complex operator+(const Complex& c);
  bool operator==(const Complex& c);
  void print() const{
    cout << re << " + " << im << "i" << endl;
  }
};

Complex Complex::operator+(const Complex& c){
  Complex result(re + c.re, im + c.im);
  return result;
}
bool Complex::operator==(const Complex& c){
  return ((re == c.re) && (im == c.im));
}
Complex Complex::add(const Complex& c) const{
  Complex result(re + c.re, im + c.im);
  return result;
}
```



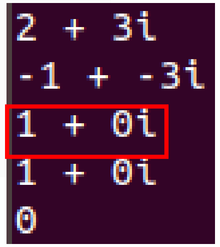
```cpp
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  x.print();
  y.print();
  z = x + y;
  x.print();
  z.print();
  cout << (x==y) << endl;
  return 0;
```



```
2 + 3i
-1 + -3i
2 + 3i
1 + 0i
0
```

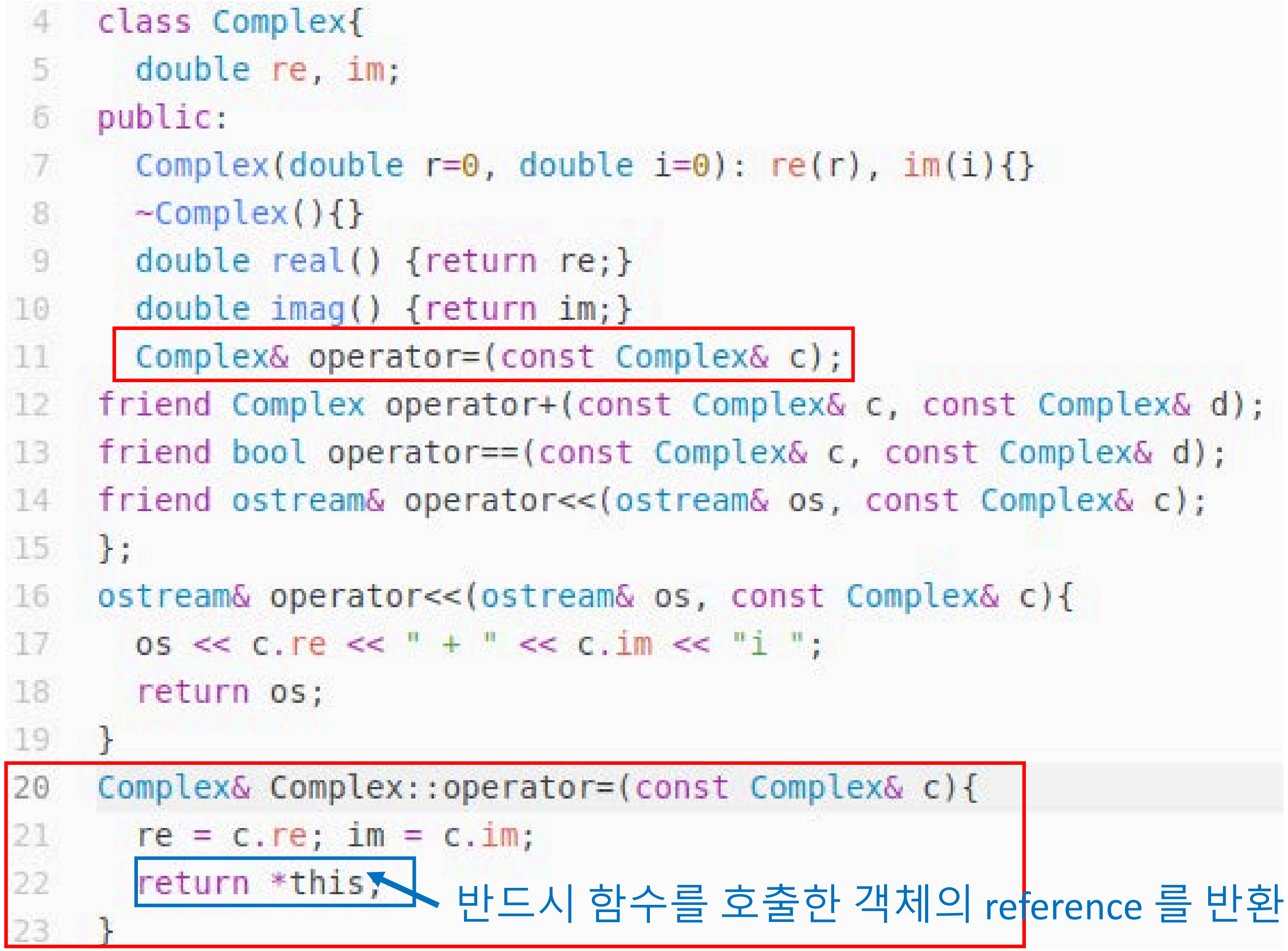


```cpp
Complex Complex::operator+(const Complex& c){
  re += c.re;
  im += c.im;
  return *this;
}
```



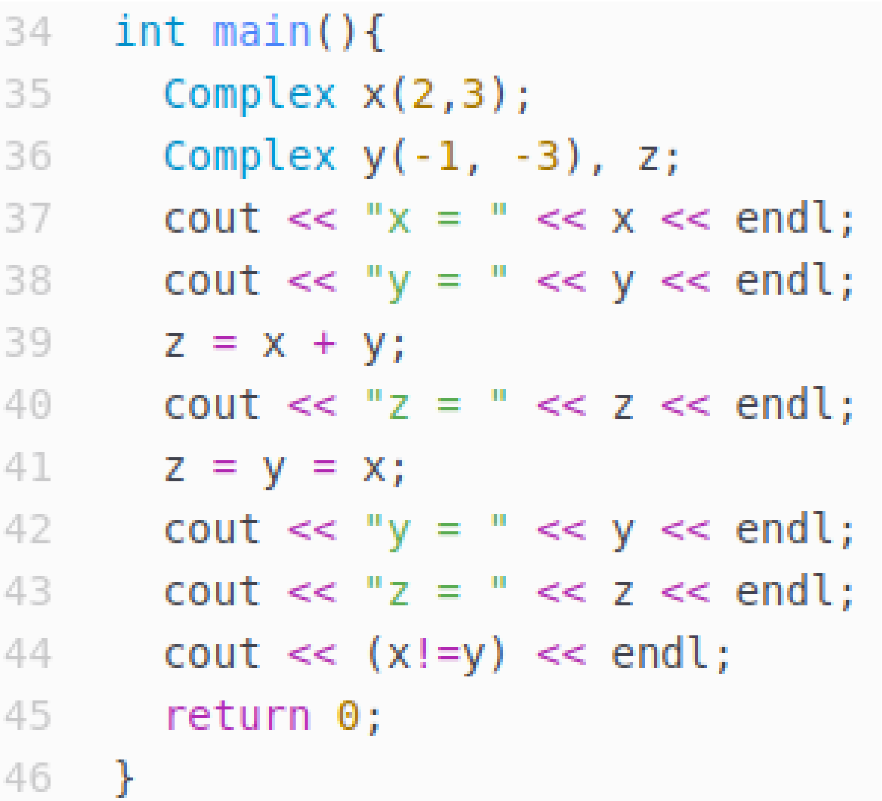
```
2 + 3i
-1 + -3i
1 + 0i
1 + 0i
0
```

= (치환) 연산자의 중복

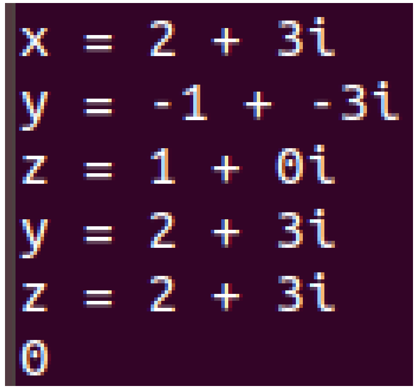


```cpp
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex& operator=(const Complex& c);
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
friend ostream& operator<<(ostream& os, const Complex& c);
};
ostream& operator<<(ostream& os, const Complex& c){
  os << c.re << " + " << c.im << "i ";
  return os;
}
Complex& Complex::operator=(const Complex& c){
  re = c.re; im = c.im;
  return *this;
}
```

반드시 함수를 호출한 객체의 reference 를 반환해야 함



```cpp
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = x + y;
  cout << "z = " << z << endl;
  z = y = x;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;
  cout << (x!=y) << endl;
  return 0;
}
```

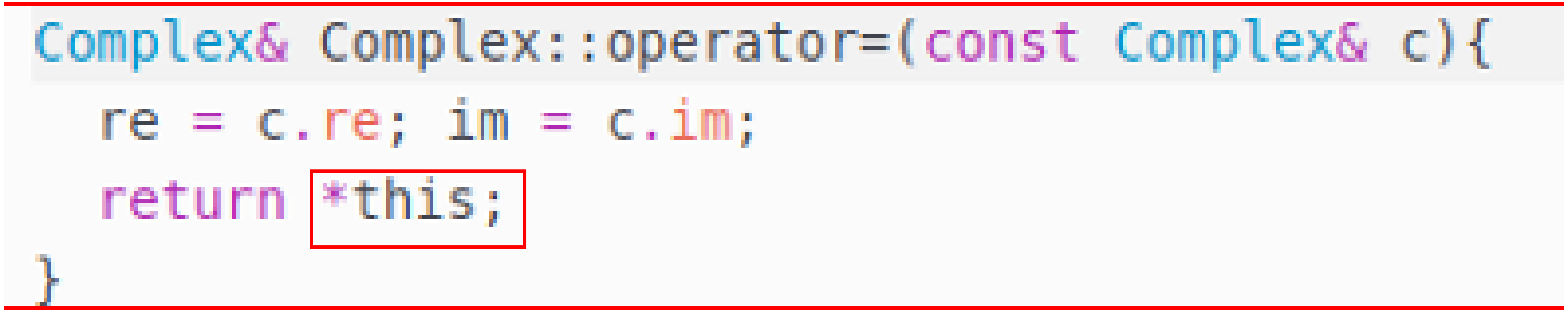


```
x = 2 + 3i
y = -1 + -3i
z = 1 + 0i
y = 2 + 3i
z = 2 + 3i
0
```

= 연산자 중복

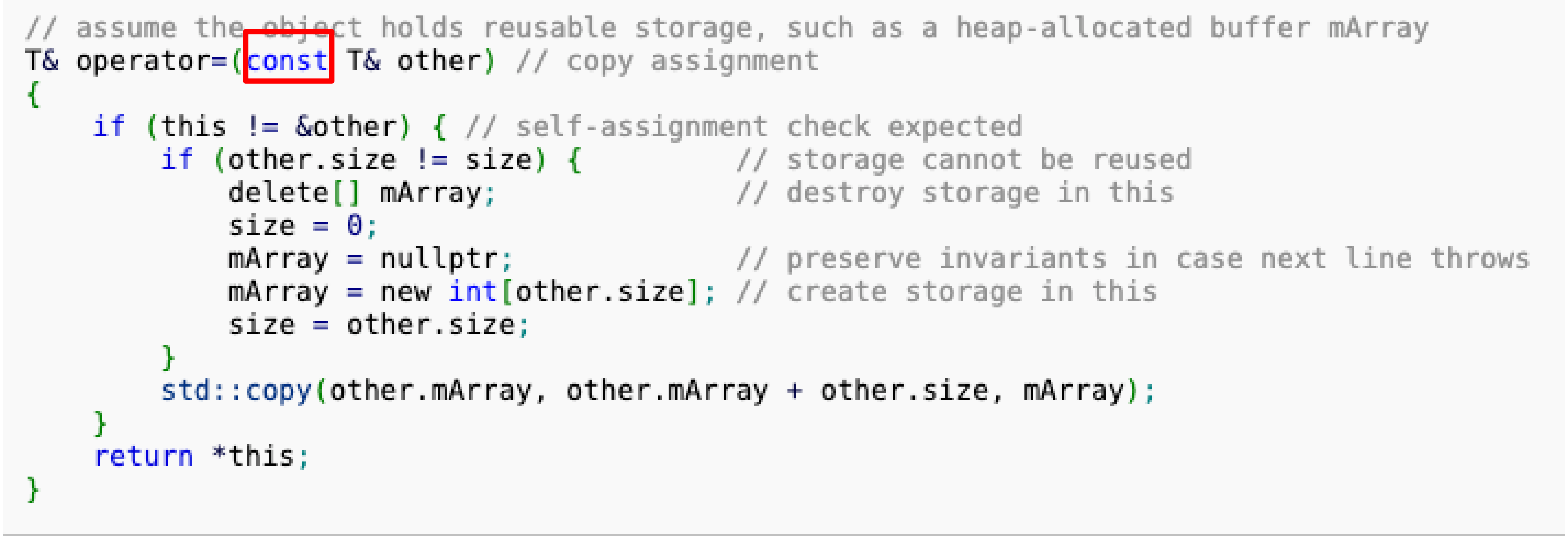
=의 연산순서는 ←

반드시 현재 객체의 reference 를 반환해야 c3 = c2 = c1; 형태의 수식이 가능.



```
Complex& Complex::operator=(const Complex& c){
   re = c.re; im = c.im;
   return *this;
}
```

typical ‘=‘ operator

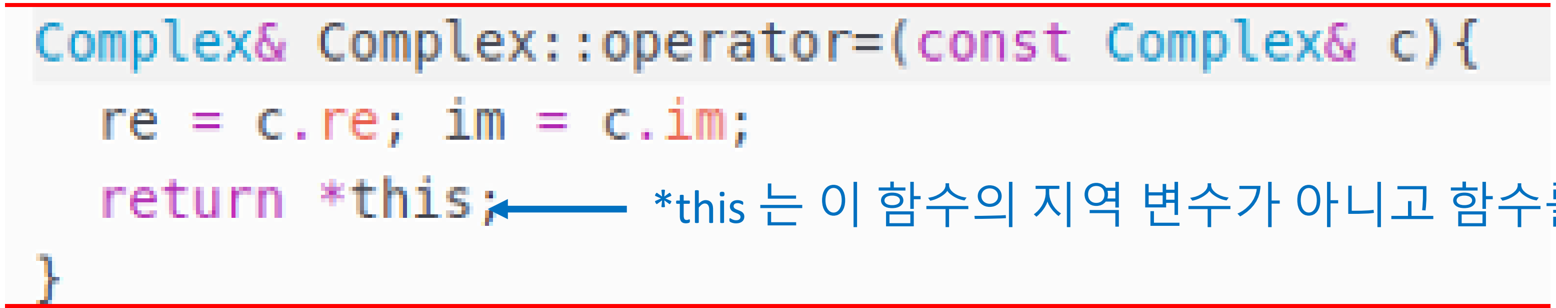


```
// assume the object holds reusable storage, such as a heap-allocated buffer mArray
T& operator=(const T& other) // copy assignment
{
    if (this != &other) { // self-assignment check expected
        if (other.size != size) {         // storage cannot be reused
            delete[] mArray;              // destroy storage in this
            size = 0;
            mArray = nullptr;             // preserve invariants in case next line throws
            mArray = new int[other.size]; // create storage in this
            size = other.size;
        }
        std::copy(other.mArray, other.mArray + other.size, mArray);
    }
    return *this;
}
```

https://en.cppreference.com/w/cpp/language/operators

reference 를 반환형으로 하는 경우 주의!

```
Compex& Complex::add(& b){
    Complex r;
    r.re = re + b.re;
    r.im = im + b.im;
    return r;          지역 변수의 레퍼런스를 반환
}
int main(){
  Complex a(1,2), b(3,4), c(5,6);
  Complex e(0,0);
  e = a.add(b);
  e = (a.add(b)).add(c) ;
}   뭐가 문제일까요???
```



```
Complex& Complex::operator=(const Complex& c){
    re = c.re; im = c.im;
    return *this;
}
```

*this 는 이 함수의 지역 변수가 아니고 함수를 호출한 객체이기 때문에 가능

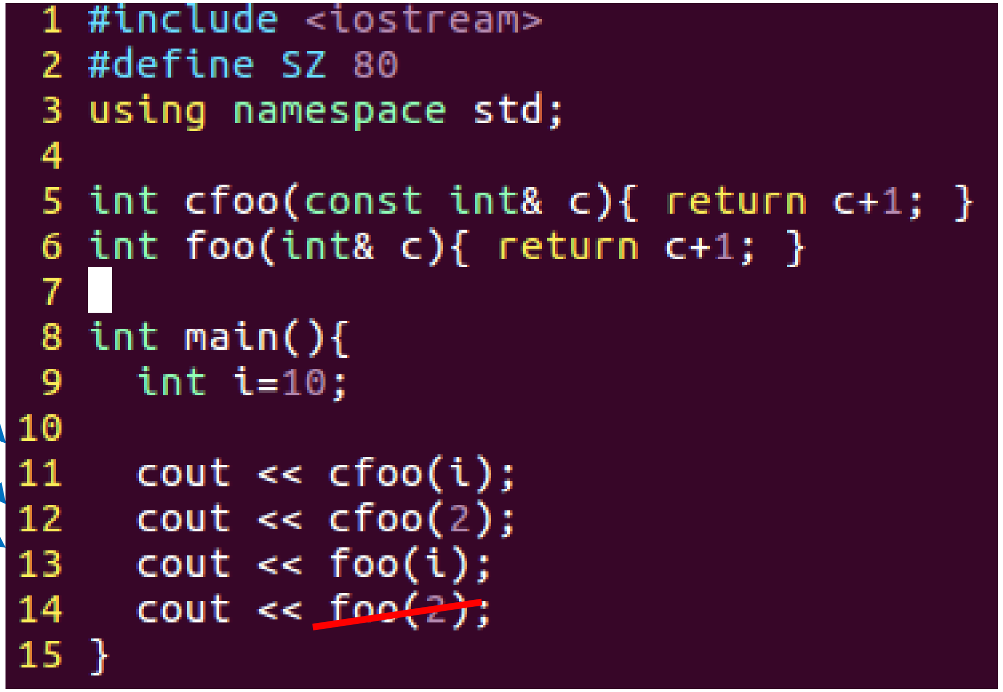
passing a “const” parameter in reference/pointer

- If a function has a non-const parameter, it cannot be passed a const argument while making a call.
- If a function has a const type parameter, it can be passed a const type argument as well as a non-const argument.
- Temporary objects created while program execution are always of const type.

const RHS ← const/non-const LHS

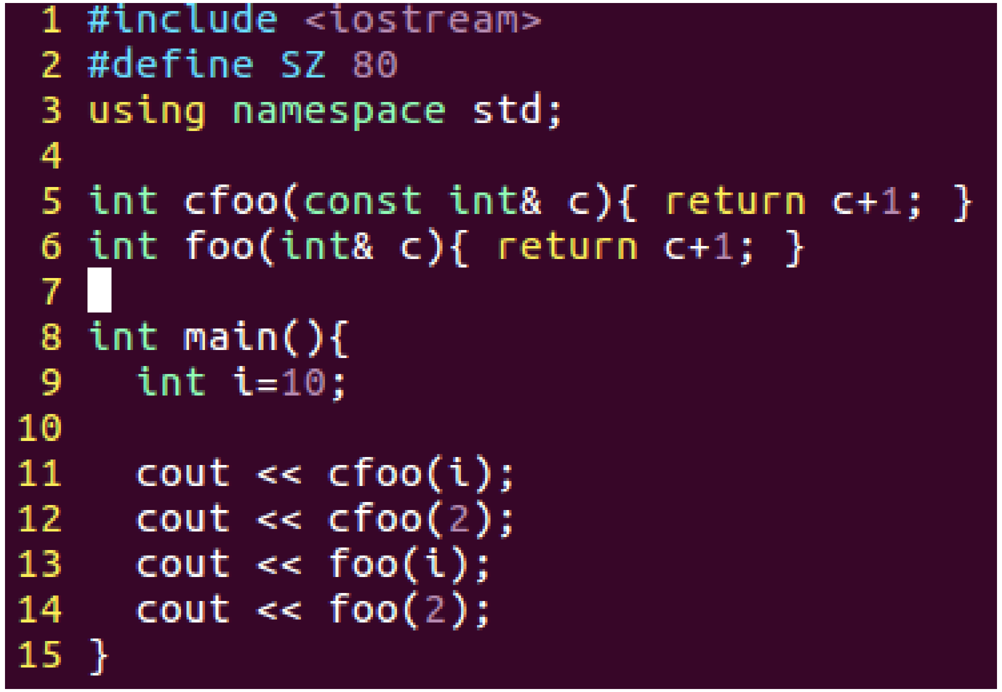
non-const RHS ← non-const LHS

~~non-const RHS ← const LHS~~

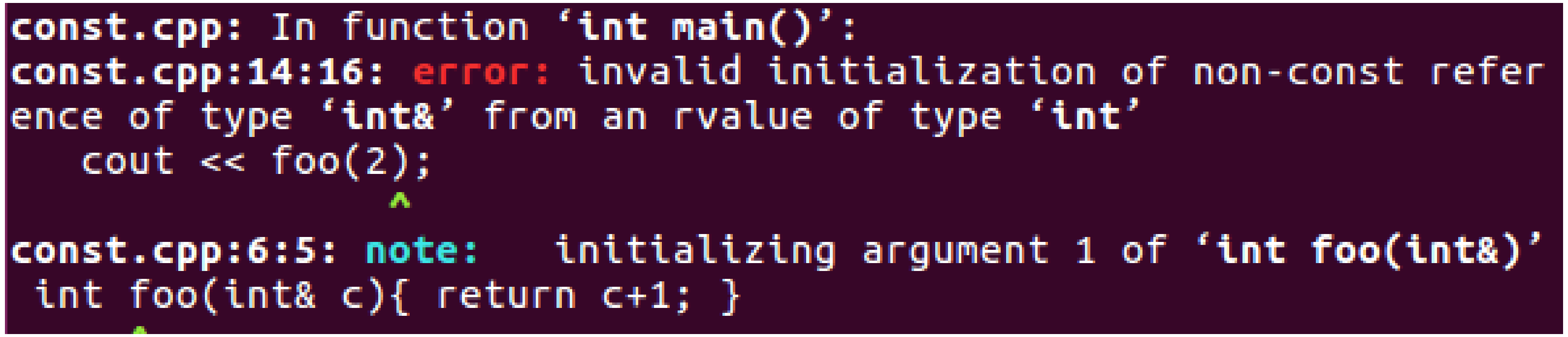


```
#include <iostream>
#define SZ 80
using namespace std;

int cfoo(const int& c){ return c+1; }
int foo(int& c){ return c+1; }

int main(){
  int i=10;

  cout << cfoo(i);
  cout << cfoo(2);
  cout << foo(i);
  cout << ~~foo(2)~~;
}
```

compile error (reference)



```
 1 #include <iostream>
 2 #define SZ 80
 3 using namespace std;
 4
 5 int cfoo(const int& c){ return c+1; }
 6 int foo(int& c){ return c+1; }
 7
 8 int main(){
 9   int i=10;
10
11   cout << cfoo(i);
12   cout << cfoo(2);
13   cout << foo(i);
14   cout << foo(2);
15 }
```



```
const.cpp: In function ‘int main()’:
const.cpp:14:16: error: invalid initialization of non-const refer
ence of type ‘int&’ from an rvalue of type ‘int’
   cout << foo(2);
               ^
const.cpp:6:5: note:   initializing argument 1 of ‘int foo(int&)’
 int foo(int& c){ return c+1; }
     ^
```

const 포인터

- const int *p1;
- p1은 const int에 대한 포인터이다. 즉 p1이 가리키는 내용이 상수가 된다.
- *p1 = 100;(X)

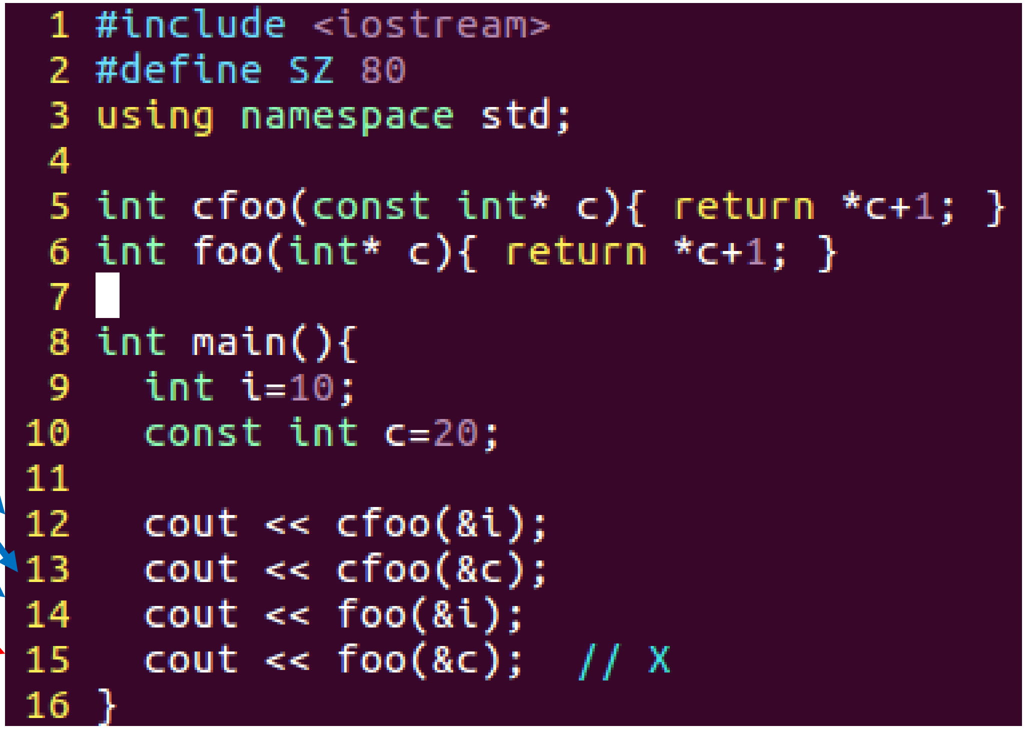
- int * const p2;
- 이번에는 정수를 가리키는 p2가 상수라는 의미이다. 즉 p2의 내용이 변경될 수 없다.
- p2 = p1; (X)

compile error (pointer)

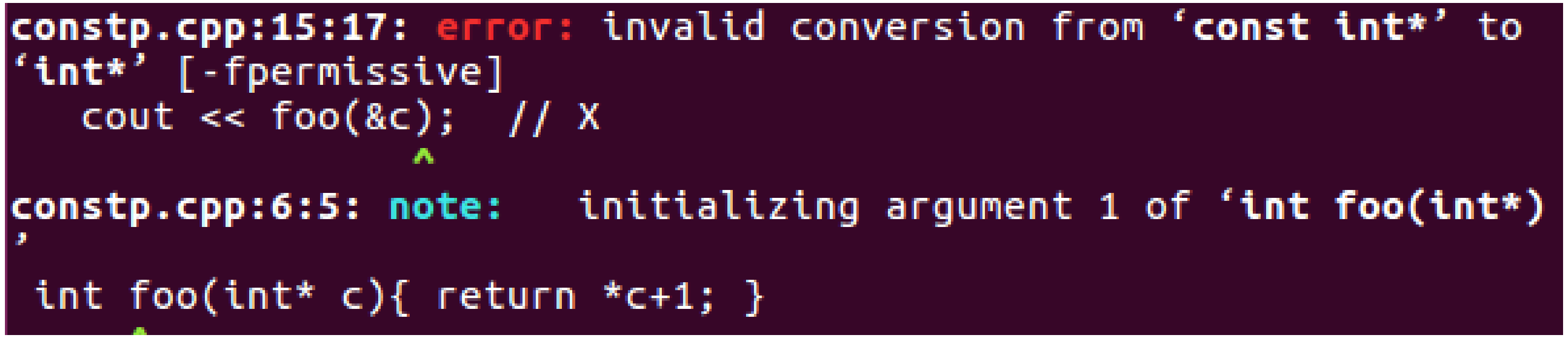
const RHS ← const/non-const LHS

non-const RHS ← non-const LHS

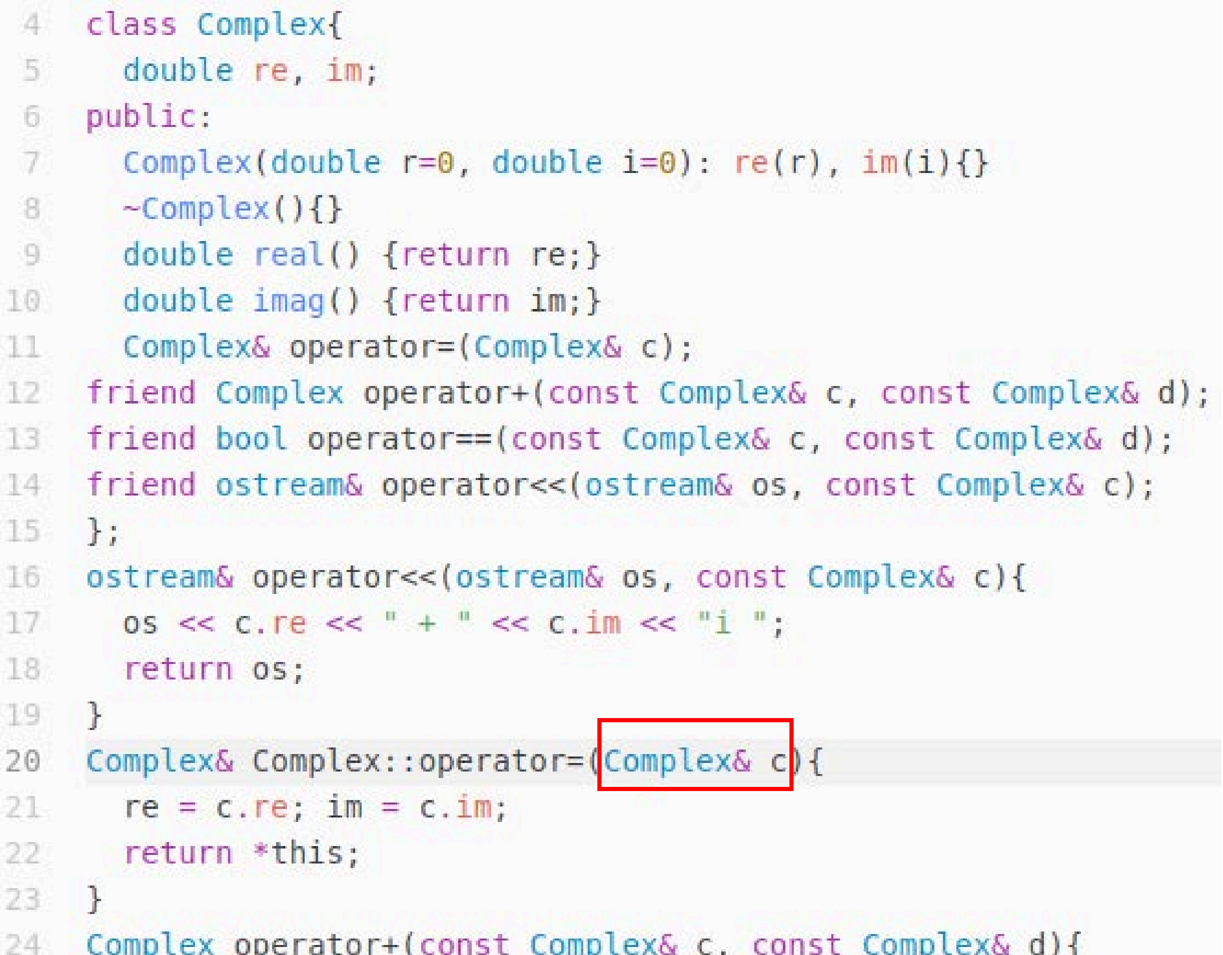
~~non-const RHS ← const LHS~~



```
 1 #include <iostream>
 2 #define SZ 80
 3 using namespace std;
 4
 5 int cfoo(const int* c){ return *c+1; }
 6 int foo(int* c){ return *c+1; }
 7
 8 int main(){
 9   int i=10;
10   const int c=20;
11
12   cout << cfoo(&i);
13   cout << cfoo(&c);
14   cout << foo(&i);
15   cout << foo(&c);   // X
16 }
```

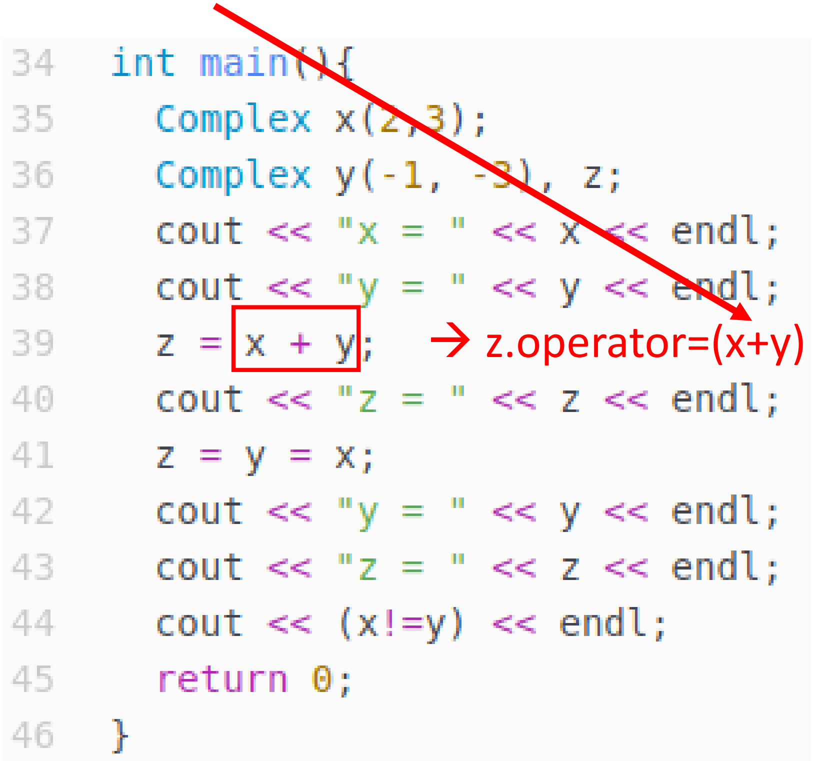


```
constp.cpp:15:17: error: invalid conversion from ‘const int*’ to
‘int*’ [-fpermissive]
   cout << foo(&c);   // X
                 ^
constp.cpp:6:5: note:   initializing argument 1 of ‘int foo(int*)
’
 int foo(int* c){ return *c+1; }
```



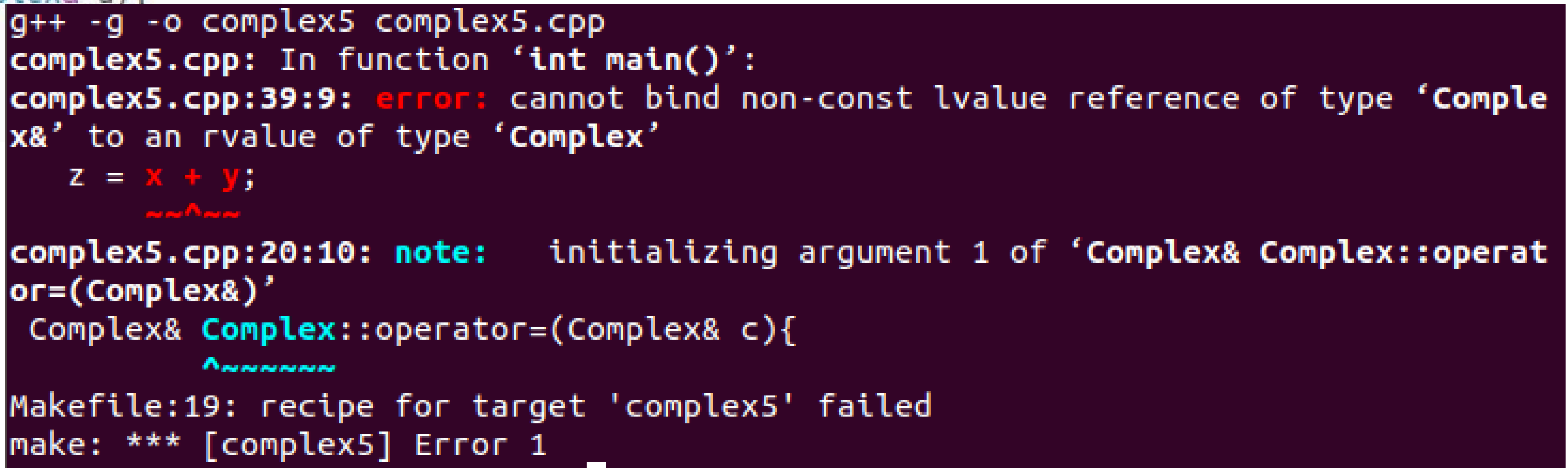
```cpp
 4  class Complex{
 5    double re, im;
 6  public:
 7    Complex(double r=0, double i=0): re(r), im(i){}
 8    ~Complex(){}
 9    double real() {return re;}
10    double imag() {return im;}
11    Complex& operator=(Complex& c);
12  friend Complex operator+(const Complex& c, const Complex& d);
13  friend bool operator==(const Complex& c, const Complex& d);
14  friend ostream& operator<<(ostream& os, const Complex& c);
15  };
16  ostream& operator<<(ostream& os, const Complex& c){
17    os << c.re << " + " << c.im << "i ";
18    return os;
19  }
20  Complex& Complex::operator=(Complex& c){
21    re = c.re; im = c.im;
22    return *this;
23  }
24  Complex operator+(const Complex& c, const Complex& d){
25    Complex result(c.re + d.re, c.im + d.im);
26    return result;
27  }
```

Temporary objects created while program execution are always of const type.

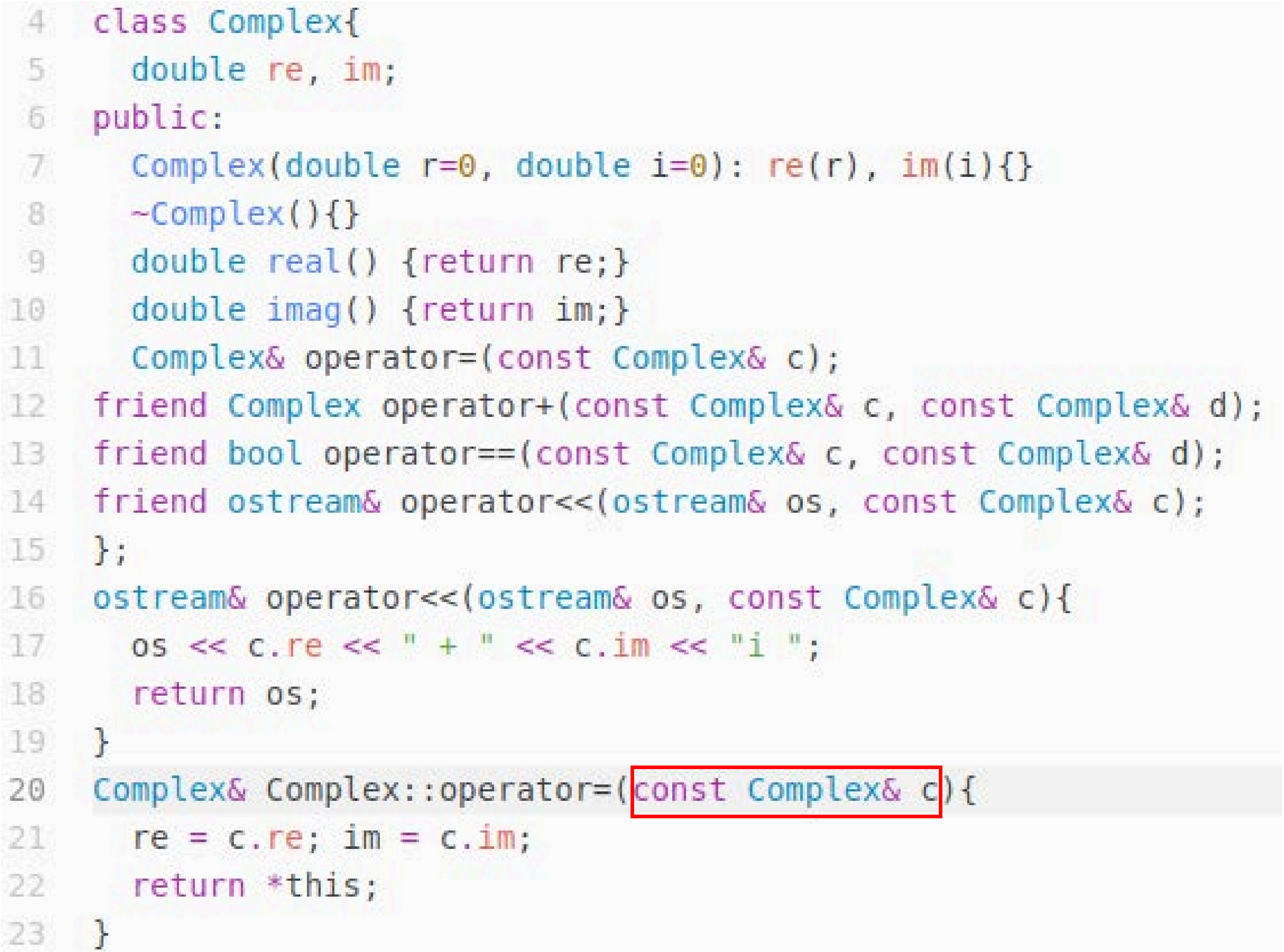


```cpp
34  int main(){
35    Complex x(2,3);
36    Complex y(-1, -3), z;
37    cout << "x = " << x << endl;
38    cout << "y = " << y << endl;
39    z = x + y;
40    cout << "z = " << z << endl;
41    z = y = x;
42    cout << "y = " << y << endl;
43    cout << "z = " << z << endl;
44    cout << (x!=y) << endl;
45    return 0;
46  }
```

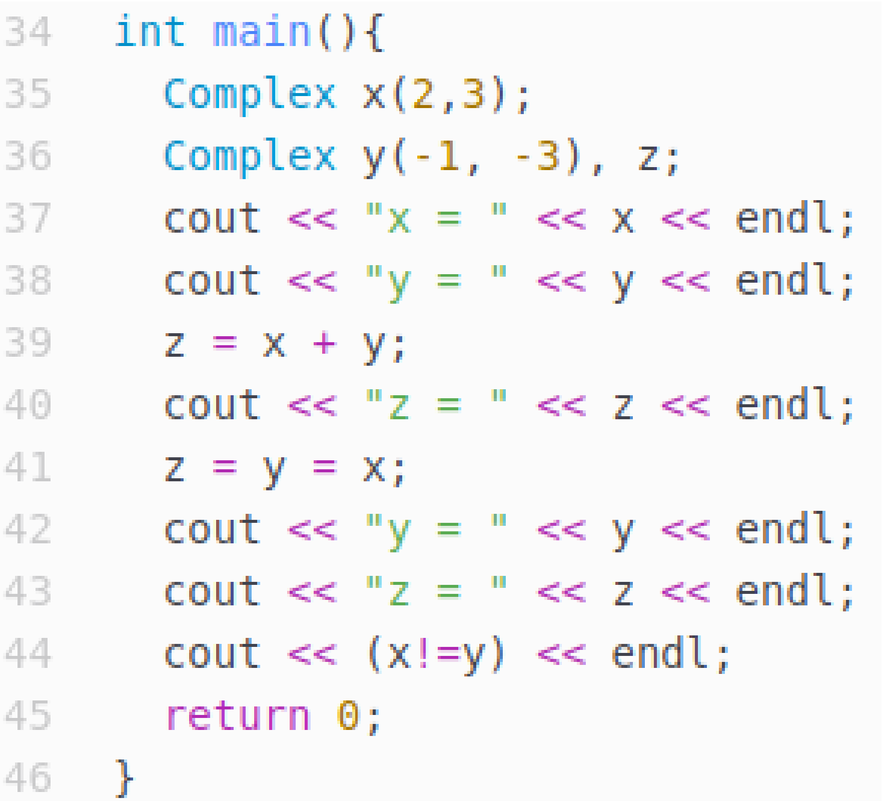
→ z.operator=(x+y)



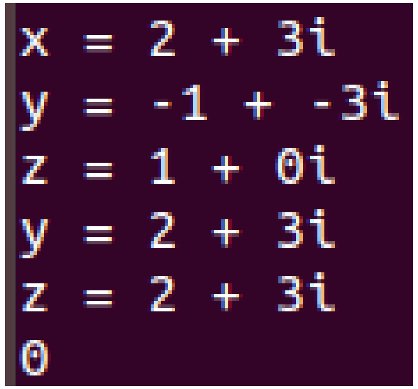
```
g++ -g -o complex5 complex5.cpp
complex5.cpp: In function ‘int main()’:
complex5.cpp:39:9: error: cannot bind non-const lvalue reference of type ‘Comple
x&’ to an rvalue of type ‘Complex’
   z = x + y;
       ~~^~~
complex5.cpp:20:10: note:   initializing argument 1 of ‘Complex& Complex::operat
or=(Complex&)’
 Complex& Complex::operator=(Complex& c){
          ^~~~~~~
Makefile:19: recipe for target 'complex5' failed
make: *** [complex5] Error 1
```



```
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex& operator=(const Complex& c);
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
friend ostream& operator<<(ostream& os, const Complex& c);
};
ostream& operator<<(ostream& os, const Complex& c){
  os << c.re << " + " << c.im << "i ";
  return os;
}
Complex& Complex::operator=(const Complex& c){
  re = c.re; im = c.im;
  return *this;
}
```

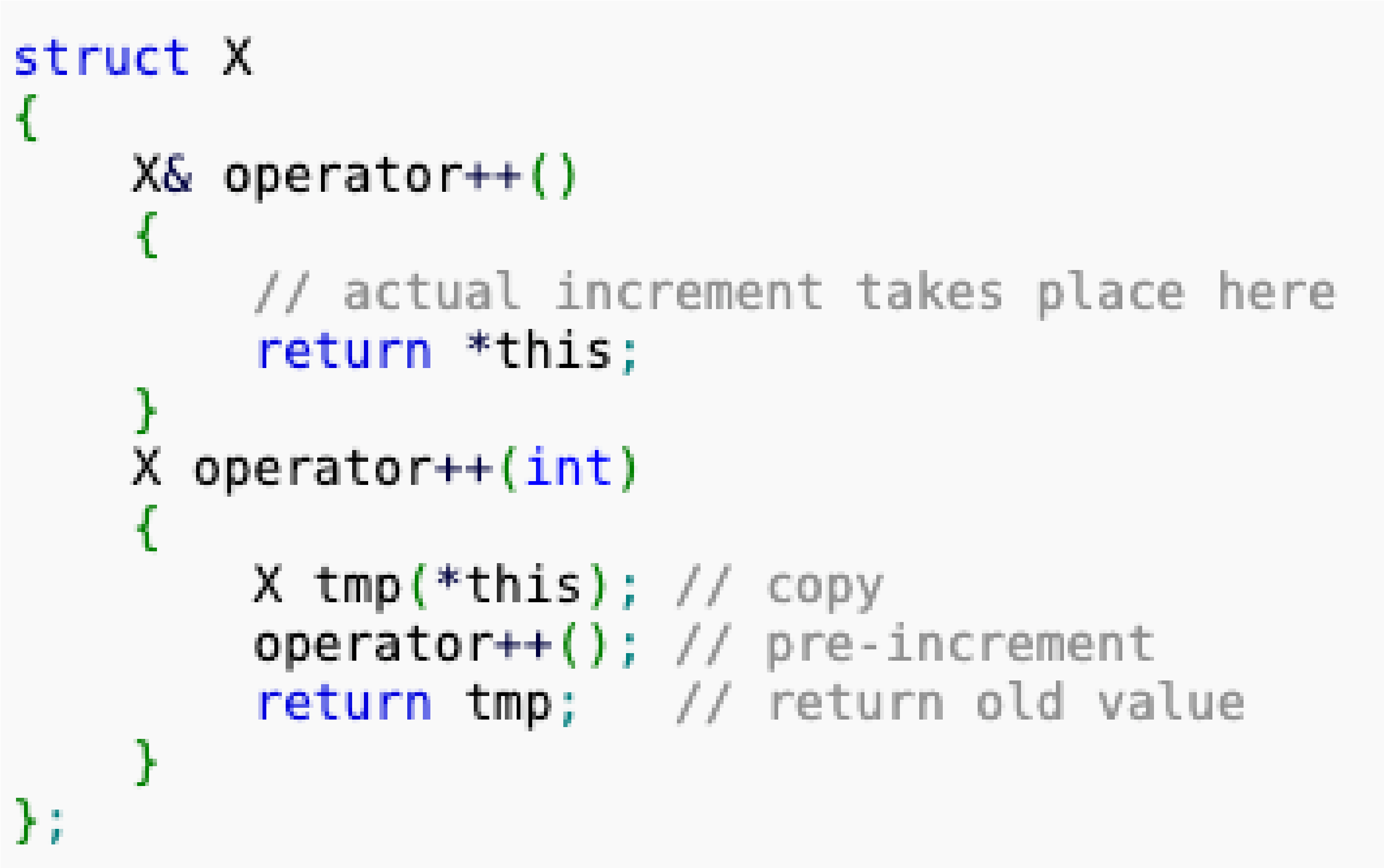


```
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = x + y;
  cout << "z = " << z << endl;
  z = y = x;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;
  cout << (x!=y) << endl;
  return 0;
}
```



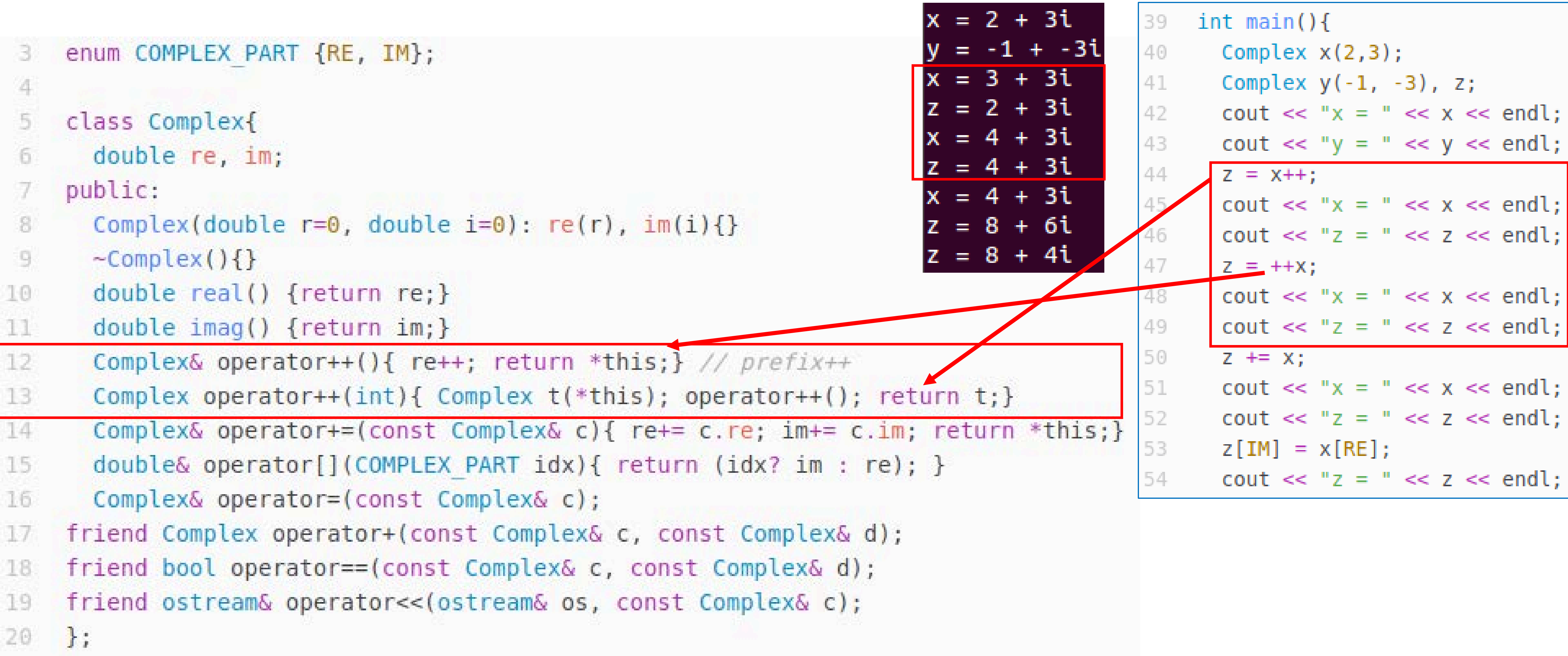
```
x = 2 + 3i
y = -1 + -3i
z = 1 + 0i
y = 2 + 3i
z = 2 + 3i
0
```

typical pre-/postfix ++ and -- operators



```
struct X
{
    X& operator++()
    {
        // actual increment takes place here
        return *this;
    }
    X operator++(int)
    {
        X tmp(*this); // copy
        operator++(); // pre-increment
        return tmp;   // return old value
    }
};
```

prefix++, postfix++ operators

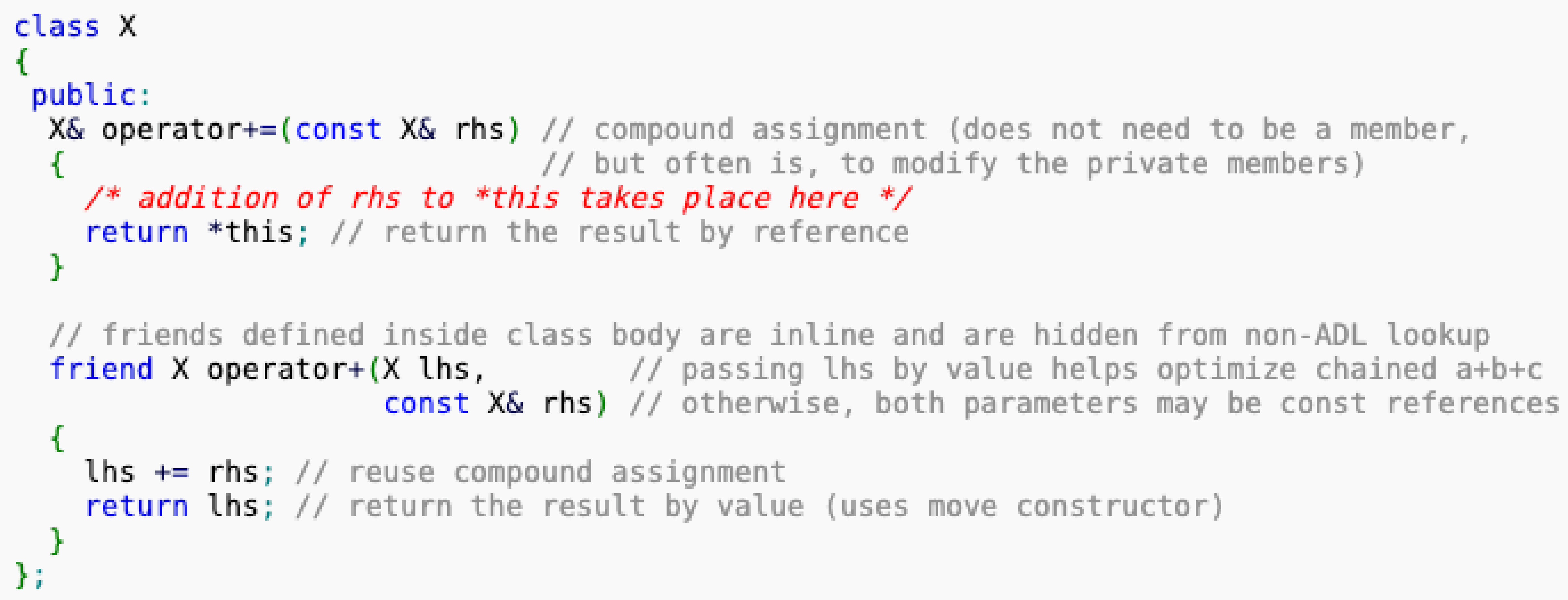


```cpp
enum COMPLEX_PART {RE, IM};

class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex& operator++(){ re++; return *this;} // prefix++
  Complex operator++(int){ Complex t(*this); operator++(); return t;}
  Complex& operator+=(const Complex& c){ re+= c.re; im+= c.im; return *this;}
  double& operator[](COMPLEX_PART idx){ return (idx? im : re); }
  Complex& operator=(const Complex& c);
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
friend ostream& operator<<(ostream& os, const Complex& c);
};
```

```
x = 2 + 3i
y = -1 + -3i
x = 3 + 3i
z = 2 + 3i
x = 4 + 3i
z = 4 + 3i
x = 4 + 3i
z = 8 + 6i
z = 8 + 4i
```

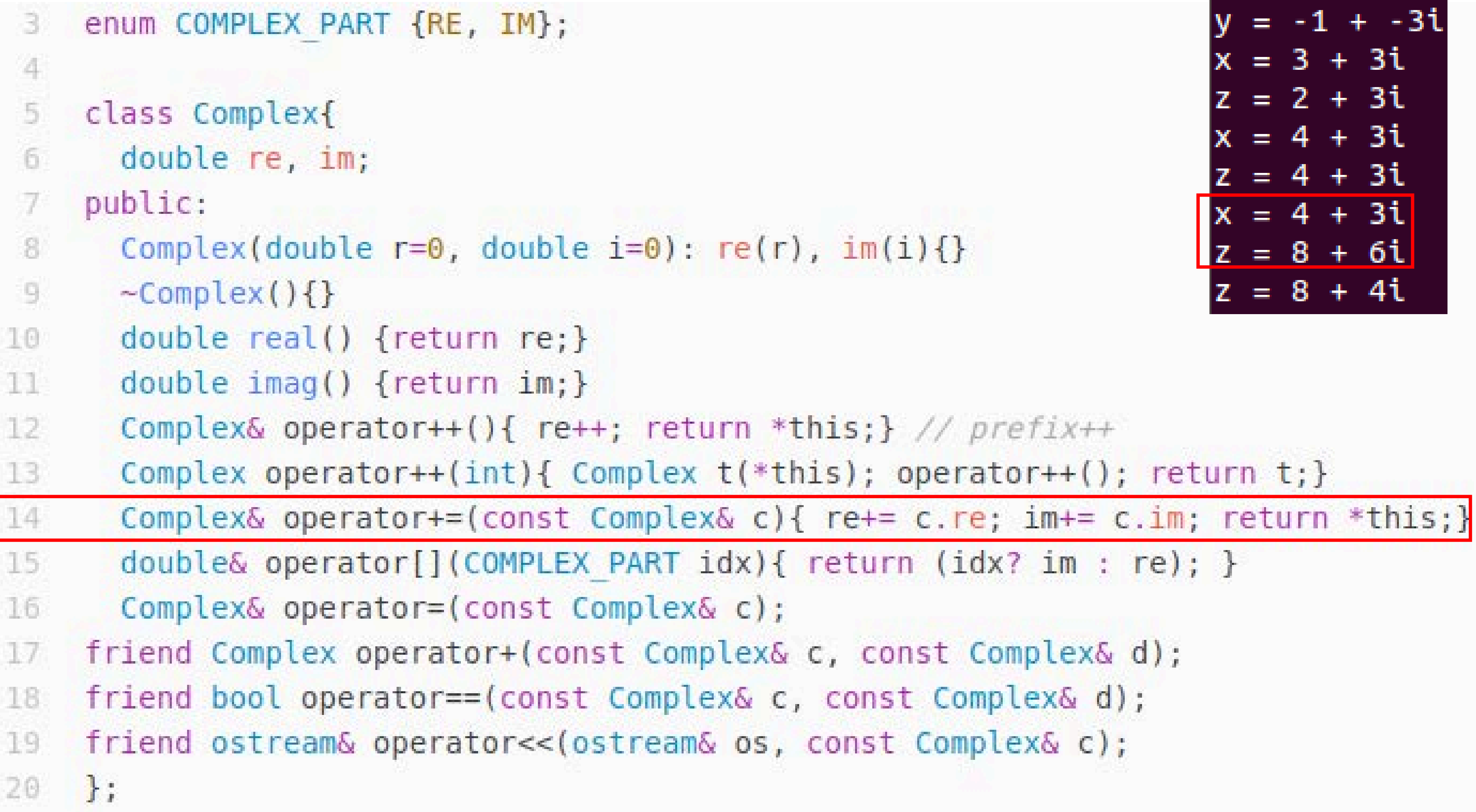
```cpp
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = x++;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z = ++x;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z += x;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z[IM] = x[RE];
  cout << "z = " << z << endl;
```

typical + and += operators

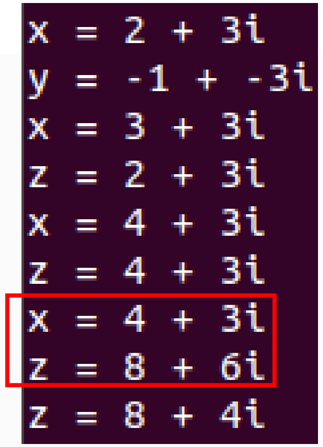


```
class X
{
 public:
  X& operator+=(const X& rhs) // compound assignment (does not need to be a member,
  {                           // but often is, to modify the private members)
    /* addition of rhs to *this takes place here */
    return *this; // return the result by reference
  }

  // friends defined inside class body are inline and are hidden from non-ADL lookup
  friend X operator+(X lhs,        // passing lhs by value helps optimize chained a+b+c
                     const X& rhs) // otherwise, both parameters may be const references
  {
    lhs += rhs; // reuse compound assignment
    return lhs; // return the result by value (uses move constructor)
  }
};
```

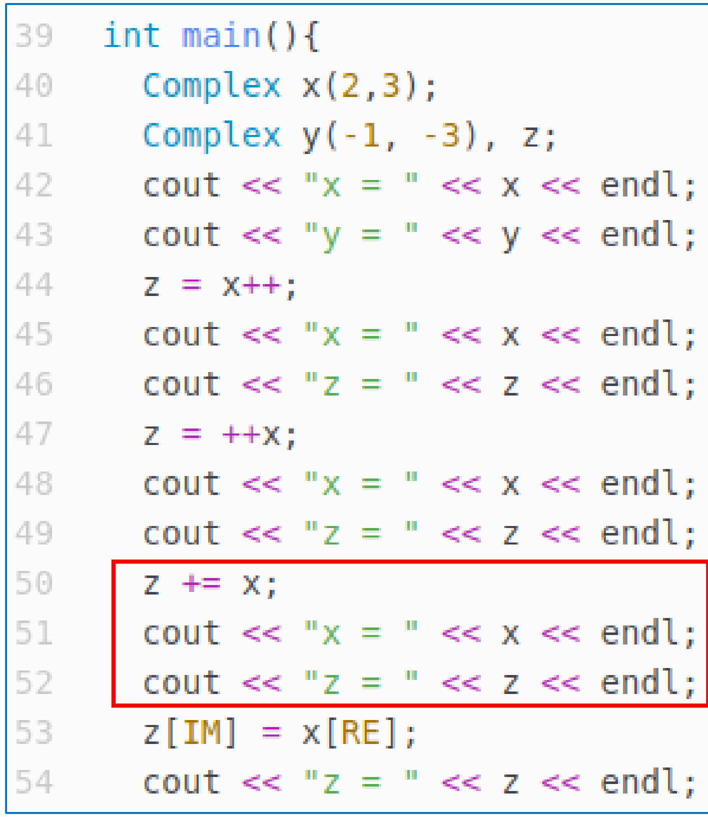
+= operator



```
enum COMPLEX_PART {RE, IM};

class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex& operator++(){ re++; return *this;} // prefix++
  Complex operator++(int){ Complex t(*this); operator++(); return t;}
  Complex& operator+=(const Complex& c){ re+= c.re; im+= c.im; return *this;}
  double& operator[](COMPLEX_PART idx){ return (idx? im : re); }
  Complex& operator=(const Complex& c);
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
friend ostream& operator<<(ostream& os, const Complex& c);
};
```

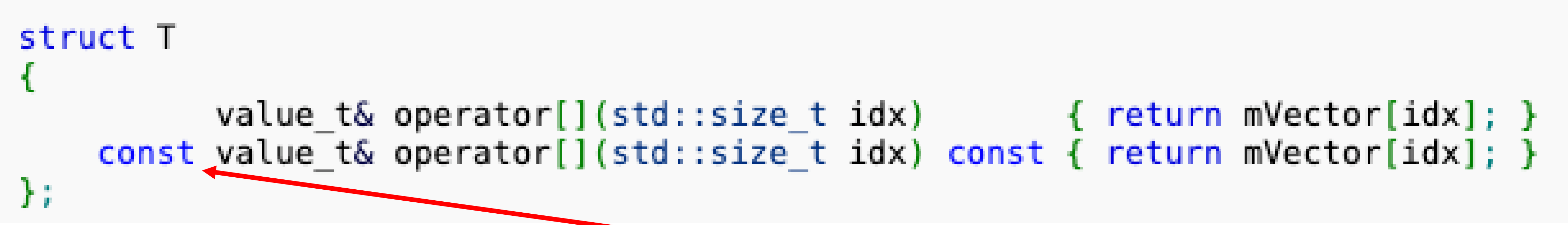


```
x = 2 + 3i
y = -1 + -3i
x = 3 + 3i
z = 2 + 3i
x = 4 + 3i
z = 4 + 3i
x = 4 + 3i
z = 8 + 6i
z = 8 + 4i
```



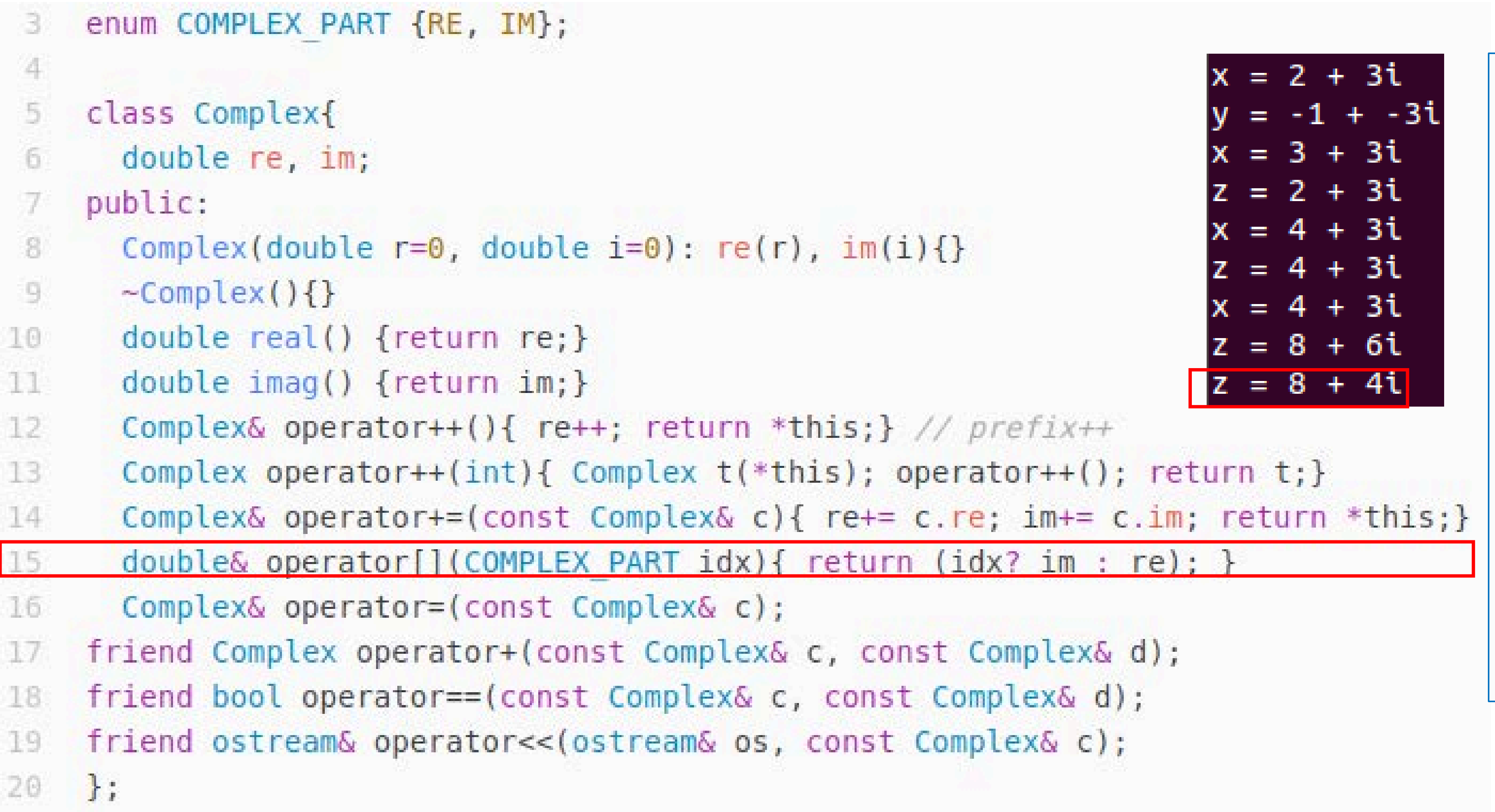
```
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = x++;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z = ++x;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z += x;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z[IM] = x[RE];
  cout << "z = " << z << endl;
```

typical [] operator



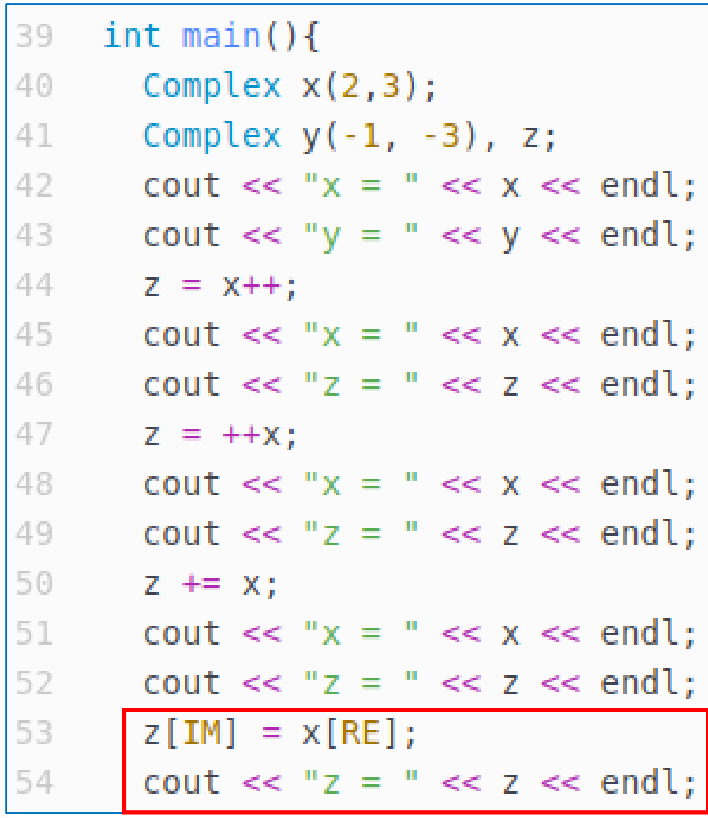
```
struct T
{
          value_t& operator[](std::size_t idx)       { return mVector[idx]; }
    const value_t& operator[](std::size_t idx) const { return mVector[idx]; }
};
```

상수형 객체를 위해 정의할 필요가 있다.



```cpp
enum COMPLEX_PART {RE, IM};

class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  ~Complex(){}
  double real() {return re;}
  double imag() {return im;}
  Complex& operator++(){ re++; return *this;} // prefix++
  Complex operator++(int){ Complex t(*this); operator++(); return t;}
  Complex& operator+=(const Complex& c){ re+= c.re; im+= c.im; return *this;}
  double& operator[](COMPLEX_PART idx){ return (idx? im : re); }
  Complex& operator=(const Complex& c);
friend Complex operator+(const Complex& c, const Complex& d);
friend bool operator==(const Complex& c, const Complex& d);
friend ostream& operator<<(ostream& os, const Complex& c);
};
```

```
x = 2 + 3i
y = -1 + -3i
x = 3 + 3i
z = 2 + 3i
x = 4 + 3i
z = 4 + 3i
x = 4 + 3i
z = 8 + 6i
z = 8 + 4i
```



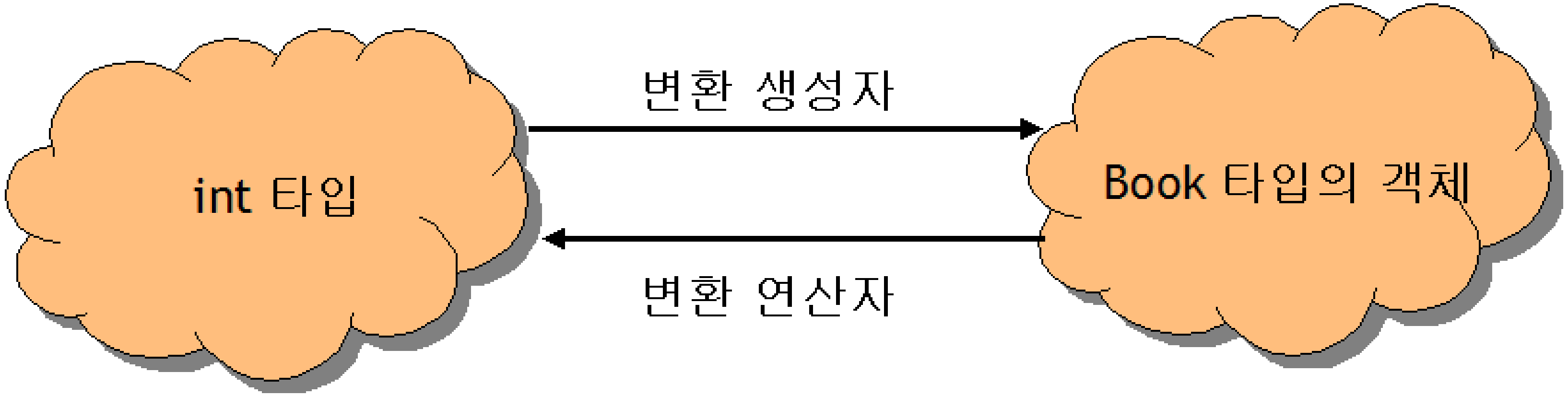
```cpp
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = x++;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z = ++x;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z += x;
  cout << "x = " << x << endl;
  cout << "z = " << z << endl;
  z[IM] = x[RE];
  cout << "z = " << z << endl;
```

연산자 중복시 주의할 점

- 새로운 연산자를 만드는 것은 허용되지 않는다.
- :: 연산자, .* 연산자, . 연산자, ?: 연산자는 중복이 불가능하다.
- 내장된 int형이나 double형에 대한 연산자의 의미를 변경할 수는 없다.
- 연산자들의 우선 순위나 결합 법칙은 변경되지 않는다.
- 만약 + 연산자를 오버로딩하였다면 일관성을 위하여 +=, -= 연산자도 오버로딩하는 것이 좋다.
- 일반적으로 산술 연산자와 관계 연산자는 비멤버 함수로 정의한다.
- 반면에 할당 연산자는 멤버 함수로 정의한다.

타입 변환

- 클래스의 객체들도 하나의 타입에서 다른 타입으로 자동적인 변환이 가능하다.
- 이것은 변환 생성자(conversion constructor)와 변환 연산자(conversion operator)에 의하여 가능하다.



변환 생성자와 변환 연산자

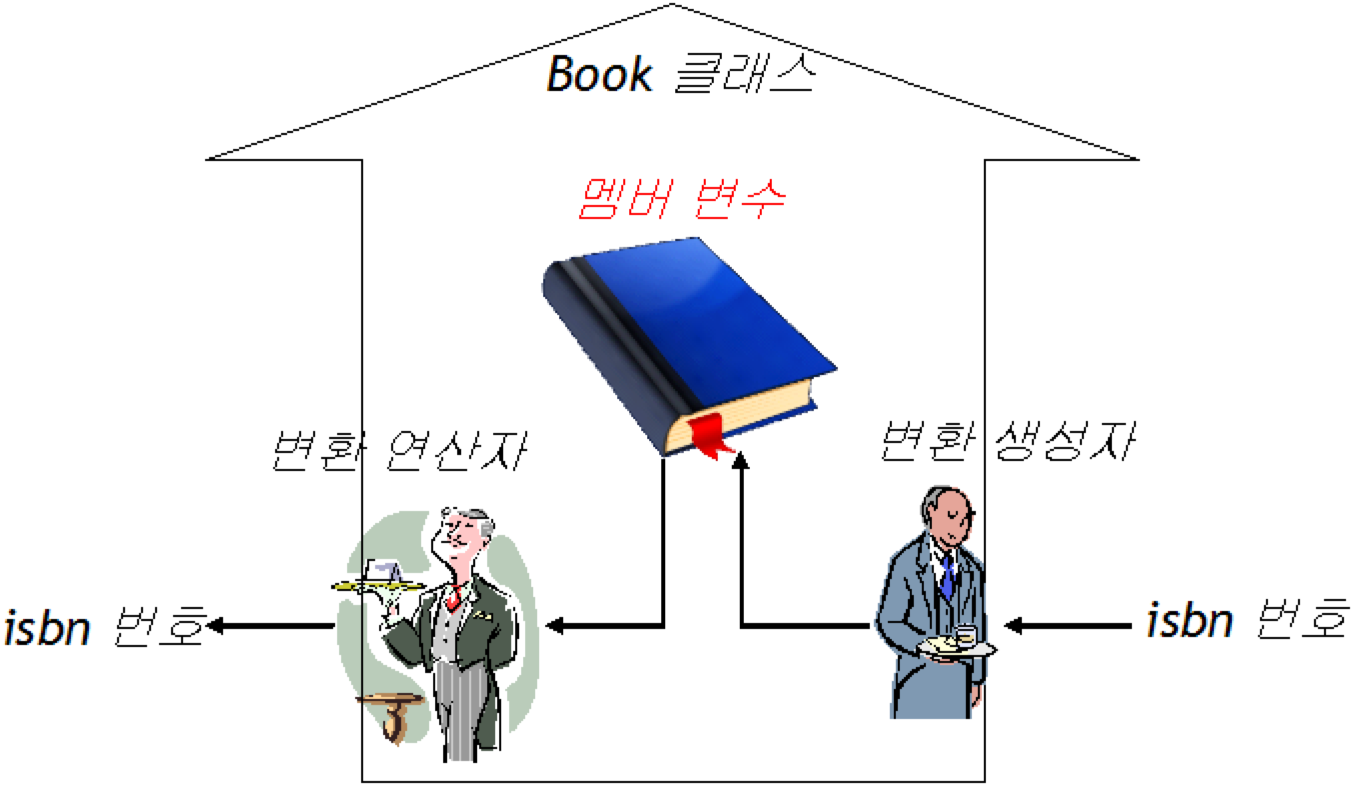
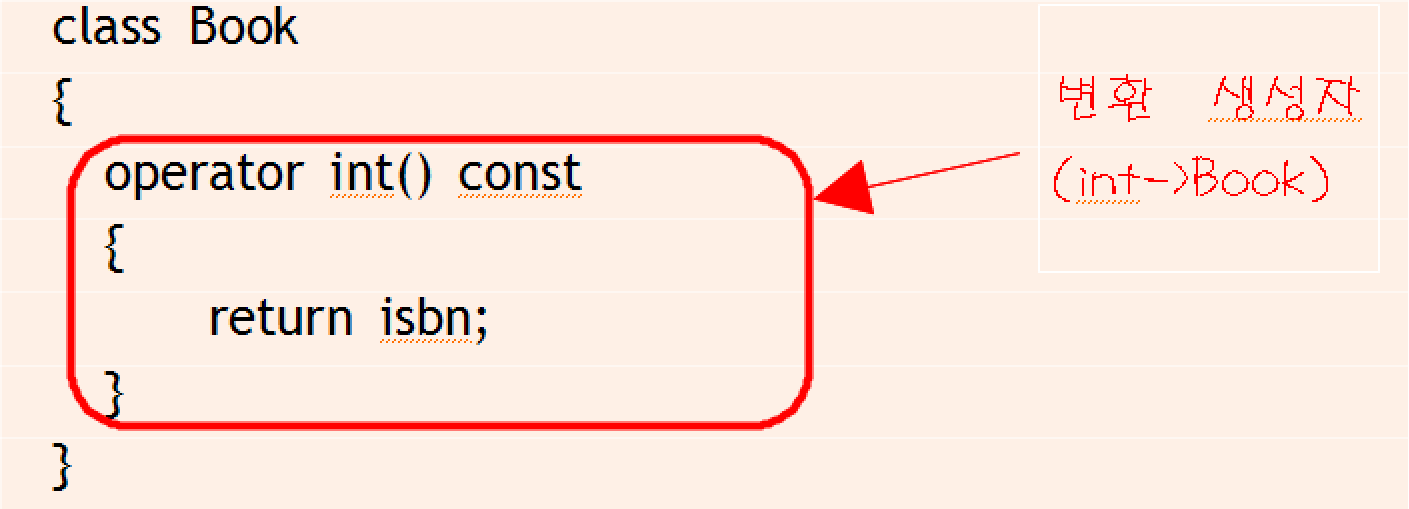


그림 12.7 변환 생성자와 변환 연산자

변환 연산자 함수

```
operator 데이터_타입()
{
....
}
```



```
class Book
{
    operator int() const
    {
        return isbn;
    }
}
```

변환 생성자
(int->Book)

변환 연산자

```
#include <iostream>
#include <string>
using namespace std;
class Book
{
private:
	int isbn;
	string title;
public:
	Book(int isbn, string& title) {
		this->isbn = isbn;
		this->title = title;
	}
	Book(int isbn) {
		this->isbn = isbn;
		this->title = "unknown";
	}
	operator int() const
	{
		return isbn;
	}
```

변환 생성자
(int->Book)

변환 연산자
(Book->int)

예제

```
void display() {
          cout << isbn << ":" << title << endl;
    }
};

bool check(int isbn)
{
    cout << isbn << endl;
    return true;
}

int main()
{
    Book b1 = 9782001;        // int->Book 변환 생성자 실행!
    b1.display();
    int isbn = b1;            // Book-> int 변환 연산자 실행!
    cout << isbn << endl;
    check(b1);                // Book->int 변환 연산자 실행!

    return 0;
}
```

버그의 원인

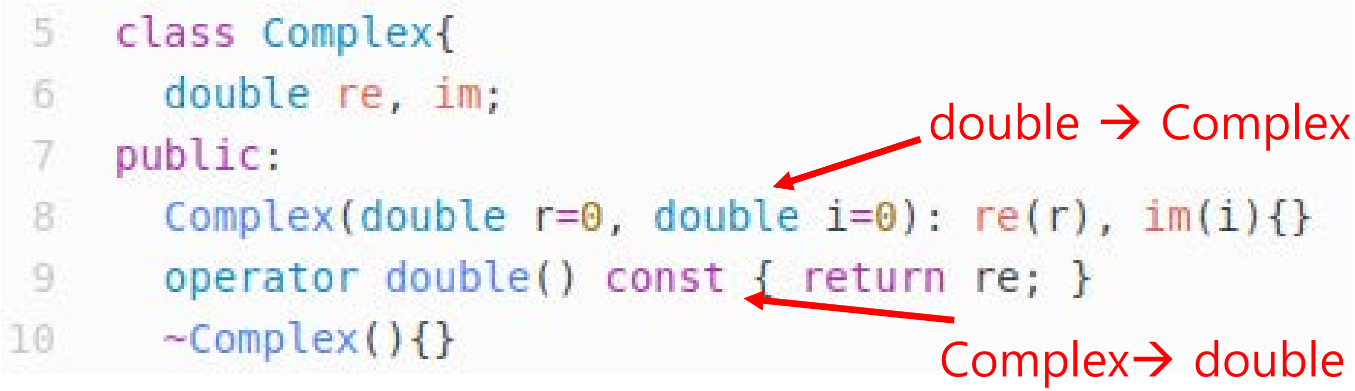
- 변환 생성자는 버그의 원인이 될 수도 있다.

- Book b2 = 3.141592;
- b2.display();

- (설명) 실수->정수-> 객체

- (해결책)만약 생성자 앞에 explicit를 붙이면 컴파일러가 자동적으로 타입 변환을 하지 못한다.

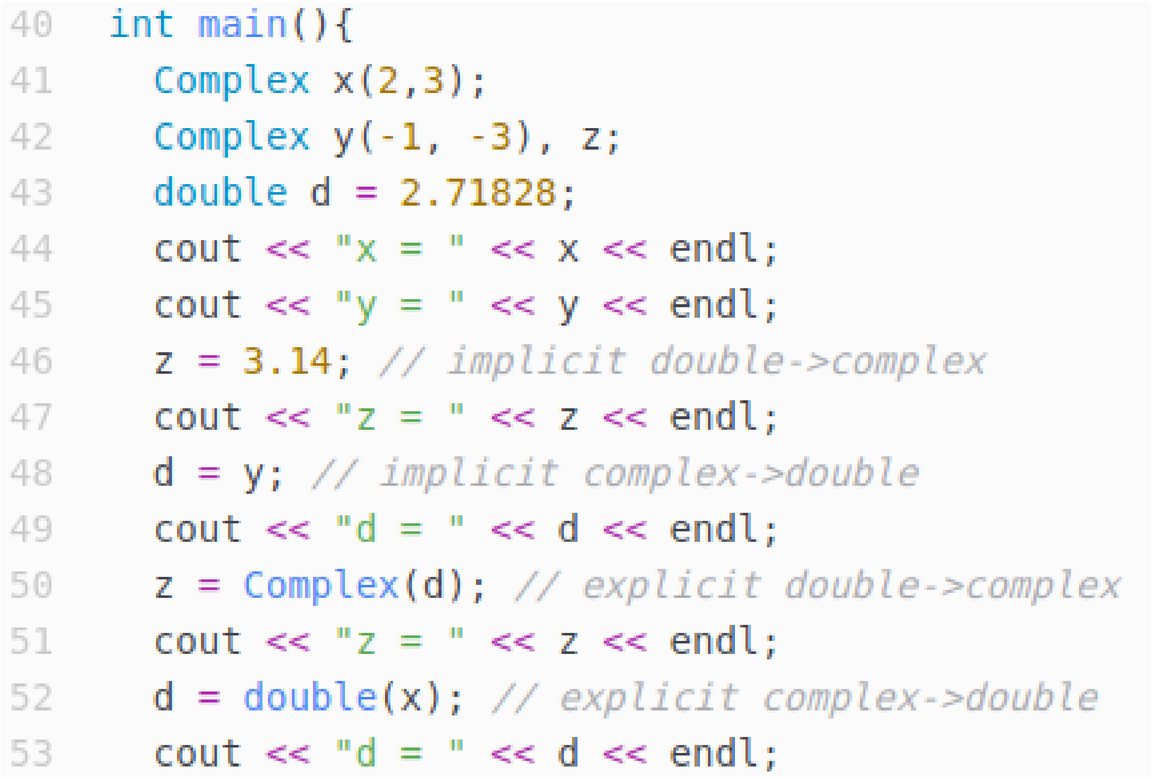
(Complex)d   same as Complex(d)



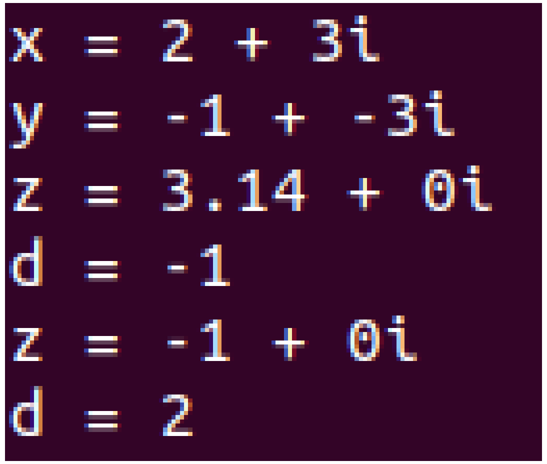
```
class Complex{
  double re, im;
public:
  Complex(double r=0, double i=0): re(r), im(i){}
  operator double() const { return re; }
  ~Complex(){}
```

double → Complex

Complex→ double

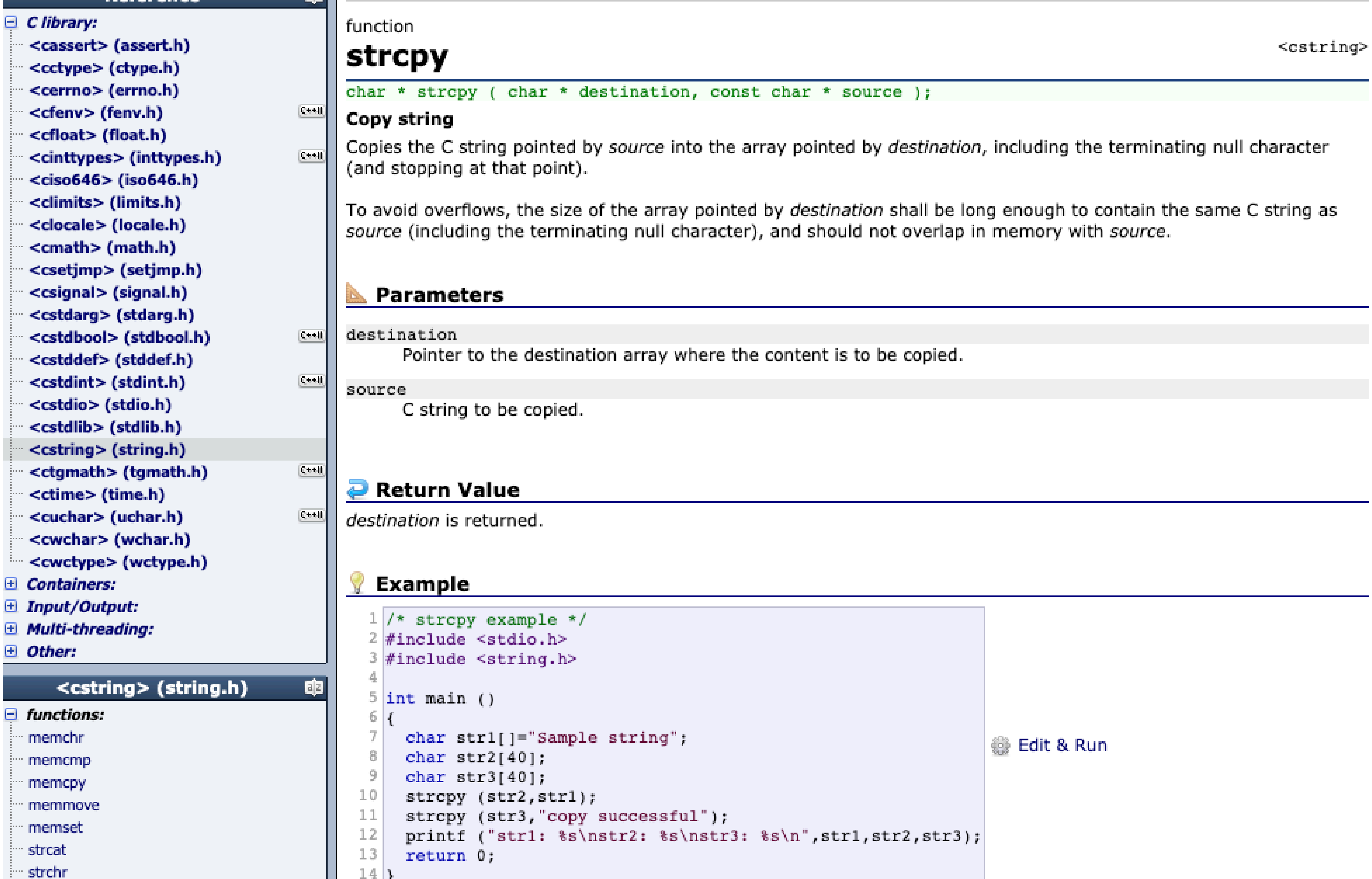


```
int main(){
  Complex x(2,3);
  Complex y(-1, -3), z;
  double d = 2.71828;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  z = 3.14; // implicit double->complex
  cout << "z = " << z << endl;
  d = y; // implicit complex->double
  cout << "d = " << d << endl;
  z = Complex(d); // explicit double->complex
  cout << "z = " << z << endl;
  d = double(x); // explicit complex->double
  cout << "d = " << d << endl;
```



```
x = 2 + 3i
y = -1 + -3i
z = 3.14 + 0i
d = -1
z = -1 + 0i
d = 2
```

MyString class 의 연산자 중복



C library:

- <cassert> (assert.h)
- <cctype> (ctype.h)
- <cerrno> (errno.h)
- <cfenv> (fenv.h) C++11
- <cfloat> (float.h)
- <cinttypes> (inttypes.h) C++11
- <ciso646> (iso646.h)
- <climits> (limits.h)
- <clocale> (locale.h)
- <cmath> (math.h)
- <csetjmp> (setjmp.h)
- <csignal> (signal.h)
- <cstdarg> (stdarg.h)
- <cstdbool> (stdbool.h) C++11
- <cstddef> (stddef.h)
- <cstdint> (stdint.h) C++11
- <cstdio> (stdio.h)
- <cstdlib> (stdlib.h)
- <cstring> (string.h)
- <ctgmath> (tgmath.h) C++11
- <ctime> (time.h)
- <cuchar> (uchar.h) C++11
- <cwchar> (wchar.h)
- <cwctype> (wctype.h)

⊞ **Containers:**

⊞ **Input/Output:**

⊞ **Multi-threading:**

⊞ **Other:**

<cstring> (string.h)

functions:

- memchr
- memcmp
- memcpy
- memmove
- memset
- strcat
- strchr

function

strcpy

<cstring>

```
char * strcpy ( char * destination, const char * source );
```

Copy string

Copies the C string pointed by *source* into the array pointed by *destination*, including the terminating null character (and stopping at that point).

To avoid overflows, the size of the array pointed by *destination* shall be long enough to contain the same C string as *source* (including the terminating null character), and should not overlap in memory with *source*.

Parameters

destination
: Pointer to the destination array where the content is to be copied.

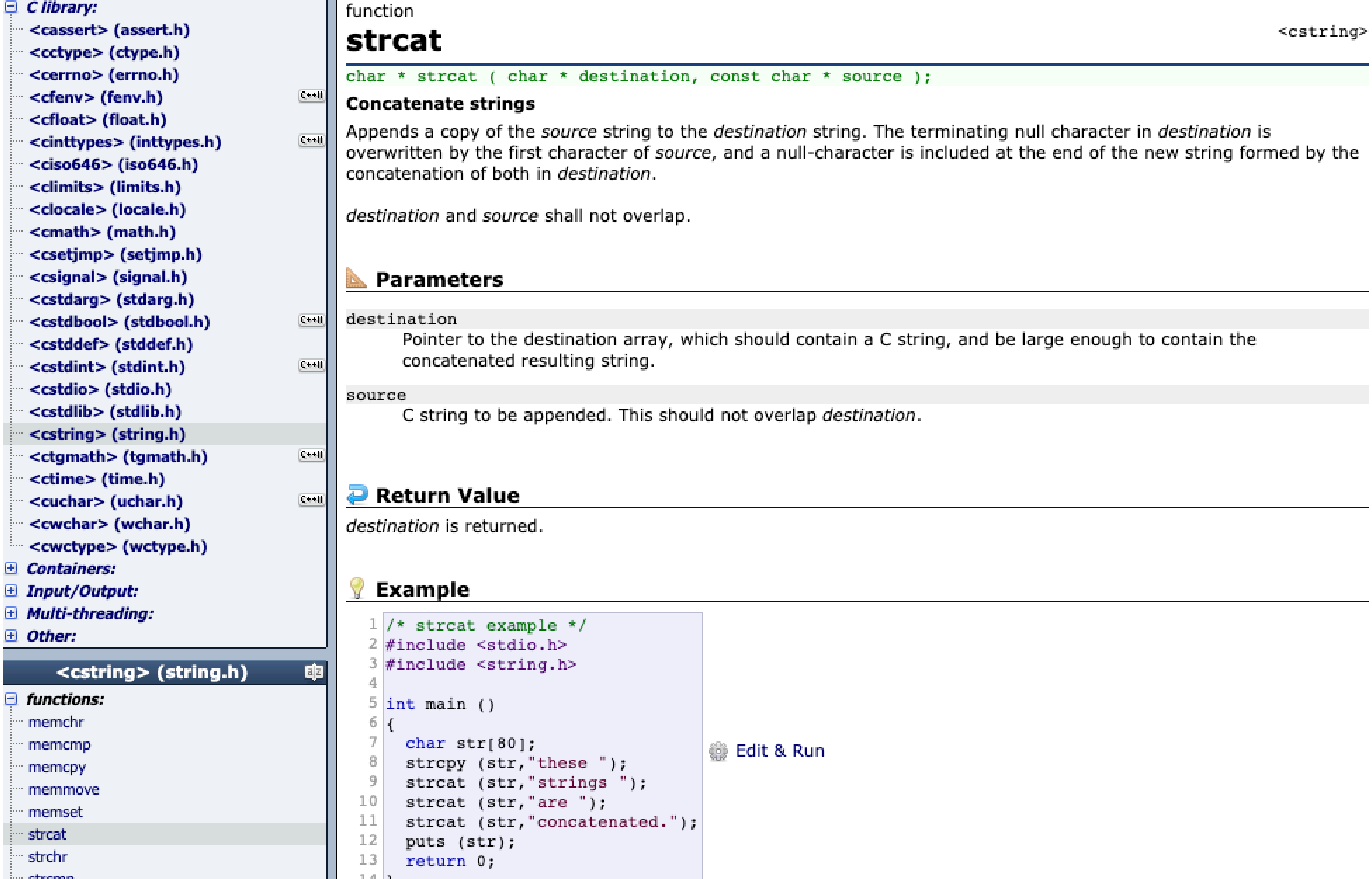
source
: C string to be copied.

Return Value

destination is returned.

Example

```
/* strcpy example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[]="Sample string";
  char str2[40];
  char str3[40];
  strcpy (str2,str1);
  strcpy (str3,"copy successful");
  printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
  return 0;
}
```



function

strcat

<cstring>

```
char * strcat ( char * destination, const char * source );
```

Concatenate strings

Appends a copy of the *source* string to the *destination* string. The terminating null character in *destination* is overwritten by the first character of *source*, and a null-character is included at the end of the new string formed by the concatenation of both in *destination*.

destination and *source* shall not overlap.

Parameters

`destination`
Pointer to the destination array, which should contain a C string, and be large enough to contain the concatenated resulting string.

`source`
C string to be appended. This should not overlap *destination*.

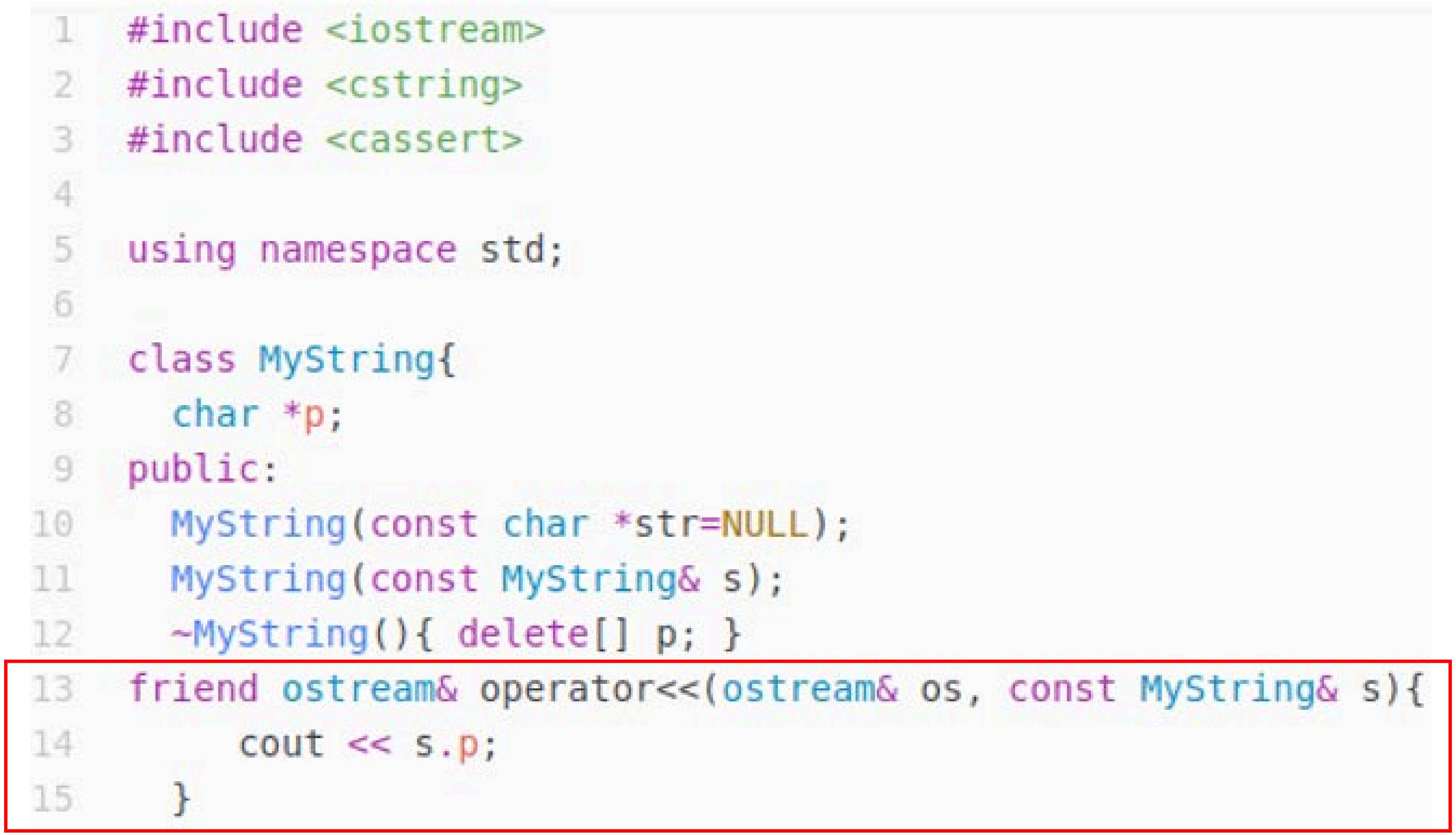
Return Value

destination is returned.

Example

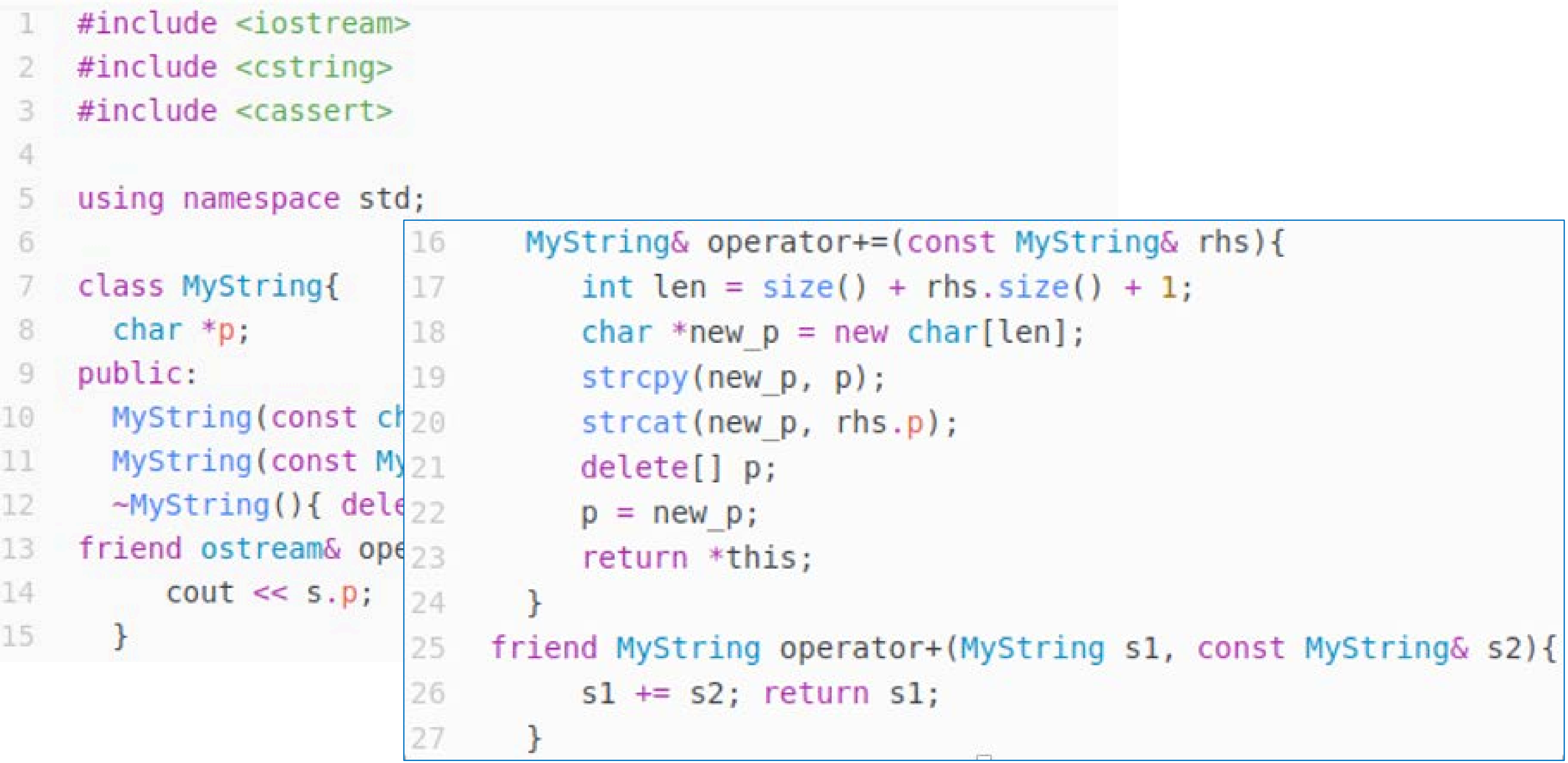
```
/* strcat example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[80];
  strcpy (str,"these ");
  strcat (str,"strings ");
  strcat (str,"are ");
  strcat (str,"concatenated.");
  puts (str);
  return 0;
}
```

MyString class 의 << 연산자



```cpp
#include <iostream>
#include <cstring>
#include <cassert>

using namespace std;

class MyString{
  char *p;
public:
  MyString(const char *str=NULL);
  MyString(const MyString& s);
  ~MyString(){ delete[] p; }
friend ostream& operator<<(ostream& os, const MyString& s){
     cout << s.p;
  }
```

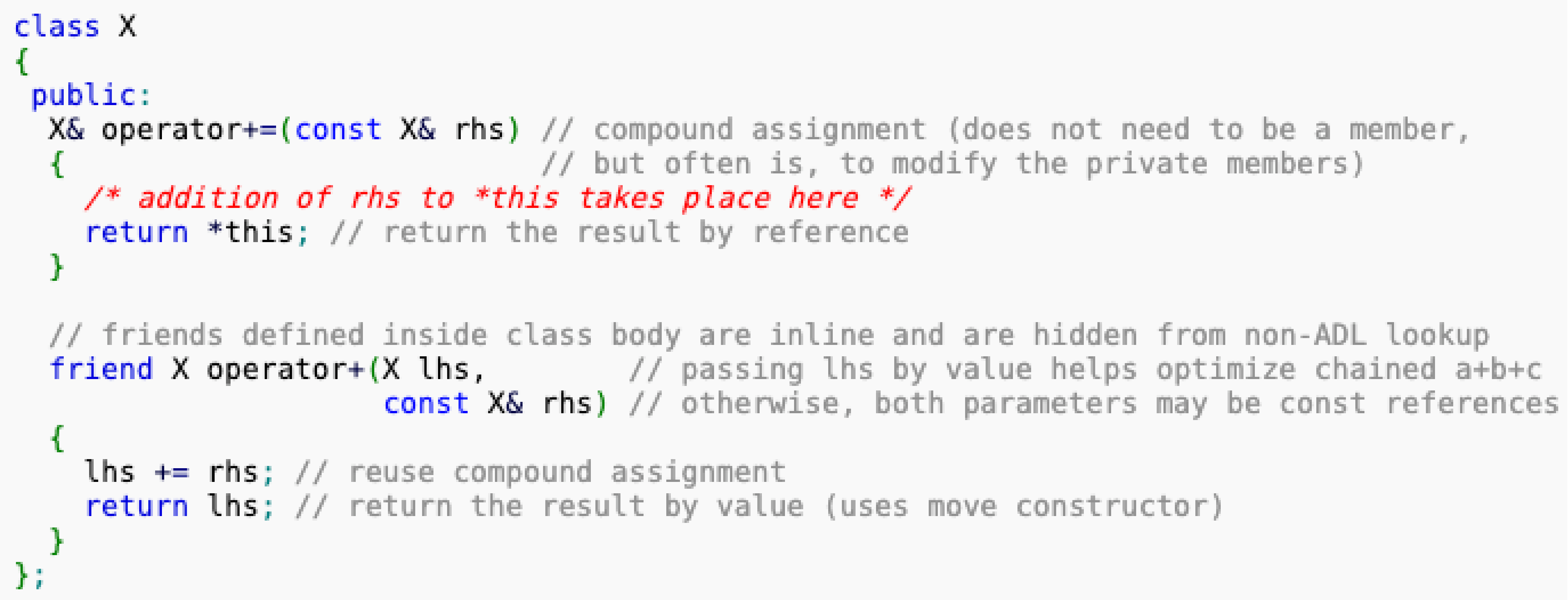
MyString class 의 +=, + 연산자



```
#include <iostream>
#include <cstring>
#include <cassert>

using namespace std;

class MyString{
  char *p;
public:
  MyString(const ch
  MyString(const My
  ~MyString(){ dele
friend ostream& ope
      cout << s.p;
  }
```

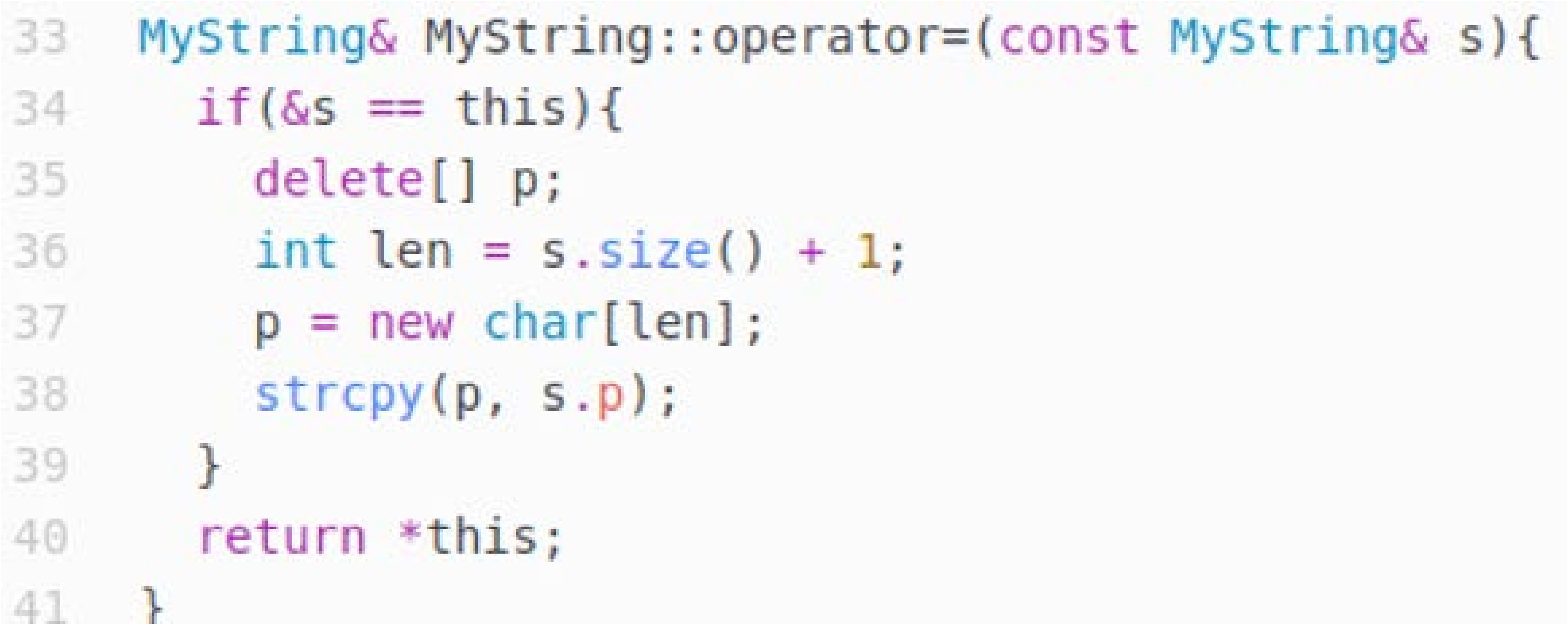
```
  MyString& operator+=(const MyString& rhs){
    int len = size() + rhs.size() + 1;
    char *new_p = new char[len];
    strcpy(new_p, p);
    strcat(new_p, rhs.p);
    delete[] p;
    p = new_p;
    return *this;
  }
friend MyString operator+(MyString s1, const MyString& s2){
    s1 += s2; return s1;
  }
```

typical + and += operators



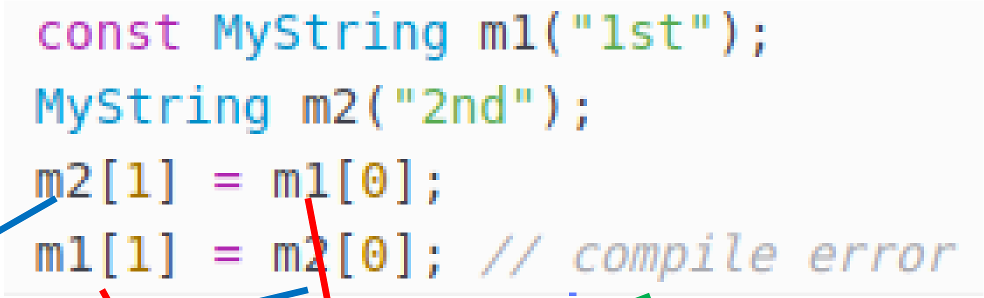
```
class X
{
 public:
  X& operator+=(const X& rhs) // compound assignment (does not need to be a member,
  {                           // but often is, to modify the private members)
    /* addition of rhs to *this takes place here */
    return *this; // return the result by reference
  }

  // friends defined inside class body are inline and are hidden from non-ADL lookup
  friend X operator+(X lhs,        // passing lhs by value helps optimize chained a+b+c
                     const X& rhs) // otherwise, both parameters may be const references
  {
    lhs += rhs; // reuse compound assignment
    return lhs; // return the result by value (uses move constructor)
  }
};
```

MyString class 의 = operator (깊은 복사)

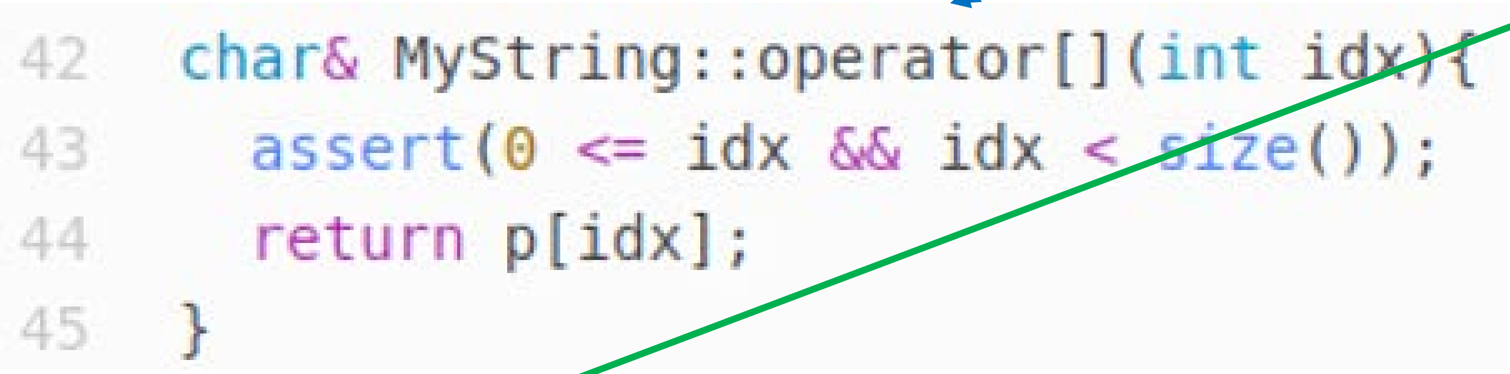


```
MyString& MyString::operator=(const MyString& s){
  if(&s == this){
    delete[] p;
    int len = s.size() + 1;
    p = new char[len];
    strcpy(p, s.p);
  }
  return *this;
}
```

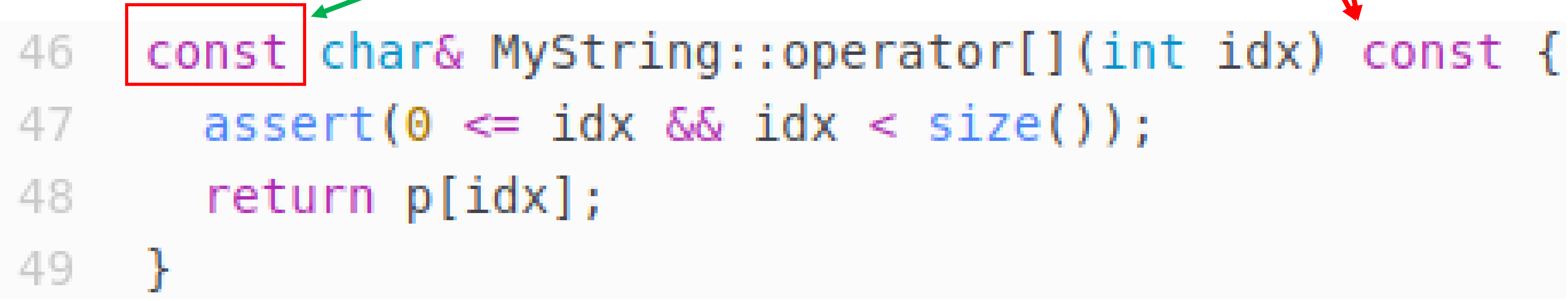
MyString class 의 [] operator



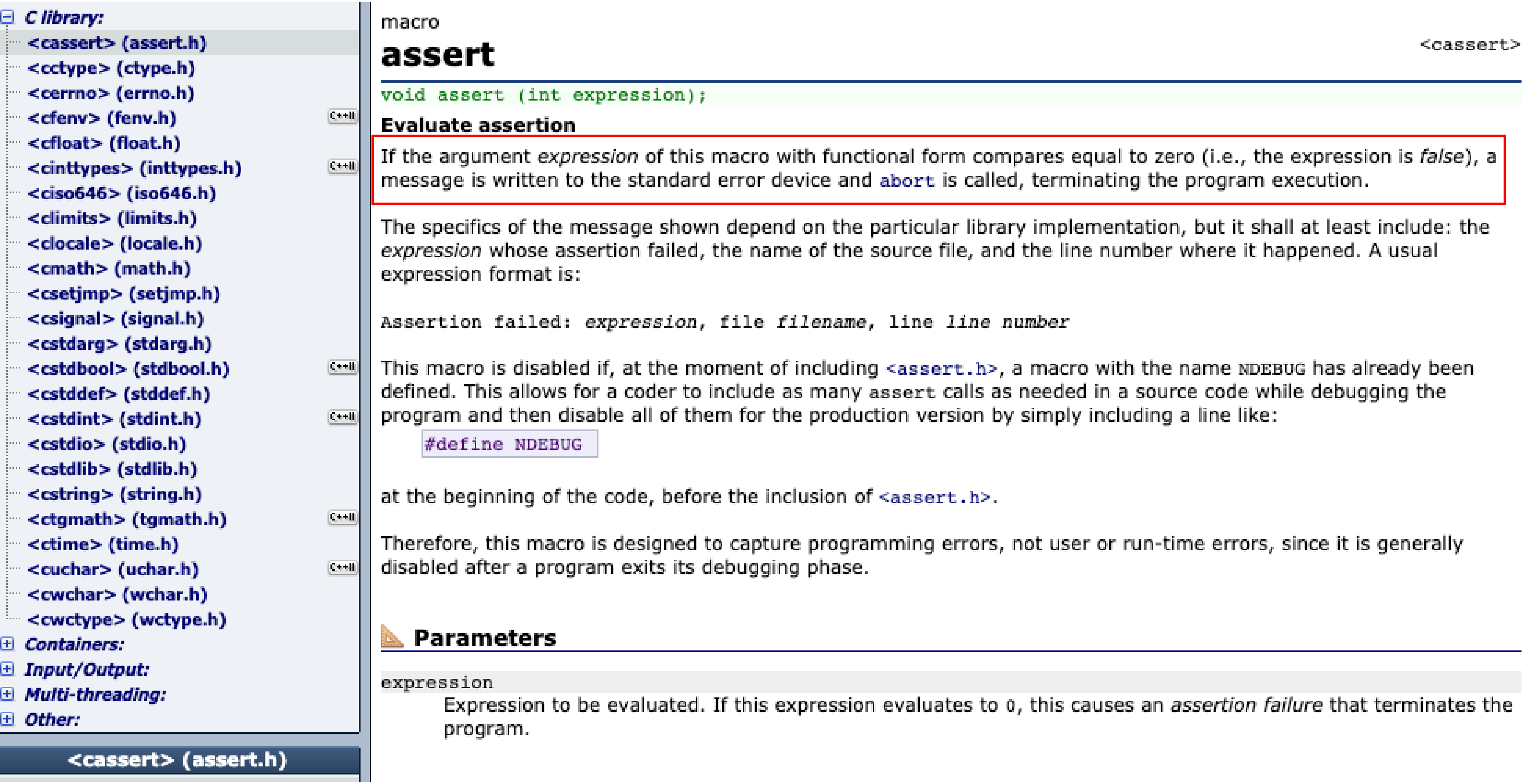
```
const MyString m1("1st");
MyString m2("2nd");
m2[1] = m1[0];
m1[1] = m2[0]; // compile error
```



```
char& MyString::operator[](int idx){
  assert(0 <= idx && idx < size());
  return p[idx];
}
```



```
const char& MyString::operator[](int idx) const {
  assert(0 <= idx && idx < size());
  return p[idx];
}
```



macro

assert

<cassert>

```
void assert (int expression);
```

Evaluate assertion

If the argument *expression* of this macro with functional form compares equal to zero (i.e., the expression is *false*), a message is written to the standard error device and abort is called, terminating the program execution.

The specifics of the message shown depend on the particular library implementation, but it shall at least include: the *expression* whose assertion failed, the name of the source file, and the line number where it happened. A usual expression format is:

```
Assertion failed: expression, file filename, line line number
```

This macro is disabled if, at the moment of including <assert.h>, a macro with the name NDEBUG has already been defined. This allows for a coder to include as many assert calls as needed in a source code while debugging the program and then disable all of them for the production version by simply including a line like:

```
#define NDEBUG
```

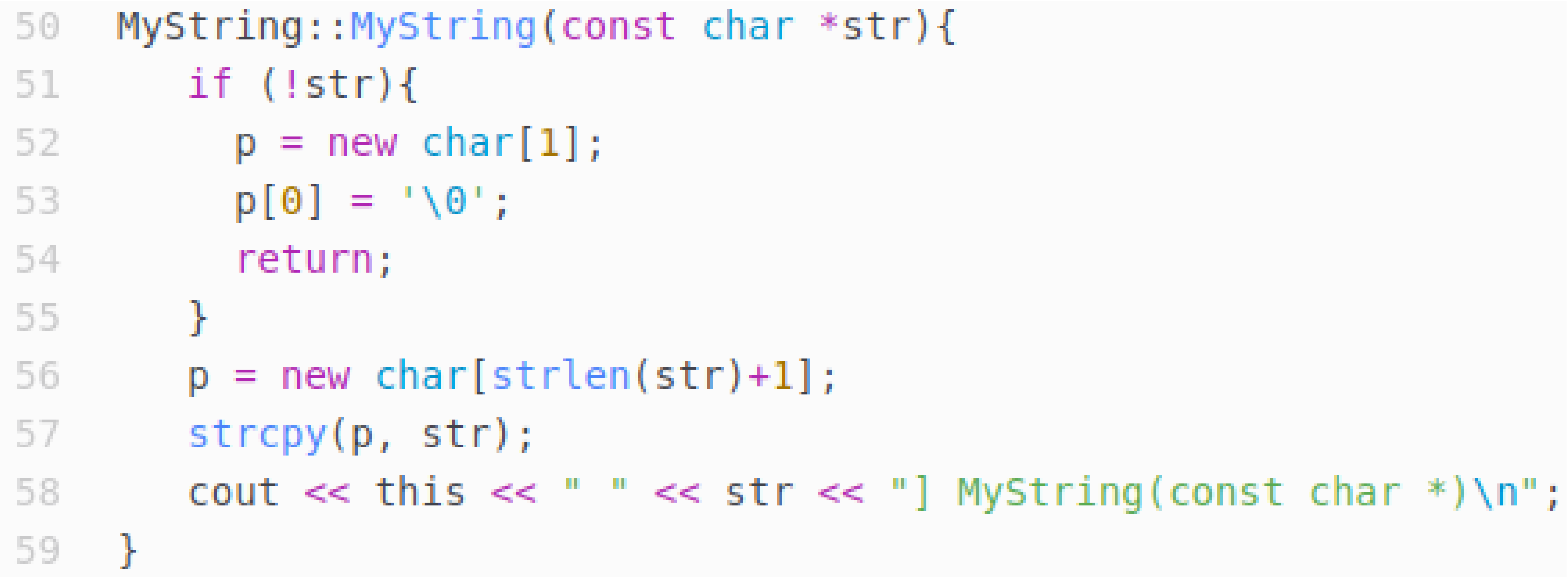
at the beginning of the code, before the inclusion of <assert.h>.

Therefore, this macro is designed to capture programming errors, not user or run-time errors, since it is generally disabled after a program exits its debugging phase.

Parameters

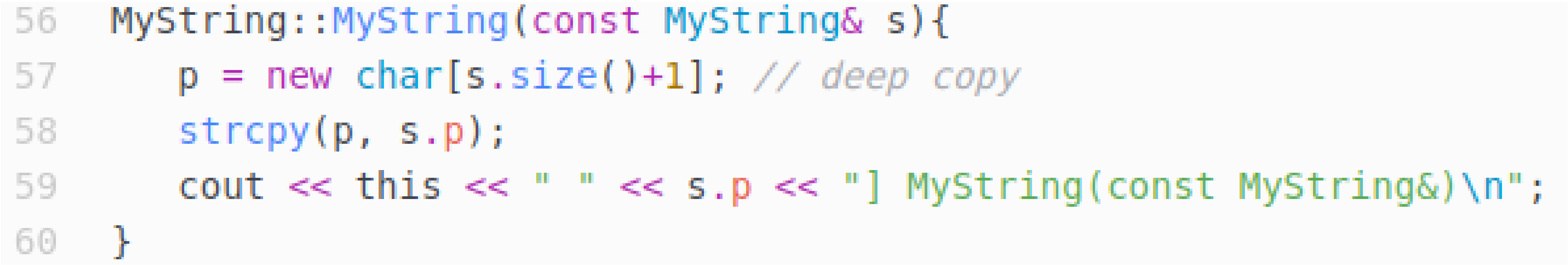
expression
: Expression to be evaluated. If this expression evaluates to 0, this causes an *assertion failure* that terminates the program.

MyString class 의 변환 생성자 (char * → MyString)

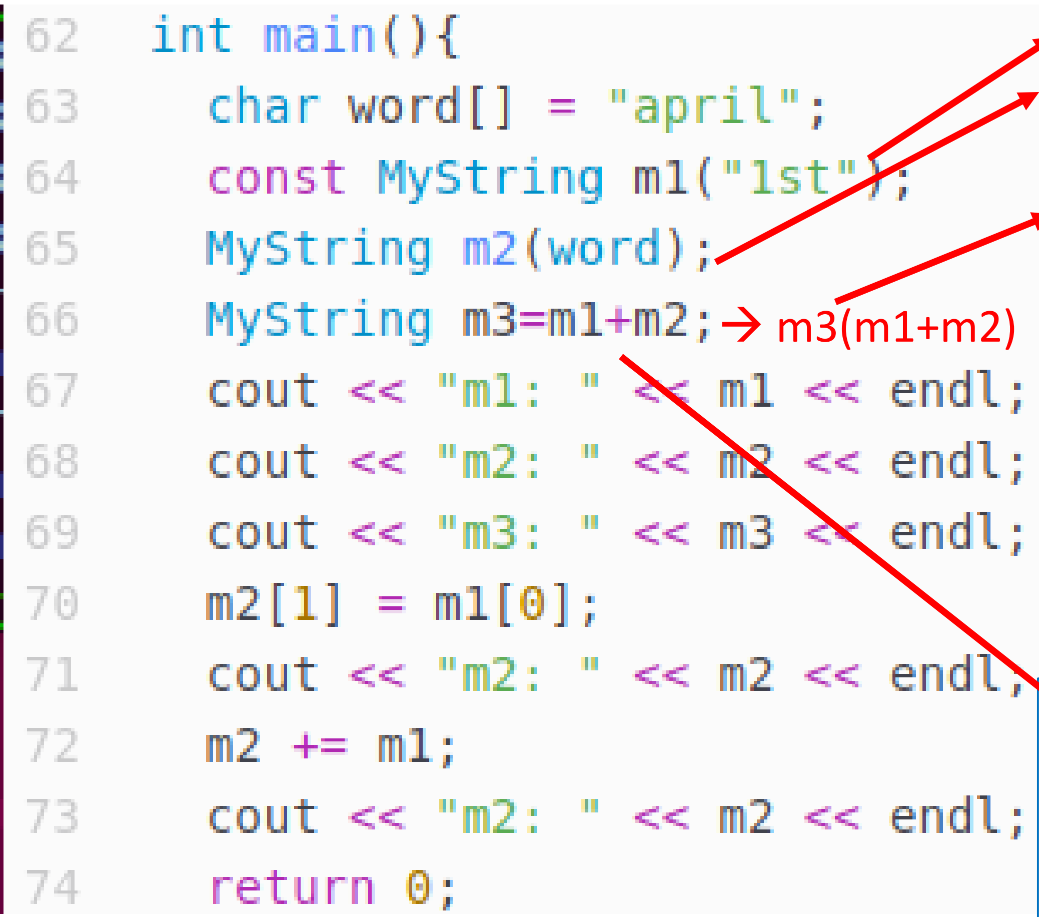


```cpp
MyString::MyString(const char *str){
   if (!str){
     p = new char[1];
     p[0] = '\0';
     return;
   }
   p = new char[strlen(str)+1];
   strcpy(p, str);
   cout << this << " " << str << "] MyString(const char *)\n";
}
```

MyString class 의 복사 생성자 (깊은 복사)

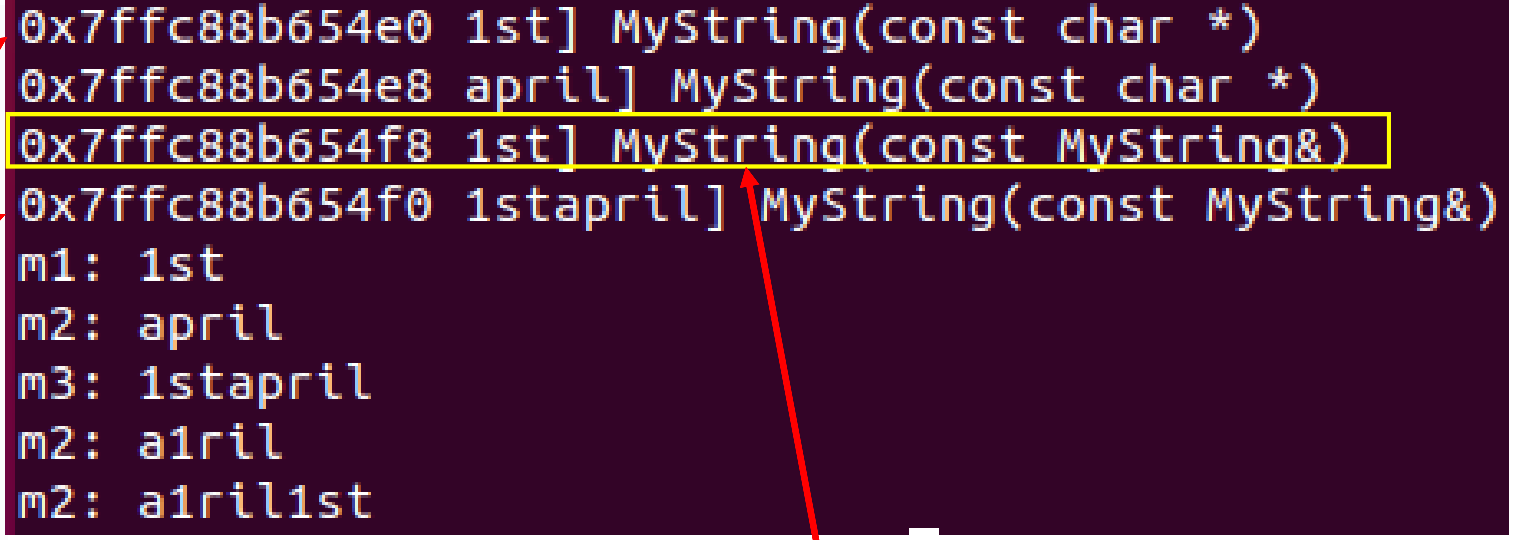


```cpp
MyString::MyString(const MyString& s){
    p = new char[s.size()+1]; // deep copy
    strcpy(p, s.p);
    cout << this << " " << s.p << "] MyString(const MyString&)\n";
}
```

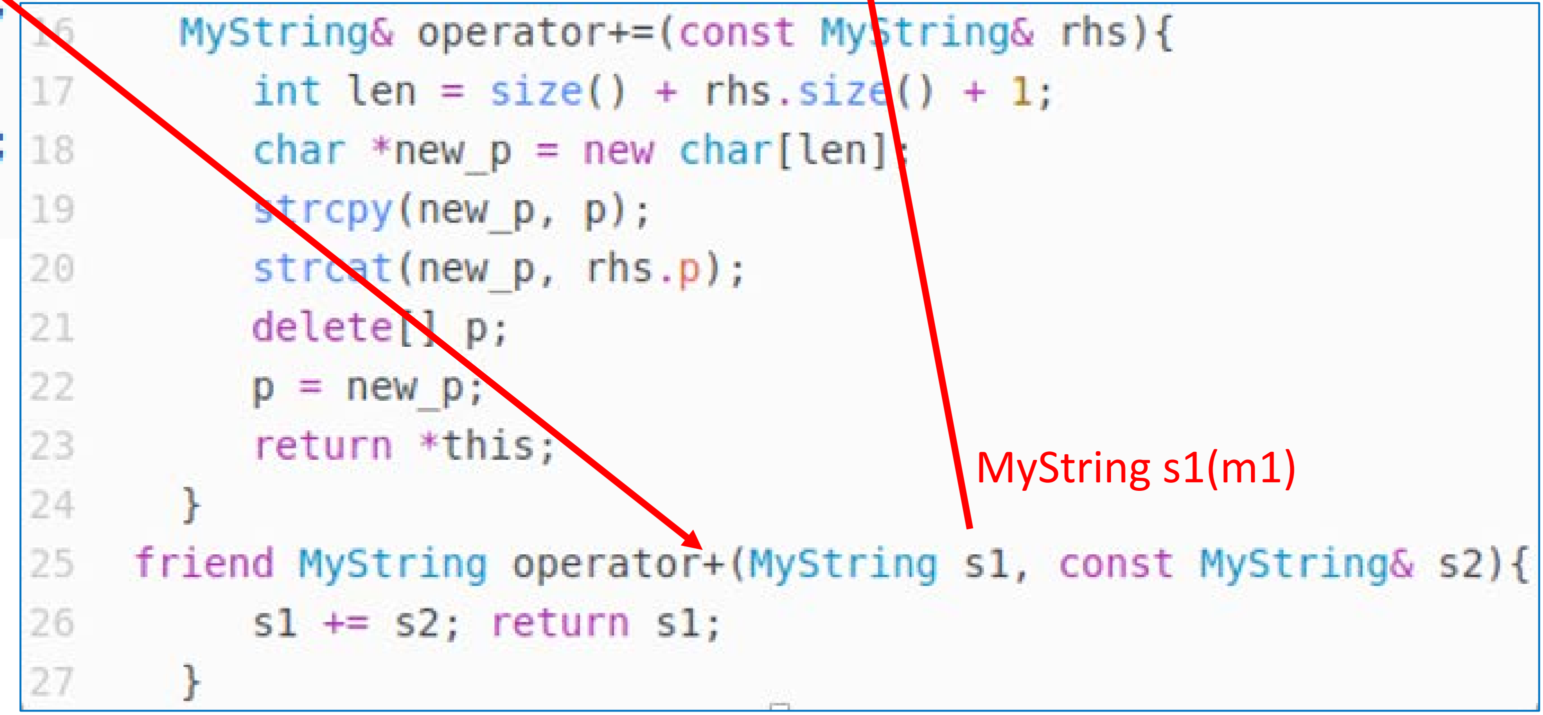


```
int main(){
  char word[] = "april";
  const MyString m1("1st");
  MyString m2(word);
  MyString m3=m1+m2;
  cout << "m1: " << m1 << endl;
  cout << "m2: " << m2 << endl;
  cout << "m3: " << m3 << endl;
  m2[1] = m1[0];
  cout << "m2: " << m2 << endl;
  m2 += m1;
  cout << "m2: " << m2 << endl;
  return 0;
```

→ m3(m1+m2)



```
0x7ffc88b654e0 1st] MyString(const char *)
0x7ffc88b654e8 april] MyString(const char *)
0x7ffc88b654f8 1st] MyString(const MyString&)
0x7ffc88b654f0 1stapril] MyString(const MyString&)
m1: 1st
m2: april
m3: 1stapril
m2: a1ril
m2: a1ril1st
```



```
    MyString& operator+=(const MyString& rhs){
        int len = size() + rhs.size() + 1;
        char *new_p = new char[len];
        strcpy(new_p, p);
        strcat(new_p, rhs.p);
        delete[] p;
        p = new_p;
        return *this;
    }
  friend MyString operator+(MyString s1, const MyString& s2){
        s1 += s2; return s1;
    }
```

MyString s1(m1)

실습 : Kvector class 에 =, ==, !=, [], << 연산자들을 class 외부에 구현하라.

(1) = 대입 연산자는 깊은 복사로 구현해야 한다.

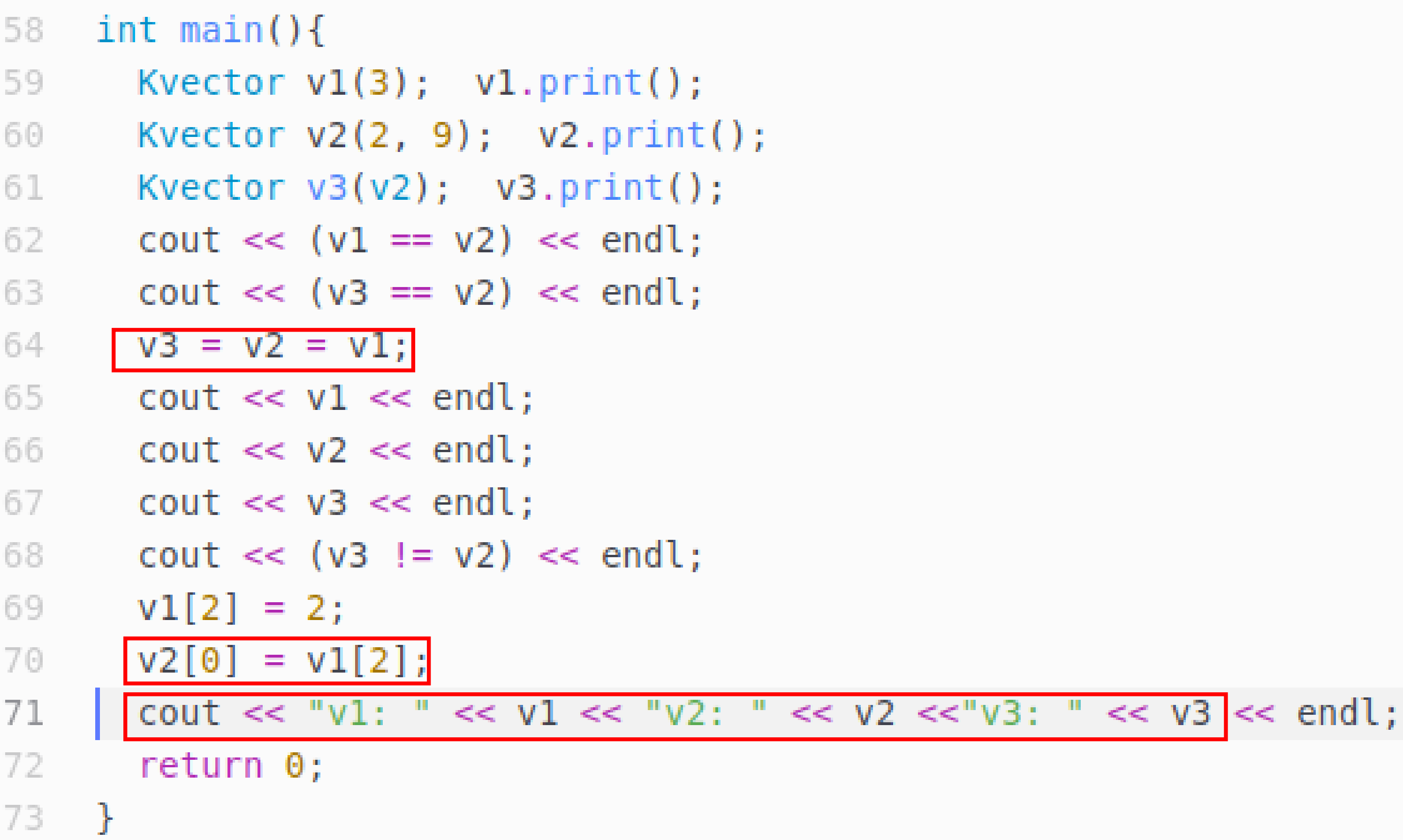
(2) 연산 결과는 0 혹은 1이다.

(3) == 연산자를 호출하여 구현해야 한다.

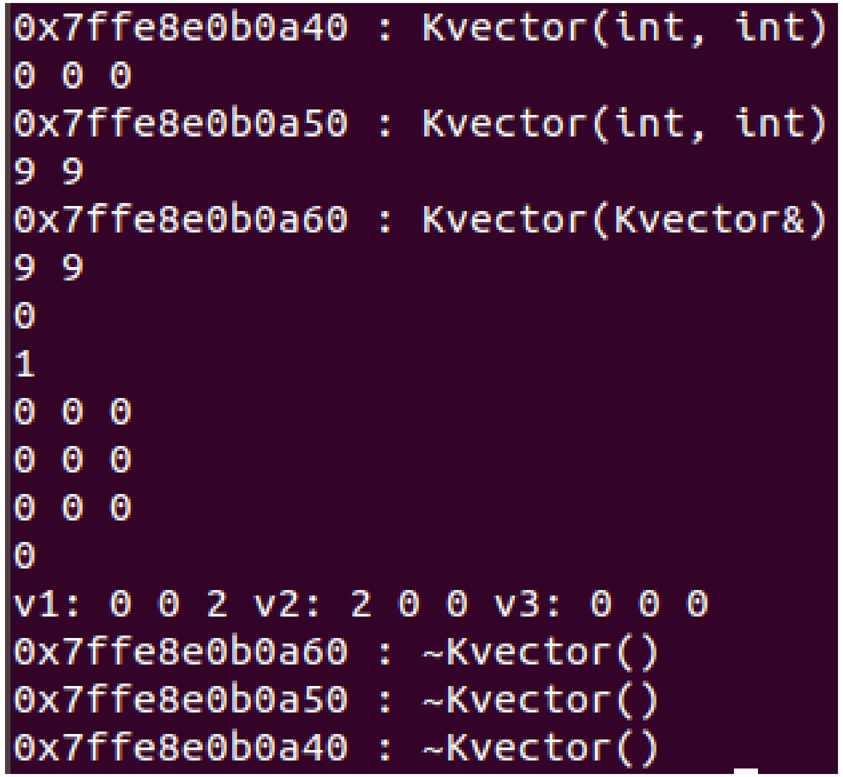
(4) v[1] = 10; 과 같이 v[1] 가 대입 연산자의 RHS 에 사용될 수 있어야 한다.

(5) cout << v << w 의 형태로 사용될 수 있어야 한다.

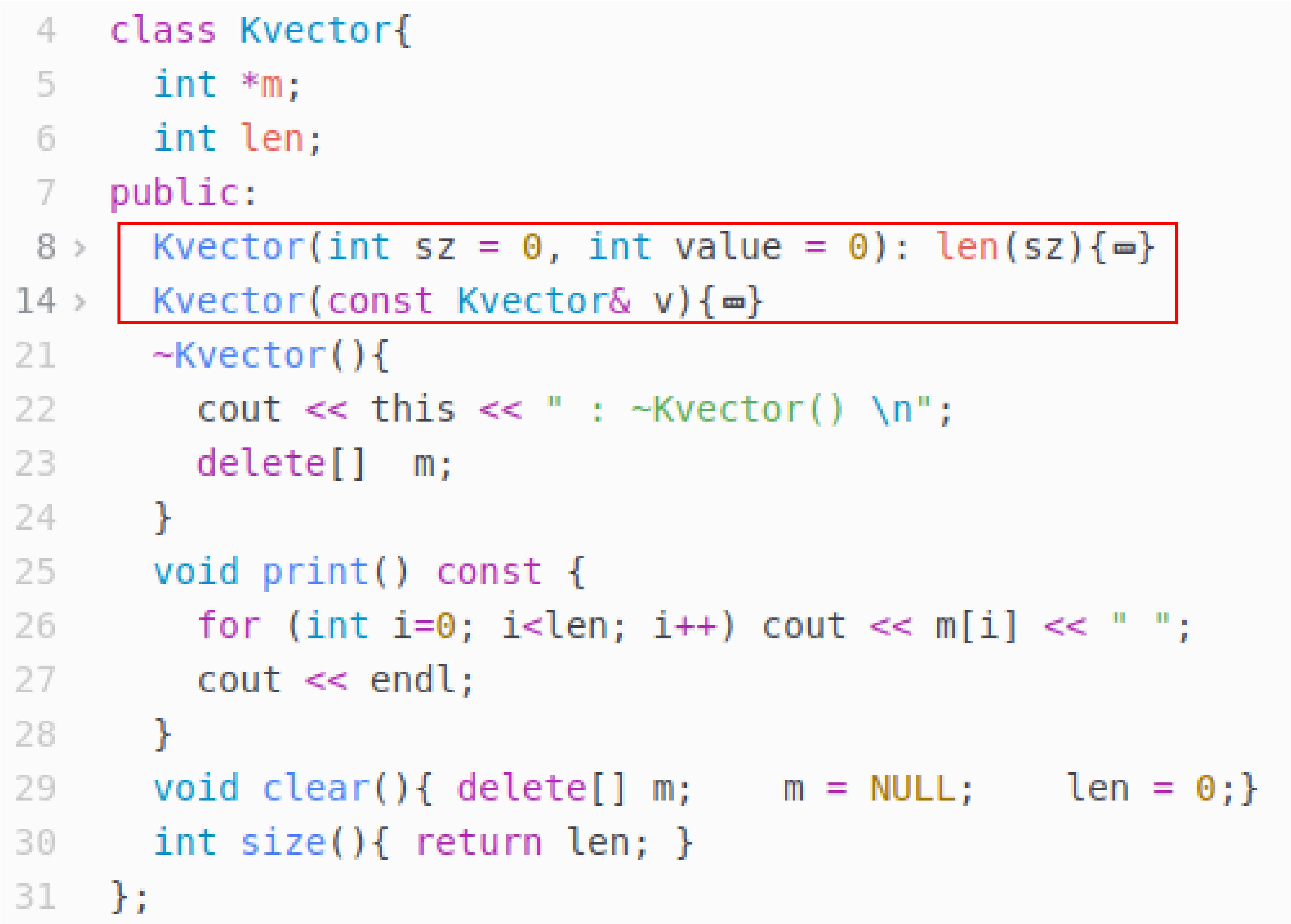
main() 함수에 대해서 출력이 다음과 같아야 한다.



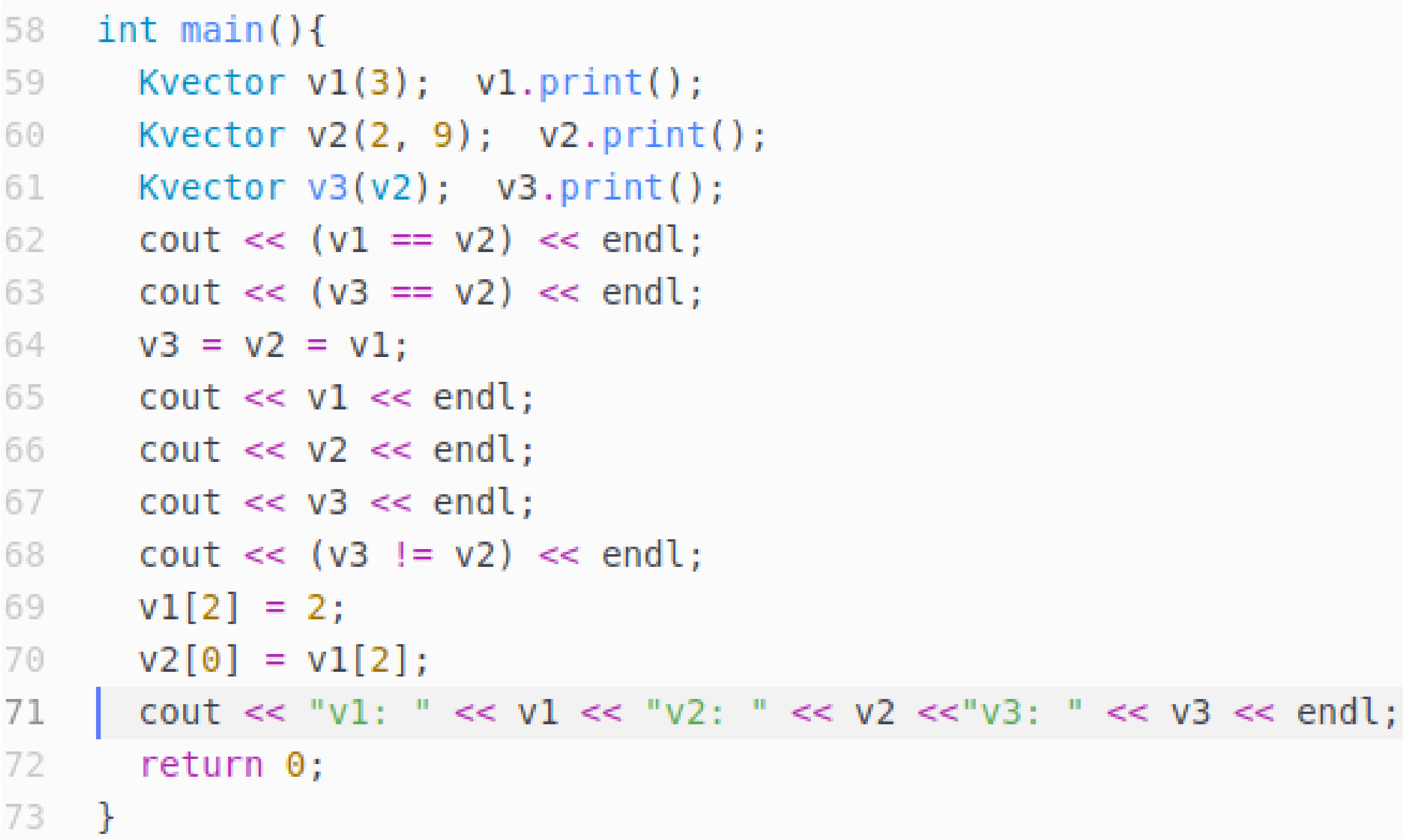
```cpp
int main(){
  Kvector v1(3);  v1.print();
  Kvector v2(2, 9);  v2.print();
  Kvector v3(v2);  v3.print();
  cout << (v1 == v2) << endl;
  cout << (v3 == v2) << endl;
  v3 = v2 = v1;
  cout << v1 << endl;
  cout << v2 << endl;
  cout << v3 << endl;
  cout << (v3 != v2) << endl;
  v1[2] = 2;
  v2[0] = v1[2];
  cout << "v1: " << v1 << "v2: " << v2 <<"v3: " << v3 << endl;
  return 0;
}
```



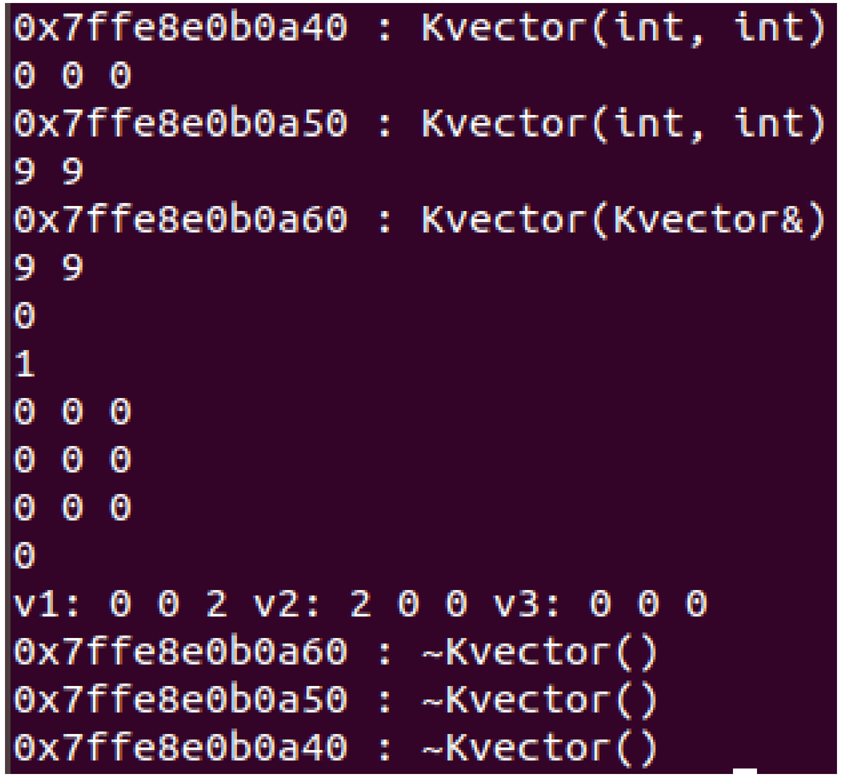
```
0x7ffe8e0b0a40 : Kvector(int, int)
0 0 0
0x7ffe8e0b0a50 : Kvector(int, int)
9 9
0x7ffe8e0b0a60 : Kvector(Kvector&)
9 9
0
1
0 0 0
0 0 0
0 0 0
0
v1: 0 0 2 v2: 2 0 0 v3: 0 0 0
0x7ffe8e0b0a60 : ~Kvector()
0x7ffe8e0b0a50 : ~Kvector()
0x7ffe8e0b0a40 : ~Kvector()
```



```cpp
class Kvector{
  int *m;
  int len;
public:
  Kvector(int sz = 0, int value = 0): len(sz){⋯}
  Kvector(const Kvector& v){⋯}
  ~Kvector(){
    cout << this << " : ~Kvector() \n";
    delete[]  m;
  }
  void print() const {
    for (int i=0; i<len; i++) cout << m[i] << " ";
    cout << endl;
  }
  void clear(){ delete[] m;    m = NULL;    len = 0;}
  int size(){ return len; }
};
```



```cpp
int main(){
  Kvector v1(3);  v1.print();
  Kvector v2(2, 9);  v2.print();
  Kvector v3(v2);  v3.print();
  cout << (v1 == v2) << endl;
  cout << (v3 == v2) << endl;
  v3 = v2 = v1;
  cout << v1 << endl;
  cout << v2 << endl;
  cout << v3 << endl;
  cout << (v3 != v2) << endl;
  v1[2] = 2;
  v2[0] = v1[2];
  cout << "v1: " << v1 << "v2: " << v2 <<"v3: " << v3 << endl;
  return 0;
}
```



```
0x7ffe8e0b0a40 : Kvector(int, int)
0 0 0
0x7ffe8e0b0a50 : Kvector(int, int)
9 9
0x7ffe8e0b0a60 : Kvector(Kvector&)
9 9
0
1
0 0 0
0 0 0
0 0 0
0
v1: 0 0 2 v2: 2 0 0 v3: 0 0 0
0x7ffe8e0b0a60 : ~Kvector()
0x7ffe8e0b0a50 : ~Kvector()
0x7ffe8e0b0a40 : ~Kvector()
```