

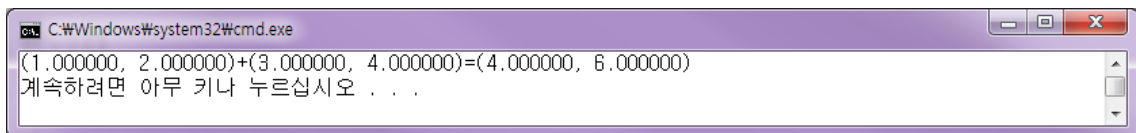
## 제10장 연산자중복과 프렌드함수

1. 연산자 중복의 개념을 이해한다.
2. 여러 가지 연산자들을 중복 정의해본다.
3. 프렌드 함수의 개념을 이해한다.
4. 프렌드 함수로 연산자를 중복 정의할 수 있다.

1

### 이번 장에서 만들어 볼 프로그램

(1) 2차원 벡터를 나타내는 클래스 MyVector에 + 연산자를 중복 정의해보자.



```
C:\Windows\system32\cmd.exe
(1.000000, 2.000000)+(3.000000, 4.000000)=(4.000000, 6.000000)
계속하려면 아무 키나 누르십시오 . . .
```

(2) 카운터를 나타내는 클래스 Counter에 ++ 연산자를 중복 정의해본다.



```
C:\Windows\system32\cmd.exe
카운터의 값: 0
카운터의 값: 1
계속하려면 아무 키나 누르십시오 . . .
```

2

### □ 연산자 함수

: 연산자를 이용하듯 호출할 수 있는 메서드. 사용자 코드에 보이는 연산자(예를 들어 '+' 연산자)가 실제로는 함수이고 사용자가 직접 그 의미를 구현하는 문법이다.

: 함수 형태로 연산자를 사용

### □ 연산자 다중 정의

: 필요에 따라 연산자 함수를 다중 정의하는 것

: 주요 연산 형태를 함수로 만든 다음 필요하면 다중 정의

: 논리 연산자는 다중 정의하면 안됨 -> 논리적 오류 유발

## operator+ 라는 이름의 함수

FirstOperationOverloading.cpp

```
class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<< '['<<xpos<< ", "<<ypos<< ']'<<endl;
    }
    Point operator+(const Point &ref) // operator+라는 이름의 함수
    {
        Point pos(xpos+ref.xpos, ypos+ref.ypos);
        return pos;
    }
};
```

```
int main(void)
{
    Point pos1(3, 4);
    Point pos2(10, 20);
    Point pos3=pos1.operator+(pos2);

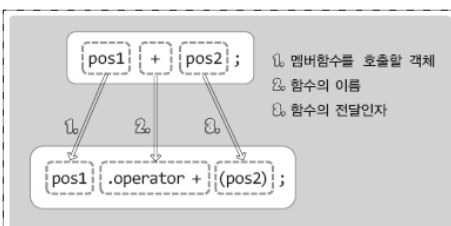
    pos1.ShowPosition();
    pos2.ShowPosition();
    pos3.ShowPosition();
    return 0;
}
```

```
int main(void)
{
    Point pos1(3, 4);
    Point pos2(10, 20);
    Point pos3=pos1+pos2;

    pos1.ShowPosition();
    pos2.ShowPosition();
    pos3.ShowPosition();
    return 0;
}
```

```
[3, 4]
[10, 20]
[13, 24]
```

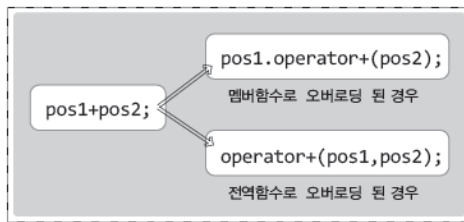
실행결과



연산자 오버로딩에서 이야기하는 함수호출의 규칙을 이해하는 것이 중요!

## 연산자를 오버로딩 하는 두 가지 방법

GFunctionOverloading.cpp



오버로딩 형태에 따라서 스스로 변환!

```
int num = 3 + 4;
Point pos3 = pos1 + pos2;
```

이렇듯 피연산자에 따라서 진행이 되는 + 연산의 형태가 달라지므로 연산자 오버로딩이라 한다.

```
class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'<<xpos<<"", "<<ypos<<'"<<endl;
    }
    friend Point operator+(const Point &pos1, const Point &pos2);
};

Point operator+(const Point &pos1, const Point &pos2)
{
    Point pos(pos1.xpos+pos2.xpos, pos1.ypos+pos2.ypos);
    return pos;
}
```

```
[3, 4]
[10, 20]
[13, 24]
```

실행결과

```
int main(void)
{
    Point pos1(3, 4);
    Point pos2(10, 20);
    Point pos3=pos1+pos2;
    pos1.ShowPosition();
    pos2.ShowPosition();
    pos3.ShowPosition();
    return 0;
}
```

5

## 오버로딩이 불가능한 연산자의 종류

.	멤버 접근 연산자
.*	멤버 포인터 연산자
::	범위 지정 연산자
?:	조건 연산자(3항 연산자)
sizeof	바이트 단위 크기 계산
typeid	RTTI 관련 연산자
static_cast	형변환 연산자
dynamic_cast	형변환 연산자
const_cast	형변환 연산자
reinterpret_cast	형변환 연산자

오버로딩 불가능!

=	대입 연산자
()	함수 호출 연산자
[]	배열 접근 연산자(인덱스 연산자)
->	멤버 접근을 위한 포인터 연산자

멤버함수의 형태로만 오버로딩 가능!

6

## 연산자를 오버로딩 하는데 있어서의 주의사항

✓ 본래의 의도를 벗어난 형태의 연산자 오버로딩은 좋지 않다!  
프로그램을 혼란스럽게 만들 수 있다.

✓ 연산자의 우선순위와 결합성은 바뀌지 않는다.  
따라서 이 둘을 고려해서 연산자를 오버로딩 해야 한다.

✓ 매개변수의 디폴트 값 설정이 불가능하다.  
매개변수의 자료형에 따라서 호출되는 함수가 결정되므로.

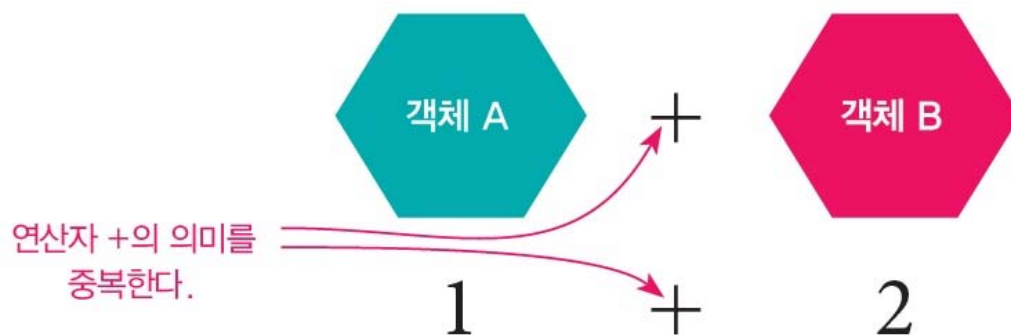
✓ 연산자의 순수 기능까지 빼앗을 수는 없다.

```
int operator+(const int num1, const int num2)
{
    return num1*num2;
}
```

정의 불가능한 형태의 함수

7

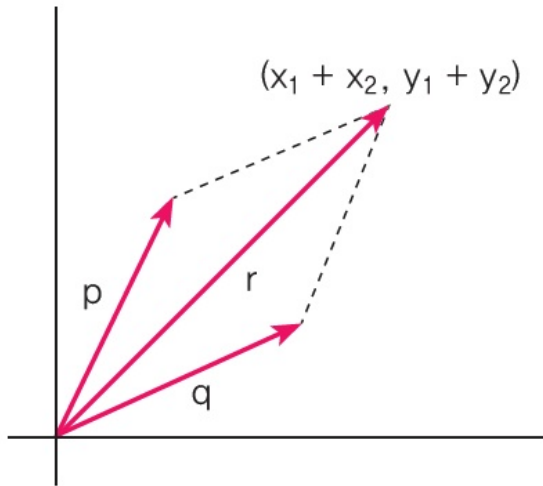
## 10.2 연산자 중복(operator overloading)



8

## 연산자 중복의 예

벡터의 경우, + 연산자로 표시하는 것이 더 직관적임



```
MyVector v1, v2, v3;
```

```
cout << (v1 + v2 + v3); // ① 연산자 중복 사용
```

```
cout << add(v1, add(v2, v3)); // ② 함수 사용
```

9

## string 클래스는 연산자 중복을 사용하고 있음

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 = "Rogue One: ";
    string s2 = "A Star Wars Story";

    string s3;
    s3 = s1 + s2;

    cout << "s1=" << s1 << endl;
    cout << "s2=" << s2 << endl;
    cout << "s1+s2= " << s3 << endl;
    cout << "s1==s2 " << boolalpha << (s1 == s2) << endl;
    return 0;
}
```

10

## 실행결과

## 중복할 수 없는 연산자

연산자	설명
::	범위 지정 연산자
.	멤버 선택 연산자
.*	멤버 포인터 연산자
?:	조건 연산자

## 10.3 연산자 중복 정의

### 문법 10.1

#### 연산자 중복

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

연산자	중복 함수 이름
+	operator+()
-	operator-()
*	operator*()
/	operator/()

13

## + 연산자의 중복

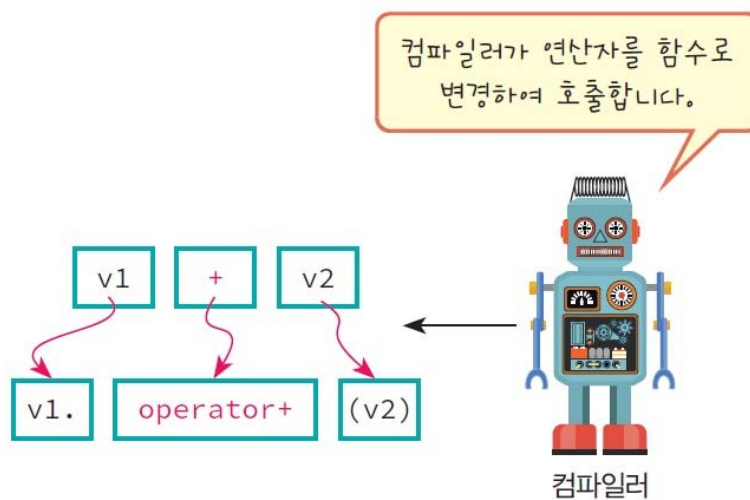


그림 10.1 멤버 함수로 연산자 중복

14

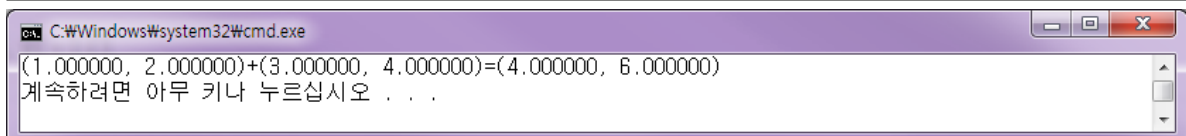
## 예제

```
#include <iostream>
#include <string>
using namespace std;
class MyVector
{
private:
    double x, y;
public:
    MyVector(double x = 0.0, double y = 0.0) : x{x}, y{y} {}
    string toString() {
        return "("+to_string(x) +", "+to_string(y)+"");
    }
    MyVector operator+(const MyVector& v2);
};
```

15

## 예제

```
MyVector MyVector::operator+(const MyVector& v2)
{
    MyVector v;
    v.x = this->x + v2.x;
    v.y = this->y + v2.y;
    return v;
}
int main()
{
    MyVector v1(1.0, 2.0), v2(3.0, 4.0);
    MyVector v3 = v1 + v2;
    cout << v1.toString() << "+"<<v2.toString() << "="<<
    v3.toString()<<endl;
    return 0;
}
```



16



## 실습

MyVector 클래스에 적절한 연산자를 추가하여 다음 연산이 가능하도록 하라.

```
MyVector a(0, 0), b(3, 3), c(4, 4);  
a = b - 10;  
a = 20 - c;
```

17

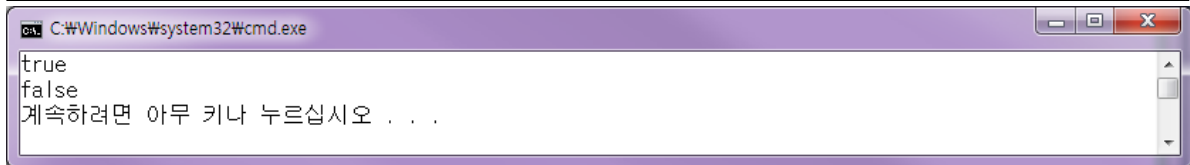
## 10.4 == 연산자의 중복

```
#include <iostream>  
using namespace std;  
class Time  
{  
    int hour, min, sec;  
public:  
    Time(int h=0, int m=0, int s=0) : hour(h), min(m), sec(s) { }  
    bool operator==(Time &t2) {  
        return (hour == t2.hour &&  
                min == t2.min &&  
                sec == t2.sec);  
    }  
    bool operator!=(Time &t2) {  
        return !(*this == t2);  
    }  
};
```

18

## == 연산자의 중복

```
int main()
{
    Time t1(1, 2, 3), t2(1, 2, 3);
    // 참과 거짓을 1, 0이 아니라 true, false로 출력하도록 설정한다.
    cout.setf(cout.boolalpha);
    cout << (t1 == t2) << endl;
    cout << (t1 != t2) << endl;
    return 0;
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the output of the program: "true" on the first line and "false" on the second line. Below the output, there is a Korean message: "계속하려면 아무 키나 누르십시오 . . .".

19

## 10.5 ++ 연산자의 중복

```
#include <iostream>
using namespace std;
class Counter {
private:
    int value;
public:
    Counter() : value{0} {};
    ~Counter() {}
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
    Counter& operator++()
    {
        ++value;
        return *this;
    }
};
```

20

## ++ 연산자의 중복

```
int main()
{
    Counter c;
    cout << "카운터의 값: " << c.getValue() << endl;
    ++c;
    cout << "카운터의 값: " << c.getValue() << endl;

    return 0;
}
```



21

## 후위 연산자 ++의 중복

연산자	중복 함수 이름
++c	c.operator++()
--c	c.operator--()

```
const Counter operator++(int i)
{
    Counter temp={*this};    // 현재의 상태를 저장한다.
    ++value;
    return temp;
}
```

22

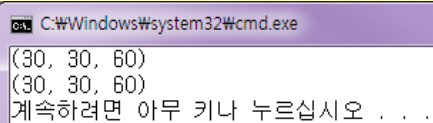
- **Counter** 클래스에 --연산자를 중복 정의하라. -- 연산자는 카운터의 값을 1 감소시키는 것으로 하라.

## 10.6 대입연산자 중복

```
#include <iostream>
using namespace std;
class Box
{
private: double length, width, height;
public:  Box(double l = 0.0, double w = 0.0, double h = 0.0)
        : length{l}, width{w}, height{h} {   }
        void display() {
cout << "(" << length << ", " << width << ", " << height << ")" << endl;
        }
        Box& operator=(const Box& b2)   {
            this->length = b2.length;
            this->width = b2.width;
            this->height = b2.height;
            return *this;
        }
};
```

## 대입연산자 중복

```
int main()
{
    Box b1(30.0, 30.0, 60.0), b2;
    b1.display();
    b2 = b1;
    b2.display();
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
(30, 30, 60)
(30, 30, 60)
계속하려면 아무 키나 누르십시오 . . .
```

25

대입 연산자는 복사 생성자처럼 깊은 복사, 얕은 복사 문제가 있다.

```
class CMyData
{
    // (1) a=a; -> 원인 찾기, 해결 실습
public:    // (2) c(4); a=b=c; -> 해결 실습
    ...
    void operator=(const CMyData &rhs)
    {
        // 본래 가리키던 메모리를 삭제하고
        delete m_pnData;
        // 새로 할당한 메모리에 값을 저장한다.
        m_pnData = new int(*rhs.m_pnData);
    }
private:
    int *m_pnData = nullptr;
};

int main( )
{
    CMyData a(0), b(5);
    a = b;
    cout << a << endl;
    return 0;
}
```

26

a = a; 와 같은 코드나 a = b = c; 같은 코드도 고려해야 한다.

```
class CMyData
{
public:
    ...
    CMyData& operator=(const CMyData &rhs)
    {
        cout << "operator=" << endl;
        if (this == &rhs)
            return *this;
        delete m_pnData;
        m_pnData = new int(*rhs.m_pnData);
        return *this;
    }
};

int main( )
{
    CMyData a(0), b(3), c(4);
    a = b = c;
    return 0;
}
```

27

## 실습 복합대입 연산자

```
// CMyData 클래스에 += 연산자 함수를 추가하여 다음을 수행하라.
CMyData a(0), b(5), c(10);
a += b;
a += c;
cout << a << endl;
```

28

## 이동 대입 연산자 (이동 시맨틱)

임시 객체가 r-value인 단순 대입 연산을 고려해야 한다.

```
class CMyData // 아래부분 추가하여 실행
{
public:
    ...
    CMyData& operator=(CMyData &&rhs)
    {
        cout << "operator = (Move)" << endl;
        // 얕은 복사를 수행하고 원본은 NULL로 초기화한다.
        m_pnData = rhs.m_pnData;
        rhs.m_pnData = NULL;
        return *this;
    }
    ...
};
int main( )
{
    CMyData a(0), b(3), c(4);
    a = b + c;
    return 0;
}
```

29

## 10.7 인덱스 연산자 [ ]의 중복

```
#include <iostream>
using namespace std;
const int SIZE = 10;
class MyArray {
private: int a[SIZE];
public: MyArray() {
        for (int i = 0; i < SIZE; i++)
            a[i] = 0;
    }
    int &operator[](int i) {
        if (i >= SIZE) {
            cout << "잘못된 인덱스:" ;
            return a[0];
        }
        return a[i];
    }
};
```

30

## 인덱스 연산자 [ ]의 중복

```
int main() {  
    MyArray A;  
  
    A[3] = 9;  
    cout << "A[3]= " << A[3] << endl;  
    cout << "A[16]= " << A[16] << endl;  
  
    return 0;  
}
```



C:\Windows\system32\cmd.exe  
A[3]= 9  
잘못된 인덱스: A[16]= 0  
계속하려면 아무 키나 누르십시오 . . .

31

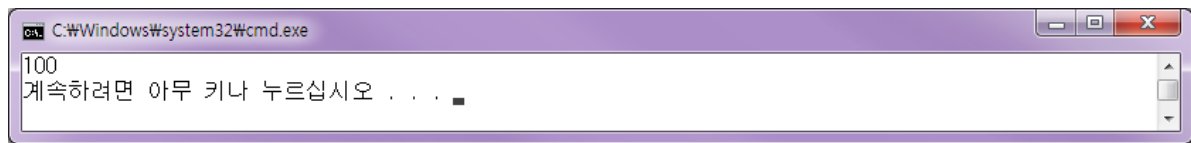
## 포인터 연산자의 중복

```
#include <iostream>  
using namespace std;  
class Pointer {  
    int *p;  
public:  
    Pointer(int *p) : p{p} {}  
    ~Pointer() { delete p; }  
    int* operator->() const { return p; }  
    int& operator*() const { return *p; }  
};  
int main()  
{  
    Pointer p(new int);  
    *p = 100;  
    cout << *p << endl;  
    return 0;  
}
```

32



## 실행결과



33

## 10.9 프렌드 메커니즘

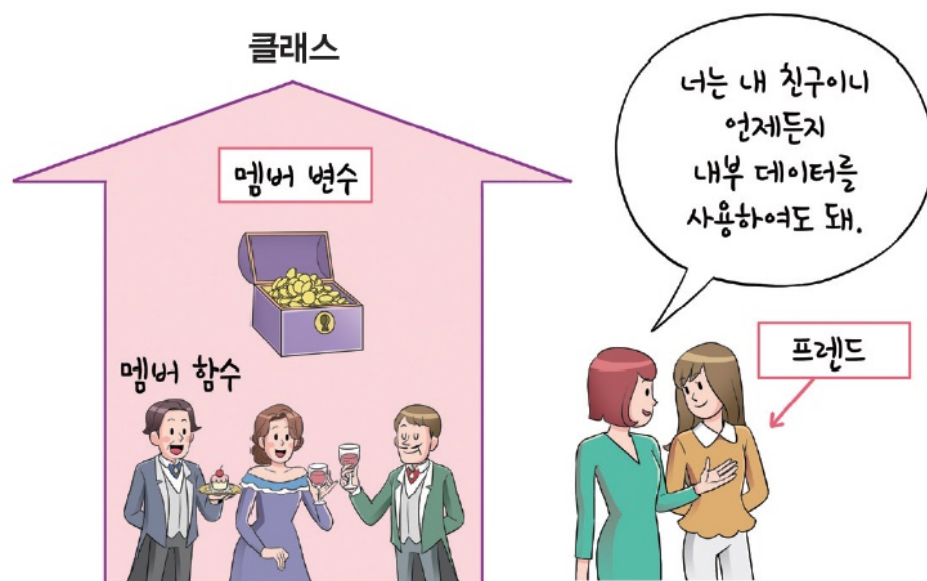


그림 10.2 프렌드의 개념

34

## 프렌드 함수

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

class Box {
    double length, width, height;
public:
    Box(double l, double w, double h) : length{l}, width{w}, height{h} {}
    friend void printBox(Box box);
};

void printBox(Box box) {
    cout << "Box( " << box.length << ", " << box.width << ", "
    << box.height<<") " << endl;
}
```

35

## 프렌드 함수

```
int main() {
    Box box(10, 20, 30);
    printBox(box);

    return 0;
}
```



36

## 프렌드 클래스

```
class A {
public:
    friend class B;    // B는 A의 프렌드가 된다.
    A(string s = "") : secret{s} { }
private:
    string secret;    // B는 여기에 접근할 수 있다.
};

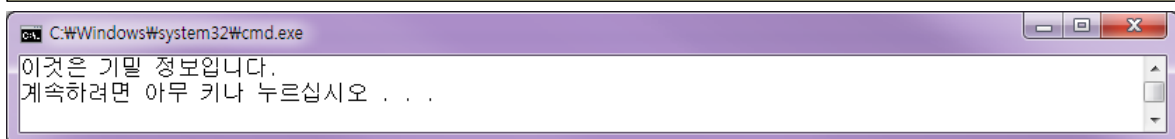
class B {
public:
    B() { }
    void print(A obj) {
        cout << obj.secret << endl;
    }
};
```

37

## 프렌드 클래스

```
int main()
{
    A a("이것은 기밀 정보입니다.");
    B b;
    b.print(a);

    return 0;
}
```



38

## 프렌드의 용도

- 프렌드 함수는 두개의 객체를 비교할 때 많이 사용된다.

```
class Date
{
    int year, month, day;
public:
    Date(int y=0, int m=0, int d=0) : year(y), month(m), day(d) { }
    bool equals(Date obj) {
        return year == obj.year && month == obj.month && day
        == obj.day;
    }
};

int main() {
    Date d1(1960, 5, 23), d2(2002, 7, 23);
    if( d1.equals(d2) == true ) {
        ...
    }
}
```

39

## 프렌드를 사용하면

```
#include <iostream>
using namespace std;
class Date
{
    friend bool equals(Date d1, Date d2);
    int year, month, day;
public:
    Date(int y=0, int m=0, int d=0) : year(y), month(m), day(d) { }
};
// 프렌드 함수
bool equals(Date d1, Date d2)
{
    return d1.year == d2.year && d1.month == d2.month && d1.day == d2.day;
}
```

40

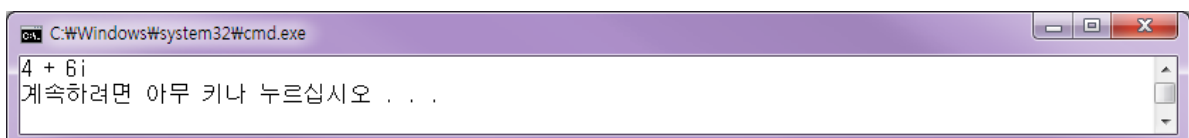
## 예제

```
#include <iostream>
using namespace std;
class Complex {
public:
    friend Complex add(Complex, Complex);
    Complex(double r=0.0, double i=0.0) { re = r; im = i; }
    void print() {
        cout << re << " + " << im << "i" << endl;
    }
private:
    double re, im;
};
```

41

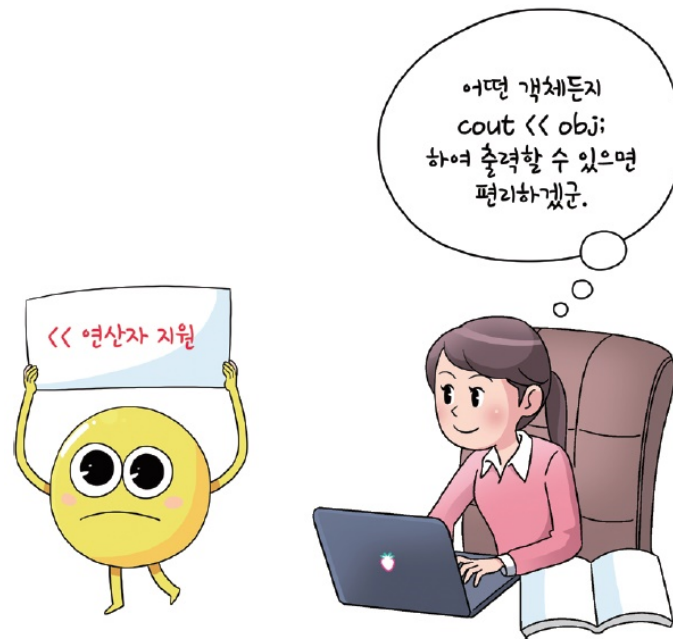
## 예제

```
Complex add(Complex a1, Complex a2)
{
    return Complex(a1.re + a2.re, a1.im + a2.im);
}
int main()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = add(c1, c2);
    c3.print();
    return 0;
}
```



42

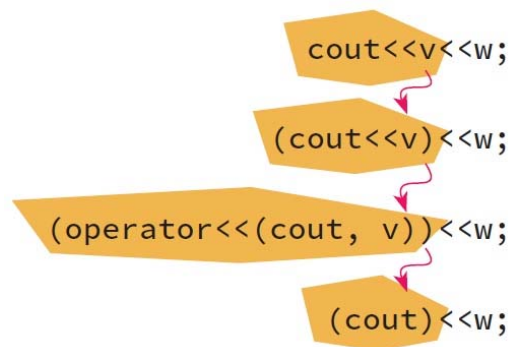
## 10.10 << 연산자의 중복 정의



43

## 중복정의 형태

```
friend ostream& operator<<(ostream& os, const MyVector& v)
{
    ...
}
```



44

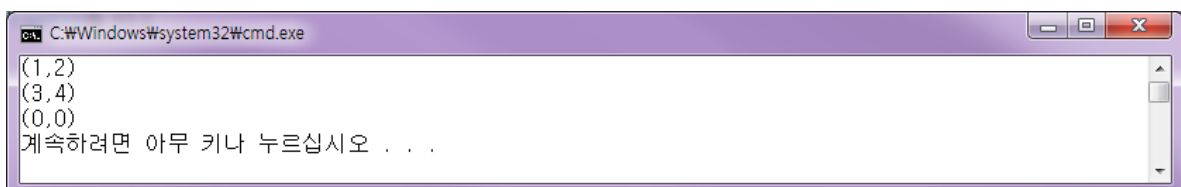
## 예제

```
#include <iostream>
using namespace std;
class MyVector
{
private:
    double x, y;
public:
    MyVector(double xvalue = 0.0, double yvalue = 0.0) : x(xvalue),
y(yvalue) {      }
    friend ostream& operator<<(ostream& os, const MyVector& v){
        os << "(" << v.x << "," << v.y << ")" << endl;
        return os;
    }
};
```

45

## 예제

```
int main()
{
    MyVector v1(1.0, 2.0), v2(3.0, 4.0), v3;
    cout << v1 << v2 << v3;
    return 0;
}
```



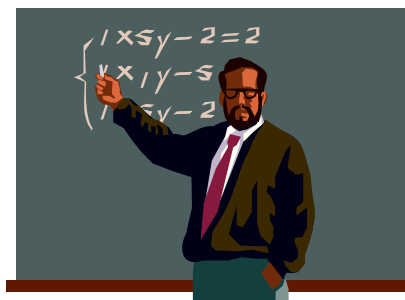
46

## 연산자 중복 시에 유의할 점

- 새로운 연산자를 만드는 것은 허용되지 않는다. 예를 들어서 지수승을 나타내기 위하여  $\wedge$  연산자를 새롭게 정의할 수 없다.
- 거의 모든 연산자가 중복이 가능하다. 하지만  $::$  연산자,  $*$  연산자,  $.$  연산자,  $?:$  연산자는 중복이 불가능하다.
- 연산자들의 우선순위나 결합 법칙은 변경되지 않는다.
- 만약  $+$  연산자를 중복하였다면 일관성을 위하여  $-$ ,  $+=$ ,  $-=$  연산자도 중복하는 것이 좋다.

47

## Q & A



48