

## = 연산자 중복 (멤버함수로만 가능, 표 10-2) 대입 연산자(할당 연산자)

### 복사 생성자의 호출

```
int main(void)
{
    Point pos1(5, 7);
    Point pos2=pos1;
    . . .
}
```

### 대입 연산자의 호출

```
int main(void)
{
    Point pos1(5, 7);
    Point pos2(9, 10);
    pos2=pos1;
    . . .
}
```

- 멤버변수가 포인터인 경우 작성 필요
- 복사 생성자 : 함수 이름이 클래스 이름과 동일
- 대입 연산자 : 함수 이름은 operator=

pos2.operator=(pos1);

Point 클래스의 디폴트 대입 연산자(자동 생성)

```
Class Point{   복사 생성자
int x, y;
Public:
Point(const Point &a) {
    x =a.x;  y = a.y;
}
}
```

```
Point& operator=(const Point& a){
    x = a.x;
    y = a.y;
    return *this;
}  멤버 대 멤버의 단순 복사를 진행하는 디폴트
대입연산자
```



## Vector 클래스 = 연산자 중복(멤버함수만 가능)

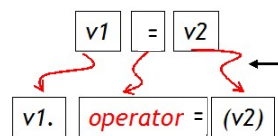
- "=" 연산자의 operator=(...) 멤버 함수 구현

- Vector v1(2.0, .0), v2, v3;

- v3 = v2 = v1; ← (v3 = (v2 = v1));
  - v2 = v1; 수행 후 → v3 = v2; 수행
  - v2 = v1; 수행하면 반환 값은 v2

- 따라서

- v2 = v1; 는 v2.operator=(v1); 와 같이 호출
- 함수 operator= 의 반환값은 v2(호출한 객체, \*this)가 되어야 함.
- 함수에서 호출한 객체(\*this)를 참조자로 반환해야함.



## Vector 클래스 = 연산자 중복(멤버함수만 가능)

```
class Vector
{
...
    Vector& operator=(const Vector& v2)
    {
        this->x = v2.x;
        this->y = v2.y;
        return *this;
    }
...
};
```

*this->x 대신 x 해도 무관*

*(참조자)*

*주의: 반드시 현재 객체의 레퍼런스를 반환*

*this : 자신 지칭 포인터  
\*this : 자신*

```
Vector v1(2.0, 3.0);
v3 = v2 = v1; // 가능!
```

*v2.operator=(v1);  
v3.operator=(v2);*



## 얕은 대입 문제(대입 연산자 미작성시 문제점)

- 동적 할당 공간(포인터 변수)이 있으면 반드시 = 연산자를 중복 정의해야 함

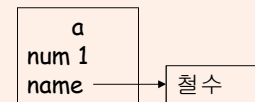
student.cpp

```
#include <iostream>
using namespace std;
class Student {
    char *name; // 이름
    int number;
public:
    Student(char *p, int n) {
        cout << "메모리 할당" << endl;
        name = new char[strlen(p)+1];
        strcpy(name, p);
        number = n;
    }
    ~Student() {
        cout << "메모리 소멸" << endl;
        delete [] name;
    }
};
```

대입연산자를 정의하지 않으면 이전에 공부한 디폴트 복사 생성자의 문제점과 동일한 문제점 발생 !

→ 포인터 변수 있으면 대입연산자/소멸자/복사생성자 작성 필요

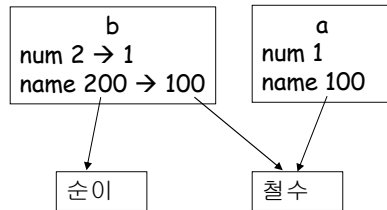
Student a("철수", 1);



## 얕은 대입 문제(대입 연산자 미작성시)

- 멤버에 동적 할당 공간(포인터)이 있으면 반드시 = 연산자를 중복 정의하여야 함

```
Student a("철수", 1), b("순이", 2);
b = a;           // b.operator=(a);
return 0;        // 소멸자에서 오류 발생
```



```
// 자동 생성된 디폴트 대입연산자, → 단순 복사
Student& operator=(const Student &a){
    name = a.name;    // 같은 장소 지정
    number = a.number;
    return *this;
}
```



## 얕은 대입 문제(대입 연산자 작성시)

- 멤버에 동적 할당 공간(포인터)이 있으면 반드시 = 연산자를 중복 정의하여야 함

```
class Student {
```

```
...
```

```
public:
```

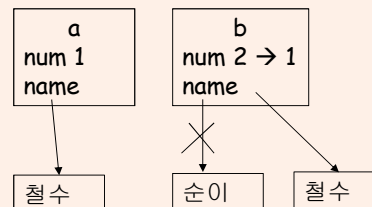
```
...
```

```
Student& operator=(const Student& s)
{
    delete [] name;
    name = new char[strlen(s.name)+1];
    strcpy(name, s.name);
    number = s.number;
    return *this;
}
};
```

```
Student a("철수", 1), b("순이", 2);
b = a; // b.operator=(a);
```

```
// b.name 공간 지우고 새로 할당
```

= 연산자 함수 정의



## Report 10\_1

- 벡터를 나타내는 Vector 클래스에 다음 연산자들을 구성 (oop\_학번\_101.txt 제출)

```
p10_1() {
    Vector a(1, 2), b(2, 3), c;
    c = a - b;           // c = a - b; 에서 a 값 변화 없음
    cout << a << " , " << b << " , " << c;

    a += b;              // a = a+b 수행, a를 반환 → operator+= 함수 작성
    cout << a << " , " << b << " , " << c;

    a -= c;              // a = a-c 수행, a를 반환 → operator-= 함수 작성
    cout << a << " , " << b << " , " << c;

    return 0;
}
```

### 전위/후위의 문제

참고) C 에서 전위, 후위 증감 연산자 반환

// C 코드

```
int main() {
    int a = 10, b;
    b = ++a;
    cout << a << " " << b << endl; // ... (1)

    a = 10;
    b = ++(++a);
    cout << a << " " << b << endl; // ... (2)

    a = 10;
    b = a++;
    cout << a << " " << b << endl; // ... (3)

    a = 10;
    cout << (a++)++ << endl; // error

    return 0;
}
```

(1) 11 11

- ++a 함수 반환값은 변수 a 자신(저장공간)
- ++a → 증가된 변수 a 반환

(2) 12 12

(3) 11 10

- a++ 함수 반환값은 변수 a의 저장 값 (상수, **const**)
- a 저장 값을 반환하고 변수 a를 증가
- a++ → 10(값) 반환됨
- 참고) (a++)++ 안되는 이유도 (a++)는 값(상수) 이기 때문에

< 객체지향에서도 >

- ++a 함수 반환값은 a 자신 (변수, 객체)
- a++ 함수 반환값은 값 (상수) <- **const 반환**

## 전위/후위의 문제 참고) C 에서 전위, 후위 증감 연산자 반환

```
// C 코드
int main() {
    int a = 5;

    ++a = 10;          //
    cout << a << endl; // (1)

    a = 5;
    a++ = 20;          // (2)
    cout << a << endl;

    return 0;
}
```

- (1) 10 출력 ... 가능 이유
- ++a 함수 반환값은 변수 a 자신(저장공간)
  - ① ++a → 변수 a(=6) 반환됨
  - ② a = 10; 으로 해석 → ok
- (2) error ... 오류 이유
- a++ 함수 반환값은 변수 a의 저장 값 (상수, **const**)
  - ① a++ → 5(값) 반환됨
  - ② 5 = 20; 으로 해석 → error
  - 참고) (a++)++ 안되는 이유도 (a++)는 값(상수)이기 때문에
- < 결론 >
- ++a 함수 반환값은 a 자신 (변수, 객체)
  - a++ 함수 반환값은 a의 저장 값 (상수) ← const 반환

## 증가/감소 연산자의 중복

- ++와 -- 연산자의 중복

연산자	중복 함수 이름
++V	<u>v.operator++()</u>
--V	<u>v.operator--()</u>

- v(1, 2) → ++v → v(2, 3)
- 교재에서는 멤버함수로만 구현      cf) friend 함수로도 구현 가능
- ++v → 멤버함수 operator++() 로 구현



## 증가/감소 연산자의 중복

- ++ 연산자를 멤버함수로 구현 :  $++x \rightarrow x.operator++()$
- ++ 를 멤버 함수로 구현시 `operator++()`
  - 함수 `operator++` 의 반환값은 호출한 객체(\*this) 자신. (--x 연산도 동일)
  - 호출한 객체(\*this)를 참조자로 반환해야함.

```
Vector& operator++()  
{  
    x = x + 1.0;  
    y = y + 1.0;  
    return *this;  
}
```

## 증가/감소 연산자의 중복

vector.cpp

```
#include <iostream>  
using namespace std;  
class Vector  
{  
private:  
    double x, y;  
public:  
    Vector(double xvalue=0.0, double yvalue=0.0) : x(xvalue), y(yvalue){ }  
    void display(){  
        cout << "(" << x << ", " << y << ")" << endl;  
    }  
    Vector& operator++()  
    {  
        x = x + 1.0;  
        y = y + 1.0;  
        return *this;  
    }  
};
```

$++v \rightarrow v.operator++()$

++ 연산자 함수 정의



## 증가/감소 연산자의 중복

```
int main()
{
    Vector v;
    ++v;
    v.display();
    ++(++v);
    v.display();

    return 0;
}
```

실행결과

(1, 1)  
(3, 3)



## 전위/후위의 문제

- 전위(++v) 연산자를 멤버함수 구현 → operator++( )
- 후위(v++) 연산자를 멤버함수 구현 → operator++( )
- 전위와 후위 연산자를 구별하기 위하여 ++가 피연산자 뒤에 오는(후위) 경우를 처리 함수는 int형 매개 변수를 추가한다.

연산자	중복 함수 이름
++V	<u>v.operator++()</u> 구별 위한 단순 표기
V++	<u>v.operator++(int)</u> 의미 없음(인자 전달 불필요)



## 전위/후위의 문제 정리

- ++x, x++ 차이

```
x = 1;
cout << ++x;
```

```
x = 1;
cout << x++;
```

- ++x 는 → x 자신의 값 증가 후, x 자신 return → 2 출력
- x++ 는 x의 값(상수) 반환하고 x 증가 → 1 출력
- ++x 반환값은 변수 x,
- x++ 반환값은 증가하기 전의 x 저장 값 → const
- ++v, ++(++v) 는 모두 사용 가능,
- v++ 은 사용가능, 하지만 (v++)++ 는 사용 불가



## 전위/후위의 문제

- ++x 반환값은 호출한 객체 x 자신,
- x++ 반환값은 증가하기 전 호출한 객체 x의 값(const)
- ++x 전위 함수 구현시 operator++()
- 함수 operator++ 는 호출한 객체(\*this)를 참조자로 반환. → Vector& 로 반환

```
Vector& operator++()
{
    x++;
    y++;
    return *this;
}
```

```
const Vector operator++(int)
{
    Vector saveObj = *this;
    x++;
    y++;
    return saveObj;
}
```

- x++ 후위 함수 구현시 operator++(int)
- 함수 operator++(int)의 반환값은 호출한 객체(\*this)의 증가하기 전 값을 반환.
- → const Vector 로 반환





## 전위/후위의 문제

v++ 반환값은 값, const

```
Vector& operator++()
{
    x++;
    y++;
    return *this;
}
```

++v 형태의 증가 연산자  
함수 정의

자신(호출한 객체)의 값 증가 후  
변화된 자신 return

```
const Vector operator++(int)
{
    Vector saveObj = *this;
    x++;
    y++;
    return saveObj;
};
```

v++ 형태의 증가 연산자  
함수 정의

자신(호출한 객체)의 값(상수) 반환하고 자신  
증가

- 증가하기 전 자신(호출한 객체)의 복사본  
만들고 복사본(변경전 객체)을 return
- 이후 자신의 값 증가

cout << v++; → cout << v.operator++(int);



## [ ] 연산자의 중복(멤버함수로만 가능, 표 10-2)

- 인덱스 연산자의 중복 - C# 에서 인덱서
  - 객체의 멤버변수 배열을 객체 배열로 취급

연산자	중복 함수 이름
v[]	v.operator[](int)

```
MyArray A;
A[3] = 10;
A.operator[](3) = 10;
```

- MyArray 클래스의 멤버변수 배열 data[ ] 존재
- "A.data[3] = 2;" 대신에 "A[3]=2" 사용 위한 것.
- 멤버 함수 작성시 배열 인덱싱 오류 A[-2] 등을 대비 가능.

A[3] → A.operator[](3)

→ A[3] 하면 멤버함수 operator[] 함수의 인자로 배열 인덱스 3 전달

A[3] = 10; → A.data[3] = 10 과 동일

- A[3] 하면 A.data[3] 저장공간이 반환되어 A.data[3] 공간에 10저장
- 값이 아니라 A.data[3] 자체(저장공간) 반환 → operator[] 함수에서  
참조자로 반환




## 인덱스 연산자 중복

```
my_array.cpp
#include <iostream>
#include <assert.h> // assert() 함수 사용으로 필요
using namespace std;

// 항상된 배열을 나타낸다.
class MyArray {
    friend ostream& operator<<(ostream &, const MyArray &); // 출력 연산자 <<
private:
    int *data; // 배열의 데이터
    int size; // 배열의 크기
public:
    MyArray(int size = 10); // 디폴트 생성자
    ~MyArray(); // 소멸자

    int getSize() const; // 배열의 크기를 반환
    MyArray& operator=(const MyArray &a); // = 연산자 중복 정의
    int& operator[](int i); // [] 연산자 중복: 설정자
};
```




## 인덱스 연산자 중복(생성자, 소멸자)

```
MyArray::MyArray(int s) {
    size = (s > 0 ? s : 10); // 디폴트 크기를 10으로 한다.
    data = new int[size]; // 동적 메모리 할당

    for (int i = 0; i < size; i++)
        data[i] = 0; // 요소들의 초기화
}

MyArray::~MyArray() {
    delete [] data; // 동적 메모리 반납
    data = NULL;
}
```



## 인덱스 연산자 중복(대입연산자)

`a = a;` 인 경우 바로 `return`, `b=a` 는 수행

```
MyArray& MyArray::operator=(const MyArray& a){
    if(&a != this) { // 자기 자신인지 체크
        delete [] data; // 호출 객체가 기존에 있던 배열 삭제
        size = a.size; // 인자 a의 배열 크기
        data = new int[size];
        for(int i=0; i<size; i++) // a 멤버변수 값들을 호출한 객체에 대입
            data[i] = a.data[i];
    }
    return *this; // a = b = c; 경우에서
                // b=c 하면 b return;
}
```

- `&a != this` ← `this` 는 주소,
- `&a` 도 주소(참조자 아님) → 참조자는 객체 선언하며 사용



## 인덱스 연산자 중복(인덱스 연산자)

```
int MyArray::getSize() const
{
    return size;
}
```

- **Assert(조건)**: 디버깅용 함수, 조건이 거짓이면 메시지 창이 발생, 오류 위치 알려줌, 참이면 지나감
- **assert.h** 헤더파일 필요

인덱스 연산자 정의

```
int& MyArray::operator[](int index) {
    assert(0 <= index && index < size);
    return data[index];
}
```

- `A[3] → A.operator[](3) → A.data[3]` 을 return
- 인자 는 멤버 배열 **index**
- 메모리 공간 자체를 return 해야 하기에 참조자 반환

// 프렌드 함수 정의

```
ostream& operator<<(ostream &output, const MyArray &a) {
    int i;
    for (i = 0; i < a.size; i++) {
        output << a.data[i] << ' ';
    }
    output << endl;
    return output;
}
```

// `cout << a1 << a2 << a3`와 같은 경우 대비



## 인덱스 연산자 중복

```
int main()
{
    MyArray a1(10);

    a1[0] = 1;
    a1[1] = 2;
    a1[2] = 3;
    a1[3] = 4;
    cout << a1 ;

    return 0;
}
```

- MyArray 클래스의 멤버변수 배열 data[ ] 존재
  - 멤버변수 배열 이름은 임의 사용 가능
- "A.data[3] = 2;" 대신에 "A[3]=2" 사용 위한 것.

a1[0] → a1.operator[](0) 호출,  
반환값 a1.data[0] → 결국 a1.data[0] = 1; 이 됨

실행결과

```
{ 2 3 4 0 0 0 0 0 0 0 }
```



## Report 10\_3(학번\_103.txt)

- 10장 연습문제 Programming 1번(426쪽)
  - == 연산자는 두 객체 멤버 배열 공간 크기 다르면 false

```
int main() {
    Array a1(10), a2(10), a3(10);

    a1[0] = 1;      a1[1] = 2;      a1[2] = 3;      a1[3] = 4;
    a2[0] = 1;      a2[1] = 2;      a2[2] = 3;      a2[3] = 4;
    a3 = a1;        a3[3] = 5;

    cout << " a1 배열은 : " << a1 << endl; // 1 2 3 4 0 0 ...
    cout << " a2 배열은 : " << a2 << endl; // 1 2 3 4 0 0 ...
    cout << " a3 배열은 : " << a3 << endl; // 1 2 3 5 0 0 ...

    cout << " a1 == a2 을 중복 정의 : " << (a1 == a2) << endl; // 1
    cout << " a1 != a3 을 중복 정의 : " << (a1 != a3) << endl; // 1
    cout << " a3 = a1 을 중복 정의 : " << (a3 = a1) << endl; // 1 2 3 4 0 0 ...
    return 0;
}
```

## Report 10\_3



```
class Array {  
    int *data;    // 저장 공간  
    int size;     // data 배열 크기 저장  
public:  
    Array(int size = 10);  
    ~Array();  
    int getSize() const;  
    friend ostream& operator<<(ostream &, const Array &);  
  
    ....  
};
```

## 포인터 연산자의 중복(생략)

- 간접 참조 연산자 \*와 멤버 연산자 ->의 중복 정의

연산자	중복 함수 이름
*	operator*()
->	operator->()

참고) Car 클래스 멤버는 speed

```
Car a(2), *p;          // a.speed = 2;
```

```
p = &a;
```

```
cout << a.speed << p->speed << (*p).speed;    // 2 출력
```

## 함수 호출 연산자 ()의 중복

함수 호출때 사용하는 () 도 중복이 가능하다

연산자	중복 함수 이름
<code>f(...)</code>	<code>f.operator()(...)</code>

- 객체를 함수처럼 사용하게 하는 연산자

```
Car a(1, 10, "red"); // gear, speed, color 멤버 변수 초기화
a(3, 50);            // Car 클래스 멤버함수 operator(...) 함수 호출
                    // gear, speed 를 각각 3, 50으로 지정하는 함수
                    // 객체를 함수 처럼 사용
```

## 함수 호출 연산자 ()의 중복

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

```
class Negate {
public:
    int operator()(int value) {
        return -value;
    }
};
```

```
int main()
{
    Negate neg;
    cout << neg(100) << endl;
}
```

함수 호출 연산자 중복  
정의

실행결과

-100

## 타입 변환(이후 생략)

클래스의 객체들도 하나의 타입에서 다른 타입으로 자동적인 변환이 가능하다.

이것은 변환 생성자(**conversion constructor**)와 변환 연산자(**conversion operator**)에 의하여 가능하다.

