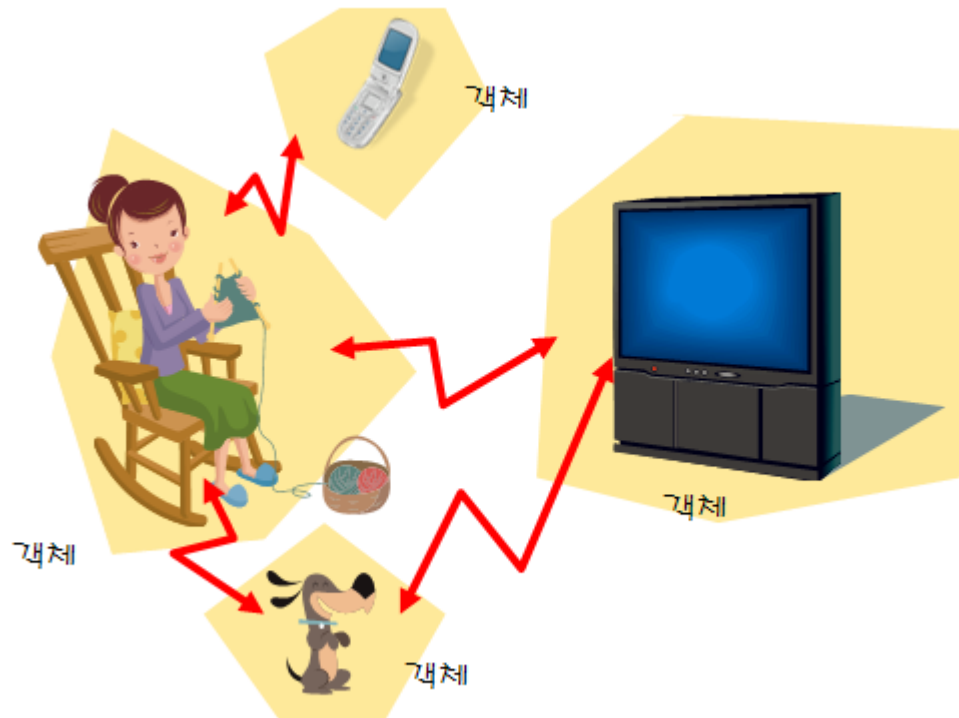




C++ Espresso

제10장 프렌드와 연산자 중복





이번 장에서 학습할 내용



- 프렌드 함수
- 연산자 중복
- 타입 변환

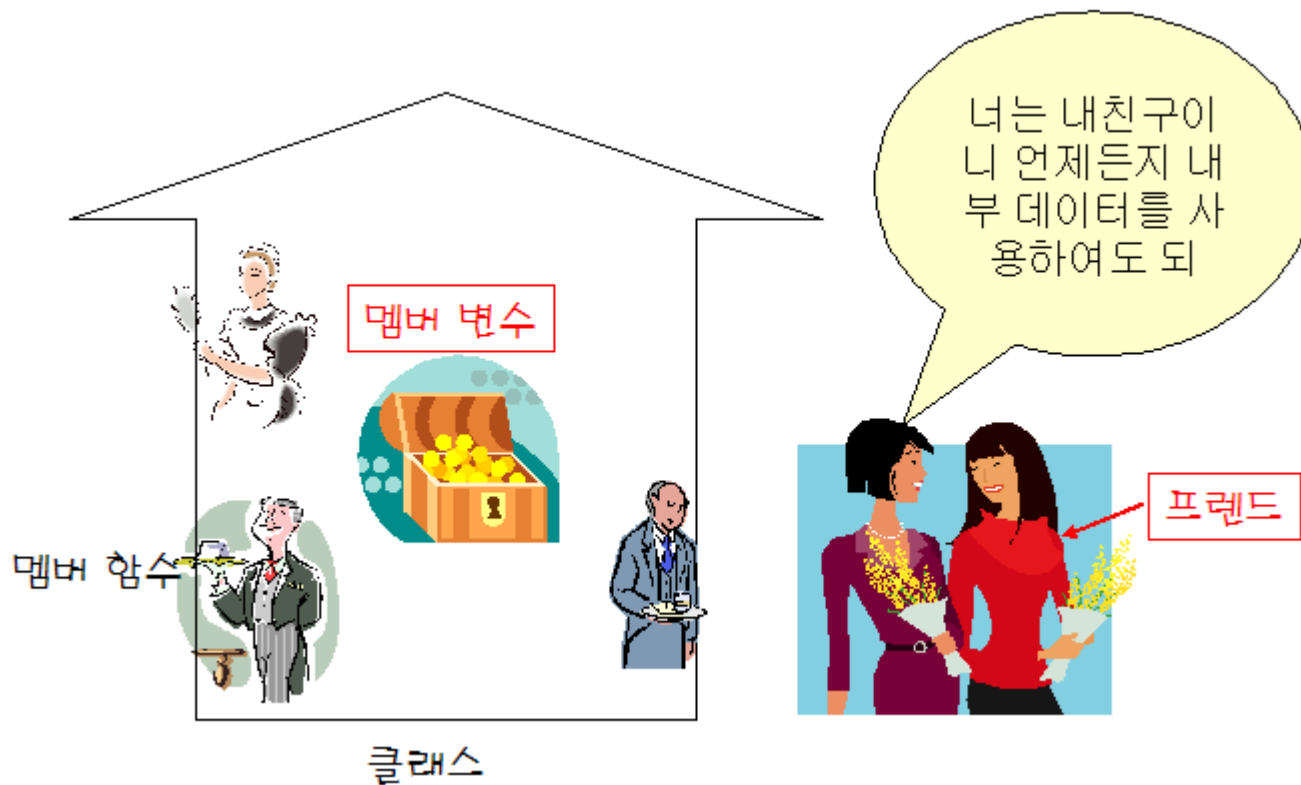
C++의 고급
기능인
프렌드와
연산자 중복을
살펴봅니다.





프렌드 함수

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





프렌드 함수 선언 방법

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

```
class MyClass
```

```
{
```

```
    friend void sub();
```

```
    ....
```

```
};
```

프렌드 함수



예제



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

```
class Company {
private:
    int sales, profit;
```

```
// sub()는 Company의 전용부분에 접근할 수 있다.
friend void sub(Company& c);
```

```
public:
    Company(): sales(0), profit(0)
    {
    }
};
void sub(Company& c)
{
    cout << c.profit << endl;
}
```



예제



```
int main()
{
    Company c1;
    sub(c1);
    return 0;
}
```



0



프렌드 클래스

- 클래스도 프렌드로 선언할 수 있다.
- (예) **Manager**의 멤버들은 **Employee**의 전용 멤버를 직접 참조할 수 있다.

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

프렌드 클래스



프렌드 함수의 용도

- 두개의 객체를 비교할 때 많이 사용된다.

① 일반 멤버 함수 사용

```
if( obj1.equals(obj2) )  
{  
    ...  
}
```

② 프렌드 함수 사용

```
if( equals(obj1, obj2) )  
{  
    ...  
}
```

이해하기가 쉽다



예제



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

class Date
{
    friend bool equals(Date d1, Date d2);
private:
    int year, month, day;
public:
    Date(int y, int m, int d)
    {
        year = y;
        month = m;
        day = d;
    }
}
```



예제



// 프렌드함수

```
bool equals(Date d1, Date d2)
```

```
{
```



```
}
```

```
    return d1.year == d2.year && d1.month == d2.month && d1.day == d2.day;
```

```
int main()
```

```
{
```



```
}
```

```
    Date d1(1960, 5, 23), d2(2002, 7, 23);  
    cout << equal_f(d1, d2) << endl;
```

멤버 변수 접근
가능



예제



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

class Complex {

public:
    friend Complex add (Complex, Complex);
    Complex (double r, double i) {re=r; im=i; }
    Complex(double r) { re=r; im=0; }
    Complex () { re = im = 0; }
    void Output(){
        cout << re << " + " << im << "i" << endl;
    }

private:
    double re, im;
};
```



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.Output();
    return 0;
}
```



$4 + 6i$

계속하려면 아무 키나 누르십시오 . . .



중간 점검 문제

1. 프렌드 함수란 무엇인가?
2. 어떤 경우에 프렌드 함수가 유용한가?
3. 두 개의 **Vector** 객체를 더하는 프렌드 함수를 정의하라.





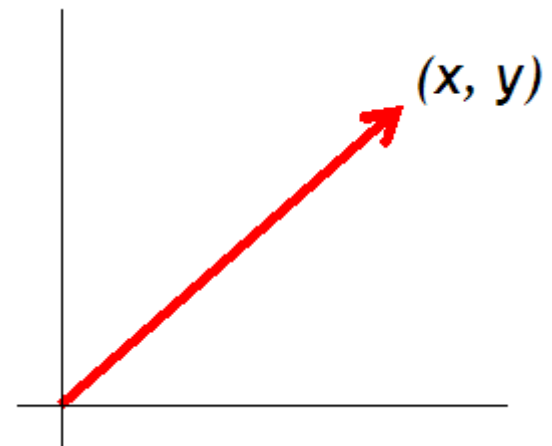
연산자 중복

- 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));`



원점 벡터 예제

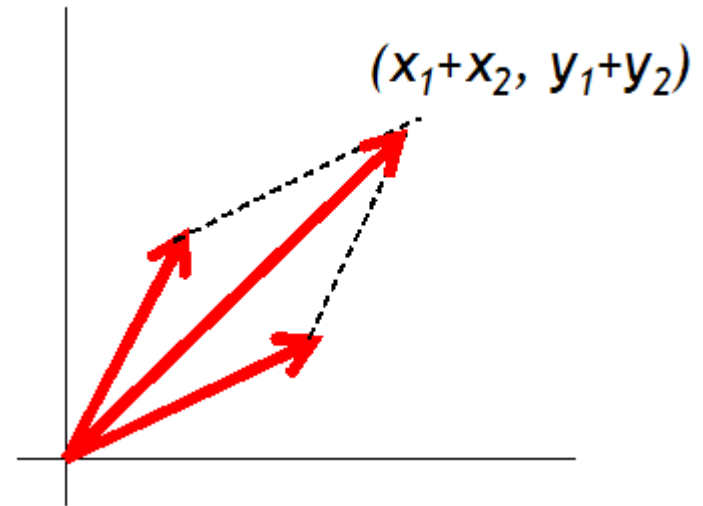
```
class Vector
{
private:
    double x, y;
public:
    Vector(double x, int double){
        this->x = x;
        this->y = y;
    }
}
```





벡터간의 연산을 연산자로 표기

Vector v_1, v_2, v_3 ;
 $v_3 = v_1 + v_2$;





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
}
```

(예) `Vector operator+(const Vector&, const Vector&);`



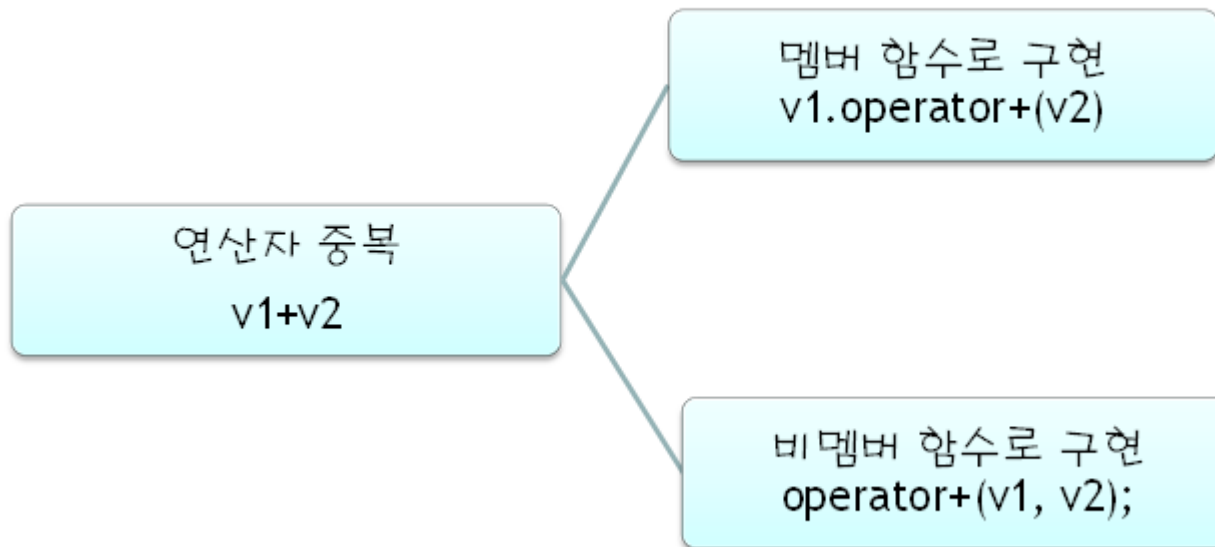
연산자 중복 함수 이름

- 중복 함수 이름은 `operator`에 연산자를 붙이고 함수 기호

연산자	중복 함수 이름
+	<code>operator+()</code>
-	<code>operator-()</code>
*	<code>operator*()</code>
/	<code>operator/()</code>

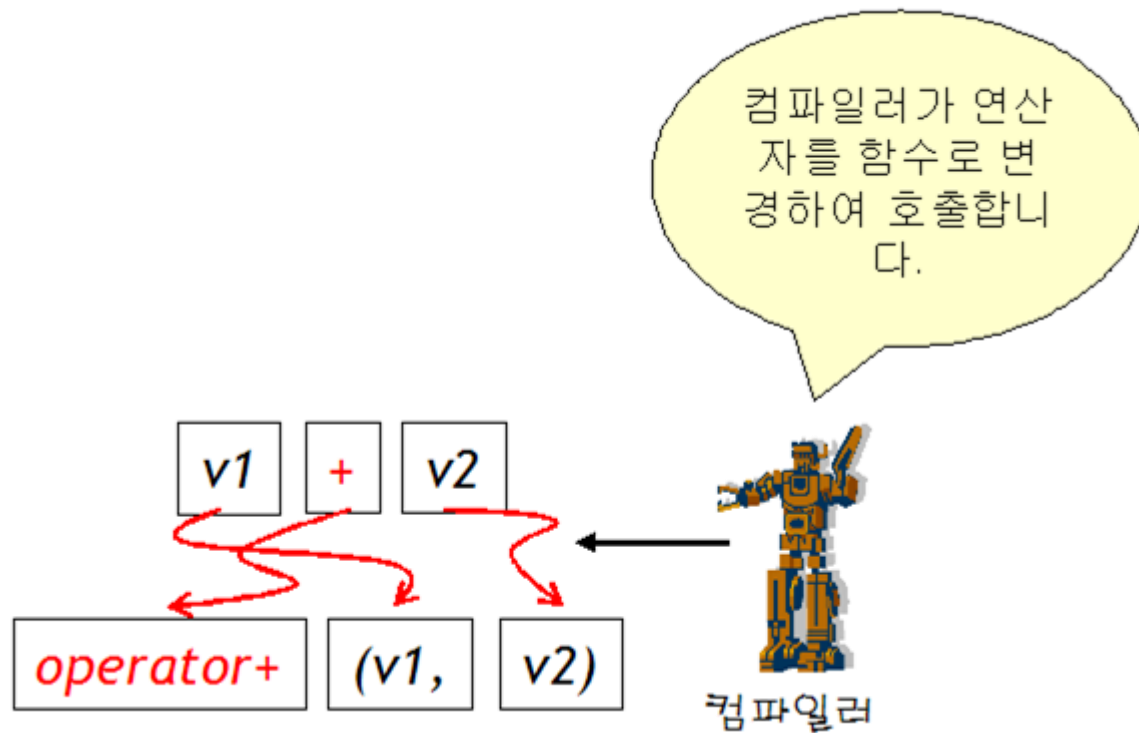


연산자 중복 구현의 방법





전역 함수로 구현하는 방법





예제



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

class Vector
{
private:
    double x, y;
public:
    Vector(double x, double y){
        this->x = x;
        this->y = y;
    }
    friend Vector operator+(const Vector& v1, const Vector& v2);
    void display()
    {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
```



예제



```
Vector operator+(const Vector& v1, const Vector& v2)
{
    Vector v(0.0, 0.0);
    v.x = v1.x + v2.x;
    v.y = v1.y + v2.y;
    return v;
}
```

```
int main()
{
    Vector v1(1, 2), v2(3, 4);
    Vector v3 = v1 + v2;
    v3.display();

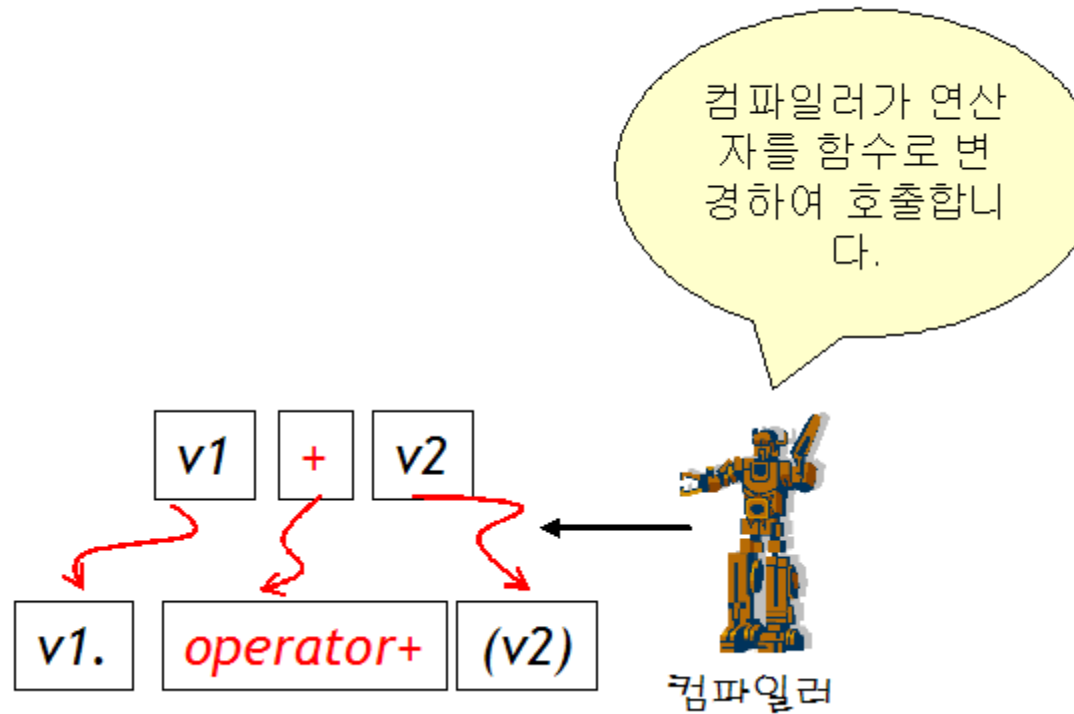
    return 0;
}
```

(4, 6)





멤버 함수로 구현하는 방법





예제



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

class Vector
{
private:
    double x, y;
public:
    Vector(double x, double y){
        this->x = x;

        this->y = y;
    }
    Vector operator+(Vector& v2)
    {
        Vector v(0.0, 0.0);
        v.x = this->x + v2.x;
        v.y = this->y + v2.y;
        return v;
    }
}
```




예제



```
void display()
{
    cout << "(" << x << ", " << y << ")" << endl;
}

int main()
{
    Vector v1(1.0, 2.0), v2(3.0, 4.0);
    Vector v3 = v1 + v2;
    v3.display();

    return 0;
}
```

(4, 6)





멤버 함수로만 구현가능한 연산자

- 아래의 연산자는 항상 멤버 함수 형태로만 중복 정의가 가능하다.

연산자	설명
=	대입 연산자
()	함수 호출 연산자
[]	배열 원소 참조 연산자
->	멤버 참조 연산자



중복이 불가능한 연산자

- 아래의 연산자는 중복 정의가 불가능하다.

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



중간 점검 문제

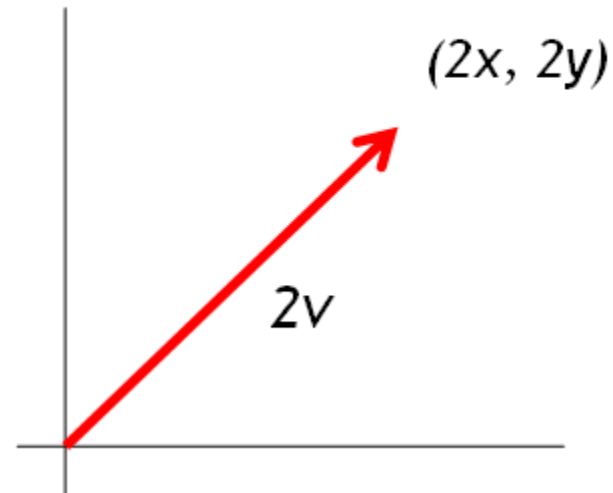
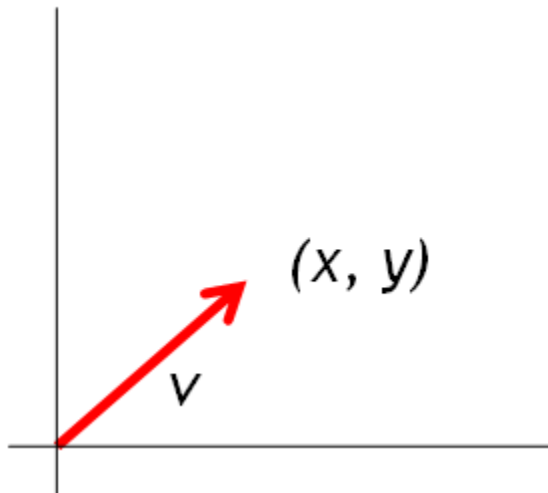
1. 벡터 사이의 뺄셈 연산자 `-`을 중복하여 보자.
2. 두개의 벡터가 같은지를 검사하는 `==` 연산자를 중복하라.
3. 문자열을 나타내는 **String** 클래스를 작성하고 `+` 연산자를 중복하라.





피연산자 타입이 다른 연산

- 벡터의 스칼라곱(scalar product)이라고 불리는 연산을 구현
- 벡터가 (x, y) 이고 α 가 스칼라일 때에 벡터 스칼라곱은 $(\alpha x, \alpha y)$





곱셈 연산자 중복

- 곱셈 연산자를 중복하여 정의한다.
- 교환 법칙이 성립하여야 함

`Vector operator*(Vector& v, double alpha); // $v * 2.0$ 형태 처리`

`Vector operator*(double alpha, Vector& v); // $2.0 * v$ 형태 처리`



곱셈 연산자 중복

Vector.cpp

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

class Vector
{
    friend Vector operator*(Vector& v, double alpha);
    friend Vector operator*(double alpha, Vector& v);
private:
    double x, y;
public:
    Vector(double xvalue=0.0, double yvalue=0.0) : x(xvalue), y(yvalue){ }
    void display(){
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
```



곱셈 연산자 중복

```
Vector operator*(Vector& v, double alpha)
{
    return Vector(alpha*v.x, alpha*v.y);
}
```

* 연산자 함수 정의

* 연산자 함수 정의

```
Vector operator*(double alpha, Vector& v)
{
    return Vector(alpha*v.x, alpha*v.y);
}
```

```
int main()
{
    Vector v(1.0, 1.0);
    Vector w = v * 2.0;
    Vector z = 2.0 * v;
    w.display();
    z.display();
    return 0;
}
```

실행결과

(2, 2)

(2, 2)



== 연산자 중복

- 두개의 객체가 동일한 데이터를 가지고 있는지를 체크하는데 사용

연산자	중복 함수 이름
==	<code>operator==()</code>
!=	<code>operator!=()</code>



== 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;
    }

    friend bool operator==(const Vector& v1, const Vector& v2);
    friend bool operator!=(const Vector& v1, const Vector& v2);
};
```



== 연산자의 중복

```
bool operator==(const Vector &v1, const Vector &v2)
{
    return v1.x == v2.x && v2.y == v2.y;
}
bool operator!=(const Vector &v1, const Vector &v2)
{
    return !(v1 == v2); // 중복된 == 연산자를 이용
}
```

==와 !=연산자 함수를
전역 함수로 구현

```
int main()
{
    Vector v1(1, 2), v2(1, 2);

    cout.setf(cout.boolalpha);
    cout << (v1 == v2) << endl;
    cout << (v1 != v2) << endl;
    return 0;
}
```

실행결과

true
false



<< 연산자의 중복

```
Vector v(2, 3);
```

```
cout << v;
```

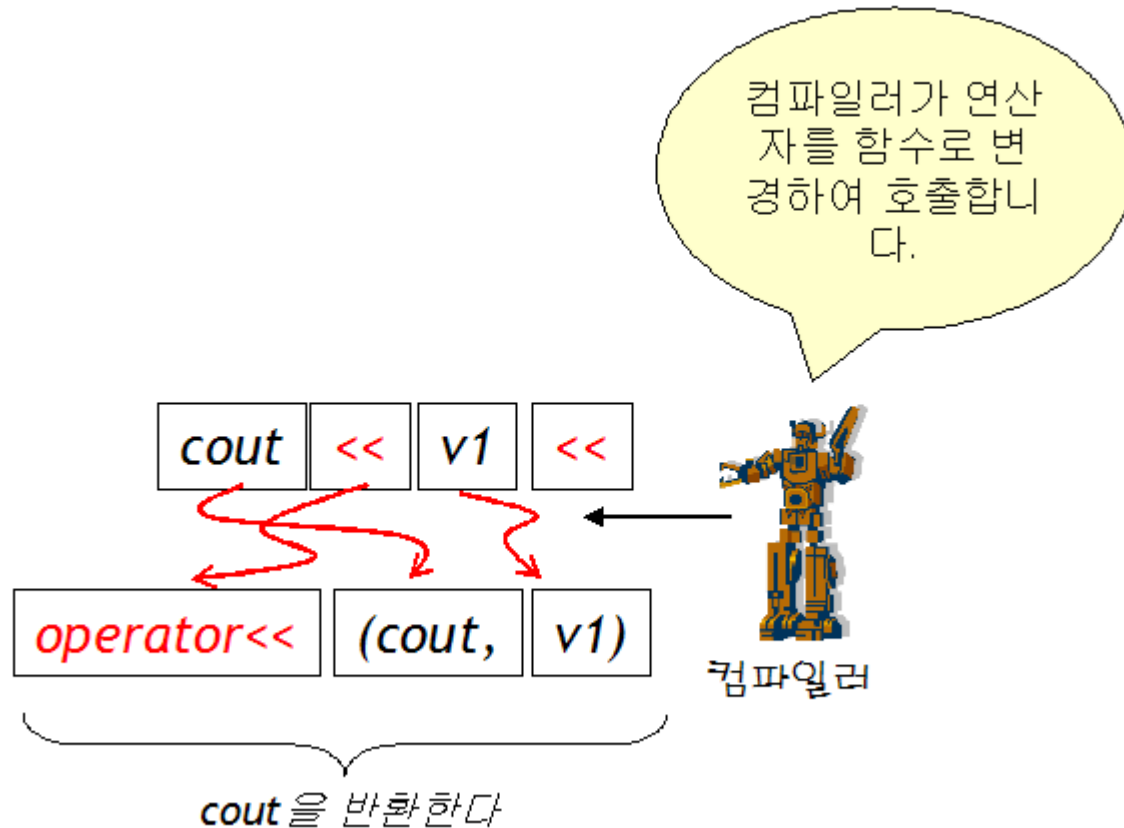
// 화면에 (2, 3)이 출력된다.





<<과 >> 연산자 중복

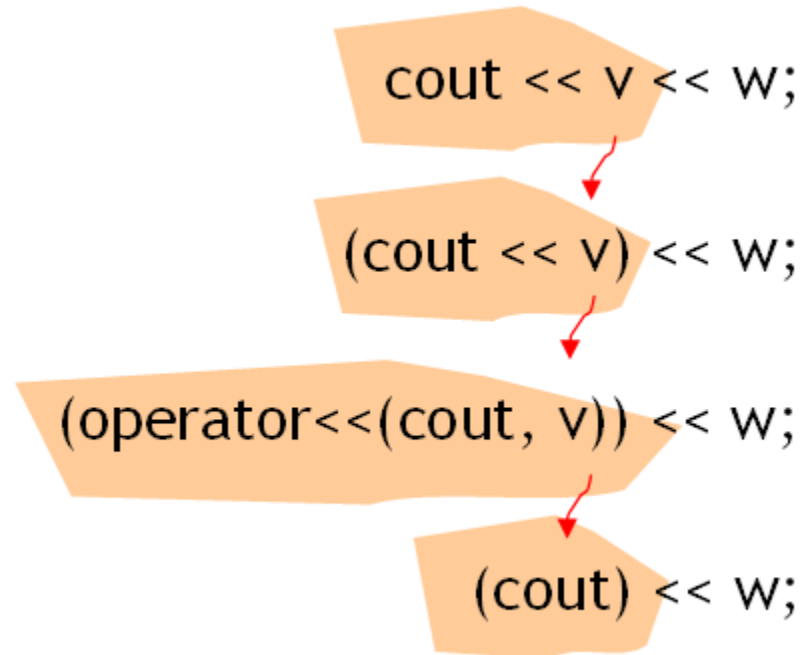
- 연산을 수행한 후에 다시 스트림 객체를 반환하여야 함





주의할 점

- 전역 함수 형태만 사용 가능: 우리가 **ostream** 클래스를 다시 정의할 수 없다.
- 반드시 **ostream** 참조자를 반환





<< 연산자의 중복

Vector.cpp

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



<< 연산자의 중복

```
ostream& operator<<(ostream& os, const Vector& v)
{
    os << "(" << v.x << "," << v.y << ")" << endl;
    return os;
}
```

<< 연산자 함수 정의

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

실행결과

(1,2)

(3,4)

(0,0)



>> 연산자의 중복

- 입력 연산자 >>의 중복
- 오류 처리를 하는 것이 좋음

```
istream& operator>>(istream& in, Vector& v)
```

```
{
```

```
    in >> v.x >> v.y;
```

```
    if(!in)
```

```
        v = Vector(0, 0);
```

```
    return in;
```

```
}
```

입력 오류 처리



= 연산자 중복

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

주의: 반드시 현재 객체의 레퍼런스를 반환

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



얇은 대입 문제

- 동적 할당 공간이 있으면 반드시 = 연산자를 중복 정의하여야 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;
    }
};
```



얕은 대입 문제

- 동적 할당 공간이 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;
```

```
}
```

```
};
```

= 연산자 함수 정의



증가/감소 연산자의 중복

- ++와 - 연산자의 중복

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



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;
    }
};
```

++ 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

- 전위와 후위 연산자를 구별하기 위하여 ++가 피연산자 뒤에 오는 경우에는 **int**형 매개 변수를 추가한다.

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



전위 / 후위의 문제

```
Vector& operator++()
```

```
{  
    x++;  
    y++;  
    return *this;  
}
```

++v 형태의 증가 연산자
함수 정의

```
const Vector operator++(int)
```

```
{  
    Vector saveObj = *this;  
    x++;  
    y++;  
    return saveObj;  
}  
};
```

v++ 형태의 증가 연산자
함수 정의



[] 연산자의 중복

- 인덱스 연산자의 중복

연산자	중복 함수 이름
v[]	<u>v.operator[]()</u>

```
MyArray A;
```

```
A[3] = 10;
```

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



인덱스 연산자 중복

my_array.cpp

```
#include <iostream>
#include <assert.h>
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

```
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

```
MyArray& MyArray::operator=(const MyArray& a) {  
    if (&a != this) {  
        delete [] data;  
        size = a.size;  
        data = new int[size];  
  
        for (int i = 0; i < size; i++)  
            data[i] = a.data[i];  
    }  
    return *this;  
}
```

대입 연산자 중복 정의

// 자기 자신인지를 체크 의

// 동적 메모리 반납

// 새로운 크기를 설정

// 새로운 동적 메모리 할당

// 데이터 복사

// a = b = c와 같은 경우를 대비



인덱스 연산자 중복

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

인덱스 연산자 정의

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

// 인덱스가 범위에 있지 않으면 중지

// 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 << a3와 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;
}
```

실행결과

1 2 3 4 0 0 0 0 0 0



포인터 연산자의 중복

- 간접 참조 연산자 *와 멤버 연산자 ->의 중복 정의

연산자	중복 함수 이름
*	<code>operator*()</code>
->	<code>operator->()</code>



포인터 연산자의 중복

smartp.cpp

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

class Pointer {
    int *pi;
public:
    Pointer(int *p): pi(p)
    {
    }
    ~Pointer()
    {
        delete pi;
    }
}
```



포인터 연산자의 중복

```
int* operator->() const  
{  
    return pi;  
}
```

-> 연산자 중복 정의

```
int& operator*() const  
{  
    return *pi;  
}  
};
```

* 연산자 중복 정의

```
int main()  
{  
    Pointer p(new int);  
    *p = 100;  
    cout << *p << endl;  
    return 0;  
}
```

실행결과

100

계속하려면 아무 키나 누르십시오 . . .



스마트 포인터

- 포인터 연산 정의를 이용하여서 만들어진 향상된 포인터를 스마트 포인터(**smart pointer**)라고 한다.
- 주로 동적 할당된 공간을 반납할 때 사용된다.

```
class Pointer {  
    Car *pc;  
public:  
    Pointer(Car *p): pc(p){ }  
    ~Pointer(){ delete pc; }  
    Car* operator->() const { return pc; }  
    Car& operator*() const { return *pc; }  
};
```

스마트 포인터

```
int main()  
{  
    Pointer p(new Car(0,1,"red"));  
    p->speed = 100;  
    cout << *p;  
    (*p).speed = 200;  
    cout << *p;  
    p->setSpeed(300);  
    cout << *p;  
    return 0;  
}
```



함수 호출 연산자 ()의 중복

- 함수 호출때 사용하는 () 도 중복이 가능하다

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



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



중간 점검 문제

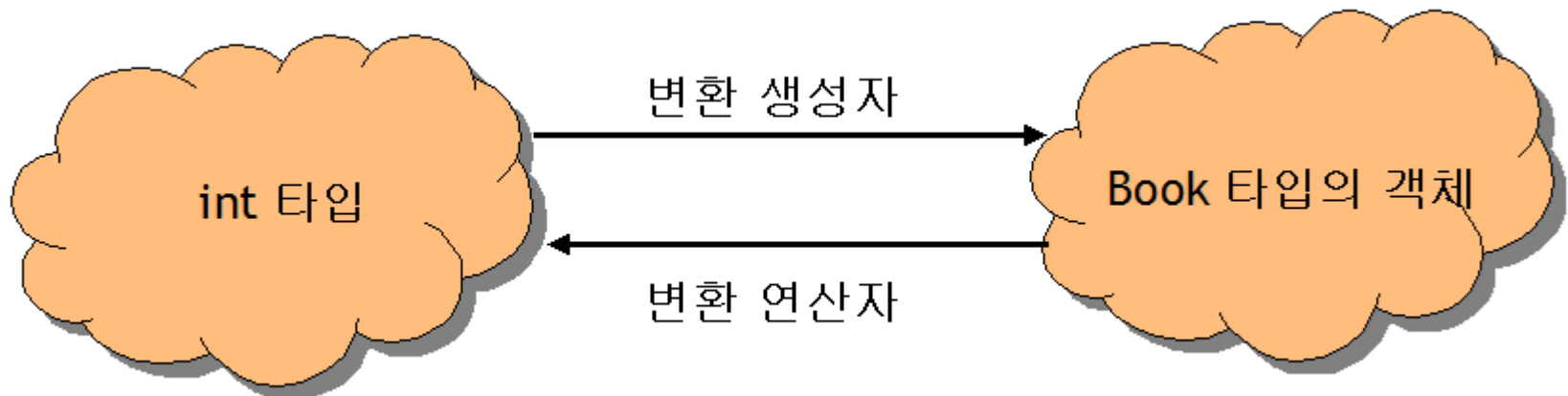
1. 벡터를 나타내는 **Vector** 클래스에 `-` 연산자를 중복하라.
2. 벡터를 나타내는 **Vector** 클래스에 `+=` 연산자를 중복하라.
3. 문자열을 나타내는 **String** 클래스를 작성하고 `<<` 연산자를 중복하라.





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

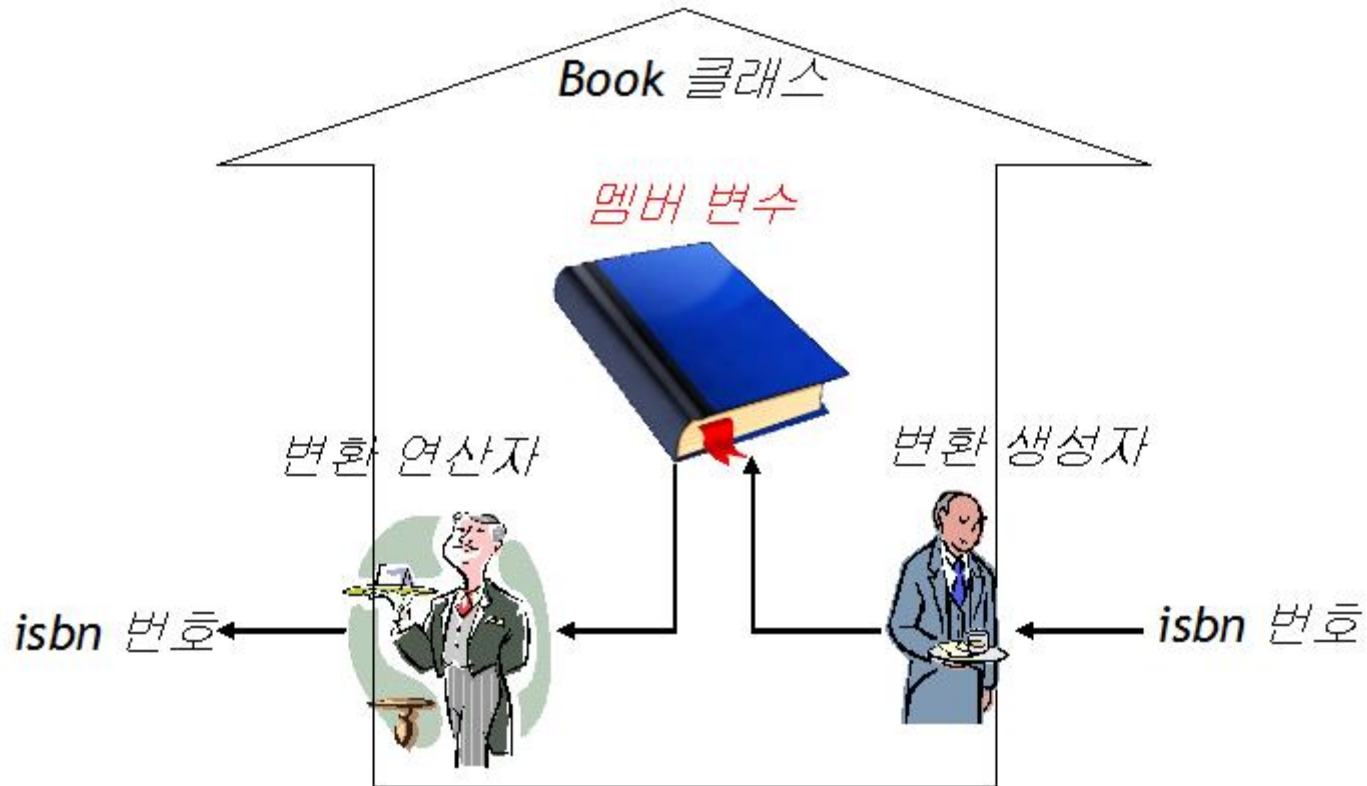


그림 12.7 변환 생성자와 변환 연산자



변환 생성자

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

class Book
{
private:
    int isbn; // 책의ISBN
    string title; // 책의제목
public:
    Book() { // 생성자
        isbn = 0;
        title = "unknown";
    }
    Book(int isbn) {
        this->isbn = isbn;
        this->title = "unknown";
    }
    void display() {
        cout << isbn << ":" << title << endl;
    }
};
```

변환 생성자
(int->Book)



예제



```
int main()
{
    Book b1 = 9782001;      // int 타입을 Book 타입에 대입
    b1.display();
    b1 = 9783001;  // 가능
    b1.display();
    return 0;
}
```



9782001:unknown
9783001:unknown
계속하려면 아무 키나 누르십시오 . . .

컴파일러는 변환 생성자를 이용하여서 정수 9782001을 Book 객체로 변환하는 것이다.



버그의 원인

- 변환 생성자는 버그의 원인이 될 수도 있다.
- `Book b2 = 3.141592;`
- `b2.display();`
- (설명) 실수->정수-> 객체
- (해결책) 만약 생성자 앞에 **explicit**를 붙이면 컴파일러가 자동적으로 타입 변환을 하지 못한다.



변환 연산자 중복 정의

```
operator 데이터_타입()
{
....
}
```

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

변환 생성자
(int->Book)



변환 연산자의 중복 정의

```
#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;           // 변환생성자실행!  
    b1.display();  
    int isbn = b1;               // 변환연산자실행!  
    cout << isbn << endl;  
    check(b1);                   // 변환연산자실행!  
  
    return 0;  
}
```



연산자 중복시 주의할 점

- 새로운 연산자를 만드는 것은 허용되지 않는다.
- :: 연산자, .* 연산자, . 연산자, ?: 연산자는 중복이 불가능하다.
- 내장된 **int**형이나 **double**형에 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.



중간 점검 문제

1. 클래스 **Car**를 **string**으로 변환하는 변환 연산자를 작성하시오.
2. 변환 연산자의 위험성은 무엇인가?





MyString

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

class MyString
{
private:
    char *pBuf;                //동적으로 할당된 메모리의 주소값 저장

public:
    MyString(const char *s=NULL);
    MyString(MyString& s);
    ~MyString();

    void print();               // 문자열을 화면에 출력
    int getSize();              // 문자열의 길이 반환
    MyString operator+(MyString& s);    // + 연산자 중복정의
};
```



MyString

// 생성자

```
MyString::MyString(const char *s)
{
    if( s == NULL )
    {
        pBuf = new char[1];
        pBuf[0] = NULL;
    }
    else
    {
        pBuf = new char[::strlen(s)+1];
        strcpy(pBuf, s);
    }
}
```

// 복사생성자

```
MyString::MyString(MyString &s)
{
    pBuf = new char[s.GetSize()+1];
    strcpy(pBuf, s.pBuf);
}
```



MyString

```
MyString::~MyString()
{
    if ( pBuf )
        delete [] pBuf;
}
void MyString::print()
{
    cout << pBuf << endl;
}
int MyString::getSize()
{
    return strlen(pBuf);
}
MyString MyString::operator+(MyString& s)
{
    char *temp = new char[getSize() + s.getSize() + 1];
    strcpy(temp, pBuf);
    strcat(temp, s.pBuf);
    MyString r(temp);
    delete [] temp;
    return r;
}
```



MyString

```
int main() {  
  
    MyString s1("Hello ");  
    MyString s2("World!");  
    MyString s3 = s1 + s2;  
  
    s1.print();  
    s2.print();  
    s3.print();  
  
    return 0;  
}
```



Hello
World!
Hello World!



MyArray

```
#include <iostream>
#include <assert.h>
using namespace std;

// 항상된배열을나타낸다.
class MyArray {
    friend ostream& operator<<(ostream &, const MyArray &);           //
출력연산자<<
private:
    int *data;                // 배열의데이터
    int size;                 // 배열의크기

public:
    MyArray(int size = 10);    // 디폴트생성자
    ~MyArray();                // 소멸자

    int getSize() const;       // 배열의크기를반환
    MyArray& operator=(const MyArray &a);    // = 연산자중복정의
    int& operator[](int i);     // [] 연산자중복: 설정자
};
```



MyArray

```
MyArray::MyArray(int s) {  
    size = (s > 0 ? s : 10); // 디폴트크기를10으로한다.  
    data = new int[size];    // 동적메모리할당  
  
    for (int i = 0; i < size; i++)  
        data[i] = 0;        // 요소들의초기화  
}  
MyArray::~MyArray() {  
    delete [] data;          // 동적메모리반납  
    data = NULL;  
}  
MyArray& MyArray::operator=(const MyArray& a) {  
    if (&a != this) {        // 자기자신인지를체크  
        delete [] data;      // 동적메모리반납  
        size = a.size;       // 새로운크기를설정  
        data = new int[size]; // 새로운동적메모리할당  
  
        for (int i = 0; i < size; i++)  
            data[i] = a.data[i]; // 데이터복사  
    }  
    return *this;            // a = b = c와같은경우를대비  
}
```



MyArray

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

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

// 프렌드함수정의
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 << a3와같은경우대비
}
```



MyArray

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

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

    return 0;
}
```



1 2 3 4 0 0 0 0 0 0



Q & A

