C++ Espresso

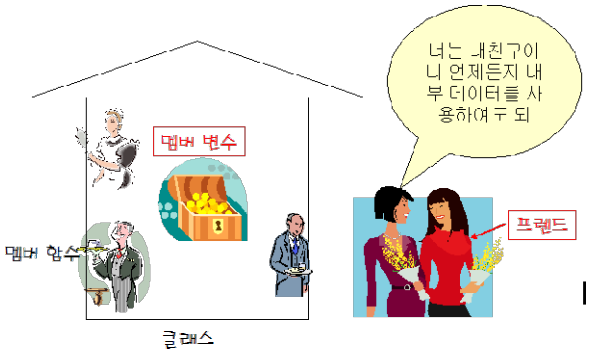
제10장 프렌드와 연산자 중복

이번 장에서 학습할 내용

- 프렌드 함수
- 연산자 중복
- 타입 변환

프렌드 함수

- *프렌드 함수(friend function):* 멤버함수가 아닌 일반함수 이지만 클래스의 내부 데이터에 접근할 수 있는 특수한 함수
  - friend 는 정보은닉에 위배 → 자주 사용하면 안됨. 필요한 곳만.



프렌드 함수 선언 방법

- 프렌드 함수의 원형은 비록 클래스 안에 포함하지만 멤버 함수는 아니다. → 일반함수이다.
- 프렌드 함수는 클래스 내부에 원형만 선언
  - 프렌드 함수의 본체는 외부에서 따로 정의
- 프렌드 함수는 클래스 내부의 모든 멤버 변수를 사용 가능
- 함수 sub( )는 MyClass 의 모든 멤버 접근 가능

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

프렌드 함수

예제

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

class Company {
private:
  int sales, profit;

  // sub()는Company의 전용에 접근 가능.
  friend void sub(Company& c);
public:
  Company(): sales(0), profit(0) { }
};

// 클래스 Company 의 객체를 매개변수로
// 받고 그 안의 멤버 접근 가능
void sub(Company& c) {
  cout << c.profit << endl;
}
```

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

0

- 프렌드함수를 Public 으로 선언해도 결과 동일
- sub( ) 는 일반함수,
- Company 가 sub( ) 를 friend 로 생각
- Sub( ) 는 Company 모든 멤버 접근 가능

- 참고) 함수에서 인자로 객체를 받는 경우 참조자로 받아야 함 → 아니면 복사 생성자 호출로 시간 낭비

프렌드 클래스(비중요)

- 클래스도 프렌드로 선언할 수 있다.
- 관리자 밑에 여러 직원이 있는 경우, 관리자는 직원 내부 데이터에 접근 필요 → friend 선언
- Manager의 멤버들은 Friend 이므로 Employee의 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

- 두 개의 객체를 비교할 때, 클래스의 연산자 함수 작성시 사용.

① 일반 멤버 함수 사용

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

② 프렌드 함수(일반 함수) 사용

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

이해하기가 쉽다

예제(equal 연산자)

멤버 변수 접근 가능

```
// 프렌드함수
bool equals(Date d1, Date d2) {
    return (d1.year == d2.year) && (d1.month == d2.month) && (d1.day == d2.day);
}
```

```
class Date {
private:
  friend bool equals(Date d1, Date d2);
  int year, month, day;

public:
  Date(int y, int m, int d) {
          year = y;
          month = m;
          day = d;
  }
};
```

```
int main() {
  Date d1(1960, 5, 23), d2(2002, 7, 23);
  cout << equals(d1, d2) << endl;
}
```

예제(add 연산자)

```
class Complex {      // 복소수 클래스
  double re, im;
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;
  }
};
```

```
// 객체를 반환
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 . . .
```

연산자 중복

- 일반적으로는 연산자 기호를 사용하는 편이 함수를 사용하는 것보다 이해하기가 쉽다.
- 다음의 두 가지 문장 중에서 어떤 것이 더 이해하기 쉬운가?

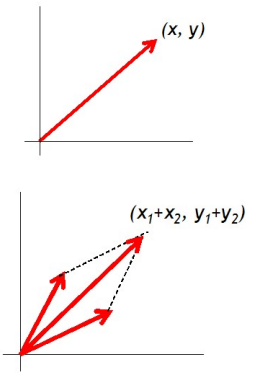
  1. sum = x + y + z;
  2. sum = add(x, add(y, z));

- int, float 등은 내부적으로 +, - 등 연산자 미리 정의
- string class 등은 모든 필요 연산자 정의되어 있음.
- 개발자가 만든 class/객체들 → 이들간의 +, - 등 연산자는 개발자가 만들어야...

벡터 예제

```
class Vector{
        double x, y;
public:
        Vector(double x=1, int double y=1){
                this->x = x;
                this->y = y;
        }
}
```

```
Vector v1(1, 2), v2(2, 3), v3;
v3 = v1 + v2;
```



- Vector 는 개발자가 만든 class, 이 클래스 객체간의 연산자는 존재하지 않음
- 개발자가 연산자를 만들어야 함.

연산자 중복

- C++에서 연산자는 함수(멤버함수 혹은 friend 함수)로 정의

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

(예) cf) // **함수 이름 같아도 매개변수 다름** → 연산자 중복, 오버로딩(overloading)

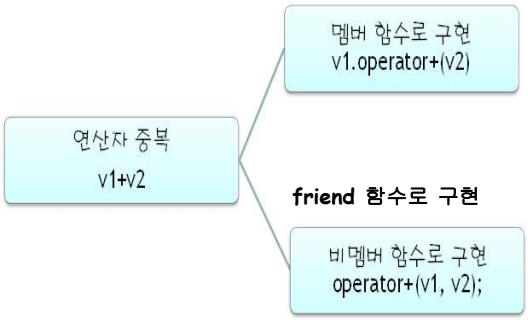
```
Vector operator+(const Vector&, const Vector&);
Vector operator+(const Vector&);
```

- 중복 함수 이름은 operator에 연산자를 붙이고 (...) 붙여 사용

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

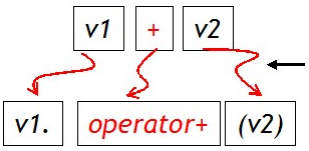
연산자 중복 구현의 방법

- 멤버함수와 일반함수(friend 함수) 같이 있으면 멤버함수 우선



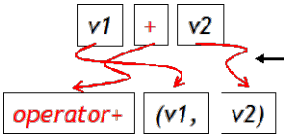
연산자 중복 함수 해석

```
class Vector {
...
        // 멤버함수로 구현
   Vector operator+(const Vector&);  // 매개변수 1개
}
...
Vector v1(2, 3), v2(3, 4), v3;
v3 = v1 + v2;          // v1.operator+(v2) 로 해석
```



컴파일러가 연산자를
함수로 변경하여 호출

```
class Vector {
...
        // 일반 함수로 구현    // 매개변수 2개
   friend Vector operator+(const Vector&, const Vector&);
}
...
Vector v1(2, 3), v2(3, 4), v3;
v3 = v1 + v2;          // operator+(v1, v2) 로 해석
```



+ 는 함수 이름

연산자 중복 구현의 방법

- Vector a(1.1, 2.2), b(1.5, 2.1), c(0, 0); 클래스의 연산자 함수 구현시
  - 반환형은 모두 Vector → 벡터끼리 더하면 결과는 벡터

- 연산자 함수는 다음 두 가지로 구현 가능. 아래 예와 같이 함수 호출 코드는 동일
  - 멤버 함수로 구현 → Vector operator+(const Vector&);

```
// 멤버함수 구현시 해석
c = b + a;          // c = b.operator+(a); 로 해석 by 컴파일러 , 매개변수 1개
c = a + b;          // c = a.operator+(b); 로 해석
```

  - 일반 함수(friend 함수, 전역함수)로 구현
    friend Vector operator+(const Vector&, const Vector&);

```
// friend 함수 구현시 해석
c = b + a;          // c = operator+(b, a); 로 해석 by 컴파일러, 매개변수 2개
c = a + b;          // c = operator+(a, b); 로 해석
```

일반함수로 구현 예제

```
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;     }
};

Vector operator+(const Vector& v1, const Vector& v2) {  // 인자는 모두 참조자로 선언
     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 = operator+(v1, v2) 로 해석
     v3.display();

     return 0;
}
```

```
(4, 6)
```

멤버 함수로 구현 예제

```
class Vector {
private:
  double x, y;
public:
  Vector(double x, double y){
    this->x = x;      this->y = y;
  }
  Vector operator+(Vector& a)    {
    Vector v(0.0, 0.0);
    v.x = this->x + a.x;
    v.y = this->y + a.y;
    return v;
  }
  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)

v3 = v1.operator+(v2);

멤버 함수로만 구현가능한 연산자

- 아래의 연산자는 항상 멤버 함수 형태로만 중복 정의가 가능하다.

연산자	설명
=	대입 연산자 복사생성자 아님
()	함수 호출 연산자
[]	배열 원소 참조 연산자 cf) 인덱서
->	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

<> 다음을 멤버함수로 작성

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

중간 점검 문제 1

```
Vector operator-(Vector& v2) {  // 멤버함수 구현
        Vector v(0.0, 0.0);
        v.x = this->x - v2.x;          // v.x = x - v2.x;
        v.y = this->y - v2.y;
        return v;
}
                                       // friend 함수 구현
Vector operator-(Vector& v1 , Vector& v2) {
        Vector v(0.0, 0.0);
        v.x = v1.x - v2.x;
        v.y = v1.y - v2.y;
        return v;
}
```

중간 점검 문제 2

```
// 멤버함수로 구현
bool operator==(Vector& a) {
    if(x==a.x && y==a.y)
        return 1;
    else
        return 0;
}
bool operator!=(Vector& a) {
    return ! (*this == a);
}
```

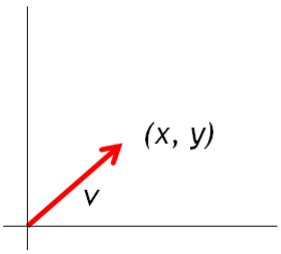
```
// 일반함수로 구현
bool operator==(Vector& a , Vector& b) {
    if(a.x==b.x && a.y==b.y)
        return 1;
    else
        return 0;
}
// friend bool operator!=(Vector& a , Vector& b) { ..} 오류_
bool operator!=(Vector& a , Vector& b) {
        return !(a==b);
}
```

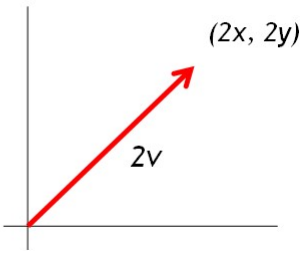
```
Vector v1(1, 2), v2(2, 3);

cout << (v1 == v2) << (v1 != v2);
```

피연산자 타입이 다른 연산

- 앞에서는 두 벡터 덧셈, 뺄셈 등 피연산자가 동일 type 인 경우

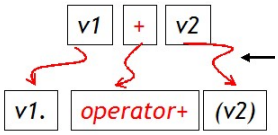
  Vector a, b, c; c = a+b;
- 여기서는 2 * a(→vector) 와 같이 피연산자 type 다른 경우 연산자 구현
  - 벡터의 스칼라곱(scalar product)이라고 불리는 연산을 구현
  - 벡터가 (x, y)이고 α가 스칼라일 때에 벡터 스칼라곱은 (αx, αy)





스칼라 곱셈 연산자 중복

- 곱셈 연산자 → 교환 법칙이 성립하여야 함 → 2*a, a*2 둘 다 가능해야 함
- 일반 함수로 구현 → 가능 → 2개 함수 모두 작성
  - Vector operator*(Vector& v, double alpha); // 인자 (v, 2.0) ← v * 2.0 형태 처리
  - Vector operator*(double alpha, Vector& v); // 인자 (2.0, v) ← 2.0 * v 형태 처리
- 멤버함수로 구현 → 불가능
  - v * 2.0 형태 처리 → v.operator*(2.0)
  - 2.0 * v 형태 처리 → (2.0).operator*(v) ??
  - 멤버함수로 구현 불가



- 스칼라 곱셈연산자 중복은 일반 함수로만 구현 → 멤버함수로 구현 불가

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;
   }
};
```

교환 법칙이 성립해야 하기에 함수 2개 작성 → 2*a, a*2 둘 다 가능해야 함

곱셈 연산자 중복

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

* 연산자 함수 정의

* 연산자 함수 정의

실행결과

```
(2, 2)
(2, 2)
```

== 연산자 중복

- 두개의 객체가 동일한 데이터를 가지고 있는지를 체크하는데 사용

연산자	중복 함수 이름
==	operator==()
!=	operator!=()

== 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
}

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

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

==와 !=연산자 함수를 전역 함수로 구현

멤버함수로도 구현 가능(매개변수 2개 동일)

// 출력시 0, 1 대신 true, false 로 출력

실행결과

```
true
false
```

참고) 참조자 반환 함수

```
// 값 반환
int func_1(int x) {
        ++x;
        return x;
}
// 지역변수 x 값을 전달
```

```
// 참조자 반환_1
// int x = n;    x는 지역변수
int& func_3(int x) {
        ++x;
        return x;
        // 지역변수 x를 전달
}

// 참조자 반환_2
// int &x = n;    x는 n 자체
int& func_4(int &x) {
        ++x;
        return x;
        // 매개변수 x를 전달
}
```

```
void main() {
        int n = 10;
        int k1 = func_1(n);
                // int k1 = x;    // k1 에 x 값 전달 후 x 소멸

        int& k3 = func_3(n);
                // int& k3 = x;  // k3은 함수 지역변수 x 를 참조, x 소멸

        int& k4 = func_4(n);
                // int& k4 = x (=n);  // k4는 인자 n을 참조

        cout << k1 << " " << k3 << " " << k4 << endl;
}
```

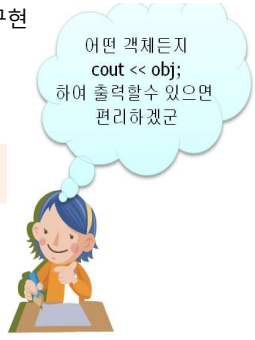
- 출력 → 11, xxx(쓰레기), 11
- 함수 안의 지역변수는 참조자로 반환하면 안됨

- 매개변수로 전달된 변수(객체) 그 자체를 반환하는 경우
  - 함수의 해당 매개변수는 참조자로 받아야 함.
  - 매개변수로 받은 참조자를 반환해야 함.

<< 연산자의 중복

- 현재 Vector 객체 출력시 display( ) 함수 호출하여 사용
- "cout << v;" 와 같이 "<<" 연산자 사용 출력하면 편리, 다른 데이터형과 일관성 유지
- cout << x; 에서 (int x=1)
  - cout 은 ostream 클래스(출력 기능의 클래스)의 객체
  - "<<" 는 연산자 → "<<" 는 ostream 클래스 멤버함수로 구현
    - operator<<(...) 형식으로 함수 작성

```
Vector v(2, 3);
cout << v;              //화면에 (2, 3)이 출력된다.
```



"<<" 연산자

- 다음이 가능하게 << 연산자를 중복 정의 필요.

```
Point pos(1, 2);
cout << pos;
```

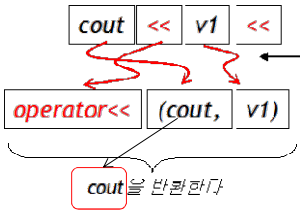
```
Vector a(2, 3);
cout << a;
```

- cout 은 ostream 클래스의 객체 → iostream.h 에 정의
  - class 안에 옆 코드와 같이 << 연산자(함수)들 중복 정의되어 있음.
  - << 함수들은 ostream 클래스의 멤버함수
  - 문자열, 정수, 실수 등 다양한 type 출력 위한 << 연산자 중복 정의
- 위와 같이 cout << pos; 출력하려면
  - 멤버함수로 구현하려면 ostream 클래스에 구현해야 함.
    - but, ostream 클래스에 Point, Vector 객체 출력하는 새로운 멤버함수 추가 불가 → 기존 설치내용 변경 불가
  - 따라서 일반 함수로 구현해야 함.
- ostream 클래스의 멤버 함수들은 모두 ostream 클래스의 참조자를 반환
  - 새로 작성하는 출력 함수들도 ostream 클래스의 참조자 반환

```
class ostream
{
public:
    ostream& operator<< (char * str)
    {
        printf("%s", str);
        return *this;
    }
    ostream& operator<< (char str)
    {
        printf("%c", str);
        return *this;
    }
    ostream& operator<< (int num)
    {
        printf("%d", num);
        return *this;
    }
    ostream& operator<< (double e)
    {
        printf("%g", e);
        return *this;
    }
    ostream& operator<< (ostream& (*fp)(ostream &ostm))
    {
        return fp(*this);
    }
};
ostream& endl(ostream &ostm)
{
    ostm<<'\n';
    fflush(stdout);
    return ostm;
}
```

<< 연산자의 중복

- "<<" 연산자는 일반 함수로 작성
- cout 은 ostream 클래스의 객체



- cout << a << b; → ((cout << a) << b);
- (cout << a); 수행하고 cout << b; 를 수행, "<<" 함수는 cout 을 반환

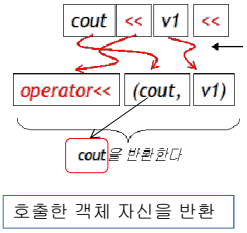
- << 함수 작성시 주의 :
  - << 연산을 수행한 후에 인자로 받은 cout(ostream 클래스 객체)를 반환하여야 함

  → 함수의 인자로 cout 을 참조자로 받고 cout 을 참조자로 반환해야함.

<< 연산자 중복

- 다음 형태의 전역함수(friend 함수) 작성 → #include <iostream> 필요(ostream 클래스 정의)

```
Vector a(2.0, 3.0);
cout << a;          // operator<<(cout, a); 와 동일
...
ostream& operator<<(ostream& os, const Vector& v){
        os << "(" << v.x << ", " << v.y << ")" << endl;
        return os;
}
```

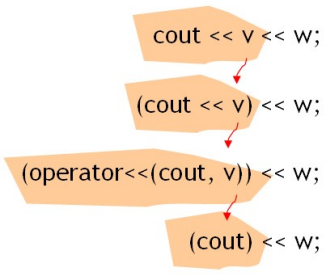


- cout << a; → operator<<(cout , a) 형태로 해석

- 1st 매개변수로 받은 cout 객체 자신 반환해야 cout << a << b; 가능
  - cout << a << b; → ((cout << a) << b); → (cout << b;)
  - (cout << a); 을 수행하고 cout << b; 를 수행

<< 연산자 구현 주의할 점

- 전역 함수(friend 함수) 형태만 사용 가능: 개발자가 기존의 ostream 클래스를 수정할 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(){  // 불필요,
        cout << "(" << x << ", " << y << ")" << endl;
   }
};
```

<< 연산자의 중복

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

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

<< 연산자 함수 정의

두번 째 매개변수 const

- cout 을 참조자로 받고 참조자로 반환

실행결과

```
(1,2)
(3,4)
(0,0)
```

>> 연산자의 중복

- 입력 연산자 >> 의 중복
  - cin 은 istream 클래스의 객체
  - cin >> v1 >> v2; 가능해야 함 → ">>" 연산자 반환값은 cin
    - (cin >> v1) >> v2 → cin >> v2;
- cout 과 같이 friend 함수로만 작성 가능
- cin >> a; → operator>>(cin , a) 형태로 함수 작성
- Vector 클래스인 경우 operator>>(cin , a) 함수에서 a.x, a.y 2개를 한번에 입력 받음.

>> 연산자의 중복

- cin >> a; → operator>>(cin , a) 형태로 함수 작성
  - 함수 작성시 두번 째 매개변수는 const 하면 안됨.(<< 연산자는 const)
    - 입력 값을 두번 째 매개변수에 저장하기 때문
- 입력 오류인 경우, 오류 처리를 하는 것이 좋음
  - v.x, v.y 는 정수 → 문자가 입력되면 ? → 오류 처리

```
istream& operator>>(istream& in, Vector& v)
{
   in >> v.x >> v.y;
   if(!in)
        v = Vector(0, 0);
   return in;
}
```

입력 오류 처리

- Vector 클래스의 v.x, v.y 는 int
- 입력시 문자가 들어오면 오류 → 입력이 v.x, v.y 에 저장되지 않고 in 이 0 값 가짐

“<<“ , “>>” 연산자 사용 예제

```
class Vector {
private:
   double x, y;
public:
   Vector(double x, double y){
         this->x = x;
         this->y = y;
   }
   friend istream& operator>>(istream& is,
                        Vector &v);
   friend ostream& operator<<(ostream& os,
                        const Vector &v);
};

istream& operator>>(istream& is, Vector &v){
   is >> v.x >> v.y;
   if( !is )   // 입력 오류인 경우
         v = Vector(0, 0);
   return  is;
}
```

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

int main() {

   Vector v1(1.0, 2.0);
   cin >> v1;
   cout << v1;
   return 0;
}
```