연산자 오버로딩1

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

연산자 오버로딩의 이해와 유형

연산자 오버로딩 (operator overloading)

- C++에서는 기존의 C 언어에서 제공하고 있는 연산자에 대하여 그 의미를 다시 부여하는 것을 "연산자 오버로딩" 또는 "연산자 중복(재정의)"라 함
  - 연산자 오버로딩은 기본적으로 함수의 오버로딩과 같이 연산자도 하나의 함수라는 개념을 사용하여 중복 정의
  - 중복되는 연산자 함수는 클래스의 멤버함수나 프렌드 함수로 정의
  - 함수 이름 대신에 operator 키워드를 사용하고 다음에 연산자를 기술

```
반환형 operator 연산자(가인수 리스트);
```

- 두 가지 형태로 표현
  - 멤버함수에 의한 오버로딩
  - 전역함수에 의한 오버로딩, friend 함수 선언

연산자 오버로딩 정의 예

- 복소수형 클래스 + 연산자 오버로딩 정의 예
  - operator + 는 함수와 같이 간주
  - c = a + b는 c = a.operator + (b); 로 기술할 수도 있음
    - 객체 a의 멤버함수 operator + 를 호출하고 인수 b를 전달

```
class Complex {
.....
// + 연산자 오버로딩
Complex operator +(Complex x); // 멤버함수
// friend Complex operator+(Complex x,Complex y); // 전역함수
.....
};
Complex a, b, c;
c = a + b; // 중복된 연산자를 사용한 복소수 덧셈 연산 표현
```

멤버함수 연산자 오버로딩 예

```
#include <iostream>
using std::endl;
using std::cout;
class Point {
      int x, y;
public:
      Point(int _x=0, int _y=0)   { x = _x; y = _y; }
      void ShowPosition();
      Point operator+(const Point& p);
};
void Point::ShowPosition()
{
      cout<<x<<" "<<y<<endl;
}
Point Point::operator+(const Point& p)
{
      Point temp;
      temp.x = x + p.x;
      temp.y = y + p.y;
      return temp;
}
int main(void)
{
      Point p1(1, 2);
      Point p2(2, 1);
      Point p3=p1+p2;   // p3 = p1.operator+(p2);
      p3.ShowPosition();
      return 0;
}
```

```
#include <iostream>
using std::endl;
using std::cout;
class Point {
      int x, y;
public:
      Point(int _x=0, int _y=0) : x(_x), y(_y) { }
      void ShowPosition();
      Point operator+(const Point& p);
};
void Point::ShowPosition()
{
      cout<<x<<" "<<y<<endl;
}
Point Point::operator+(const Point& p)
{
      Point temp(x+p.x, y+p.y);
      return temp;
}

int main(void)
{
      Point p1(1, 2);
      Point p2(2, 1);
      Point p3=p1+p2;   // p3 = p1.operator+(p2);
      p3.ShowPosition();
      return 0;
}
```

전역함수 연산자 오버로딩 예

```
#include <iostream>
using std::endl;
using std::cout;

class Point {
	int x, y;
public:
	Point(int _x=0, int _y=0):x(_x), y(_y){}
	void ShowPosition();
	friend Point operator+(const Point& p1, const Point& p2);
};
void Point::ShowPosition(){
	cout<<x<<" "<<y<<endl;
}

Point operator+(const Point& p1, const Point& p2)
{
	Point temp(p1.x+p2.x, p1.y+p2.y);
	return temp;
}
int main(void)
{
	Point p1(1, 2);
	Point p2(2, 1);
	Point p3=p1+p2;   // p3 = operator+(p1,p2)
	p3.ShowPosition();

	return 0;
}
```

복소수 연산 예 (1)

```
#include <iostream>
using std::endl;
using std::cout;

class Complex {
   double re, im;
public:
    Complex(double r, double i)
                          { re = r; im = i; }
    Complex()            { re = 0; im = 0; }
    Complex operator +(const Complex& c);
                          // + 연산자 중복
    Complex operator -(const Complex& c);
                          // - 연산자 중복
     void show()
          { cout << re << " + i" << im << '₩n'; }
};

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

```
Complex Complex::operator -(const Complex& c)
{
   Complex tmp;

   tmp.re = re - c.re;
   tmp.im = im - c.im;
   return tmp;
}

int main()
{
   Complex a(1.2,10.6), b(2.3, 5), c(2.0,4.4);
   Complex sum, dif;

   sum = a + b;          // 연산자 함수 + 호출
   cout << "a + b = "; sum.show();
   dif = a - b;          // 연산자 함수 - 호출
   cout << "a - b = "; dif.show();

   sum = a + b + c;   // 연산자 함수 + 호출
   cout << "a + b + c = "; sum.show();

   return 0;
}
```

복소수 연산 예 (2)

```
a + b = 3.5 + i15.6
a - b = -1.1 + i5.6
a + b + c = 5.5 + i20
```

복소수 덧셈, 뺄셈을 위해 연산자 +, -를 중복 정의하였다.

각 연산자 함수가 Complex 형의 객체를 반환하므로

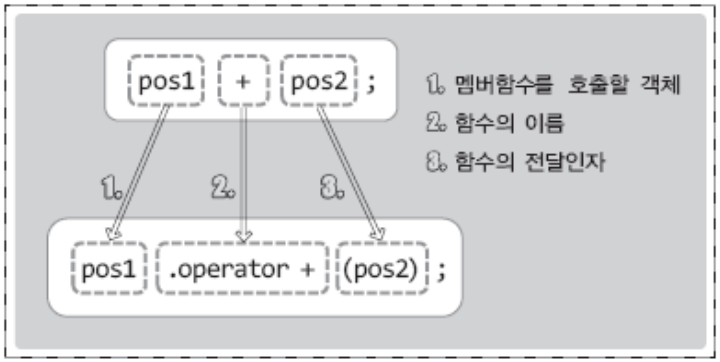
a + b + c 와 같이 연속적인 연산이 가능하다. 각 연산자는

다음과 같이 연산자 함수를 호출한다.

```
a + b      =>  a.operator +(b)
a - b      =>  a.operator -(b)
a + b + c =>  a.operator +(b).operator +(c)
```

operator+ 라는 이름의 함수

```
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;
}
```

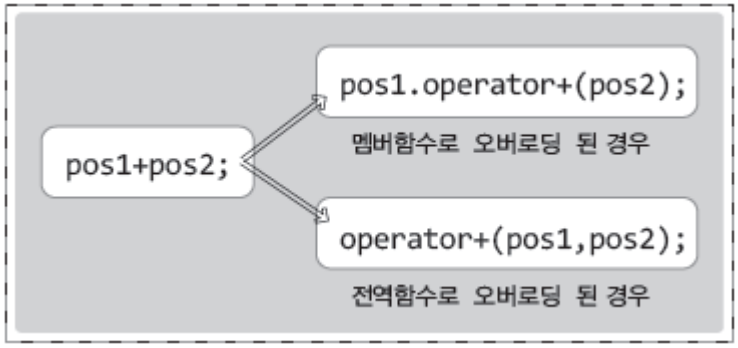
```
int main(void)
{
    Point pos1(3, 4);
    Point pos2(10, 20);
    Point pos3=pos1+pos2;
    pos1.ShowPosition();
    pos2.ShowPosition();
    pos3.ShowPosition();
    return 0;
}
```

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

실행결과

연산자 오버로딩에서 이야기하는 함수호출의 규칙을 이해하는 것이 중요!

연산자를 오버로딩 하는 두 가지 방법



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;
}
```

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!

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

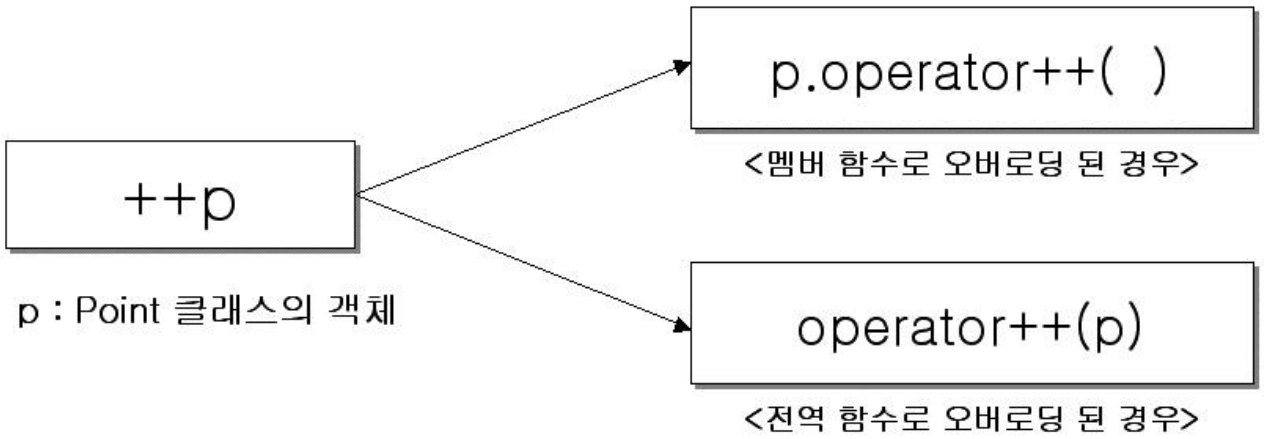
단항 연산자 오버로딩

단항 연산자 중복

- ++, --, ~, ! 등과 같은 단항 연산자(unary operator)도 이항 연산자와 같이 중복 정의

- 멤버함수에 의한 오버로딩
  - 단항 연산자의 경우 하나의 오퍼랜드는 연산자 함수를 정의하는 객체가 되므로 연산자 멤버함수의 인수가 없게 됨

- 전역함수에 의한 오버로딩
  - 연산자 전역함수에 하나의 오퍼랜드가 전달



단항 연산자 예 1

```
#include <iostream>
using std::endl;
using std::cout;
class Point {
private:
	int x, y;
public:
	Point(int _x=0, int _y=0):x(_x), y(_y){}
	void ShowPosition();
	Point& operator++();
	friend Point& operator--(Point& p);
};
void Point::ShowPosition(){
	cout<<x<<" "<<y<<endl;
}
Point& Point::operator++()
{
	x++;  y++;
	return *this;
}
Point& operator--(Point& p)
{
	p.x--;    p.y--;
	return p;
}
```

```
int main(void)
{
	Point p(1, 2);
	++p;   //p의 x, y 값을 1씩 증가.
		// p.operator++()
	p.ShowPosition();  //2, 3

	--p;    //p의 x, y 값을 1씩 감소.
		// operator--(p)
	p.ShowPosition();  //1, 2

	++(++p);
	// (p.operator++()).operator++()

	p.ShowPosition();  //3, 4

	--(--p);
	// operator--(operator--(p))
	p.ShowPosition();  //1, 2

	return 0;
}
```

단항 연산자 예 2

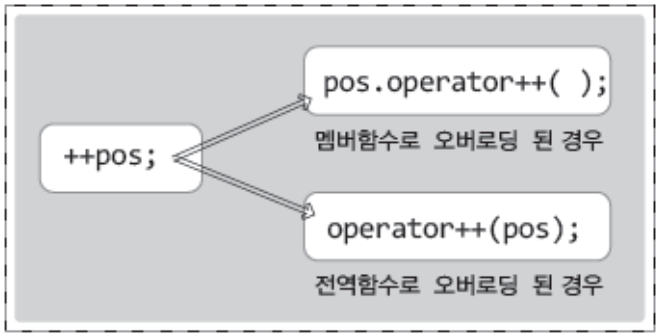
```
#include <iostream>
using std::endl;
using std::cout;

class Point {
	int x, y;
public:
	Point(int _x=0, int _y=0) : x(_x), y(_y) { }
	void ShowPosition();
	Point operator-(const Point& p);   // 이항 연산자 -
	Point operator-();         // 단항연산자 -
};
void Point::ShowPosition()
{
	cout<<x<<" "<<y<<endl;
}

Point Point::operator-(const Point& p)
{
	Point temp(x-p.x, y-p.y);
	return temp;
}

Point Point::operator-()
{
	Point temp(-x, -y);
	return temp;
}
```

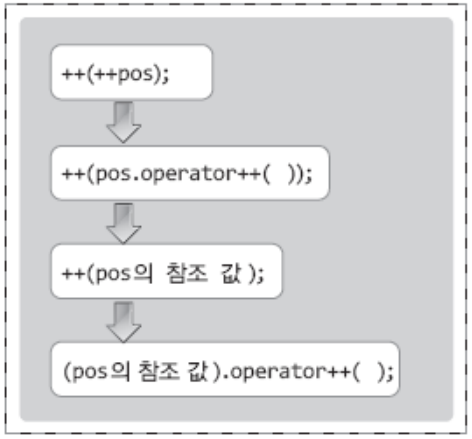
```
int main(void)
{
	Point p1(3, 8);
	Point p2(2, 1);

	Point p3=p1-p2;   // p3 = p1.operator-(p2);
	p3.ShowPosition();

	Point p4=-p1;   // p4 = p1.operator-();
	p4.ShowPosition();

	return 0;
}
```

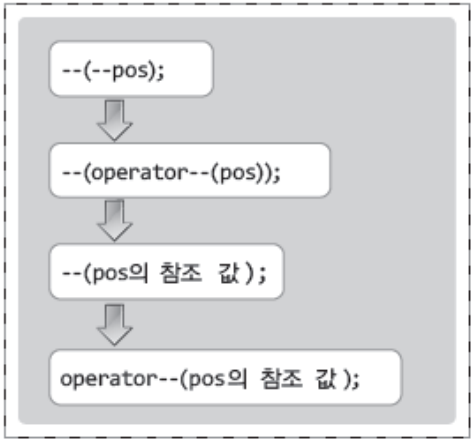
증가, 감소 연산자의 오버로딩

```
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++()
    {
        xpos+=1;
        ypos+=1;
        return *this;
    }
    friend Point& operator--(Point &ref);
};

Point& operator--(Point &ref)
{
    ref.xpos-=1;
    ref.ypos-=1;
    return ref;
}
```

```
int main(void)
{
    Point pos(1, 2);
    ++pos;
    pos.ShowPosition();
    --pos;
    pos.ShowPosition();
    ++(++pos);
    pos.ShowPosition();
    --(--pos);
    pos.ShowPosition();
    return 0;
}
```







전위증가와 후위증가의 구분

```
++pos      →      pos.operator++();
pos++      →      pos.operator++(int);
```

```
const Point operator++(int)    // 후위증가
{
    const Point retobj(xpos, ypos);
    xpos+=1;
    ypos+=1;
    return retobj;
}
```

멤버함수 형태의 후위 증가

```
--pos      →      pos.operator--();
pos--      →      pos.operator--(int);
```

```
const Point operator--(Point &ref, int)    // 후위감소
{
    const Point retobj(ref);    // const 객체라 한다.
    ref.xpos-=1;
    ref.ypos-=1;
    return retobj;
}
```

전역함수 형태의 후위 감소

단항 연산자의 오버로딩

```
class Point {
private:
    int x, y;
public:
    Point(int _x=0, int _y=0):x(_x), y(_y){}
    void ShowPosition();
    Point& operator++();
    Point operator++(int);
};
void Point::ShowPosition(){
    cout<<x<<" "<<y<<endl;
}

Point& Point::operator++(){
    x++;
    y++;
    return *this;
}
Point Point::operator++(int){
    Point temp(x, y);  // Point temp(*this);
    x++;
    y++;
    return temp;
}
```

```
int main(void)
{
    Point p1(1, 2);
    (p1++).ShowPosition();     //1, 2
    p1.ShowPosition();          //2, 3

    Point p2(1, 2);
    (++p2).ShowPosition();     //2, 3
    return 0;
}
```

반환형에서의 const 선언과 const 객체

```
int main(void)
{
    const Point pos(3, 4);
    const Point &ref=pos;     // 컴파일 OK!
    . . . .
}
```

const 객체는 멤버변수의 변경이 불가능한 객체!

const 객체는 const 참조자로만 참조가 가능하다.

const 객체를 대상으로는 const 함수만 호출 가능하다.

```
const Point operator++(int)
{
    const Point retobj(xpos, ypos);
    xpos+=1;
    ypos+=1;
    return retobj;
}
```

반환형이 const란 의미는 반환되는 객체를 const 객체화 하겠다는 의미!

따라서 반환되는 객체를 대상으로 const로 선언되지 않은 함수의 호출이 불가능하다.

본론으로 돌아와서

```
const Point operator++(int)
{
    const Point retobj(xpos, ypos);
    xpos+=1;
    ypos+=1;
    return retobj;
}

const Point operator--(Point &ref, int)
{
    const Point retobj(ref);
    ref.xpos-=1;
    ref.ypos-=1;
    return retobj;
}
```

후위 증가 및 감소연산을 대상으로 반환형을 const로 선언한 이유는?

아래와 같이 C++이 허용하지 않는 연산의 컴파일을 허용하지 않기 위해서

```
int main(void)
{
    Point pos(3, 5);
    (pos++)++;     // 컴파일 Error!
    (pos--)--;     // 컴파일 Error!
    . . . .
}
```

(pos++)++; → (Point형 const 임시객체)++; → (Point형 const 임시객체).operator++();

(pos--)--; → (Point형 const 임시객체)--; → operator--(Point형 const 임시객체);

결국! 컴파일 에러

교환법칙 문제의 해결

자료형이 다른 두 피연산자를 대상으로 하는 연산

```
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*(int times)
    {
        Point pos(xpos*times, ypos*times);
        return pos;
    }
};
```

* 연산자는 교환법칙이 성립한다.

따라서 pos와 cpy가 point 객체라 할 때 다음 두 연산은 모두 허용이 되어야 하며, 그 결과도 같아야 한다.

```
cpy = pos * 3;
cpy = 3 * pos;
```

그러나 왼편의 클래스는 * 연산에 대해서 교환법칙을 지원하지 않는다.

교환법칙의 성립을 위한 구현

문제의 요는 다음 연산이 가능하게 하는 것! 이는 전역함수의 형태로 오버로딩 할 수밖에 없는 상황

```
cpy = 3 * pos;
```

```
Point operator*(int times, Point& ref)
{
    Point pos(ref.xpos*times, ref.ypos*times);
    return pos;
}
```

```
Point operator*(int times, Point& ref)
{
    return ref*times;
}
```

3 * pos를 pos * 3 의 형태로 바꾸는 방식

CommuMultipleOperation.cpp

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

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*(int times)
    {
        Point pos(xpos*times, ypos*times);
        return pos;
    }
    friend Point operator*(int times, Point & ref);
};

Point operator*(int times, Point & ref)
{
    return ref*times;
}

int main(void)
{
    Point pos(1, 2);
    Point cpy;

    cpy=3*pos;
    cpy.ShowPosition();

    cpy=2*pos*3;
    cpy.ShowPosition();
    return 0;
}
```

cout, cin 그리고 endl의 정체

입출력 연산자 개요

- C++의 입출력 연산자 Cout <<, Cin >> 도 연산자 오버로딩에 의해 구현된 것
  - 이름영역 std의 클래스 ostream, istream 클래스에서 정의
  - 미리 정의된 자료형에 대해 입출력

- 입출력 연산자 <<, >> 로 사용자 정의 객체에 적용하려면 아래와 같은 전역함수(friend 함수) 정의

```
ostream& operator<<(ostream& os, class ob)
{
    ……
    return os;
}
```

```
istream& operator>>(istream& is, class ob)
{
    ……
    return is;
}
```

cout과 endl 이해하기

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

이름공간 mystd 안에 선언되었다고 가정!

예제에서 cout과 endl을 흉내내었으니, 예제의 분석을 통해서 이 둘의 실체를 이해할 수 있다.

```
int main(void)
{
    using mystd::cout;
    using mystd::endl;

    cout<<"Simple String";
    cout<<endl;
    cout<<3.14;
    cout<<endl;
    cout<<123;
    endl(cout);
    return 0;
}
```

실행결과

```
Simple String
3.14
123
```

```
cout.operator<<("Simple String");
cout.operator<<(3.14);
cout.operator<<(123);

cout.operator<<(endl);
```

cout<<123<<endl<<3.14<<endl;

```
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;
}
```

***this 를 반환함으로써, 연이은 오버로딩 함수의 호출이 가능해진다.**

cout<<123<<endl<<3.14<<endl;

<<, >> 연산자의 오버로딩

```
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 ostream& operator<<(ostream&, const Point&);
};

ostream& operator<<(ostream& os, const Point& pos)
{
    os<<'['<<pos.xpos<<", "<<pos.ypos<<']'<<endl;
    return os;
}
```

```
int main(void)
{
    Point pos1(1, 3);
    cout<<pos1;
    Point pos2(101, 303);
    cout<<pos2;
    return 0;
}
```

실행결과

```
[1, 3]
[101, 303]
```

Point 클래스를 대상으로 하는 << 연산자의 오버로딩 사례를 보인다!

Q & A