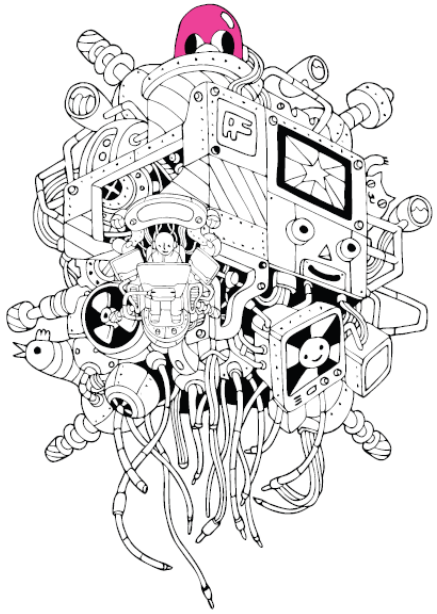
윤성우의
열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 10. 연산자 오버로딩1

윤성우의

열혈 C++ 프로그래밍

Chapter 10-2. 단항 연산자 오버로딩

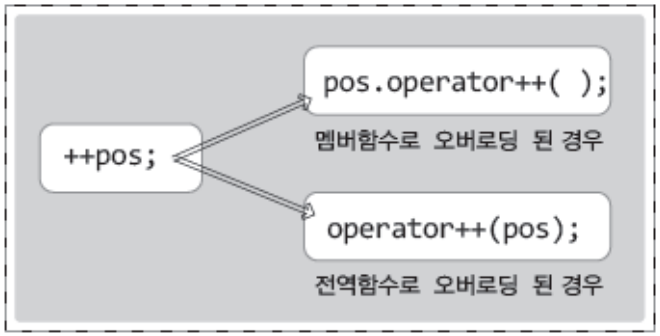
윤성우 저 열혈강의 C++ 프로그래밍 개정판

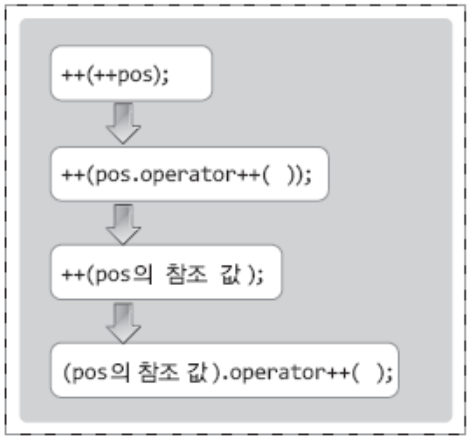
증가, 감소 연산자의 오버로딩

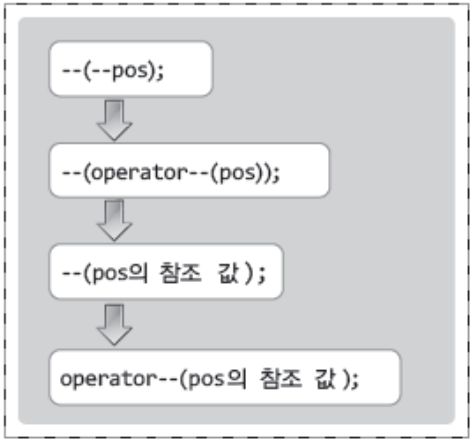

```
class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
    }
    Point& operator++()
    {
        xpos+=1;
        ypos+=1;
        return *this;
    }
    friend 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

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

윤성우의

열혈 C++ 프로그래밍

Chapter 10-3. 교환법칙 문제의 해결

윤성우 저 열혈강의 C++ 프로그래밍 개정판

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