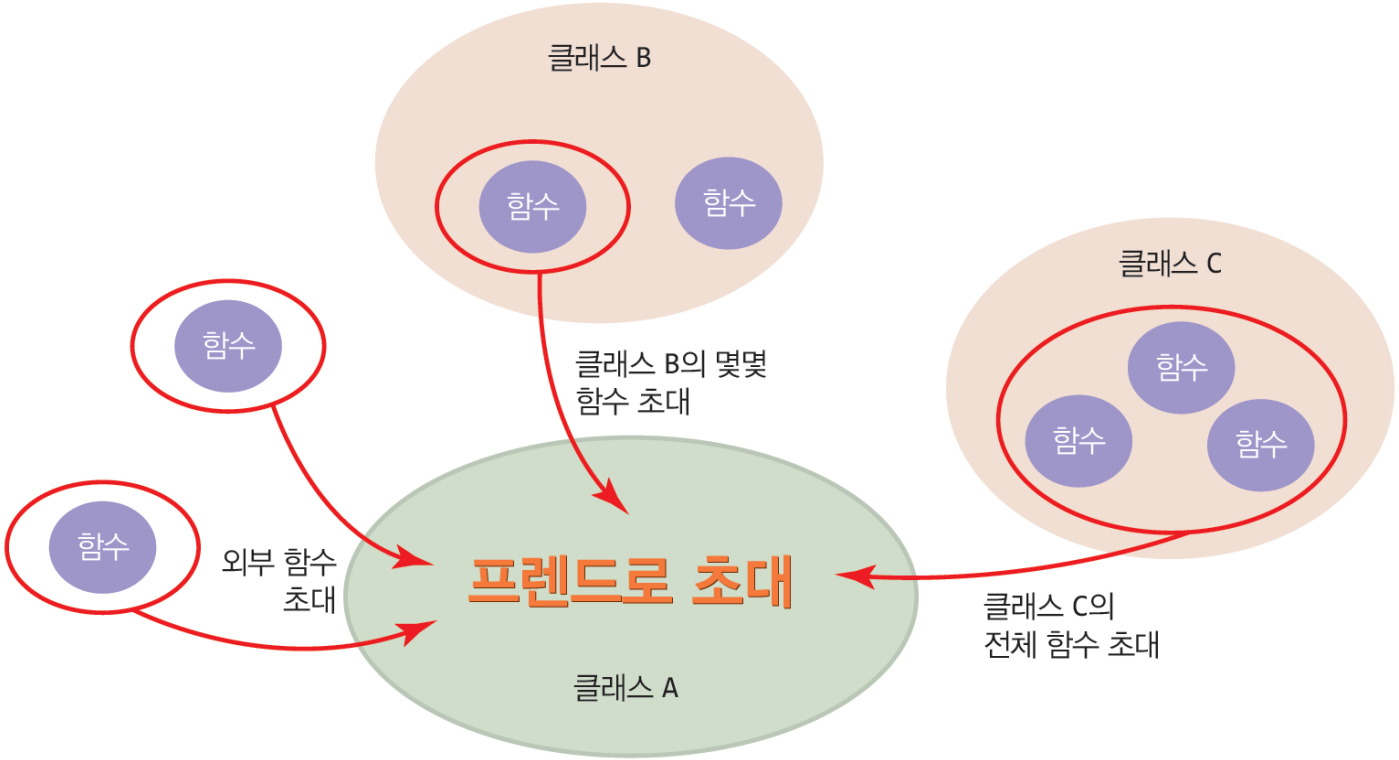
C++ 프렌드

- 프렌드 함수
  - 클래스의 멤버 함수가 아닌 외부 함수
    - 전역 함수
    - 다른 클래스의 멤버 함수
  - friend 키워드로 클래스 내에 선언된 함수
    - 클래스의 모든 멤버를 접근할 수 있는 권한 부여
    - 프렌드 함수라고 부름
  - 프렌드 선언의 필요성
    - 클래스의 멤버로 선언하기에는 무리가 있고, 클래스의 모든 멤버를 자유롭게 접근할 수 있는 일부 외부 함수 작성 시

항목	세상의 친구	프렌드 함수
존재	가족이 아님. 외부인	클래스 외부에 작성된 함수. 멤버가 아님
자격	가족의 구성원으로 인정받음. 가족의 모든 살림살이에 접근 허용	클래스의 멤버 자격 부여. 클래스의 모든 멤버에 대해 접근 가능
선언	친구라고 소개	클래스 내에 `friend` 키워드로 선언
개수	친구의 명수에 제한 없음	프렌드 함수 개수에 제한 없음

프렌드로 초대하는 3 가지 유형

- 프렌드 함수가 되는 3 가지
  - 전역 함수 : 클래스 외부에 선언된 전역 함수
  - 다른 클래스의 멤버 함수 : 다른 클래스의 특정 멤버 함수
  - 다른 클래스 전체 : 다른 클래스의 모든 멤버 함수



프렌드 선언 3 종류

1. 외부 함수 equals()를 Rect 클래스에 프렌드로 선언

```
class Rect { // Rect 클래스 선언
    ...
    friend bool equals(Rect r, Rect s);
};
```

2. RectManager 클래스의 equals() 멤버 함수를 Rect 클래스에 프렌드로 선언

```
class Rect {
    ..............
    friend bool RectManager::equals(Rect r, Rect s);
};
```

3. RectManager 클래스의 모든 멤버 함수를 Rect 클래스에 프렌드로 선언

```
class Rect {
    ..............
    friend RectManager;
};
```

연산자 오버로딩

- **클래스 안에서 연산자의 의미를 새롭게 부여**함으로서 **클래스의 오브젝트를 마치 기본형의 변수**처럼 다룰 수 있도록하는 기능이다. **연산자 중복정의는 연산자 함수를 통하여 구현**한다.
- **operator** 는 이함수가 연산자 함수임을 나타내는 키워드다.

멤버 함수에 의한 연산자 오버로딩 : 연산자 함수가 사용자가 정의한 **클래스 내의 멤버 함수**에 의해 정의되고 호출되는 것. **P1+p2; p1.Operator+(p2);**

전역 함수에 의한 연산자 오버로딩 : 연산자 함수가 **사용자가 정의한 전역함수에** 의해 정의되고 호출되는 것.(friend 함수 사용) 함수의 **첫 번째 인자가 *this가 될 수 없는 함수를** friend 함수로 구현한다. **P1+p2; Operator+(p1,p2);**

연산자 함수

- 연산자 함수 구현 방법 2 가지
  1. 클래스의 멤버 함수로 구현
  2. 외부 함수로 구현하고 클래스에 프렌드 함수로 선언
- 연산자 함수 형식

리턴타입 **operator**연산자(*매개변수리스트*);

연산자 오버로딩

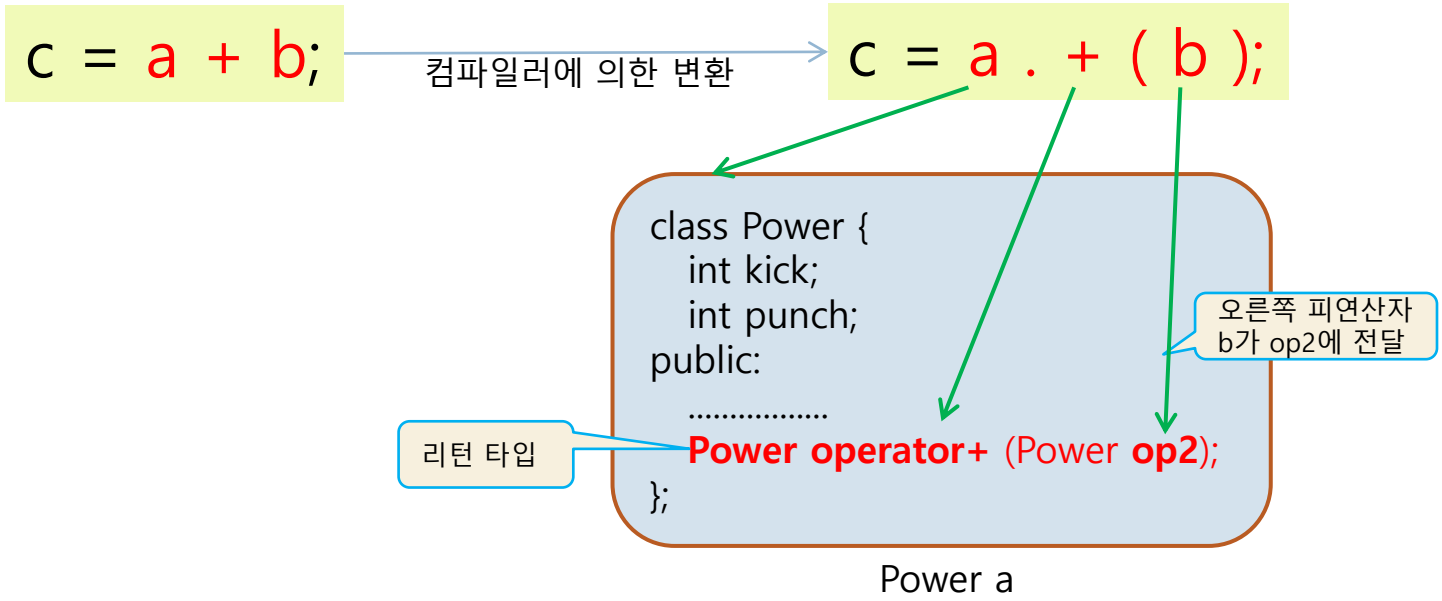
연산자 오버로딩의 주의 사항

- 연산자 우선 순위와 결합성은 유지된다.
- 디폴트 매개변수 설정이 불가능하다.

- 디폴트 연산자들의 기본 기능 변경 불가

```
int operator+(int a, int b)  // 정의 불가능한 함수
{
    return a+b+3;  //원래의 기능을 훼손한다,
}
```

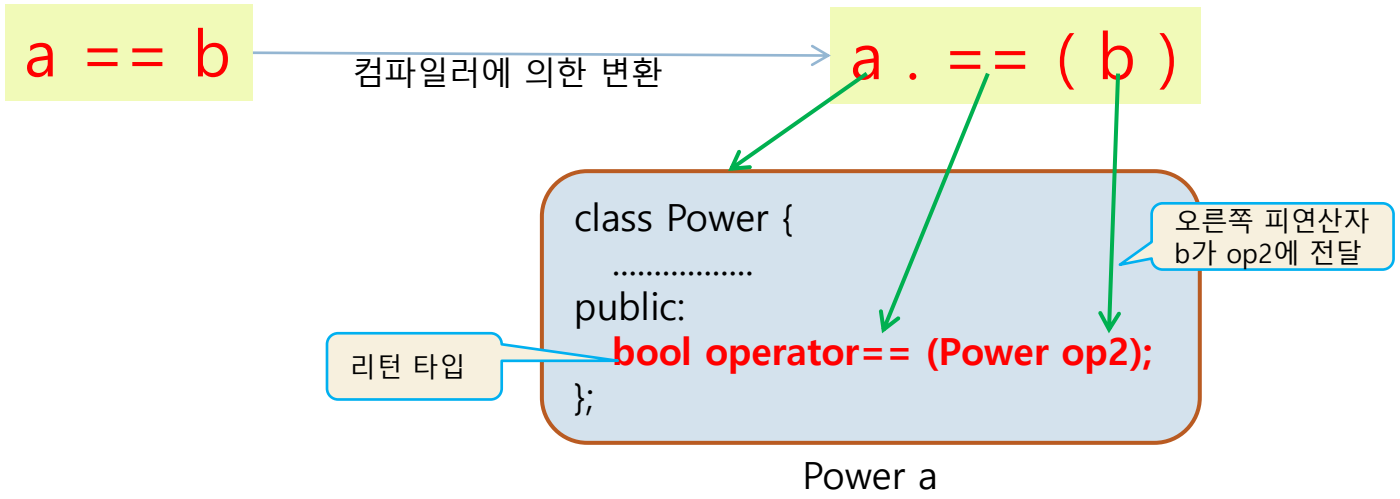
이항 연산자 중복 : + 연산자




```
Power Power::operator+(Power op2) {
    Power tmp;
    tmp.kick = this->kick + op2.kick;
    tmp.punch = this->punch + op2.punch;
    return tmp;
}
```

+ 연산자 함수 코드

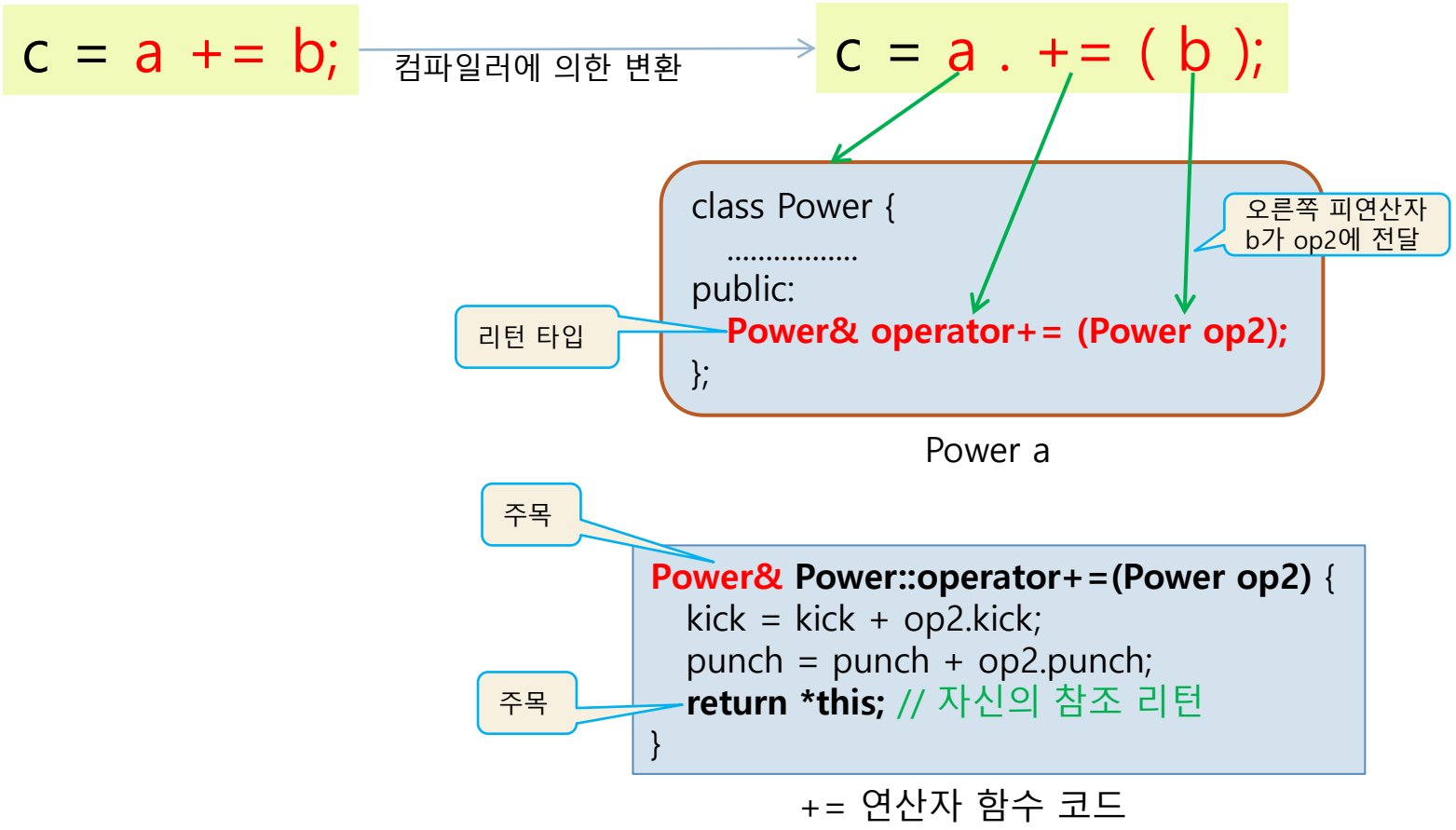
== 연산자 중복




```
bool Power::operator==(Power op2) {
   if(kick==op2.kick && punch==op2.punch)
      return true;
   else
      return false;
}
```

== 연산자 함수 코드

+= 연산자 중복

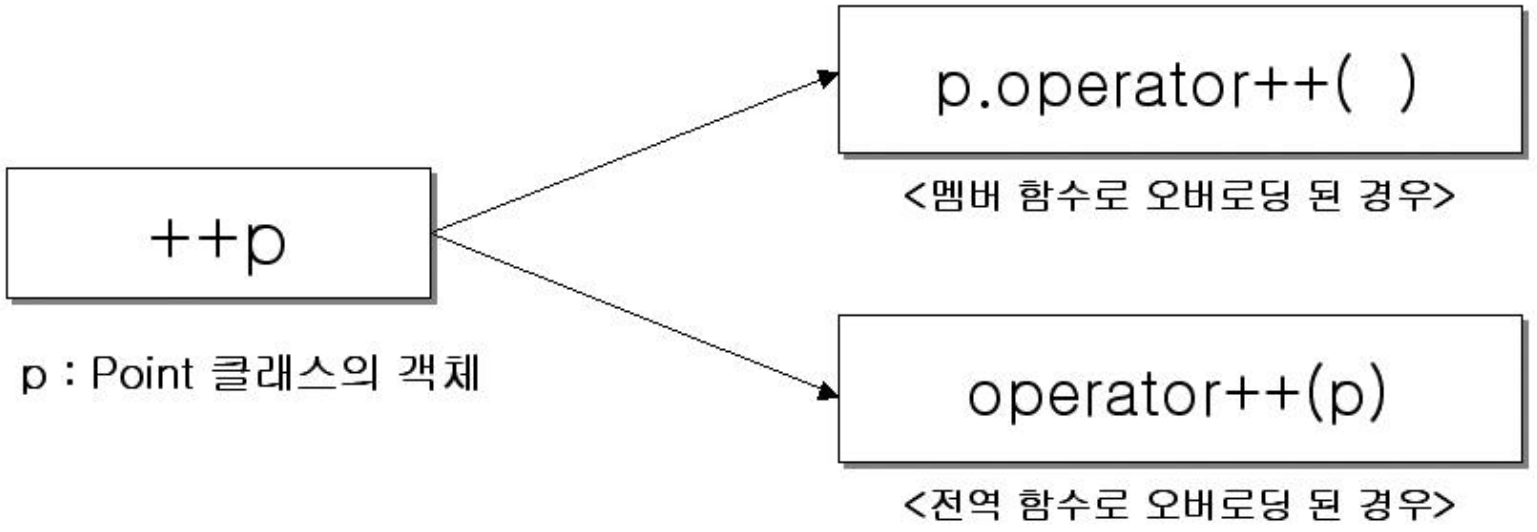

c = a += b;
컴파일러에 의한 변환
c = a . += (b);
class Power {
.....
public:
Power& operator+= (Power op2);
};
오른쪽 피연산자
b가 op2에 전달
리턴 타입
Power a
주목
Power& Power::operator+=(Power op2) {
kick = kick + op2.kick;
punch = punch + op2.punch;
return *this; // 자신의 참조 리턴
}
주목
+= 연산자 함수 코드

단항 연산자 오버로딩

- 단항 연산자
  - 피연산자가 하나 뿐인 연산자
    - 연산자 중복 방식은 이항 연산자의 경우와 거의 유사함
  - 단항 연산자 종류
    - 전위 연산자(prefix operator)
      - *!op, ~op, ++op, --op*
    - 후위 연산자(postfix operator)
      - *op++, op--*

단항 연산자의 오버로딩

- 증가, 감소 연산자 오버로딩



단항 연산자의 오버로딩

선 연산과 후 연산의 구분

연산자의 이름도 같고 피연산자의 수도 같기 때문에 동일하게 해석한다

```
p1++; =>  p1.operator++ ()
++p2; =>  p1.operator++ ()
```

++, -- 를 중복정의할 때에는 전위형과 후위형에 따라 연산자 함수를 다르게 만들어야 한다.

전위형 연산자 함수는 인자를 갖지 않지만, **후위형** 연산자 함수는 **int 형 인자**를 갖는다.
후위형 연산자 함수가 인수를 갖는 것은 전위형 연산자 함수와 구별하기 위함이지 실제 그 인수가 사용되는 것은 아니다.

단항 연산자의 오버로딩

- **선 연산과 후 연산의 구분**

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

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

연산자 중복의 특징

- C++에 본래 있는 연산자만 중복 가능
  - 3%%5 // 컴파일 오류
  - 6## 7 // 컴파일 오류
- 피 연산자 타입이 다른 새로운 연산 정의
- 연산자는 함수 형태로 구현 - 연산자 함수(operator function)
- 반드시 클래스와 관계를 가짐
- 피연산자의 개수를 바꿀 수 없음
- 연산의 우선 순위 변경 안됨
- 모든 연산자가 중복 가능하지 않음

중복 가능한 연산자

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^-	&=	|=
<<	>>	>>=	<<=	==	!=	>=
<=	&&	||	++	--	->*	,
->	[]	()	new	delete	new[]	delete[]

중복 불가능한 연산자

.	.*	::(범위지정 연산자)	? : (3항 연산자)

디폴트 대입 연산자란 ?

디폴트 대입 연산자 : 자신과 같은 클래형의 레퍼런스를 인자로 갖고 자신과 같은 클래스형의 레퍼런스를 리턴하는 연산자이다. 오브젝트가 대입될 때 호출된다.
(**사용자 정의 없으면 제공**되며, 멤버 대 멤버 대입이 일어난다)

```
Point p1(1, 5);
Point p2;
P2 = p1; // 디폴트 대입연산자 호출

Point& operator=(const Point& p)
{
        x=p.x;
        y=p.y;
        return *this;
}
```