



객체지향프로그래밍

Lecture 10 : friend와 연산자 중복

충북대 소프트웨어학부

이 태 겸(showm321@gmail.com)

본 강의노트는 아래의 자료를 기반으로 수정하여 제작된 것으로, 본 자료의 배포를 절대 금지합니다.

- 황기태. 명품 C++ Programming, 생능출판사

목차

❖ C++에서의 **friend**

❖ 연산자 중복(Operator overloading)

C++ friend

❖ friend 함수

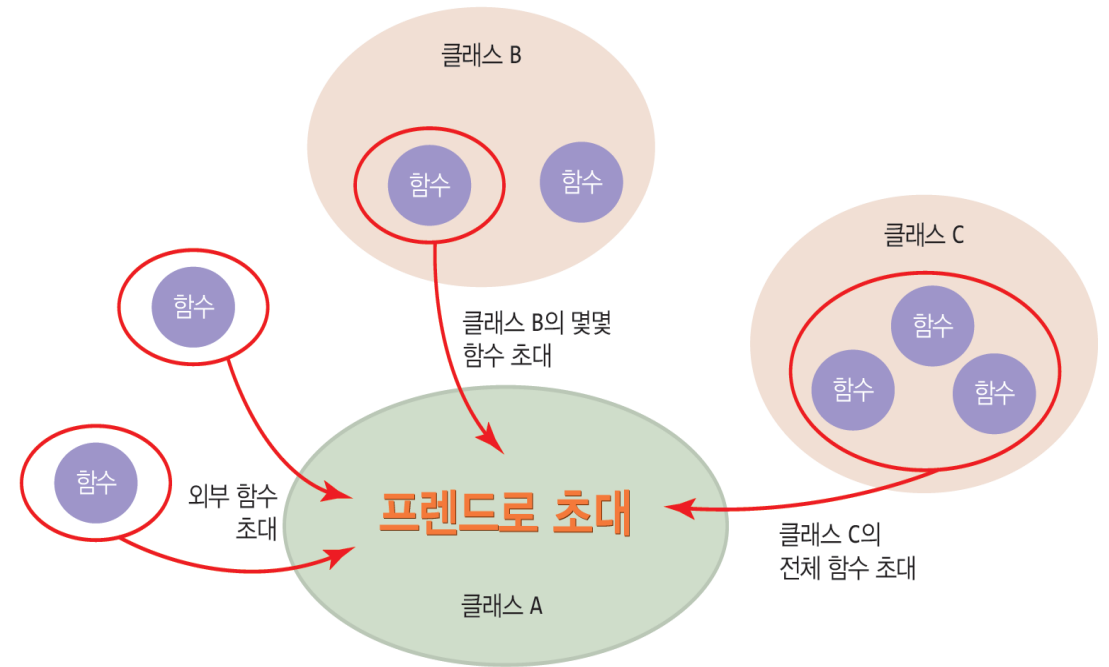
- 클래스의 멤버 함수가 아닌 외부 함수
 - 전역 함수, 다른 클래스의 멤버 함수
- friend 키워드로 클래스 내에 선언된 함수
 - 클래스 정의 안쪽에 **friend** 키워드를 쓰고 함수선언
 - 클래스의 모든 멤버를 접근할 수 있는 권한 부여
- friend 선언의 필요성
 - 외부에 선언된 함수가 클래스 내부의 멤버에 접근해야 하는 경우

```
class MyClass {  
private:  
    int value;  
  
public:  
    MyClass(int v) : value(v) {}  
  
    friend void printValue(const MyClass& obj); // friend 함수 선언  
};  
  
void printValue(const MyClass& obj) {  
    std::cout << "Value: " << obj.value << std::endl; // private 멤버 접근 가능  
}
```

friend로 초대하는 3가지 유형

❖ friend 함수가 되는 3가지

- 전역 함수 : 클래스 외부에 선언된 전역 함수
- 다른 클래스의 멤버 함수 : 다른 클래스의 특정 멤버 함수
- 다른 클래스 전체 : 다른 클래스의 모든 멤버 함수



friend 선언 3종류

1. 외부 함수 equals()를 Rect 클래스에 friend로 선언

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

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

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

3. RectManager 클래스의 모든 멤버 함수를 Rect 클래스에 friend로 선언

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

전역함수를 friend로 선언

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

Rect 클래스가 선언되기 전에 먼저 참조되는 컴파일 오류(forward reference)를 막기 위한 선언문

```
class Rect;
bool equals(Rect r, Rect s); // equals() 함수 선언
```

```
class Rect { // Rect 클래스 선언
```

```
    int width, height;
```

```
public:
```

```
    Rect(int width, int height) { this->width = width; this->height = height; }
```

```
    friend bool equals(Rect r, Rect s);
```

```
};
```

equals() 함수를
프렌드로 선언

```
bool equals(Rect r, Rect s) { // 외부 함수
```

```
    if (r.width == s.width && r.height == s.height) return true;
```

```
    else return false;
```

```
}
```

equals() 함수는 private 속성을 가진 width, height에 접근할 수 있다.

```
int main() {
```

```
    Rect a(3,4), b(4,5);
```

```
    if (equals(a, b)) cout << "equal" << endl;
```

```
    else cout << "not equal" << endl;
```

```
}
```

객체 a와 b는 동일한 크기의 사각형이므로 "not equal" 출력

not equal

다른 클래스의 멤버 함수를 friend로 선언

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

```
class Rect;
```

Rect 클래스가 선언되기 전에 먼저 참조되는 컴파일 오류(forward reference)를 막기 위한 선언문

```
class RectManager { // RectManager 클래스 선언
public:
    bool equals(Rect r, Rect s);
};
```

```
class Rect { // Rect 클래스 선언
    int width, height;
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend bool RectManager::equals(Rect r, Rect s);
};
```

```
bool RectManager::equals(Rect r, Rect s) {
    if (r.width == s.width && r.height == s.height) return true;
    else return false;
}
```

```
int main() {
    Rect a(3,4), b(3,4);
    RectManager man;

    if (man.equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
}
```

RectManager 클래스의 equals() 멤버를 프렌드로 선언

객체 a와 b는 동일한 크기의 사각형이므로 "equal" 출력

equal

다른 클래스 전체를 friend로 선언

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

```
class Rect;
```

Rect 클래스가 선언되기 전에 먼저 참조되는 컴파일 오류(forward reference)를 막기 위한 선언문

```
class RectManager { // RectManager 클래스 선언
```

```
public:
```

```
    bool equals(Rect r, Rect s);
```

```
    void copy(Rect& dest, Rect& src);
```

```
};
```

```
class Rect { // Rect 클래스 선언
```

```
    int width, height;
```

```
public:
```

```
    Rect(int width, int height) { this->width = width; this->height = height; }
```

```
    friend RectManager;
```

```
};
```

RectManager 클래스를 프렌드 함수로 선언

```
bool RectManager::equals(Rect r, Rect s) { // r과 s가 같으면 true 리턴
```

```
    if(r.width == s.width && r.height == s.height) return true;
```

```
    else return false;
```

```
}
```

```
void RectManager::copy(Rect& dest, Rect& src) { // src를 dest에 복사
```

```
    dest.width = src.width; dest.height = src.height;
```

```
}
```

```
int main() {
```

```
    Rect a(3,4), b(5,6);
```

```
    RectManager man;
```

객체 b의 width, height 값이 a와 같아진다.

```
    man.copy(b, a); // a를 b에 복사한다.
```

```
    if (man.equals(a, b)) cout << "equal" << endl;
```

```
    else cout << "not equal" << endl;
```

```
}
```

man.copy(b,a)를 통해 객체 b와 a의 크기가 동일하므로 "equal" 출력

equal

목차

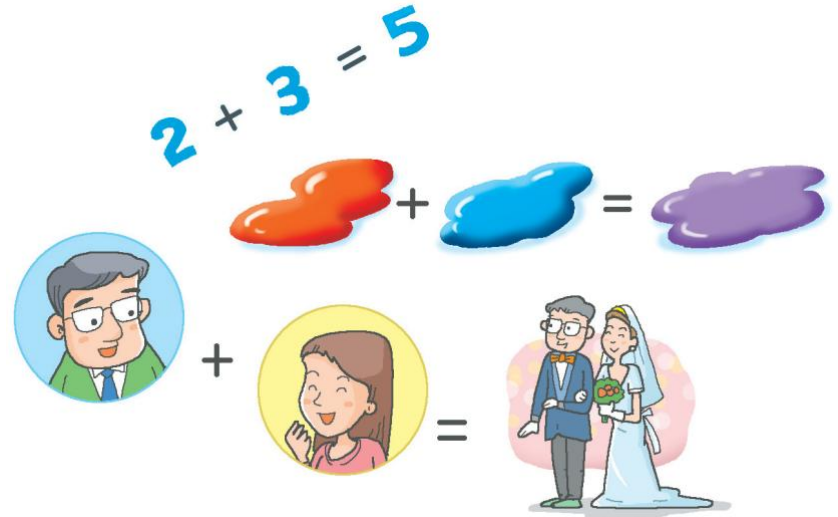
❖ C++에서의 friend

❖ 연산자 중복(**Operator overloading**)

연산자 중복(Operator overloading)

❖ 일상 생활에서의 기호 사용

- + 기호의 사례
 - 숫자 더하기 : $2 + 3 = 5$
 - 색 혼합 : 빨강 + 파랑 = 보라
 - 생활 : 남자 + 여자 = 부부
- + 기호를 숫자와 물체에 적용, 중복 사용
- + 기호를 숫자가 아닌 곳에도 사용
- 간결한 의미 전달



❖ C++ 언어에서도 연산자 중복 가능

- C++ 언어에 본래부터 있던 연산자에 새로운 의미 정의
- 높은 프로그램 가독성

연산자 중복의 사례 : + 연산자에 대해

❖ 정수 더하기

```
int a=2, b=3, c;  
c = a + b; // + 결과 5. 정수가 피연산자일 때 2와 3을 더하기
```

❖ 문자열 합치기

```
string a="C", c;  
c = a + "++"; // + 결과 "C++". 문자열이 피연산자일 때 두 개의 문자열 합치기
```

❖ 색 합치기

```
Color a(BLUE), b(RED), c;  
c = a + b; // c = VIOLET. a, b의 두 색을 섞은 새로운 Color 객체 c
```

❖ 배열 합치기

```
SortedArray a(2,5,9), b(3,7,10), c;  
c = a + b; // c = {2,3,5,7,9,10}. 정렬된 두 배열을 결합한(merge) 새로운 배열 생성
```

연산자 중복의 특징

❖ 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 불가능한 연산자

.	.*	::(범위지정 연산자)	? : (3항 연산자)
---	----	--------------	--------------

중복 가능한 연산자

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

연산자 함수

❖ 연산자 함수 구현 방법 2 가지

1. 클래스의 멤버 함수로 구현
2. 외부 함수로 구현하고 클래스에 friend 함수로 선언

❖ 연산자 함수 형식

리턴타입 **operator**연산자(매개변수리스트);

+와 == 연산자의 작성 사례

연산자 함수 작성에 필요한 코드 사례

```
Color a(BLUE), b(RED), c;  
  
c = a + b; // a와 b를 더하기 위한 + 연산자 작성 필요  
if (a == b) { // a와 b를 비교하기 위한 == 연산자 작성  
    필요  
    ...  
}
```

외부 함수로 구현되고
클래스에 friend로 선언되는 경우

```
Color operator+(Color op1, Color op2); // 외부 함수  
bool operator==(Color op1, Color op2); // 외부 함수  
  
class Color {  
    ...  
    friend Color operator+ (Color op1, Color op2);  
    friend bool operator== (Color op1, Color op2);  
};
```

클래스의 멤버 함수로 작성되는 경우

```
class Color {  
    ...  
    Color operator+(Color op2);  
    bool operator==(Color op2);  
};
```

앞으로 연산자 함수 작성에 사용할 클래스

```
class Power { // 에너지를 표현하는 파워 클래스
    int kick; // 발로 차는 힘
    int punch; // 주먹으로 치는 힘

public:
    Power(int kick=0, int punch=0) {
        this->kick = kick;
        this->punch = punch;
    }
};
```

멤버 함수로 이항 연산자 중복 구현1 : + 연산자 중복

```
class Power { // 에너지를 표현하는 파워 클래스
    int kick; // 발로 차는 힘
    int punch; // 주먹으로 치는 힘

public:
    Power(int kick=0, int punch=0) {
        this->kick = kick;
        this->punch = punch;
    }
};
```

```
int main() {
    Power a(3,5), b(4,6), c;
    c = a + b; // 파워 객체 + 연산
    a.show();
    b.show();
    c.show();
}
```

c = a + b;

컴파일러에 의한 변환

c = a . + (b);

```
class Power {
    int kick;
    int punch;
public:
    .....
    Power operator+ (Power op2);
};
```

리턴 타입

오른쪽 피연산자 b
가 op2에 전달

Power a

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

+ 연산자 함수 코드

2개의 Power 객체를 더하는 + 연산자 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator+(Power op2); // + 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
    << endl;
}

Power Power::operator+(Power op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = this->kick + op2.kick; // kick 더하기
    tmp.punch = this->punch + op2.punch; // punch 더하기
    return tmp; // 더한 결과 리턴
}
```

+ 연산자 멤버 함수 구현

```
int main() {
    Power a(3,5), b(4,6), c;
    c = a + b; // 파워 객체 + 연산
    a.show();
    b.show();
    c.show();
}
```

객체 a의 operator+() 멤버 함수 호출

```
kick=3,punch=5
kick=4,punch=6
kick=7,punch=11
```

객체 a, b, c
순으로 출력

멤버 함수로 이항 연산자 중복 구현2 : == 연산자 중복

a == b

컴파일러에 의한 변환

a . == (b)

```
int main() {  
    Power a(3,5), b(3,5); // 2 개의 동일한 파워 객체  
    a.show();  
    b.show();  
    if(a == b) cout << "두 파워가 같다." << endl;  
    else cout << "두 파워가 같지 않다." << endl;  
}
```

리턴 타입

```
class Power {  
    .....  
public:  
    bool operator==(Power op2);  
};
```

오른쪽 피연산자
b가 op2에 전달

Power a

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

== 연산자 함수 코드

2개의 Power 객체를 비교하는 == 연산자 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    bool operator==(Power op2); // == 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ','
         << "punch=" << punch << endl;
}

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

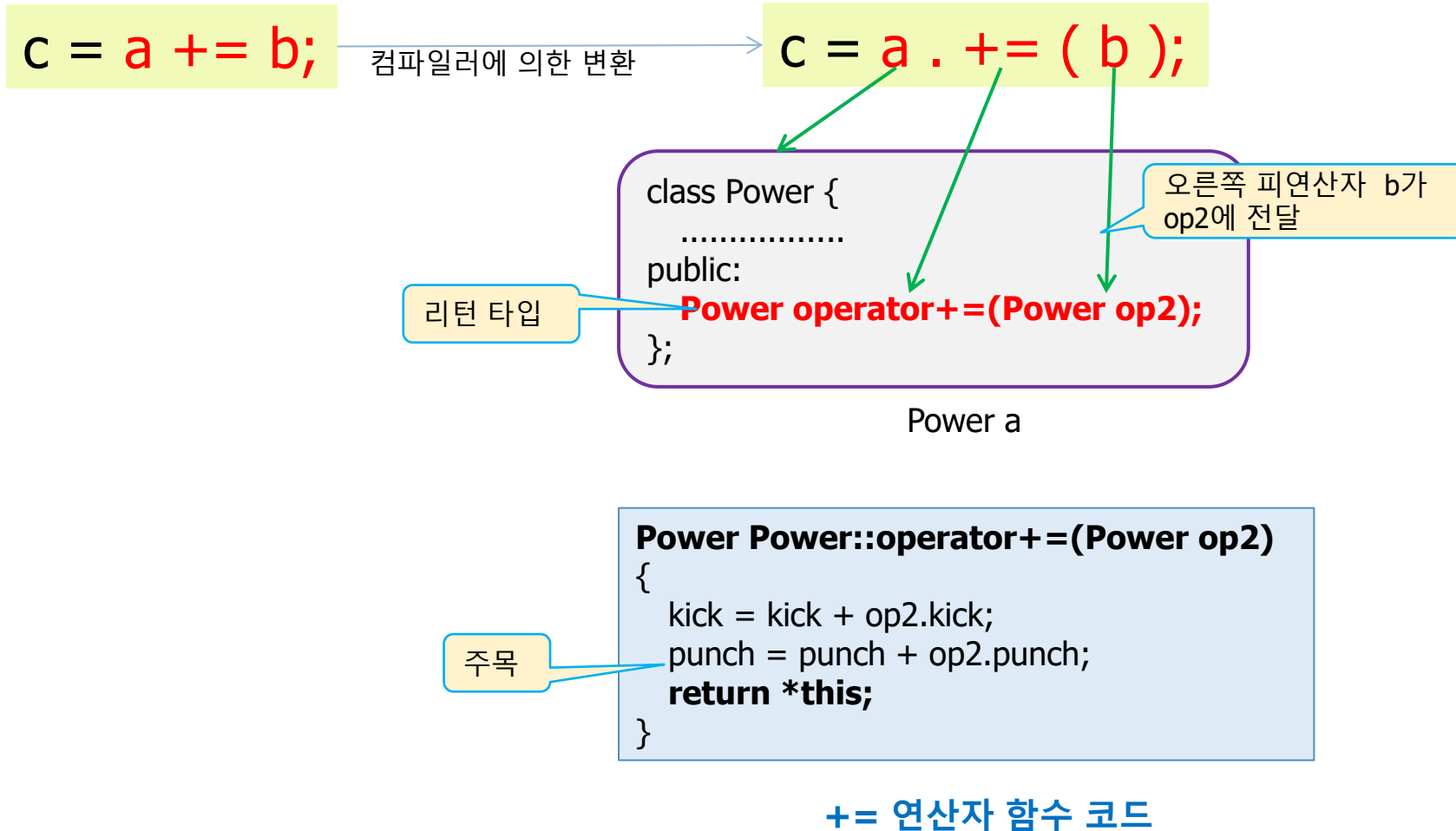
== 연산자 멤버 함수 구현

```
int main() {
    Power a(3,5), b(3,5); // 2 개의 동일한 파워 객체
    생성
    a.show();
    b.show();
    if(a == b) cout << "두 파워가 같다." << endl;
    else cout << "두 파워가 같지 않다." << endl;
}
```

operator==() 멤버 함수 호출

kick=3,punch=5
kick=3,punch=5
두 파워가 같다.

멤버 함수로 이항 연산자 중복 구현3 : += 연산자 중복



두 Power 객체를 더하는 += 연산자 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator+= (Power op2); // += 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
    << endl;
}

Power Power::operator+=(Power op2) {
    kick = kick + op2.kick; // kick 더하기
    punch = punch + op2.punch; // punch 더하기
    return *this; // 합한 결과 리턴
}
```

+= 연산자 멤버 함수 구현

```
int main() {
    Power a(3,5), b(4,6), c;
    a.show();
    b.show();
    c = a += b; // 파워 객체 더하기
    a.show();
    c.show();
}
```

operator+=() 멤버 함수 호출

```
kick=3,punch=5
kick=4,punch=6
kick=7,punch=11
kick=7,punch=11
```

a, b 출력

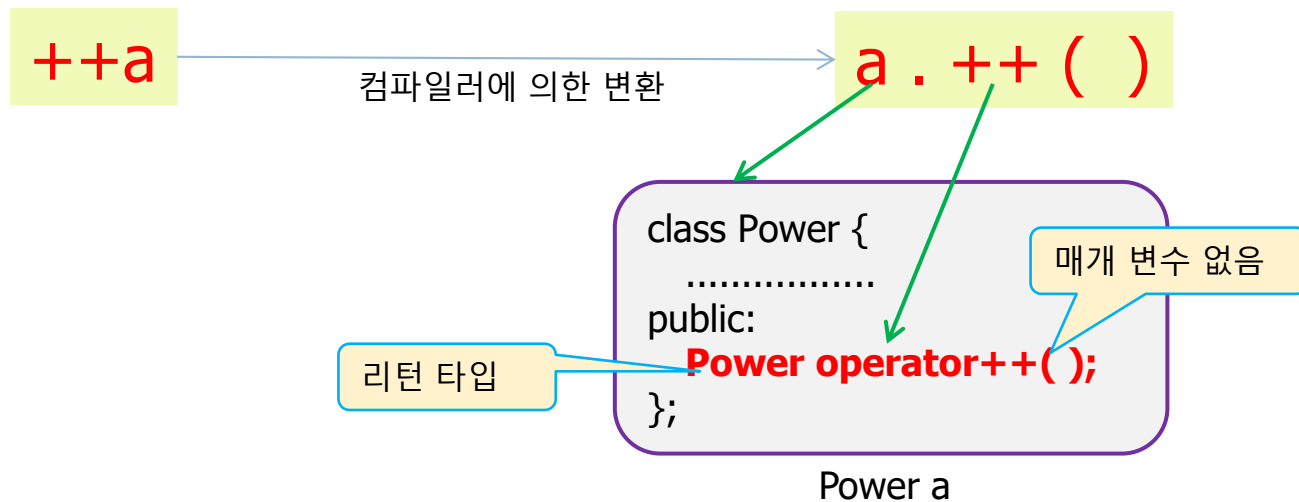
a+=b 후 a, c 출력

멤버 함수로 단항 연산자 중복 구현

❖ 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--

멤버 함수로 단항 연산자 중복 구현1 : 전위 ++ 연산자 중복



```
Power Power::operator++( ) {  
    // kick과 punch는 a의 멤버  
    kick++;  
    punch++;  
    return *this; // 변경된 객체 자신(객체 a) 리턴  
}
```

전위 ++ 연산자 함수 코드

전위 ++ 연산자 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator++ (); // 전위 ++ 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
    << endl;
}

Power Power::operator++() {
    kick++;
    punch++;
    return *this; // 변경된 객체 자신(객체 a) 리턴
}
```

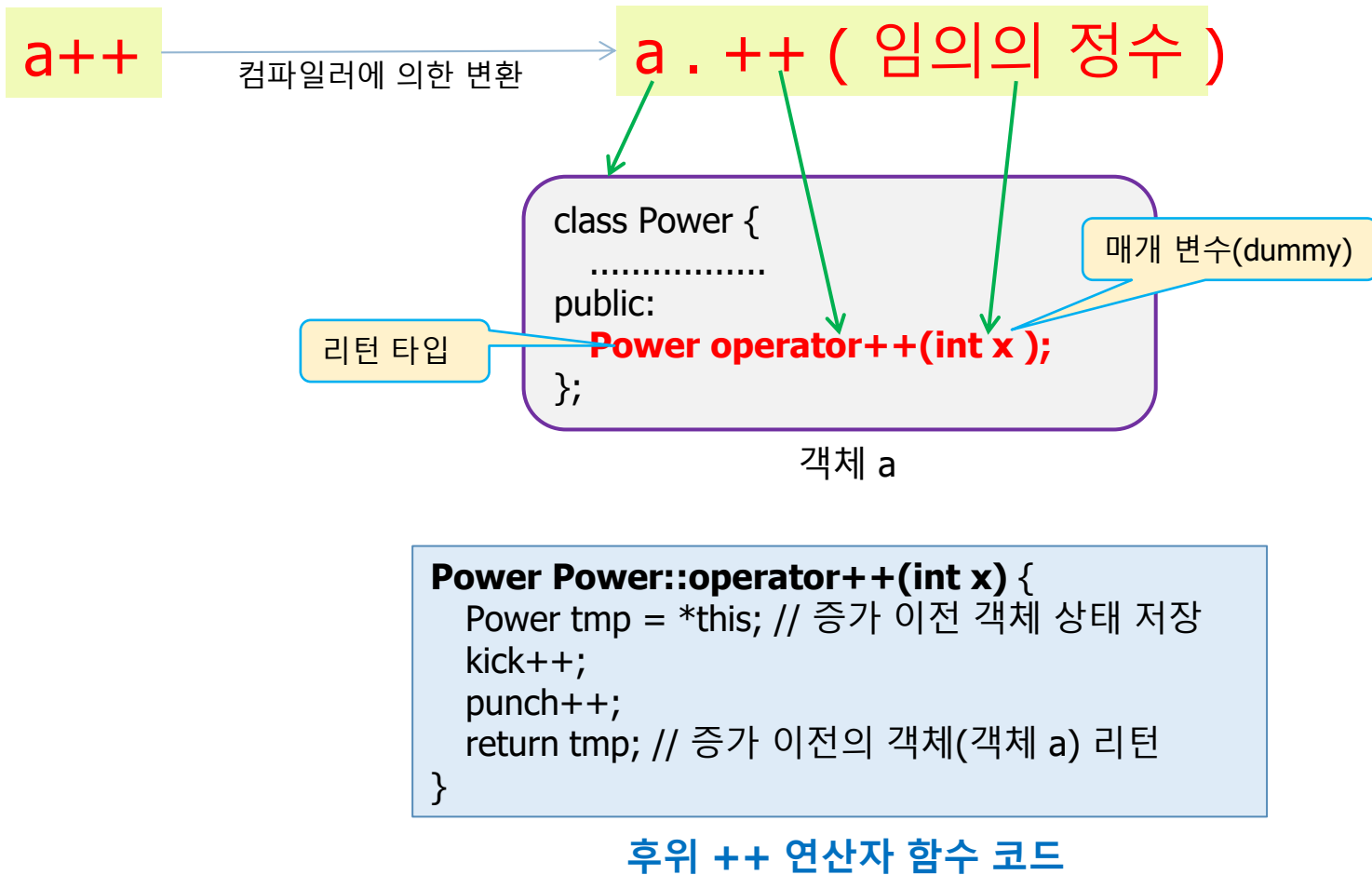
전위 ++ 연산자 멤버 함수 구현

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
    b = ++a; // 전위 ++ 연산자 사용
    a.show();
    b.show();
}
```

operator++() 함수 호출

kick=3,punch=5	}	a, b 출력
kick=0,punch=0		
kick=4,punch=6	}	b = ++a 후 a, b 출력
kick=4,punch=6		

멤버 함수로 단항 연산자 중복 구현2 : 후위 ++ 연산자 중복



후위 ++ 연산자 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator++(int x); // 후위 ++ 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ' '
        << "punch=" << punch << endl;
}

Power Power::operator++(int x) {
    Power tmp = *this; // 증가 이전 객체 상태를 저장
    kick++;
    punch++;
    return tmp; // 증가 이전 객체 상태 리턴
}
```

후위 ++ 연산자 멤버 함수 구현

`a++ -> a.++(int x)`

`int x -> dummy value`

`++a -> a.++()`

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
    b = a++; // 후위 ++ 연산자 사용
    a.show(); // a의 파워는 1 증가됨
    b.show(); // b는 a가 증가되기 이전 상태를 가짐
}
```

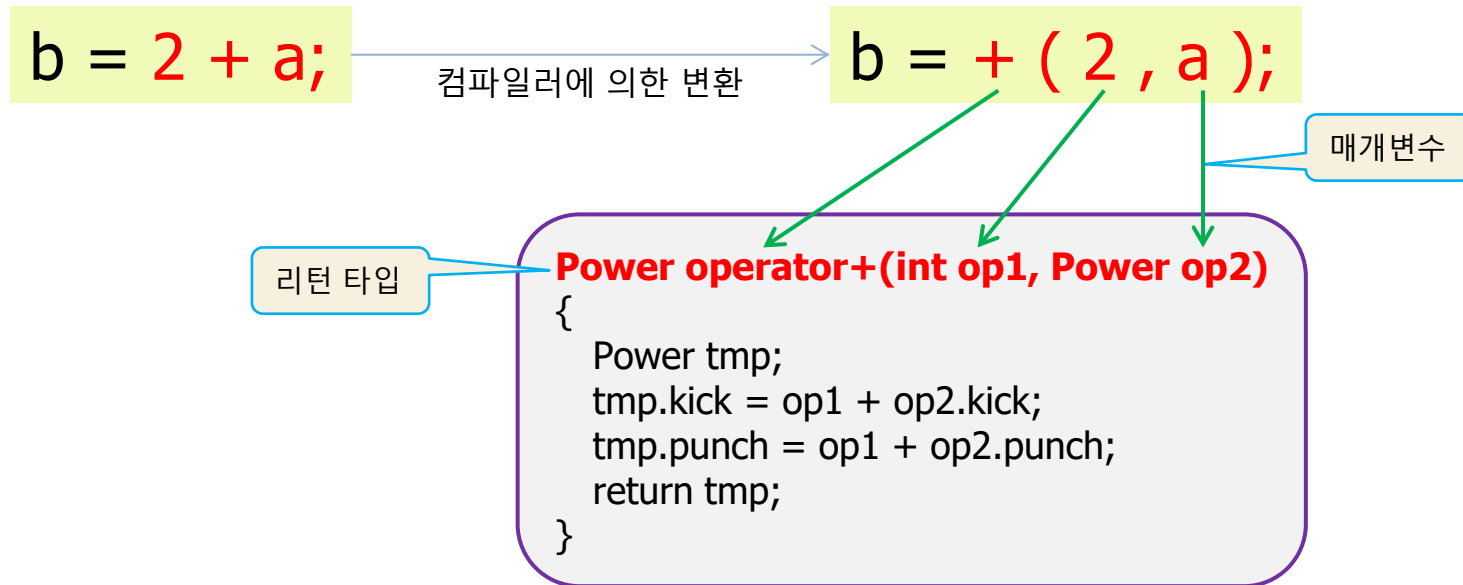
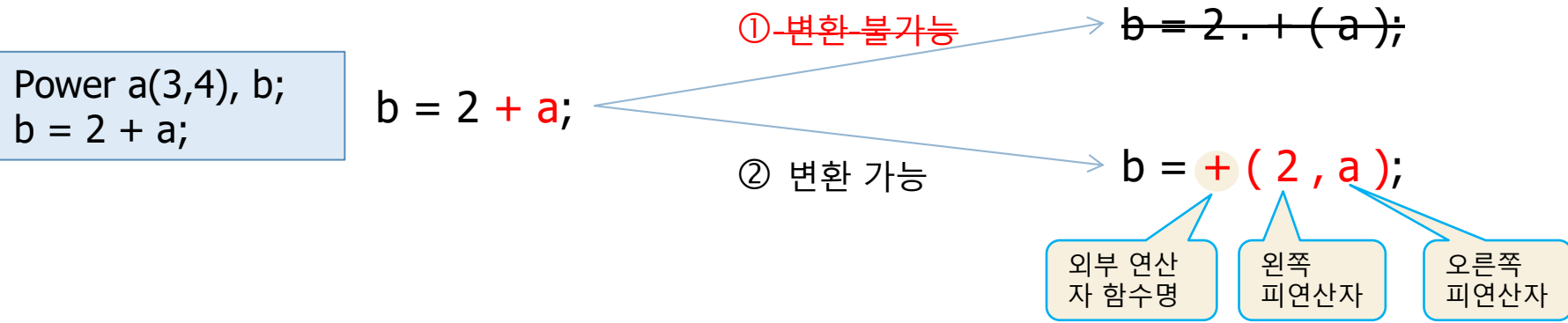
operator++(int) 함수 호출

kick=3,punch=5
kick=0,punch=0
kick=4,punch=6
kick=3,punch=5

a, b 출력

b = a++ 후 a, b 출력

2 + a 덧셈을 위한 + 연산자 함수 작성



2+a를 위한 + 연산자 함수를 friend로 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    friend Power operator+(int op1, Power op2); // 프렌드 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power operator+(int op1, Power op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = op1 + op2.kick; // kick 더하기
    tmp.punch = op1 + op2.punch; // punch 더하기
    return tmp; // 임시 객체 리턴
}
```

+ 연산자 함수를 외부 함수로 구현

private 속성인 kick, punch를 접근하도록 하기 위해, 연산자 함수를 friend로 선언해야 함

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
    b = 2 + a; // 파워 객체 더하기 연산
    a.show();
    b.show();
}
```

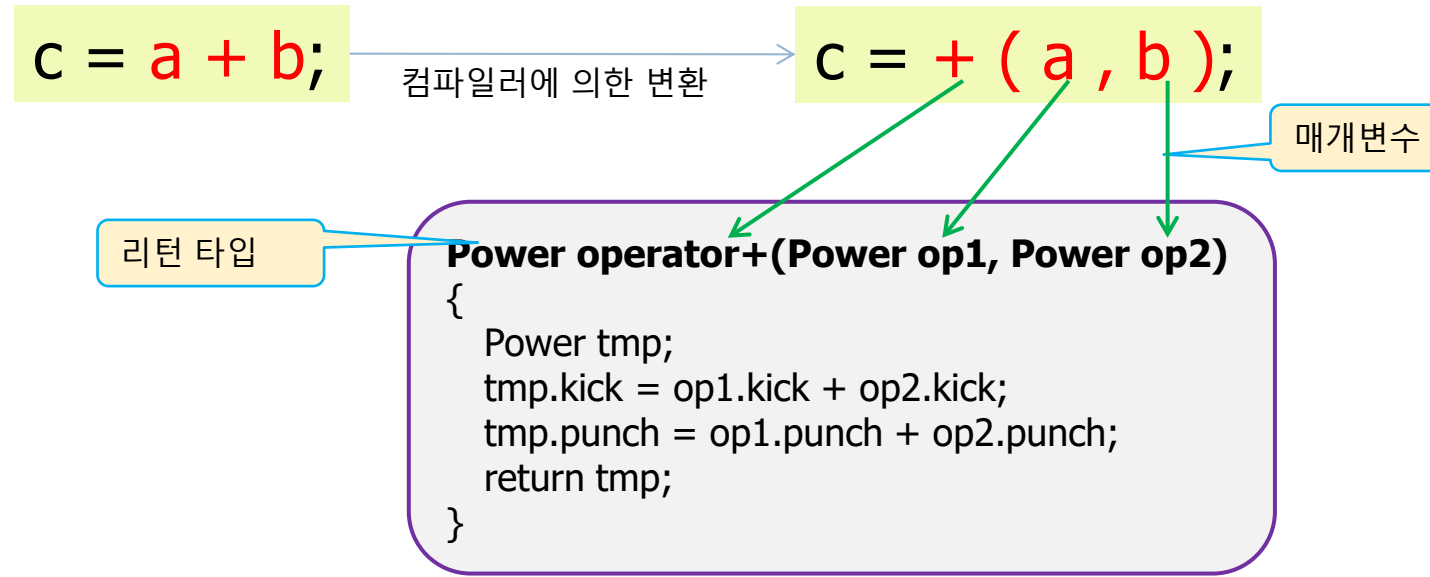
operator+(2, a) 함수 호출

```
kick=3,punch=5
kick=0,punch=0
kick=3,punch=5
kick=5,punch=7
```

a, b 출력

b = 2+a 후 a, b 출력

+ 연산자를 외부 friend 함수로 구현



a+b를 위한 연산자 함수를 friend로 작성

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    friend Power operator+(Power op1, Power op2); // 프렌드 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power operator+(Power op1, Power op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = op1.kick + op2.kick; // kick 더하기
    tmp.punch = op1.punch + op2.punch; // punch 더하기
    return tmp; // 임시 객체 리턴
}
```

+ 연산자 함수 구현

```
int main() {
    Power a(3,5), b(4,6), c;
    c = a + b; // 파워 객체 + 연산
    a.show();
    b.show();
    c.show();
}
```

operator+(a,b) 함수 호출

kick=3,punch=5
kick=4,punch=6
kick=7,punch=11

객체 a, b, c
순으로 출력

단항 연산자 ++를 friend로 작성하기

전위 연산자

++a

컴파일러에 의한 변환

++ (a)

리턴 타입

Power& operator++(Power& op)

```
{  
    op.kick++;  
    op.punch++;  
    return op;  
}
```

0은 의미 없는 값
으로 전위 연산자
와 구분하기 위함

후위 연산자

a++

컴파일러에 의한 변환

++ (a, 0)

리턴 타입

Power operator++(Power& op, int x)

```
{  
    Power tmp = op;  
    op.kick++;  
    op.punch++;  
    return tmp;  
}
```

++연산자를 friend로 작성한 예

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

class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    friend Power& operator++(Power& op);
    // 전위 ++ 연산자 함수 프렌드 선언
    friend Power operator++(Power& op, int x);
    // 후위 ++ 연산자 함수 프렌드 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}
```

```
Power& operator++(Power& op) { // 전위 ++ 연산자 함수 구현
    op.kick++;
    op.punch++;
    return op; // 연산 결과 리턴
}
```

참조 매개 변수 사용에 주목

참조 매개 변수 사용에 주목

```
Power operator++(Power& op, int x) { // 후위 ++ 연산자 함수 구현
    Power tmp = op; // 변경하기 전의 op 상태 저장
    op.kick++;
    op.punch++;
    return tmp; // 변경 이전의 op 리턴
}
```

```
int main() {
    Power a(3,5), b;
    b = ++a; // 전위 ++ 연산자
    a.show(); b.show();

    b = a++; // 후위 ++ 연산자
    a.show(); b.show();
}
```

kick=4,punch=6
kick=4,punch=6
kick=5,punch=7
kick=4,punch=6

b = ++a 실행 후
a, b 출력

b = a++ 실행 후
a, b 출력

다음 수업

❖ 상속

- 1_ 상속 개념
- 2_ 파생 클래스 정의 및 객체 생성 방법
- 3_ 파생 클래스의 생성자와 소멸자
- 4_ 접근 지정자와 접근 변경자
- 5_ 다중 상속