



프렌드와 연산자 중복

프렌드와 연산자 중복

- 이번 강의에서는
 - 프렌드 함수
 - 연산자 중복에 대하여 알아보려고 합니다.
- 특히 지난번 참조와 관련하여 잠시 언급되었던 복사 대입 연산자 & 이동 대입 연산자 구현도 다루려 합니다.
- 시작하도록 하겠습니다.

학습 목차

- 프렌드 함수
- 프렌드 선언
- 연산자 중복
- 연산자 함수

학습 목표

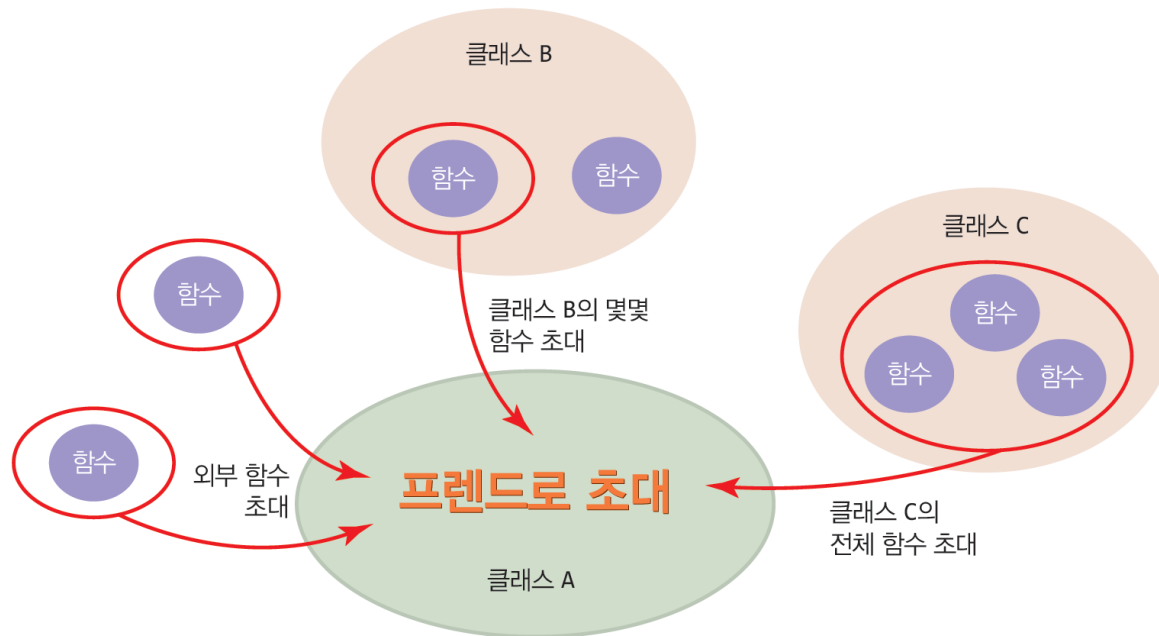
- 프렌드 함수를 이해하고 연산자 중복에 활용할 수 있다.
- 연산자 함수를 이해하고 활용할 수 있다.
- 다양한 연산자를 중복 정의할 수 있다.
- 멤버 함수로 연산자 중복을 구현하고 활용할 수 있다.
- 외부 함수로 연산자 중복을 구현하고 활용할 수 있다

C++ 프렌드

- 프렌드 함수
 - 클래스의 멤버 함수가 아닌 외부 함수 – 상속 불가
 - 전역 함수
 - 다른 클래스의 멤버 함수
 - 프렌드 함수 개수에 제한 없음
- friend 키워드로 클래스 내에 선언된 함수
 - 클래스의 모든 멤버를 접근할 수 있는 권한 부여 –private, protected 멤버
- 프렌드 선언의 필요성
 - 클래스의 멤버로 선언하기에는 무리가 있고, 클래스의 모든 멤버를 자유롭게 접근해야 할 경우 – 연산자 중복

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;
};
```

프렌드 선언 - 외부 함수

//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 {
```

```
    int width, height;
```

```
public:
```

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

```
//외부 함수 equals()를 프렌드로 선언, private 속성을 가진 width, height에 접근 할 수 있음
```

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

```
};
```

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

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

```
    else return false; }
```

```
int main() {
```

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

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

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


프렌드 선언 - 다른 클래스의 멤버 함수

```
class Rect;
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); //RectManager 클래스의 equals() 멤버를 프렌드로 선언
};

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; }
```

프렌드 선언 - 다른 클래스 전체

```

class Rect;
class RectManager {
public:
    bool equals(Rect r, Rect s);
    void copy(Rect& dest, Rect& src); };

class Rect {
    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;

    man.copy(b, a); // a를 b에 복사한다.
    if(man.equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
}

```

연산자 중복

- 피 연산자에 따라 서로 다른 연산을 하도록 연산자를 중복 작성하는 것
 - + 기호의 사례
 - 숫자 더하기 : $2 + 3 = 5$
 - 색 혼합 : 빨강 + 파랑 = 보라
 - 간결한 의미 전달
 - 다형성
- C++ 언어에서도 연산자 중복 가능
 - C++ 언어에 본래부터 있던 연산자에 새로운 의미 정의
 - 높은 프로그램 가독성
- 자신이 정의한 클래스를 기본 타입처럼 다룰 수 있다.
- 프로그램을 세밀하게 제어 할 수 있다

연산자 중복의 사례 : + 연산자에 대해

- 정수 더하기

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

- 문자열 합치기

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

- 색 섞기

```
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

- 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
- 피 연산자의 개수를 바꿀 수 없음
- 연산의 우선 순위 변경 안됨
- 모든 연산자가 중복 가능하지 않음
- 디폴트 매개변수 사용 불가

연산자 중복의 특징

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

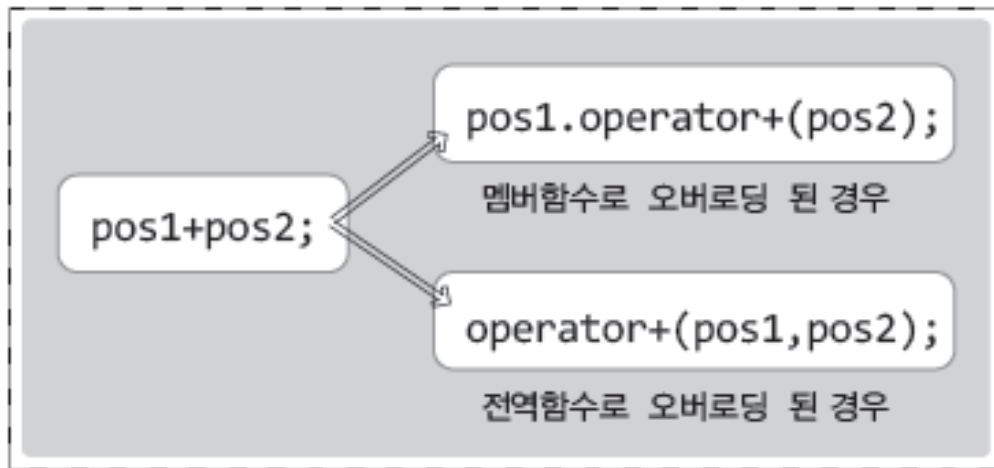
중복 가능한 연산자

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

중복 불가능한 연산자

연산자 함수

- 연산자 함수 구현 방법 2 가지
 - 클래스의 멤버 함수로 구현
 - 외부 함수로 구현하고 클래스에 프렌드 함수로 선언
- 연산자 함수 형식 리턴타입 **operator** 연산자(매개변수리스트);
- 해석 방법



+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 프렌드로 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 연산자 구현

이항 연산자 + 중복

`c = a + b;`

컴파일러에 의한 변환

`c = a . + (b);`

```
class Power {
    int kick;
    int punch;
public:
```

```
.....
Power operator+ (Power op2);
};
```

리턴 타입

Power a

오른쪽 피연산자 b가 op2에 전달

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

+ 연산자 함수 코드

두 개의 Power 객체를 더하는 + 연산자 작성

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

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

//+ 연산자 멤버 함수 구현

```
Power Power::operator+(const 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
}
```

```
Power Power::operator+(const Power& op2) {
    return Power(this->kick + op2.kick, this->punch + op2.punch);
}
```

```
int main() {
    Power a(3,5), b(4,6), c;
```

// 객체 a의 operator+() 멤버 함수 호출

c = a + b; // 파워 객체 + 연산

```
a.show();
b.show();
c.show();
}
```

== 연산자 중복

a == b

컴파일러에 의한 변환

a . == (b)

class Power {

.....

public:

bool operator== (const Power& op2);

};

오른쪽 피연산자 b가 op2에 전달

리턴 타입

Power a

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

== 연산자 함수 코드

두 개의 Power 객체를 비교하는 == 연산자 작성

```
class Power {
public:
    void show();
    bool operator==(const Power& op2); // == 연산자 멤버 함수 선언
};

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

//== 연산자 멤버 함수 구현
bool Power::operator==(const Power& op2) {
    if(kick==op2.kick && punch==op2.punch)
        return true;
    else
        return false;
}
```

```
int main() {
    Power a(3,5), b(3,5); // 2 개의 동일한 파워 객체 생성
    a.show();
    b.show();

    //operator==() 멤버 함수 호출
    if(a == b)
        cout << "두 파워가 같다." << endl;
    else
        cout << "두 파워가 같지 않다." << endl;
}
```

+= 연산자 중복

c = a += b;

컴파일러에 의한 변환

c = a . += (b);

class Power {

.....

public:

리턴 타입

Power& **operator+=** (const Power& op2);

};

Power a

오른쪽 피연산자 b가 op2에 전달

```
Power& Power::operator+= (const Power& op2) {
    kick = kick + op2.kick;
    punch = punch + op2.punch;
    return *this; // 자신의 참조 리턴
}
```

+= 연산자 함수 코드

두 Power 객체를 더하는 += 연산자 작성

```
class Power {
public:
    void show();
    // += 연산자 함수 선언
    Power& operator+= (const Power& op2);
};

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

```
//+= 연산자 멤버 함수 구현
Power& Power::operator+=(const Power& op2) {
    kick = kick + op2.kick; // kick 더하기
    punch = punch + op2.punch; // punch 더하기
    return *this; // 합한 결과 리턴
}
```

```
int main() {
    Power a(3,5), b(4,6), c;
    a.show();
    b.show();

    // operator+=() 멤버 함수 호출
    c = a += b; // 파워 객체 더하기
    a.show();
    c.show();
}
```


+ 연산자 작성 : $b = a + 2$;

```
class Power {
public:
    void show();
    Power operator+ (int op2); // + 연산자 함수 선언
};
```

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

//+ 연산자 멤버 함수 구현

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

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
```

```
//operator+(int) 함수 호출
b = a + 2; // 파워 객체와 정수 더하기
a.show();
b.show();
}
```

대입(=) 연산자 구현(1)

```
#include <iostream>
using namespace std;
class Person {
    char* name;
    int id;
public:
    Person(int id, const char* name);
    Person(const Person& p);
    ~Person();
    void show() { cout << id << ' ' << name << endl; }
    Person& operator= (const Person &rhs); // = 연산자 함수
};
Person::Person(int id, const char* name) {
    this->id = id;
    int len = strlen(name);
    this->name = new char[len + 1];
    strcpy(this->name, name);
}
```

복사 과정 중
예외가 발생하면 문제가 생김

```
Person& Person::operator= (const Person &rhs) {
    if (this == &rhs) //자기 자신을 대입
        return *this;
    delete name; //기존 메모리 해제
    name = nullptr;

    //메모리 새로 할당
    int len = strlen(rhs.name);
    name = new char[len + 1];

    //데이터 복사
    id = rhs.id;
    strcpy(name, rhs.name);
    return *this;
}

int main() {
    Person father(1, "Kitae");
    Person daughter(father);
    daughter.changeName("Grace");
    father = daughter; // 대입 연산자 함수 호출
    father.show();
    return 0;
}
```

대입(=) 연산자 구현(2)

- 구현 1에서 예외 발생 시 문제가 생기지 않도록 복사 후 바꾸기 패턴 적용 – 예외 안전성 보장

```
class Person{
    char *name;
    int id;
public:
    Person &operator=(const Person &person);
```

//표준 라이브러리 알고리즘에서 활용할 수 있게 외부 함수로 swap_person() 추가
//단, 복사 과정에서 예외가 발생하지 않도록 한다.

```
friend void swap_person(Person& first, Person& second) noexcept;
```

```
};
```

```
Person &Person::operator=(const Person &person){
    if (this == &person) //자기 자신을 대입
        return *this;
```

```
    Person temp(person); //임시 객체 생성
    swap_person(*this, temp); //현재 객체를 생성된 임시 복사본으로 교체
    return *this;
}
```

```
void swap_person(Person& first, Person& second) noexcept{
    //표준 라이브러리(<utility>) 에서 제공하는 유틸리티 swap() 함수를 사용 간단히 처리
    //추후 Person 클래스 멤버 변수 추가 시 swap_person() 함수만 수정하면 됨
    swap(first.id, second.id);
    swap(first.name, second.name);
}
```

이동 대입(=) 연산자

```
#include <iostream>
#include <utility> //swap()
using namespace std;
class Person {
Public:
    Person& operator= (const Person &&person ) noexcept; //이동 대입 연산자
};

Person &Person::operator=(const Person &&person) noexcept {
    Person temp(person);
    swap_person(*this, temp);    return *this;
}

void swap_person(Person &first, Person &second) noexcept {
    swap(first.id, second.id);
    swap(first.name, second.name);
}

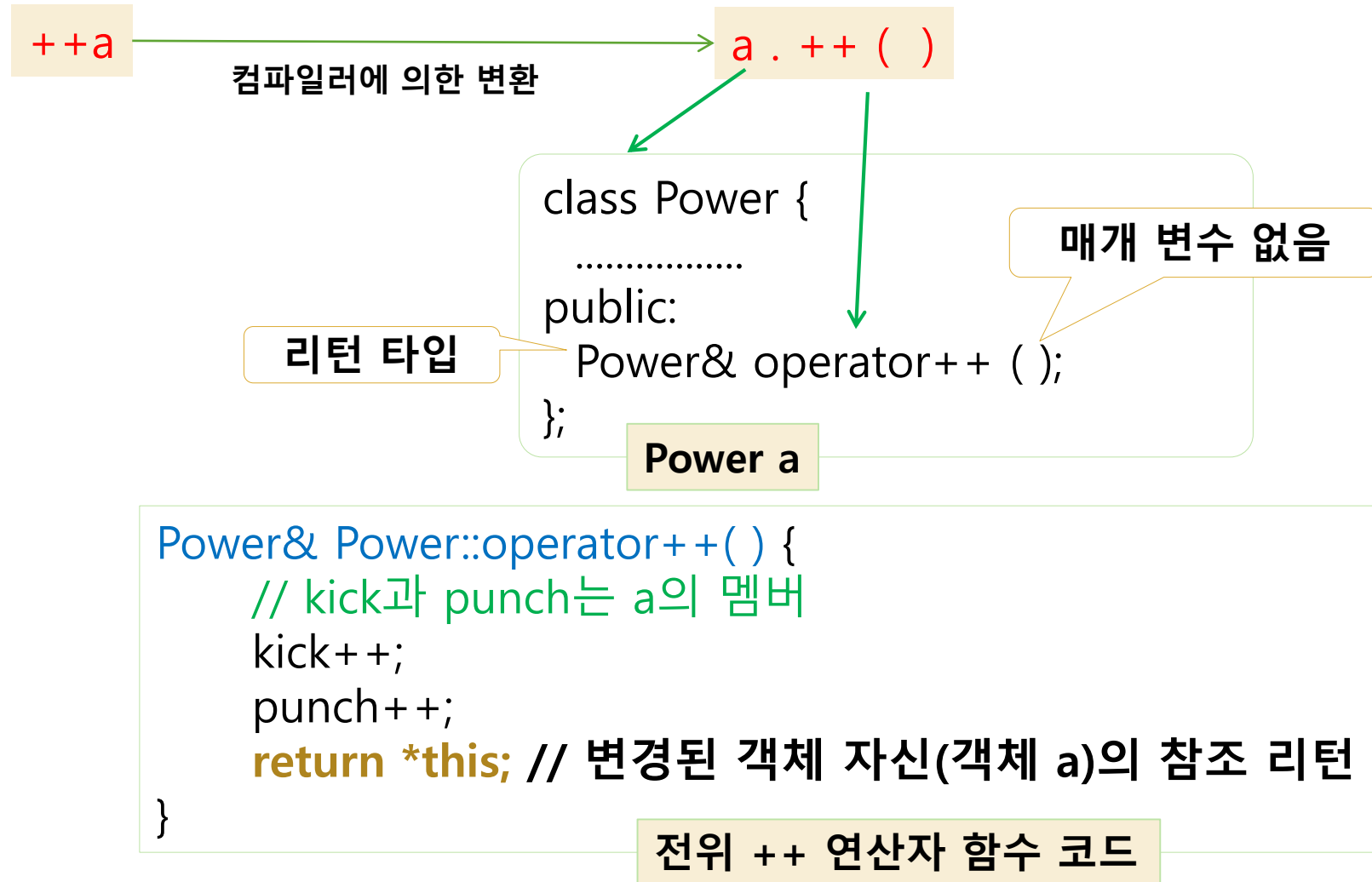
Person createObject() {
    return Person(10, "Hallym"); //rvalue(값) 반환
}

int main() {
    Person p1(1, "software");
    p1 = createObject(); //이동 대입 연산자 함수 호출
    p1.show();
    return 0;
}
```

단항 연산자 중복

- 단항 연산자
 - 피 연산자가 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--

전위 ++ 연산자 중복



전위 ++ 연산자 작성

```
class Power {
public:
    void show();
    Power& operator++ (); // 전위 ++ 연산자 함수 선언
};
```

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

//전위 ++ 연산자 멤버 함수 구현

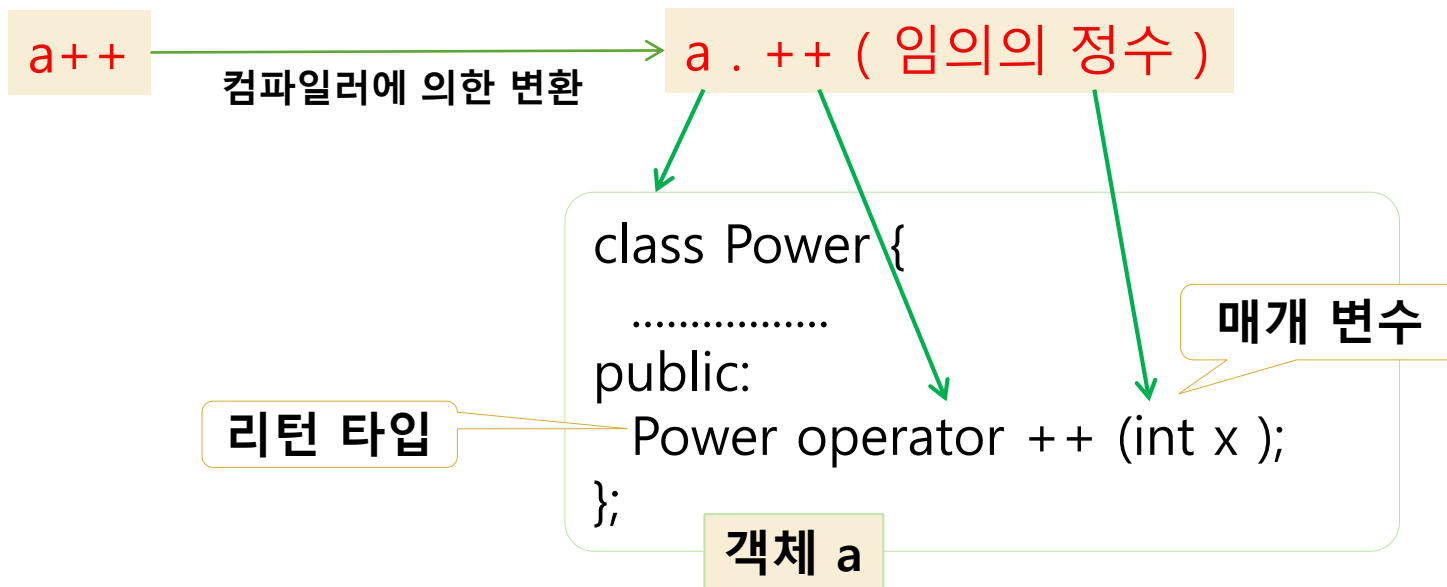
```
Power& Power::operator++() {
    kick++;
    punch++;
    return *this; // 변경된 객체 자신(객체 a)의 참조 리턴
}
```

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
```

//operator++() 함수 호출, 전위 ++ 연산자 사용

```
b = ++a;
a.show();
b.show();
}
```

후위 ++ 연산자 중복



```

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

후위 ++ 연산자 함수 코드

후위 ++ 연산자 작성

```
class Power {
public:
    void show();
    Power operator++ (int x); // 후위 ++ 연산자 함수 선언
};
```

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

//후위 ++ 연산자 멤버 함수 구현

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

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
```

//operator++(int) 함수 호출

```
b = a++; // 후위 ++ 연산자 사용
a.show(); // a의 파워는 1 증가됨
b.show(); // b는 a가 증가되기 이전 상태를 가짐
}
```

Power 클래스에 ! 연산자 작성

! 연산자를 Power 클래스의 멤버 함수로 작성하라.
!a는 a의 kick, punch 파워가 모두 0이면 true, 아니면 false를 리턴한다.

```
class Power {
public:
    void show();
    bool operator!(); // ! 연산자 함수 선언
};

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

//! 연산자 멤버 함수 구현
bool Power::operator!() {
    if(kick == 0 && punch == 0)
        return true;
    else
        return false;
}
```

```
int main() {
    Power a(0,0), b(5,5);
    //operator!() 함수 호출
    if(!a) // ! 연산자 호출
        cout << "a의 파워가 0이다." << endl;
    else
        cout << "a의 파워가 0이 아니다." << endl;
    if(!b) // ! 연산자 호출
        cout << "b의 파워가 0이다." << endl;
    else
        cout << "b의 파워가 0이 아니다." << endl;
}
```

프렌드를 이용한 연산자 중복

2 + a 덧셈을 위한 + 연산자 함수 작성

Power a(3,4), b;
b = 2 + a;

① ~~변환 불가능~~ → ~~b = 2 . + (a);~~

b = 2 + a;

② 변환 가능

b = + (2 , a);

외부 연산자
함수명

왼쪽
피연산자

오른쪽
피연산자

b = 2 + a;

컴파일러에 의한 변환

b = + (2 , a);

매개변수

리턴 타입

```
Power operator+ (int op1, Power op2) {
    Power tmp;
    tmp.kick = op1 + op2.kick;
    tmp.punch = op1 + op2.punch;
    return tmp;
}
```

2+a를 위한 + 연산자 함수

```
class Power {
public:
    void show();
    friend Power operator+(int op1, const Power& op2); // 프렌드 선언
};
```

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

//+ 연산자 함수를 외부 함수로 구현

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

```
Power operator+(int op1, const 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
}
```

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

이항 + 연산자 중복

`c = a + b;`

컴파일러에 의한 변환

`c = + (a , b);`

리턴 타입

```
Power operator+ (Power op1, Power op2) {  
    Power tmp;  
    tmp.kick = op1.kick + op2.kick;  
    tmp.punch = op1.punch + op2.punch;  
    return tmp;  
}
```

매개변수

a+b를 위한 연산자 함수를 프렌드로 작성

```
class Power {
public:
    void show();
    friend Power operator+(const Power& op1, const Power& op2); //
    프렌드 선언
};

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

//+ 연산자 함수 구현
Power operator+(const Power& op1, const 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
}
```

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

단항 연산자 ++ 중복

(a) 전위 연산자

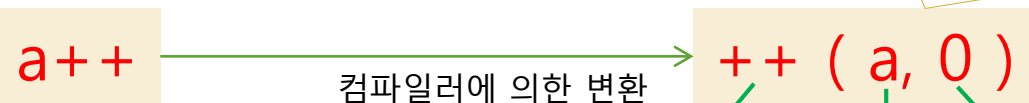


리턴 타입

```
Power& operator++ (Power& op) {
    op.kick++;
    op.punch++;
    return op;
}
```

0은 의미 없는 값으로 전위 연산자와 구분하기 위함

(b) 후위 연산자



리턴 타입

```
Power operator++ (Power& op, int x) {
    Power tmp = op;
    op.kick++;
    op.punch++;
    return tmp;
}
```


++ 연산자를 프렌드로 작성한 예

```
class Power {
public:
    void show();
    // 전위 ++ 연산자 함수 프렌드 선언
    friend Power& operator++(Power& op);
    // 후위 ++ 연산자 함수 프렌드 선언
    friend Power operator++(Power& op, int x);
};

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

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

//참조 매개 변수 사용에 주목
// 후위 ++ 연산자 함수 구현
Power operator++(Power& op, int x) {
    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
    b = a++; // 후위 ++ 연산자
}
```

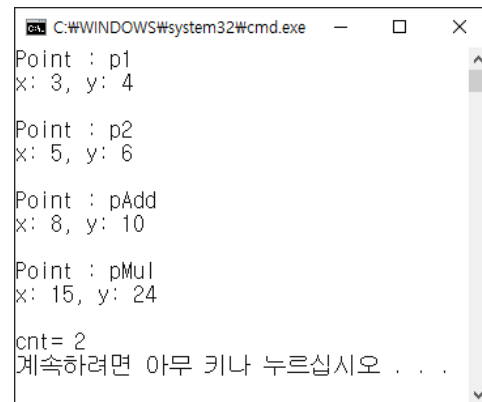
활용1. 프렌드 함수

- PointCom 클래스 멤버 함수와 전역함수 showPoint()를 프렌드 함수로 갖는 Point 클래스를 구현하고 실행하는 프로그램을 완성 하시오

```
class Point;
void showPoint(const Point &p, const string pname);

class PointCom {
    int cnt;
public:
    PointCom():cnt(0){}
    Point pointAdd(const Point& p1, const Point& p2);
    Point pointMul(const Point& p1, const Point& p2);
    ~PointCom() { cout << "cnt= " << cnt << endl; }
};

class Point {
    int x, y;
public:
    Point(const int &xp, const int &yp):x(xp), y(yp){}
    //이곳에 프렌드 함수 선언
};
```



```
C:\WINDOWS\system32\cmd.exe
Point : p1
x: 3, y: 4

Point : p2
x: 5, y: 6

Point : pAdd
x: 8, y: 10

Point : pMul
x: 15, y: 24

cnt= 2
계속하려면 아무 키나 누르십시오 . . .
```

```
int main() {
    Point p1(3, 4);
    Point p2(5, 6);
    PointCom pc;
    showPoint(p1, "p1");
    showPoint(p2, "p2");
    showPoint(pc.pointAdd(p1, p2), "pAdd");
    showPoint(pc.pointMul(p1, p2), "pMul");
}
```

활용2. 멤버 함수로 이항 연산자 구현 예

- Book 객체에 대하여 다음의 연산을 할 수 있도록 +=, == 연산자를 Book 클래스 멤버 함수로 구현 하시오

```
class Book {
    string title;
    int price, pages;
public:
    Book(string title = "", int price = 0, int pages = 0) {
        this->title = title;
        this->price = price;
        this->pages = pages;
    }
    void show() { cout << title << ' ' << price << "원 " << pages << " 페이지" << endl; }
    string getTitle() { return title; }
};

int main() {
    Book a("청춘", 20000, 300), b("고품 C++", 30000, 500);
    a += 500; // 책 a의 가격 500원 증가
    a.show();
    if (b == 30000)
        cout << "정가 30000원" << endl; // price 비교
}
```

청춘	20500원	300	페이지
정가	30000원		

활용3. 참조를 리턴하는 << 연산자 작성

Power 객체의 kick과 punch에 정수를 더하는 << 연산자를 멤버 함수로 작성하라

```
class Power {
public:
    void show();
    Power& operator << (int n); // 연산 후 Power 객체의 참조 리턴
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}
//참조 리턴
Power& Power::operator <<(int n) {
    kick += n;
    punch += n;
    return *this; // 이 객체의 참조 리턴
}
```

```
int main() {
    Power a(1, 2);
    a << 3 << 5 << 6;
    a.show();
}
```

Q & A

- “프렌드와 연산자 중복”에 대한 학습이 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 과제와 퀴즈(제한 기간내 2회 응시 가능)가 있습니다.
- 다음은 상속에 대하여 학습하도록 하겠습니다.
- 수고하셨습니다.^ ^