

I2PXO2CUBE

---

Week 5

---

방찬웅

# C++ STUDY

Week 5  
함수, 연산자 오버로딩  
Static

01

하나,

Function Overloading

02

둘,

Static

03

셋,

Operator Overloading

01

하나,

Function Overloading

02

둘,

Static

03

셋,

Operator Overloading

# 함수 중복 (Function Overloading)

같은 이름의 함수를 여러 개 만들 수 있다.

## 함수 중복의 조건

이름이 동일하여야 한다.

매개 변수 "타입"이나 매개 변수의 "개수"가 달라야 한다.

"리턴 타입"은 고려되지 않는다.

## 함수 중복의 장점

같은 이름의 함수로 작성하면 같은 기능을 다른 이름으로 지을 필요가 없고, 구분지어 기억할 필요가 없다.

# 함수 중복 (Function Overloading)

## sum()

```
int sum(int a, int b, int c)
{
    return a + b + c;
}
```

가능

```
double sum(double a, double b)
{
    return a + b;
}
```

가능

```
int sum(int a, int b)
{
    return a + b;
}
```

가능

```
int main()
{
    // overloading된 sum 함수 호출 (컴파일러가 구분)
    cout << sum(1, 2, 3) << sum(1.1, 2.2) << sum(1, 2);
}
```

불가능

```
double sum(int a, int b)
{
    return (double)(a + b);
}
```

# 함수 중복 (Function Overloading)

## greet()

```
void greet()
{
    cout << "Hello" << endl;
}
```

그저 인사

```
void greet(string name)
{
    cout << "Hello " << name << endl;
}
```

이름부르며 인사

```
void greet(string name, int id)
{
    cout << "Hello " << name << " your id number is " << id << endl;
}
```

이름부르며 인사 후 ID 안내

```
greet();
greet("Chanwoong");
greet("Chanwoong", 123);
```

```
Hello
Hello Chanwoong
Hello Chanwoong your id number is 123
```

함수 중복으로 가능 확장

# 실습(1)

big() 함수를 만들어 Function Overloading 연습

1. int 형 두 수중 큰 수를 반환하는 big()
2. int 형 배열의 원소 중 가장 큰 수를 반환하는 big()

# 실습(2)

sum() 함수를 만들어 Function Overloading 연습

1. int 형 두 수를 더하여 반환하는 sum()
2. int 형 하나의 수를 전달 받고 0부터 그 수 까지의 합을 반환하는 sum()

# 디폴트 매개변수 (default parameter)

함수가 호출될 때 매개 변수에 값이 넘어오지 않는다면, 미리 정해진 디폴트 값을 받도록 선언된 매개변수

```
void msg(string text = "Good Morning!")
{
    cout << text << endl;
}
```

Good Morning!  
Good Night!

```
msg();
msg("Good Night!");
```

# 디폴트 매개변수 (default parameter)

## 디폴트 매개 변수 사용 제약 조건

디폴트 매개 변수는 모두 "끝 쪽"에 몰려 선언되어야 한다!

```
void calc(int a, int b = 5, int c, int d = 0); // 컴파일 오류
```

```
void sum(int a = 0, int b, int c); // 컴파일 오류
```

```
void calc(int a, int b = 0, int c = 0, int d = 0); // 컴파일 성공
```

컴파일러는 앞에서부터 순서대로 함수의 매개 변수에 전달하고, 나머지는 디폴트 값으로 전달한다.

# 디폴트 매개변수 (default parameter)

## 컴파일러가 매개 변수에 값을 정하는 과정

```
void func(int a, int b = 0, int c = 0, int d = 0);
```

g(10);              -> g(10, \_\_, \_\_, \_\_);      ->

g(10, 5);          -> g(10, 5, \_\_, \_\_);      ->

g(10, 5, 20);     -> g(10, 5, 20, \_\_);     ->

g(10, 5, 20, 30); -> g(10, 5, 20, 30);   ->

# 실습(3)

## 디폴트 매개 변수를 가진 함수 만들기 연습

printLine(); // 한 줄에 '-' 문자를 10개 출력한다.

printLine('%'); // 한 줄에 '%' 문자를 10개 출력한다.

printLine('@', 5) // 다섯 줄에 '@' 문자를 10개 출력한다.

# 중복 생성자

```
Circle();
Circle(int radius);
```

```
Circle::Circle()
{
    radius = 1;
}

Circle::Circle(int radius)
{
    this->radius = radius;
}
```

슬라이드 넘기지 말고 생각해보기!

## 중복 생성자

```
Circle();  
Circle(int radius);
```

```
Circle::Circle()  
{  
    radius = 1;  
}  
  
Circle::Circle(int radius)  
{  
    this->radius = radius;  
}
```

```
// 위의 두 생성자를 합친 생성자  
Circle(int radius = 1);
```

```
// 위의 두 생성자를 합친 생성자  
Circle::Circle(int radius = 1)  
{  
    this->radius = radius;  
}
```

```
Circle c1 = Circle();  
Circle c2 = Circle(10);
```

# Function Overloading의 모호성

함수 중복 조건을 갖추었다 하더라도 중복된 함수에 대한 호출이 "모호"(ambiguous) 해지는 경우가 발생한다.

1. 형 변환으로 인한 모호성
2. 참조 매개 변수로 인한 모호성
3. 디폴트 매개 변수로 인한 모호성

# Function Overloading의 모호성

## 1. 형 변환으로 인한 모호성

```
float square(float a);  
double square(double a);
```

```
square(3);
```

정수 3을 float로 형변환 할지, double로 형변환 할지 모호함. 컴파일 오류

\* 자동 형 변환 : char -> int -> long -> float -> double (화살표로 가능)

# Function Overloading의 모호성

## 2. 참조 매개 변수로 인한 모호성

```
int add(int a, int b)
int add(int a, int &b)
```

```
int a = 10, b = 20;
```

```
add(a, b)
```

b가 모호 하기 때문에 컴파일 오류

# Function Overloading의 모호성

## 3. 디폴트 매개 변수로 인한 모호성

```
void msg(int id);  
void msg(int id, string text = "text");
```

```
msg(6);
```

인자가 없는 경우가 두 가지이기 때문에 모호. 컴파일 오류

01

하나,

Function Overloading

02

둘,

Static

03

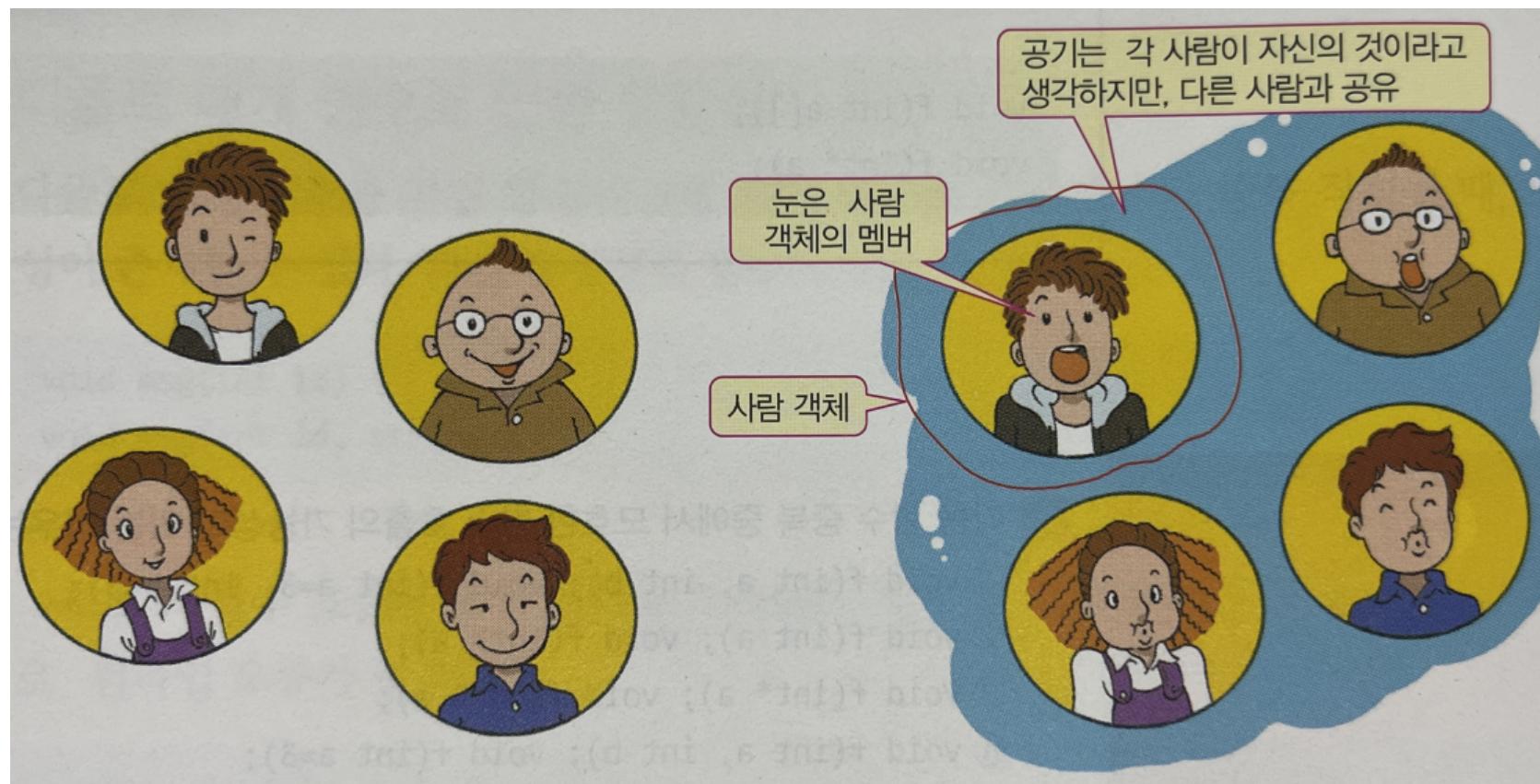
셋,

Operator Overloading

# static 멤버

**생명주기 - 프로그램이 시작할 때 생성되고 종료할 때 소멸**

**사용 범위 - 전역(global)**

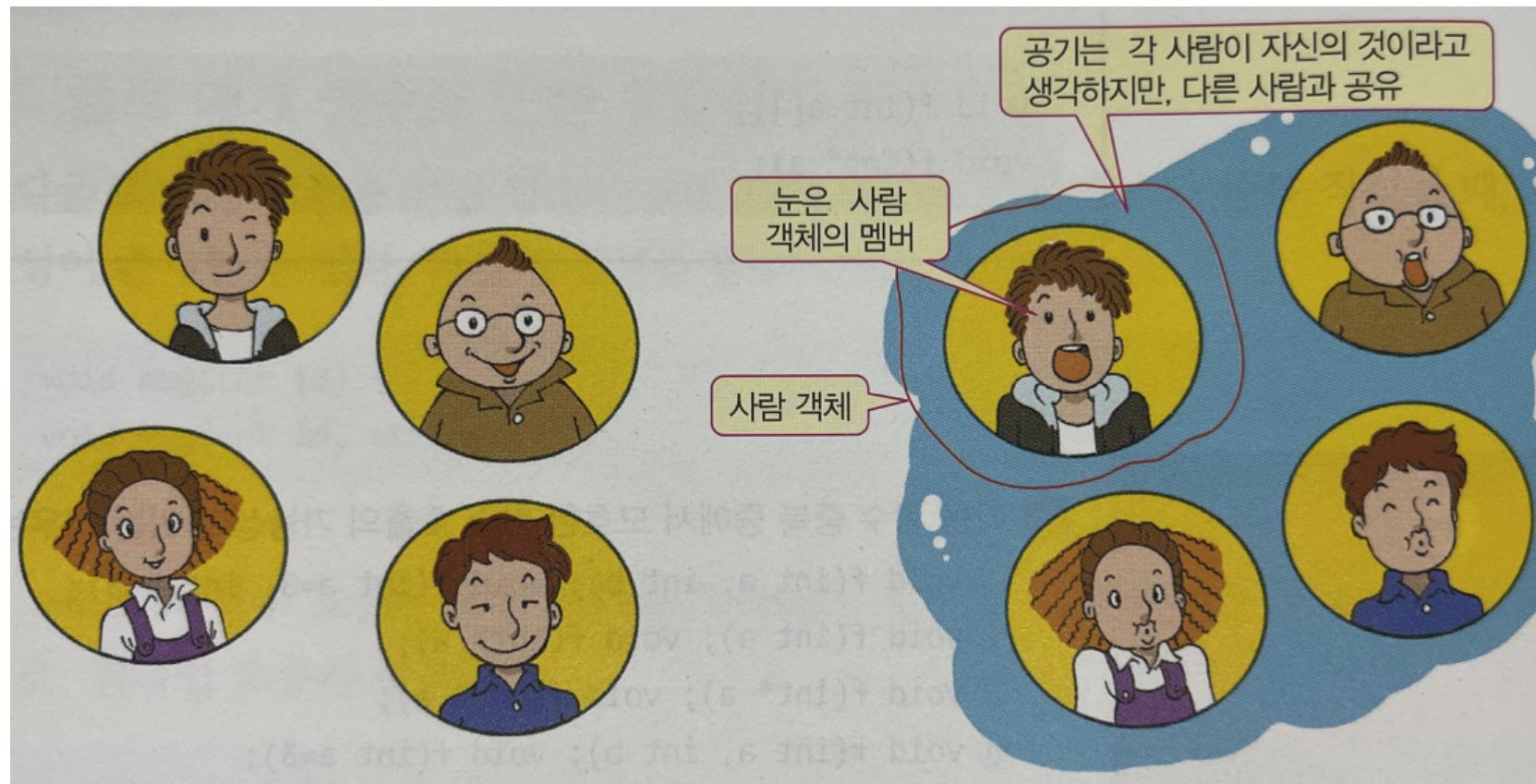


**객체 - 사람, 눈 - 멤버, 공기 - static 멤버(사람이 있기 이전부터 생성, 모두 공유)**

# static 멤버

**생명주기 - 프로그램이 시작할 때 생성되고 종료할 때 소멸**

**사용 범위 - 전역(global)**



**객체 - 사람, 눈 - non-static 멤버 → 인스턴스 멤버**

**공기 - static 멤버(사람이 있기 이전부터 생성, 모두 공유) → 클래스 멤버**

# static 멤버



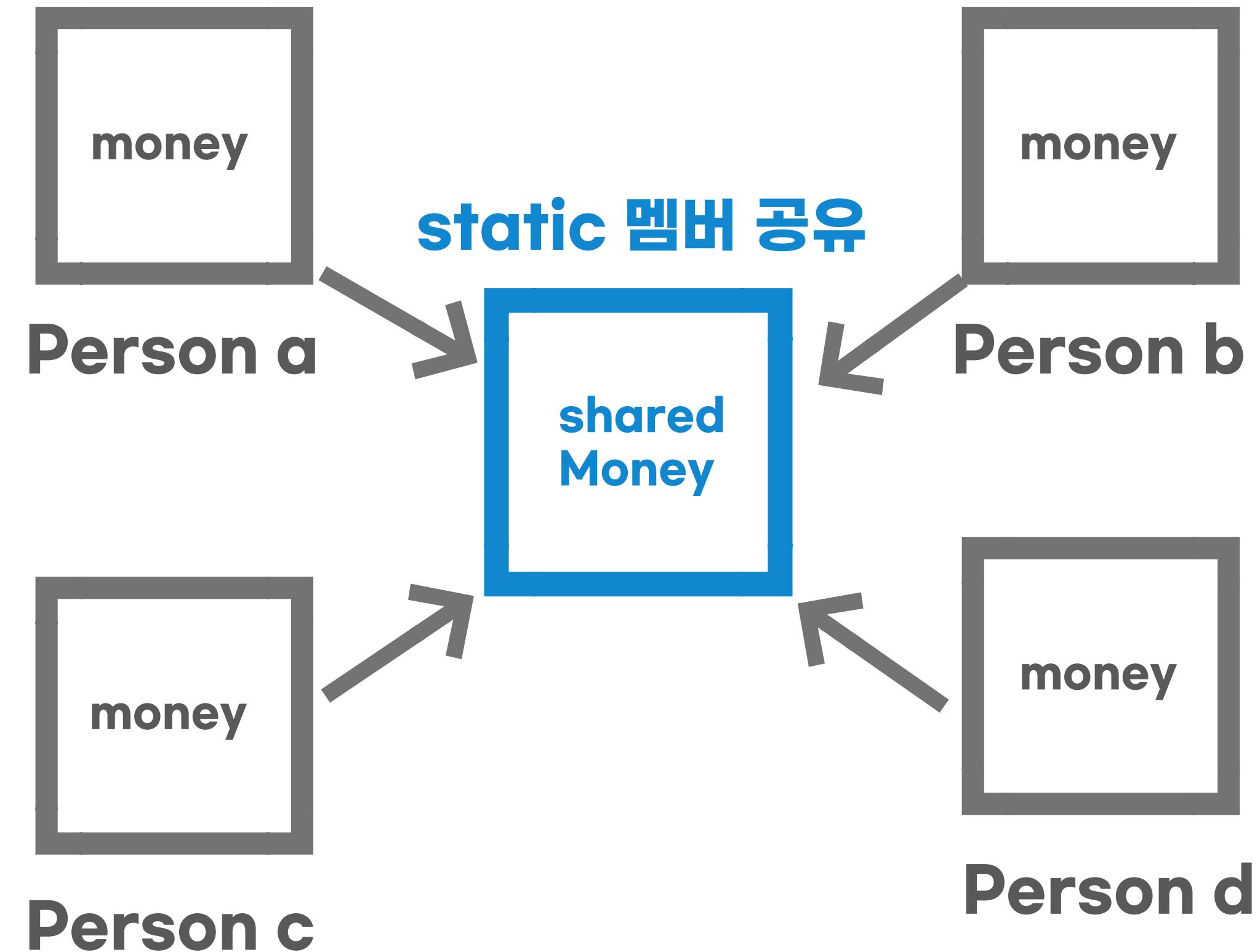
# static 멤버

```
class Person
{
public:
    Person();
    ~Person();

    int money;
    void addMoney(int money)
    {
        this->money += money;
    }

    static int sharedMoney;
    static void addSharedMoney(int money)
    {
        sharedMoney += money;
    }
};
```

```
// static 변수 공간 할당.
// 반드시 프로그램의 전역 공간에 선언.
int Person::sharedMoney = 10;
```



# static 멤버

```
int main()
{
    // static 멤버는 객체가 생성되기 전에도 호출이 가능하다.
    //Person::addSharedMoney(10000);

    Person bang;
    bang.money = 100; // bang의 개인 돈 100
    bang.addSharedMoney(200); // static 멤버 접근. 공금 = 200

    Person chan;
    chan.money = 150;
    chan.addMoney(200);
    chan.addSharedMoney(200);

    cout << "bang : " << bang.money << endl;
    cout << "chan : " << chan.money << endl;
    cout << "sharedMoney : " << bang.sharedMoney << endl;
    cout << "sharedMoney : " << chan.sharedMoney << endl;
    cout << "sharedMoney : " << Person::sharedMoney << endl;
    // 객체 이름으로 static 멤버 호출, non-static 멤버는 불가능
    Person::addSharedMoney(1000);
    cout << "sharedMoney : " << Person::sharedMoney << endl;
}
```

## 실습(4)

### 전역 변수나 전역 함수를 클래스에 캡슐화

**Math**라는 이름의 클래스를 만든다.

클래스 안에 static 함수 3개를 만든다.

1) **sum(int a, int b);**

2) **max(int a, int b);**

3) **min(int a, int b);**

**main()**에서 테스트 한다.

## static 멤버 함수의 특징

**static 멤버 함수는 오직 static 멤버들만 접근**

**이유?**

**static 함수는 객체가 생성되지 않은 어떤 시점에도 호출될 수 있고,  
클래스 이름으로 직접 호출이 가능하기 때문에  
static 멤버 함수에서 non-static 멤버 함수 접근이 불가능하다.**

**반대로 non-static 멤버는 static 멤버를 접근하는데 전혀 제약이 없다.**

**this 키워드 사용 불가능**

**이유?**

**this는 객체를 가리키는 키워드이므로, 객체 생성 전에도 호출 가능한  
static 멤버 함수에선 사용할 수 없다.**

01

하나,

Function Overloading

02

두울,

Static

03

세엣,

Operator Overloading

# 프렌드 (friend)

## friend 키워드

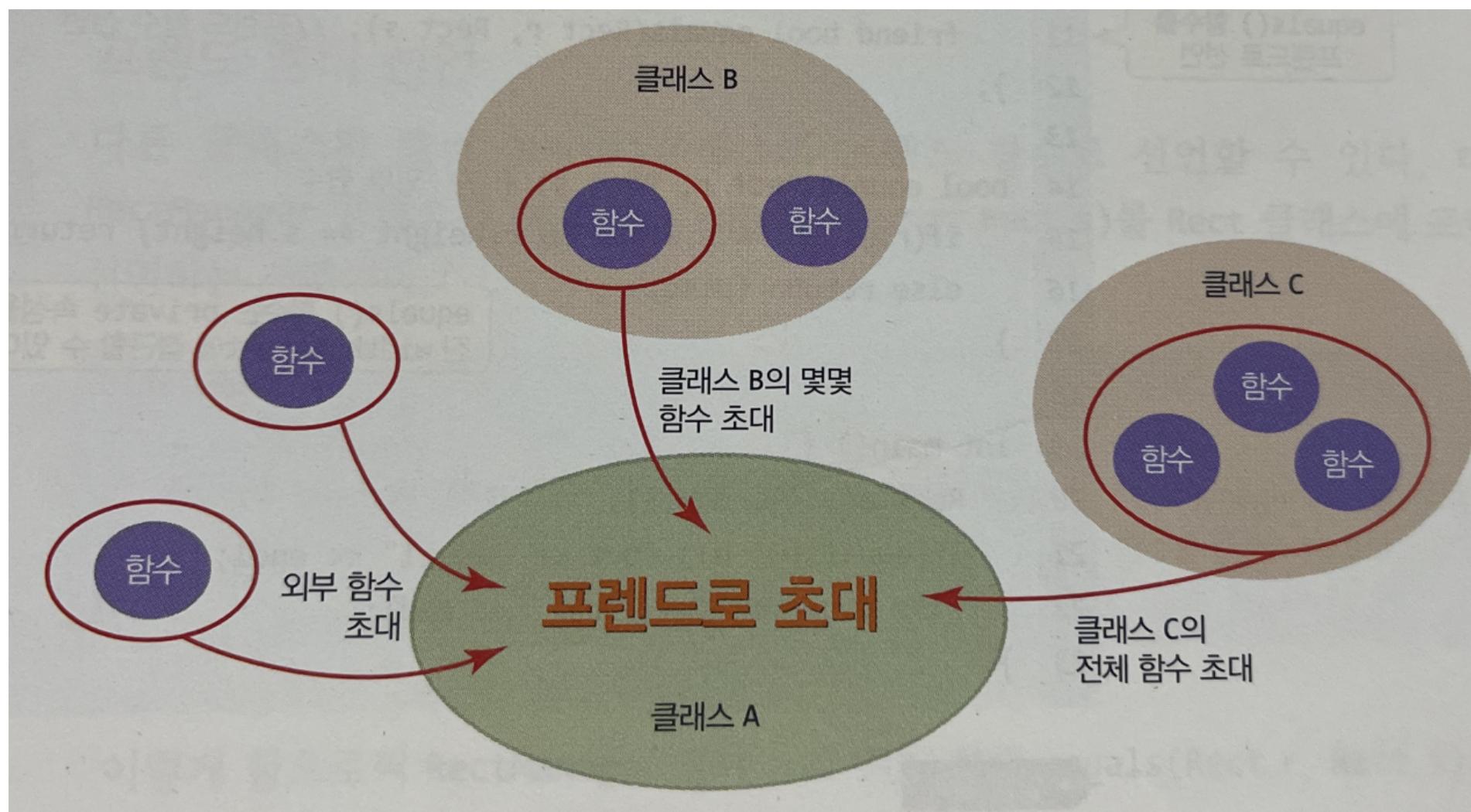
**class** 내에 friend 키워드로 선언된 외부 함수를 "프렌드 함수"라고 부르며,  
프렌드 함수는 클래스의 멤버인 것처럼 클래스의 모든 멤버에 접근 가능하다.

## 왜 필요할까?

클래스 멤버 함수로는 적합하지 않지만, 클래스의 private, protected  
멤버를 접근해야 하는 특별한 경우,  
이 함수를 외부 함수로 작성하고 프렌드로 선언한다. (대표적으로 연산자 함수)

# 프렌드 (friend)

- 1) 클래스 외부에 작성된 함수를 프렌드로 선언
- 2) 다른 클래스의 멤버 함수를 프렌드로 선언
- 3) 다른 클래스의 전체를 프렌드로 선언



# 프렌드 (friend)

```
class Rect
{
public:
    Rect(int width, int height)
    {
        this->width = width;
        this->height = height;
    }
    ~Rect();

    friend bool equals(Rect r, Rect s); // 외부 함수를 friend로 선언
    friend bool RectManager1::equals(Rect r, Rect s); // 다른 클래스의 함수를 friend로 선언
    friend RectManager2; // 다른 클래스 전체를 friend로 선언

    int getWidth() { return width; }
    int getHeight() { return height; }

private:
    int width;
    int height;
};
```

# 프렌드 (friend)

## 1) 클래스 외부에 작성된 함수를 프렌드로 선언

```
friend bool equals(Rect r, Rect s); // 외부 함수를 friend로 선언
```

```
// 1. 클래스 외부에 작성된 함수를 프렌드로 선언
bool equals(Rect r, Rect s)
{
    if (r.width == s.width && r.height == s.height)
    {
        return true;
    }
    return false;
}
```

```
if (equals(a, b))
{
    cout << "a == b" << endl;
}
else
{
    cout << "a != b" << endl;
}
```

# 프렌드 (friend)

## 2) 다른 클래스의 멤버 함수를 프렌드로 선언

```
friend bool RectManager1::equals(Rect r, Rect s); // 다른 클래스의 함수를 friend로 선언
```

```
class RectManager1
{
public:
    bool equals(Rect r, Rect s);
    //void copy(Rect& dest, Rect& src);
};
```

```
if (RM1.equals(a, b))
{
    cout << "a == b" << endl;
}
else
{
    cout << "a != b" << endl;
}
```

**friend 선언된 함수만 Rect 모든 멤버(private 포함)에 접근이 가능하지  
RectManager1에 있는 다른 함수가 가능한 것은 아니다!**

# 프렌드 (friend)

## 3) 다른 클래스의 전체를 프렌드로 선언

```
friend RectManager2; // 다른 클래스 전체를 friend로 선언
```

```
class RectManager2
{
public:
    bool equals(Rect r, Rect s);
    void copy(Rect& dest, Rect& src);
};
```

모든 멤버 함수가  
Rect 멤버에 접근할 수 있다.

```
RM2.copy(a, b);
if (RM2.equals(a, b))
{
    cout << "a == b" << endl;
}
else
{
    cout << "a != b" << endl;
}
```

# 연산자 중복 (Operator Overloading)

피연산자에 적합한 연산자를 새로 작성할 수 있다!

숫자 더하기 :  $2 + 3 = 5$

색 혼합 : 빨강 + 파랑 = 보라

좌표 더하기 :  $(1, 2) + (3, 4) = (4, 6)$

모두 + 기호로 표현되었지만 서로 다른 의미로 해석할 수 있다!

->다형성

# 연산자 중복 (Operator Overloading)

## 특징

1) C++ 언어에 있는 연산자만 Overloading이 가능하다!

예) +,-,\*,/,:==,!:=,%,&& 가능      ##,^^,%% 불가능

2) 피연산자의 개수를 바꿀 수 없다!

예) + 는 이항 연산자이므로, 피연산자가 1개 혹은 3개일 수 없다.

3) 연산자 중복으로 연산의 우선 순위를 바꿀 수 없다!

예) 연산자 중복을 하더라도 \*가 +보다 우선 순위가 높기 때문에 먼저 계산된다.

4) 모든 연산자가 중복이 가능한 것은 아니다!

예) . :\* :: ?: 이 친구들은 안된다.

# 연산자 중복 (Operator Overloading)

## 구현 방법

- 1) 클래스의 멤버 함수로 구현
- 2) 외부 함수로 구현하고 클래스의 프렌드 함수로 선언

## 이항 연산자 중복 - 클래스 내부

```
// + 이항 연산자
Power operator+(Power a)
{
    Power temp;
    temp.kick = this->kick + a.kick;
    temp.punch = this->punch + a.punch;

    return temp;
}
```

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

    c = a + b; // 컴파일러는 c = a.+b) 로 변형한다.
    c.show();
```

## 비교 연산자 중복 - 클래스 내부

```
// == 비교 연산자
bool operator==(Power a)
{
    if (this->kick == a.kick && this->punch == a.punch)
        return true;
    else
        return false;
}
```

```
if (a == d) // a.==(d)
    cout << "same" << endl;
else
    cout << "not same" << endl;

if (a == c)
    cout << "same" << endl;
else
    cout << "not same" << endl;
```

## 실습(5)

클래스 내부 함수로 += 연산자 오버로딩 하기

리턴 타입은 Power  
자기 자신을 반환 (즉, 객체를 반환)

## 실습(6)

클래스 내부 함수로 + 연산자 오버로딩 하기

예)  $b = a + 2$  (객체와 정수를 더하기)

예와 같이 a의 kick과 punch에 2를 모두 더하는 함수

## 단항 연산자 중복 - 클래스 내부

### 전위 연산자 ++, !

```
// 전위 ++ 단항 연산자
Power operator++()
{
    this->kick++;
    this->punch++;

    return *this;
}
```

```
// 전위 ! 단항 연산자
bool operator!()
{
    if (this->kick == 0 && this->punch == 0)
        return true;
    else
        return false;
}
```

```
++a; // a.++()
a.show();
```

```
if (!e)
    cout << "e is empty" << endl;
```

## 단항 연산자 중복 - 클래스 내부

### 후위 연산자 ++

```
// 후위 ++ 단항 연산자
Power operator++(int x)
{
    Power temp = *this; // 증가 이전의 객체 상태 저장
    this->kick++;
    this->punch++;

    return temp; // 증가 이전의 객체 리턴
}
```

```
b = a++; // a.++(임의의 정수)
a.show(); // a의 파워는 1 증가
b.show(); // b는 a가 증가되기 이전의 상태를 가짐
```

## 프렌드를 이용한 연산자 오버로딩

연산자 함수는 클래스 바깥의 외부 전역 함수로도 작성 가능하다.

**Power a(3, 4), b;**

**b = 2 + a; -> 2.+(a) // 2는 객체가 아니다!**

따라서 위의 식은 잘못된 문장이다.

이처럼 첫 번째 피연산자가 객체가 아닌경우 컴파일러는 아래와 같이 변환한다.

**b = 2 + a; -> +(2, a)**

따라서 + 연산자를 Power 클래스의 외부 함수로 밖에 구현할 수 없다!

# 이항 연산자 중복 - 프렌드 이용

```
friend Power operator+(int op1, Power op2);  
friend Power operator+(Power op1, Power op2);
```

```
c = a + b; // c = + (a, b)  
c.show();  
c = 2 + a; // c = + (2, a)  
c.show();
```

```
// + 이항 연산자 정수 + 객체  
Power operator+(int op1, Power op2)  
{  
    op2.kick = op1 + op2.kick;  
    op2.punch = op1 + op2.punch;  
  
    return op2;  
}
```

```
// + 이항 연산자 객체 + 객체
Power operator+(Power op1, Power op2)
{
    Power temp;
    temp.kick = op1.kick + op2.kick;
    temp.punch = op1.punch + op1.punch

    return temp;
}
```

## 실습(7)

프렌드를 이용하여 == 비교 연산자 오버로딩 하기

예) if (a == b)  
cout << "Same" << endl;

## 단항 연산자 중복 - 프렌드 이용

```
// 전위 ++ 단항 연산자
Power operator++(Power& op)
{
    Power temp;
    op.kick++;
    op.punch++;

    return op;
}
```

```
friend Power operator++(Power& op);
friend Power operator++(Power& op, int x);
```

```
// 후위 ++ 단항 연산자
Power operator++(Power& op, int x)
{
    Power temp = op; // 증가 이전의 객체 상태 저장
    op.kick++;
    op.punch++;

    return temp; // 증가 이전의 객체 리턴
}
```

```
++a; // ++ (a)
a.show();
```

b = a++; // ++ (a, 0) 0은 의미 없는 값  
a.show(); // a의 파워는 1 증가  
b.show(); // b는 a가 증가되기 이전의 상태를 가짐

## 실습(8)

**프렌드를 이용하여 ! 단항 연산자 오버로딩 하기**

**예) kick, punch가 모두 0이라면 true return**

## 그래서 결론은?

개발하다보면 연산자 함수를 클래스의 멤버로 할 것인지,  
외부 함수로 작성할 것인지 선택해야한다.

새로운 연산자는 클래스와 연계하여 작동하기 때문에, 클래스 멤버로 선언하면  
외부의 연산자 함수를 `friend`로 취할 필요도 없고 가독성도 높아진다.

따라서 가능하면 클래스의 멤버 함수로 작성하기를 권한다.

**2 + a 와 같은 어쩔 수 없는 상황에선 `friend`를 사용하자!**

## 과제

연산자는 굉장히 많다. < > != & ^ ~ [] () 등등

Point class를 만들자.

멤버로는 int x, int y 좌표를 변수로 가지고 있다.

좌표를 더하거나 빼고, 여러가지 기능을 만들어보자!

I2PXO2CUBE

---

Week5

---

방찬웅

# Q&A 및 토킹

감사합니다!