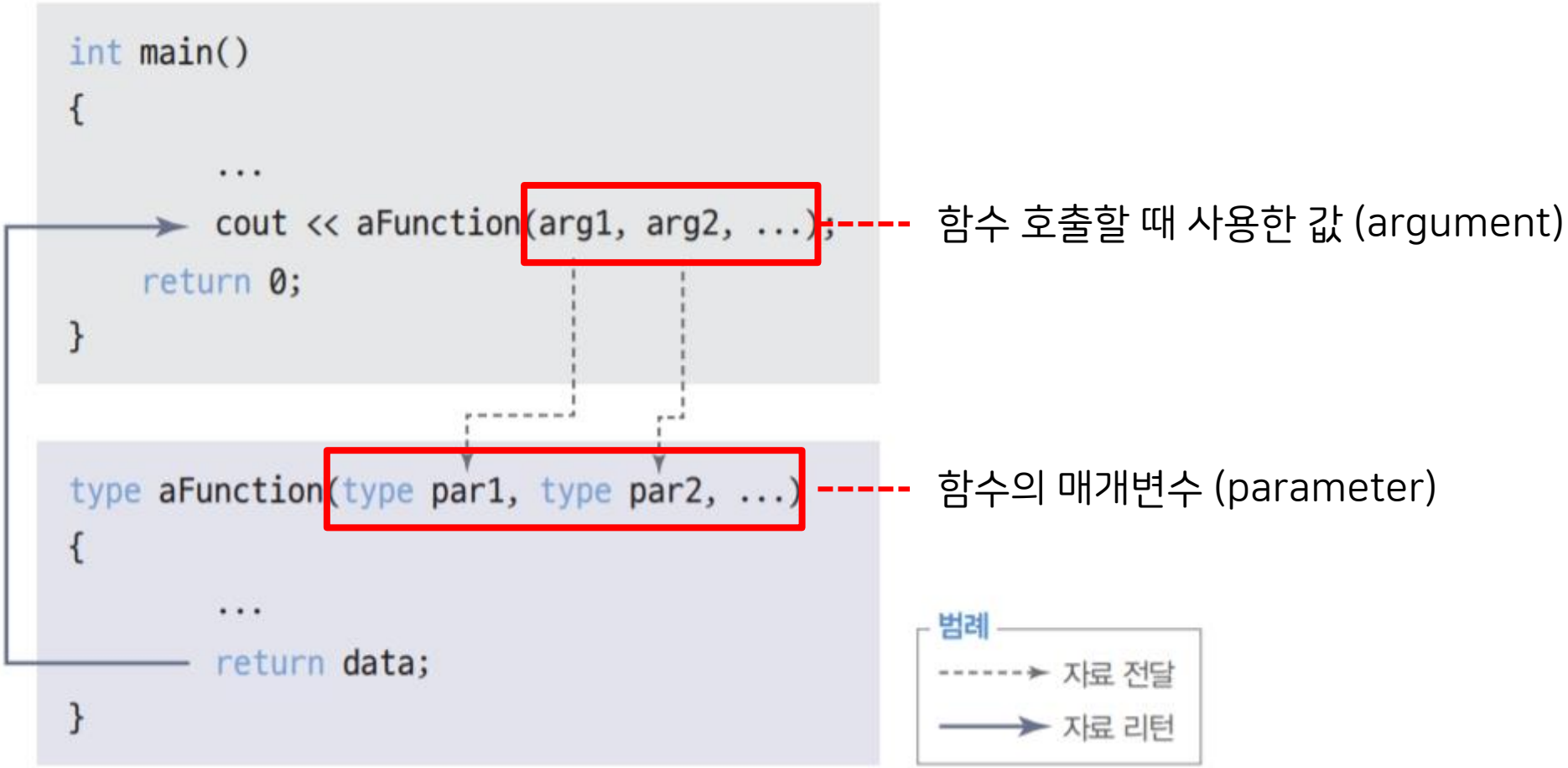
C++ / JAVA 프로그래밍

06주차 실습

함수 사용하기2

자료 교환의 개념

- 일반적으로 함수 사이의 커뮤니케이션은 아래 그림과 같이 양방향으로 이루어짐



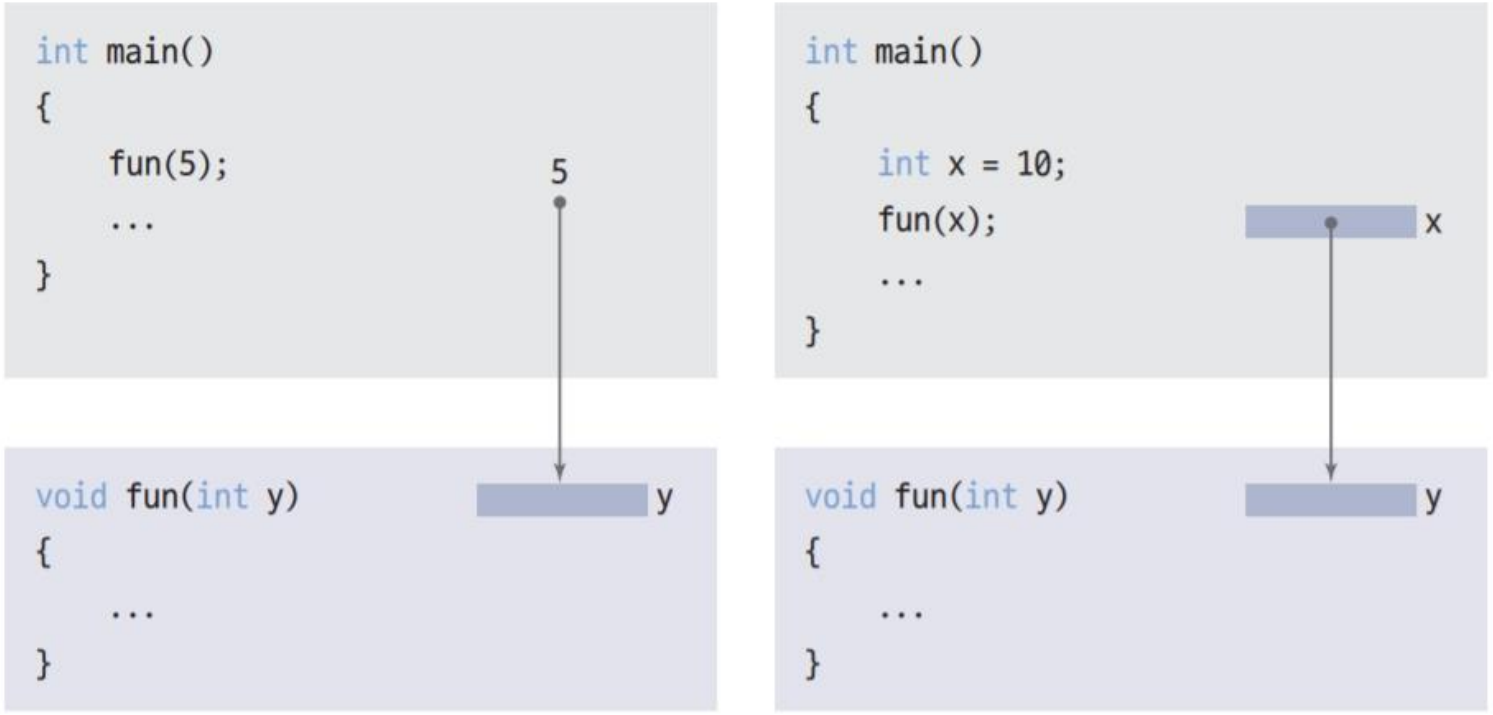
자료를 전달하는 방법

- 호출되는 함수에 매개변수가 존재한다면,
  - 함수 호출 때의 값(argument)이 호출되는 함수의 매개변수(parameter)로 전달

- 자료 전달은 다음과 같이 3가지 메커니즘으로 구분함
  - 값으로 호출(call-by-value), 또는 값으로 전달(pass-by-value)
  - 주소로 호출(call-by-address), 또는 포인터로 전달(pass-by-pointer)
  - 참조로 호출(call-by-reference), 또는 참조로 전달(pass-by-reference)

값으로 호출 (call-by-value)

- 인수(argument)의 값이 복사되어서 매개변수(parameter)에 할당됨
- 호출되는 함수 쪽에서 인수를 변경하지 않게 만들고 싶을 때 사용
- 호출되는 함수는 인수의 값을 읽기만 할 수 있으므로 '읽기 전용 접근(read-only access)'이라 표현함



(a) 리터럴 값 전달하기

(b) 변수의 값 전달하기

값으로 호출 사용 예

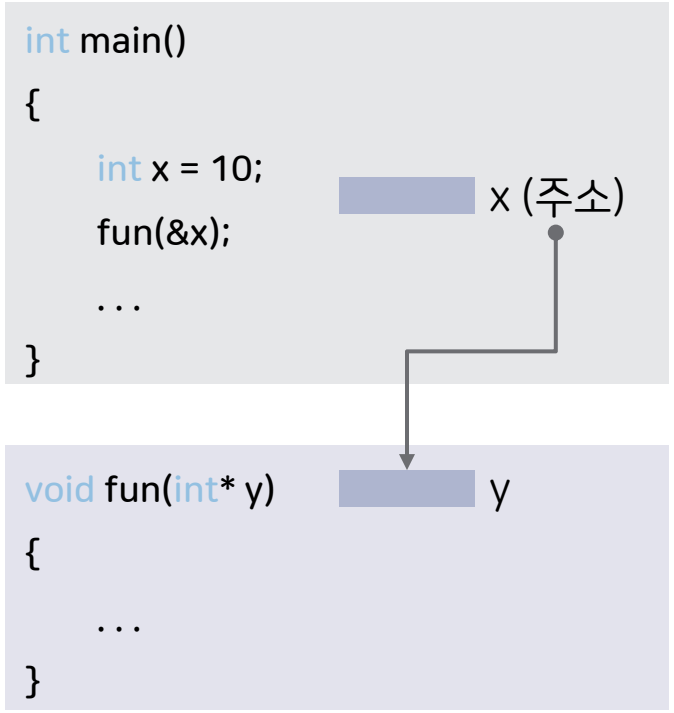
```
#include <iostream>
using namespace std;

// 함수 선언
void fun (int y);

int main () {
    // 변수선언 및 초기화
    int x = 10;
    // 함수 fun을 호출하면서 인자로 x를 전달
    fun(x);
    // 함수 호출 후 x값 출력
    cout << "Value of x in main: " << x << endl;
    return 0;
}

void fun (int y) {
    y++;
    cout << "Value of y in fun: " << y << endl;
    return;
}
```

실행결과:

Value of y in fun: 11
Value of x in main: 10

주소로 호출 (call-by-address)

- 인수로 메모리 주소를 매개변수에 전달하는 것
- 주소를 전달하므로 매개변수를 사용해서 인수의 메모리 위치에 접근할 수 있음
- 참조 호출 기능이 없는 C언어에서 주로 사용되었음
- C++에서는 많이 사용하진 않지만, 전달할 자료가 포인터 특성을 갖는 경우 사용함



주소로 호출 사용 예

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

// 함수 선언, y에 '정수를 저장한 주소'를 저장한다는 의미
void fun (int* y);

int main () {
    // 변수선언 및 초기화
    int x = 10;
    // 함수 fun을 호출하면서 인자로 x의 주소를 전달
    fun(&x);
    // 함수 호출 후 x값 출력
    cout << "Value of x in main: " << x << endl;
    return 0;
}

void fun (int* y) {
    (*y)++;
    cout << "Value of y in fun: " << *y << endl;
    return;
}
```

실행결과:

Value of y in fun: 11
Value of x in main: 11

참조로 호출 (call-by-reference)

- 인수와 매개변수는 메모리 주소를 공유
- 각각의 함수에서 변수 이름은 다를 수 있지만, 양쪽 함수 모두 같은 메모리 공간에 접근해서 값을 읽거나 씀
- 매개변수 앞의 &는 컴파일러에게 별도의 메모리 영역을 사용하지 말라고 알려주기 위한 표시

```
int main()
{
    int x = 10;
    fun(x);
    ...
}

void fun(int& y)
{
    ...
}
```

fun 함수에서는 변수 y를 위한
메모리 공간이 추가로 할당되지 않음

매개변수로 전달된 변수 x의
메모리 공간을 사용

참조로 호출 사용 예

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

// 함수 선언, &기호는 y가 별칭이라는 의미
void fun (int& y);

int main () {
    // 변수선언 및 초기화
    int x = 10;
    // 함수 fun을 호출하면서 인자로 x를 전달
    fun(x);
    // 함수 호출 후 x값 출력
    cout << "Value of x in main: " << x << endl;
    return 0;
}

void fun (int& y) {
    y++;
    cout << "Value of y in fun: " << y << endl;
    return;
}
```

실행결과:

```
Value of y in fun: 11
Value of x in main: 11
```

자료 전달 방식들의 장단점

- 값으로 호출
  - 매우 간단하고, 호출되는 함수 쪽에서 인수를 조작하지 않게 차단
  - 인수의 값을 복사해서 매개변수로 전달하므로 전달해야 하는 값의 크기가 작다면 유용함
  - 객체 지향 프로그래밍에서 객체의 크기가 클 경우, 값으로 호출 메커니즘을 사용하지 않음
- 주소로 호출
  - 참조로 호출 메커니즘과 같은 장점을 가짐
  - 일반적으로 C++에서는 많이 사용하지 않음
  - 전달해야 하는 자료가 포인터의 특성(예를 들어서 C 언어 문자열, 배열 등)을 갖고 있을 경우 사용함
- 참조로 호출
  - 호출되는 함수 쪽에서 매개변수를 변경해서, 호출한 함수 쪽의 원본(argument)을 변경할 수 있음
  - 스왑 등을 구현할 때 가장 좋은 선택
  - 이 메커니즘은 복사가 필요하지 않다는 장점이 있음

자료를 반환하는 방법

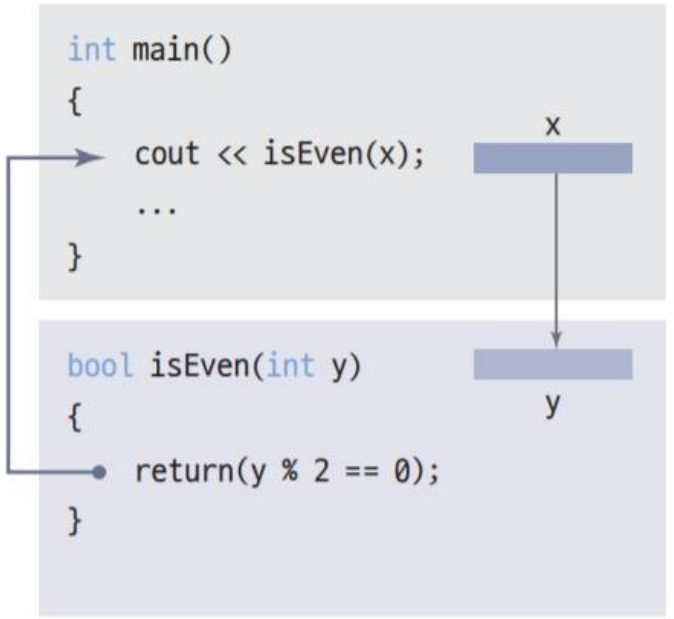
- 리턴값의 종류에 따라 다음과 같이 3가지 메커니즘으로 구분함
  - 값으로 리턴(return-by-value)
    - 가장 일반적으로 사용되는 메커니즘
    - 호출되는 함수 쪽에서 어떤 표현식을 생성하고, 이를 반환
    - 함수를 호출하면 값이 반환되므로, 값이 필요한 위치에 함수를 직접 활용할 수 있음
  - 참조로 리턴(return-by-reference)
    - 객체 지향 프로그래밍 메커니즘에서는 크기가 큰 객체를 반환해야 하는 경우가 있음
    - 이때 복사로 인해서 생기는 비용을 줄이려면, 참조로 리턴하는 것이 좋음
    - 호출되는 함수에서 객체를 생성하면, 함수가 종료된 이후에 객체가 사라지므로 여러 주의 사항을 지켜야 함
  - 포인터로 리턴(return-by-pointer)
    - 포인터로 리턴 메커니즘은 참조로 리턴 메커니즘과 효과가 같지만 거의 사용하지 않음

값으로 리턴 사용 예

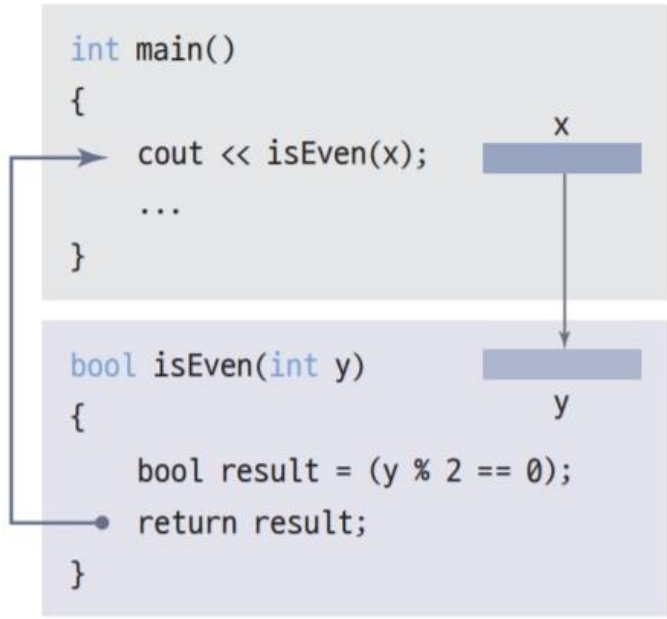
```
#include <iostream>
using namespace std;

// 함수 선언
bool isEven (int y);

int main ( ) {
    // 함수 호출
    cout << boolalpha << isEven (5) << endl;
    cout << boolalpha << isEven (10);
    return 0;
}
/*********************************************
 * isEven 함수는 매개변수로 y 하나를 받음.
 * 나머지 연산을 통해 y값이 짝수인지 검사함.
 * 검사 후 결과 값을 반환함
 *********************************************/
bool isEven (int y) {
    return ((y % 2) == 0);
}
```

실행결과:

false
true



(a) 리터럴 값 리턴하기



(b) 변수의 값 리턴하기

기본 매개변수

- 기본 매개변수(default parameter)를 사용해서 변수에 기본값(default value)을 지정할 수 있음

- 일부 매개변수만 기본 매개변수를 적용하려면 오른쪽에 위치하는 매개변수들에만 적용할 수 있음

- 예를 들어 다음 함수는 매개변수 hours 자리에 인자를 넣지 않으면 40.0이 사용됨
  - `double calcEarnings(double rate, double hours = 40.0);`

- 직원이 40시간 일하면 calcEarnings 함수에 rate만 매개변수로 전달해서 호출할 수 있음
- 만약 직원이 40시간 미만으로 일한 경우에는 rate와 hours를 매개변수로 모두 전달해서 호출함
  - `calcEarnings(payRate);`
  - `calcEarnings(payRate, hourWorked);`

기본 매개변수 사용 예

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

// 함수 선언, 두 번째 인자의 기본 값을 40으로 설정
double calcEarnings (double rate, double hours = 40);

int main ( ) {
    // 첫 번째 함수 호출은 두 번째 인자를 넣는 대신 기본 값을 사용함
    cout << "Emplyee 1 pay: " << calcEarnings (22.0) << endl;
    cout << "Emplyee 2 pay: " << calcEarnings (12.50, 18);
    return 0;
}
/**********************************************************
 * 이 함수는 두 개의 매개변수를 사용함
 * 함수 선언에서 이미 기본 값을 설정한 경우, 함수 정의에서 매개변수의 기본 값을 다시 설정할 필요가 없음
 *  **********************************************************/
double calcEarnings (double rate, double hours) {
    double pay;
    pay = hours * rate;
    return pay;
}
```

실행결과:

Emplyee 1 pay: 880
Emplyee 2 pay: 225

함수 오버로딩 (Function Overloading)

- 이름이 같은 함수를 2개 정의할 수 있을까?
  - 매개변수(매개변수의 자료형, 개수, 순서)가 다르다면 가능
  - 이를 함수 오버로딩(function overloading)이라고 부름

- 프로그램이 같은 이름의 함수를 허용할 때, 함수들을 구분하기 위해 사용하는 기준을 함수 시그니처(function signature)라고 부름
  - 함수 시그니처는 **매개변수들의 자료형과 조합**
  - 함수 시그니처가 다르다면, 호출 시점에 컴파일러가 자신이 어떤 함수를 호출해야 하는지 구분할 수 있음
  - 함수 호출 시점에는 리턴값을 어떻게 활용할지 구분할 수 없으므로, **리턴 자료형은 함수 시그니처에 포함되지 않음**

함수 시그니처 예시

- 다음은 두 int 자료형과 double 자료형의 최대값을 찾기 위해 다른 함수를 정의하는 예시

```
int max(int a, int b)
{
    ...
}
```

```
double max(double a, double b)
{
    ...
}
```

- 다음 두 함수는 함수 시그니처가 같으므로 오버로딩할 수 없음

```
int get()
{
    ...
}
```

```
double get()
{
    ...
}
```

함수 오버로딩 사용 예

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

int max (int num1, int num2) {
    return num1 >= num2?num1:num2;
}

int max (int num1, int num2, int num3) {
    return max(max(num1, num2), num3);
}

int max (int num1, int num2, int num3, int num4) {
    return max(max(num1, num2, num3), num4);
}

int main () {
    cout << "maximum (5, 7): " << max (5, 7) << endl;
    cout << "maximum (7, 9, 8): " << max (7, 9, 8) << endl;
    cout << "maximum (14, 3, 12, 11): " << max (14, 3, 12, 11);
    return 0;
}
```

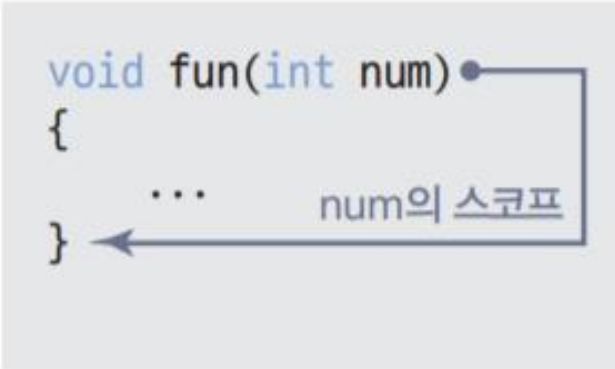
실행결과:

```
maximum (5, 7): 7
maximum (7, 9, 8): 9
maximum (14, 3, 12, 11): 14
```

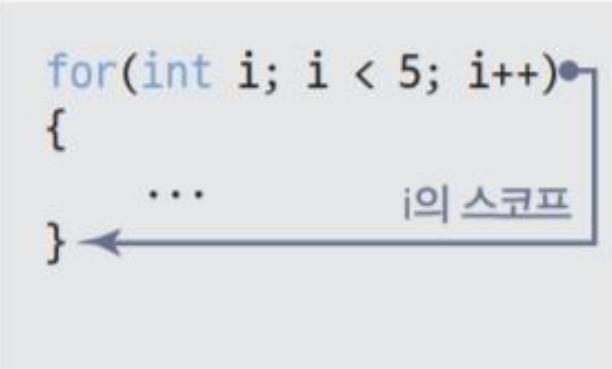
엔티티의 스코프(사용 범위)

- **스코프**는 어떤 엔티티(상수, 변수, 객체, 함수 등)를 사용할 수 있는 범위 나타냄

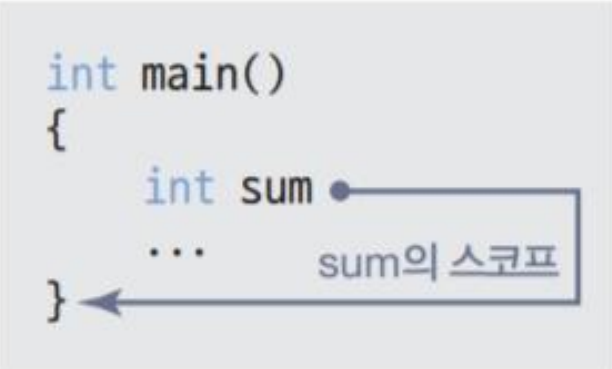
- 지역 스코프 (local scope)
  - 지역 스코프를 가진 엔티티는 선언된 위치부터 블록이 끝나는 부분(닫는 중괄호) 내부에서 사용할 수 있음



(a) 매개변수의 스코프



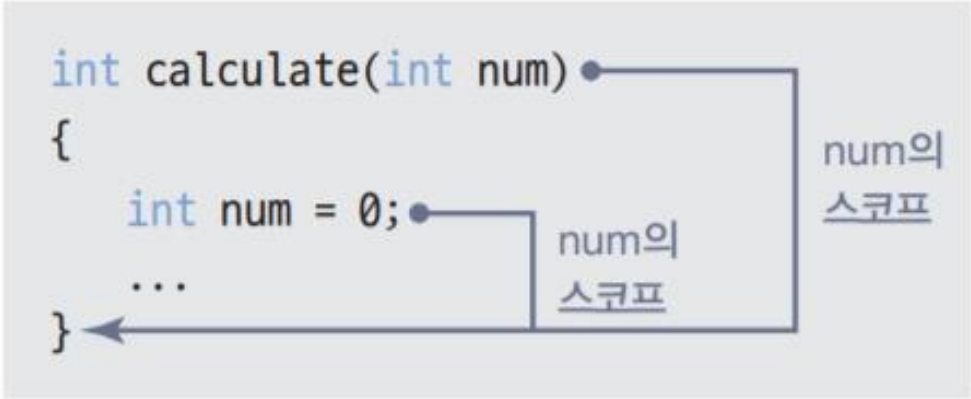
(b) 카운터의 스코프



(c) 지역변수의 스코프

스코프 겹침 문제

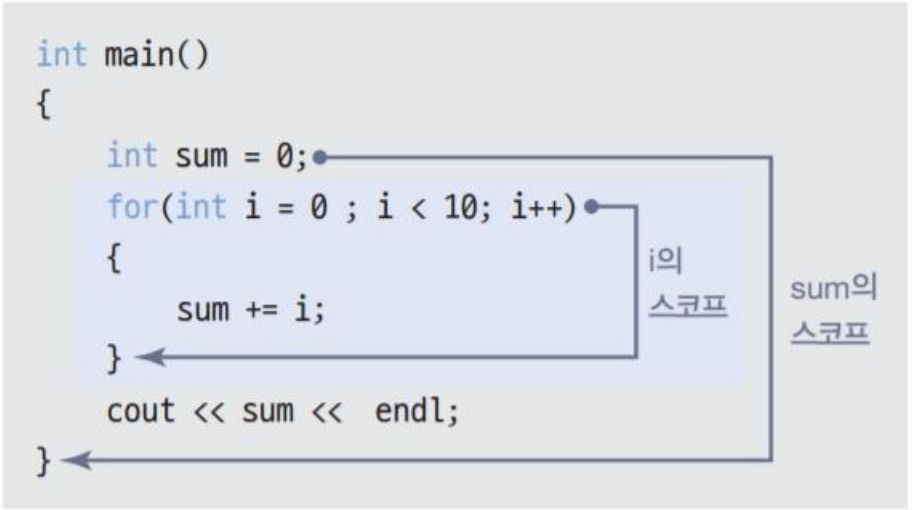
- 블록 내부에 같은 이름을 가진 엔티티 2개가 올 수는 없음
- 이러한 경우에는 컴파일 오류가 발생



단일 블록의 스코프 겹침(오류 발생)

중첩 블록의 스코프

- 프로그램을 작성하다보면 블록이 중첩되는 경우가 있음
  - 외부 블록에 선언한 엔티티는 내부 블록에서도 사용할 수 있는 넓은 스코프를 가짐
  - 내부 블록에 선언한 엔티티는 내부 블록에서만 사용할 수 있는 좁은 스코프를 가짐
  - 만약 내부 블록에 선언한 엔티티를 외부 블록에서 사용할 경우 오류 발생



중첩 블록의 스코프

지역 스코프의 쉐도잉(shadowing)

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

int main ( ) {
    int sum = 5;
    cout << sum << endl;
    {
        int sum = 3;
        cout << sum << endl; // 내부 블록의 sum이 보임
    }
    cout << sum << endl; // 외부 블록의 sum이 보임
    return 0;
}
```

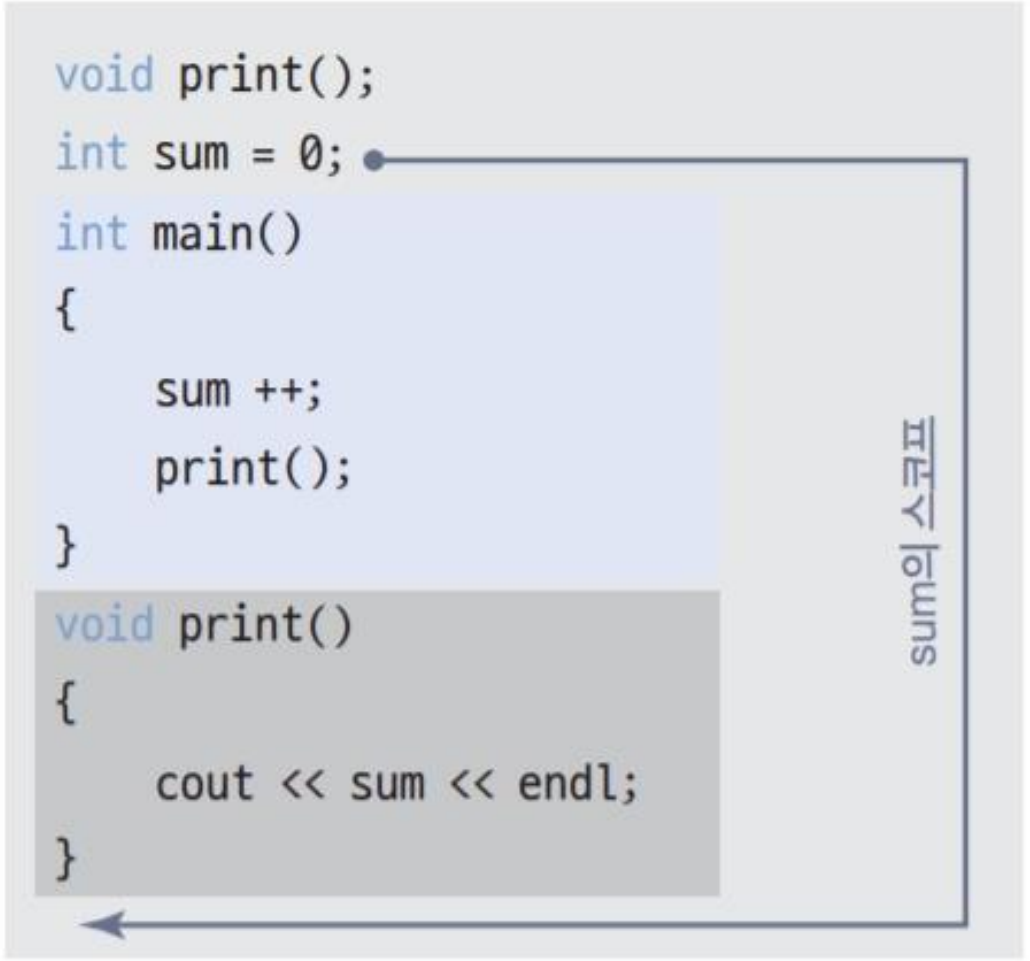
실행결과:

```
5
3
5
```

전역 스코프 (global scope)

- 모든 함수의 외부에 선언된 엔티티는 전역 스코프를 가짐
- 전역 엔티티의 스코프는 프로그램의 끝 부분까지임

- 두 함수 외부에서 sum이라는 변수가 선언된 경우
  - main() 함수와 print() 함수 모두에서 sum 사용 가능



전역 엔티티의 스코프

전역 스코프 쉐도잉

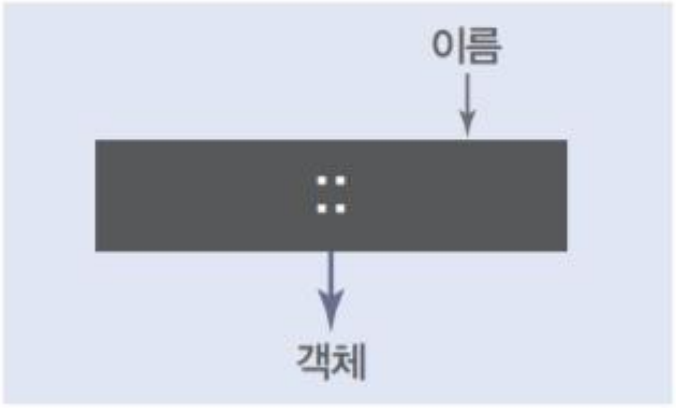
```
#include <iostream>
using namespace std;

int num = 5; // 전역 변수 선언 및 초기화

int main ( ) {
    cout << num << endl;     // 전역 변수 num 출력
    int num = 25;            // 지역 변수 num 선언 및 초기화
    cout << num;             // 지역 변수 num이 전역 변수 num을 가림
    return 0;
}
```

실행결과:

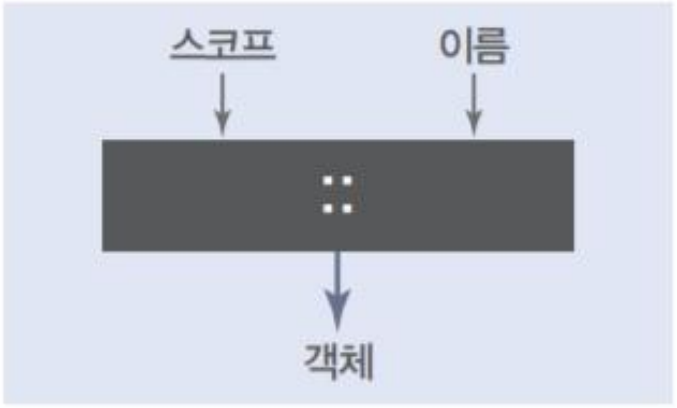
```
5
25
```

범위 해결 연산자(Scope Resolution Operator)

- 지역 엔티티가 전역 객체를 가리는 셰도잉을 무시하고, 전역 엔티티에 접근해야 하는 경우도 있음
- C++은 이러한 때를 위해 범위 해결 연산자(::)를 제공.
- 피연산자를 하나만 사용하는 경우, 암묵적으로 전역 스코프를 지정함
- 피연산자를 두 개 사용하는 경우, 특정 스코프의 엔티티를 지정할 수 있음



(a) 피연산자가 하나인 형태



(b) 피연산자가 두 개인 형태

범위 해결 연산자 사용 예

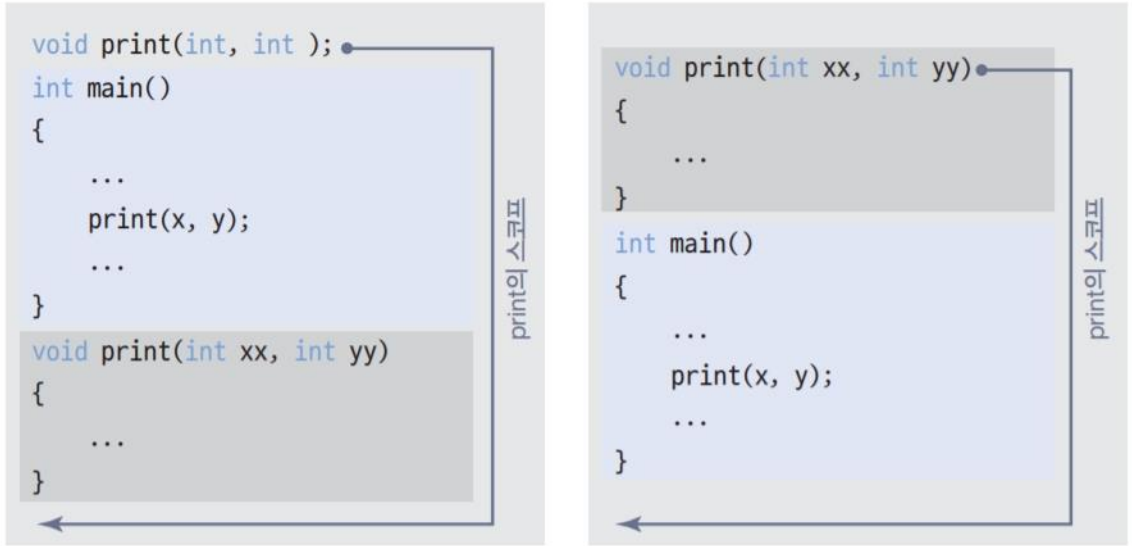
```
#include <iostream>
using namespace std;

int num = 5; // 전역 변수 선언 및 초기화

int main ( ) {
    int num = 25; // 지역 변수 선언 및 초기화
    cout <<" Value of Global num: " << ::num << endl;
    cout <<" Value of Local num: " << num << endl;
    return 0;
}
```

실행결과:

```
Value of Global num: 5
Value of Local num: 25
```

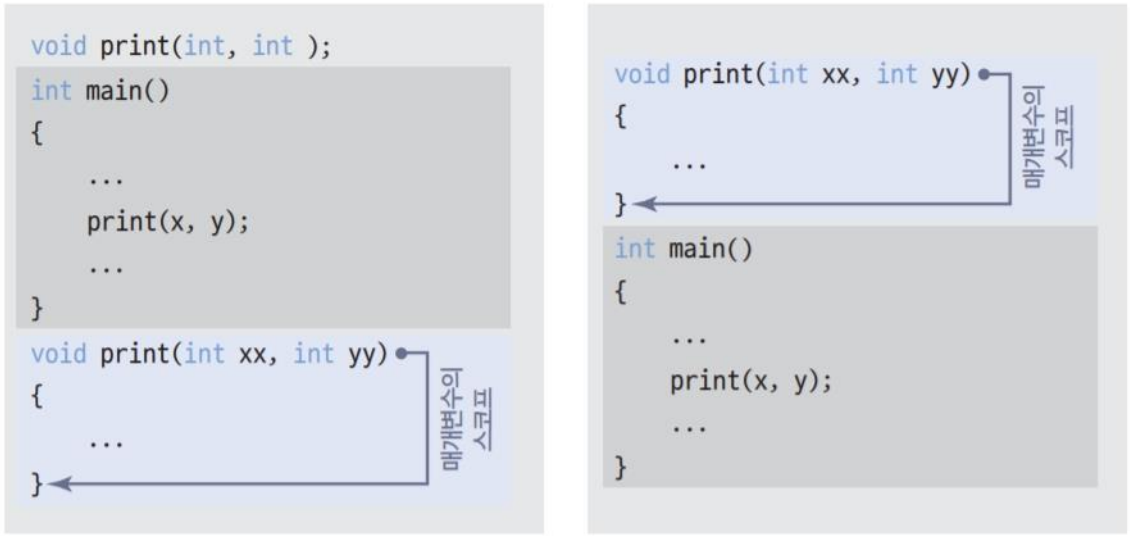
함수 이름과 매개변수의 스코프

- 함수는 이름을 가진 엔티티며, 이 이름에도 스코프가 있음
- 함수는 함수를 선언한 시점부터 프로그램의 마지막 부분까지를 스코프로 가짐
- 함수 매개변수도 이름이 있는 엔티티이므로 스코프를 가지며 함수 헤더부터 함수 블록 끝까지를 스코프로 가짐



(a) 프로토타입이 있는 경우 (b) 프로토타입이 없는 경우

함수 이름의 스코프



(a) 프로토타입이 있는 경우 (b) 프로토타입이 없는 경우

함수 매개변수의 스코프

수명

- 자동 지역 변수
  - 자동 지역 변수(automatic local variable)는 함수가 호출될 때 생성되고, 함수가 종료될 때 소멸
  - 기본적으로 함수 내부의 모든 지역 변수들은 자동 지역 변수

- 정적 지역 변수(static local variable)는 static 변경자를 앞에 붙여서 만듦
  - 정적 지역 변수는 프로그램이 종료되기 전까지 유지
  - 한 번만 초기화되며 프로그램이 살아있는 동안(실행되고 있는 동안), 프로그램은 메모리 상의 변수를 추적
  - 따라서 함수를 여러 번 호출하면, 모든 함수들이 같은 변수를 공유

정적 지역 변수 사용 예

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

void fun ( );

int main ( ){
    fun ( );
    fun ( );
    fun ( );
    return 0;
}

void fun ( ) {
    static int count = 0; // 명시적 정적변수 선언
    count++;
    cout << "count = " << count << endl;
}
```

실행결과:

count = 1
count = 2
count = 3

만약 static을 붙이지 않았다면:

count = 1
count = 1
count = 1

변수의 초기화

- 지금까지 살펴보았던 변수의 종류(전역 변수, 자동 지역 변수, 정적 지역 변수)의 초기화를 비교
  - 자동 지역 변수는 초기화하지 않으면, 메모리에 남아있는 쓰레기 값(garbage value)을 가짐
  - 전역 변수와 정적 지역 변수는 초기화하지 않으면 기본값(정수는 0.0, 부동 소수점은 0.0, 불은 false)으로 초기화
  - 전역 변수와 정적 지역 변수는 같은 형태로 초기화된다는 것을 기억


HANYANG
UNIVERSITY