컴퓨터 프로그래밍 I
(CSE2003-3)

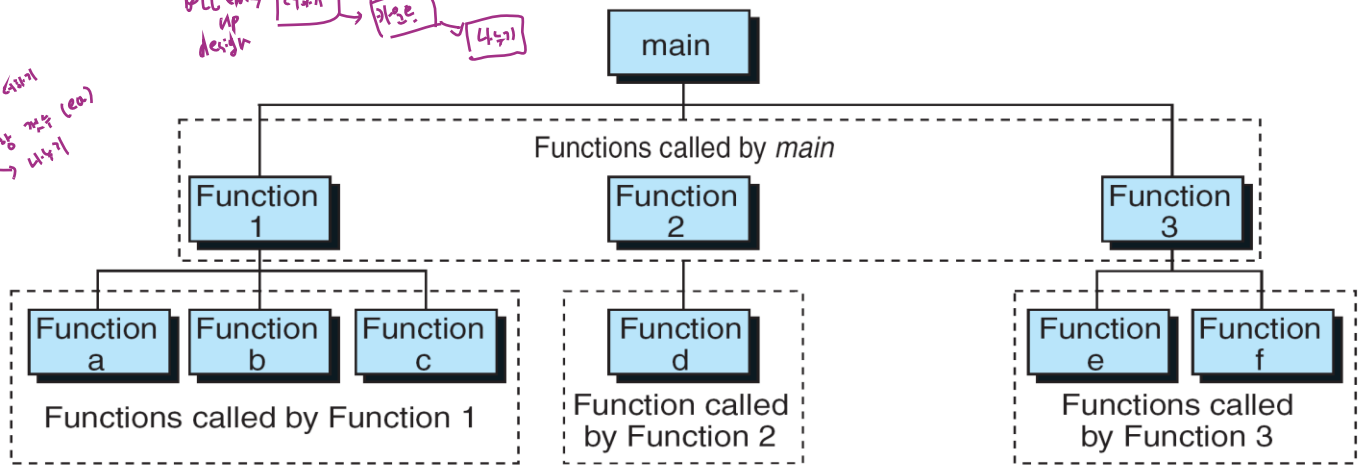
Mon/Wed 16:30-17:45 pm

Lecture 7

Function (함수)

4.1 Designing Structured Programs

- **Top-down design**
  - 어려운 문제를 더 단순하고 쉬운 작은 문제로 나누어서 단계적으로 구체화하는 방법
  - 각 단순화한 문제를 계속하여 더 단순한 프로그램으로 분리하여 구현함으로써 단계적으로 프로그램을 하게 함 (problem decomposition)
- **Structure Chart(구조 차트)**
  - Main 함수를 calling module이라고 부르고, 하위 함수들을 called module이라고 부른다.(1) 함수를 호출할 때 데이터를 전달할 수 있어야 하는데, **parameter passing** 방법으로 전달한다.



<구조차트(structure chart)>

(1) 중간에 있는 함수들은 calling module도 되고 called module 도 됨

4.2 function in C

- C언어에서 함수는 특정한 작업을 수행하기 위하여 불려지는 독립적인 단위 프로그램(Module).
- C언어는 함수를 활용해서 top-down design된 구조차트를 구현
  - C 프로그램은 하나 또는 하나 이상의 함수를 통해 만들어지며, main함수는 반드시 하나.
- C언어에서의 함수사용은 다음과 같은 장점 제공 (Modularization:모듈화)
  - 문제를 분리하여 단순화 시킴: one module for one simple problem
  - 함수는 다른 프로그램에서도 쉽게 코드를 재사용(reuse) 할 수 있도록 함
  - 유용한 여러 함수들이 미리 개발되어 library 형태로 제공되도록 함으로써 프로그램 개발이 매우 편리해짐
  - 함수는 데이터를 보호 (Encapsulation: 캡슐화)
    - 함수 안에서 선언된 지역변수는 그 함수 안에서만 사용 가능하며, 그 함수가 실행될 때만 사용됨. 외부 함수는 다른 함수의 지역변수를 건드릴 수 없음!!!

- 함수의 종류
  - Standard functions
  - User-defined functions

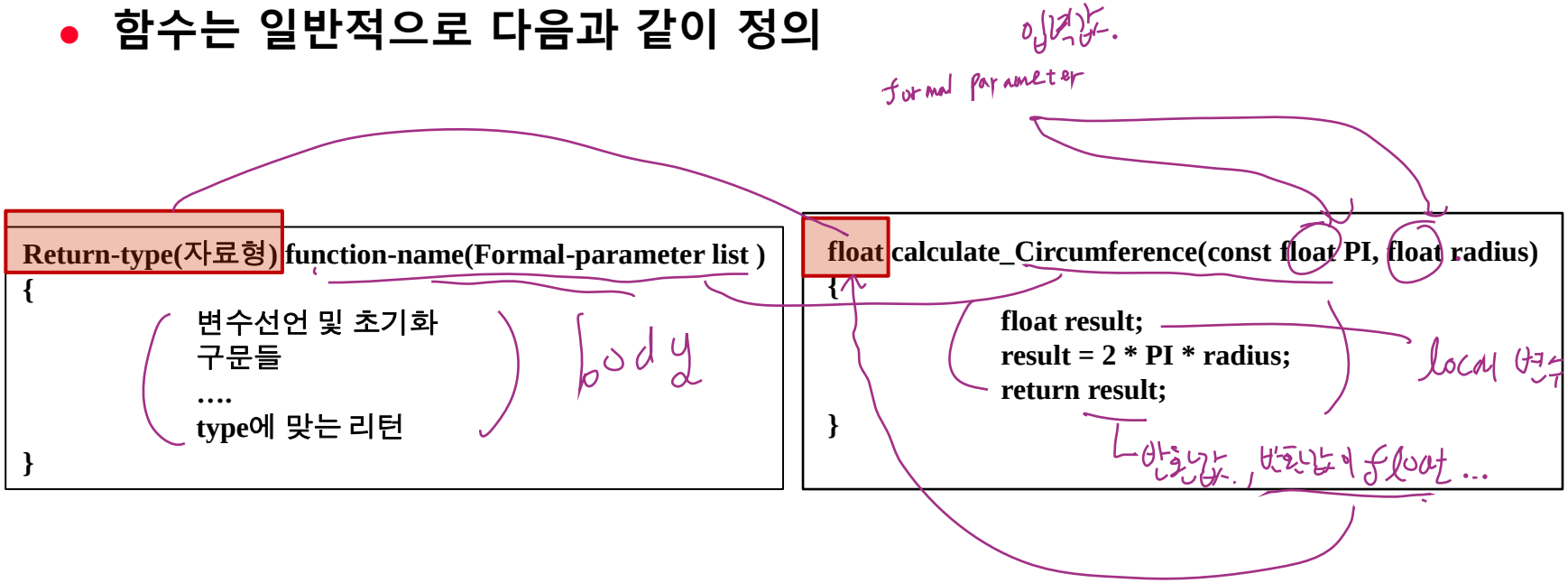
우리는 이미 scanf, printf 등을 통해 함수를 사용해 보았다. 이러한 함수들은 사용자 정의(User-defined) 에 의해 만들어진 함수가 아니라 C에서 라이브러리 형태로 제공하는 표준 함수(Standard functions) 이다.

Designing Structured Programs

- **C프로그래밍에서 함수**
  - main 프로그램으로부터 인수(argument)를 전달 받아 일련의 작업을 수행한 후 생성된 결과를 main 프로그램으로 전달하는 하나의 단위 프로그램
  - 수학에서의 함수의 정의와 용도가 비슷함!!!
- **함수를 사용하는 이유**
  - 반복되는 코드 작성을 없애 줌
  - 프로그램을 이해하기 쉽고 이식(porting)과 재사용을 용이하게 함
  - 모듈성(modularity) 향상시킴
- **함수 간의 통신**
  - 인수(parameter)의 전달과 반환 되는 값(return value)
    - Parameter passing은 actual parameter와 formal parameter 매칭 방식으로 수행
    - Return되는 값은 하나임!! (수학적인 함수와 동일)
  - 전역 변수(global variable)를 이용 (이것은 절대로 사용하지 않는 것이 좋음!!!)

함수 정의(Function Definition)

- 함수는 일반적으로 다음과 같이 정의



- Return-type
  - 함수의 수행의 끝나고 caller에게로 결과값을 리턴할 때 그 결과값의 데이터형(type) 지정 (예. int, float, char, etc.)
  - 만일 caller에게 리턴하는 결과값이 없는 경우 return-type으로 키워드 void 사용
    (ex) void main(void)

int main(void) → int라서 return 0; 하는것.
void main(void)

함수 정의(Function Definition)

- 함수는 일반적으로 다음과 같이 정의

```
Return-type(자료형) function-name(Formal-parameter list )
{
        변수선언 및 초기화
        구문들
        ….
        type에 맞는 리턴
}
```

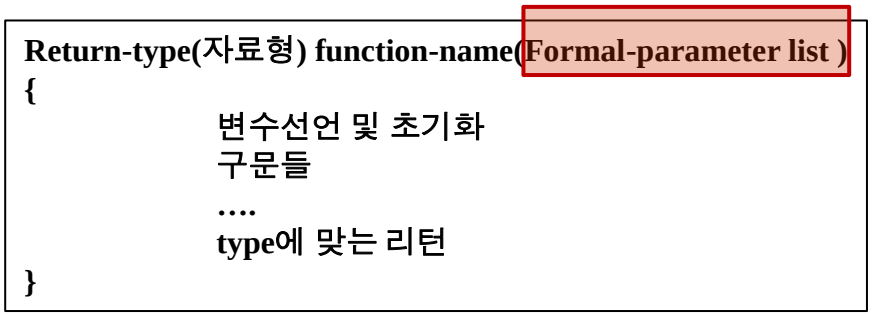
```
float calculate_Circumference(const float PI, float radius)
{
        float result;
        result = 2 * PI * radius;
        return result;
}
```

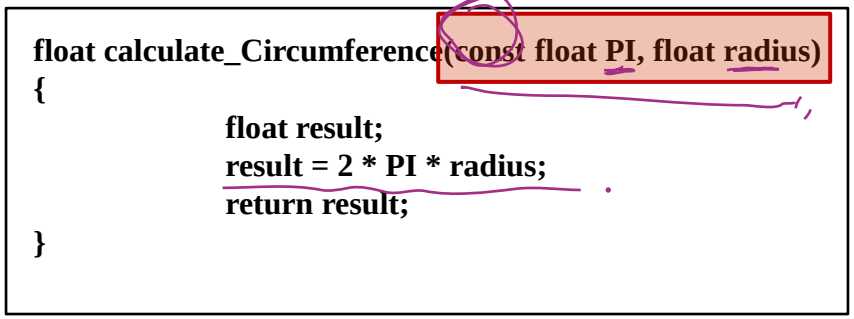
- Function-name
  - 함수의 이름은 그 기능의 의미를 알 수 있도록 만들면 됨
  - Standard function들의 이름과 겹치게 정의하면 에러가 남

ex) printf...

함수 정의(Function Definition)

- 함수는 일반적으로 다음과 같이 정의





- Formal-parameter list
  - 이 함수를 호출하는 calling 함수가 보내주는 데이터 형식을 정의한 리스트
  - 데이터 타입과 그 데이터를 받는 변수 이름을 지정한 리스트. 이 변수들은 이 함수에서 사용되는 변수임

함수 정의(Function Definition)

- 함수는 일반적으로 다음과 같이 정의

```
Return-type(자료형) function-name(Formal-parameter list )
{
        변수선언 및 초기화
        구문들
        ....
        type에 맞는 리턴
}
```

```
float calculate_Circumference(const float PI, float radius)
{
        float result;
        result = 2 * PI * radius;
        return result;
}
```

- 변수 선언 및 초기화, 명령들
  - 이 함수가 수행되기 위해서 필요한 변수들의 추가 선언 및 프로그램 명령문
  - Formal parameter list에서 선언된 변수들은 추가 선언 없이 그대로 사용 가능

함수 정의(Function Definition)

- 함수는 일반적으로 다음과 같이 정의

```
Return-type(자료형) function-name(Formal-parameter list )
{
        변수선언 및 초기화
        구문들
        ....
        type에 맞는 리턴
}
```

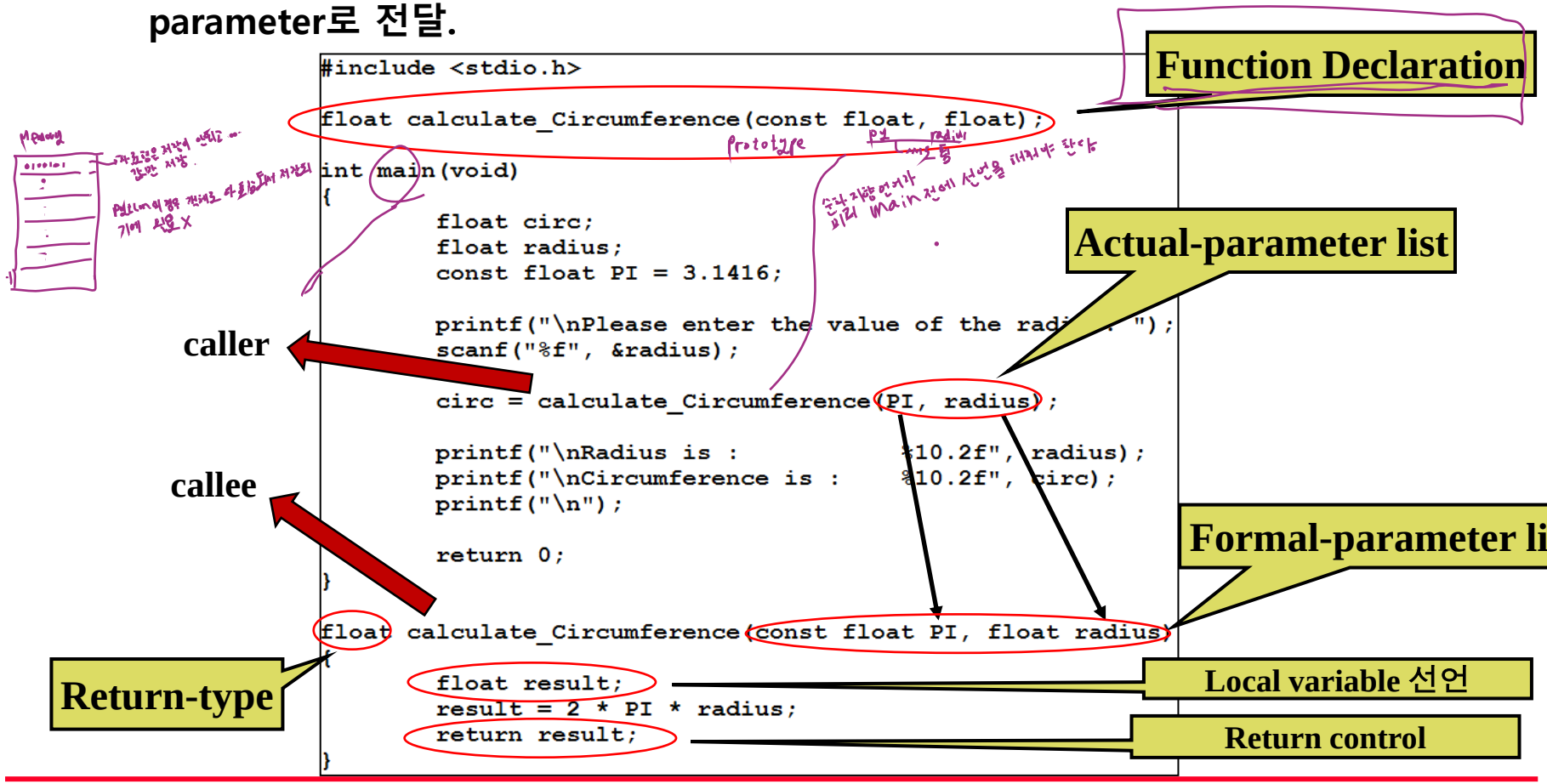
```
float calculate_Circumference(const float PI, float radius)
{
        float result;
        result = 2 * PI * radius;
        return result;
}
```

- Return 명령
  - 함수의 수행의 끝나고 caller에게로 결과값을 리턴하는 명령.
  - 모든 함수는 반드시 return 명령문이 있어야 하는 것이 원칙이므로 습관화 하는 것이 좋음. Void 타입의 함수의 경우라도 그냥 "return ;"이라는 명령을 추가하는 것이 올바른 코딩 방법

null statement..

함수 정의(definition)와 선언(declaration)

- Formal-parameter list
  - 호출하는 함수(caller)가 호출할 때 호출당하는 함수(callee)에서 사용할 데이터를 인수(parameter)로 전달
  - Parameter list는 comma-separated list로 각 argument는 순서대로 해당 parameter로 전달.



함수 정의(Function Definition)

- **함수 선언(Function Declaration)**
  - **컴파일러에게 함수의 이름과, 함수가 갖는 parameter의 데이터형, 그리고 함수에 의해 반환되는 결과 값의 데이터형을 알려줌.**
  - 컴파일러에게 calculate-Circumference 라는 함수의 존재와 이 함수가 float type의 값을 준다는 사실을 미리 알려주어야 컴파일 에러가 나지 않는다

- **Return 명령**
  - **리턴 값이 없을 경우 (즉, void 타입)**

    Return;

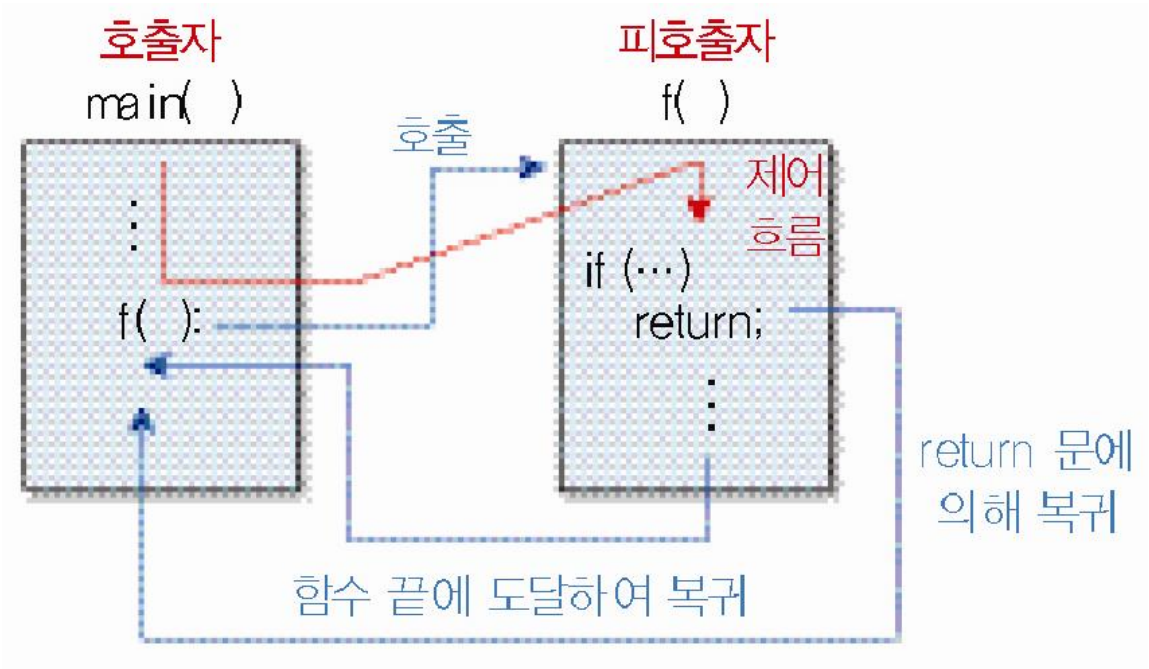
  - **리턴 값이 있을 경우**

    Return *expression;*

- **함수호출에서 형식 매개변수 (formal parameters) 와 실 매개변수 (actual parameters) 의 개수와 타입 및 순서는 반드시 일치해야 함**
  - **실 매개변수(caller가 보내는 데이터)와 형식 매개변수(caller가 보낸 데이터를 받는 변수)의 이름은 같을 필요가 없으며, 순서에 따라 서로 연결**

제어흐름과 자료흐름(control flow & data flow)

- **제어흐름 (control flow)**
  - 프로그램 수행 순서를 제어흐름(control flow)라고 함
- **자료흐름 (data flow)**
  - 데이터가 전달되는 순서를 자료흐름(data flow)라고 함



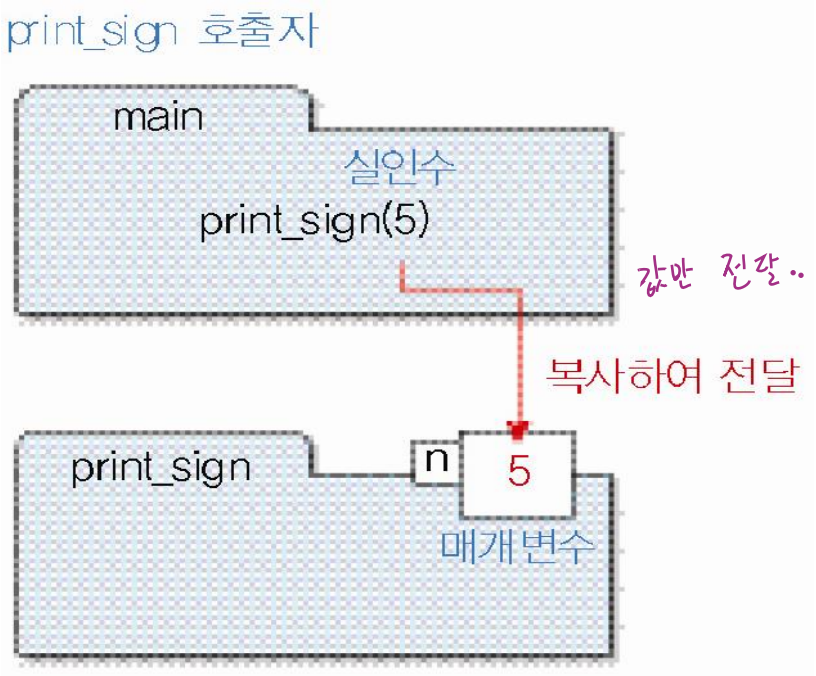
제어흐름(control flow)

자료흐름(data flow) – 인수 전달

- **함수 인수(argument)는 값을 구하여 매개변수(parameter)에 "복사"하여 전달한다.**
  - 함수 사이의 자료흐름은 인수 전달(parameter passing)과 반환 값 전달에 의해 결정됨
- **예:**

```
void print_sign(int n)
{
    printf("정수 %d는 ", n);
    if (n > 0)
        printf("양수입니다.\n");
    else if (n < 0)
        printf("음수입니다.\n");
    else
        printf("양수도 음수도 아닙니다.\n");
}
```

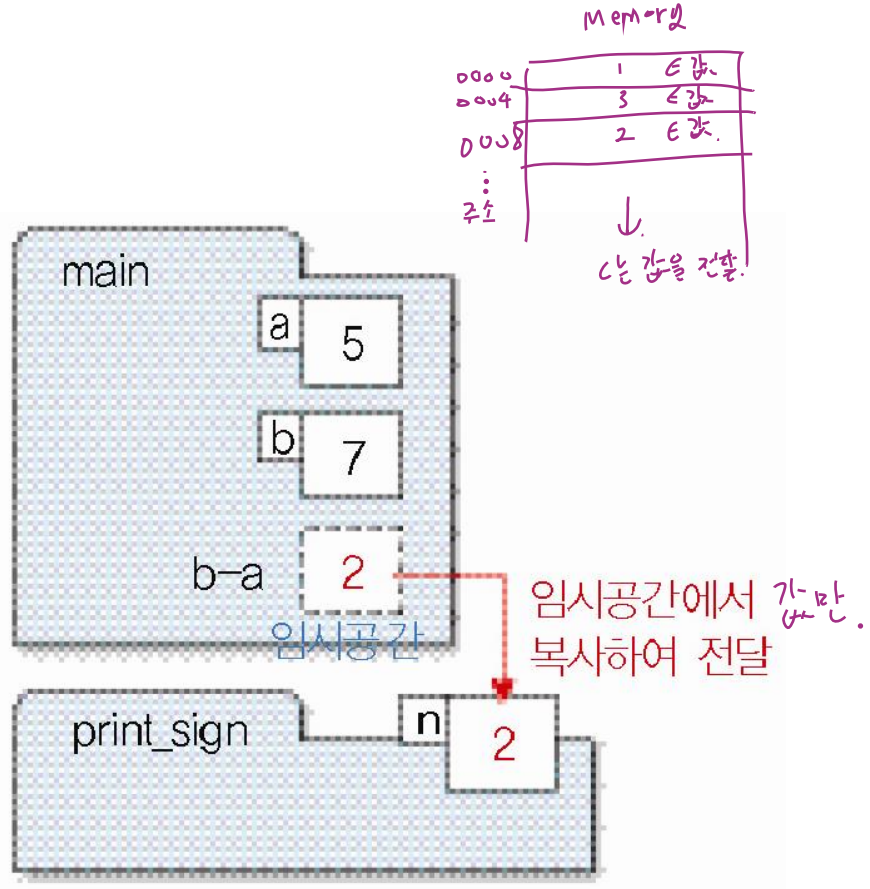
main 함수에서
print_sign(5);를 호출했을 경우



자료흐름(data flow) – 임시변수

- 인수가 수식인 경우에는 호출자의 임시공간(임시변수)를 활용하여 수식 값을 구한 후 전달한다.
- **예:**
  - 다음과 같이 호출한 경우 수식 b – a 값은 main의 임시공간에서 계산된다.

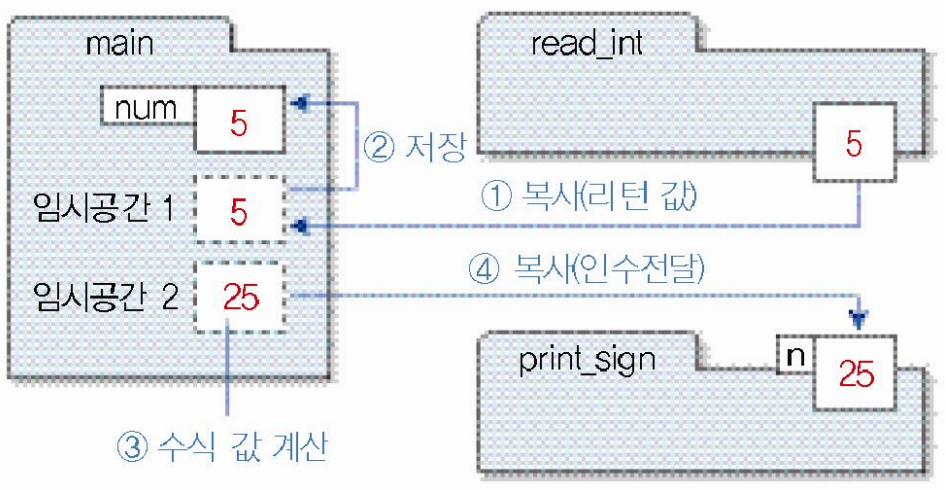
```
int main (void)
{
        int a=5;
        int b=7;
        print_sign (b-a) ;
        return 0;
}
```



자료흐름(data flow) – 리턴값

- **값 반환 함수 호출** Call-by-Value
  - 값 반환 함수 호출은 수식으로 간주하기 때문에 리턴 값도 역시 호출자의 임시변수에 저장한다.
  - 값 반환 함수는 사용자 정의 '연산자'라는 사실을 기억하자.
- **예:**
  - 다음과 같이 호출한 경우 read_int() 값과 num*num 값은 임시공간에 저장된다.

```
int main (void)
{
        int num;
        num = read_int();
        print_sign(num*num);
        return 0;
}
```



함수 호출 메커니즘

- **함수 f가 함수 g를 호출했을 때, 함수호출 메커니즘**
  - (1) f의 본체 내에서 g에 보낼 실 인수 값을 계산한다.
  - (2) g에 대한 데이터 영역이 생성된다.
  - (3) 실 인수 값을 g의 매개변수로 복사한다. 인수가 여러 개일 때, 복사 순서는 컴파일러마다 다르다.
  - (4) 제어흐름이 f에서 g로 전달된다. f는 수행을 멈춘다.
  - (5) g의 본체 끝에 도달하거나 수행 중 return 문에 도달할 때까지 g의 본체 내 문장을 수행한다.
  - (6) 리턴 수식이 있는 return 문에 도달한 경우 리턴 값을 구한다.
  - (7) 리턴 값을 f 영역 내의 임시변수에 복사한다.
  - (8) 제어흐름이 g에서 f로 전달된다. g 영역은 메모리에서 소멸된다.
  - (9) f의 본체가 이전에 멈추었던 위치 바로 다음부터 계속 수행된다.

예제 프로그램 2 - 나눗셈

예제 소스

```c
#include <stdio.h>

int Calc_devide(int, int);
int Calc_remainder(int, int);

int main (void)
{
    int intNum1;
    int intNum2;
    int intCalc;

    printf("Enter two integral number: ");
    scanf("%d %d", &intNum1, &intNum2);

    intCalc = Calc_devide(intNum1, intNum2);
    printf("%d / %d is %d", intNum1, intNum2, intCalc);

    intCalc = Calc_remainder(intNum1, intNum2);
    printf(" with a remainder of: %d\n", intCalc);

    return 0;
}

int Calc_devide(int in1, int in2)
{
    int result;
    result = in1 / in2;
    return result;
}

int Calc_remainder(int in1, int in2)
{
    int result;
    result = in1 % in2;
    return result;
}
```

```
kimjihwan@cspro:~/test_dr$ gcc -o calc.o calc.c
kimjihwan@cspro:~/test_dr$ ./calc.o
Enter two integral number: 13 2
13 / 2 is 6 with a remainder of: 1
kimjihwan@cspro:~/test_dr$
```

결과

함수 호출(function call)시 인수 전달 방법

- **Caller는 Callee에게 여러 개의 데이터 값을 인수로 보낼 수 있지만, Callee는 Caller에게 단 한 개의 값만 Return할 수 있다면 함수간 소통이 너무 제한적이다!!!**
  - **다양한 인수전달(parameter passing) 방식을 통하여 간단히 해결**

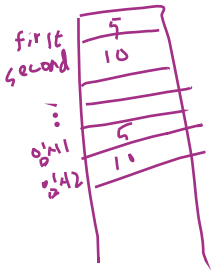
- **Parameter passing**
  - C 언어에서 함수를 호출할 때 인수를 전달하는 방법이 두 가지가 있다.
    - **값에 의한 전달 (Pass by Value 또는 Call by Value)**
      - 호출한 함수(caller)의 argument(인수) 값을 전달하는 방법
      - 호출한 함수(caller)의 argument(인수) 값이 절대로 변하지 않는다.
    - **참조에 의한 전달 (Pass by Reference 또는 Call by Reference)**
      - 호출한 함수(caller)의 argument(인수) 주소를 전달하는 방법
      - 호출한 함수(caller)의 argument(인수) 값도 변할 수 있다.

예제 프로그램 2 – Pass by value

예제 소스

```
#include <stdio.h>

void add(int first, int second);

int main(void)
{
    int first;
    int second;

    first = 5;
    second = 10;

    add(first, second);
    printf("first = %d, second = %d\n", first, second );

    return 0;
}

void add(int first, int second)
{
    first = first + second;
}
```

add()함수가 호출되면 그 함수 내에서 계산이 수행되어 first의 값을 바꾸지만 그것은 복사본의 계산일 뿐이다.



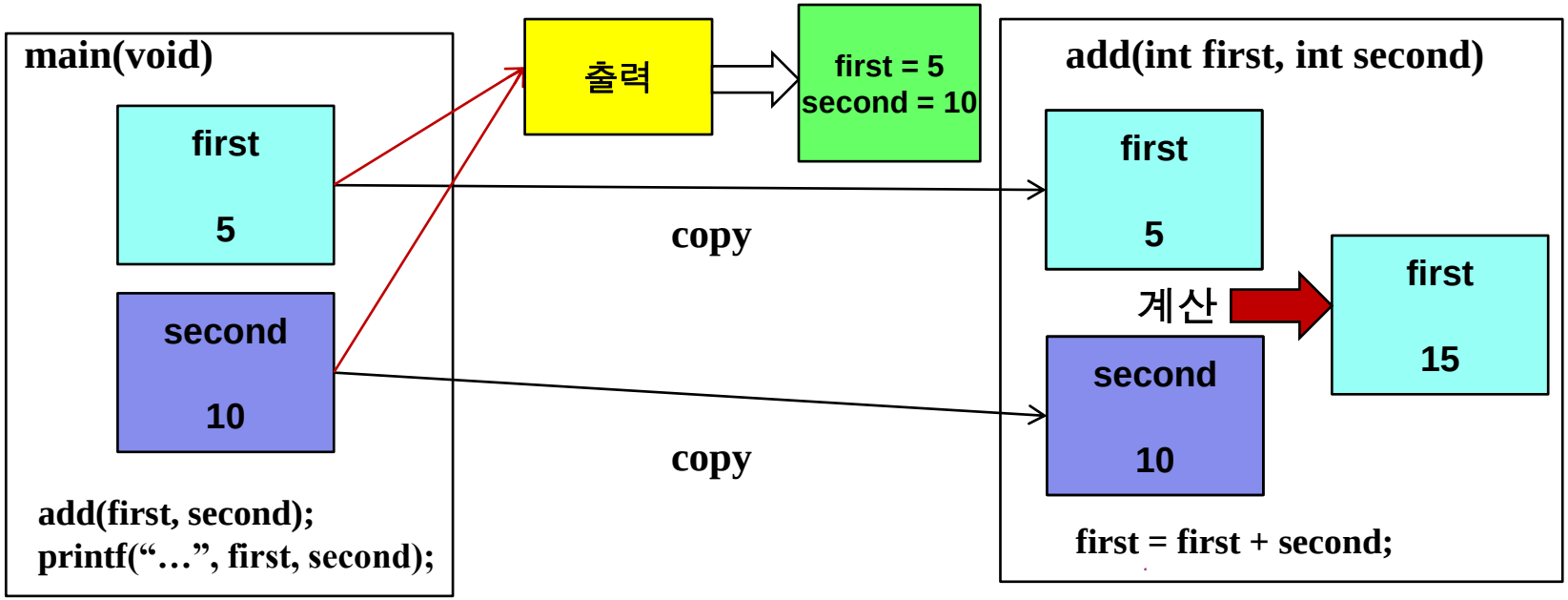
Caller의 인자와 Callee의 인자는 지역 변수와 같이 서로 다른 메모리에 저장되어 있어 아무 관계가 없다.

결과

```
kimjihwan@cspro:~/test_dr$ gcc -o pass.o pass.c
kimjihwan@cspro:~/test_dr$ ./pass.o
first = 5, second = 10
kimjihwan@cspro:~/test_dr$
```

Pass by value

- **Pass by value (Call by value)**
  - 값에 의한 전달이란 뜻은 Caller에서 함수를 호출할 때 argument의 값(value)을 Callee로 전달한다는 것 (즉, 인수의 값만 복사됨)
  - actual parameter와 formal parameter가 서로 다른 메모리에 위치
  - actual parameter의 값을 formal parameter에 복사하여 사용
  - Formal parameter의 값을 바꾸어도 actual parameter의 값은 바뀌지 않음



예제 프로그램 1 - Pass by reference

```c
#include <stdio.h>

void add(int * first_addr, int*  second_addr);

int  main(void)
{
    int first;
    int second;

    first = 5;
    second = 10;

    add(&first, &second);
    printf("first = %d, second = %d\n", first, second );

    return 0;
}

void add(int* first_addr, int* second_addr)
{
    *first_addr = *first_addr + *second_addr;
}
```

- main은 add() 함수를 호출하면서 변수 first와 second의 주소 넘겨줌
- add()는 이를 포인터(주소를 저장하는 변수) first_addr, second_addr에 받음
- 두 값을 더함
- 실행결과 main함수에서 두 변수의 값이 바뀌는 것을 확인할 수 있음

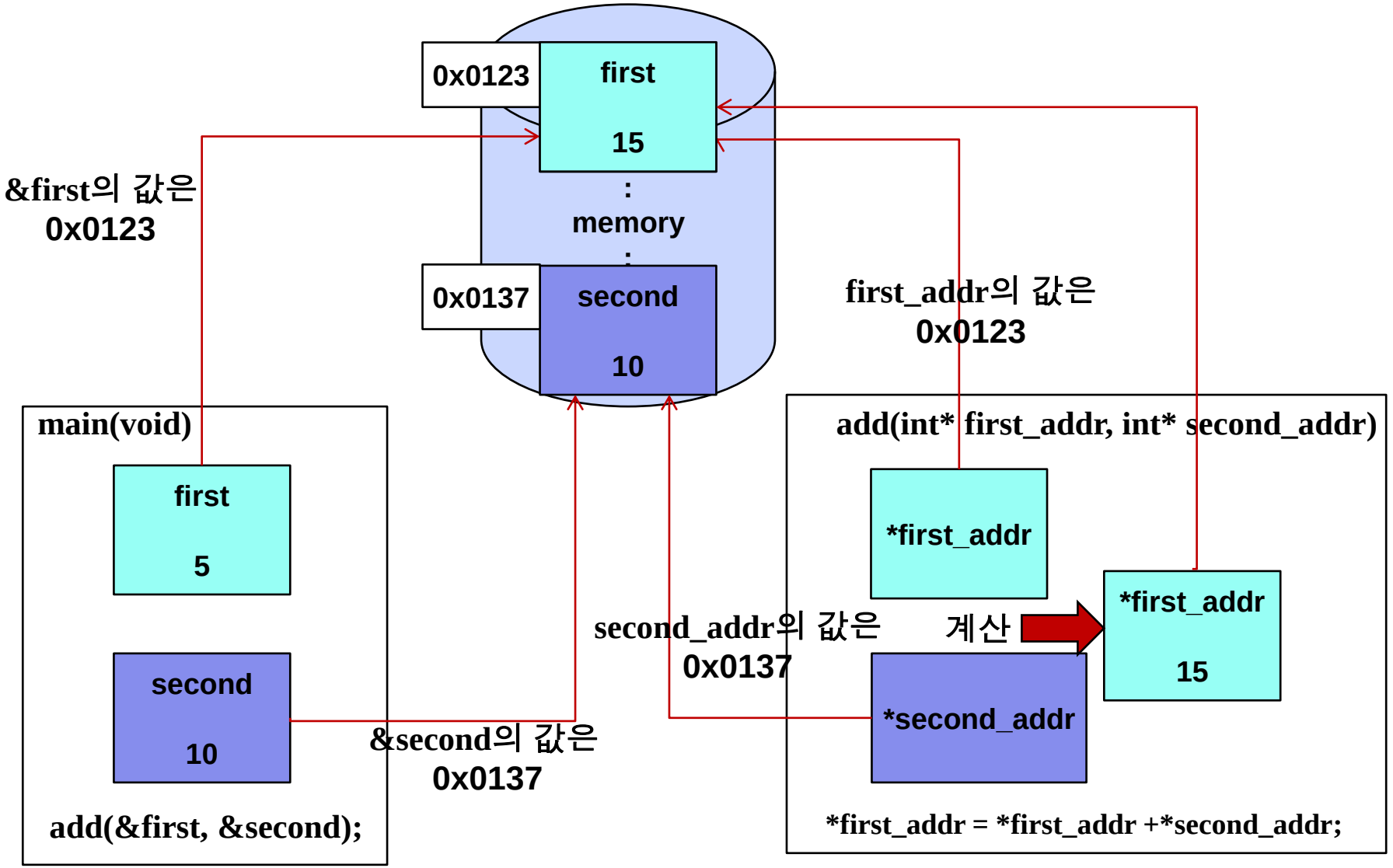
결과

```
kimjihwan@cspro:~/test_dr$ gcc -o pass.o pass.c
kimjihwan@cspro:~/test_dr$ ./pass.o
first = 15, second = 10
kimjihwan@cspro:~/test_dr$
```

Pass by Reference

- **Pass by Reference (Call by Reference)**
  - **참조에 의한 전달은 caller가 callee에게 argument의 값을 전달하는 것이 아니라, 'argument의 메모리 주소(address)값'을 전달하는 것**
  - **메모리 주소 값을 전달하면 callee는 그 주소 값을 이용하여 caller의 actual parameter를 access할 수 있어서 그 것의 값을 바꿀 수 있음**
- **Address operator (&) : "Give me the address of this data"**
  - 변수의 주소를 얻어냄: ex) **`&first, &second`**
  - caller 측에서 사용 (변수의 주소를 구해서 callee에게 넘겨줌)
- **Indirection operator (*) : "Use this value as an address to find the data"**
  - 주소에 해당하는 위치(또는 그곳의 값)를 나타냄
  - callee 측에서 사용 (넘겨받은 주소를 이용해서 그 주소에 있는 변수에 access)
  - ex) **`int* first_addr;`**, … **`*first_addr = *first_addr + 1;`**
- **Indirection operator와 address operator는 반대 개념**

Pass by Reference



예제 프로그램 2 - 나눗셈

```c
#include <stdio.h>

 void Calc_divide(int in1, int in2, int* result);
 int Calc_remainder(int in1, int in2);

int main(void)
{
    int intNum1;
    int intNum2;
    int intCalc;

    printf("Enter two integral number: ");
    scanf("%d %d", &intNum1, &intNum2);

    Calc_divide(intNum1, intNum2, &intCalc);
    printf("%d / %d is %d", intNum1, intNum2, intCalc);

    intCalc = Calc_remainder(intNum1, intNum2);
    printf(" with a remainder of: %d\n", intCalc);

    return 0;
}

void Calc_divide(int in1, int in2, int* result)
{
    *result = in1 / in2;
}

int Calc_remainder(int in1, int in2)
{
    int result_remainder;
    result_remainder = in1 % in2;
    return result_remainder;
}
```

예제 소스

두 가지 방법의 함수간 데이터(결과값) 전송

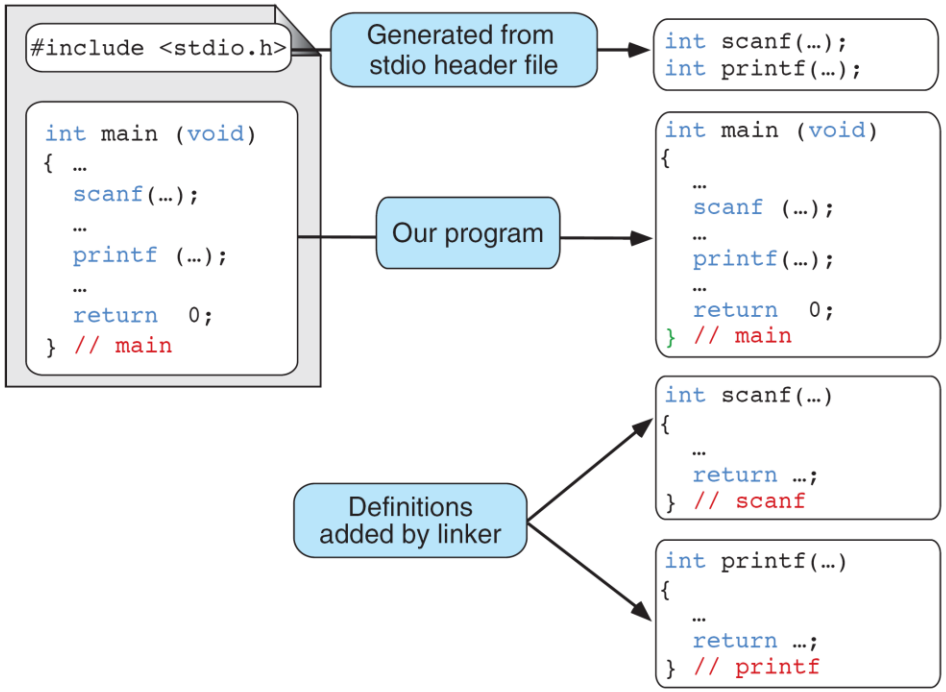
1. **Pass-by-reference로 parameter variable에 결과값을 남기는 방식**
2. **Return 값으로 결과값 전송**

```
kimjihwan@cspro:~/test_dr$ gcc -o calc.o calc.c
kimjihwan@cspro:~/test_dr$ ./calc.o
Enter two integral number: 13 2
13 / 2 is 6 with a remainder of: 1
kimjihwan@cspro:~/test_dr$
```

결과

Standard Functions

- C언어에는 프로그래머의 편의성을 도모하기 위해
  - 프로그래밍 언어 개발자들에 의해 작성되어 포함된 함수들이 존재
  - 널리 사용하는 기능들을 함수로 정의하여 모아놓은 라이브러리들이 있음
  - Standard Function들을 사용하기 위해 전처리 구문 영역에 #include문을 사용하여 해당 라이브러리를 헤더파일로 불러들임
    - 대표적인 표준 라이브러리 함수인 scanf()와 printf()함수를 사용하기 위해 <stdio.h> 헤더파일 이용
  - 대표적인 표준 라이브러리
    - 표준 입출력 함수 <stdio.h>
    - 문자열 조작 함수 <string.h>
    - 문자 관련 함수 <ctype.h>
    - 유틸리티 함수 <stdlib.h>
    - 시간 및 날짜 관련 함수 <time.h>
    - 수학 관련 함수 <math.h>



Standard Functions

- 일반적으로 널리 사용하는 기능들을 함수로 정의하여 제공하는 라이브러리
- 표준 입출력 함수 <stdio.h>
  - `printf(), scanf(), getchar(), putchar()`
- 수학 관련 함수 <math.h>
  - `double fabs(double number),`
  - `double ceil(double number),`
  - `double floor(double number),`
  - `double pow(double x, double y),`
  - `double sqrt(double number)`
- 유틸리티 함수 <stdlib.h>
  - `int abs(int number),`
  - `long labs(long number),`
  - `void srand(unsigned int seed)` // seed 제공,
  - `int rand(void)` // 0에서 RAND_MAX 값 사이의 정수가 나옴

Scope of a variable

- 프로그램 내에서 정의한 변수(또는 상수, 함수이름, function declaration)가 유효한 범위를 scope 라 함
  - 예를 들면, main 함수 안에서 선언한 변수의 scope(그 변수를 사용할 수 있는 범위)는 main 함수 내부

- 유효 범위에 따라 변수를 크게 두 가지로 구분
  - 지역(local) 변수 : main함수포함 모든 함수들의 안에서 정의
  - 전역(global) 변수[1] : main함수포함 모든 함수들의 밖에서 정의

- 변수들은 그것이 정의된 함수나 블록 안에서만 유효
  - 즉, 함수 안에서 선언된 변수(지역변수)는 그 함수 안에서만 이용 가능
  - 함수 안의 블록 안에서 선언된 변수(지역변수)는 그 블록 안에서만 이용 가능
  - 함수 바깥 부분은 아무런 블록으로도 묶여있지 않기 때문에 함수 밖에서, 즉 global 지역에서 선언된 변수(전역변수)는 프로그램 전체에서 이용 가능

↳ 여러이유중 대표적인 것이 가독성

[1] 전역변수는 절대로 사용하지 않는 것이 좋습니다!!!

Scope

- **전역 변수(global variable)**
  - 함수 외부에 정의되는 변수
  - main(void)도 함수이기 때문에 main(void)함수 밖에 정의
  - 외부 변수(external variable) 라고도 부른다

자동지역변수

```
int add(int a, int b){
    int result; ⟶ 생성
    result = add;
    return result; → 사라짐
```

- **지역 변수(local variable)**
  - 함수 내에서 정의한 변수
  - 정의한 함수 내에서만 사용 가능
  - 정적 지역 변수(static local variable) : 함수 호출한 뒤, 다시 함수 호출하는 사이에 변수 값 유지
  - 자동 지역 변수 : 지역 변수가 함수를 호출할 때마다 새롭게 만들어 지고, 함수 실행이 끝나면 변수가 없어짐.
    모든 일반적인 지역 변수들이 다 자동 지역 변수임!!

Scope

- 예제 프로그램- scope rule

```
[root@mclab chap4]# vi chap4-5.c
[root@mclab chap4]# gcc -o chap4-5 chap4-5.c
[root@mclab chap4]# ./chap4-5
1 2 12
1 4 5.000000
4 4 4
4 4 5.000000
4 2 12
[root@mclab chap4]#
```

```c
#include <stdio.h>

int a=10, b=11, c=12;

int main(void)
{
        int a=1, b=2;
        printf("%d %d %d\n", a, b, c);
        {
                int b=4;
                float c=5.0;
                printf("%d %d %f\n", a, b, c);
                a=b;
                {
                        int c;
                        c=b;
                        printf("%d %d %d\n", a, b, c);
                }
                printf("%d %d %f\n", a, b, c);
        }
        printf("%d %d %d\n", a, b, c);
        return 0;
}
```

block 1

a, b : 각각 block1 내부의 a, b를 가리킨다.

c : block1 내부에 변수 c가 없으므로 그 상위 영역인 전역변수 c를 보게 된다.

block 2

a : block2 내부에 변수 a가 없으므로 그 상위 영역인 block1의 변수 a를 보게 된다. 그러므로 block2에서의 a의 값의 변경은 block1의 변수 a의 값의 변경을 의미한다.

b, c : 각각 block2 내부의 a, b를 가리킨다.

block 3

a, b : block3 내부에 변수 a, b가 없으므로 상위영역의 변수를 가리키게 된다. (a는 block1의 a를, b는 block2의 b를 가리킴)

c : block 3 내부의 c를 가리킨다. c의 변경은 상위 영역의 c의 변화를 초래하지 않는다.

Scope

- 예제 프로그램 - scope rule (Parallel and Nested Blocks)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          int a=1, b=2;
6          {
7                  float b=3.0;
8                  printf("a=%d, b=%f\n", a, b);
9          }
10
11         {
12                 float a=4.0;
13                 printf("a=%f, b=%d\n", a, b);
14         }
15 }
```

block 1

➔ int a는 보이지만 int b는 볼 수 없다. 대신 float b가 보인다.

block 2

➔ int b는 보이지만 int a는 볼 수 없다. 대신 float a가 보인다. Block 1의 float b는 block 2에서는 볼 수 없다. (block 1과 block 2가 parallel한 관계에 있기 때문)

```
[root@mclab chap4]# vi chap4-6.c
[root@mclab chap4]# gcc -o chap4-6 chap4-6.c
[root@mclab chap4]# ./chap4-6
a=1, b=3.000000
a=4.000000, b=2
[root@mclab chap4]#
```

Scope – 전역 변수, 지역 변수

예제 소스

```
#include <stdio.h>

void add(int first, int second);

int result;

int main(void)
{
    int first;
    int second;

    first = 5;
    second = 10;

    add(first, second);
    printf("result = %d\n", result);

    return 0;
}

void add(int first, int second)
{
    result = first + second;
}
```

전역변수
(global variable)

지역변수
(local variable)

```
kimjihwan@cspro:~/test_dr$ gcc -o pass.o pass.c
kimjihwan@cspro:~/test_dr$ ./pass.o
result = 15
kimjihwan@cspro:~/test_dr$
```

결과

Scope 추가 보조 자료

```
/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include  <stdio.h>
int fun (int a, int b);
```

Global area

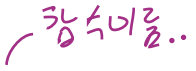
```
 int main (void)
 {
   int   a;
   int   b;
   float y;
   …
      {  // Beginning of nested block
        float a = y / 2;
        float y;
        float z;
        …
        z = a * b;
        …
      }  // End of nested block
   …
 }  // End of main
```

main's area

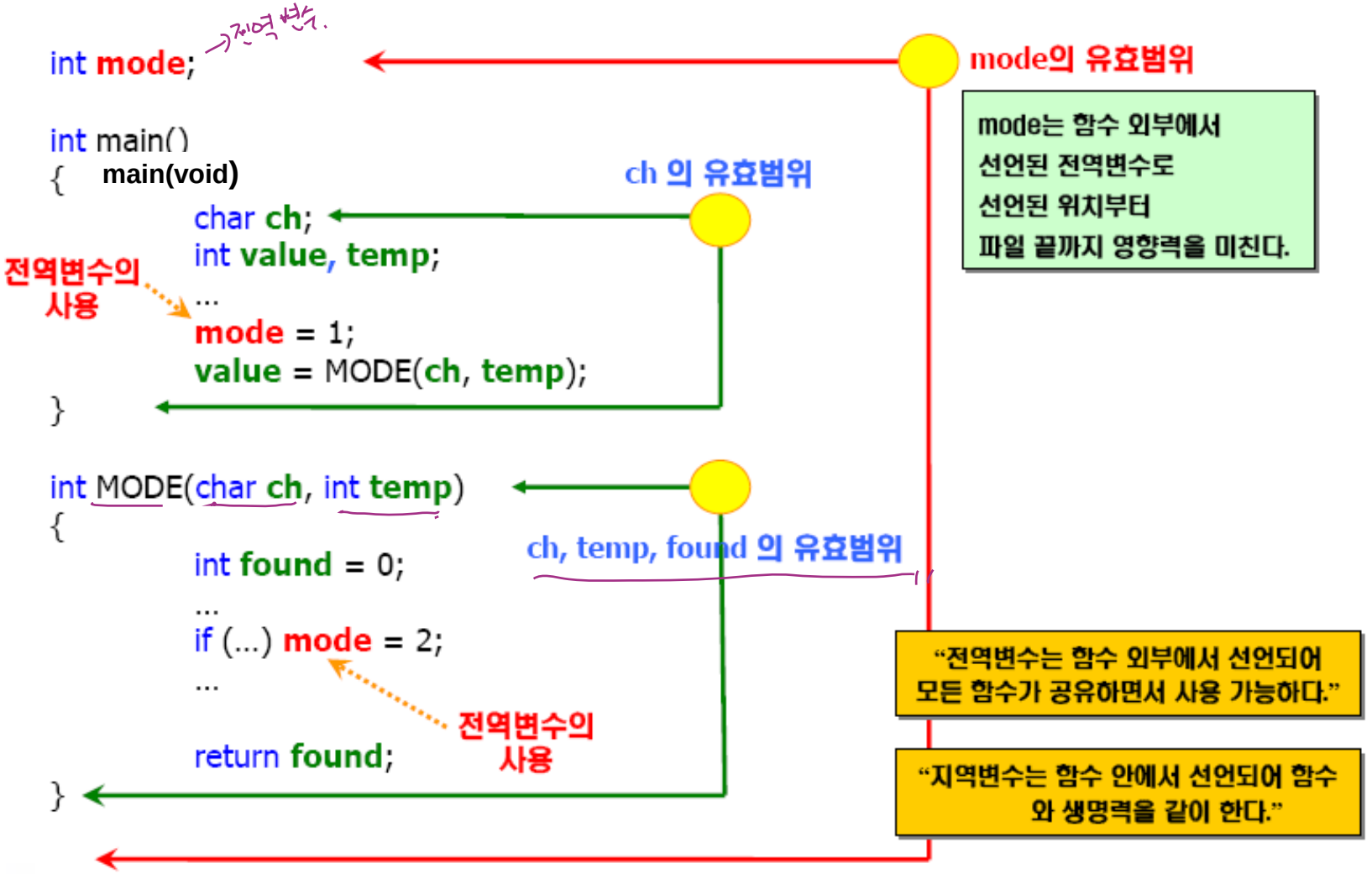
Nested block area

```
int fun (int i, int j)
{
   int a;
   int y;
   …
}  // fun
```

fun's area



Scope 추가 보조 자료



추가 예제 프로그램 3 - 나눗셈

```c
#include <stdio.h>

 void Calc_divide(int in1, int in2);
 int Calc_remainder(int in1, int in2);

int result;

int main(void)
{
    int intNum1;
    int intNum2;
    int intCalc;

    printf("Enter two integral number: ");
    scanf("%d %d", &intNum1, &intNum2);

    Calc_divide(intNum1, intNum2);
    intCalc = result;
    printf("%d / %d is %d", intNum1, intNum2, intCalc);

    intCalc = Calc_remainder(intNum1, intNum2);
    printf(" with a remainder of: %d\n", intCalc);

    return 0;
}

void Calc_divide(int in1, int in2);
{
    result = in1 / in2;
}

int Calc_remainder(int in1, int in2)
{
    int result_remainder;
    result_remainder = in1 % in2;
    return result_remainder;
}
```

예제 소스

전역변수(**global variable**), 지역변수(**local variable**)의 위치를 확인하고 변수들의 유효범위(**scope**)를 확인한다.

```
kimjihwan@cspro:~/test_dr$ gcc -o calc.o calc.c
kimjihwan@cspro:~/test_dr$ ./calc.o
Enter two integral number: 13 2
13 / 2 is 6 with a remainder of: 1
kimjihwan@cspro:~/test_dr$
```

결과