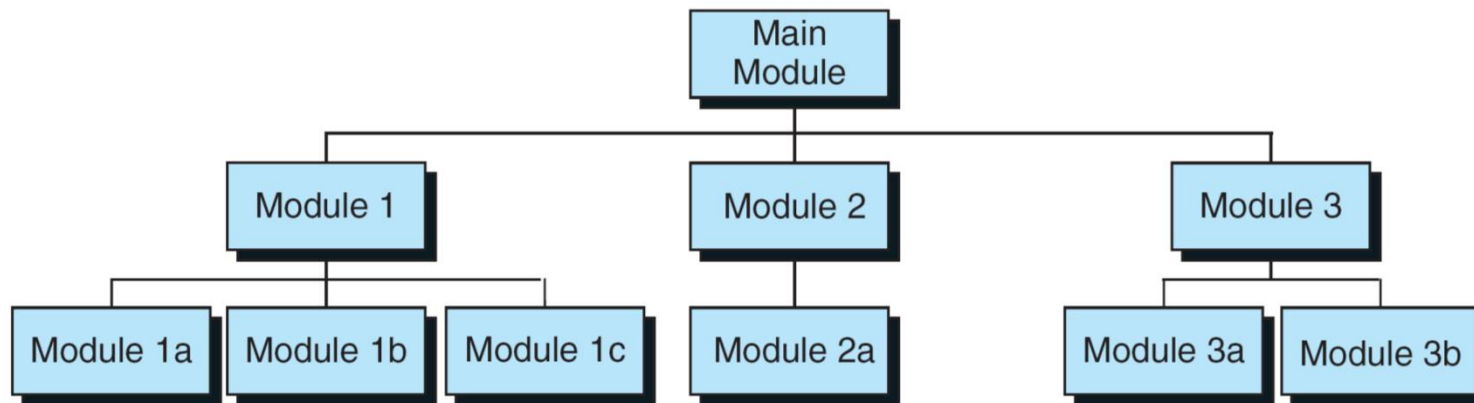

C 언어

8. Functions

Designing Structured Programs

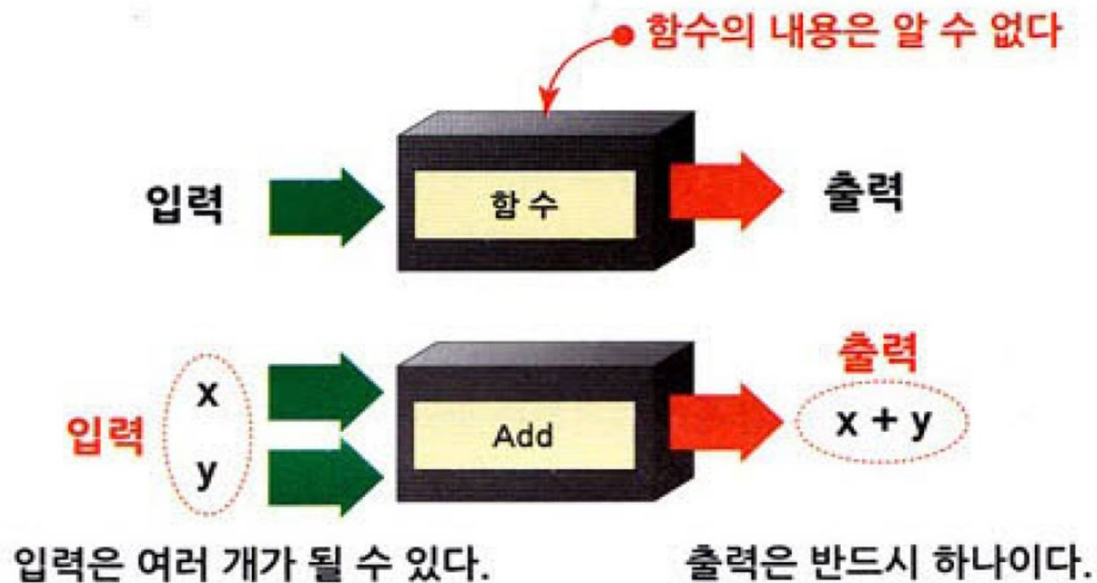
◆ Top-down Structured Programming

- 어려운 문제를 더 단순하고 쉬운 작은 문제로 나누어서 단계적으로 구체화하는 방법
- 각 단순화한 문제를 계속하여 단순한 프로그램 모듈로 분리하여 구현함으로써 단계적으로 프로그램을 하게 함



Functions in C

- ◆ C언어에서
함수는 특정 작업을 수행하기 위하여 불려지는 블랙박스
- ◆ C언어는 함수를 통해 top-down design의 방식 취함
 - C 프로그램은 하나 또는 하나 이상의 함수를 통해 만들어지며,
 - Main 함수는 하나만 존재할 수 있다



함수의 정의와 선언

◆ main 함수 다시 보기 : 함수의 기본 형태



왜 함수를 쓰는가?

```
int main(void)
{
    int num, i, sum;
    scanf("%d", &num);

    sum = 0;
    for ( i = 1 ; i <= num ; i++ )
        sum += i;

    printf("합계 : %d\n", sum);
    return 0;
}
```

왜 함수를 쓰는가?

```
int main(void)
{
```

```
    int num, i, sum;
    int result, value;
    scanf("%d", &num);
```

```
    sum = 0;
    for ( i = 1 ; i <= num ; i++ )
        sum += i;
```

```
    printf("합계 : %d\n", sum);
```

```
    ...
```

```
    sum = 0;
    for ( i = 1 ; i <= 100 ; i++ )
        sum += i;
```

```
    result = sum / 100;
```

```
    ...
```

```
    sum = 0;
    for ( i = 1 ; i <= value ; i++ )
        sum += i;
```

```
    printf("%d까지의 합계 : %d\n", value, sum);
```

```
    return 0;
```

```
}
```

num까지의 합계를
구하는 코드

100까지의 합계를
구하는 코드

value까지의 합계를
구하는 코드

왜 함수를 쓰는가?

```
int GetSum(int num)
{
    int i, sum;
    for ( i = 1, sum = 0 ; i <= num ; i++ )
        sum += i;
    return sum;
}
```

→ 합계를 구하는 코드는
한 번만 작성

```
int main(void)
{
    int num, i, sum;
    int result, value;
    scanf("%d", &num);
    sum = GetSum(num);
    printf("합계 : %d\n", sum);

    ...

    result = GetSum(100) ; 100;

    ...

    sum = GetSum(value);
    printf("%d까지의 합계 : %d\n", value, sum);
    return 0;
}
```

→ num까지의 합계를
구하는 함수 호출

→ 100까지의 합계를
구하는 함수 호출

→ value까지의 합계를
구하는 함수 호출

재귀 함수

◆ 재귀 함수의 기본적 이해

- 자기 자신을 다시 호출하는 형태의 함수

```
/* recursive_basic.c */
#include <stdio.h>

void Recursive(void)
{
    printf("Recursive Call! \n");
    // Recursive();
}

int main(void)
{
    Recursive();
    return 0;
}
```

Functions in C

◆ 예제 프로그램 - 함수를 사용한 프로그램의 예

```
1 #include <stdio.h>
2
3 int sum(int a, int b);
4
5 int main(void)
6 {
7     int x, y, s;
8
9     x=1;
10    y=2;
11
12    s=sum(x, y);
13    printf("%d + %d = %d\n", x, y, s);
14
15    x=3;
16    y=5;
17    s=sum(x, y);
18    printf("%d + %d = %d\n", x, y, s);
19 }
20
21 int sum(int a, int b)
22 {
23     return a+b;
24 }
```

- 함수 sum(a,b) 은 user-defined function 임
- 함수 sum(a, b)은 두개의 인자를 받아서 그것을 더한 값 반환
- main에서 함수 sum()을 두 번 호출하고 있으며 정확하게 그 값을 반환

```
1 + 2 = 3
3 + 5 = 8
```

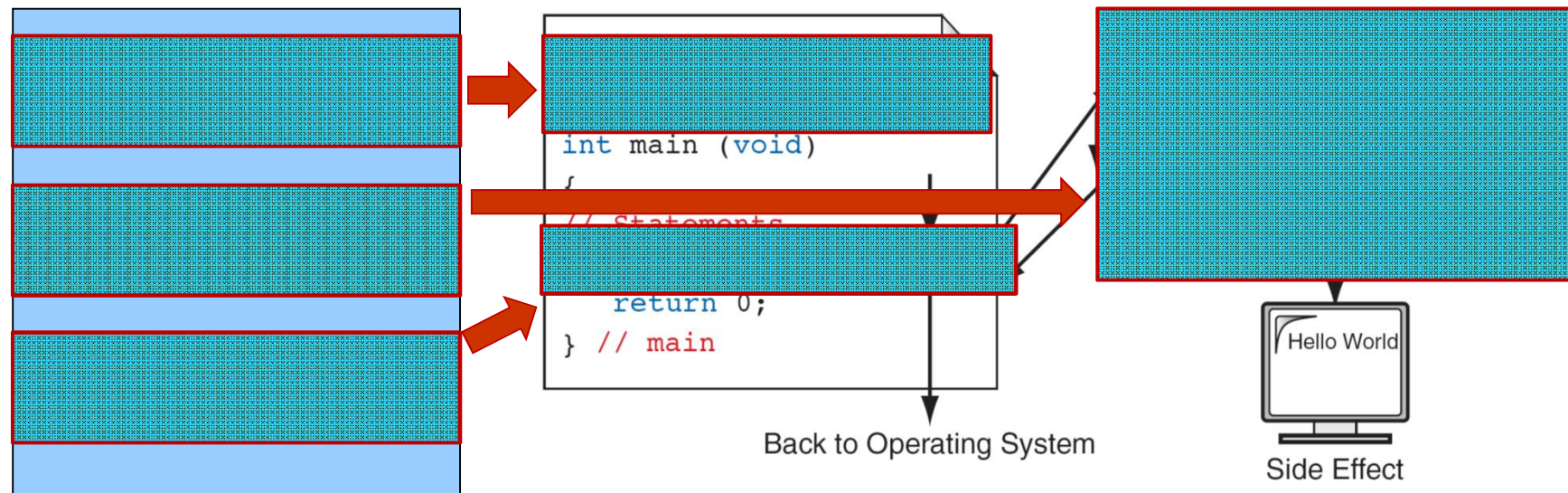
Function

- ◆ C언어에서의 함수사용은 다음과 같은 장점 제공
 - 문제를 분리하여 **단순화** 시킴
 - 함수는 한곳 이상에서 필요로 하는 코드를 **재사용** 할 수 있는 방법
 - 제공되는 **library**를 통해 다양한 함수를 제공받아 개발에 **편의성** 지님
 - 함수는 데이터를 보호
 - ◆ 함수 안에서 선언된 지역변수는 그 함수 안에서만 사용 가능하며, 그 함수가 실행될 때만 이용되어짐
- ◆ 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 제공하는 **표준 함수(Standard functions)**이다.

User-defined Functions

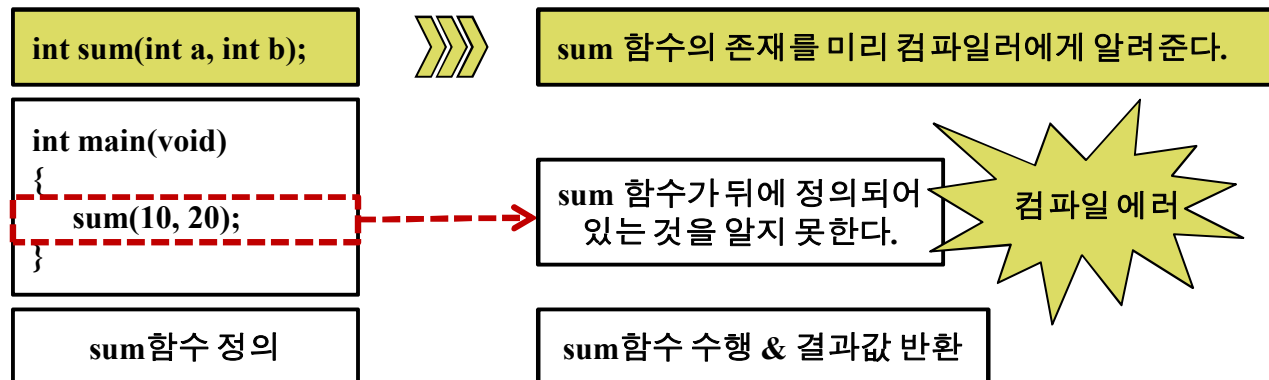
- ◆ 사용자 정의 함수는 사용자에게 의해 원하는 의도에 따라 내용을 구성하여 호출하여 사용할 수 있음
- ◆ C언어에서 함수를 사용하려면 다음과 같은 요소 필요



Function Declaration

◆ Function Declaration

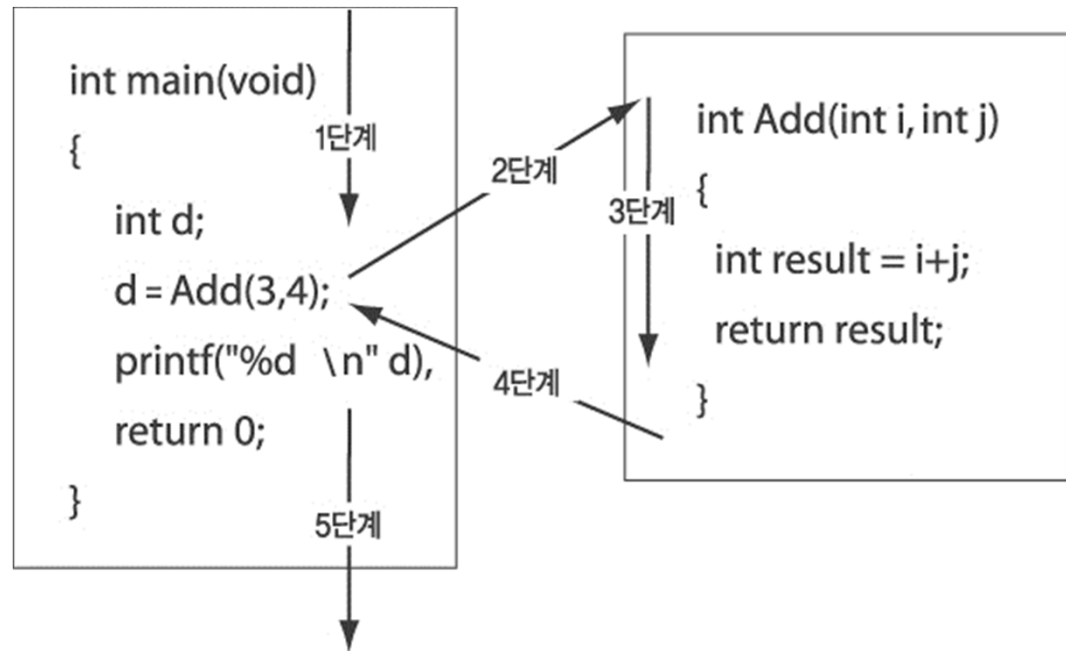
- 함수의 헤더부분을 프로그램 앞부분에 적어주는 것
- 컴파일러에게 함수의 이름과, 함수가 갖는 **parameter**의 개수와 데이터형, 그리고 함수에 의해 **return**되는 결과값의 데이터형을 알려줌
 - ◆ **return-type function-name(formal-parameter list)**
- **formal** 과 **actual** 전달인자는 타입, 순서, 갯수는 일치해야 하지만 이름은 일치하지 않아도 됨



함수 호출 과정의 이해

```
#include <stdio.h>
```

```
int main(void)
{
    int d;
    d = Add(3, 4);
    printf("%d \n", d);
    return 0;
}
```



Function Declaration

함수 선언

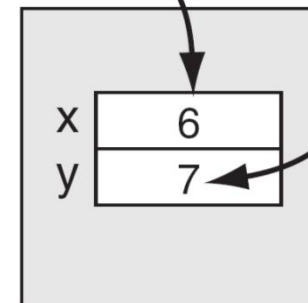
```
// Function Declaration  
int multiply (int multiplier, int multiplicand );
```

함수 호출

```
int main (void)  
{  
    int product;  
    product = multiply (6, 7);  
    ...  
    return 0;  
} // main
```

함수 정의

```
int multiply (int x, int y)  
{  
    return x * y;  
} // multiply
```



Function Definition

실습예제 1

- ◆ 두 개의 정수를 입력 받고, 두 수의 덧셈, 뺄셈, 곱셈, 나눗셈 결과를 출력하는 프로그램을 작성하시오.
- ◆ 단, 각각의 연산은 하나의 독립된 함수로 구현하며, 나눗셈의 결과는 실수가 되도록 한다.

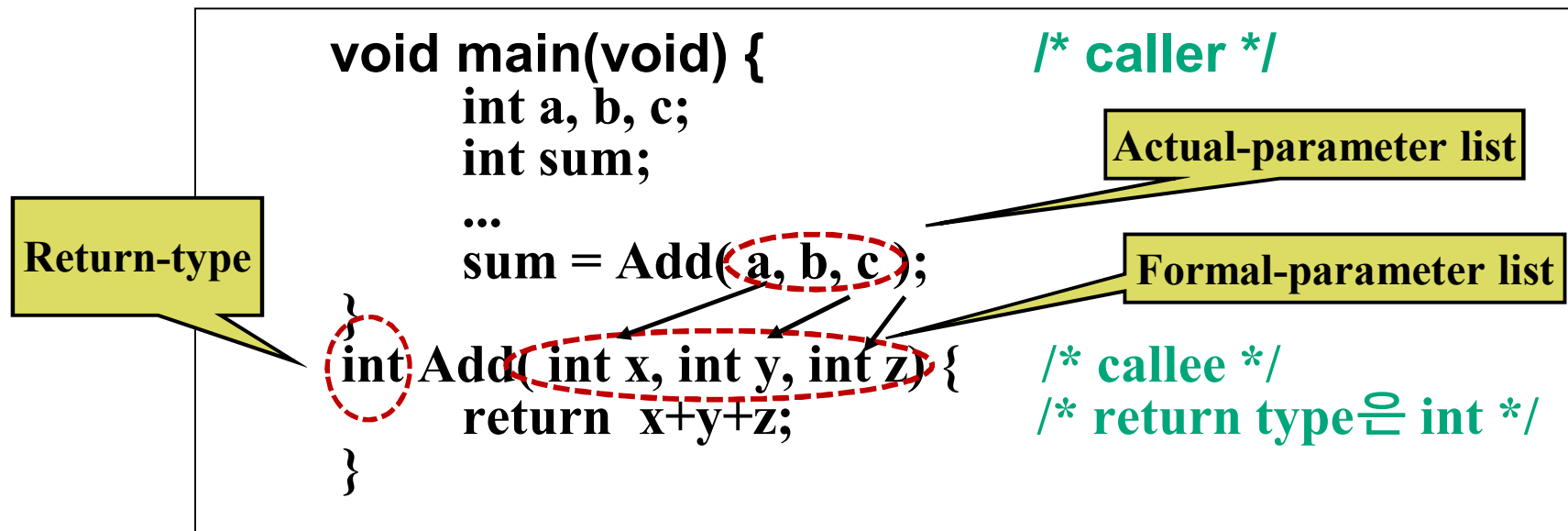
■ 실행결과예시

```
First num : 2  
Second num : 4  
Add : 6  
Sub : -2  
Mul : 8  
Div : 0.500000
```

Function Definition

- **Formal-parameter list**

- ◆ caller 가 함수를 호출할 때 그 함수(callee)에서 사용할 데이터 함께 전달 가능
- ◆ Parameter list는 comma-separated list로 각 argument는 순서대로 해당 parameter로 전달, 즉 대입됨
- ◆ Ex)



Function Definition

◆ Function Definition

- 함수는 일반적으로 다음과 같이 정의

Function Header

```
return_type function_name (formal parameter list)
```

```
{  
  // Local Declarations  
  ...  
  // Statements  
  ...  
} // function_name
```

Function Body

● 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

Function Definition

- **Declarations and statements : function body(block)**
 - ◆ 함수의 내용에 해당
 - ◆ 블록 안에서 변수의 선언이 가능 (can be nested)
 - ◆ 함수는 다른 함수 안에서 정의될 수 없음
- **Returning control**
 - ◆ 리턴 값이 없을 경우
 - `return;`
 - ◆ 리턴 값이 있을 경우
 - `return expression;`

Function return type should be explicitly defined

```
int first (...)  
{  
    ...  
    return (x + 2);  
} // first
```

A return statement should be used even if nothing is returned

```
void second (...)  
{  
    ...  
    return;  
} // second
```

Function Call

◆ Function Call

- 함수호출에서 형식 매개변수 (formal parameters) 와 실 매개변수 (actual parameters) 의 개수와 타입 및 순서는 반드시 일치해야 함
- 실제 매개변수와 형식 매개변수의 이름은 같을 필요가 없으며, 순서에 따라서로 연결
- Parameter passing
 - ◆ C 언어에서는 함수를 호출할 때 argument 전달하는 방법 두 가지
 - 값에 의한 전달 (Call by Value)
 - 참조에 의한 전달 (Call by Reference)

multiply (6, 7)
multiply (6, b)
multiply (multiply (a, b), 7)

multiply (a, 7)
multiply (a + 6, 7)
multiply (... , ...)

expression

expression

Call by value

◆ 예제 프로그램 - Call by value

```
1 #include <stdio.h>
2
3 void swap(int x, int y);
4
5 int main(void)
6 {
7     int a=5;
8     int b=3;
9
10    swap(a, b);
11    printf("a=%d, b=%d\n", a, b);
12
13    return 0;
14 }
15
16 void swap(int x, int y)
17 {
18     int temp;
19     temp = x;
20     x = y;
21     y = temp;
22 }
```

swap() 함수 내에서 아무리 x와 y를 교환해도 받은 복사본을 교환할 뿐이지 원래의 a, b를 교환하는 것이 아니다.

```
[root@mclab chap4]# vi chap4-2.c
[root@mclab chap4]# gcc -o chap4-2 chap4-2.c
[root@mclab chap4]# ./chap4-2
a=5, b=3
[root@mclab chap4]#
```

호출하는 쪽의 인자와 호출되는 쪽의 인자는 서로 다른 메모리에 저장되어 있어 아무 관계가 없다.

Call by value

◆ Pass by value (Call by value)

- 값에 의한 전달이란 뜻은 Caller에서 함수를 호출할 때 전달하는 argument로 값(value)를 사용한다는 것
- 형식인자와 실인자가 따로 메모리를 가짐
- 실인자의 값을 형식인자에 복사하여 사용
- 실인자의 값은 바뀌지 않음

```
int main (void)
{
    int a;
    ...
    downFun (a, 15);
    ...
} // main
```

```
void downFun (int x, int y)
{
    ...
    return;
} // downFun
```

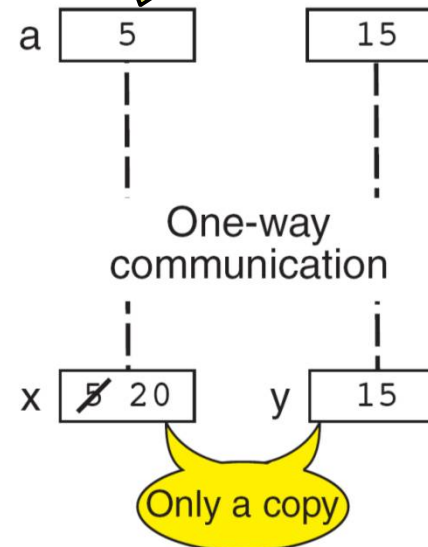
Call by value

```
// Function Declaration
void downFun (int x, int y);
int main (void)
{
    // Local Definitions
    int a = 5;
    // Statements
    downFun (a, 15);
    printf("%d\n", a);
    return 0;
} // main
```

prints 5

```
void downFun (int x, int y)
{
    // Statements
    x = x + y;
    return;
} // downFun
```

변수 a(실인자)의 값은
변하지 않고 복사만 된다.



Call by Reference

◆ 예제 프로그램 - Pass by reference

```
1 #include <stdio.h>
2
3 void swap(int *x, int *y);
4
5 int main(void)
6 {
7     int a = 5;
8     int b = 3;
9
10    swap(&a, &b);
11    printf("a=%d, b=%d\n", a, b);
12
13    return 0;
14 }
15
16 void swap(int *x, int *y)
17 {
18     int temp;
19     temp = *x;
20     *x = *y;
21     *y = temp;
22 }
```

- main은 swap()을 호출하면서 변수 a와 b의 주소 넘겨줌
- swap()은 이를 포인터(주소를 저장하는 변수) x, y에 받음
- temp에 주소 x가 가리키는 곳(a)의 값을 넣음
- x가 가리키는 곳(a)에 y가 가리키는 곳의 값(b)를 넣음
- y가 가리키는 곳(b)에 temp의 값(a)을 넣음
- 실행결과 main함수에서 두 변수의 값이 바뀌는 것을 확인할 수 있음

```
[root@mclab chap4]# vi chap4-3.c
[root@mclab chap4]# gcc -o chap4-3 chap4-3.c
[root@mclab chap4]# ./chap4-3
a=3, b=5
[root@mclab chap4]#
```

Call by Reference

- ◆ **Pass by Reference (Call by Reference)**
 - 참조에 의한 전달은 호출되는 함수에 **argument**의 값을 전달하는 것이 아니라, **argument**의 메모리 주소(**address**)값을 전달하는 것
 - 메모리 주소 값을 전달하면 호출되는 함수는 그 주소값을 이용하여 실인자의 값을 바꿀 수 있음
- ◆ **Address operator (&) : “Give me the address of this data”**
- ◆ 변수의 주소를 얻어냄
- ◆ **caller** 측에서 사용 (주소를 구해서 **callee**에게 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 변수에 **access**)
- ◆ **Indirection operator**와 **address operator**는 반대 개념

Standard Functions

◆ library 함수의 사용 예제

- **rand()** : 임의의 숫자가 생성되며 %(modulus operator)를 통하여 범위를 제한할 수 있음
 - ◆ **rand() % 51** → 0부터 50까지의 임의의 숫자 생성
 - ◆ **rand() % ((max+1) - min) + min** 를 통해서 min부터 max의 값 생성할 수 있음

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(void)
6 {
7     int rand1, rand2, rand3;
8
9     printf("rand() example...\n");
10
11     srand(time(NULL));
12     rand1 = rand();
13     rand2 = rand();
14     rand3 = rand();
15
16     printf("Result : rand1 = %d, rand2 = %d, rand3 = %d\n", rand1, rand2, rand3);
17
18     return 0;
19 }
20
```

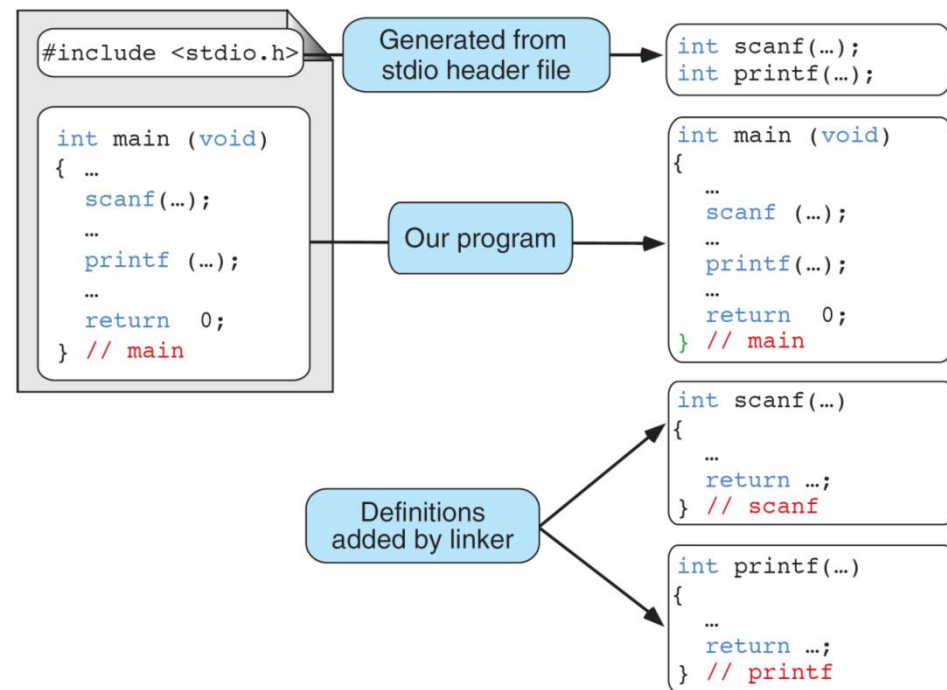
```
[root@mclab chap4]# vi chap4-4.c
[root@mclab chap4]# gcc -o chap4-4 chap4-4.c
[root@mclab chap4]# ./chap4-4
rand() example....
Result : rand1 = 2046416250, rand2 = 2067920331, rand3 = 178984914
[root@mclab chap4]#
```

Standard Functions

- ◆ 일반적으로 널리 사용하는 기능들을 함수로 정의하여 제공하는 라이브러리
- ◆ **Function declaration** 을 가지고 있음
- ◆ 다양한 라이브러리를 제공함
- ◆ 사용자도 라이브러리를 만들어 사용할 수 있음
- ◆ **<stdio.h>**
 - **printf(), scanf()**
- ◆ **<math.h>**
 - **double fabs (double number), double ceil (double number), double floor (double number), double pow (double x, double y), double sqrt (double number)**
- ◆ **<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**

Standard Functions

- ◆ C언어와 C++언어에는 프로그래머의 편의성을 도모하기 위해
 - 프로그램 언어 개발자들에 의해 작성되어 포함된 함수들이 존재
 - 널리 사용하는 기능들을 함수로 정의하여 제공하는 라이브러리
 - **Standard Functions**를 불러 들이기 위해 전처리 구문 영역에 **#include**문을 사용
 - ◆ 대표적인 표준 라이브러리 함수인 `scanf()`와 `printf()` 함수를 사용하기 위해 `<stdio.h>` 헤더파일 이용
 - 대표적인 종류
 - ◆ 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>`



Scope

◆ 예제 프로그램- scope rule

```
1 #include <stdio.h>
2
3 int a=10, b=11, c=12;
4
5 int main(void)
6 {
7     int a=1, b=2;
8     printf("%d %d %d\n", a, b, c);
9     {
10         int b=4;
11         float c=5.0;
12         printf("%d %d %f\n", a, b, c);
13         a=b;
14         {
15             int c;
16             c=b;
17             printf("%d %d %d\n", a, b, c);
18         }
19         printf("%d %d %f\n", a, b, c);
20     }
21     printf("%d %d %d\n", a, b, c);
22     return 0;
23 }
```

```
[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]#
```

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 b, c를 가리킨다.

block 3

a, b : block3 내부에 변수 a, b가 없으므로 상위 영역의 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         Block1    float b=3.0;
8                   printf("a=%d, b=%f\n", a, b);
9     }
10
11     {
12         Block2    float a=4.0;
13                   printf("a=%f, b=%d\n", a, b);
14     }
15 }
```

```
[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]#
```

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)

Scope(변수의 범위)

◆ 변수의 특성에 따른 분류

- 지역 변수(Local Variable)

- 중 괄호 내에 선언되는 변수

- 전역 변수(Global Variable)

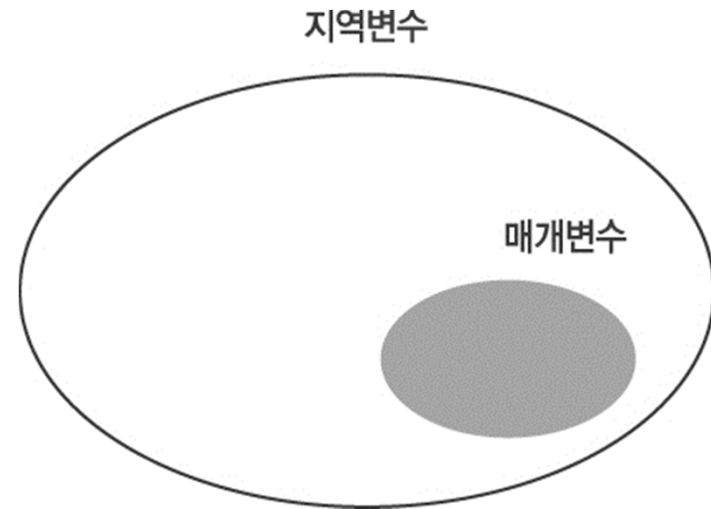
- 함수 내에 선언되지 않는 변수

- 정적 변수(Static Variable)

- 함수 내부, 외부 모두 선언 가능

- 레지스터 변수(Register Variable)

- 선언에 제한이 많이 따름



Scope

- ◆ 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)은 **main** 함수 내부
- ◆ 유효 범위에 따라 크게 두 가지로 구분
 - 지역(**local**) 변수 : **main** 함수나 기타 함수들의 안에서 정의
 - 전역(**global**) 변수 : **main** 함수나 기타 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

Scope(변수의 범위)

```
/* 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;
```

main's area

```
    int    b;
```

```
    float  y;
```

```
    ...
```

```
    { // Beginning of nested block
```

```
        float a = y / 2;
```

```
        float y;
```

```
        float z;
```

Nested block
area

```
        ...
```

```
        z = a * b;
```

```
        ...
```

```
    } // End of nested block
```

```
    ...
```

```
} // End of main
```

```
int fun (int i, int j)
```

```
{
```

```
    int a;
```

```
    int y;
```

```
    ...
```

```
} // fun
```

fun's area

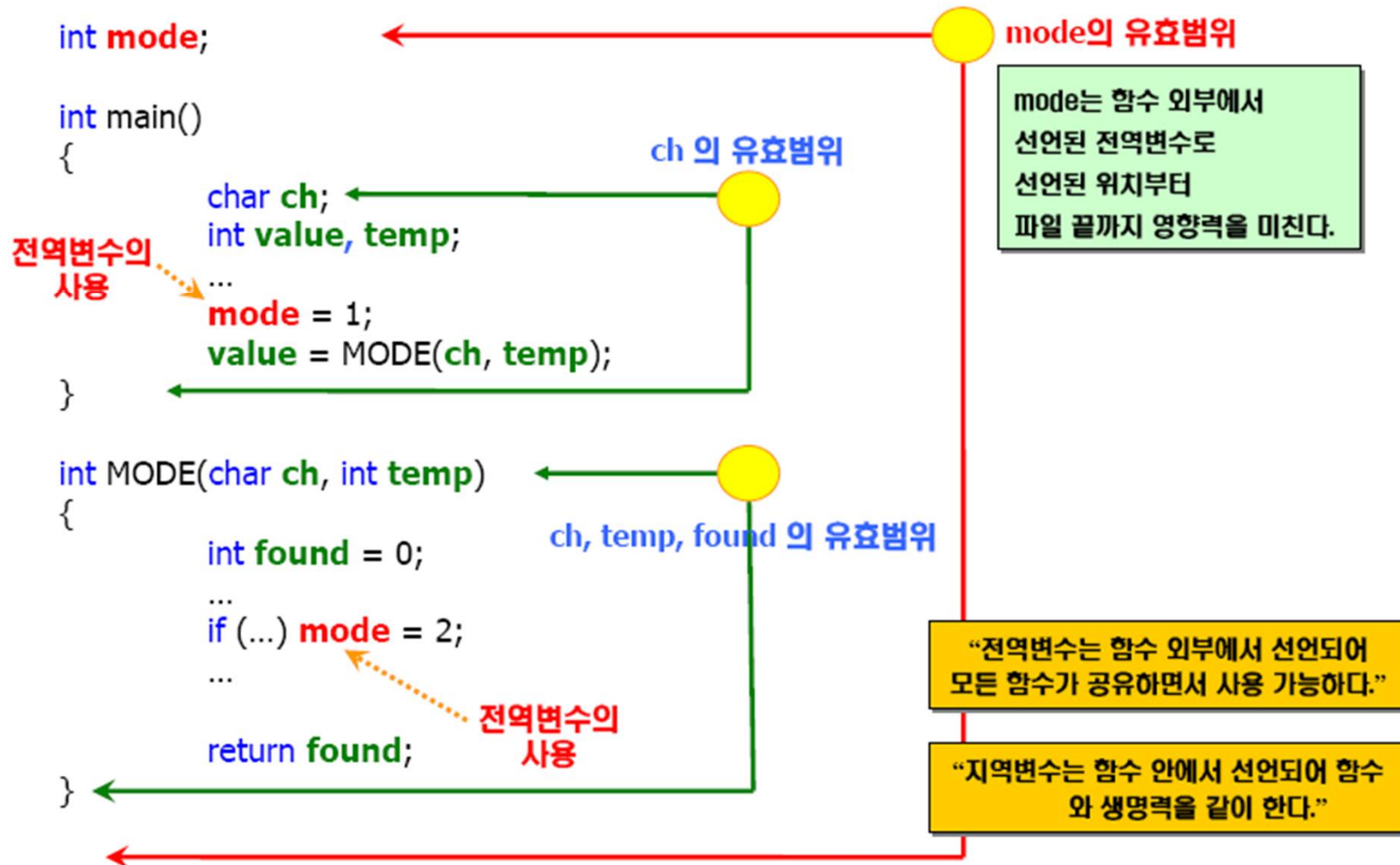
◆ 지역 변수(local variable)

- 함수 또는 어떠한 블록 안에 정의된 변수
- 사용 범위가 함수 내부로 제한
 - ◆ 함수가 호출되면 생성되었다가 리턴하면 소멸
 - ◆ 서로 다른 함수에 같은 변수명 사용 가능
- 지역 변수의 초기화
 - ◆ 함수가 호출될 때마다 초기화되고 리턴될 때 공간 반납

◆ 전역 변수(global variable)

- 함수 외부에서 선언
- 모든 함수가 함께 사용하는 공유 변수의 특징
 - ◆ 프로그램 시작 시 생성되고 종료될 때까지 공간 할당

Scope(변수의 범위)



Recursion

예제 프로그램 - recursion

```
1 #include <stdio.h>
2
3 int factorial(int n);
4
5 int main(void) {
6     int a;
7     printf("Input a number : ");
8     scanf("%d", &a);
9
10    printf("%d! = %d\n", a, factorial(a));
11 }
12
13 int factorial(int n) {
14     if(n==0)
15         return 1;
16     else
17         return n * factorial(n-1);
18 }
```

함수 factorial() 안에서 factorial()를 호출하는 프로그램

- 1) 호출할 때 인자는 매번 1씩 줄어 들기 때문에 언젠가는 factorial(0)을 호출하게 된다.
- 2) factorial(0)은 1을 반환한다.
- 3) 호출된 순서를 거꾸로 올라가며 factorial값을 구한다.

```
Input a number : 5
5! = 120
```

Recursion

- ◆ 함수가 자기 자신을 다시 호출하는 형태
- ◆ 특정문제의 정의로부터 바로 프로그램을 할 수 있다.

◆ Factorial problem

< iterative function 정의 >

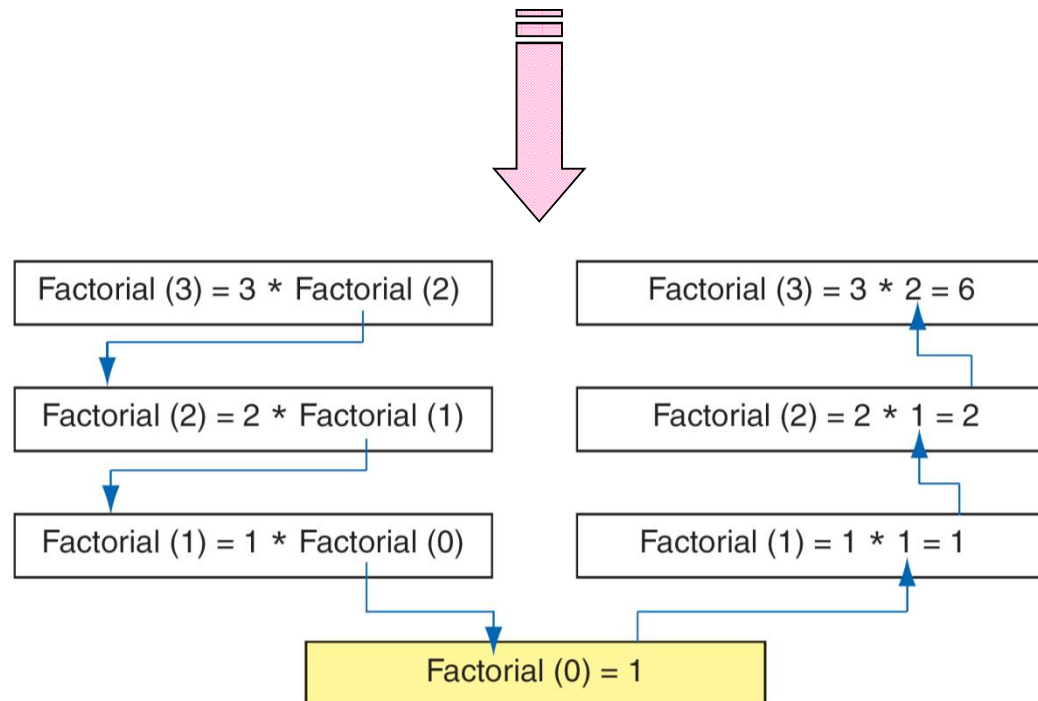
$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) \dots 3 * 2 * 1 & \text{if } n > 0 \end{cases} \quad \Rightarrow \quad \boxed{\text{factorial}(4) = 4 * 3 * 2 * 1 = 24}$$

< recursive function 정의 >

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases} \quad \Rightarrow \quad \boxed{\begin{array}{l} \text{factorial}(4) = 4 * \text{factorial}(3) \\ \text{factorial}(3) = 3 * \text{factorial}(2) \\ \text{factorial}(2) = 2 * \text{factorial}(1) \\ \dots\dots\dots \end{array}}$$

Recursion

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$



< factorial(3) recursively >

Recursion

예제 프로그램 - iterative

```
1 #include <stdio.h>
2
3 int factorial(int n);
4
5 int main(void) {
6     int a;
7     printf("Input the number : ");
8     scanf("%d", &a);
9
10    printf("%d! = %d\n", a, factorial(a));
11    return 0;
12 }
13
14 int factorial(int n) {
15     int factN = 1;
16     int i;
17     for(i = 1; i <= n; i++) {
18         factN = factN * i;
19     }
20     return factN;
21 }
22
```

recursive function은
loop문을 이용하여
iterative로 도 구현할 수도 있다.

```
gr120100205@cspro:~$ gcc -o 6-7 6-7.c
gr120100205@cspro:~$ ./6-7
Input the number : 5
5! = 120
gr120100205@cspro:~$
```

Recursion

◆ Designing Recursive Function

- 모든 recursive call은 problem의 size를 줄이거나, problem의 부분을 풀어야 한다.

- Recursive function을 design 하는 순서

1. base case 를 결정한다.

2. **general case** 를 결정한다.

3. **base case와** general case 를 function에 모두 적용시킨다.

모든 recursive function은 base case를

지닌다. Factorial 의 경우

Base case는 factorial(0)이다.

factorial 의 경우,

general case는 $n * \text{factorial}(n-1)$ 이다.

◆ Limitation of Recursion.

- 실제 속도가 iterative에 비해 떨어지므로 개념적으로만 사용
예외로 일부 특수한 문제는 recursion을 사용할 수밖에 없음
- 특수한 문제가 아니면 반복적 정의(while, for, do..while)로 처리 가능

실습예제2

- ◆ 원화(KRW)를 입력 받아 달러화(USD), 유로화(EUR), 엔화(JPY)로 환전했을 때의 금액이 얼마인지 출력하시오.
- ◆ 각 화폐 단위마다 별도의 함수가 있어서, 금액을 변환하고 출력하는 기능을 가져야 한다.
- ◆ JPY는 소수점 밑은 버리고 정수로, USD와EUR은 실수로 소수점 둘째 자리까지 출력하시오.
- ◆ 환율은USD 960.24, EUR 1269.89, JPY 817.63을 적용한다.

■ 실행결과예시

```
KRW : 23450
USD : 24.42
EUR : 18.47
JPY : 28.68
```

실습예제3

- ◆ 다섯 사람의 키를 입력 받고 평균키가 얼마인지 출력하는 프로그램을 작성하시오.
- ◆ 함수avg()는5개의 정수를 인수로 받아1개의실수(평균값)를return한다.
- ◆ 이때 키는 정수로 입력 받고, 평균은 실수가 되도록 한다.

■ 실행결과예시

```
input : 173 158 186 162 177  
result : 171.200000
```


실습예제4

- ◆ 세자리 정수를 세 번 입력 받아, 자릿수들의 합계를 출력하는 프로그램을 작성하시오
- ◆ 함수sumDigits()는 하나의 정수를 인수로 받아 자릿수들의 합계를 출력한다. return값은 없다.

■ 실행결과예시

```
input : 173  
result : 11  
input : 174  
result : 12  
input : 799  
result : 25
```



재귀 함수

```
/* static_val.c */
#include <stdio.h>

void fct(void)
{
    int val=0;          // static int val=0;
    val++;
    printf("%d ",val);
}

int main(void)
{
    int i;
    for(i=0; i<5; i++)
        fct();

    return 0;
}
```

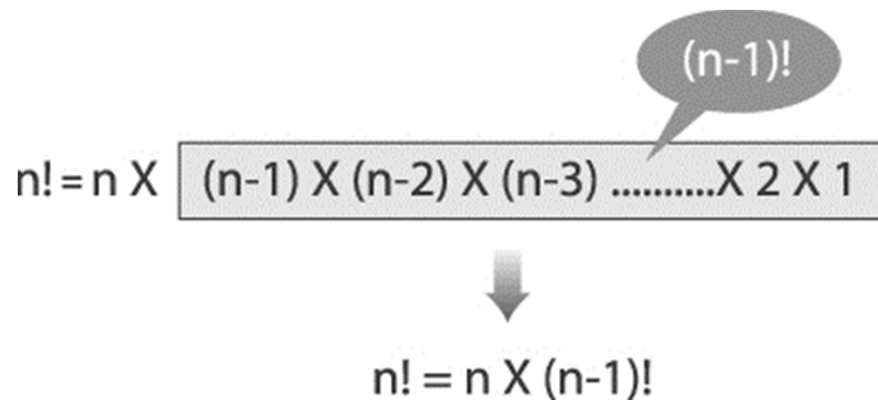
재귀 함수

◆ 재귀 함수 Design 사례

- 팩토리얼(factorial) 계산을 위한 알고리즘

$$n! = n \times \boxed{(n-1) \times (n-2) \times (n-3) \dots \times 2 \times 1}$$

↓

$$n! = n \times (n-1)!$$


$$f(n) = \begin{cases} n \times f(n-1) & n \text{이 } 1 \text{ 이상인 경우} \\ 1 & n \text{이 } 0 \text{인 경우} \end{cases}$$

재귀 함수

```
/* static_val.c */
#include <stdio.h>

void fct(void)
{
    int val=0;          // static int val=0;
    val++;
    printf("%d ",val);
}

int main(void)
{
    int i;
    for(i=0; i<5; i++)
        fct();

    return 0;
}
```

재귀 함수

```
#include <stdio.h>
```

```
void Recursive(int n)
{
    printf("Recursive Call! \n");
    if(n==1)
        return;
    Recursive(n-1);
}
```

```
int main(void)
{
    int a=2;
    Recursive(a);
    return 0;
}
```
