KU KONKUK UNIVERSITY

C 프로그래밍

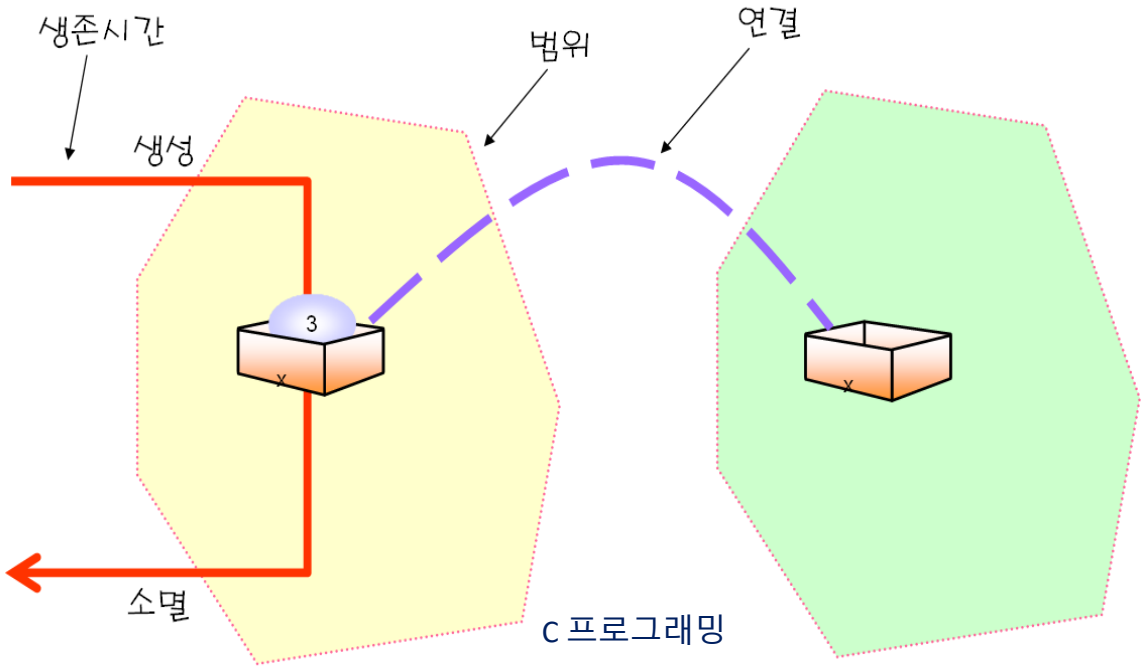
함수 II

목차

- 변수의 속성
- 전역, 지역 변수
- 재귀 호출

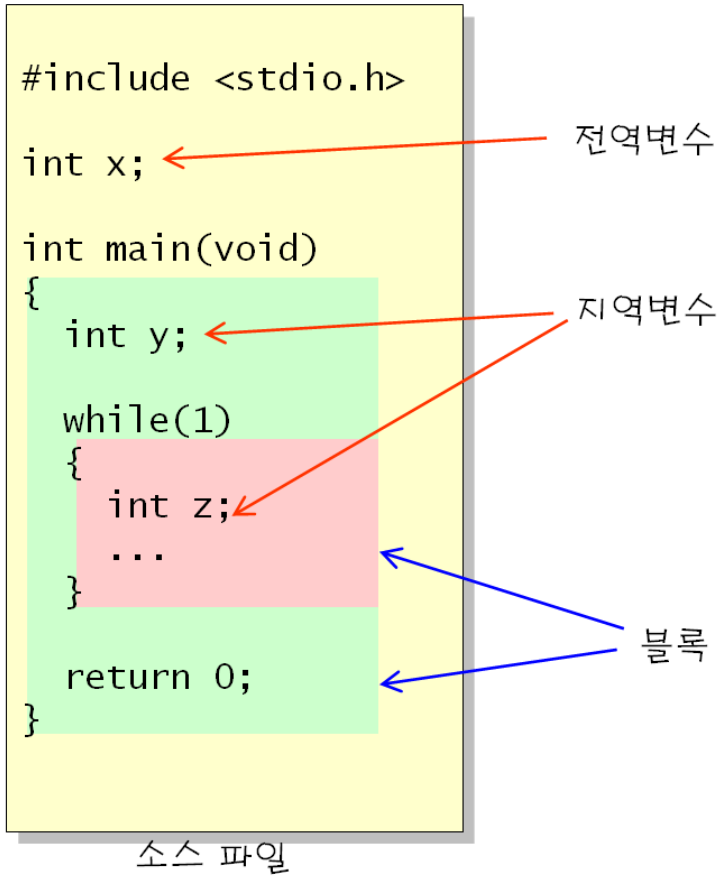
변수의 속성

- 변수의 속성: 이름, 타입, 크기, 값 + 범위, 생존 시간, 연결
  - 범위(scope) : 변수가 사용 가능한 범위, 가시성
  - 생존 시간(lifetime): 메모리에 존재하는 시간
  - 연결(linkage): 다른 영역에 있는 변수와의 연결 상태



범위

- 변수가 접근되고 사용되는 범위
- 변수 선언 위치에 따라 결정
- 범위의 종류
  - 파일 범위(file scope)
    - 함수의 외부에서 선언, 전역(global) 변수
  - 함수 범위(function scope)
    - 레이블의 범위
  - 블록 범위(block scope)
    - 블록 안에서 선언, 지역(local) 변수



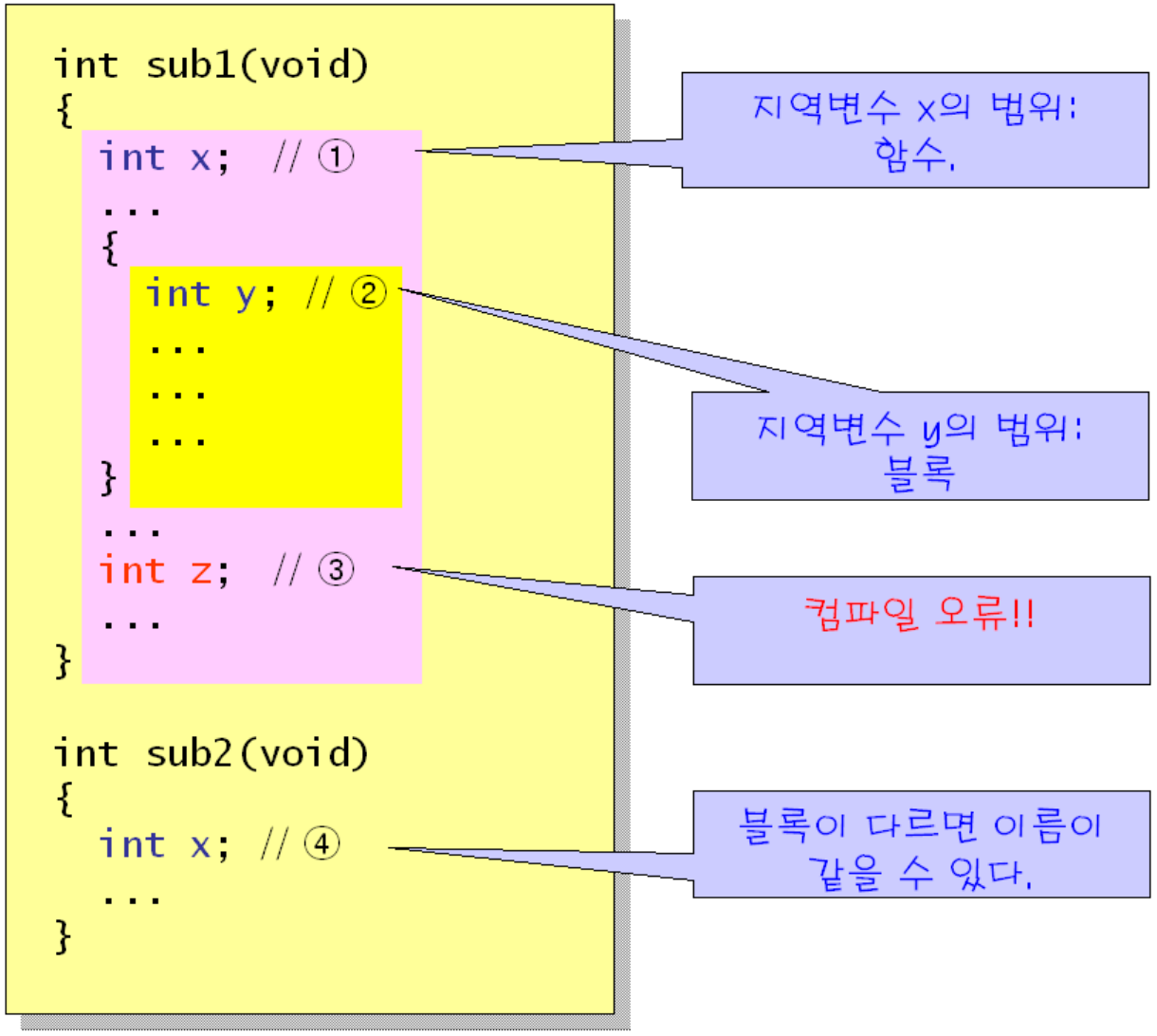
지역 변수

Q) 지역 변수란?

A) 블록 안에서 선언되는 변수

Q) 블록이란?

A) 중괄호로 둘러 싸인 영역



지역 변수

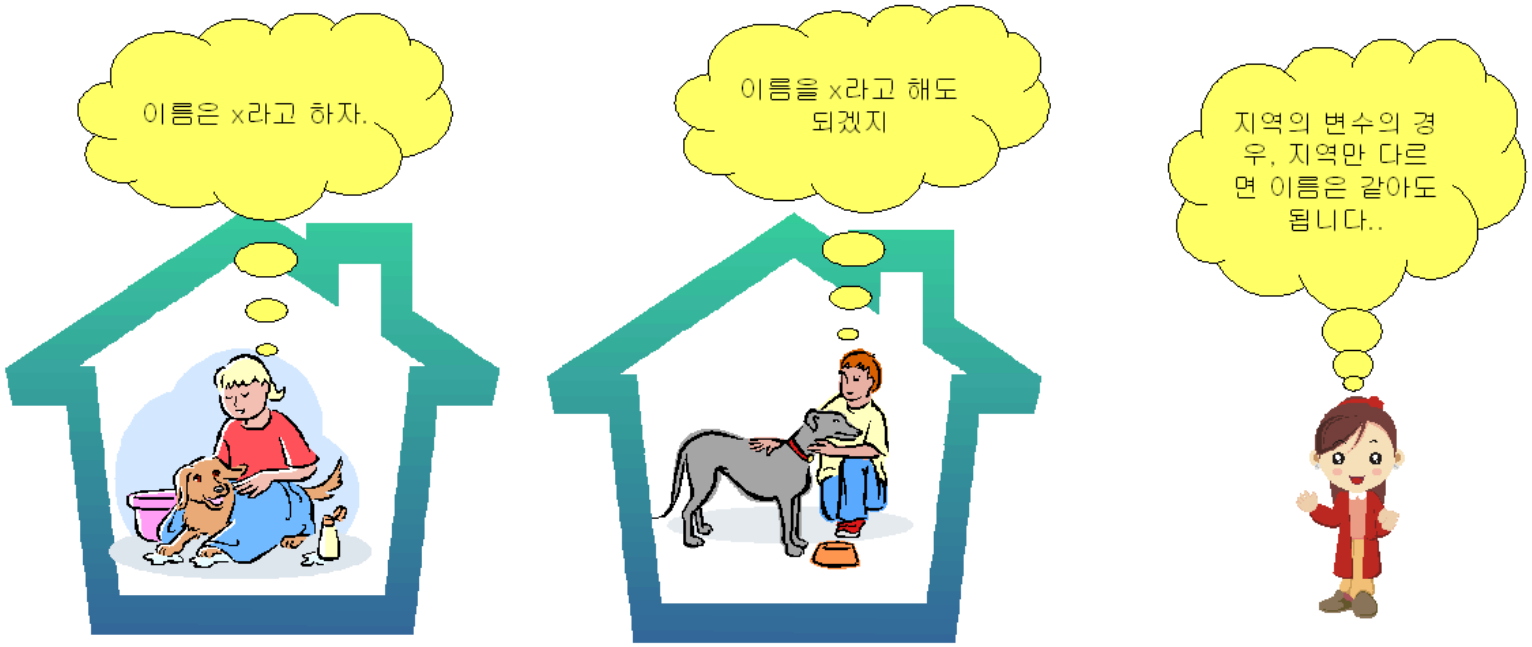
- 지역 변수의 선언: 블록의 맨 첫 부분에서 선언

```
int sub1(void)
{
  int x;   // 함수의 시작 부분에 선언
  int y;   // 오류가 아님
  ...
}
```

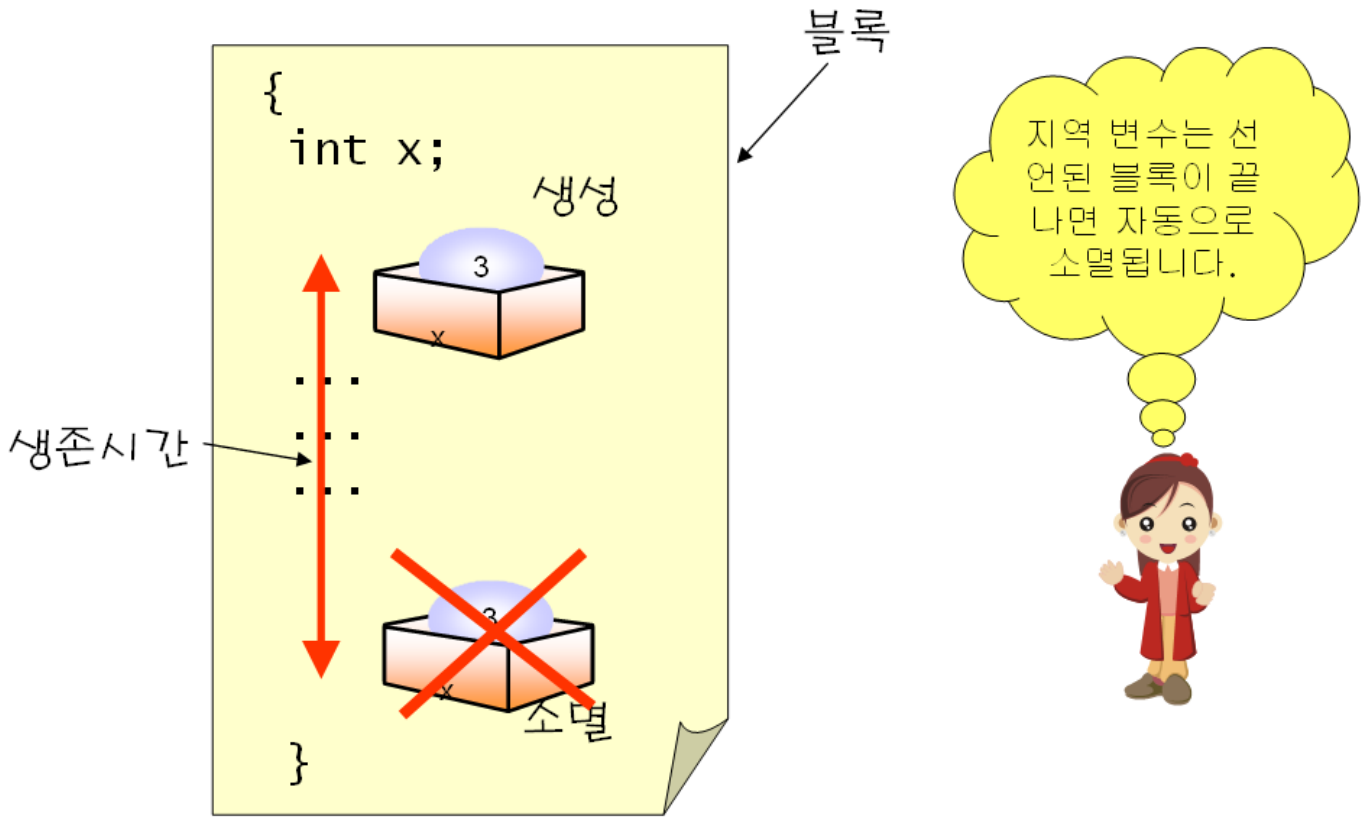
- 지역 변수의 범위: 선언된 블록 안에서만 접근과 사용이 가능

```
void sub1(void)
{
        int x;          // 지역 변수 x 선언
        x = 0;          // OK
        {
                int y;  // 지역 변수 y 선언
                x = 1;  // OK
                y = 2;  // OK
        }
        x = 3;          // OK
        y = 4;          // ① 컴파일 오류!!
}
```

같은 이름의 지역 변수



지역 변수의 생존 기간



지역 변수 예제

```c
#include <stdio.h>

int main(void)
{
        int i;

        for(i = 0;i < 5; i++)
        {
                int temp = 1;
                printf("temp = %d\n", temp);
                temp++;
         }
         return 0;
}
```

블록이 시작할 때마다
생성되어 초기화된다.

```
temp = 1
temp = 1
temp = 1
temp = 1
temp = 1
```

함수의 매개 변수

```c
#include <stdio.h>
int inc(int counter);

int main(void)
{
        int i;

        i = 10;
        printf("함수 호출전 i=%d\n", i);
        inc(i);
        printf("함수 호출후 i=%d\n", i);
        return 0;
}
int inc(int counter)
{
  counter++;
  return counter;
}
```

값에 의한 호출
(call by value)

매개 변수도 일종의
지역 변수임

```
함수 호출전 i=10
함수 호출후 i=10
```

전역 변수

```c
#include <stdio.h>
int x = 123;

void sub1()
{
        printf("In sub1()  x=%d\n",x);     // 전역 변수 x 접근
}
void sub2()
{
        printf("In sub2()  x=%d\n",x);     // 전역 변수 x 접근
}
int main(void)
{
        sub1();
        sub2();

        return 0;
}
```

전역 변수:
함수의 외부에
선언되는 변수

```
In sub1()   x=123
In sub2()   x=123
```

전역 변수의 초기값과 생존 기간

```c
#include <stdio.h>
int counter; // 전역 변수

void set_counter(int i)
{
        counter = i;        // 직접 사용 가능
}
int main(void)
{
        printf("counter=%d\n", counter);

        counter = 100;              // 직접 사용 가능
        printf("counter=%d\n", counter);

        set_counter(20);
        printf("counter=%d\n", counter);

        return 0;
}
```

```
counter=0
counter=100
counter=20
```

* 전역 변수의 **초기값은 0**

*생존 기간은 프로그램 시작부터 종료

전역 변수의 사용

```
// 전역 변수를 사용하여 프로그램이 복잡해지는 경우
#include <stdio.h>
void f(void);

int i;
int main(void)
{
        for(i = 0;i < 5; i++)
        {
                f();
        }
        return 0;
}
void f(void)
{
        for(i = 0;i < 10; i++)
                printf("#");
}
```

##########

같은 이름의 전역 변수와 지역 변수

```c
// 동일한 이름의 전역 변수와 지역 변수
#include <stdio.h>

int sum = 1;        // 전역 변수

int main(void)
{
        int i = 0;
        int sum = 0;        // 지역 변수

        for(i = 0; i <= 10; i++)
        {
                sum += i;
        }
        printf("sum = %d\n", sum);

        return 0;
}
```

지역 변수가 전역 변수를 가린다.

```
sum = 55
```

생존 기간

- 정적 할당(static allocation):
  - 프로그램 실행 시간 동안 계속 유지
- 자동 할당(automatic allocation):
  - 블록에 들어갈때 생성
  - 블록에서 나올때 소멸
- 생존 기간을 결정하는 요인
  - 변수가 선언된 위치
  - 저장 유형 지정자
- 저장 유형 지정자
  - auto
  - register
  - static
  - extern

저장 유형 지정자 auto

- 변수를 선언한 위치에서 자동으로 만들어지고 블록을 벗어나게 되며 자동으로 소멸되는 저장 유형을 지정
- 지역 변수는 auto가 생략

```
int main(void)
{
  auto int sum = 0;
  int i = 0;
  ...
  ...
}
```

전부 자동 변수로서 함수가 시작되면 생성되고 끝나면 소멸된다.

저장 유형 지정자 static

```
#include <stdio.h>
void sub(void);

int main(void)
{
        int i;
        for(i = 0;i < 3; i++)
                sub();
        return 0;
}
void sub(void)
{
        int auto_count = 0;
        static int static_count = 0;

        auto_count++;
        static_count++;
        printf("auto_count=%d\n", auto_count);
        printf("static_count=%d\n", static_count);
}
```

```
auto_count=1
static_count=1
auto_count=1
static_count=2
auto_count=1
static_count=3
```

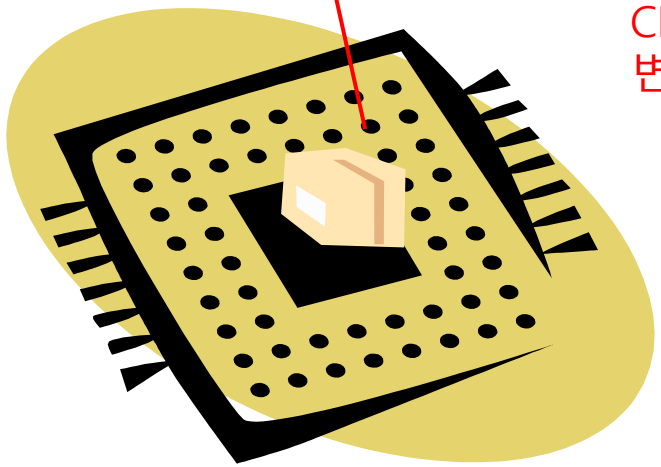
자동 지역 변수

정적 지역 변수로서 static을 붙이면 지역 변수가 정적 변수로 된다.

저장 유형 지정자 register

- 레지스터(register)에 변수를 저장.

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



CPU안의 레지스터에
변수가 저장됨

저장 유형 지정자 extern

컴파일러에게 변수가 다른 곳에서 선언되었음을 알린다

extern1.c

```
#include <stdio.h>

int x;              // 전역 변수
extern int y;       // 현재 소스 파일의 뒷부분에 선언된 변수
extern int z;       // 다른 소스 파일의 변수
int main(void)
{
        extern int x;    // 전역 변수 x를 참조한다. 없어도된다.

        x = 10;
        y = 20;
        z = 30;

        return 0;
}
int y;          // 전역 변수
```

extern2.c

```
int z;
```

연결

- 연결(linkage)
  - 다른 범위에 속하는 변수들을 서로 연결하는 것
- 전역 변수만 가능
- static 지정자를 사용
  - static이 없으면 외부 연결
  - static이 있으면 내부 연결

연결 예제

linkage1.c

```c
#include <stdio.h>
int all_files;          // 다른 소스 파일에서도 사용할 수 있는 전역 변수
static int this_file; // 현재의 소스 파일에서만 사용할 수 있는 전역 변수
extern void sub();

int main(void)
{
        sub();
        printf("%d\n", all_files);
        return 0;
}
```

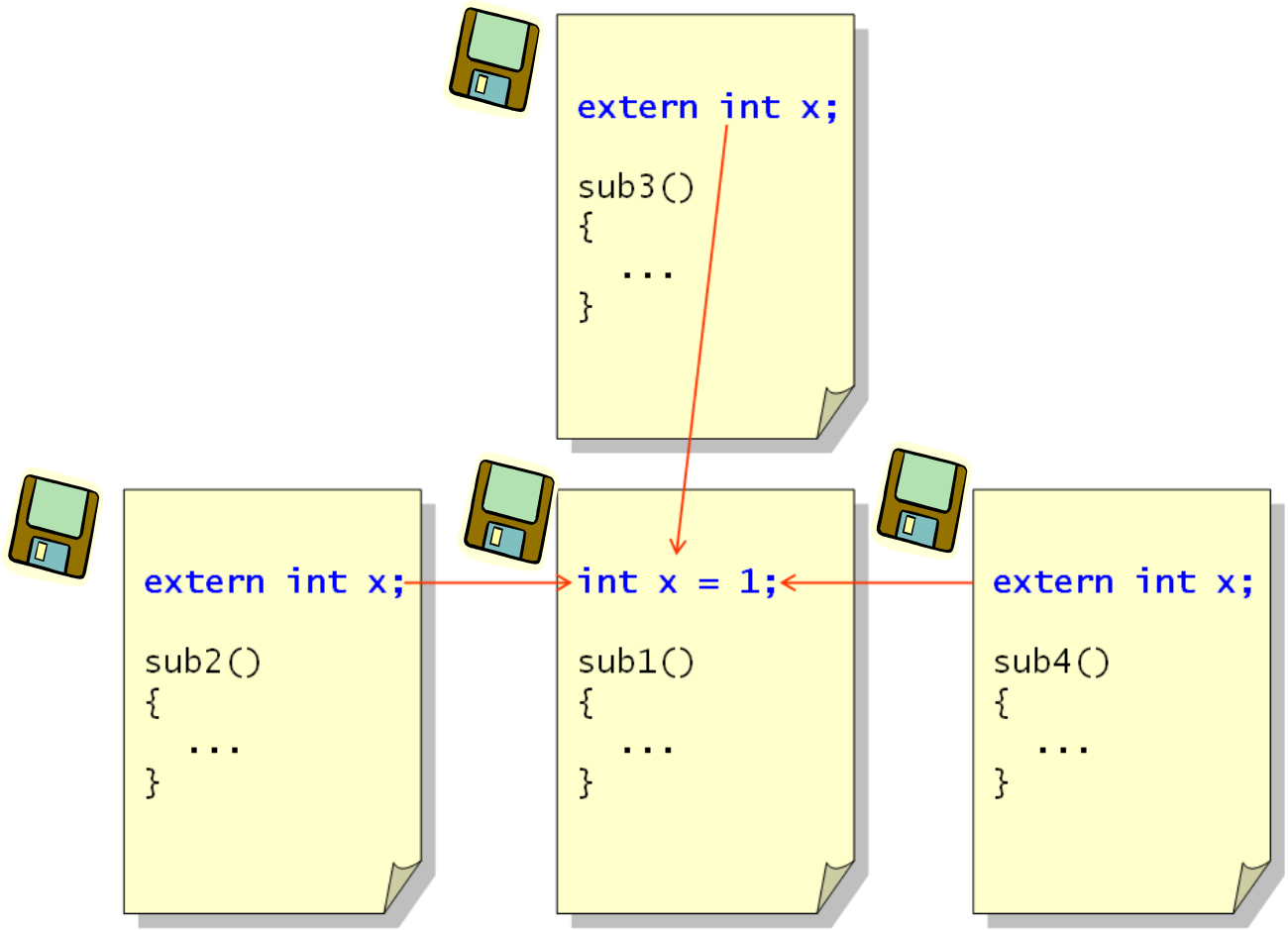
연결

linkage2.c

```c
extern int all_files;
void sub(void)
{
        all_files = 10;
}
```

다중 소스 파일

- 다중 소스 파일에서 변수들을 연결하는데 사용



함수앞의 static

main.c

```c
#include <stdio.h>

extern void f2();
int main(void)
{
        f2();
        return 0;
}
```

static이 붙는 함수는 파일 안에서만 사용할 수 있다

sub.c

```c
static void f1()
{
        printf("f1()이 호출되었습니다.\n");
}

void f2()
{
        f1();
        printf("f2()가 호출되었습니다.\n");
}
```

재귀 (recursion) 함수

- 팩토리얼의 반복적 구현

$$n! = \begin{cases} 1 & n = 1 \\ n*(n-1)*(n-2)*\cdots*1 & n \geq 2 \end{cases}$$

```
int factorial_iter(int n)
{
    int k, value=1;
    for(k=1; k<=n; k++)
        value = value*k;
    return(value);
}
```

재귀 함수

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법

- 팩토리얼의 **재귀적 정의**

$$n! = \begin{cases} 1 & n = 1 \\ n*(n-1)! & n \geq 2 \end{cases}$$

팩토리얼 구하기 (1)

- (n-1)! 팩토리얼을 현재 작성중인 함수를 다시 호출하여 계산(재귀 호출)

```
int factorial(int n)
{
    if( n == 1 ) return(1);
    else return (n * factorial(n-1) );
}
```

3!은?

3!를 계산하려면 3! = 3*2!

2!를 계산하려면 2! = 2*1!

1!은 1

3!는? 6

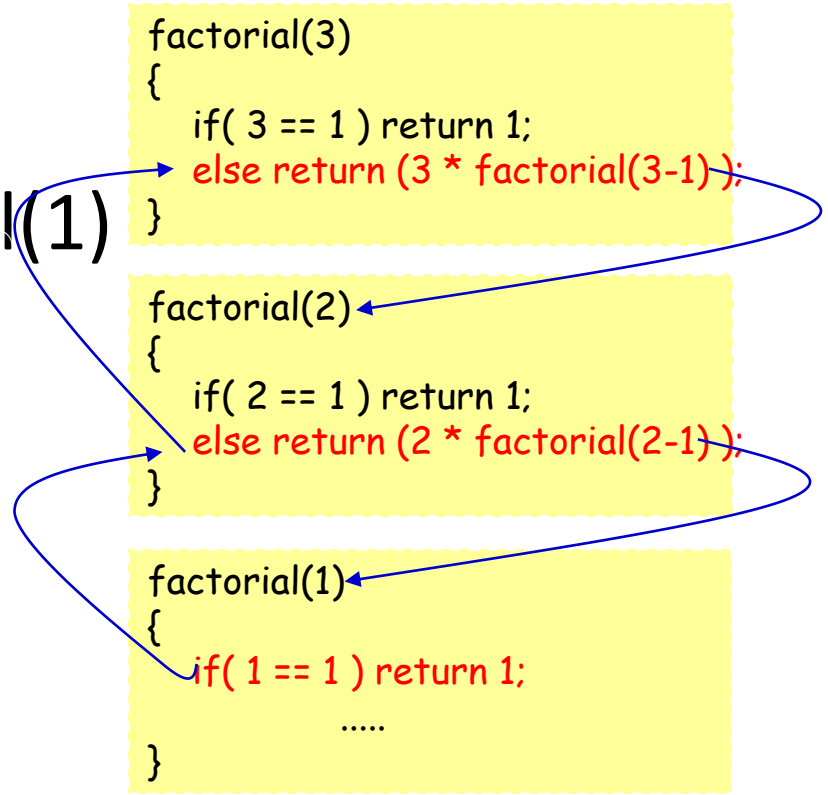
2!는? 2

1!는? 1

팩토리얼 구하기 (2)

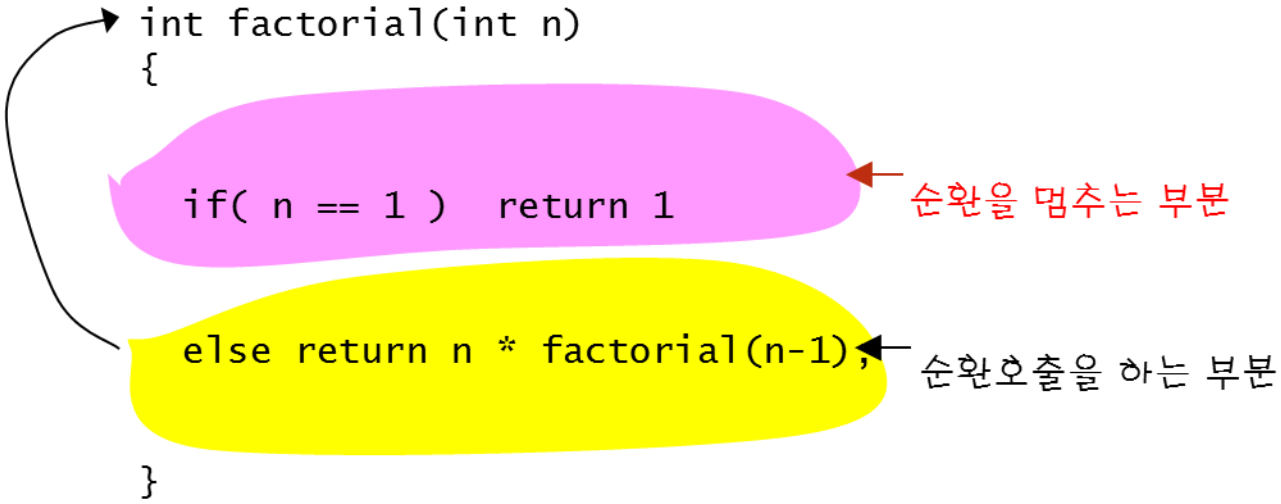
- 팩토리얼의 호출 순서

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1
= 6



재귀 알고리즘의 구조

- 재귀 호출 부분
- 재귀 호출을 멈추는 부분



- 만약 순환 호출을 멈추는 부분이 없다면?.
  - 시스템 오류가 발생할 때까지 무한정 호출하게 된다.

재귀 <-> 반복

- 컴퓨터에서의 되풀이
  - 재귀(recursion): 재귀 호출 이용
  - 반복(iteration): for나 while을 이용한 반복

- 대부분의 재귀는 반복으로 바꾸어 작성 가능
- 재귀
  - 문제 자체가 재귀적인 정의를 가지는 경우, 코딩이 단순
  - 함수 호출의 오버헤드가 큼
- 반복
  - 빠른 수행속도
  - 재귀적 정의를 갖는 문제의 경우, 반복문을 이용한 구현이 어려울 수 있음

KU KONKUK UNIVERSITY

C 프로그래밍

함수 II: 재귀 호출 실습

재귀 호출 위치에 따른 동작

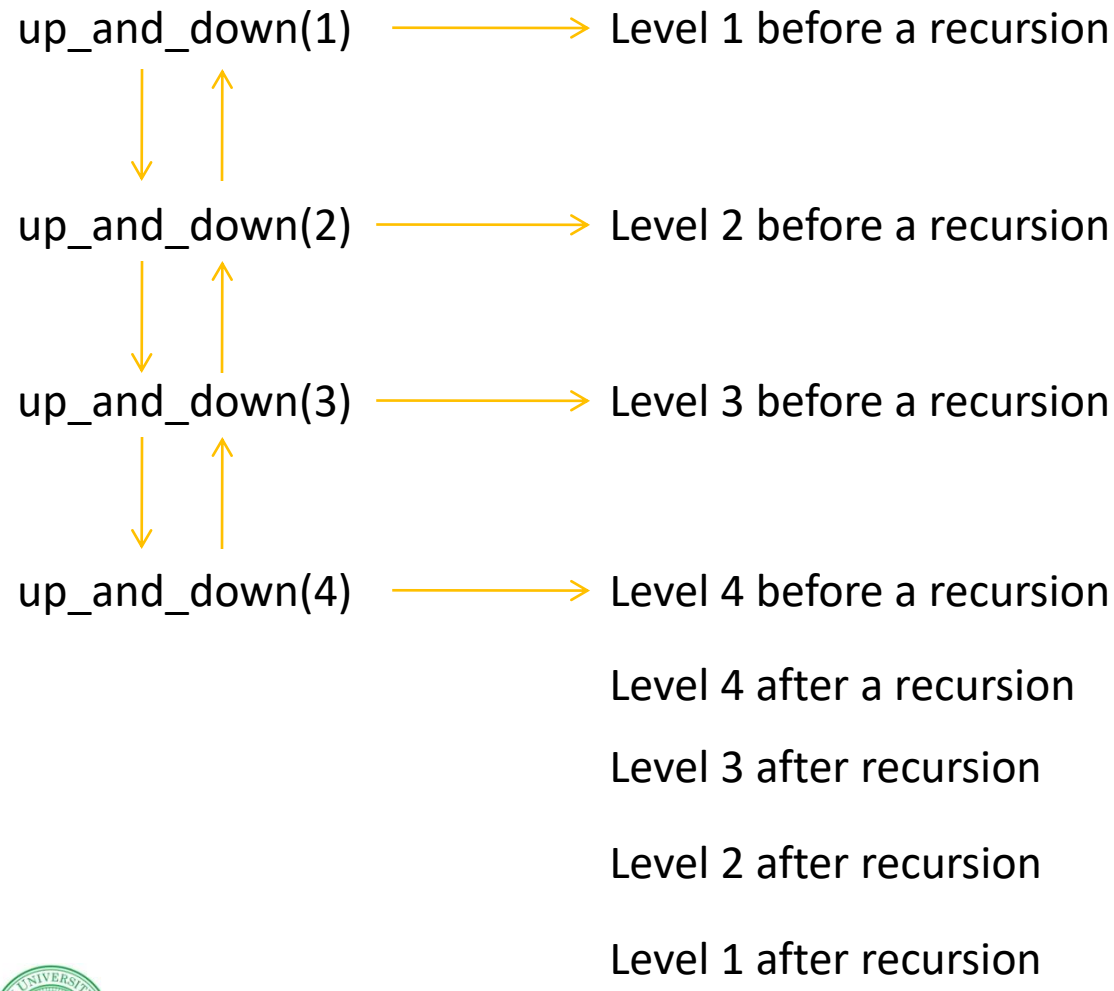
- Tail Recursion
  - 함수의 가장 마지막 부분에서 재귀 호출 발생
  - 함수의 모든 코드가 순차적으로 반복 수행

- 함수 중간에서 재귀 호출이 발생하면?

Programming Practice I

```
#include <stdio.h>
void up_and_down(int n);
int main(void)
{
  up_and_down(1);
  return 0;
}
void up_and_down(int n)
{
  printf("Level %d before a recursion\n", n);
  if (n < 4)
     up_and_down(n+1);
  printf("Level %d after a recursion\n", n);
}
```

Steps of Recursion

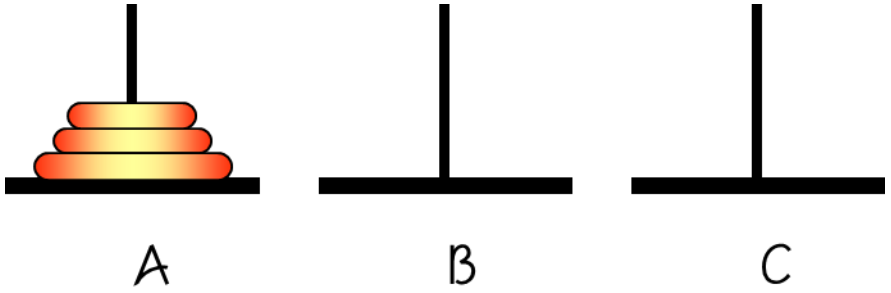



```
void up_and_down(int n)
{
  printf("Level %d before a recursion\n", n);
  if (n< 4)
    up_and_down(n+1);
  printf("Level %d after a recursion\n", n);
}
```

STACK

하노이 타워 (Hanoi Tower)

- 문제는 막대 A에 쌓여있는 원판 3개를 막대 C로 이동
- 단, 다음의 조건을 지켜야 함
  - 한 번에 하나의 원판만 이동할 수 있다
  - 맨 위에 있는 원판만 이동할 수 있다
  - 크기가 작은 원판 위에 큰 원판이 쌓일 수 없다.
  - 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다



Programming Practice II

- 피보나치 수열 구하기
  - 0, 1, 1, 2, 3, 5, 8, 13, 21.....

$$fib(n)\begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$$

- my.h

```
#include <stdio.h>
#include <time.h>
 int fib(int);
 int fib_iter(int);
```

main.c

```
#include "my.h"

int main(void)
{
        int n;
        time_t  t1, t2;
        printf("정수 입력하세요: ");
        scanf("%d",&n);
        t1 = time(NULL);
        printf("피보나치 수: %d\n", fib_iter(n));
        t2 = time(NULL);
        printf("반복문 시간: %d\n", t2-t1);
        t1 = time(NULL);
        printf("피보나치 수: %d\n", fib(n));
        t2 = time(NULL);
        printf("재귀 호출 시간 : %d\n", t2-t1);

        return 0;

}
```

반복문 vs. 재귀호출

- fib.c

```
int fib(int n)
{
        if( n==0 ) return 0;
        if( n==1 ) return 1;
        return (fib(n-1) + fib(n-2));
}
```

Test input: 10, 20, 30, 40

- fib_iter.c

```
int fib_iter(int n)
{
        if( n < 2 )
                return n;
        else
        {
                int i, tmp, f1=1, f0=0;
                for(i = 2;i <= n; i++)
                {
                        tmp = f1;
                        f1 += f0;
                        f0 = tmp;
                }
                return f1;
        }
}
```