

# 쉽게 풀어쓴 C언어 Express

## 제9장 함수와 변수



Prof. Jung Guk Kim  
CESE, HUFS  
jgkim@hufs.ac.kr

# 이번 장에서 학습할 내용



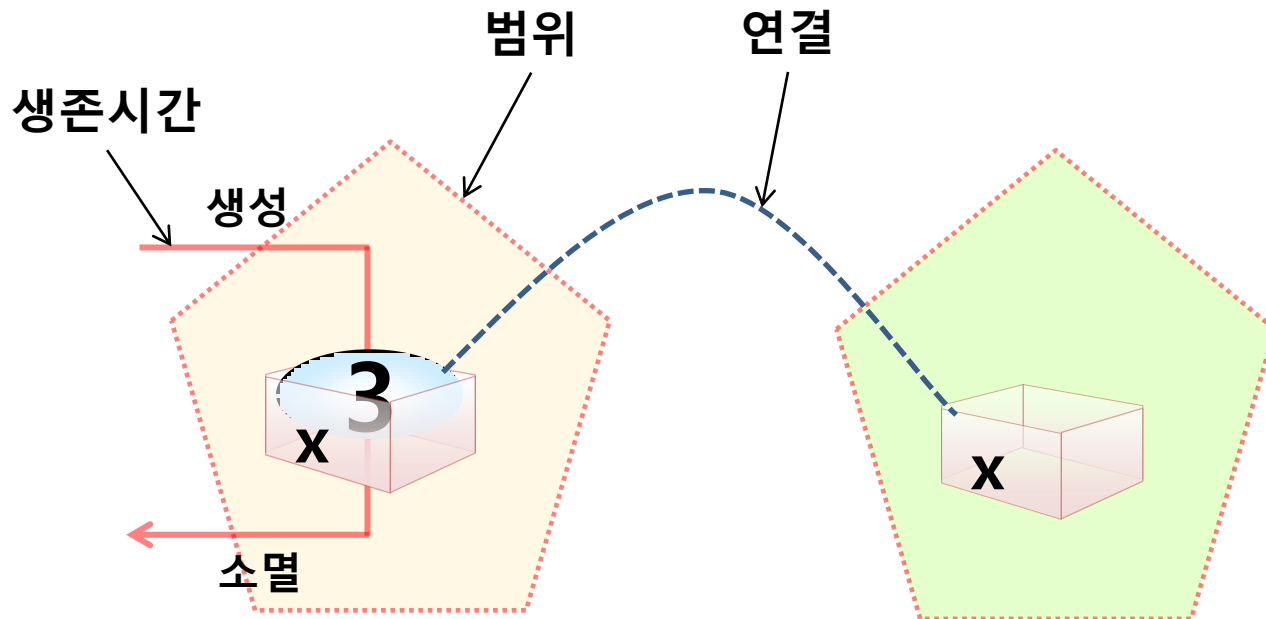
- 반복의 개념 이해
- 변수의 속성
- 전역, 지역 변수
- 자동 변수와 정적 변수
- 재귀 호출

이번 장에서는  
함수와 변수와의  
관계를 집중적으로  
살펴볼 것이다. 또한  
함수가 자기 자신을  
호출하는 재귀  
호출에 대하여  
살펴본다.

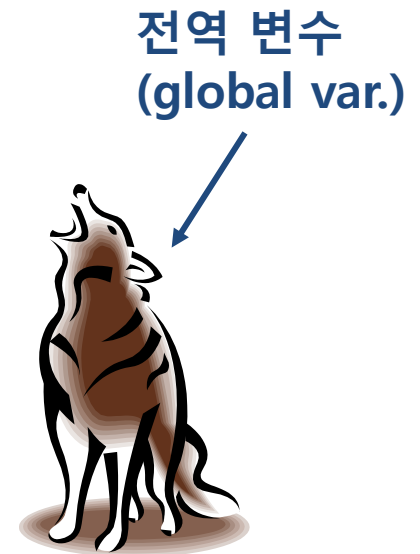
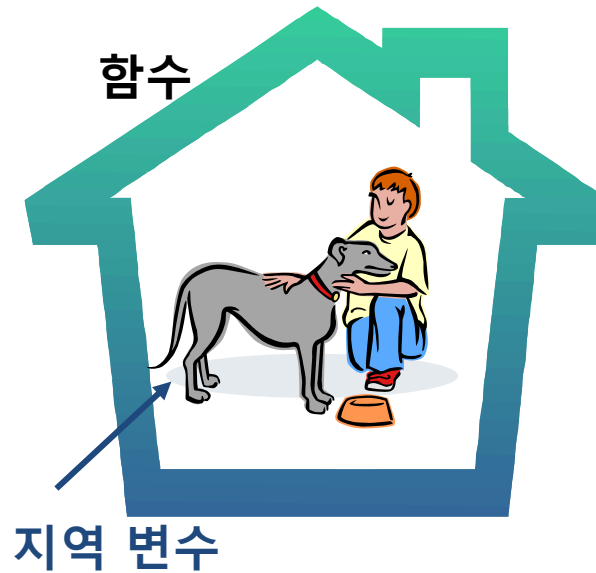
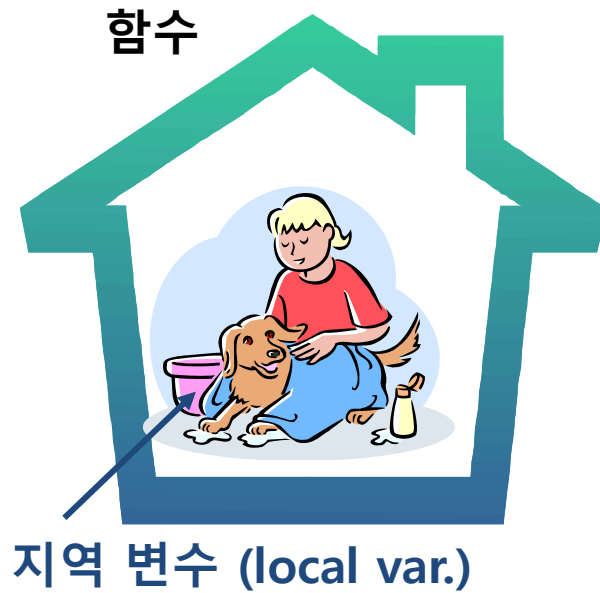


# 변수의 속성

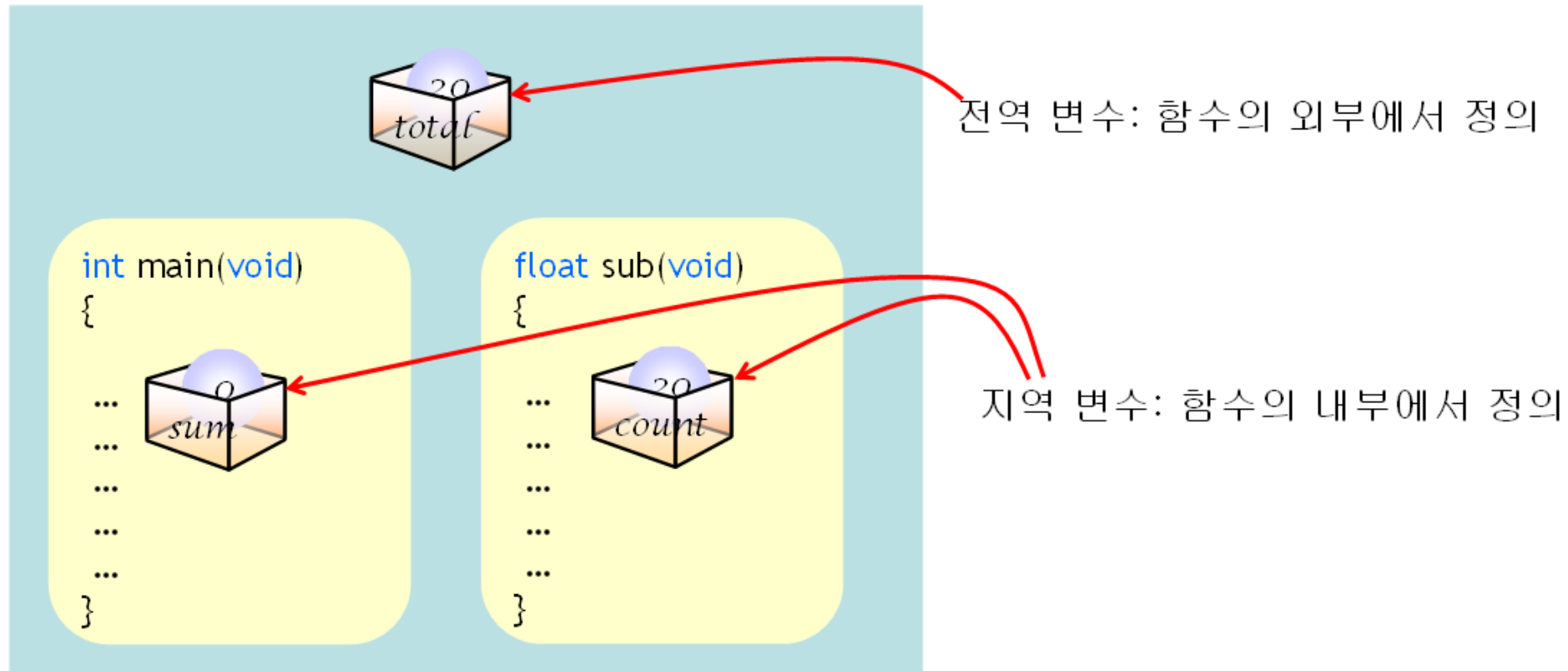
- 변수의 속성 : 이름, 타입, 크기, 값 + 범위, 생존 시간, 연결
  - 범위(scope) : 변수가 사용 가능한 범위, 가시성
  - 생존 시간(lifetime) : 메모리에 존재하는 시간
  - 연결(linkage) : 다른 영역(파일)에 있는 변수와의 연결



# 변수의 범위

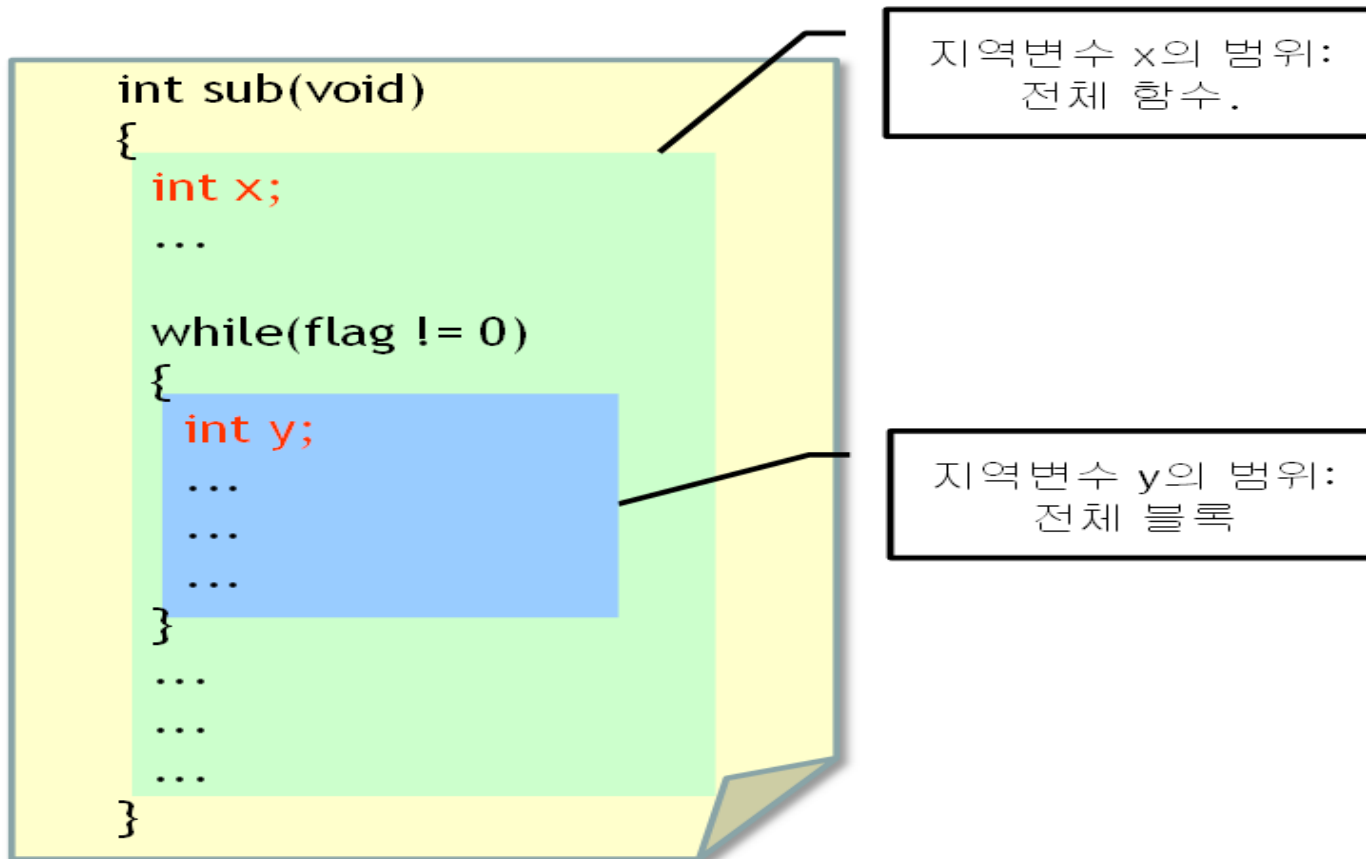


# 전역 변수(Global Var.)와 지역 변수 (Local Var.)



# 지역 변수 (Local Variables)

- 지역 변수(local variable)는 블록 안에 선언되는 변수



# 지역 변수 선언 위치

블록 첫 부분에서 정의

```
int sub(void)
{
    int x; // ①

    ...

    x = 100;

    int y; // ②
}
```

변수를 함수의 첫 부분에 선언하지 않으셨군요. 컴파일 오류입니다.

C++는 가능



# 지역 변수의 범위

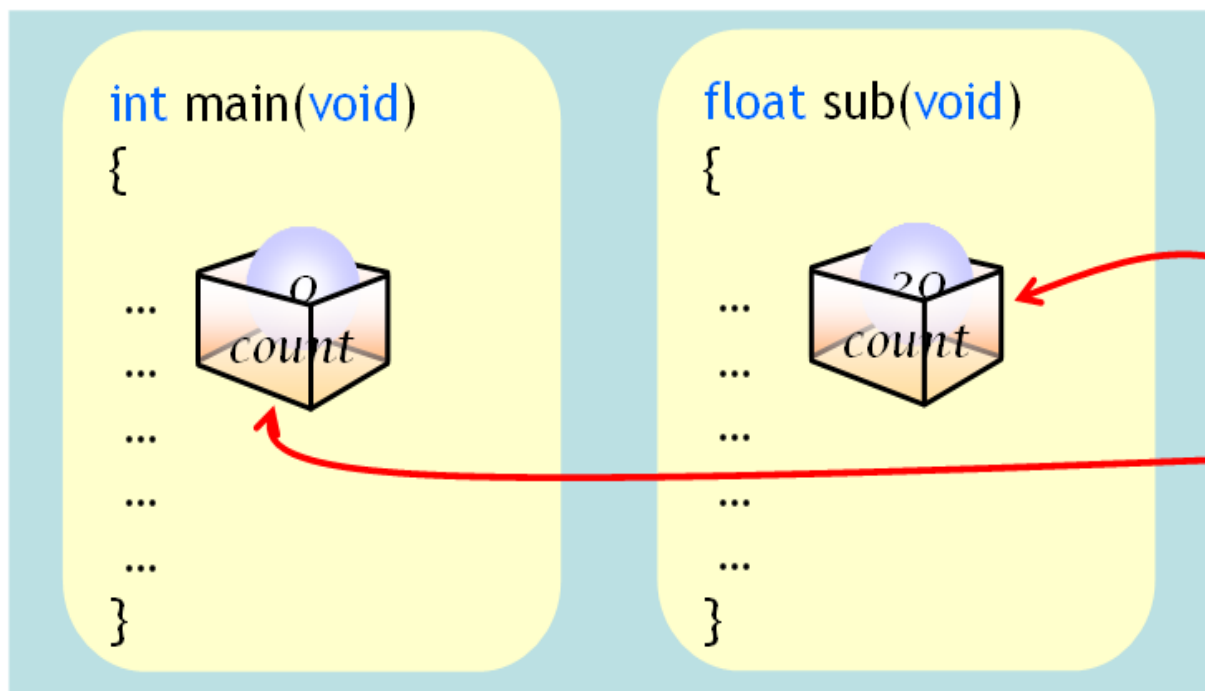
```
void sub1(void)
{
    {
        int y;
        ...
    }
    y = 4;
}
```

지역 변수는 선언된 블록을 떠나면 안됩니다.





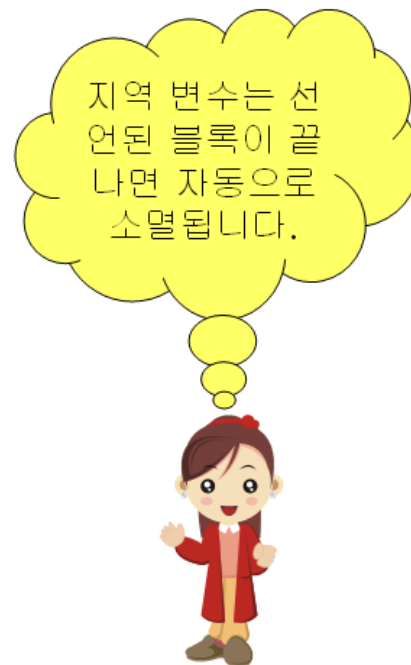
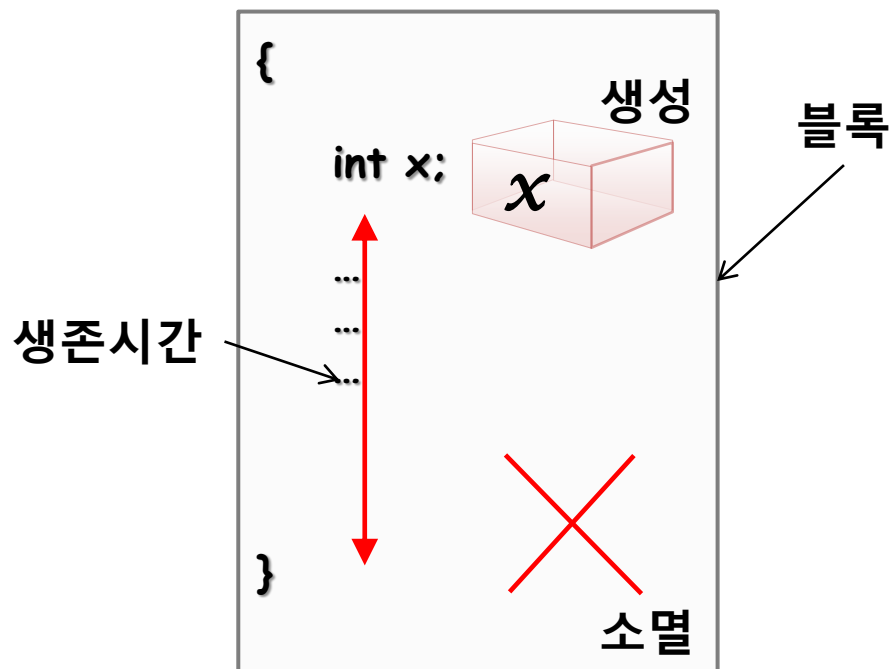
# 이름이 같은 지역 변수



블록만 다르면  
이름은 같아도  
됩니다.



# 지역 변수의 생존 기간



# 지역 변수 예제

```
#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 때 마다  
생성되어 초기화된다.

```
temp = 1
```

```
temp = 1
```

```
temp = 1
```

```
temp = 1
```

```
temp = 1
```

# 지역 변수의 초기값

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

```
int main(void)
```

```
{
```

```
    int temp;
```

```
    printf("temp = %d\n", temp);
```

```
}
```

초기화 되지  
않았으므로 쓰레기  
(unknown) 값을  
가진다.



# 함수의 매개 변수

매개 변수도 일종의  
지역 변수

```
int inc(int counter)
{
    counter++;
    return counter;
}
```

# 함수의 매개 변수

```
#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

# Argument Passing

## ▪ Call by value

- Actual **argument의 값이** 함수 내의 local var.인 formal argument에 copy 되어 수행된다.
- Argument로 사용된 calling function의 변수는 변화되지 않는다.

## ▪ Call by reference

- 함수에 **argument의 주소를** passing하는 방법으로 추후 설명

# 전역 변수

- 전역 변수(global variable)는 함수 외부에서 선언되는 변수이다.
- 전역 변수의 범위(scope)는 소스 파일 전체이다.
- 전역 변수의 생존기간(life time)은 main()이 끝날 때까지

```
int x = 123;

void sub1()
{
    x = 456;
}

void sub2()
{
    x = 789;
}
```

전역 변수



# 전역 변수의 초기값과 생존 기간

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

```
int counter;
```

```
void set_counter()
```

```
{
```

```
    counter = 20;    // 직접 사용 가능
```

```
}
```

```
int main(void)
```

```
{
```

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

```
    set_counter();
```

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

```
    return 0;
```

```
}
```

전역 변수  
초기값은 0

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

전역  
변수의  
범위

# 전역 변수의 사용

// 전역 변수를 사용하여 프로그램이 복잡해지는 경우: side effect

```
#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("#");
```

```
}
```

출력은  
어떻게  
될까요?



#####

# 전역 변수의 사용

1. 거의 모든 함수에서 사용하는 공통적인 데이터만 전역 변수로 한다.
2. 일부의 함수들만 사용하는 데이터는 전역 변수로 하지 말고 함수의 인수로 전달한다.
3. 많은 전역변수의 사용은 **side-effect**를 유발하고, modular program의 취지에 어긋난다.

# 같은 이름의 전역 변수와 지역 변수

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

```
int sum = 1; // 전역 변수
```

전역 변수와 지역 변수가  
동일한 이름으로 선언된 경우

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

```
    int sum = 0; // 지역 변수
```

```
    printf("sum = %d\n", sum); // 이 문장의 sum은 local or global?  
    // 항상 local (scope 상 가까운 선언)이 우선한다!
```

```
    return 0;
```

```
}
```

sum = 0

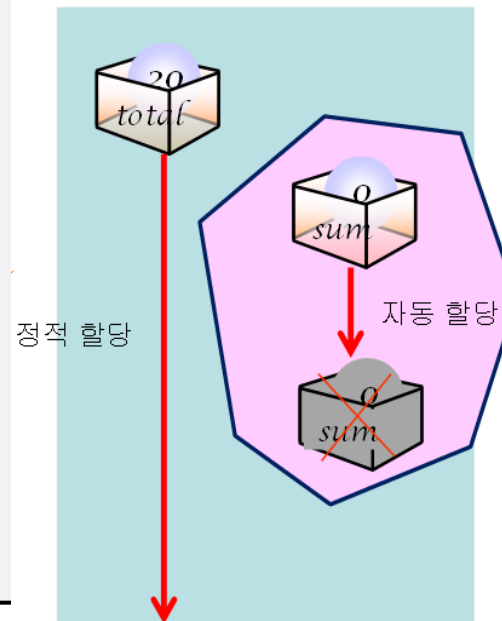
# 중간 점검

- 변수의 범위는 대개 무엇으로 결정되는가?
- 변수의 범위에는 몇 가지의 종류가 있는가?
- 지역 변수를 블록의 중간에서 정의할 수 있는가?
- 지역 변수가 선언된 블록이 종료되면 지역 변수는 어떻게 되는가?
- 지역 변수의 초기값은 얼마인가?
- 전역 변수는 어디에 선언되는가?
- 전역 변수의 생존 기간과 초기값은?
- 똑같은 이름의 전역 변수와 지역 변수가 동시에 존재하면 어떻게 되는가?



# Static vs. Automatic Allocation

- 정적 할당(static allocation):
  - 프로그램 실행 시간 동안 계속 유지
- 자동 할당(automatic allocation):
  - 블록에 들어갈 때 생성
  - 블록에서 나올 때 소멸



정적 할당은 변수가 실행 시간 내내 존재하지만 자동 할당은 블록이 종료되면 소멸됩니다.



# 생존 기간 (Life Time)

## ■ 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 생략되어도 자동 변수가 된다. (default)

```
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); // 함수 내에서 값을 계속 유지하여야 할 때!
```

```
}
```

```
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.

함수 내에서 값을 계속 유지하여야 할 때!

# Memory Hierarchy & Caching/Buffering

## ■ Access speed

- Registers > Cache memory > Memory > Flash/Optical > Disk

## ■ Capacity

- Registers < Cache memory < Memory < Flash/Optical < Disk

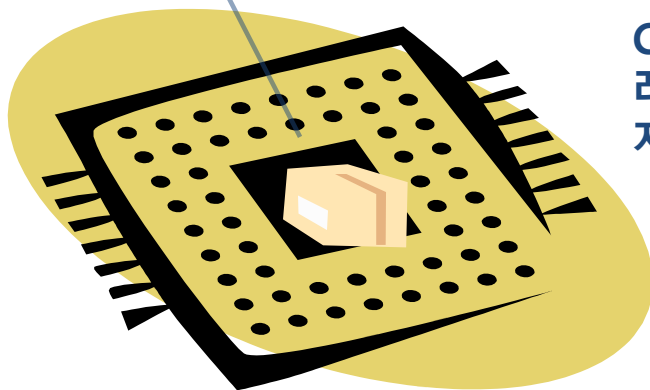
## ■ Objectives (Memory Hierarchy)

- To enhance the total access speeds and to enlarge the total capacity by “Caching/Buffering”
- Caching in memory hierarchy: 하위 메모리에서 자주 쓰이는 것을 미리 상위 메모리에 fetch 하고 가능한 한 유지한다.
- 접근 시에는 상의 cache에 있는 먼저 찾아본다.
- Cache memory <- Memory : H/W caching
- Memory <- Disk : S/W caching by OS

# 저장 유형 지정자: register

- CPU 레지스터(register)에 변수를 할당 (프로그램 속도 향상을 위해) (register: 기계어가 연산 시에 주로 사용)
- Compiler가 register 할당 도모하나, 반드시 성공하는 것은 아님.
- Register 부족 시에는 memory에 할당된다.

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



CPU안의  
레지스터에 변수가  
저장됨

# 중간 점검

- 저장 유형 지정자에는 어떤 것들이 있는가?
- 지역 변수를 정적 변수로 만들려면 어떤 지정자를 붙여야 하는가?
- 변수를 CPU 내부의 레지스터에 저장시키는 지정자는?
- 컴파일러에게 변수가 외부에 선언되어 있다고 알리는 지정자는?
- `static` 지정자를 변수 앞에 붙이면 무엇을 의미하는가?



# 실습: 로그인 횟수 제한

- 로그인 시에 제한된 횟수 만큼 지속적으로 틀리면 로그인 시도를 막는 프로그램을 작성하여 보자.



# 실행 결과



# 알고리즘

1. while(1)
2.     사용자로부터 아이디를 입력 받는다.
3.     사용자로부터 패스워드를 입력 받는다.
4.     만약 로그인 시도 횟수가 일정 한도를 넘었으면  
          프로그램을 종료한다.
5.     아이디와 패스워드가 일치하면 로그인 성공 메시지를  
          출력한다.
6.     아이디와 패스워드가 일치하지 않으면 다시 시도한다.



```
#include <stdio.h>
#include <stdlib.h>
#define SUCCESS 1
#define FAIL 2
#define LIMIT 3

int check(int id, int password);

int main(void)
{
    int id, password, result;

    while(1) {
        printf("id: ____\b\b\b\b");
        scanf("%d", &id);
        printf("pass: ____\b\b\b\b");
        scanf("%d", &password);
        result = check(id, password);
        if( result == SUCCESS ) break;
    }
    printf("로그인 성공\n");
    return 0;
}
```



```
int check(int id, int password)
{
    static int super_id = 1234;
    static int super_password = 5678;
    static int try_count = 0;

    try_count++;
    if( try_count >= LIMIT ) {
        printf("횟수 초과\n");
        exit(1);
    }
    if( id == super_id && password == super_password )
        return SUCCESS;
    else
        return FAIL;
}
```

정적 지역 변수

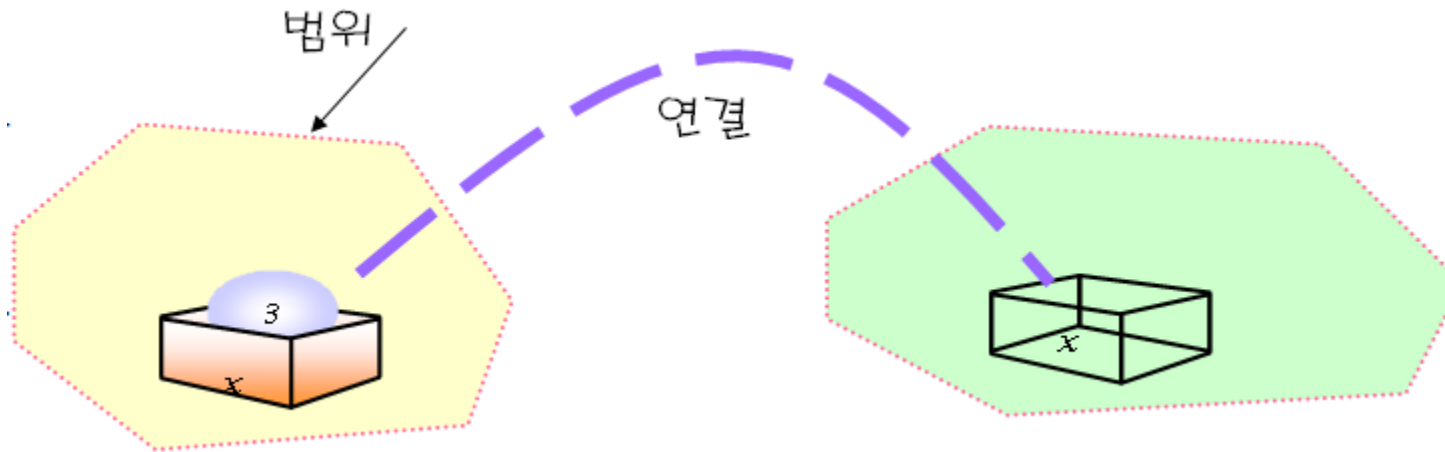
# 도전문제

- 위의 프로그램은 정적 지역 변수를 사용한다. 정적 전역 변수를 사용할 수도 있다. 정적 전역 변수를 사용하도록 위의 프로그램을 변경하여 보자.
- 아이디와 패스워드에 대하여 시도 횟수를 다르게 하여 보자.
- 은행 입출금 시스템을 간단히 구현하고 여기에 로그인 기능을 추가하여 보자.



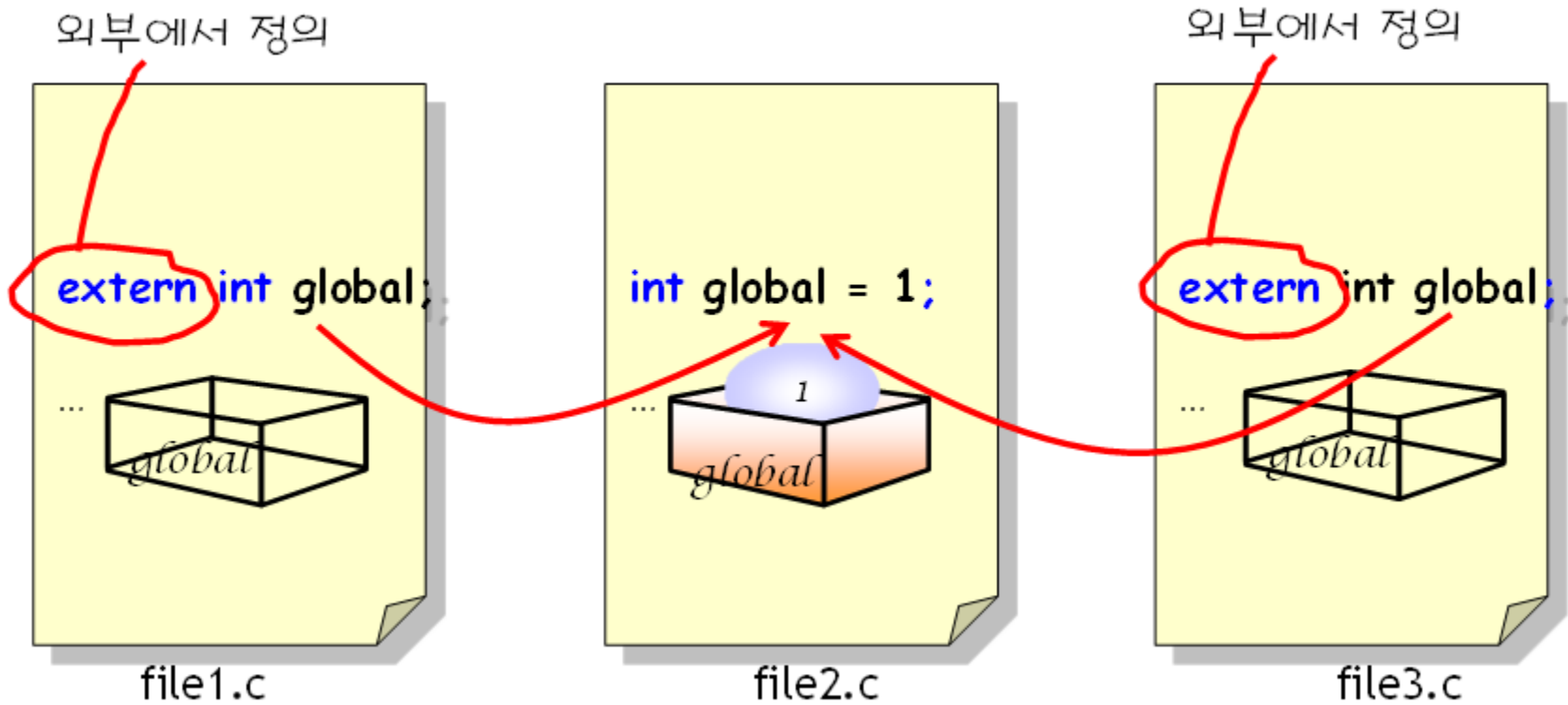
# 연결 (Linkage)

- 연결(linkage): 다른 범위(파일)에 속하는 변수들을 서로 연결하는 것
  - 외부 연결
  - 내부 연결
  - 무연결
- 전역 변수만이 연결을 가질 수 있다.



# 외부 연결

- 전역 변수를 extern을 이용하여서 서로 연결



# 연결 예제

*linkage1.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*

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



10

# 저장 유형 지정자 extern

## extern1.c

```
#include <stdio.h>
int x;           // 전역 변수
extern int z;     // 다른 소스 파일의 변수
extern int y;     // 현재 소스 파일의 뒷부분에 선언된 변수
int main(void)
{
    extern int x; // 전역 변수 x를 참조한다. 없어도 된다.

    x = 10;
    y = 20;
    z = 30;
    return 0;
}
int y;           // 전역 변수
```

## extern2.c

```
int z;
```

# 함수 앞의 static

*main.c*

```
#include <stdio.h>

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

static이 붙는 함수는 파일 안에서만 사용할 수 있다.

*sub.c*

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

# 저장 유형 정리

- 일반적으로는 **자동 저장 유형** 사용 권장
- 자주 사용되는 변수는 **레지스터 유형**
- 변수의 값이 함수 호출이 끝나도 그 값을 유지하여야 할 필요가 있다면 **지역 정적**
- 만약 많은 파일에서 공유되어야 하는 변수라면 **외부 참조 변수**

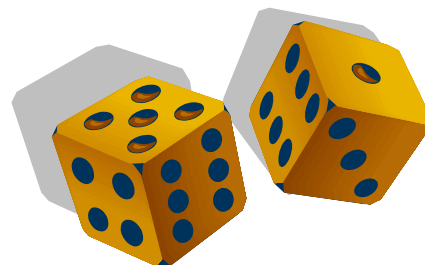
저장 유형	키워드	정의되는 위치	범위	생존 시간
자동	auto	함수 내부	지역	임시
레지스터	register	함수 내부	지역	임시
정적 지역	static	함수 내부	지역	영구
전역	없음	함수 외부	모든 소스 파일	영구
정적 전역	static	함수 외부	하나의 소스 파일	영구
외부 참조	extern	함수 외부	모든 소스 파일	영구



# 예제: 난수 발생기

- 자체적인 난수 발생기 작성
- 이전에 만들어졌던 난수를 이용하여 새로운 난수를 생성할 수 있다. 따라서 함수 호출이 종료되더라도 이전에 만들어졌던 난수를 어딘가에 저장하고 있어야 한다

$$r_{n+1} = (a \cdot r_n + b) \bmod M$$



# 예제

main.c

```
#include <stdio.h>
unsigned random_i(void);
double random_f(void);

extern unsigned call_count;    // 외부 참조 변수

int main(void)
{
    register int i;           // 레지스터 변수

    for(i = 0; i < 10; i++)
        printf("%d ", random_i());

    printf("\n");

    for(i = 0; i < 10; i++)
        printf("%f ", random_f());

    printf("\n함수가 호출된 횟수= %d \n", call_count);
    return 0;
}
```

# 예제

random.c

```
// 난수 발생 함수  
#define SEED 17  
#define MULT 25173  
#define INC 13849  
#define MOD 65536
```



48574 61999 40372 31453 39802 35227 15504  
29161 14966 52039  
0.885437 0.317215 0.463654 0.762497 0.546997  
0.768570 0.422577 0.739731 0.455627 0.720901  
함수가 호출된 횟수 = 20

```
unsigned int call_count = 0; // 전역 변수  
static unsigned seed = SEED; // 정적 전역 변수
```


```
unsigned random_i(void)  
{  
    seed = (MULT*seed + INC) % MOD;  
    call_count++;  
    return seed;  
}  
  
double random_f(void)  
{  
    seed = (MULT*seed + INC) % MOD;  
    call_count++;  
    return seed / (double) MOD;  
}
```

# 가변 매개 변수

- 매개 변수의 개수가 가변적으로 변할 수 있는 기능

`int sum (int num, ...)`

호출 때 마다 매개  
변수의 개수가 변경될  
수 있다.



# 가변 매개 변수

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

```
#include <stdarg.h>
```

```
int sum( int, ... );
```

```
int main( void )
```

```
{
```

```
    int answer = sum( 4, 4, 3, 2, 1 );
```

```
    printf( "합은 %d입니다.\n", answer );
```

```
    return( 0 );
```

```
}
```

```
int sum( int num, ... )
```

```
{
```

```
    int answer = 0;
```

```
    va_list argptr;                // 인수 목록의 주소
```

```
    va_start ( argptr, num );      // 인수 목록의 초기화 macro
```

```
    for( ; num > 0; num-- )
```

```
        answer += va_arg( argptr, int ); // 다음 인수의 fetch macro
```

```
    va_end( argptr );              // 인수 목록의 정리 macro
```

```
    return( answer );
```

```
}
```



합은 10입니다.

매개 변수의 개수

# 순환(recursion)이란?

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

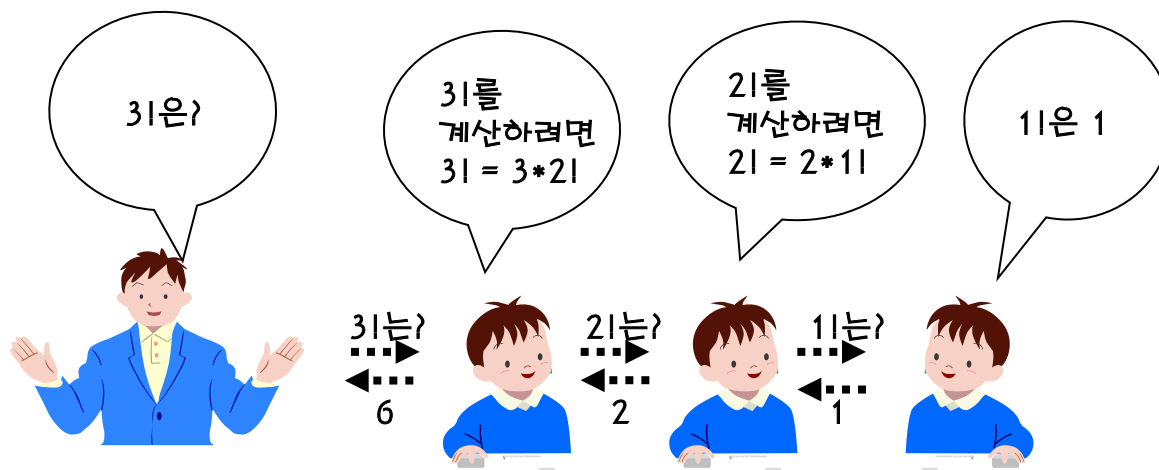
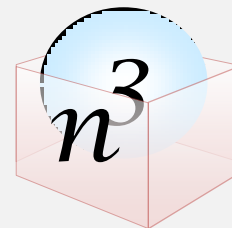
$$n! = \begin{cases} 1 & n = 1 \\ n * (n-1)! & n \geq 2 \end{cases}$$



# 팩토리얼 구하기

- ◆ 팩토리얼 프로그래밍 #2:  $(n-1)!$  팩토리얼을 현재 작성중인 함수를 다시 호출하여 계산(순환 호출)

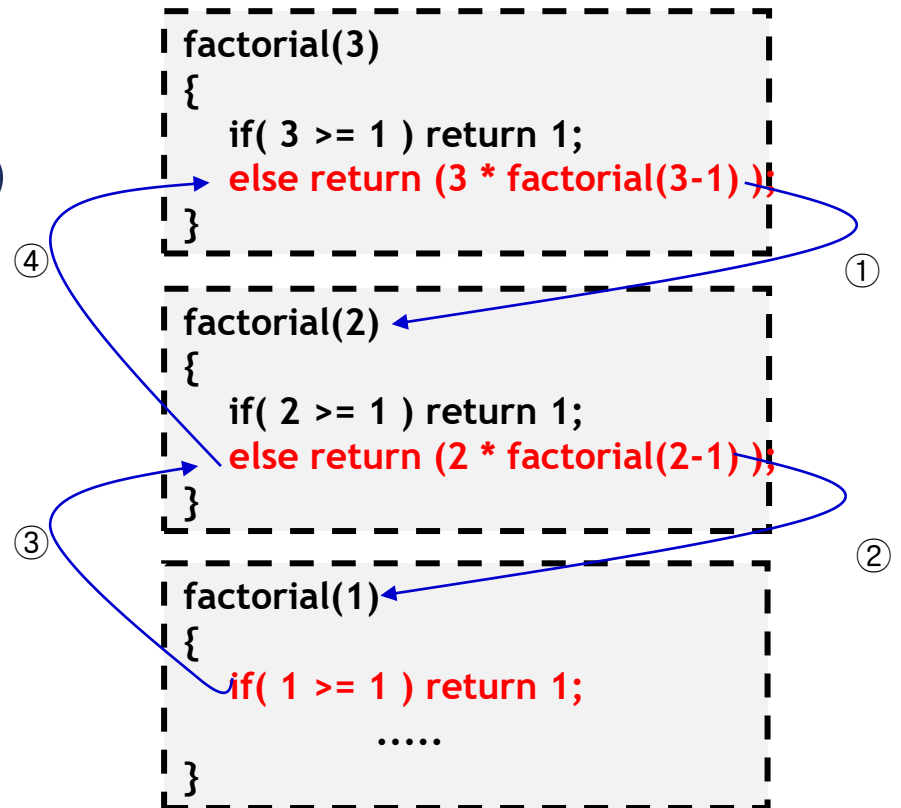
```
int factorial(int n)
{
    if( n <= 1 ) return(1);
    else return (n * factorial(n-1) );
}
```



# 팩토리얼 구하기

## ■ 팩토리얼의 호출 순서

$\text{factorial}(3) = 3 * \text{factorial}(2)$   
 $= 3 * 2 * \text{factorial}(1)$   
 $= 3 * 2 * 1$   
 $= 3 * 2$   
 $= 6$





# 순환 알고리즘의 구조

## ■ 순환 알고리즘은 다음과 같은 부분들을 포함한다.

- 순환 호출을 하는 부분
- 순환 호출을 멈추는 부분

```
int factorial(int n)
{
```

```
    if( n == 1 ) return 1
```

← 순환을 멈추는 부분

```
    else return n * factorial(n-1);
```

← 순환호출을 하는 부분

```
}
```

◆ 만약 순환 호출을 멈추는 부분이 없다면?.

- 시스템 오류가 발생할 때까지 무한정 호출하게 된다.

# 피보나치 수열의 계산 #1

◆ 순환 호출을 사용하면 비효율적인 예

◆ 피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

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

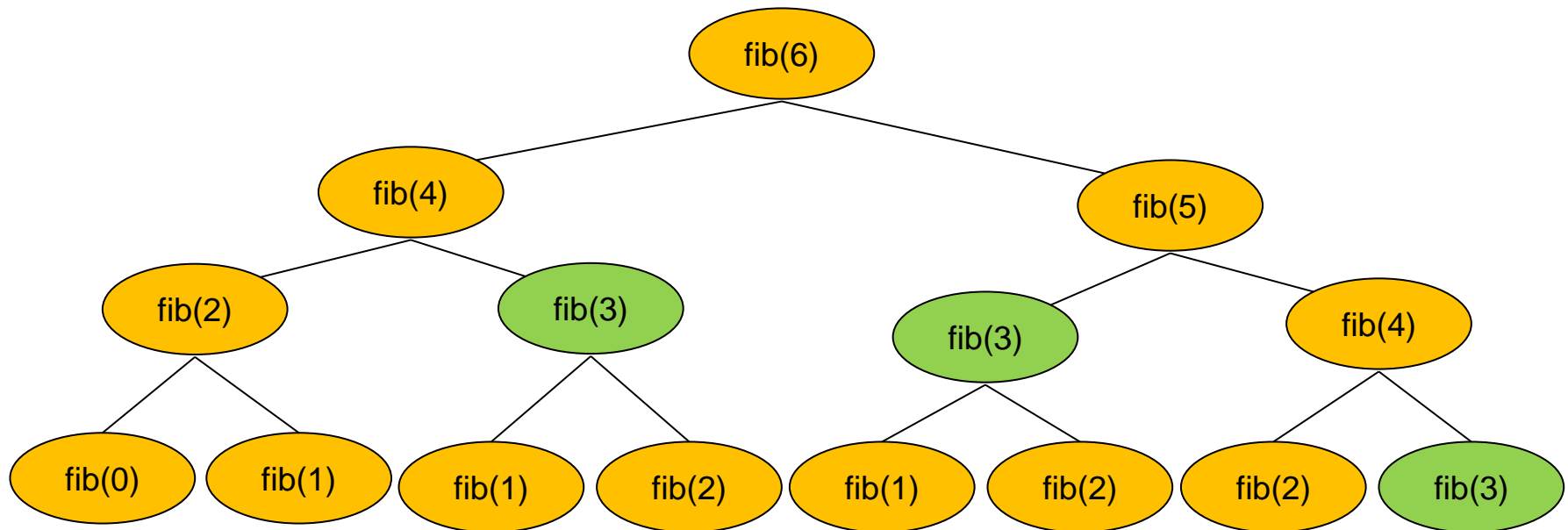
◆ 순환적인 구현

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

# 피보나치 수열의 계산

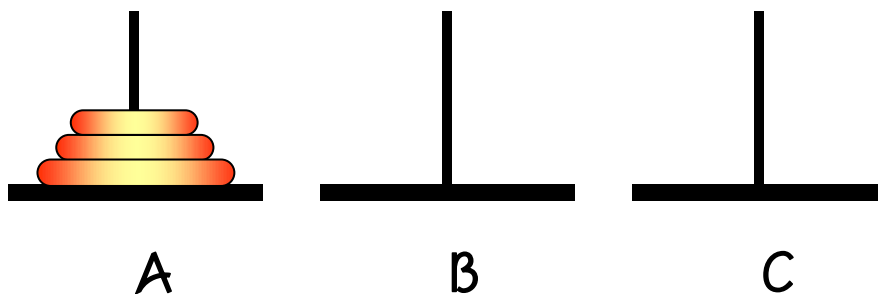
## ■ 순환 호출을 사용했을 경우의 비효율성

- 같은 항이 중복해서 계산됨
- 예로,  $\text{fib}(6)$ 을 호출하게 되면  $\text{fib}(3)$ 이 4번이나 중복되어서 계산됨
- 이러한 현상은  $n$ 이 커지면 더 심해짐

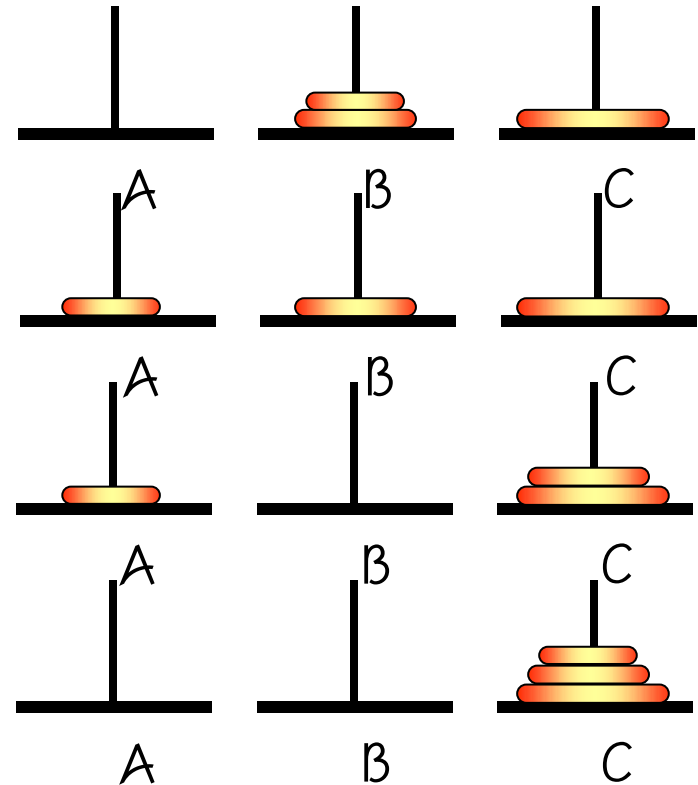
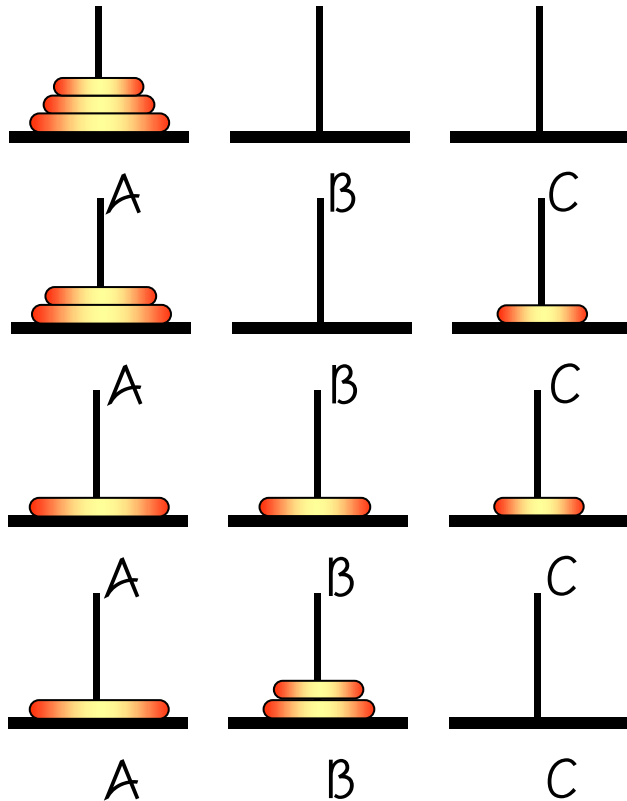


# 하노이 탑(Hanoi Tower) 문제 #1

- 막대 A에 쌓여있는 원판 3개를 막대 C로 옮기는 문제. 단 다음의 조건을 지켜야 한다.
  - 한 번에 하나의 원판만 이동할 수 있다
  - 맨 위에 있는 원판만 이동할 수 있다
  - 크기가 작은 원판 위에 큰 원판이 쌓일 수 없다.
  - 중간막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.

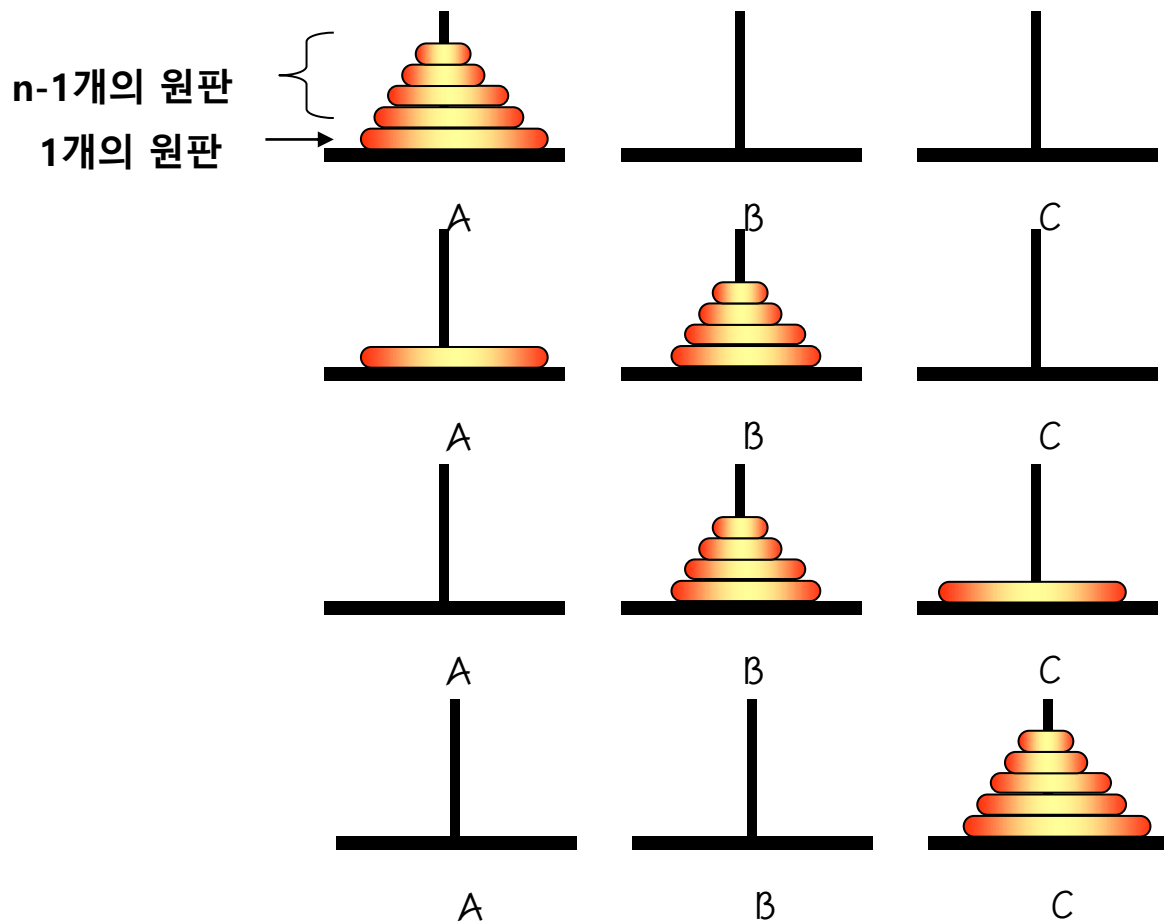


# 3개의 원판인 경우의 해답



# n개의 원판인 경우: 문제의 분리

1.  $n-1$ 개의 원판을 A에서 B로 옮기고,
2.  $n$ 번째 원판을 A에서 C로 옮긴 다음,
3.  $n-1$ 개의 원판을 B에서 C로 옮기면 된다.



# 하노이탑 알고리즘

// 막대 from에 쌓여있는 n개의 원판을 막대 tmp를 사용하여 막대 to로 옮긴다.

```
void hanoi_tower (int n, char from, char tmp, char to)
{
    if (n == 1)
    {
        from에서 to로 원판을 옮긴다.
    }
    else
    {
        hanoi_tower(n-1, from, to, tmp);
        from에 있는 한 개의 원판을 to로 옮긴다.
        hanoi_tower(n-1, tmp, from, to);
    }
}
```

# 하노이탑 실행 결과

```
#include <stdio.h>

void hanoi_tower(int n, char from, char tmp, char to)
{
    if( n==1 )
        printf("원판 1을 %c 에서 %c으로 옮긴다.\n",from,to);
    else {
        hanoi_tower(n-1, from, to, tmp);
        printf("원판 %d을 %c에서 %c으로 옮긴다.\n",n,
            from, to);
        hanoi_tower(n-1, tmp, from, to);
    }
}

int main(void)
{
    hanoi_tower(4, 'A', 'B', 'C');
    return 0;
}
```

원판 1을 A 에서 B으로 옮긴다.  
원판 2을 A에서 C으로 옮긴다.  
원판 1을 B 에서 C으로 옮긴다.  
원판 3을 A에서 B으로 옮긴다.  
원판 1을 C 에서 A으로 옮긴다.  
원판 2을 C에서 B으로 옮긴다.  
원판 1을 A 에서 B으로 옮긴다.  
원판 4을 A에서 C으로 옮긴다.  
원판 1을 B 에서 C으로 옮긴다.  
원판 2을 B에서 A으로 옮긴다.  
원판 1을 C 에서 A으로 옮긴다.  
원판 3을 B에서 C으로 옮긴다.  
원판 1을 A 에서 B으로 옮긴다.  
원판 2을 A에서 C으로 옮긴다.  
원판 1을 B 에서 C으로 옮긴다.



# 이진수 출력하기

- 정수를 이진수로 출력하는 프로그램 작성
- 순환 알고리즘으로 가능
  - binary (6)
    - $6/2 \rightarrow 3, 0$  (몫, 나머지)
    - $3/2 \rightarrow 1, 1$
    - $1/2 \rightarrow 0, 1$
    - 몫이 0 이면 return
  - Answer : 1 1 0 (**역순으로** print 해야 함)
  - 따라서 binary(6)은 binary(3)을 실행한 후 자신의 나머지 0을 print
  - binary(3)은 binary(1)을 실행한 후 자신의 나머지 1을 print
  - ...

# 순환 호출 예제

// 2진수 형식으로 출력

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

```
void print_binary(int x);
```

```
int main(void)
```

```
{
```

```
    print_binary(9);
```

```
    return 0;
```

```
}
```

```
void print_binary(int x)
```

```
{
```

```
    if( x > 0 )
```

```
    {
```

```
        print_binary(x / 2);
```

```
        printf("%d", x % 2);
```

```
    }
```

```
}
```



// 순환 호출

// 나머지를 출력

# Recursion vs. Iteration

- Recursion으로 할 수 있는 것은 iteration으로도 가능하다.
- Recursion은 Hanoi tower problem과 같이 iteration으로 algorithm을 생각하기 어려운 경우 자주 사용된다.
- Recursion은 CPU와 memory의 낭비가 iteration보다 크다.
- 따라서 생각 하기 쉬운 algorithm의 경우에는 iteration을 사용하는 것이 좋다. (ex. Factorial, binary,...)
- Recursion을 iteration으로 변환하는 algorithm이 있다. (Algorithm 강의에서 다룸.)