

개정3판

Visual  
Studio  
2017

쉽게 풀어쓴

C언어  
EXPRESS


천인국 지음

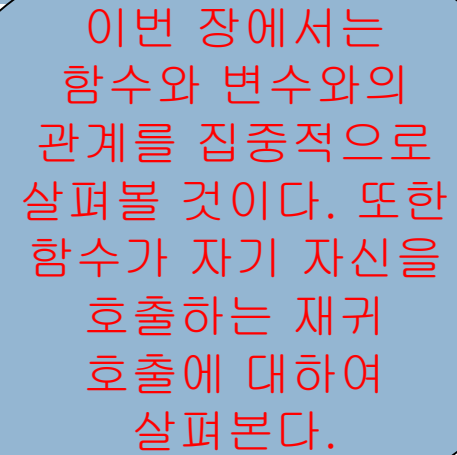
## 제9장 함수와 변수



# 이번 장에서 학습할 내용

2

- 
- 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.

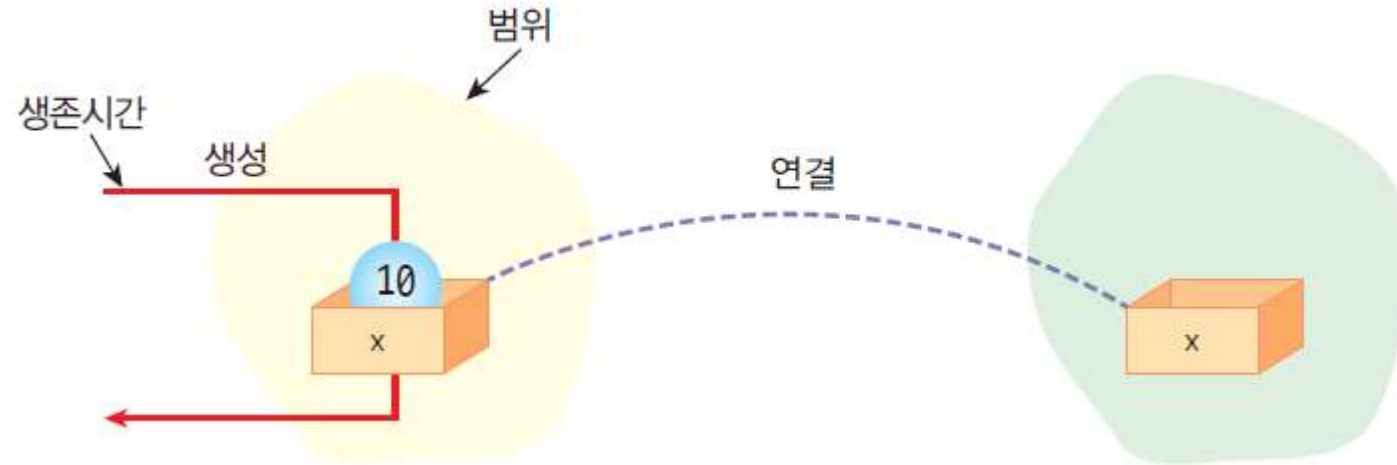




# 변수의 속성

3

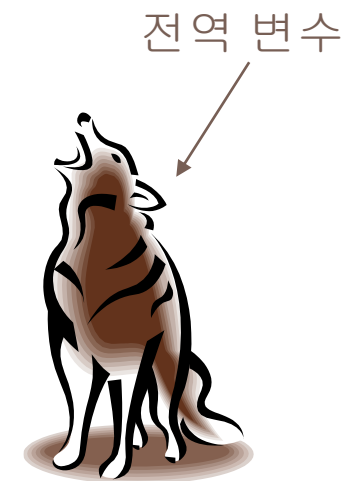
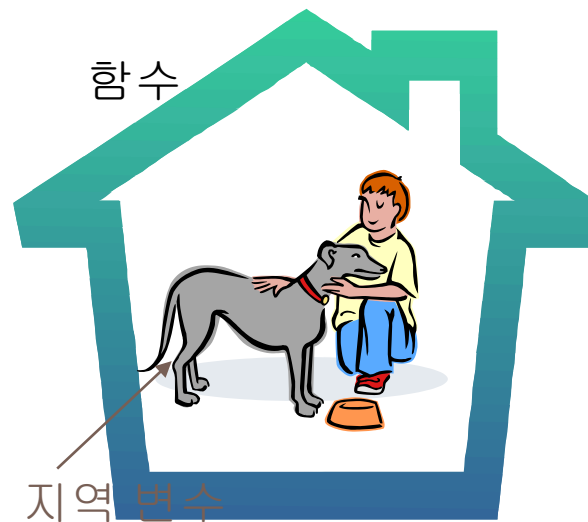
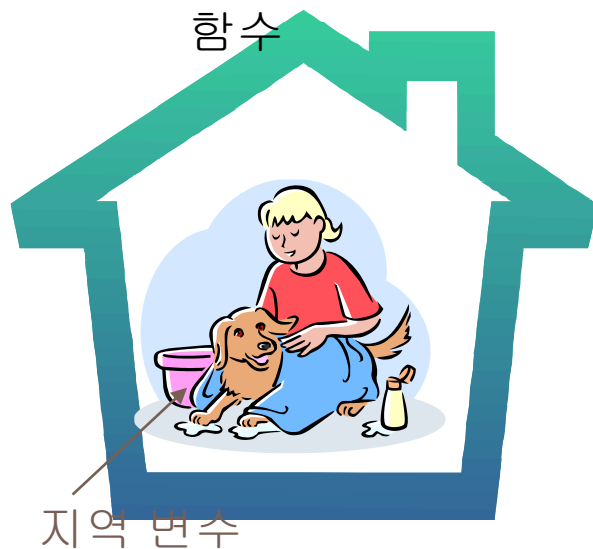
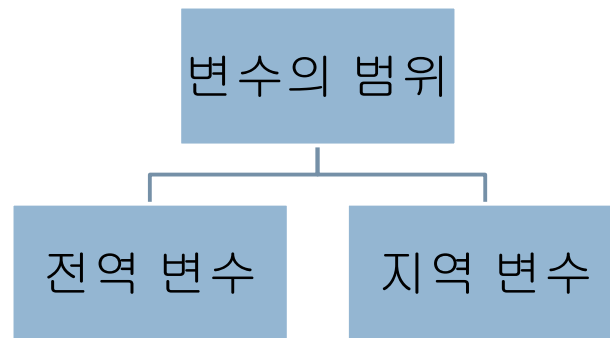
- 변수의 속성 : 이름, 타입, 크기, 값 + 범위, 생존 시간, 연결
  - 범위(scope) : 변수가 사용 가능한 범위, 가시성
  - 생존 시간(lifetime) : 메모리에 존재하는 시간
  - 연결(linkage) : 다른 영역에 있는 변수와의 연결 상태





# 변수의 범위

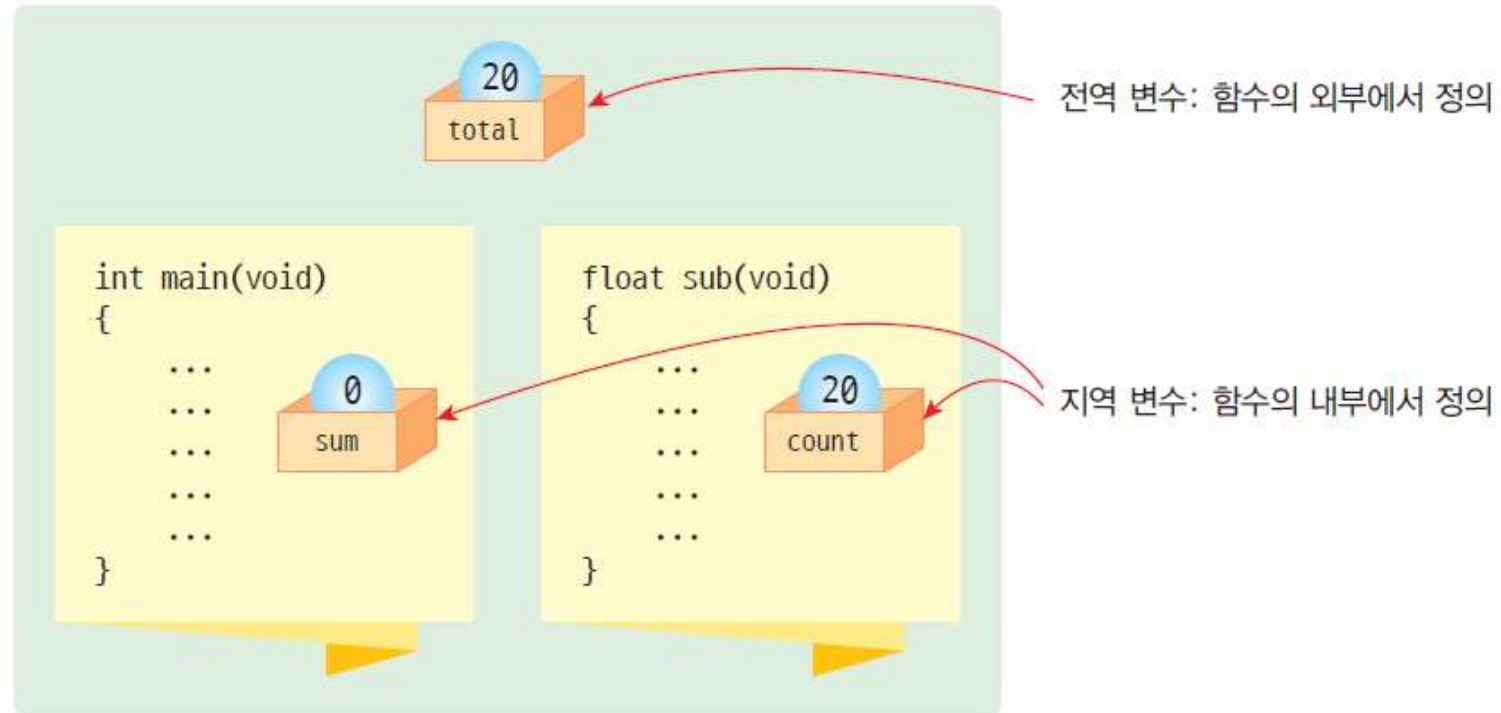
4





# 전역 변수와 지역 변수

5





# 지역 변수

6

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

```
int sub(void)
{
    int x = 0;

    while(flag != 0){
        int y;
        ...
    }

    y = 0;    // 오류!!
    ...
}
```

지역 변수 x가 사용가능한 범위

지역 변수 y가 사용가능한 범위

지역 변수는 선언된 블록을 떠나면 안됩니다.

y가 선언된 블록을 벗어나서 사용하였으므로 오류!



# 지역 변수 선언 위치

7

- 최신 버전의 C에서는 블록 안의 어떤 위치에서도 선언 가능!!

```
while(flag!=0) {
```

```
...
```

```
int x = get_integer();
```

```
...
```

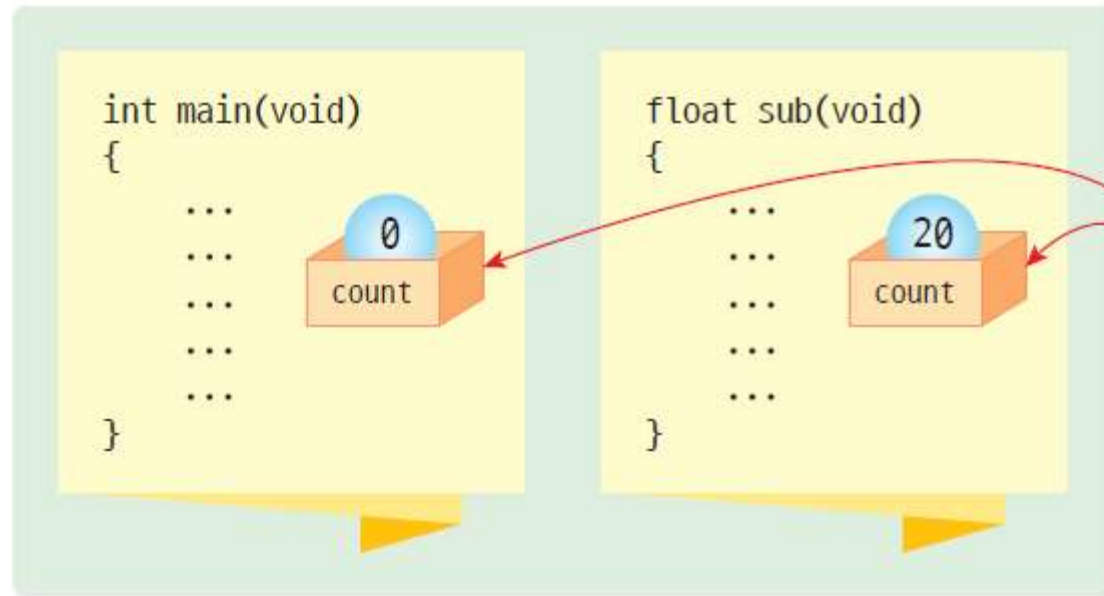
```
}
```

블록의 중간에서도 얼마든지  
지역 변수를 선언할 수 있다.



# 이름이 같은 지역 변수

8



블록만 다르면  
이름은 같아도 됩니다.

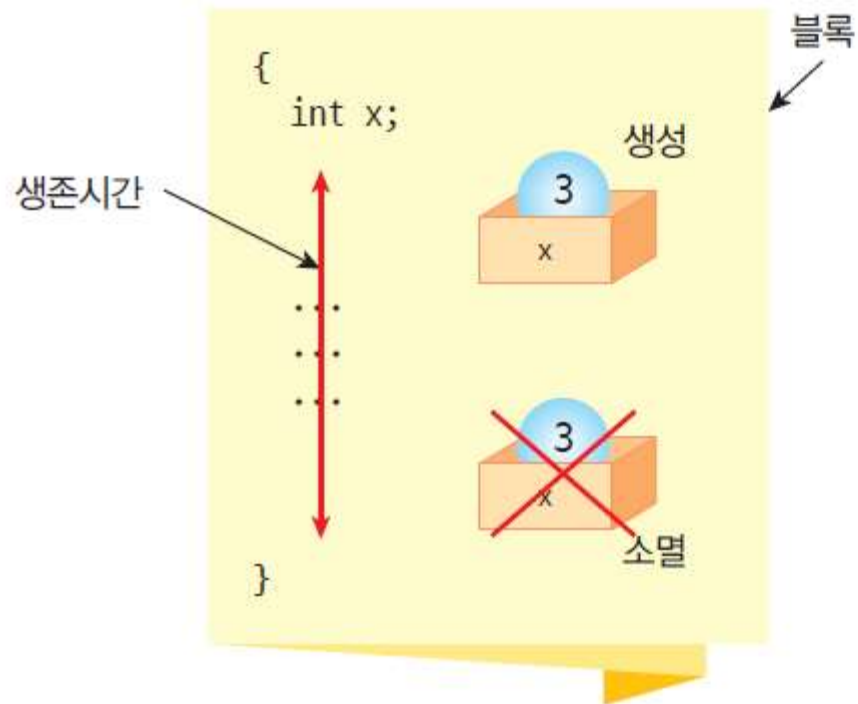






# 지역 변수의 생존 기간

9



지역 변수는 선언된 블록이 끝나면 자동으로 소멸됩니다.





# 지역 변수 예제

local(p377).c

10

```
#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

local1(p378).c

11

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

```
int main(void)
```

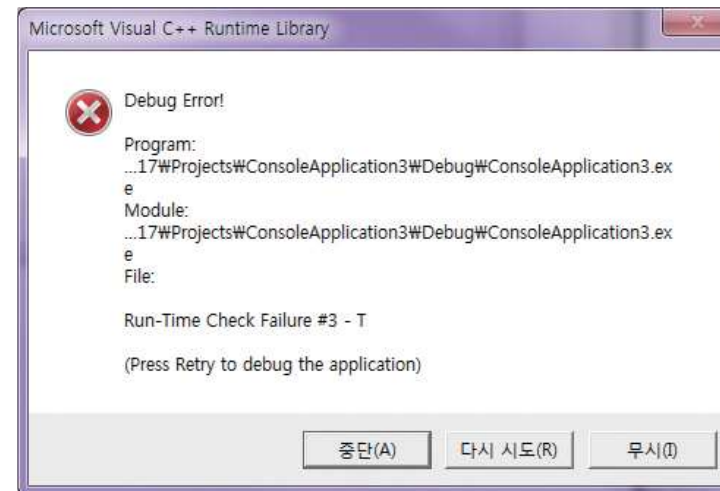
```
{
```

```
    int temp;
```

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

```
}
```

초기화 되지  
않았으므로 쓰레기  
값을 가진다.





# 함수의 매개 변수

12

```
int inc(int counter)
```

```
{
```

```
    counter++;
```

```
    return counter;
```

```
}
```

매개 변수도 일종의  
지역 변수



# 함수의 매개 변수

local2(p379).c

13

```
#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;
```

```
}
```

```
void inc(int counter)
```

```
{
```

```
    counter++;
```

```
}
```

값에 의한 호출  
(call by value)

매개 변수도 일종의  
지역 변수임

함수 호출전 i=10

함수 호출후 i=10



# 전역 변수

14

- 전역 변수(**global variable**)는 함수 외부에서 선언되는 변수이다.
- 전역 변수의 범위는 소스 파일 전체이다.



# 전역 변수의 초기값과 생존 기간

`global.c`

```
#include<stdio.h>
int A;
int B;
int add()
{
    return A + B;
}

int main()
{
    int answer;
    A = 5;
    B = 7;
    answer = add();
    printf("%d + %d = %d\n", A, B, answer);
    return 0;
}
```

전역 변수  
초기값은 0

5 + 7 = 12



# 전역 변수의 초기값

global1(p381).c

16

```
#include <stdio.h>
int counter;

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

counter=0





# 전역 변수의 사용

17

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

int main(void)
{
    for(x=0; x<10; x++)
        sub();
}

void sub()
{
    for(x=0; x<10; x++)
        printf("*");
}
```





# 전역 변수의 사용

18

- 거의 모든 함수에서 사용하는 공통적인 데이터는 전역 변수로 한다.
- 일부의 함수들만 사용하는 데이터는 전역 변수로 하지 말고 함수의 인수로 전달한다.



# 같은 이름의 전역 변수와 지역 변수

global2(p382).c

19

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

```
int sum = 1; // 전역 변수
```

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

```
    int sum = 0; // 지역 변수
```

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

```
    return 0;
```

```
}
```

전역 변수와 지역 변수가 동일한  
이름으로 선언된다.

```
sum = 0
```



# 중간 점검

20

- 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?

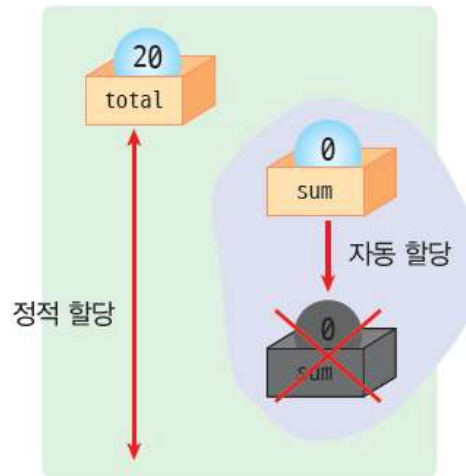




# 생존 기간

21

- 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

22

- 생존 기간을 결정하는 요인
  - ▣ 변수가 선언된 위치
  - ▣ 저장 유형 지정자
  
- 저장 유형 지정자
  - ▣ auto
  - ▣ register
  - ▣ static
  - ▣ extern



# 저장 유형 지정자 auto

23

- 변수를 선언한 위치에서 자동으로 만들어지고 블록을 벗어나게 되며 자동으로 소멸되는 저장 유형을 지정
- 지역 변수는 **auto**가 생략되어도 자동 변수가 된다.

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

전부 자동 변수로서 함수가  
시작되면 생성되고 끝나면  
소멸된다.



# 저장 유형 지정자 static

static.c

24

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

```
void sub() {  
    static int scount = 0;  
    int account = 0;  
    printf("scount = %d\t", scount);  
  
    printf("account = %d\n", account);  
    scount++;  
    account++;  
}
```

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

```
scount = 0 account = 0  
scount = 1 account = 0  
scount = 2 account = 0
```



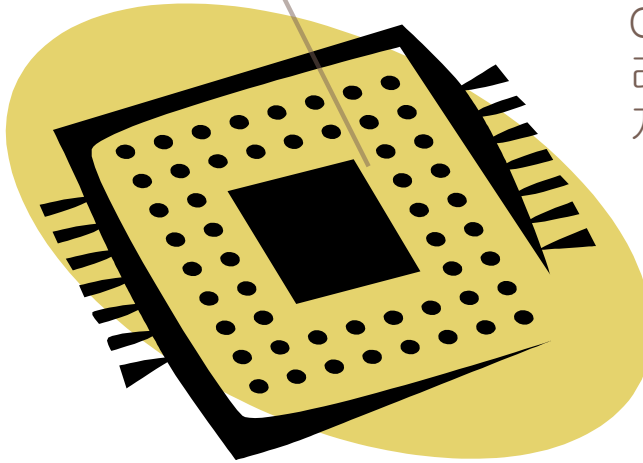


# 저장 유형 지정자 register

25

- 레지스터(register)에 변수를 저장.

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



CPU안의  
레지스터에 변수가  
저장됨



# volatile

26

- **volatile** 지정자는 하드웨어가 수시로 변수의 값을 변경하는 경우에 사용된다

```
volatile int io_port; // 하드웨어와 연결된 변수

void wait(void) {
    io_port = 0;
    while (io_port != 255)
        ;
}
```



# 중간 점검

27

- 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?

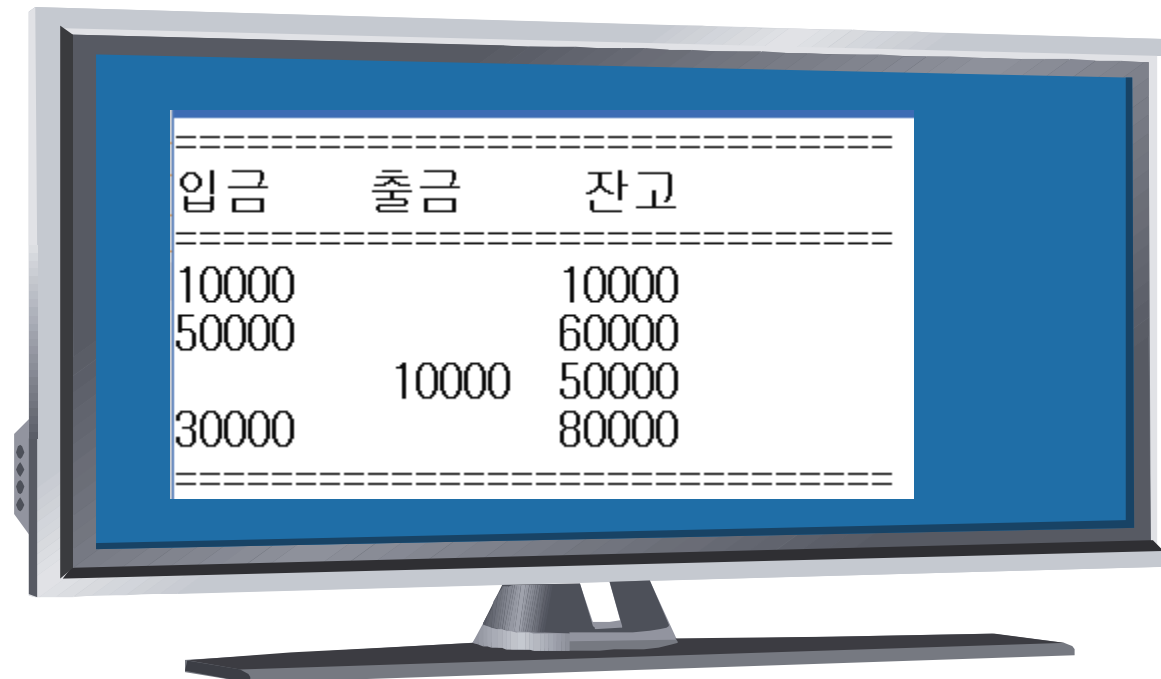




# lab: 은행 계좌 구현하기

28

- 돈만 생기면 저금하는 사람을 가정하자. 이 사람을 위한 함수 `save(int amount)`를 작성하여 보자. 이 함수는 저금할 금액을 나타내는 인수 `amount`만을 받으며 `save(100)`과 같이 호출된다. `save()`는 정적 변수를 사용하여 현재까지 저축된 총액을 기억하고 있으며 한번 호출될 때마다 총 저축액을 다음과 같이 화면에 출력한다.





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

```
// amount가 양수이면 입금이고 음수이면 출금으로 생각한다.
```

```
void save(int amount)
```

```
{
```

```
    static long balance = 0;
```

```
    if( amount >= 0)
```

```
        printf("%d \t\t", amount);
```

```
    else
```

```
        printf("\t %d \t", -amount);
```

```
    balance += amount;
```

```
    printf("%d \n", balance);
```

```
}
```



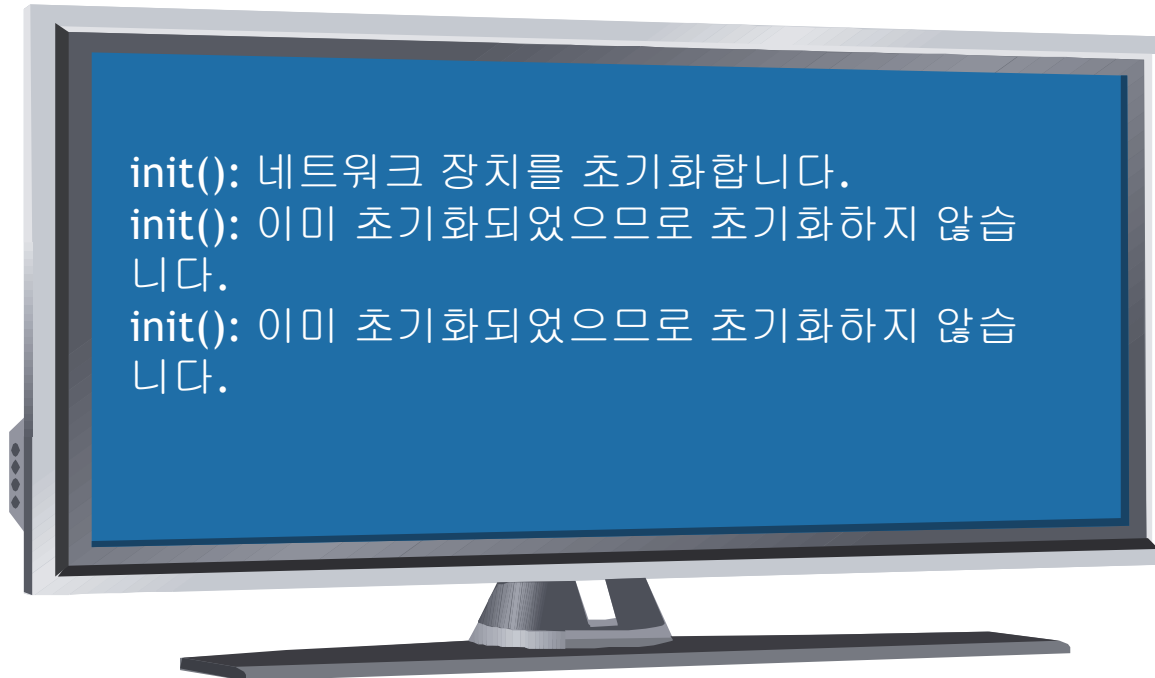
```
int main(void) {  
    printf("=====\n");  
    printf("입금 \t출금\t 잔고\n");  
    printf("=====\n");  
    save(10000);  
    save(50000);  
    save(-10000);  
    save(30000);  
    printf("=====\n");  
    return 0;  
}
```



# lab: 한번만 초기화하기

31

- 정적 변수는 한번만 초기화하고 싶은 경우에도 사용된다





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

void init();

int main(void)
{
    init();
    init();
    init();
    return 0;
}

void init()
{
    static int initd = 0;
    if( initd == 0 ){
        printf("init(): 네트워크 장치를 초기화합니다. \n");
        initd = 1;
    }
    else {
        printf("init(): 이미 초기화되었으므로 초기화하지 않습니다.
\n");
    }
}
```

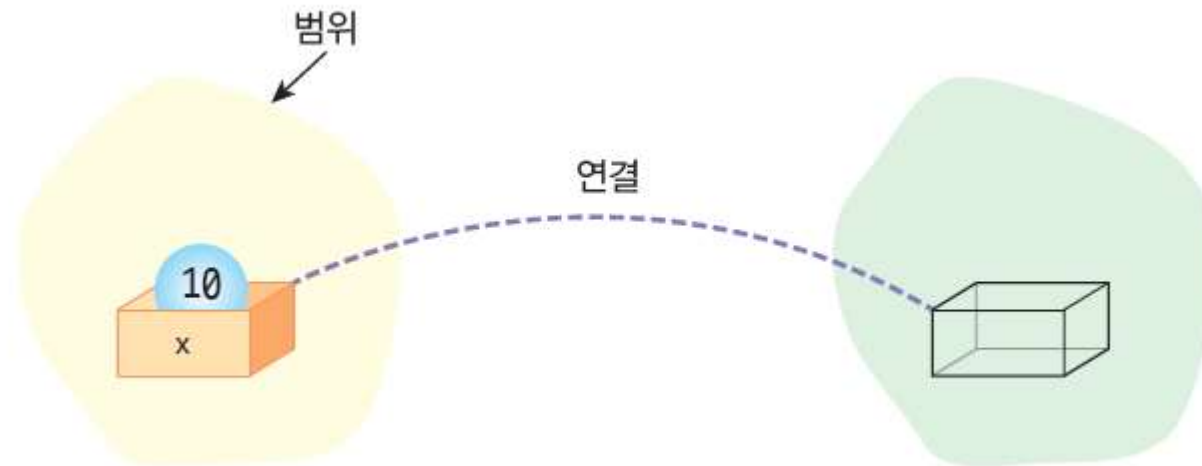




# 연결

33

- *연결(linkage)*: 다른 범위에 속하는 변수들을 서로 연결하는 것
  - ▣ 외부 연결
  - ▣ 내부 연결
  - ▣ 무연결
- 전역 변수만이 연결을 가질 수 있다.

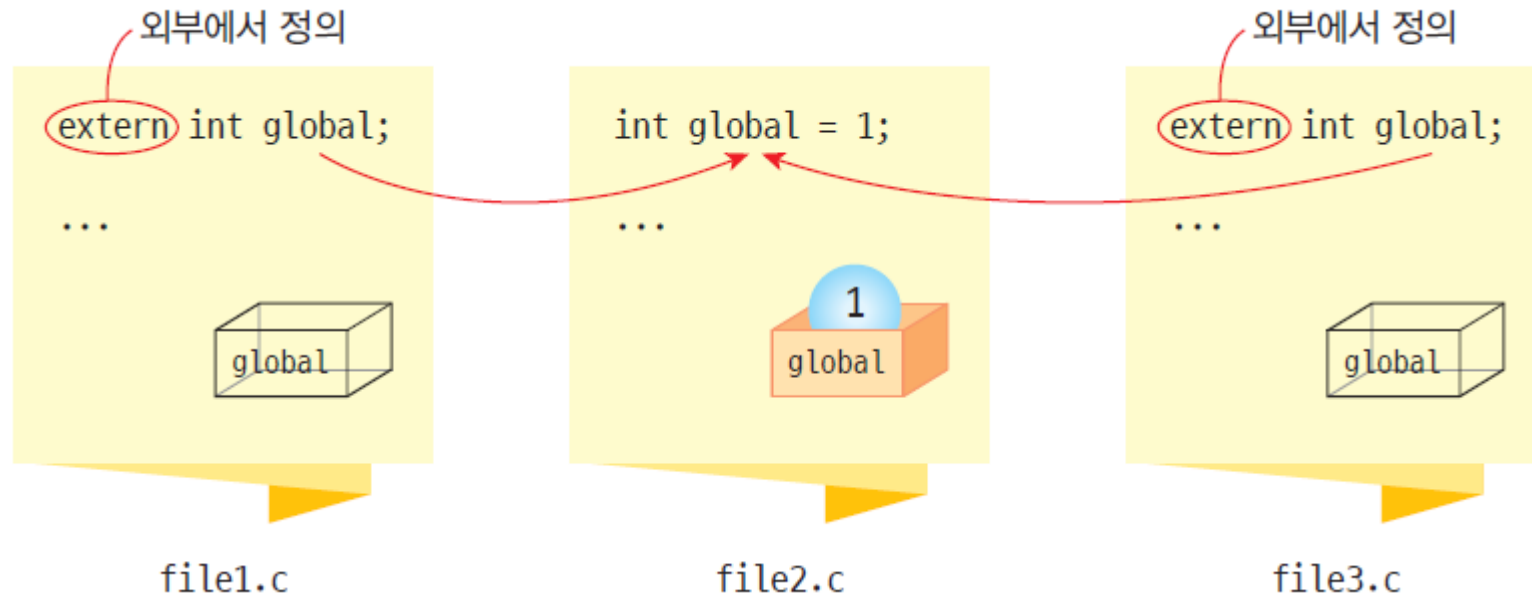




# 외부 연결

34

- 전역 변수를 **extern**을 이용하여서 서로 연결





# 연결 예제

35

*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;
}
```

*linkage2.c*

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

10



# 함수 앞의 static

36

main.c

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

```
extern void f2();
```

```
int main(void)
```

```
{
```

```
    f2();
```

```
    return 0;
```

```
}
```

Static이 붙는 함수는 파일 안에서만 사용할 수 있다.

```
static void f1()
```

```
{
```

```
    printf("f1()이 호출되었습니다.\n");
```

```
}
```

```
void f2()
```

```
{
```

```
    f1();
```

```
    printf("f2()가 호출되었습니다.\n");
```

```
}
```



# 저장 유형 정리

37

- 일반적으로는 *자동 저장 유형* 사용 권장
- 변수의 값이 함수 호출이 끝나도 그 값을 유지하여야 할 필요가 있다면 *지역 정적*
- 만약 많은 함수에서 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

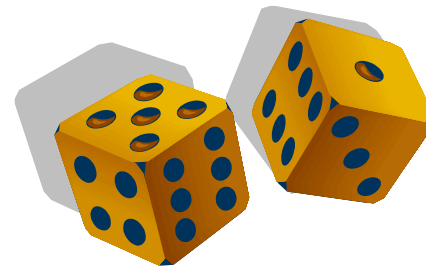


# 예제: 난수 발생기

38

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

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





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

40

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

41

- 매개 변수의 개수가 가변적으로 변할 수 있는 기능

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

호출 때 마다 매개  
변수의 개수가 변경될  
수 있다.



# 가변 매개 변수

stdarg.c

42

합은 10입니다.

```
#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;
    va_start( argptr, num );
    for( ; num > 0; num-- )
        answer += va_arg( argptr, int );
    va_end( argptr );
    return( answer );
}
```

매개 변수의 개수



# 순환(recursion)이란?

43

- 함수는 자기 자신을 호출할 수도 있다. 이것을 순환(recursion)라고 부른다.

$$n! = \begin{cases} 1 & n=0 \\ n * (n-1)! & n \geq 1 \end{cases}$$



# 팩토리얼 구하기

44

- 팩토리얼 프로그래밍:  $(n-1)!$  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

45

## □ 팩토리얼의 호출 순서

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

```
factorial(3)
{
    if( 3 >= 1 ) return 1;
    else return (3 * factorial(3-1) );
}
```

①

```
factorial(2)
{
    if( 2 >= 1 ) return 1;
    else return (2 * factorial(2-1) );
}
```

②

```
factorial(1)
{
    if( 1 >= 1 ) return 1;
    ....
}
```



# 팩토리얼 계산

factorial.c

46

```
// 재귀적인 팩토리얼 함수 계산
#include <stdio.h>

long factorial(int n)
{
    printf("factorial(%d)\n", n);

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

int main(void)
{
    int x = 0;
    long f;

    printf("정수를 입력하시오:");
    scanf("%d", &n);
    printf("%d!은 %d입니다. \n", n, factorial(n));
    return 0;
}
```



# 2진수 형식으로 출력하기

print\_binary.c

47

```
// 2진수 형식으로 출력
#include <stdio.h>

void print_binary(int x);

int main(void)
{
    print_binary(9);
}

void print_binary(int x)
{
    if( x > 0 )
    {
        print_binary(x / 2);           // 재귀 호출
        printf("%d", x % 2);          // 나머지를 출력
    }
}
```



# 최대 공약수 구하기

gcd.c

48

```
// 최대 공약수 구하기
#include <stdio.h>

int gcd(int x, int y);

int main(void)
{
    printf("%d\n", gcd(30, 20));
}

// x는 y보다 커야 한다.
int gcd(int x, int y)
{
    if( y == 0 )
        return x;
    else
        return gcd(y, x % y);
}
```

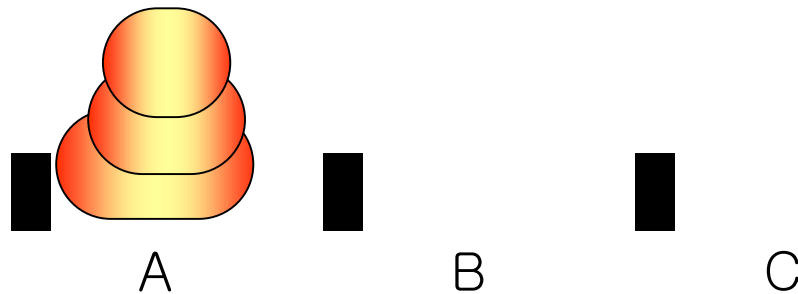




# 하노이 탑 문제 #1

49

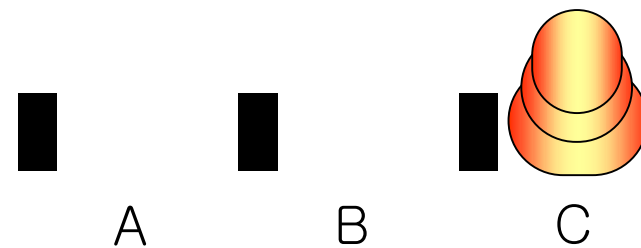
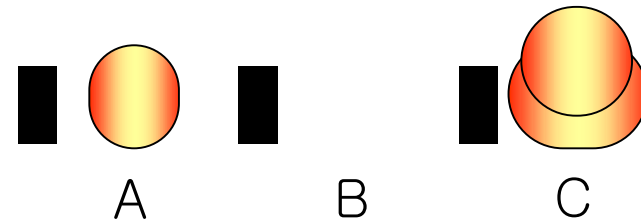
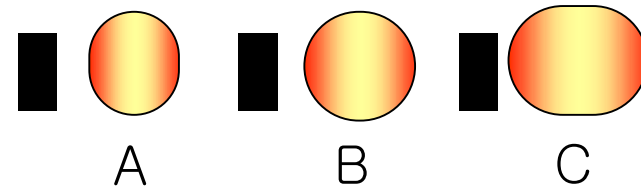
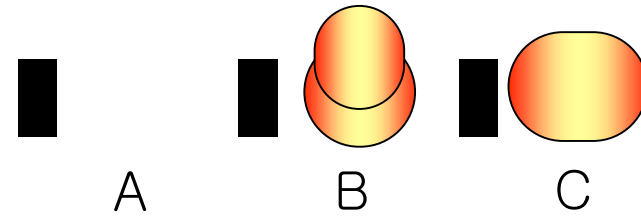
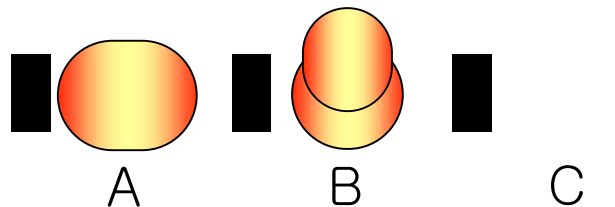
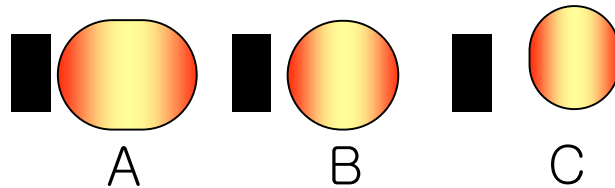
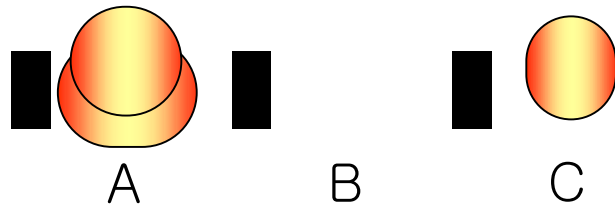
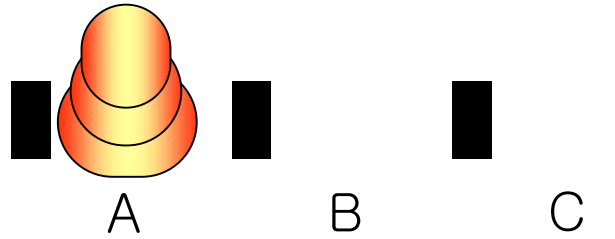
- 문제는 막대 A에 쌓여있는 원판 3개를 막대 C로 옮기는 것이다. 단 다음의 조건을 지켜야 한다.
  - ▣ 한 번에 하나의 원판만 이동할 수 있다
  - ▣ 맨 위에 있는 원판만 이동할 수 있다
  - ▣ 크기가 작은 원판 위에 큰 원판이 쌓일 수 없다.
  - ▣ 중간막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.





# 3개의 원판인 경우의 해답

50

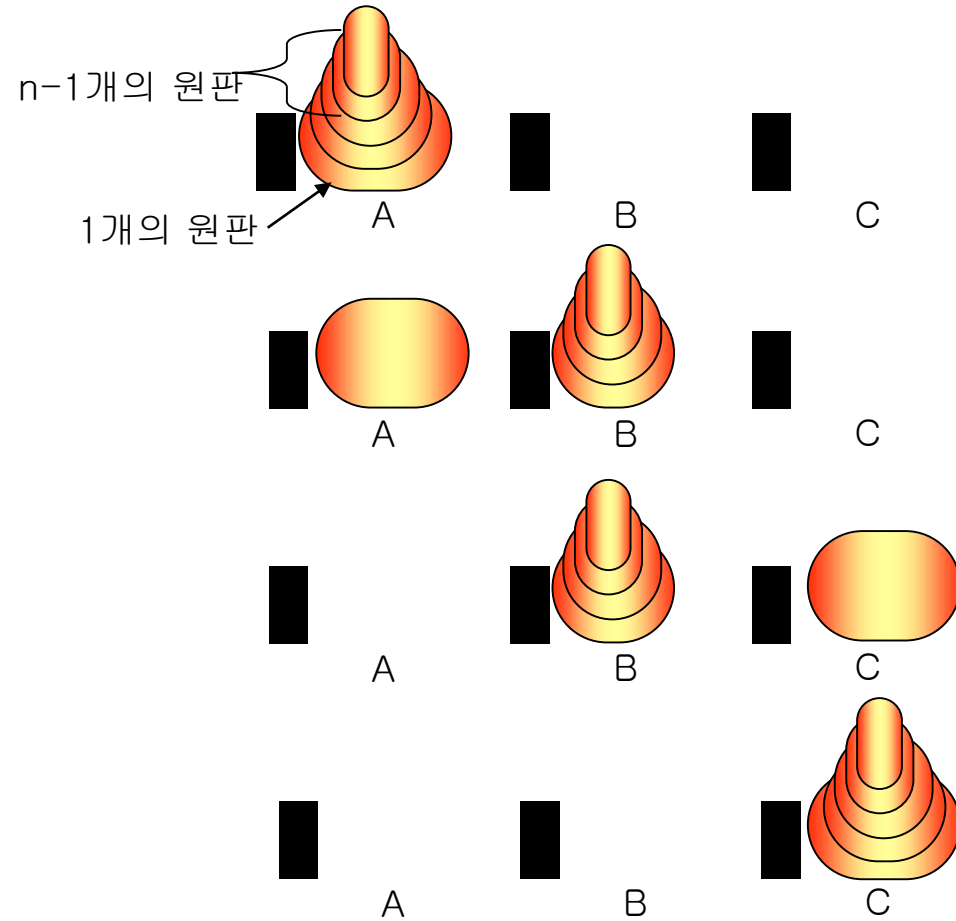




# n개의 원판인 경우

51

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





# 하노이탑 알고리즘

hanoi\_tower.c

52

// 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

hanoi\_tower.c

53

```
#include <stdio.h>

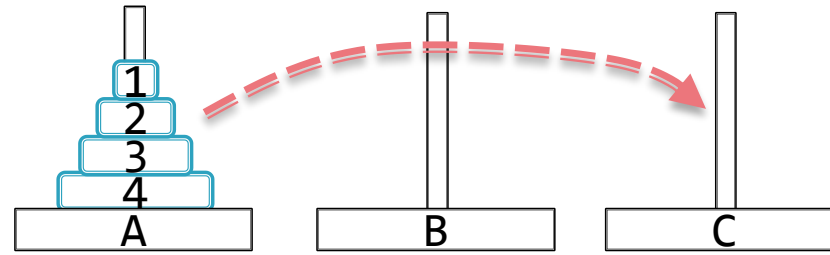
void hanoi_tower(int n, char from, char tmp, char to)
{
    if( n==1 )
        printf("원판 1을 %c 에서 %c으로 옮긴다.\n",from,to);
    else {
        hanoi_tower(n-1, from, to, tmp);
        printf("원판 %d을 %c에서 %c으로 옮긴다.\n",n, from, to);
        hanoi_tower(n-1, tmp, from, to);
    }
}

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

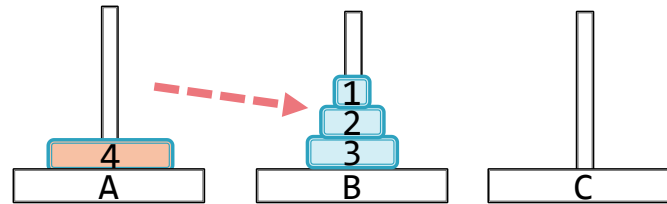
Microsoft Visual Studio 디버그 콘솔

```
원판 1을 A 에서 B으로 옮긴다.
원판 2을 A 에서 C으로 옮긴다.
원판 1을 B 에서 C으로 옮긴다.
원판 3을 A 에서 B으로 옮긴다.
원판 1을 C 에서 A으로 옮긴다.
원판 2을 C 에서 B으로 옮긴다.
원판 1을 A 에서 B으로 옮긴다.
원판 4을 A 에서 C으로 옮긴다.
원판 1을 B 에서 C으로 옮긴다.
원판 2을 B 에서 A으로 옮긴다.
원판 1을 C 에서 A으로 옮긴다.
원판 3을 B 에서 C으로 옮긴다.
원판 1을 A 에서 B으로 옮긴다.
원판 2을 A 에서 C으로 옮긴다.
원판 1을 B 에서 C으로 옮긴다.
```

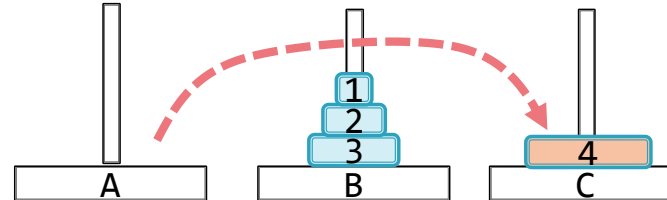
```
hanoi_tower(int n, char from, char tmp, char to)
```



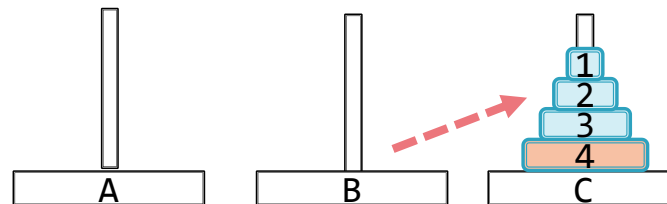
```
hanoi_tower(4, 'A', 'B', 'C');
```



```
hanoi_tower(3, 'A', 'C', 'B');
```



```
printf("원판 %d을 %c에서 %c으로 옮긴다.\n", n, from, to); //n=4, from=A, to=C  
원판 4을 A에서 C으로 옮긴다.
```



```
hanoi_tower(3, 'B', 'A', 'C');
```



# Q & A

55

