

CHAPTER

# 09

# 함수와 변수

- 변수가 선언된 위치에 따라 변수의 범위, 생존 시간, 연결 등이 어떻게 달라지는지를 이해한다.
- 저장 유형 지정자에 따라 변수의 속성이 어떻게 변경되는지를 학습한다.
- 자기 자신을 호출하는 순환 호출의 개념과 응용 예를 살펴본다.

# Contents

3

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

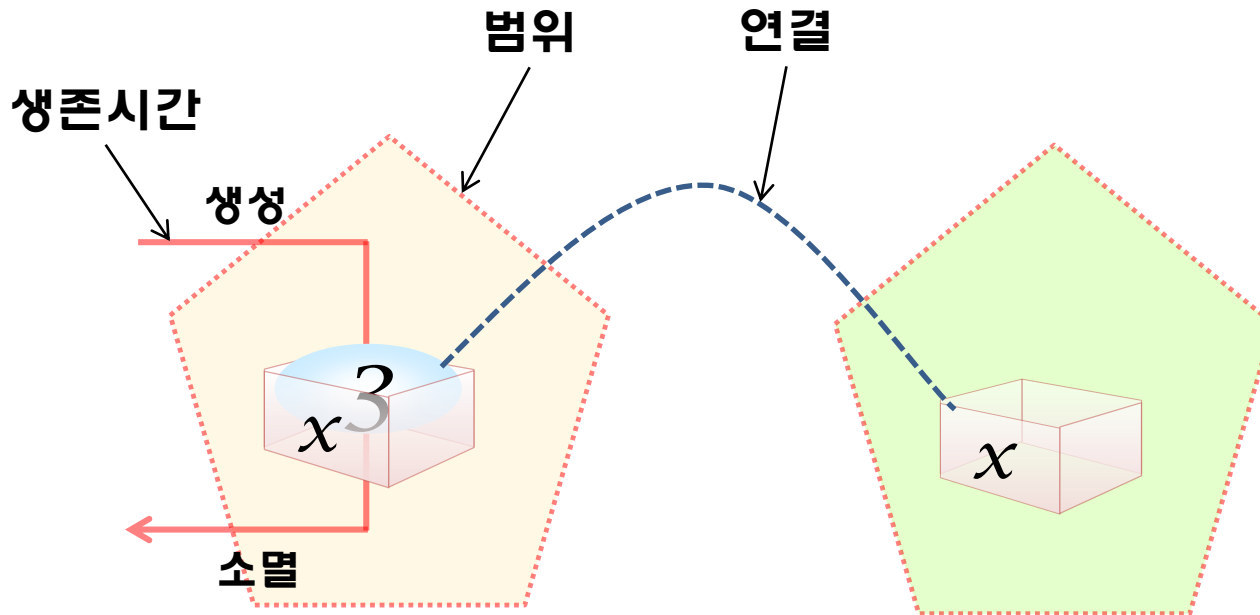
9.8

순환 호출

# 변수의 속성

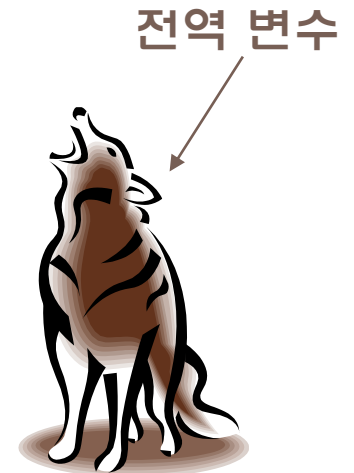
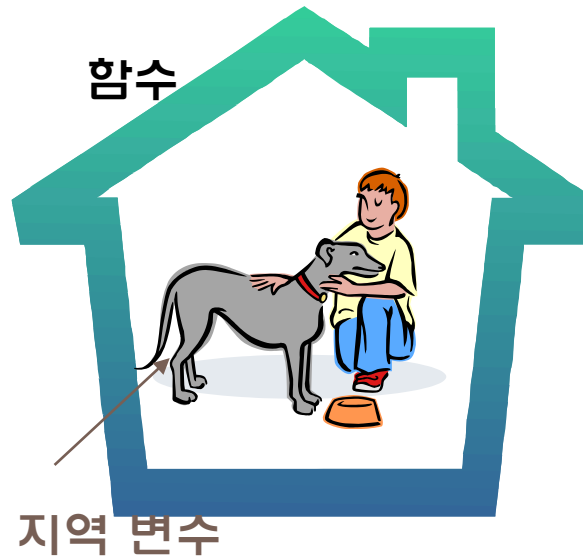
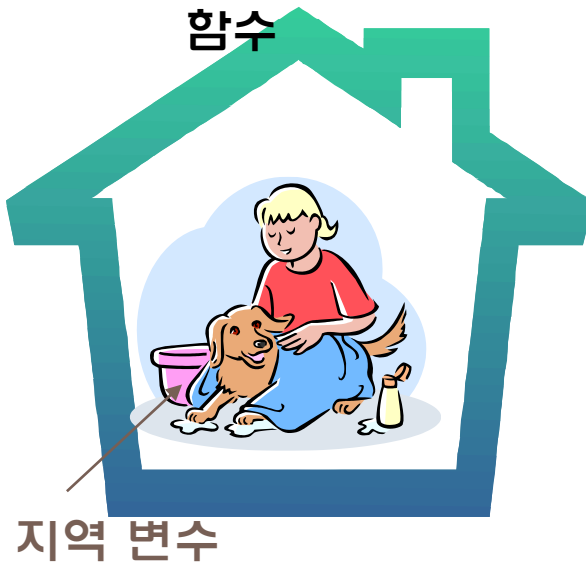
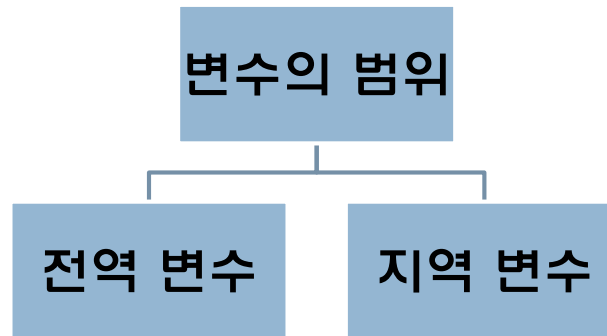
4

- 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

5



# 지역 변수와 전역 변수

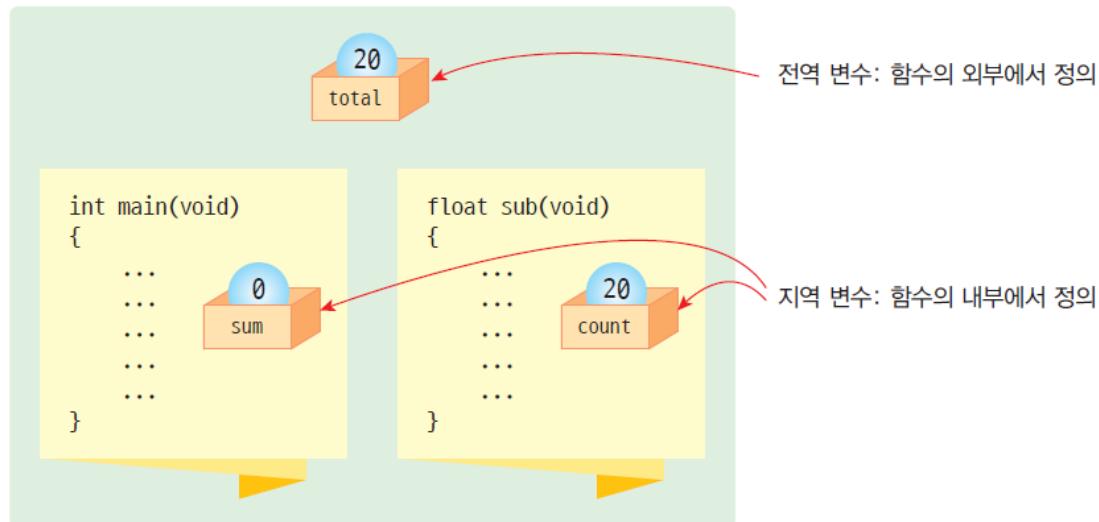
6

## □ 지역변수

- 함수 또는 블록 안에 정의되는 변수
- 해당 블록이나 함수 안에서만 사용가능

## □ 전역변수

- 함수의 외부에서 정의되는 변수
- 소스 파일의 어느 곳에서도 사용 가능



# Contents

7

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

9.8

순환 호출

# 지역 변수

8

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


```
int sub(void)
{
    int x = 0;
    int y = 0;
    while(flag != 0){
        int y;
        ...
    }
    y = 0; // 오류!!
    ...
}
```

지역 변수 x가 사용가능한 범위

지역 변수 y가 사용가능한 범위

y가 선언된 블록을 벗어나서  
사용하였으므로 오류!

지역 변수는 선언된  
블록을 떠나면 안됩니다.





# 지역 변수 선언 위치

9

- 최신 버전의 C에서는 **블록 안의 어떤 위치에서도** 선언 가능!!

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

```
...
```

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

```
...
```

```
}
```

블록의 중간에서도 얼마든지  
지역 변수를 선언할 수 있다.

# 지역 변수의 범위

10

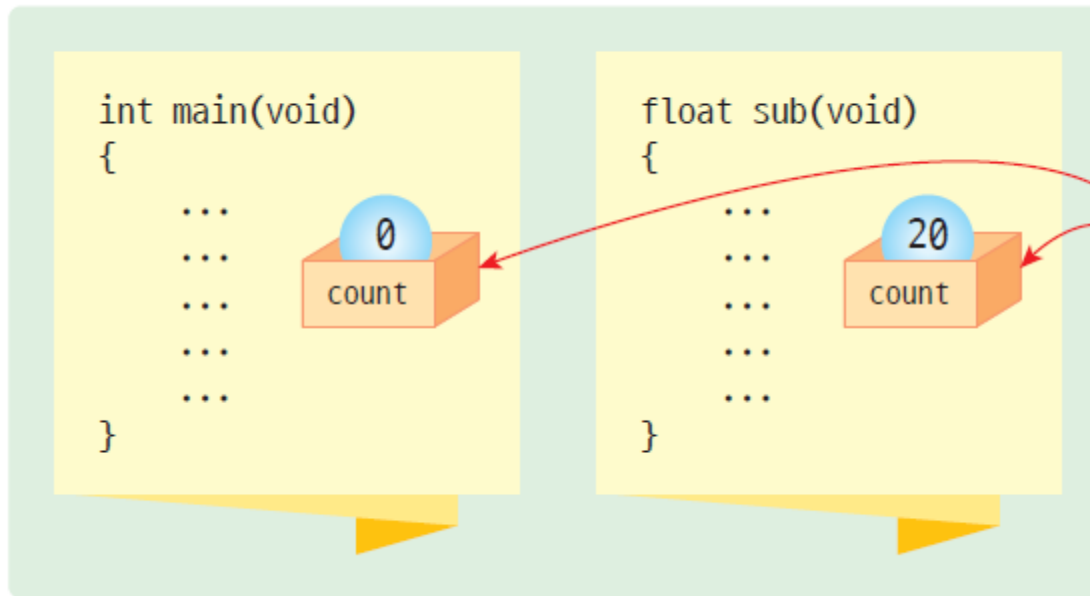
```
void sub1(void)
{
    {
        int y;
        ...
    }
    y = 4;
}
```

지역 변수는 선언된 블록을 떠나면 안됩니다.



# 이름이 같은 지역 변수

11

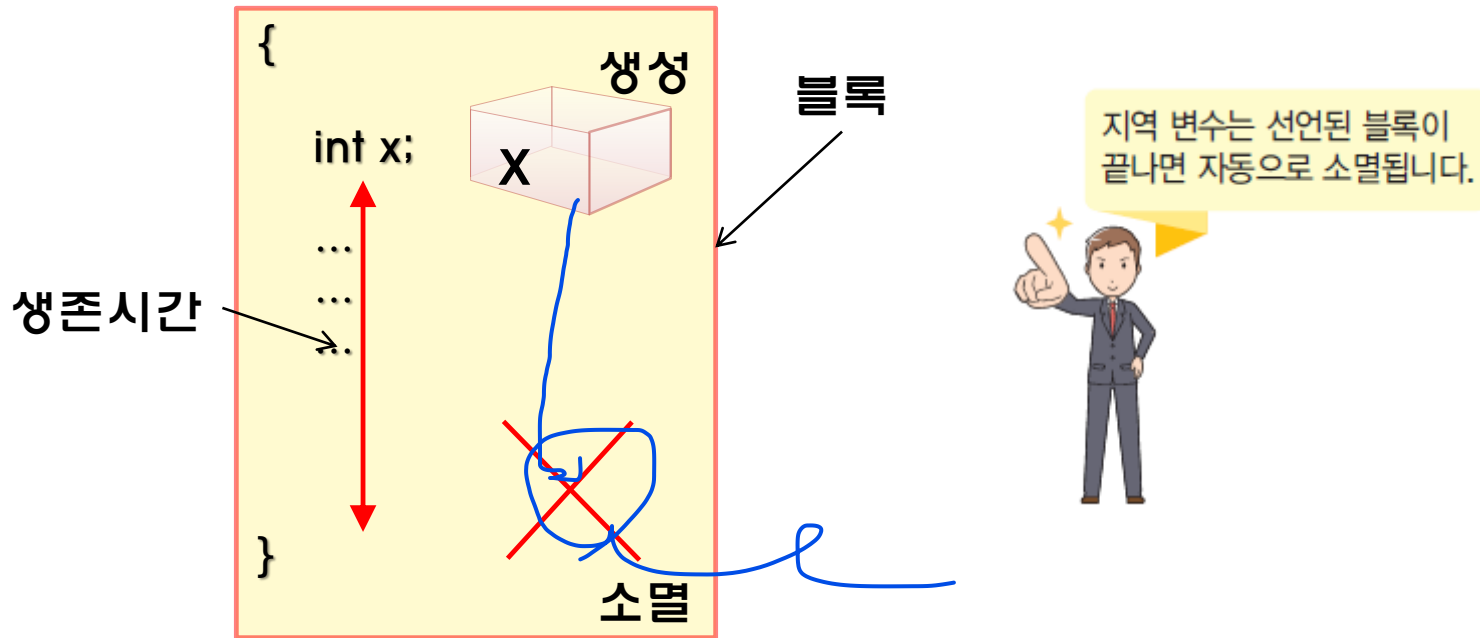


블록만 다르면  
이름은 같아도 됩니다.



# 지역 변수의 생존 기간

12



# 지역 변수 예제

13

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

```
int main(void)
```

```
{
```

```
    int i;
```

```
    for(i = 0; i < 5; i++)
```

```
    {
```

```
        int temp = 1;
```

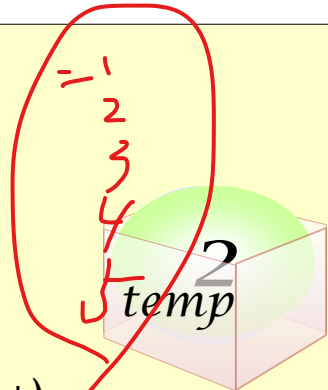
```
        printf("temp = %d\n", temp);
```

```
        temp++;
```

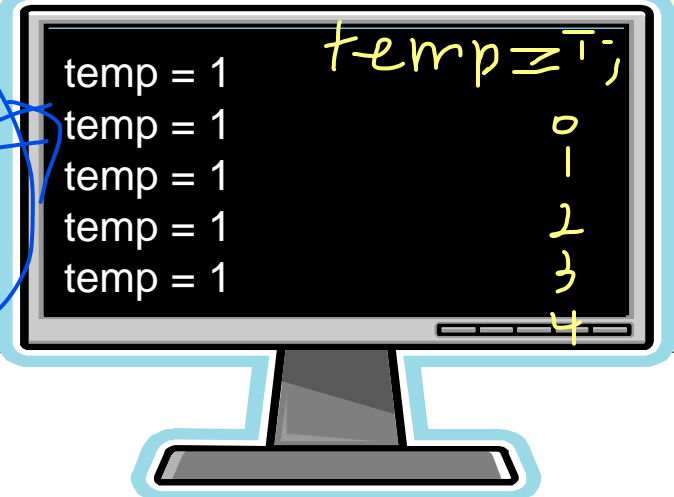
```
    }
```

```
    return 0;
```

```
}
```



블록이 시작할 때 마다  
생성되어 초기화된다.

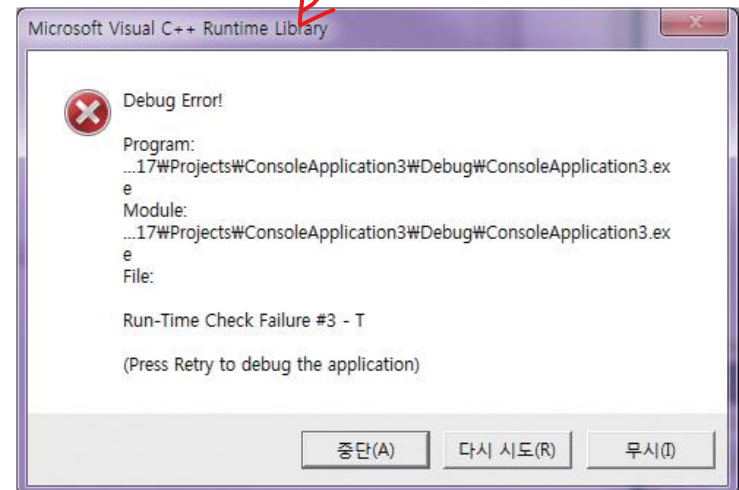


# 지역 변수의 초기값

14

```
#include <stdio.h>
int main(void)
{
    int temp;
    printf("temp = %d\n", temp);
}
```

초기화 되지  
않았으므로 쓰레기  
값을 가진다.



# 함수의 매개 변수

15

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

```
{
```

```
    counter++;
```

```
    return counter;
```

```
}
```

매개 변수도 일종의  
지역 변수

# 함수의 매개 변수

16

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

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

10

```
int main(void)
```

```
{
```

```
    int i;
```

```
    i = 10;
```

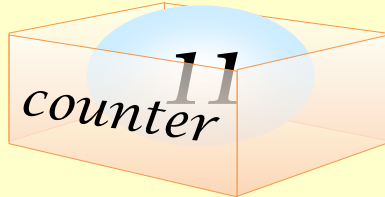
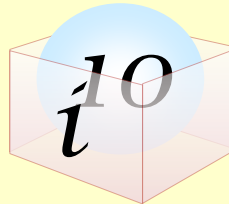
```
    printf("함수 호출전 i=%d\n", i);
```

```
    inc(i);
```

```
    printf("함수 호출후 i=%d\n", i);
```

```
    return 0;
```

```
}
```



매개 변수도  
일종의 지역변수

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

```
{
```

```
    counter++;
```

```
    return counter;
```

```
}
```

함수 호출전 i=10

함수 호출후 i=10 → 11



# Contents

17

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

9.8

순환 호출

# 전역 변수

18

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

```
int x = 123;
```

```
void sub1()
```

```
{
```

```
    x = 456;
```

```
}
```

```
void sub2()
```

```
{
```

```
    x = 789;
```

```
}
```

전역 변수



# 전역 변수의 초기값과 생존 기간

19

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

```
int counter; // 전역 변수
```

\*전역변수의 초기값 : 0

\*생존기간 : 프로그램 시작부터 종료

```
void set_counter(int i)
```

```
{
```

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

```
}
```

```
int main(void)
```

```
{
```

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

```
    counter = 100;    // 직접 사용 가능
```

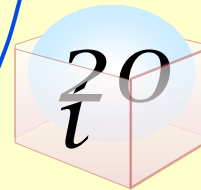
```
    printf("counter=%d\n", counter);
```

```
    set_counter(20);
```

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

```
    return 0;
```

```
}
```



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

# 전역 변수의 사용

20

출력은  
어떻게  
될까요?

// 전역 변수를 사용하여 프로그램이 복잡해지는 경우

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

```
void f(void);
```

```
int i;
```

```
int main(void)
```

```
{
```

```
    for(i = 0; i < 3; i++) (X)
```

```
    {  
        f();
```

```
    }  
    return 0;
```

```
}
```

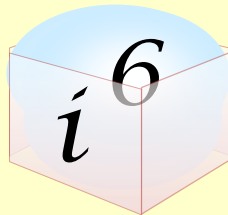
```
void f(void)
```

```
{
```

```
    for(i = 0; i < 5; i++) (O)
```

```
        printf("#\n");
```

```
}
```



# 전역 변수의 사용

21

- 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

22

// 동일한 이름의 전역 변수와 지역 변수

#include <stdio.h>

→ int sum = 1; // 전역 변수

int main(void)  
{

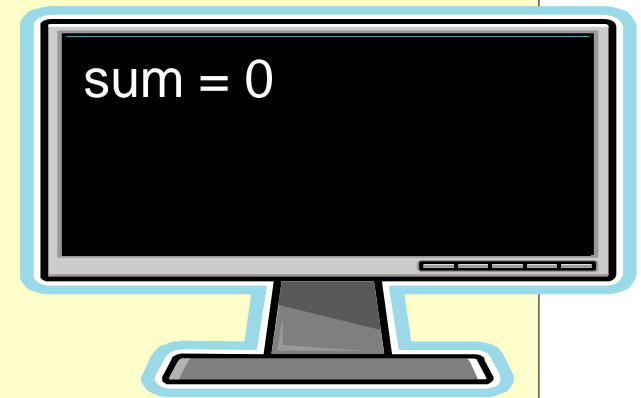
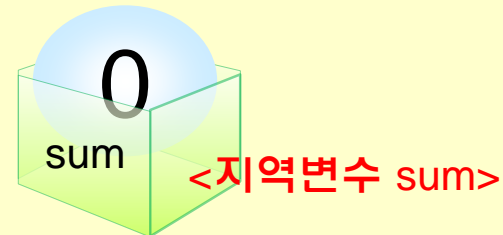
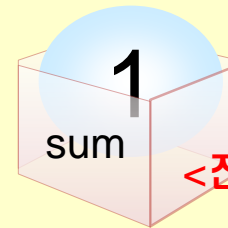
지역 변수가  
전역변수를 가린다.

→ int sum = 0; // 지역 변수

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

return 0;

}



# Contents

23

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

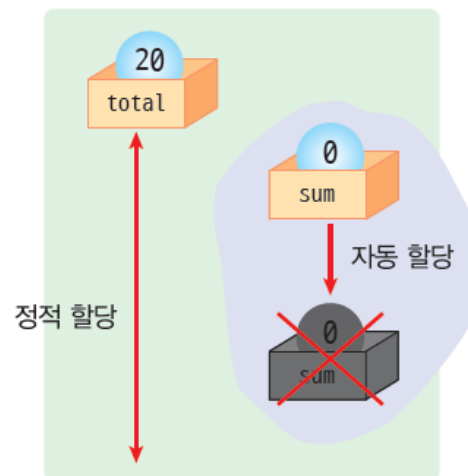
9.8

순환 호출

# 생존 시간

24

- 정적 할당(static allocation):
  - ▣ 프로그램 실행 시간 동안 계속 유지
- 자동 할당(automatic allocation):
  - ▣ 블록에 들어갈때 생성
  - ▣ 블록에서 나올때 소멸



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





# 생존 시간

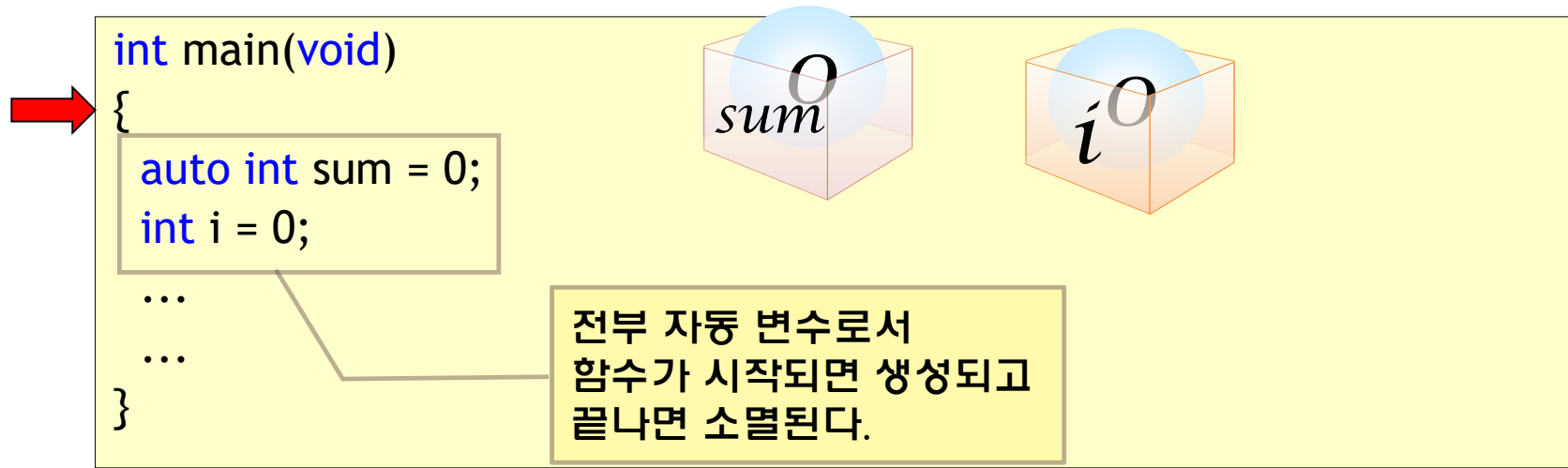
25

- 생존 시간을 결정하는 요인
  - ▣ 변수가 선언된 위치
  - ▣ 저장 유형 지정자
  
- 저장 유형 지정자
  - ▣ ~~auto~~
  - ▣ ~~register~~
  - ▣ static     ≡ 정역
  - ▣ extern

# auto 지정자

26

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



# static 지정자

27

```
#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;
```

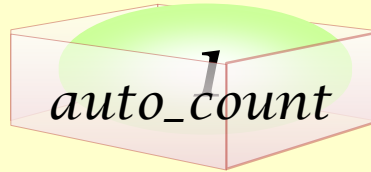
```
auto_count++;
```

```
static_count++;
```

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

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

```
}
```



자동 지역 변수

정적 지역 변수로써  
static을 붙이면 지역변수가  
정적변수로 된다.

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

8월 7일

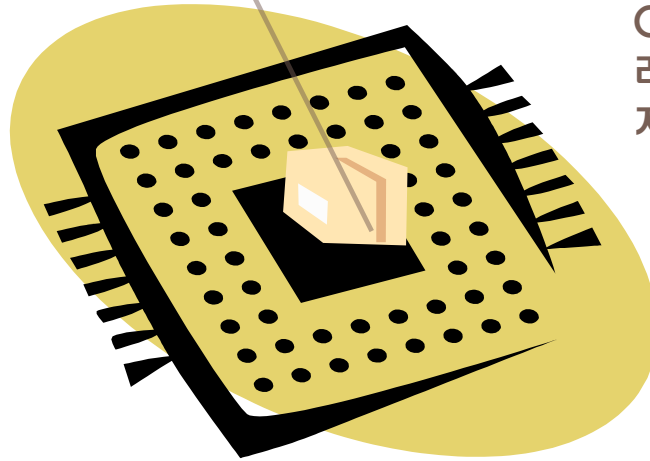
1 → 2 → 3

# register 지정자

28

- 레지스터(register)에 변수를 저장.

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



CPU안의  
레지스터에 변수가  
저장됨

# volatile 지정자

29

- 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)
        ;
}
```

# extern 지정자

30

```
#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. 없어도 된다.
```

```
x = 10;
```

```
y = 20;
```

```
z = 30;
```

```
return 0;
```

```
}
```

```
int y;
```

// 전역 변수

*extern2.c*

```
int z;
```

컴파일러에게 변수가 다른  
곳에서 선언되었음을 알린다.

# Lab: 은행 계좌 구현하기

31

- 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.

# 실행 결과

32



A computer monitor with a light blue frame and a grey base. The screen is black and displays a table of financial data in white text. The table has three columns: '입금' (Deposit), '출금' (Withdrawal), and '잔고' (Balance). The data is as follows:

입금	출금	잔고
10000		10000
50000		60000
	10000	50000
30000		80000



# 은행 계좌 구현하기

33

```
#include <stdio.h>

void save(int amount);

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;
}
```

# 은행 계좌 구현하기

34

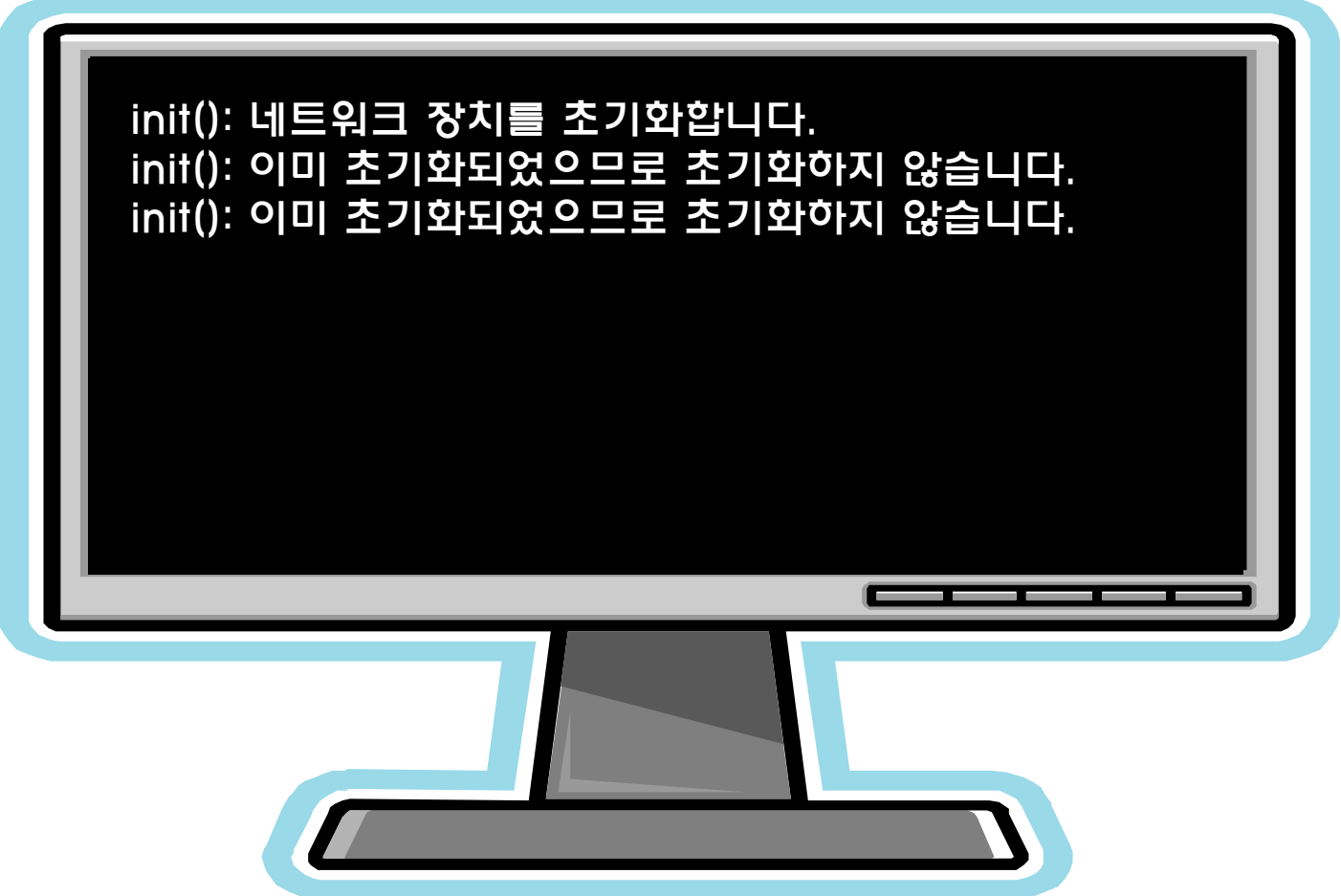
```
// amount가 양수이면 입금이고 음수이면 출금으로 생각한다.
void save(int amount)
{
    static long balance = 0;
    int
    if( amount >= 0)
        printf("%d \t\t", amount);
    else
        printf("\t %d \t", -amount);

    balance += amount;
    printf("%d \n", balance);
}
```

# Lab: 한번만 초기화하기

35

- 정적 변수는 한번만 초기화하고 싶은 경우에도 사용된다



```
init(): 네트워크 장치를 초기화합니다.  
init(): 이미 초기화되었으므로 초기화하지 않습니다.  
init(): 이미 초기화되었으므로 초기화하지 않습니다.
```



```
#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");
    }
}
```

# 실습: 로그인 횟수 제한

37

- 로그인 시에 제한된 횟수만큼 틀리면 로그인 시도를 막는 프로그램을 작성하여 보자.



# 실행 결과

38



# 알고리즘

39

- while(1) ← 무한루프
  - ▣ 사용자로부터 아이디를 입력받는다.
  - ▣ 사용자로부터 패스워드를 입력받는다.
  - ▣ 만약 로그인 시도 횟수가 일정 한도를 넘었으면 프로그램을 종료한다.
  - ▣ 아이디와 패스워드가 일치하면 로그인 성공 메시지를 출력한다.
  - ▣ 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("회수 초과\n");
        exit(1);
    }
    if( id == super_id && password == super_password )
        return SUCCESS;
    else
        return FAIL;
}
```

정적 지역 변수

# 도전문제

42

- 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.

# Contents

43

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

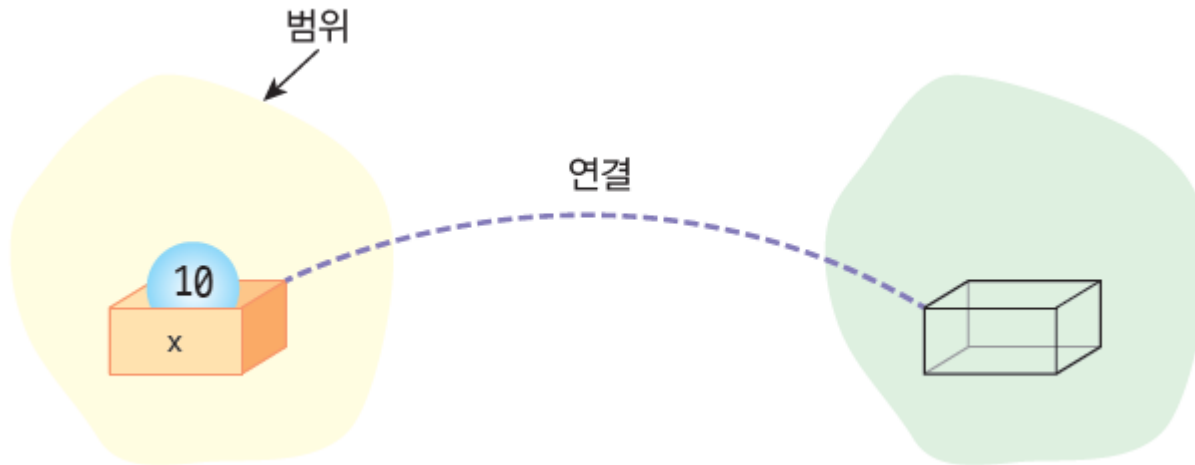
9.8

순환 호출

# 연결

44

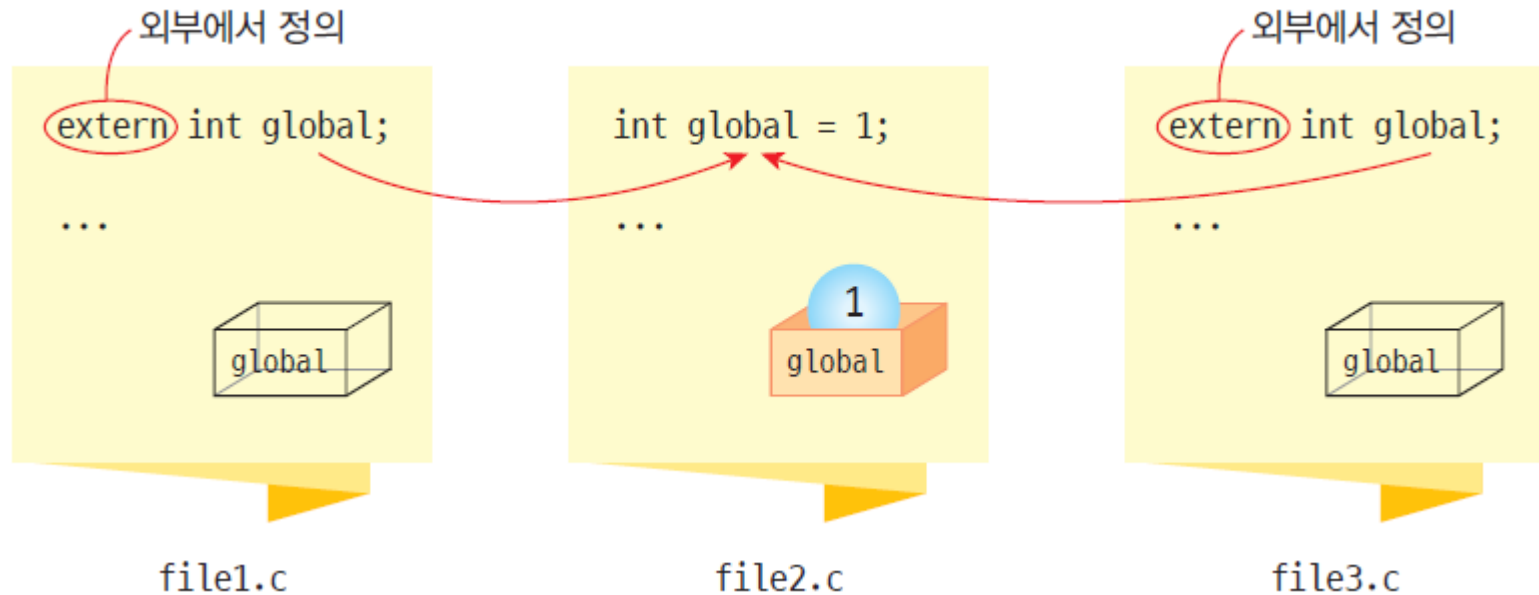
- 연결(linkage): 다른 범위에 속하는 변수들을 서로 연결하는 것
  - 외부 연결
  - 내부 연결
  - 무연결
- 전역 변수만이 연결을 가질 수 있다.



# 외부 연결

45

- 전역 변수를 extern을 이용하여서 서로 연결



# 내부 연결

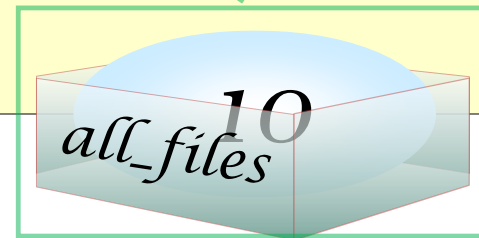
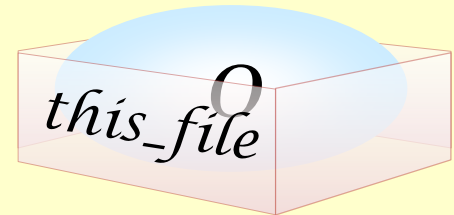
46

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;
}
```



# 함수 앞의 static

47

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");
```

```
}
```

# Contents

48

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

9.8

순환 호출



# 저장 유형 정리

49

- 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

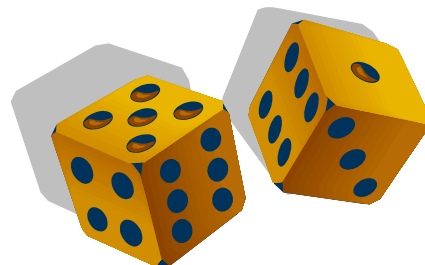
저장 유형	키워드	정의되는 위치	범위	생존 시간
자동	auto	함수 내부	지역	임시
레지스터	register	함수 내부	지역	임시
정적 지역	static	함수 내부	지역	영구
전역	없음	함수 외부	모든 소스 파일	영구
정적 전역	static	함수 외부	하나의 소스 파일	영구
외부 참조	extern	함수 외부	모든 소스 파일	영구

# Lab: 난수 발생기

50

- 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

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



# 난수 발생기

51

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;
}
```

# 난수 발생기

52

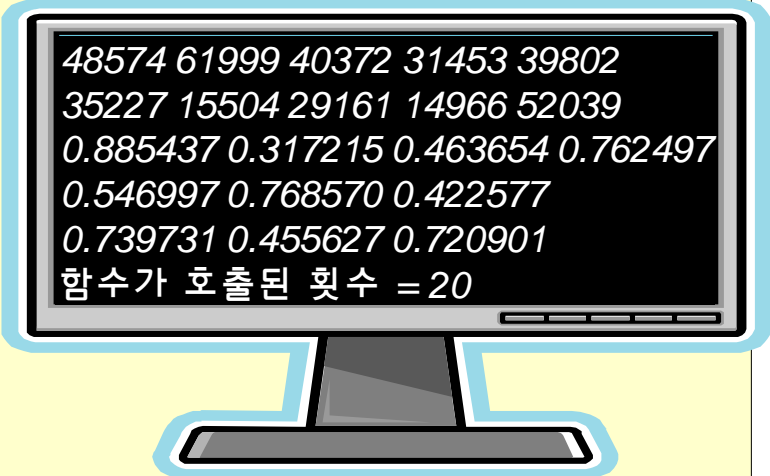
random.c

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

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;
}
```



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

# Contents

53

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

9.8

순환 호출


# 가변 매개 변수

54

- 매개 변수의 개수가 가변적으로 변할 수 있는 기능

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

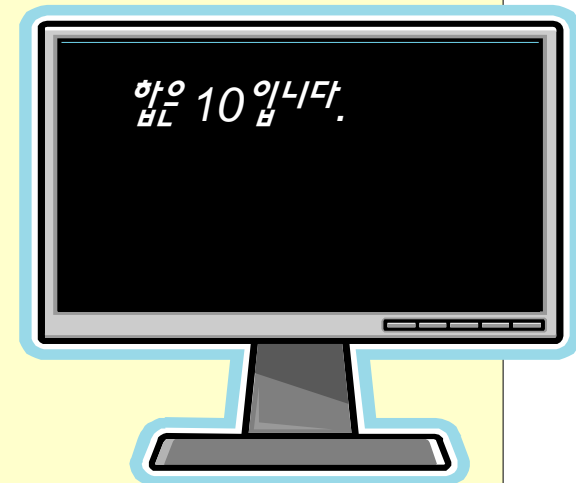
호출 때 마다 매개  
변수의 개수가 변경될  
수 있다.



# 가변 매개 변수

55

```
#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 );
}
```



# Contents

56

9.1

변수의 속성

9.2

지역 변수

9.3

전역 변수

9.4

생존 시간

9.5

연결

9.6

어떤 저장 유형을 사용해야 하는가?

9.7

가변 매개 변수 함수

9.8

순환 호출



# 순환(recursion)이란?

57

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

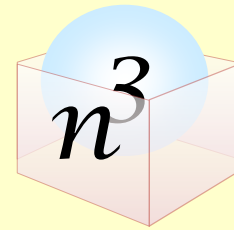
$$n! = \begin{cases} 1 & n = 1 \\ n * (n-1)! & n \geq 2 \end{cases}$$

# 팩토리얼 구하기

58

- 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) );
}
```

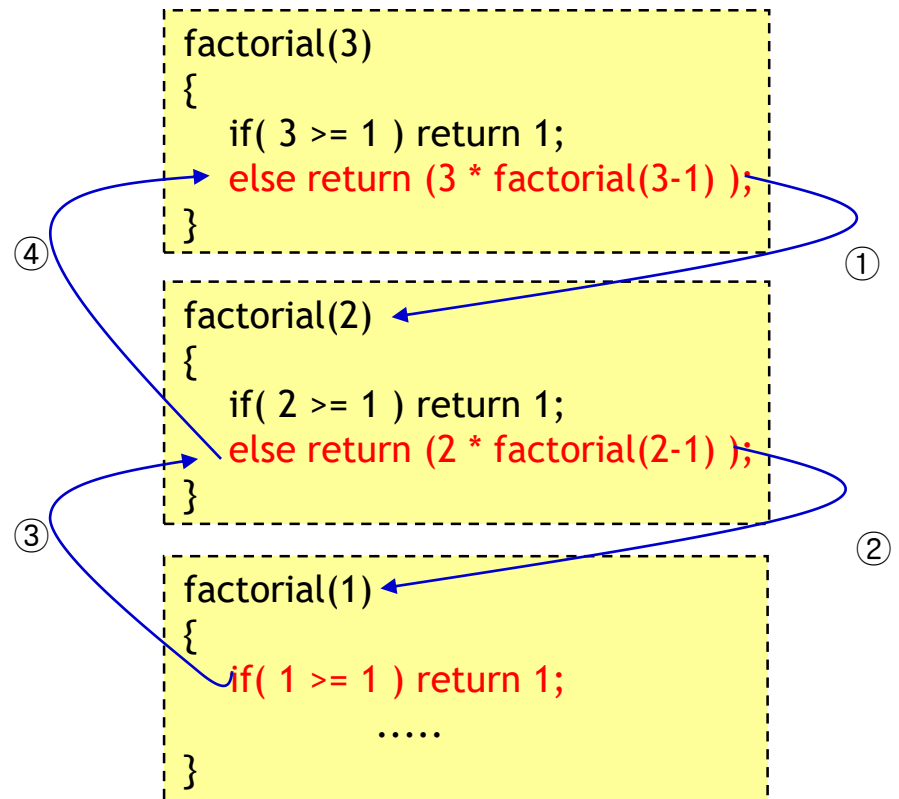


# 팩토리얼 구하기

59

## □ 팩토리얼의 호출 순서

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



# 순환 알고리즘의 구조

60

- 순환 알고리즘은 다음과 같은 부분들을 포함한다.
  - ▣ 순환 호출을 하는 부분
  - ▣ 순환 호출을 멈추는 부분

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

순환을 멈추는 부분

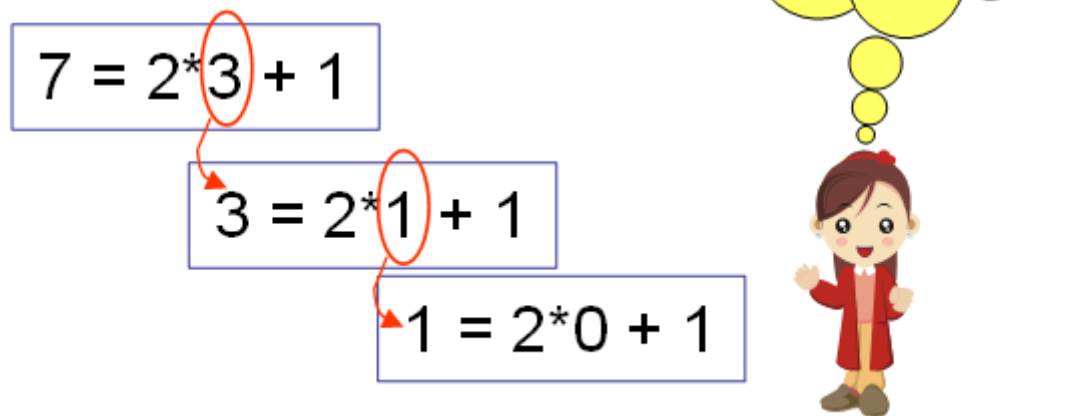
순환호출을 하는 부분

The diagram illustrates the structure of a recursive algorithm using the factorial function. The code is shown with two parts highlighted: a pink oval for the base case 'if( n == 1 ) return 1' and a yellow oval for the recursive case 'else return n \* factorial(n-1);'. A red arrow points from the text '순환을 멈추는 부분' (part that stops the loop) to the pink oval. A black arrow points from the text '순환호출을 하는 부분' (part that makes the recursive call) to the yellow oval. Additionally, a black curved arrow originates from the 'else' branch and points back to the start of the function, indicating the recursive call.

# 이진수 출력하기

61

- 정수를 이진수로 출력하는 프로그램 작성
- 순환 알고리즘으로 가능



# 이진수 출력하기

62

```
// 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);
```

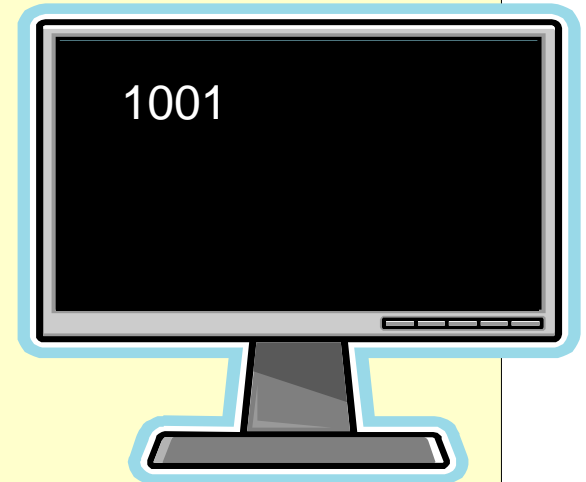
```
        printf("%d", x % 2);
```

```
    }
```

```
}
```

```
// 순환 호출
```

```
// 나머지를 출력
```



# 최대 공약수 구하기

63

```
// 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);
}
```

# 피보나치 수열의 계산

64

- 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

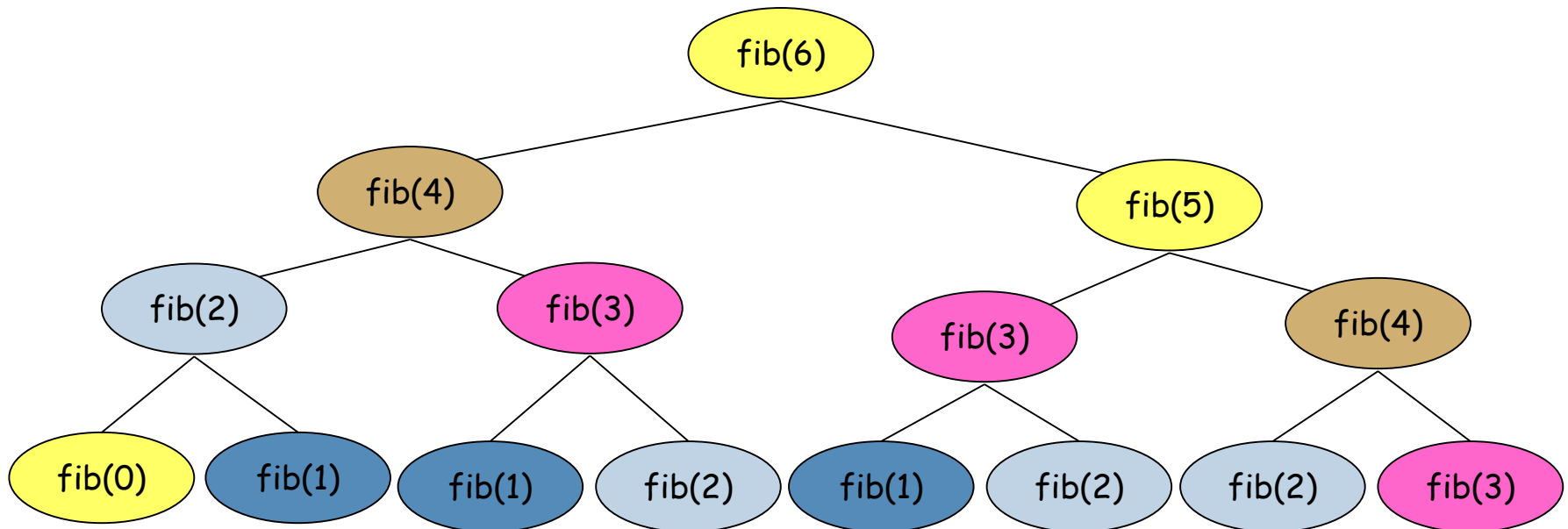
```
int fib(int n)
{
    if( n==0 ) return 0;
    if( n==1 ) return 1;
    return (fib(n-1) + fib(n-2));
}
```



# 피보나치 수열의 계산

65

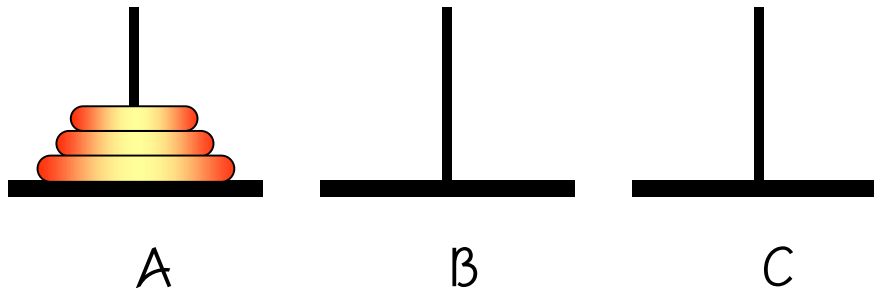
- 순환 호출을 사용했을 경우의 비효율성
  - ▣ 같은 항이 중복해서 계산됨
  - ▣ 예를 들어  $\text{fib}(6)$ 을 호출하게 되면  $\text{fib}(3)$ 이 3번이나 중복되어서 계산됨
  - ▣ 이러한 현상은  $n$ 이 커지면 더 심해짐



# 하노이 탑 문제

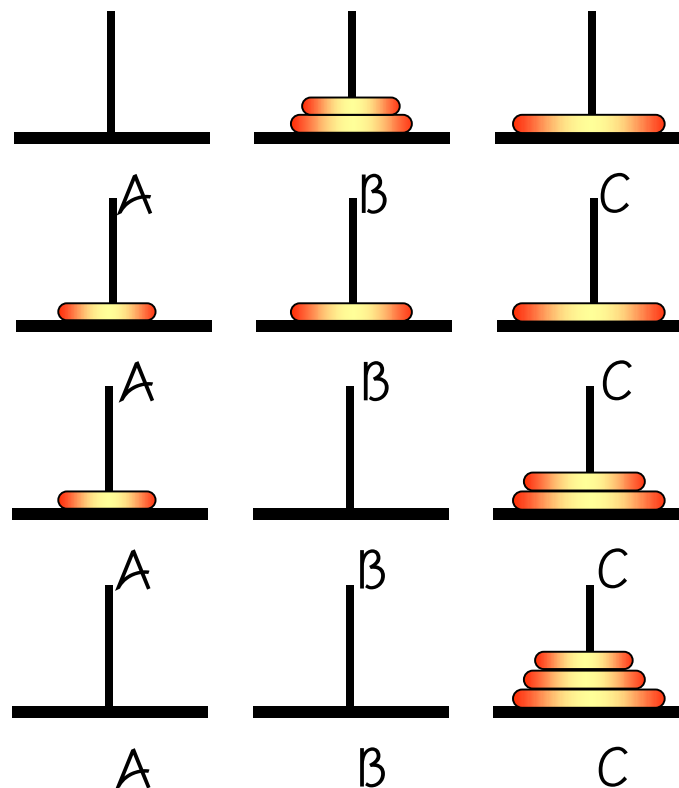
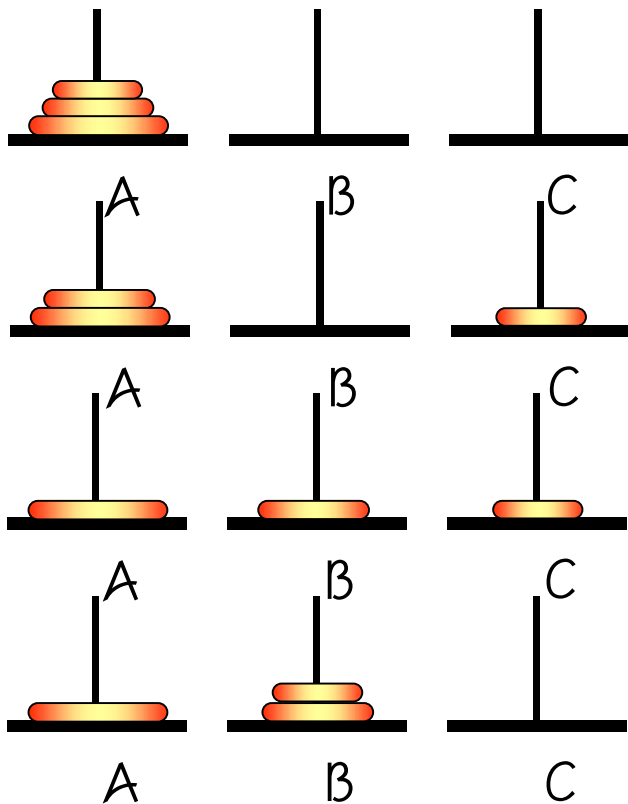
66

- 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 원판위에 큰 원판이 쌓일 수 없다.
  - ▣ 중간막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.



# 3개의 원판인 경우의 해답

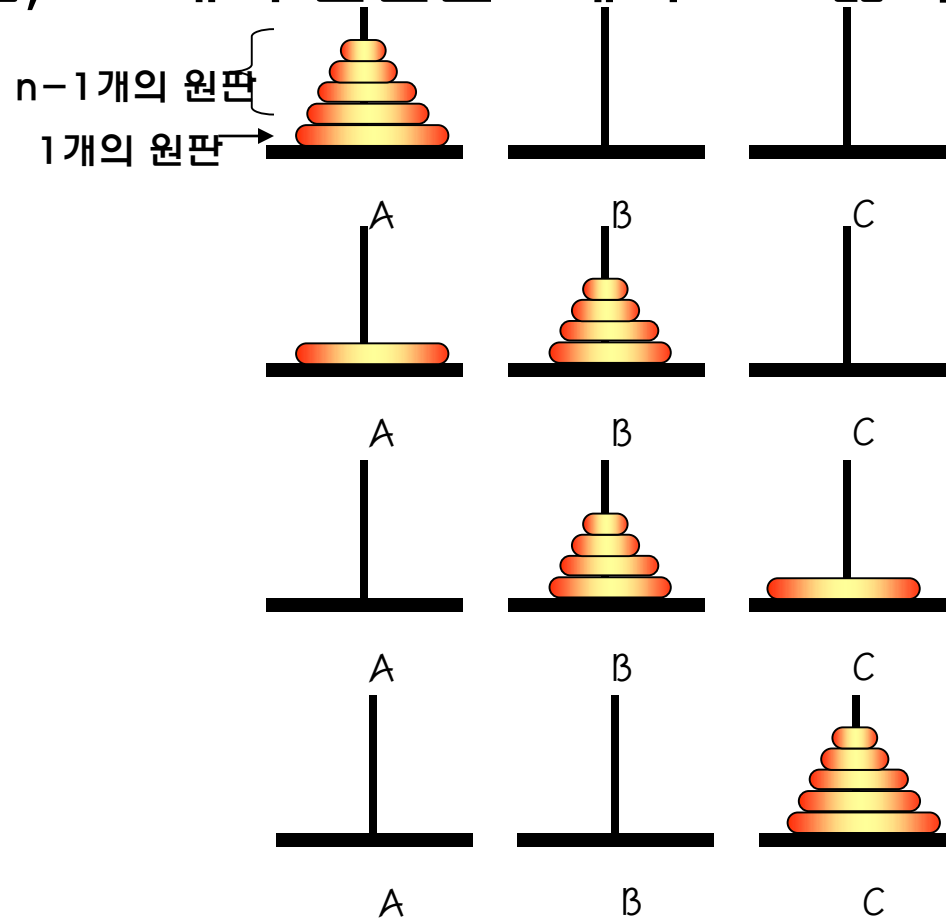
67



# n개의 원판인 경우

68

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



# 하노이탑 알고리즘

69

// 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

70

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

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