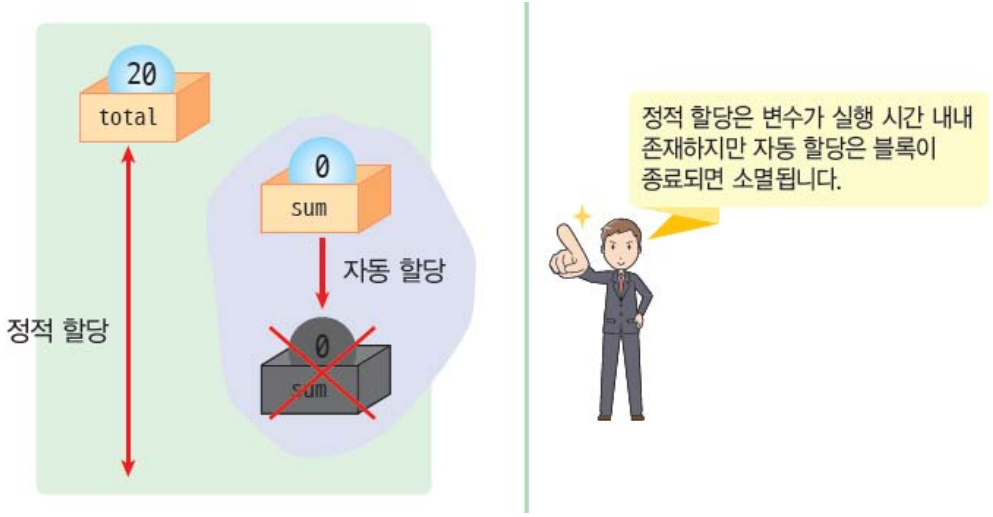
제9장 함수와 변수

생존 기간

- 정적 할당(static allocation):
  - 프로그램 실행 시간 동안 계속 유지

- 자동 할당(automatic allocation):
  - 블록에 들어갈 때 생성
  - 블록에서 나올 때 소멸



생존 기간

- 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 되면 자동으로 소멸되는 저장 유형을 지정
- 지역 변수는 auto가 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

```
#include <stdio.h>

void sub() {
    static int scount = 0;
    int acount = 0;
    printf("scount = %d\t", scount);

    printf("acount = %d\n", acount);
    scount++;
    acount++;

}

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

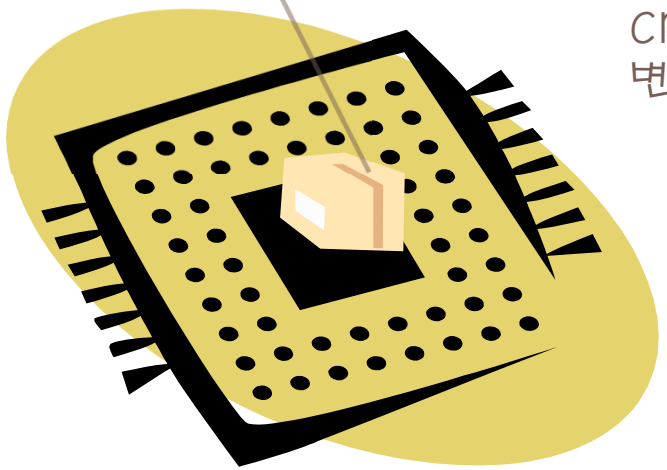
```
scount = 0 acount = 0
scount = 1 acount = 0
scount = 2 acount = 0
```

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

volatile

- 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

- 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

- 돈만 생기면 저금하는 사람을 가정하자. 이 사람을 위한 함수 save(int amount)를 작성하여 보자. 이 함수는 저금할 금액을 나타내는 인수 amount만을 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.

소스

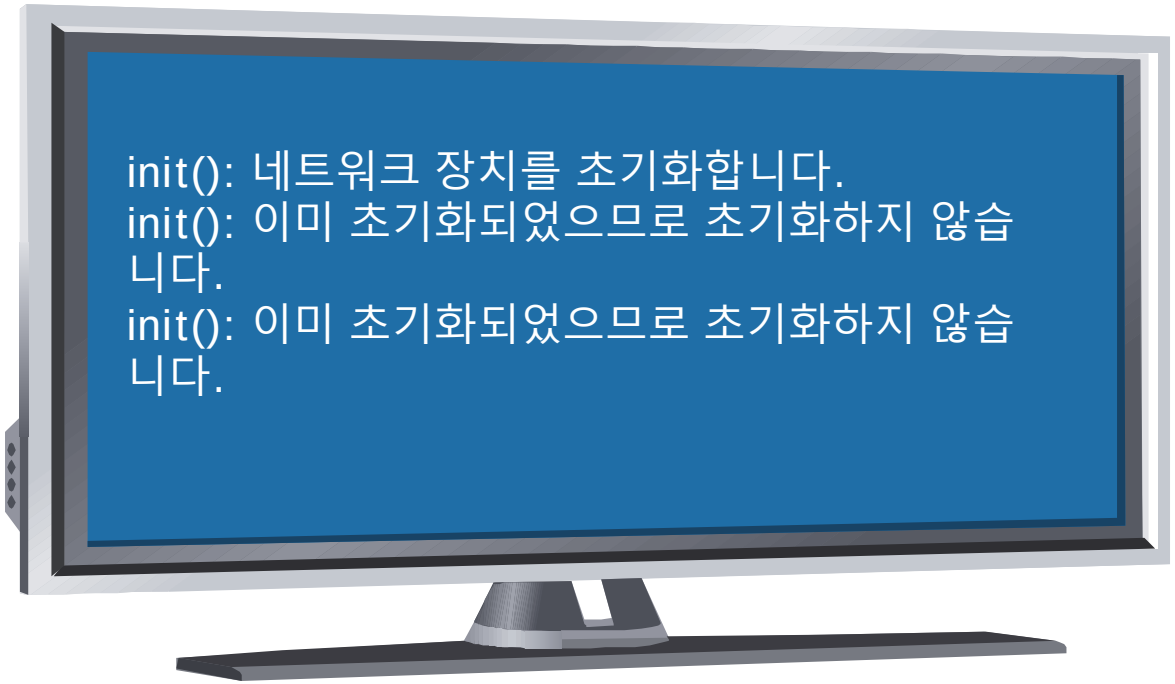
```
#include <stdio.h>

// 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);
}
```

소스

```
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

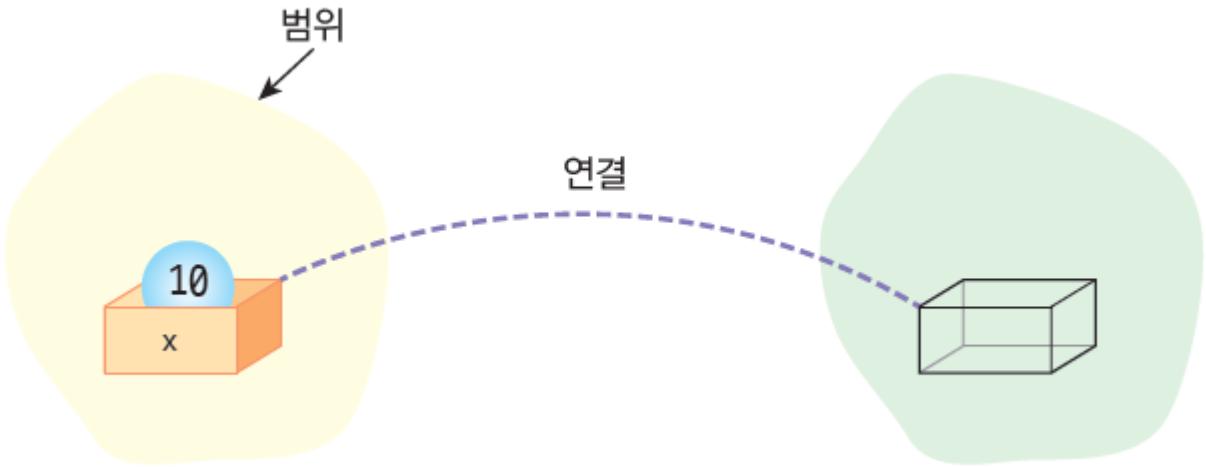
- 정적 변수는 한번만 초기화하고 싶은 경우에도 사용된다



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

void init();

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

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

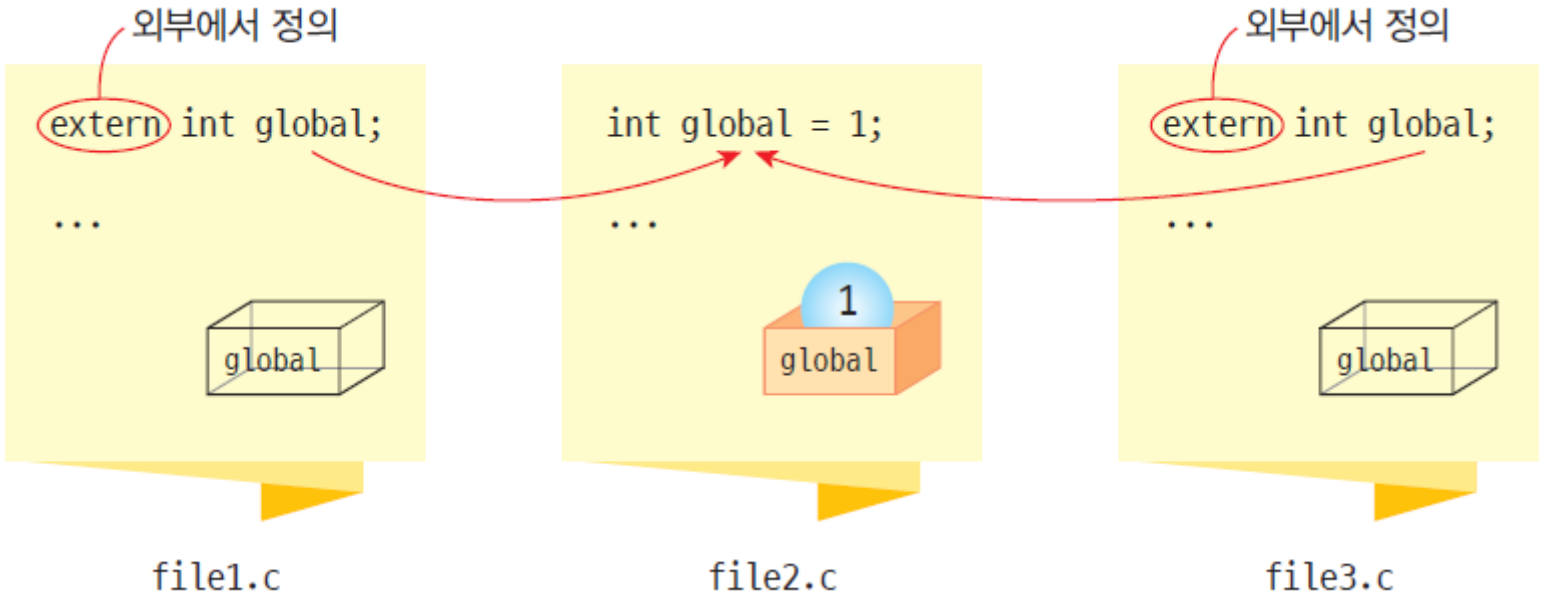
연결

- *연결(linkage):* 다른 범위에 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
```

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");
}
```

저장 유형 정리

- 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 변수(전역)*, 만약 많은 소스 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	staic	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

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

합은 10입니다.

매개 변수의 개수

순환(recursion)이란?

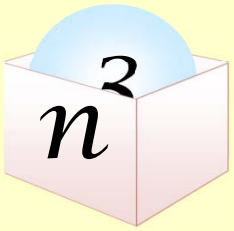
- 함수는 자기 자신을 호출할 수도 있다. 이것을 순환(recursion)라고 부른다.

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

팩토리얼 구하기

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

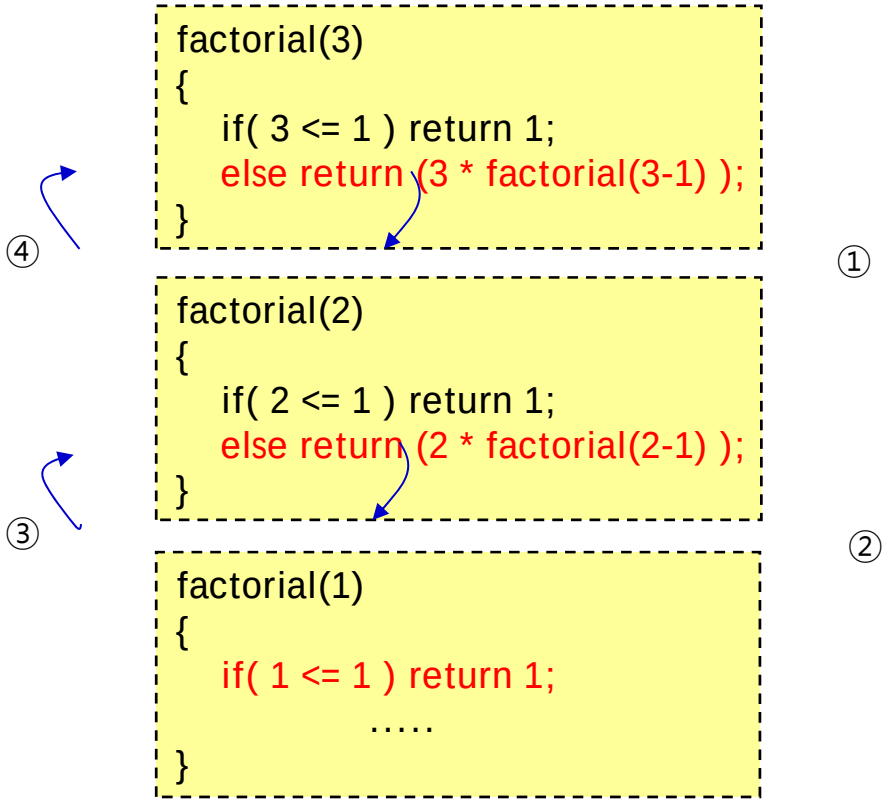
```
int factorial(int n)
{
    if( n <= 1 ) return(1);
    else return (n * factorial(n-1) );
}
```



팩토리얼 구하기

- 팩토리얼의 호출 순서

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



팩토리얼 계산

```
// 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

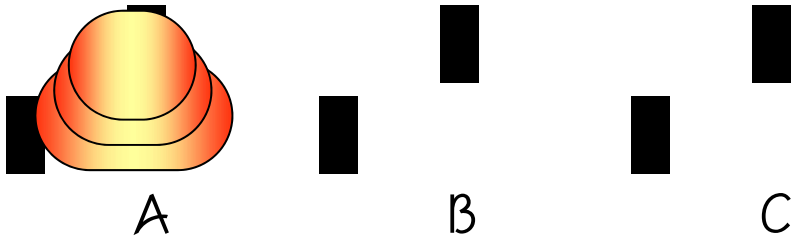
```
// 2진수 형식으로 출력
#include <stdio.h>

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

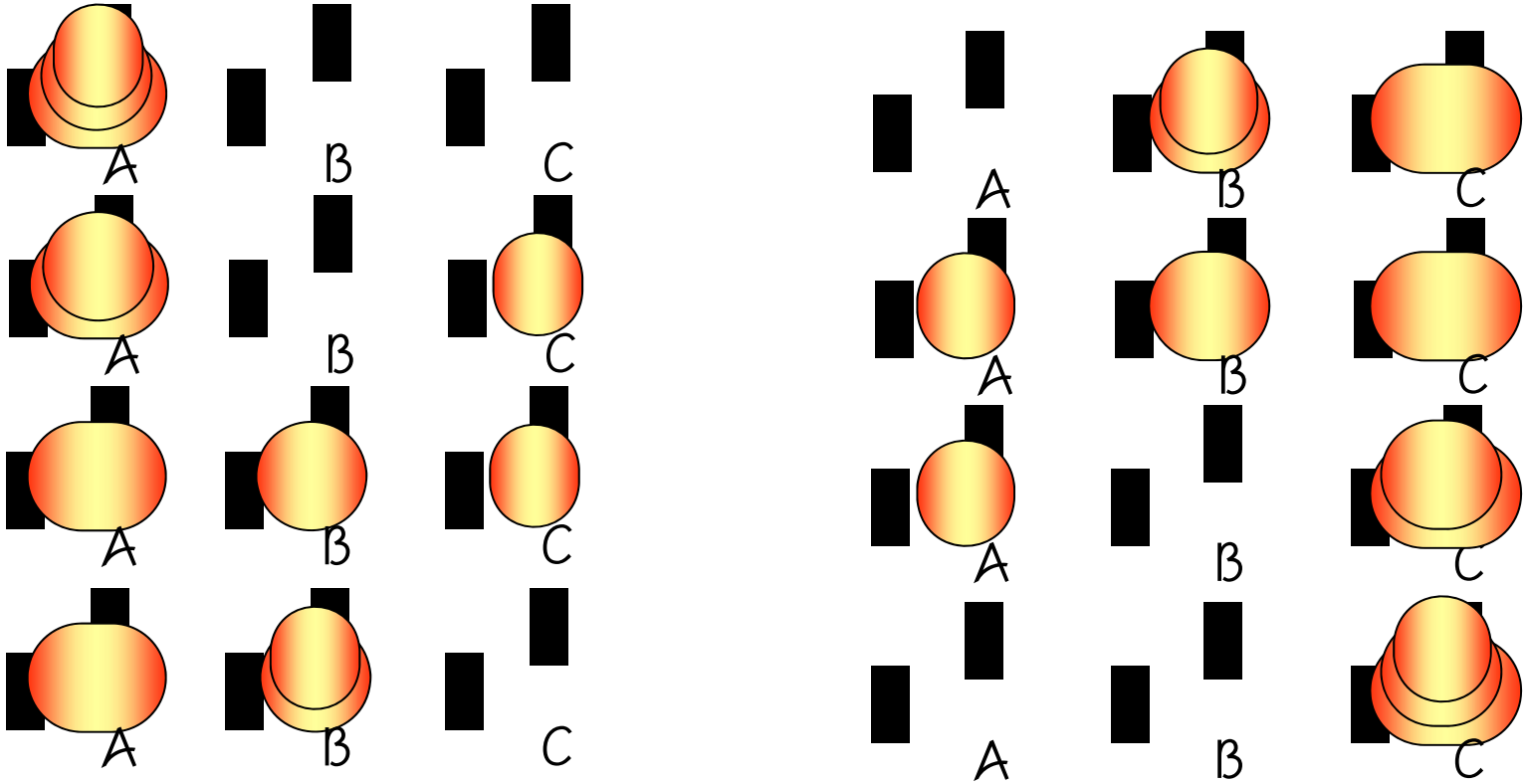
```
// 최대 공약수 구하기
#include <stdio.h>

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

- 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.
  - 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.

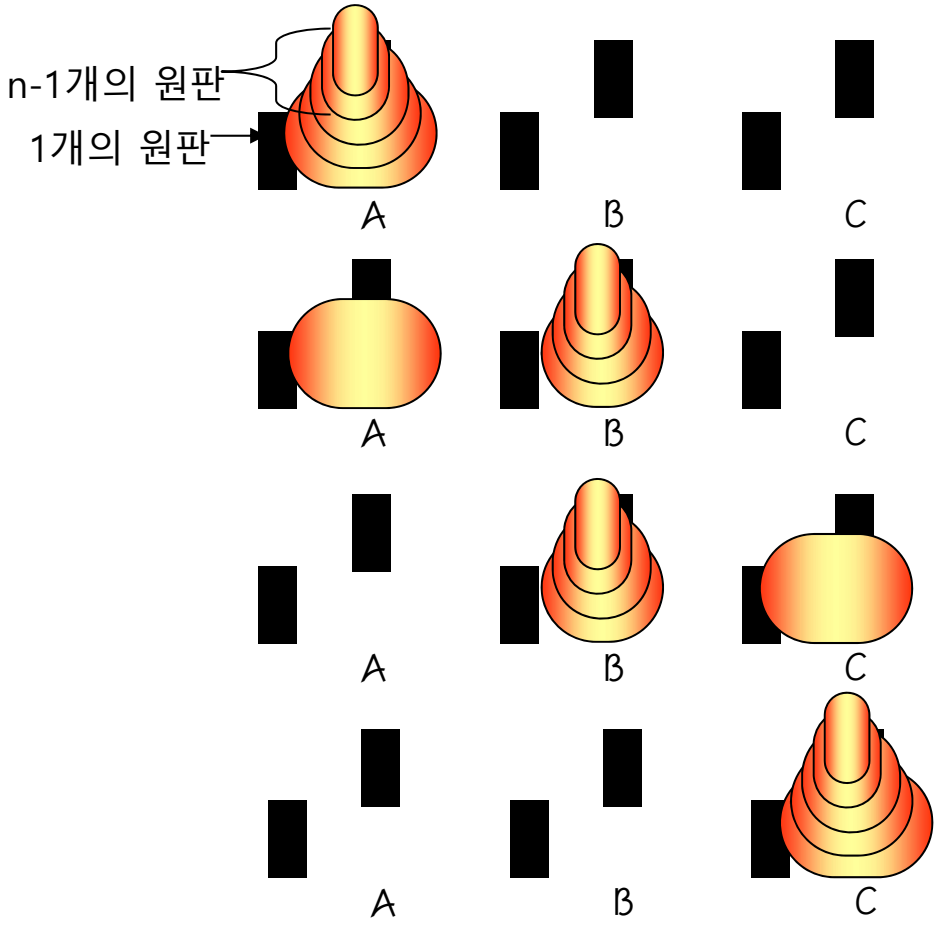


3개의 원판인 경우의 해답



n개의 원판인 경우

- n-1개의 원판을 A에서 B로 옮기고 n번째 원판을 A에서 C로 옮긴 다음, n-1개의 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.
```

Q & A