컴퓨터 프로그래밍 I
(CSE2003-3)

Mon/Wed 16:30-17:45 pm

Lecture 11

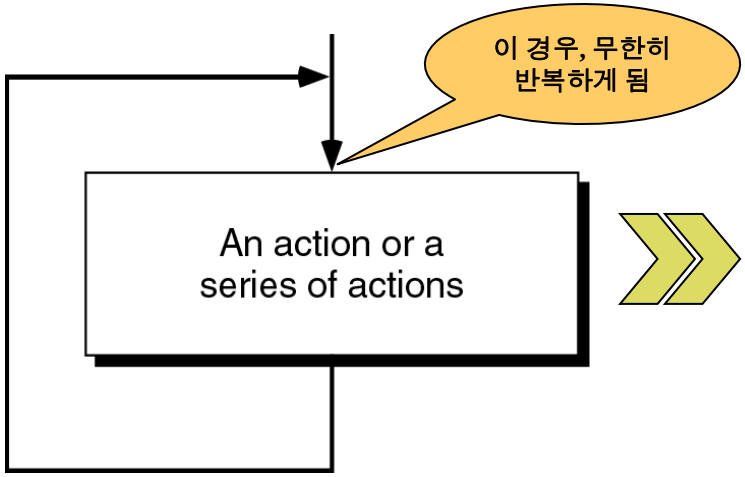
Repetition, Recursion

반복문 재귀함수.

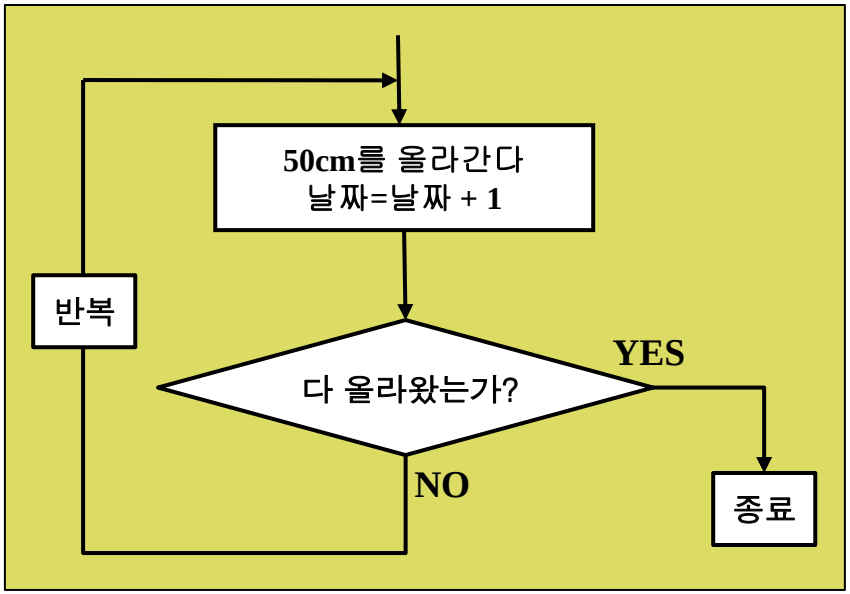
반복문(looping statement)이란?

- 반복문은 어떤 조건에 도달할 때까지 특정 조건(condition)을 만족하는 동안 동일한 작업을 반복하여 수행할 수 있도록 해준다.

우물의 깊이는 3m이고 달팽이는 하루에 50cm를 올라갑니다. 만약 달팽이가 미끄러지지 않고 똑바로 올라간다면 달팽이는 몇 일만에 우물을 벗어날 수 있을까요?



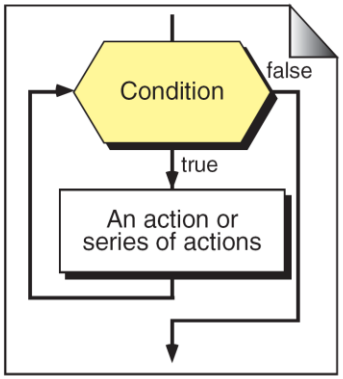
<Loop의 개념>



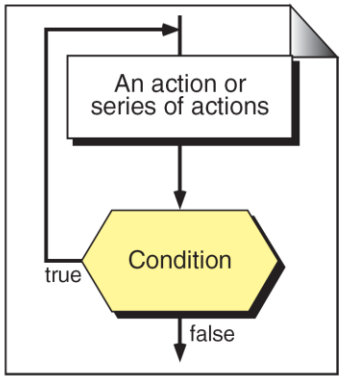
Pretest and Post-test Loops

- Pretest loop
  - loop control expression을 먼저 체크
  - true ➔ loop안의 statements를 실행하고, loop의 첫 단계로 되돌아감
  - false ➔ loop를 중단하고 빠져나감

Condition 에 따라서 statement가 아예 실행되지 않을 수도 있다.



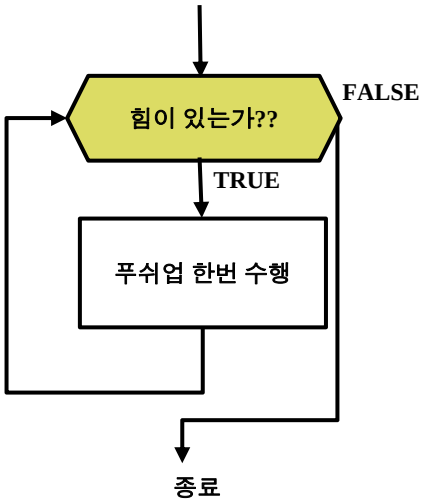
(a) Pretest Loop

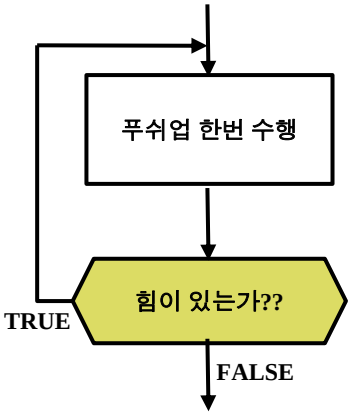


(b) Post-test Loop

- Post-test loop
  - loop 안의 statements를 먼저 실행
  - loop control expression을 체크
  - true ➔ loop의 첫 단계로 돌아감
  - false ➔ loop를 중단하고 빠져나감

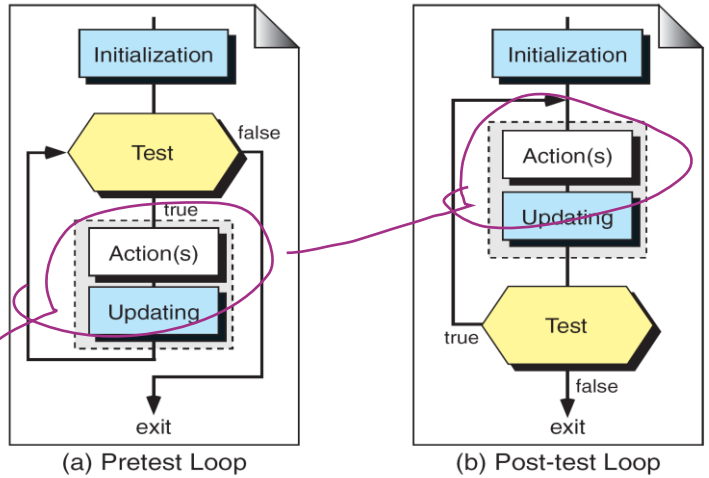
Condition 에 상관없이 statement는 최소한 1회 이상 실행된다.



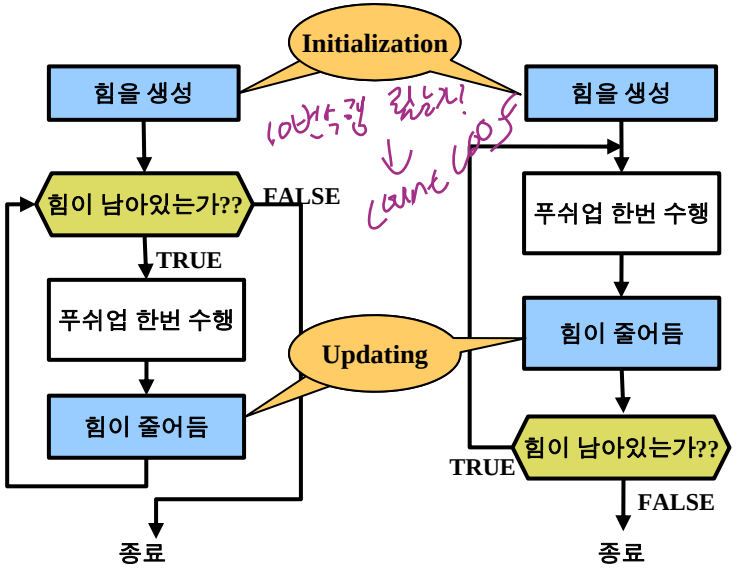


Initialization & Updating

- **Initialization**
  - Loop가 시작되기 전에, **loop initialization (초기화)**가 필요
  - 초기화에는 조건문을 맞추기 위한 초기화와 반복 작업 수행을 위한 변수값 초기화
  - 일반적으로 제어변수(control variable)값을 할당



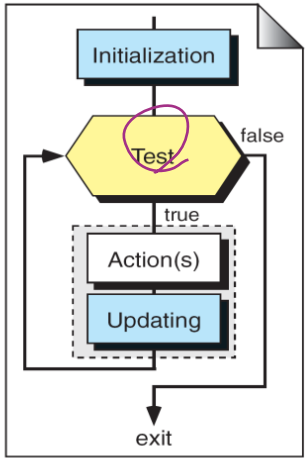
- **Updating**
  - loop body에서 **condition의 변화**가 필요
    ➔ 무한 루프를 피하기 위해서
  - Loop 안의 명령문을 한번 수행할 때마다 Loop Condition이 FALSE 값에 가까워지도록 변화를 줌 ➔ 반복하다 보면 조건값이 결국 FALSE가 되어 Loop 종료



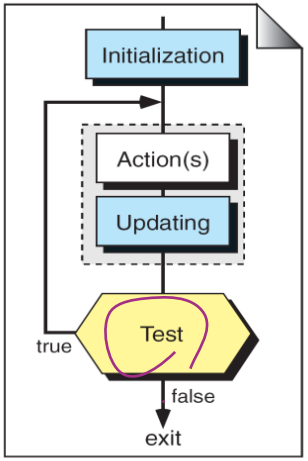
Event & Counter - controlled Loops

- **Event-Controlled Loops**
  - loop control expression 의 값을 true 에서 false로 바꾸는 어떤 **event**가 발생
  - 반복횟수를 알 수 없음.

    ➔ 반복 횟수를 알기 위해서는 별도의 count변수 필요



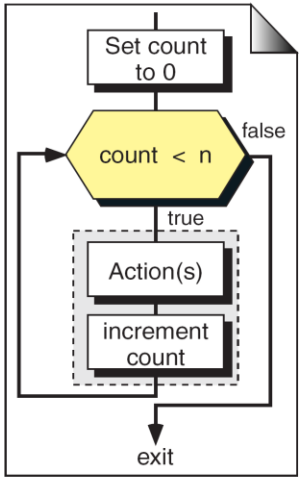
(a) Pretest Loop



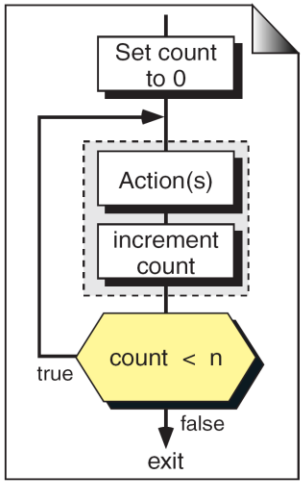
(b) Post-test Loop

- **Count-Controlled Loops**
  - 반복횟수를 condition으로 사용
  - count 변수의 값을 증가 또는 감소시키면서 원하는 횟수만큼 반복했을 때 중지

control expression 이 count 인가 아닌가.



(a) Pretest Loop



(b) Post-test Loop

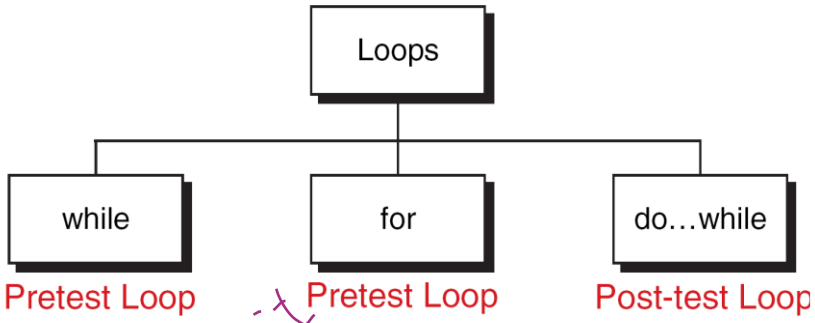
Event & Counter - controlled Loops

- **Loop Comparison**
  - 반복을 n번 수행한다고 할 때, Pretest Loop과 Post-test 반복횟수 비교

	Pretest Loop	Post-test Loop
Initialization	1	1
Number of tests	n+1	n
Action execute	n	n
Updating executed	n	n
Minimum iteration	0	1

C에서의 반복문들

<C loop construct>



<while문>

```
while( 반복 조건 )
{
    반복할 내용
}
```

"반복 조건"이 만족되는 동안 "반복할 내용"을 반복 실행한다.

<for문>

```
for( 초기문 ; 조건문 ; 증감문 )
{
    반복할 내용
}
```

최초 "초기문"을 실행하고, "조건문"을 만족하면 "반복할 내용"과 "증감문"을 차례로 실행한다.

<do while문>

```
do
{
    반복할 내용
} while ( 반복 조건 );
```

일단 루프를 한번 실행 후, "반복조건"이 만족되는 동안 "반복할 내용"을 반복실행한다.

The *while* Loop

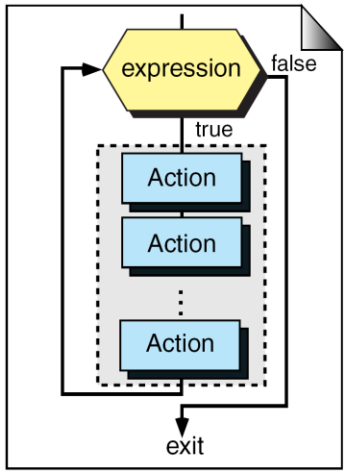
- while 문 구조

기본원리

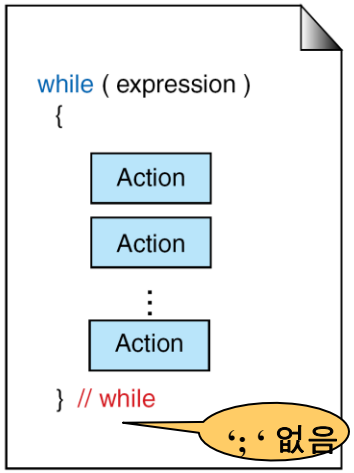
특정 조건을 주고 그 조건이 만족될 때까지
계속해서 반복을 시킨다

```
while( i < 10 )
{
    printf("Hello World!\n");
    i++;
}
```

[while문의 몸체(body)]
한 문장으로 구성되는 경우,
블록형태를 취하지 않아도 됨



(a) Flowchart



(b) C Language

- while 문 실행순서
  - 조건문인 i < 10 을 검사하여 i 가 조건을 만족하면(true) 반복문의 몸체(body)에 당하는 문장이나 블록을 실행한다.
  - 모든 몸체를 실행하면 다시 조건문을 검사하는 첫 번째 과정을 반복한다.
  - 첫 번째 과정에서 조건을 만족하지 않으면(false) while문을 종료한다.

The while loop: 달팽이 우물 탈출 문제

- 예제 프로그램 – while loop

```c
#include <stdio.h>

int main(void)
{
        int height = 0;
        int days = 0;
        int depth;

        printf("Input the depth of well(cm) : ");
        scanf("%d", &depth);

        while(height < depth)
        {
                height = height + 50;
                days = days + 1;
        }

        printf("Total days: %d\n", days);
}
```

```
[root@mclab c-lang]# vi chap6-1.c
[root@mclab c-lang]# gcc -o chap6-1 chap6-1.c
[root@mclab c-lang]# ./chap6-1
Input the depth of well(cm) : 300
Total days: 6
[root@mclab c-lang]#
```

달팽이 우물 탈출 일 수 구하는 프로그램

1) 우물의 높이에 미치지 못하면

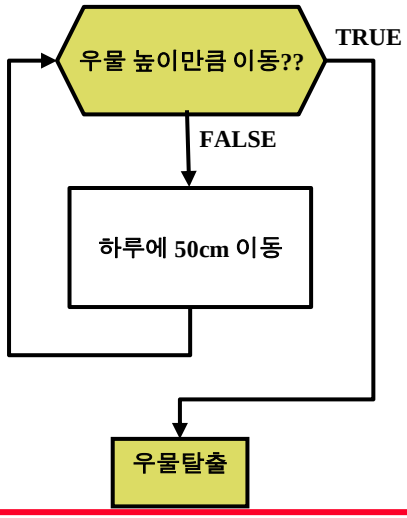
(height < depth)

➔ while 문의 내용을 반복 실행

2) 우물의 높이에 도달하면

(height >= depth)

➔ while문 종료 & 총 소요일수 출력



self 실습: While statement

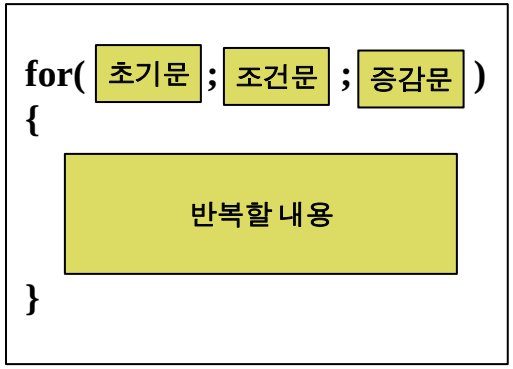
- 사용자로부터 양의 정수 n를 입력 받아 구구단 n단을 출력하는 코드를 작성하시오. (제출하지 않아도 됩니다)
  - 조건:
    - 반드시 while loop을 사용하여 구현한다.
      (for loop 이나 다른 loop문 사용 금지)

```
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
2
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
4
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
```

```
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
7
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
9
9 X 1 = 9
9 X 2 = 18
9 X 3 = 27
9 X 4 = 36
9 X 5 = 45
9 X 6 = 54
9 X 7 = 63
9 X 8 = 72
9 X 9 = 81
```

The *for* Loop

- **for문 구조**
  - **초기문** (Initialization), **조건문** (Limit-test expression), **증감문** (Updating expression)
  - 각 구문별로 세미콜론(;) 으로 구분
    - 괄호( ) 안에 세미콜론은 반드시 필요하다.
    - 괄호( ) 안에 각 세 부분은 생략되어도 문법 오류는 발생하지 않음
      - ☞ for (; ;) { 반복할 내용 }



```
for ( i=0 ; i<10 ; i++)
{
   printf("Hello World!\n");
}
```

[for문의 몸체(body)]
한 문장으로 구성되는 경우,
블록형태를 취하지 않아도 됨

The *for* Loop

- **for 문의 실행순서**

① 초기화(expr1)를 실행한다.

② 조건검사(expr2)을 실행한다.

③ 조건이 참이면

loop body (statement)를 수행한다.

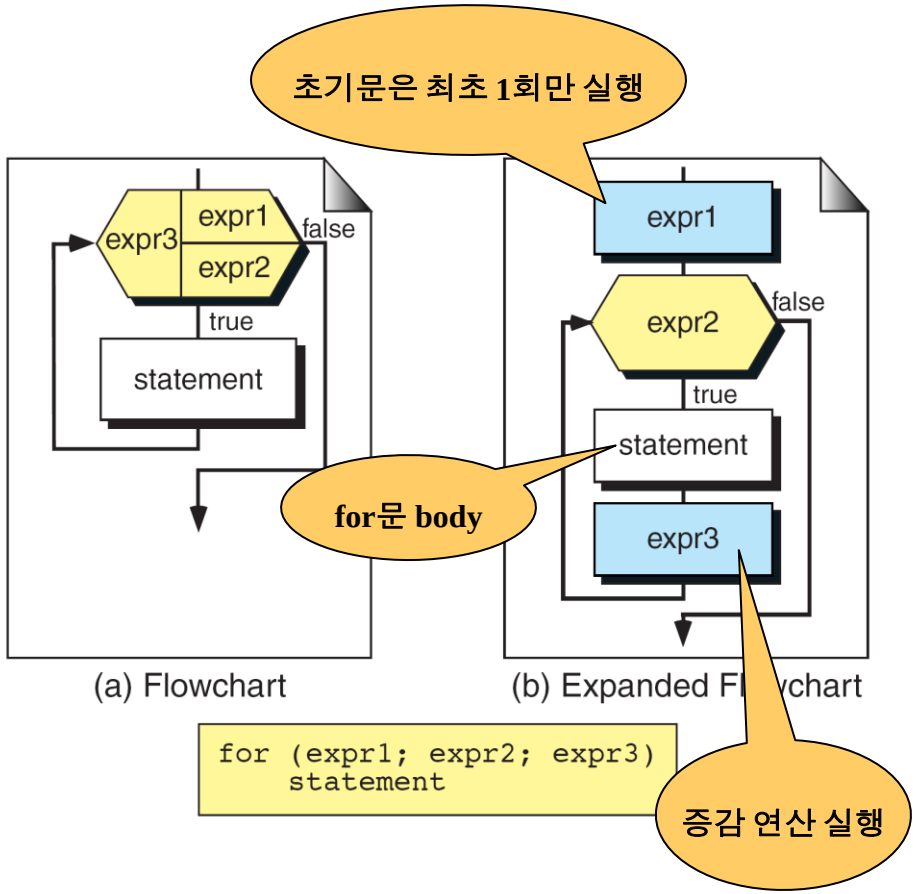
expr3을 수행하고 ②로 돌아간다.

④ 조건이 거짓이면 for문을 종료한다.

```
for ( expr1 ; expr2 ; expr3)
{

   statement

}
```



The *for* Loop

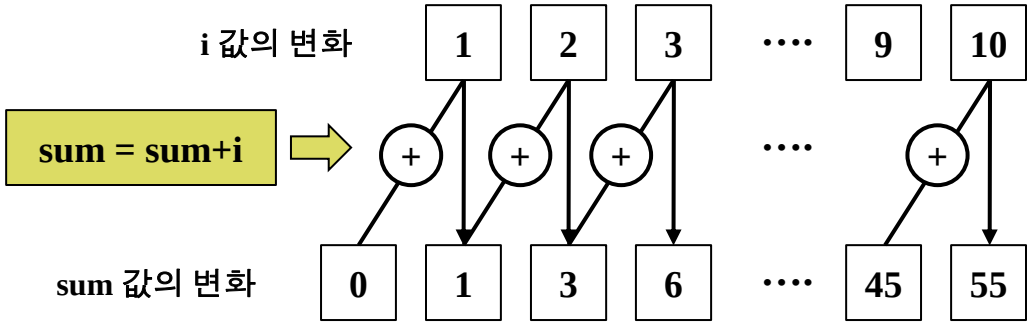
- 예제 프로그램 - for loop

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5         int i, sum=0;
 6
 7         for(i=0; i<=10; i++)
 8                 sum+=i;
 9
10         printf("sum=%d\n", sum);
11         return 0;
12 }
```

```
[root@mclab c-lang]# vi chap6-2.c
[root@mclab c-lang]# gcc -o chap6-2 chap6-2.c
[root@mclab c-lang]# ./chap6-2
sum=55
[root@mclab c-lang]#
```

1부터 10까지 더하는 프로그램

➔ i 를 1부터 10까지 증가시키면서 sum에 더해준다.



The *for* Loop

- 예제 프로그램 - nested for loop

```c
#include <stdio.h>

int main (void)
{
    int i, j;

    for(i=2; i<4; i++)
    {
        for(j=1; j<10; j++)
            printf("%d * %d = %d\n", i, j, i*j);
    }

    return 0;
}
```

구구단 2단과 3단을 출력하는 프로그램

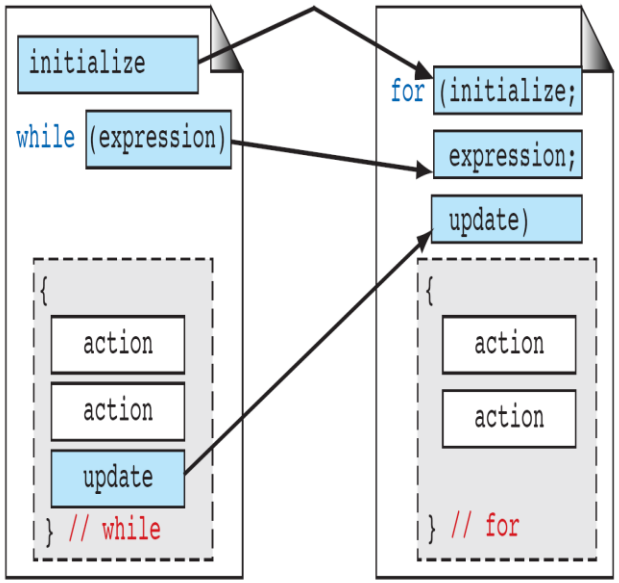
➔ for 문이 중첩되어 사용되고 있다.

(nested for-statement)

```
@cspro:~/lec6$ gcc -o chap6-3 chap6-3.c
@cspro:~/lec6$ ./chap6-3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
@cspro:~/lec6$
```

The *for* Loop

- **for**와 **while** loop의 비교
  - for loop은 while loop과 동일한 역할을 수행하지만 readability 가 좋으며 counting loop에 자연스럽다.



<while loop>

```
i=1;
sum=0;
while(i<=20)
{
    scanf("%d",&a);
    sum += a;
    i++;
} //while
```

<for loop>

```
sum = 0;
for (i = 1; i <=20 ; i++)
{
    scanf("%d",&a);
    sum += a;
} // for
```

거듭제곱 구하기

```c
#include <stdio.h>

int powers(int base, int exp);

int main (void)
{
    int base, exp;

    printf("base : ");
    scanf("%d", &base);

    printf("exponent : ");
    scanf("%d", &exp);

    printf("%d^%d=%d\n",base,exp,powers(base,exp));

    return 0;
}

int powers(int base, int exp)
{
    int result = 1;
    int i;

    if(base < 1 || exp <0)
        result = 0;
    else
        for(i=1; i<=exp; i++)
            result *= base;

    return result;
}
```

양의 정수 밑(base)과 지수(exponent)를 입력 받아 거듭제곱을 구하는 프로그램

1) 사용자로부터 밑과 지수를 입력 받는다.
2) 만약 밑이 1보다 작거나 지수가 음수이면 결과값은 0으로 한다.
3) powers()함수의 for 문에서 입력 받은 지수만큼 루프를 돌게 되어 있다.
4) 계산한 결과를 반환하여 main함수에서 출력한다.

```
spro:~$ gcc -o powers powers.c
spro:~$ ./powers
base : 4
exponent : 2
4^2=16
spro:~$ ./powers
base : 5
exponent : 0
5^0=1
```

self 실습: for-loop

- 사용자로부터 양의 정수 n를 입력 받아 구구단 n단을 출력하는 코드를 작성하시오. (제출하지 않아도 됩니다)
  - 조건:
    - 반드시 for loop을 사용하여 구현한다.
      (while loop 이나 다른 loop문 사용 금지)

```
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
2
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
4
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
```

```
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
7
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
nlp908@sogangnlpgpu1 ~/ied $ ./a.out
9
9 X 1 = 9
9 X 2 = 18
9 X 3 = 27
9 X 4 = 36
9 X 5 = 45
9 X 6 = 54
9 X 7 = 63
9 X 8 = 72
9 X 9 = 81
```

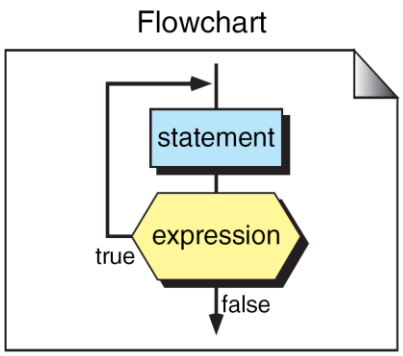
The do..while Loop: post-test loop

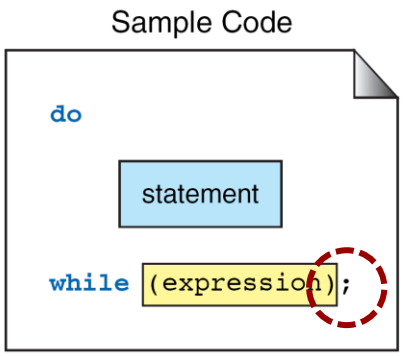
- **while** 문은 조건이 만족되지 않으면 루프를 아예 한번도 실행하지 않는 경우가 있다.
- **do...while**문은 반드시 한번은 루프를 실행하도록 되어있다.

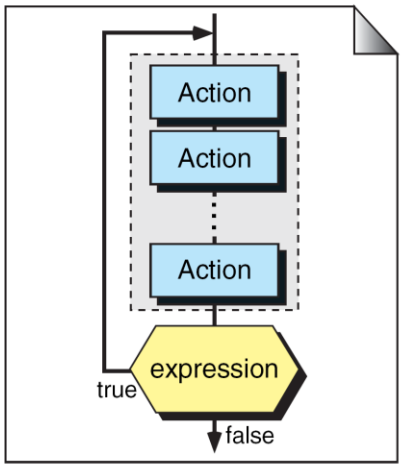
  ➔ **Post-test Loop**

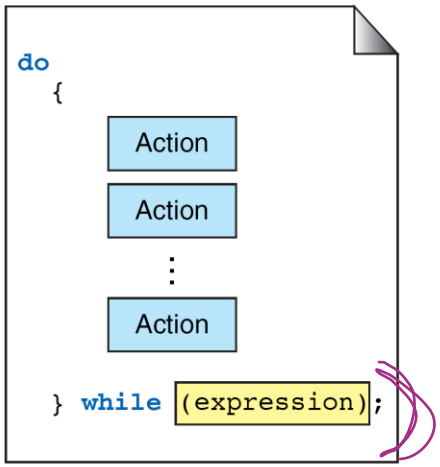
- 경우에 따라서는 **while**문보다 **do...while**문을 사용하는 것이 훨씬 자연스러운 경우가 있다

while(expression) 끝에 세미콜론(;)을 반드시 붙여야 한다!!!









The do..while Loop

- 예제프로그램 – do..while loop

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5         int total = 0;
 6         int val = 0;
 7
 8         do {
 9                 printf("Input number(Quit : 0) : ");
10                 scanf("%d", &val);
11                 total+=val;
12         }while(val != 0);
13
14         printf("Total : %d\n",total);
15         return 0;
16 }
```

사용자가 입력하는 수를 계속해서 더하는 프로그램

1) 먼저 사용자로부터 숫자를 입력 받음
2) 입력 받은 값으로 진행할 것인지 말 것인지 결정

➔ 최소한 한 번의 실행이 필요한 상황에서는 do ~ while 문이 while 문 보다 자연스럽다.

```
[root@mclab chap6]# vi chap6-4.c
[root@mclab chap6]# gcc -o chap6-4 chap6-4.c
[root@mclab chap6]# ./chap6-4
Input number(Quit : 0) : 2
Input number(Quit : 0) : 4
Input number(Quit : 0) : -5
Input number(Quit : 0) : 3
Input number(Quit : 0) : 1
Input number(Quit : 0) : 0
Total : 5
[root@mclab chap6]#
```

self 실습: do..while Loop

- **사용자에게 정수 n($n \geq 0$)을 입력 받아 n번째 Fibonacci number를 출력하는 프로그램을 작성해 보자.**
  - Fibonacci series
    - Fibonacci(0) = 0, Fibonacci(1) = 1
    - Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2), ($n \geq 2$)
  - 조건:
    - 1. 입력 받은 정수 n이 음수가 아닐 경우 Fibonacci number를 출력하고, 음수 일경우 error 메시지를 출력 후 종료한다.
    - 2. do-while 문을 사용하여 구현한다. **(for, while 등 다른 loop 문 사용 금지)**
    - 3. Main function에서는 입력, 입력 오류 검사, function call, 출력만 수행한다.
    - 4. User defined function을 통해 n번째 Fibonacci number를 return받아 출력하도록 한다.

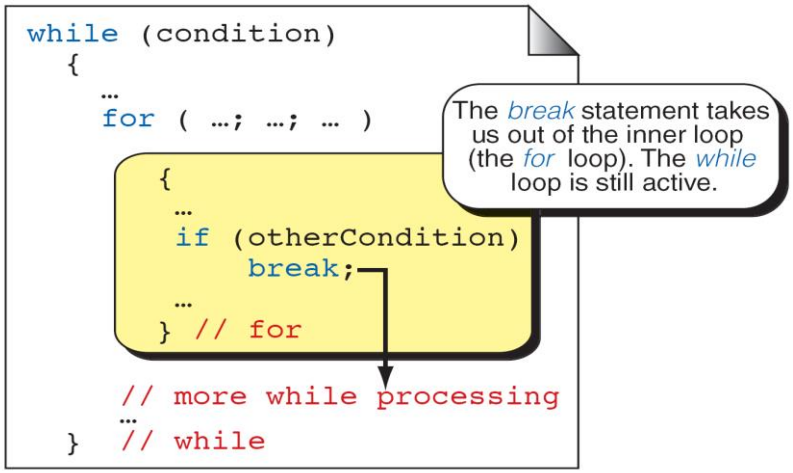
self 실습: do..while Loop

- 실행 예시:

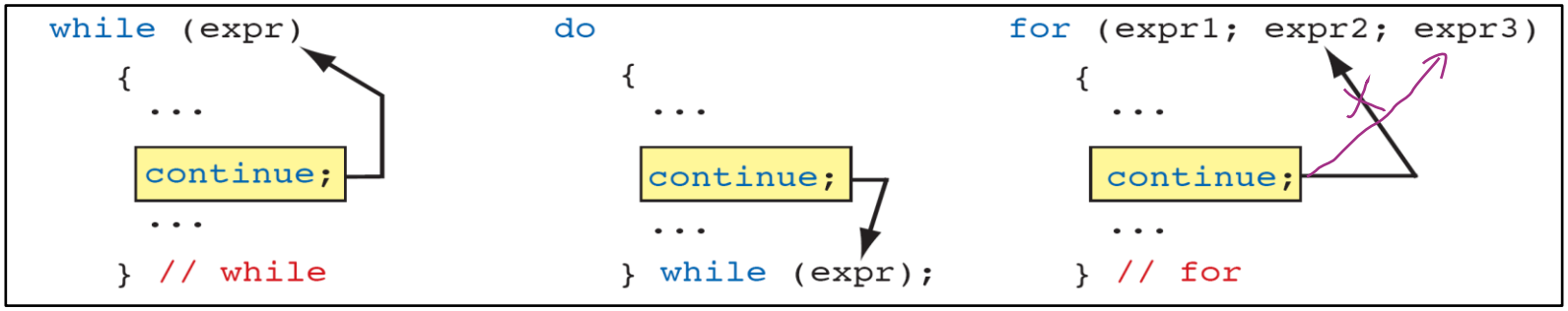
```
limjongbum@ECL-Server3:~/ied2014$ gcc -o fibonacci fibonacci.c
limjongbum@ECL-Server3:~/ied2014$ ./fibonacci
Input n: 0
fib(0) = 0
limjongbum@ECL-Server3:~/ied2014$ ./fibonacci
Input n: 8
fib(8) = 21
limjongbum@ECL-Server3:~/ied2014$ ./fibonacci
Input n: -1
n cannot be negative number.
limjongbum@ECL-Server3:~/ied2014$
```

break와 continue

break : 가장 가까이서 감싸고 있는 반복문 블록 하나를 빠져 나오게 한다.
주로 제한조건이 반복문, 조건문의 중간에 만족하였을 때 사용.



continue : 돌던 루프의 남아있는 부분을 그냥 건너뛰는 것.
- **while이나 do-while문은 검사식(conditional expression)으로 제어 이동**
- **for문은 변경식(update expression) 으로 제어 이동**



break와 continue

- 예제 프로그램 - break와 continue

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5         int i, sum = 0;
 6
 7         for(i=0; ; i++) {
 8                 if(i>=100)
 9                         break;
10                 else if(i%4 == 0)
11                         continue;
12                 sum+=i;
13         }
14         printf("sum=%d\n", sum);
15 }
```

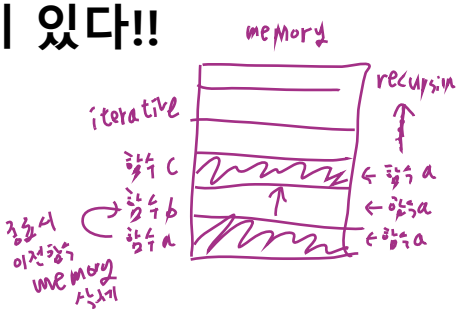
1과 99사이의 4의 배수가 아닌 숫자들의 합을 구하는 프로그램

1) for 문에서 조건식이 없으므로 기본적으로는 무한루프를 돌게 되어 있다.
2) i가 100보다 커지는 경우 break를 만나 루프를 빠져나옴.
3) 4의 배수인 경우에는 continue를 만나서 sum에 더해지지 않는다.

```
[root@mclab chap6]# vi chap6-5.c
[root@mclab chap6]# gcc -o chap6-5 chap6-5.c
[root@mclab chap6]# ./chap6-5
sum=3750
[root@mclab chap6]#
```

재귀호출(Recursion)

- **프로그램에서 동일한 작업을 반복하여 수행할 수 있는 방법은?**
  - **앞서 배운 while, for, do..while 와 같은 반복문(loop) 사용. - (iterative method)**
  - **또 다른 방법은 재귀호출(Recursion)을 이용하는 방법이 있다!!**



- **Recursion 이란?**
  - 함수가 자기 자신을 반복하여 호출하는 형태.
  - 자기 스스로 반복 호출 함으로써, 코드를 간단/명료하게 만들 수 있다는 장점이 있다. (매우 중요한 장점!!!)
  - Limitation of Recursion.
    - 실제 속도가 iterative에 비해 떨어지므로 개념적으로만 사용한다.
    - 예외로 일부 특수한 문제는 recursion을 사용할 수밖에 없음.
    - 특수한 문제가 아니면 반복적 정의(while, for, do~while)로 처리 가능.
    - 필요한 메모리의 양이 크다.

재귀호출(Recursion)

Iterative function vs Recursive function

반복문 재귀함수.

< iterative factorial definition >

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) \ldots 3 * 2 * 1 & \text{if } n > 0 \end{cases}$$

factorial(4) = 4 * 3 * 2 * 1 = 24

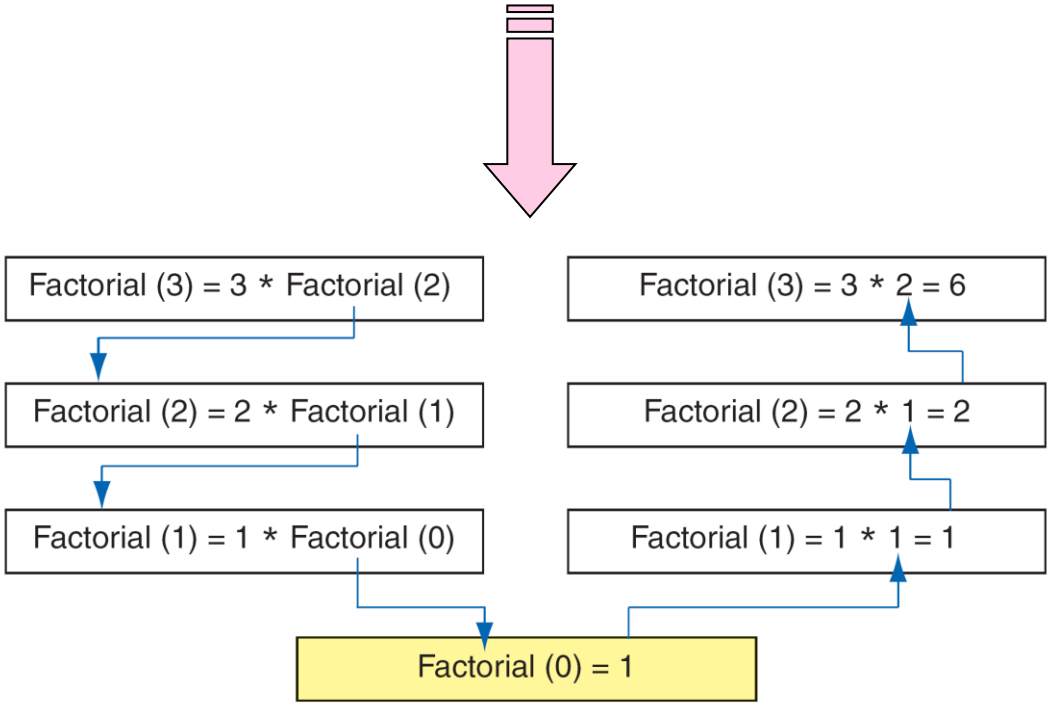
< recursive factorial definition >

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1 * factorial(0)

재귀호출(Recursion)

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$



< factorial (3) recursively >

Recursion solution(1/2)

예제 프로그램 - recursion

```
[root@mclab chap6]# vi chap6-6.c
[root@mclab chap6]# gcc -o chap6-6 chap6-6.c
[root@mclab chap6]# ./chap6-6
Input a number : 5
5! = 120
[root@mclab chap6]#
```

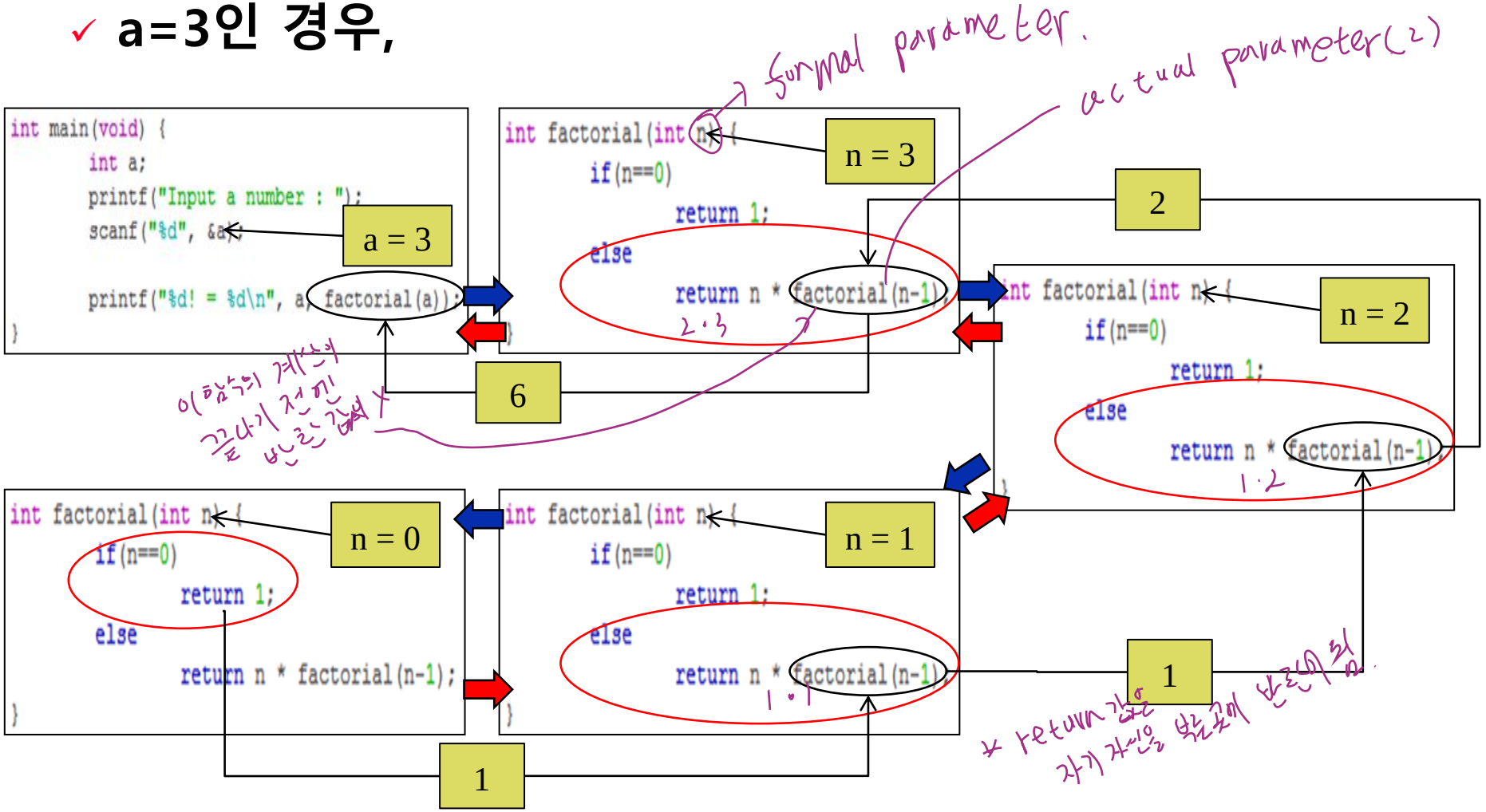
```
1  #include <stdio.h>
2
3  int factorial(int n);
4
5  int main(void) {
6          int a;
7          printf("Input a number : ");
8          scanf("%d", &a);
9
10         printf("%d! = %d\n", a, factorial(a));
11 }
12
13 int factorial(int n) {
14         if(n==0)
15                 return 1;
16         else
17                 return n * factorial(n-1);
18 }
```

함수 factorial() 안에서 factorial()를 호출하는 프로그램

1) 호출할 때 인자는 매번 1씩 줄어들기 때문에 언젠가는 factorial(0) 을 호출하게 된다.
2) factorial(0)은 1을 반환한다.
3) 호출된 순서를 거꾸로 올라가며 factorial값을 구한다.

Recursion solution(2/2)

- a=3인 경우,



Iterative solution

예제 프로그램 - iterative

```c
#include <stdio.h>

int factorial(int n);

int main(void){
    int a;
    printf("Input the number : ");
    scanf("%d", &a);

    printf("%d! = %d\n",a,factorial(a));
    return 0;
}

int factorial(int n){
    int factN = 1;
    int i;
    for(i = 1; i <= n; i++){
        factN = factN * i;
    }
    return factN;
}

```

recursive function은
loop문을 이용하여
iterative로 도 구현할 수도 있다.

```
gr120100205@cspro:~$ gcc -o 6-7 6-7.c
gr120100205@cspro:~$ ./6-7
Input the number : 5
5! = 120
gr120100205@cspro:~$
```

Recursion

- **Designing Recursive Function**
  - 모든 recursive call은 problem의 size를 줄이거나, problem의 부분을 풀어야 한다.
  - Recursive function을 design 하는 순서

1. base case 를 결정한다.
2. general case 를 결정한다.
3. base case와 general case 를 function에 모두 적용시킨다.

모든 recursive function은 base case를 지닌다. Base case는 실질적으로 문제를 해결하는 부분이다.Factorial 의 경우 base case는 factorial(0)이다

Base case를 제외한 함수의 나머지 부분. factorial 의 경우, general case는 n*factorial(n-1)이다

- **Limitation of Recursion. (again)**
  - 실제 속도가 iterative에 비해 떨어지므로 개념적으로만 사용한다.
  - 예외로 일부 특수한 문제는 recursion을 사용할 수밖에 없음.
  - 특수한 문제가 아니면 반복적 정의(while, for, do~while)로 처리 가능.

Fibonacci series

예제 프로그램 - Fibonacci series

```c
#include <stdio.h>

long fib (long);

int main (void)
{
    int seriesSize;
    int i;

    printf("This program prints a Fibonacci series.\n");
    printf("How many numbers do you want?");
    scanf("%d", &seriesSize);
    if (seriesSize < 2)
        seriesSize = 2;

    printf("First %d Fibonacci numbers: \n", seriesSize);
    for (i = 0; i < seriesSize; i++)
    {
        if (i % 5)
            printf(", %8ld", fib(i));
        else
            printf("\n%8ld", fib(i));
    }
    printf("\n");
    return 0;
}

long fib (long num)
{
    if (num == 0 || num == 1)
        return num;

    return (fib (num - 1) + fib (num-2));
}
```

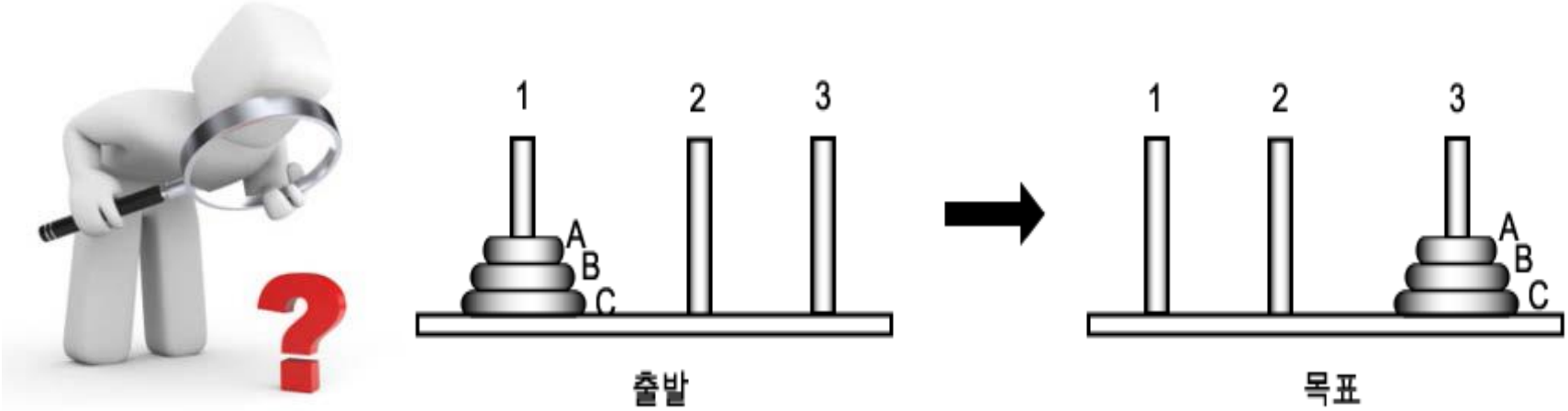
base case

general case

```
This program prints a Fibonacci series.
How many numbers do you want?30
First 30 Fibonacci numbers:

       0,       1,       1,       2,       3
       5,       8,      13,      21,      34
      55,      89,     144,     233,     377
     610,     987,    1597,    2584,    4181
    6765,   10946,   17711,   28657,   46368
   75025,  121393,  196418,  317811,  514229
```

Puzzle: Tower of Hanoi

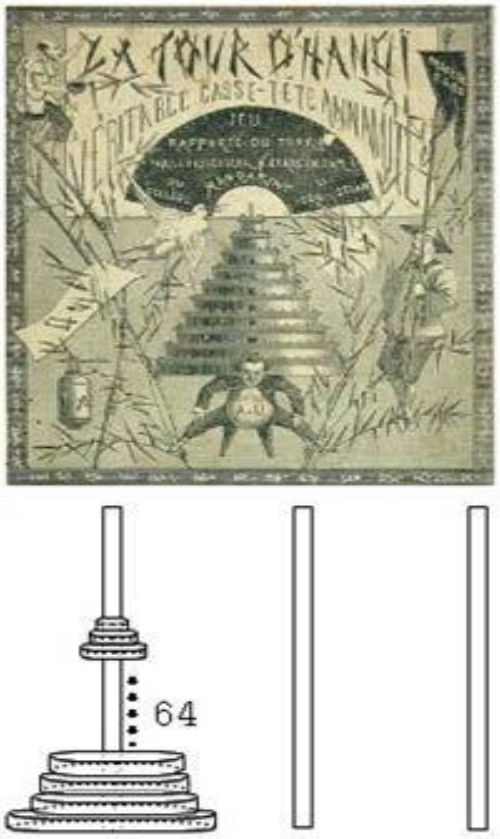


- It consists of three rods, and a number of disks of different sizes which can slide onto any rod.

- The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The puzzle was invented by the mathematician Lucas

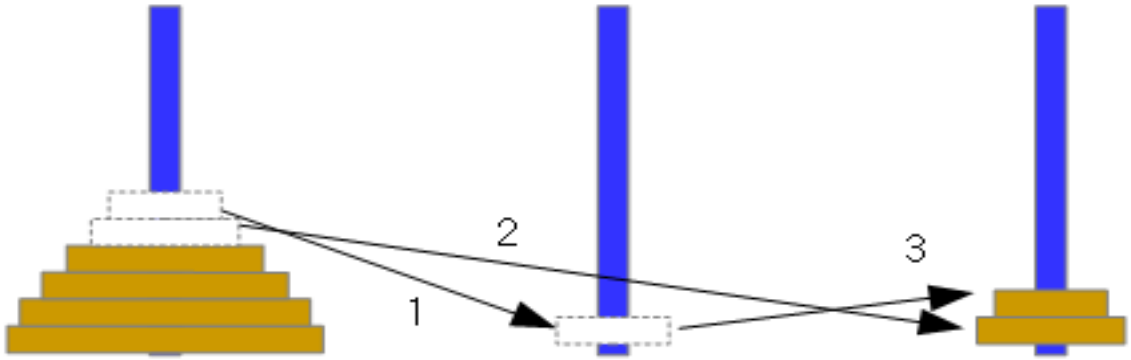
There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time. According to the legend, when the last move of the puzzle will be completed, the world will end.



- The puzzle was invented by the French mathematician Lucas in 1883.

Puzzle: Tower of Hanoi

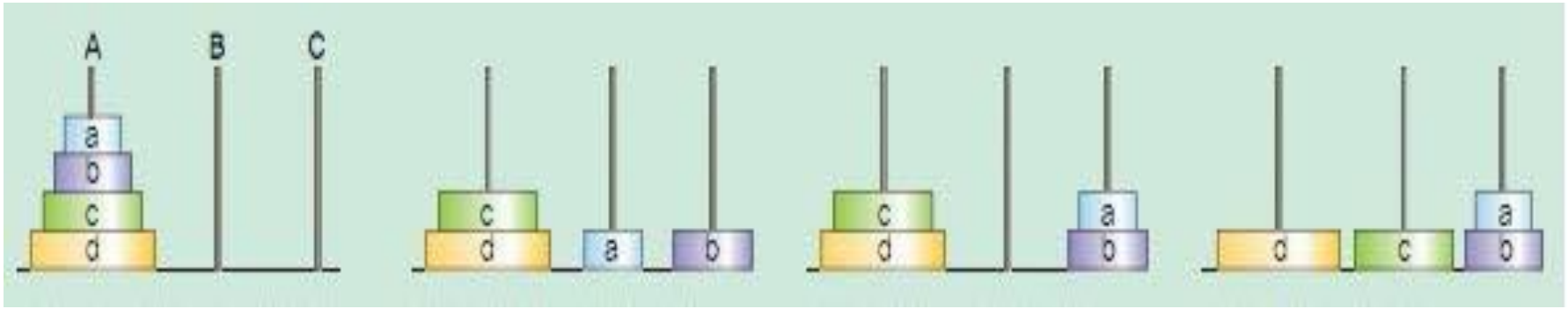
- The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
  - Only one disk can be moved at a time.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
  - No disk may be placed on top of a smaller disk.



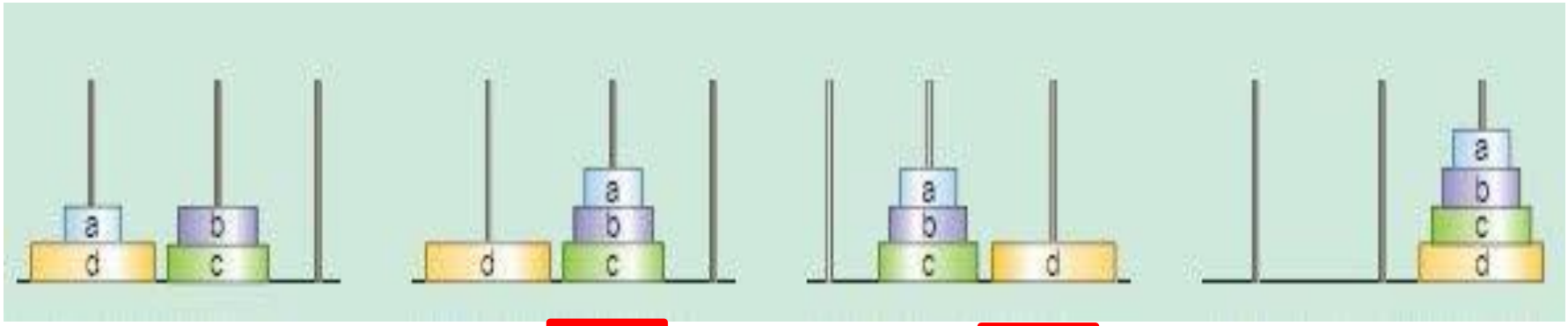
Puzzle: Tower of Hanoi

- The number of disks = 4

Puzzle: Tower of Hanoi

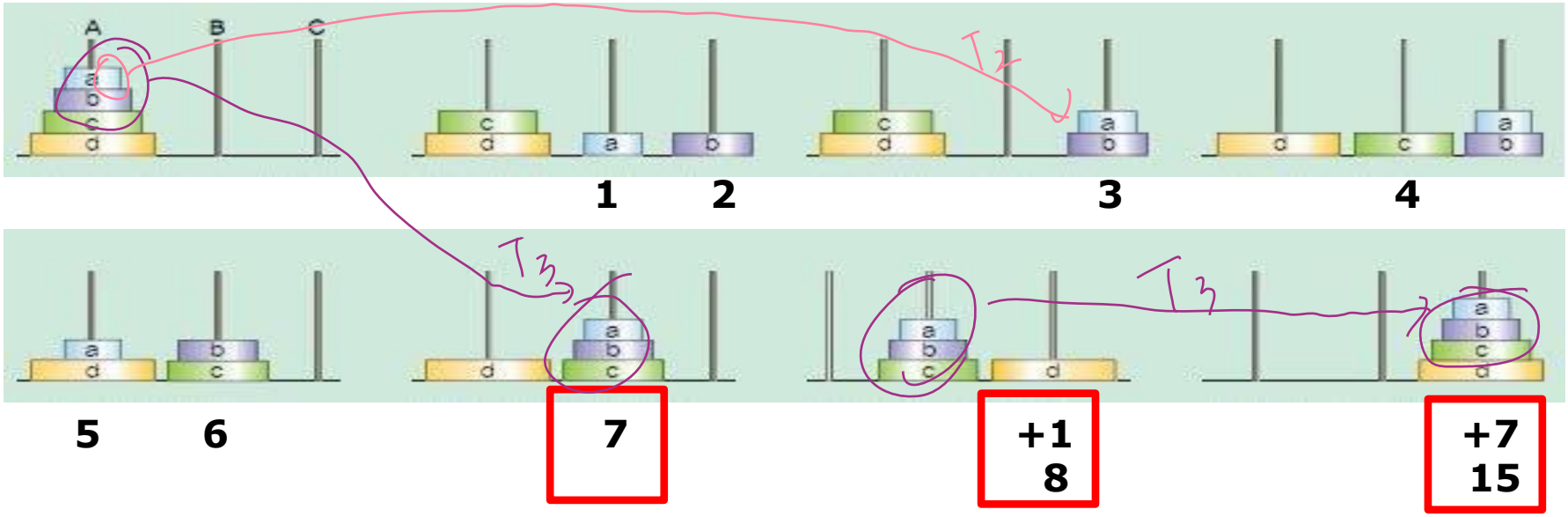


1 2 3 4



5 6 7 +1 8 +7 15

Puzzle: Tower of Hanoi



$$T_4 = T_3 + 1 + T_3 = 15$$
$$T_3 = T_2 + 1 + T_2 = 7$$
$$T_2 = T_1 + 1 + T_1 = 3$$
$$T_1 = 1$$

Puzzle: Tower of Hanoi

$$T_1 = 1 \qquad = 2^1 - 1 = 1$$
$$T_2 = T_1 + 1 + T_1 = 2^2 - 1 = 3$$
$$T_3 = T_2 + 1 + T_2 = 2^3 - 1 = 7$$
$$T_4 = T_3 + 1 + T_3 = 2^4 - 1 = 15$$
$$\cdots$$
$$T_n = T_{n-1} + 1 + T_{n-1} = 2^n - 1$$

$$T_{64} = T_{63} + 1 + T_{63} = 2^{64} - 1$$

$= 18,446,744,073,709,551,615$회 이동

약 5849억4241만7355년 (1초 1번 이동 가정)

Puzzle: Tower of Hanoi → 반복의 수식

$$T_1 = 1 = 2^1 - 1$$

$$T_2 = 2T_1 + 1 = 2^2 - 1$$

$$T_3 = 2T_2 + 1 = 2^3 - 1$$

$$T_4 = 2T_3 + 1 = 2^4 - 1$$

$$\cdots$$

$$T_n = 2T_{n-1} + 1 = 2^n - 1$$

RECURSION
"Recursive"