전산 SMP 4주차

2014. 10. 13

김범수
bskim45@gmail.com

지난시간 복습

조건문 (Selection Statement)

- 조건 (Condition)에 따라서 선택적으로 프로그램을 진행
- 프로그램의 Flow를 조종

2-Way Selection

- 두 가지 선택지 중에 한 가지

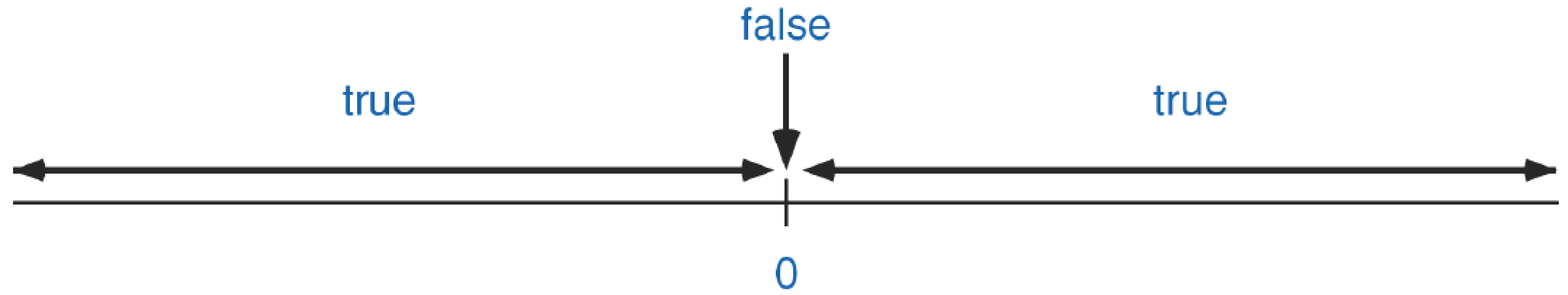
Multi-Way Selection

- 여러 가지 선택지 중에 한 가지(혹은 그 이상)

- 조건문 안에 얼마든지 또 조건문을 넣을 수 있다. (nested)

조건 Condition (TRUE / FALSE)

- 조건문과 반복문 등에서 ‘조건 검사’를 할 때 사용
- C에서는 0을 FALSE, 0이 아니면 TRUE
- ex. if(0.4)는 TRUE로 취급



2-Way Selection – if ~ else

```
if (조건)
{
        //조건이 참(True)일 때
}
else
{
        //조건이 거짓(False)일 때
}
```

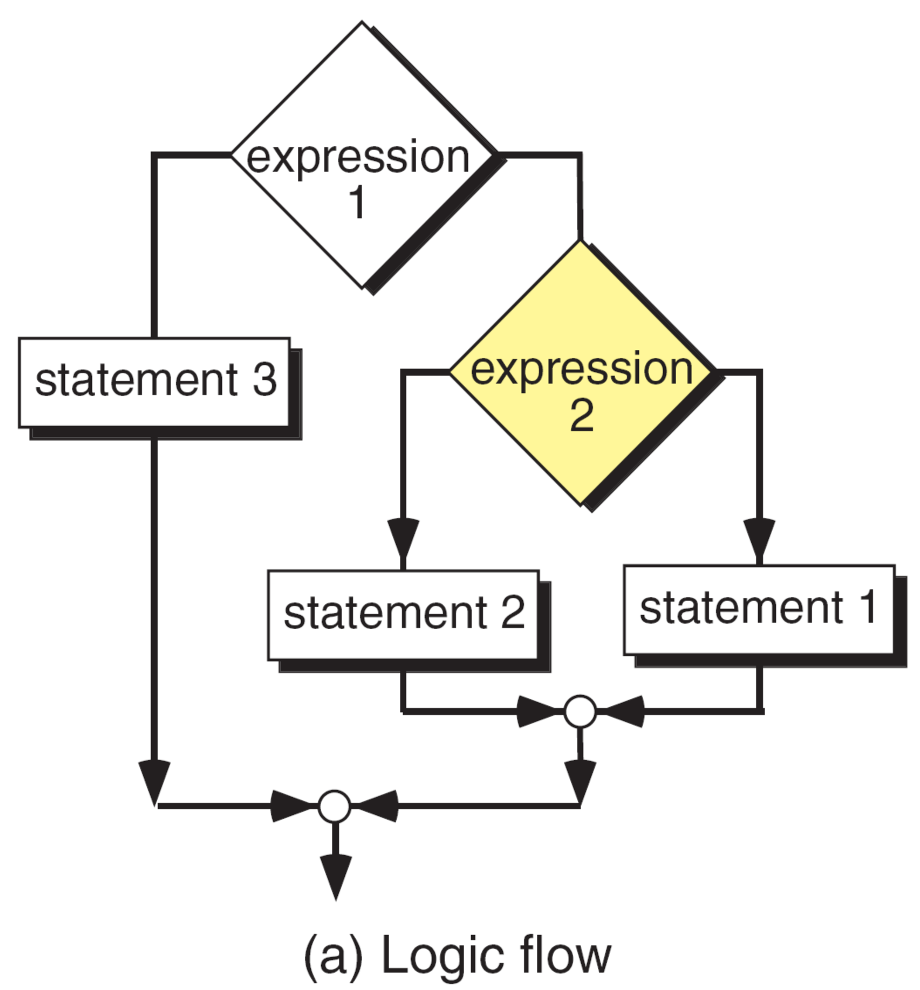
false (zero)
expression
true (!zero)
statement2
statement1

(a) Logical Flow

```
if (expression)
    statement1
else
    statement2
```

(b) Code

Nested *if* Statements



(a) Logic flow

```
if (expression 1)
    if (expression 2)
        statement 1
    else
        statement 2
else
    statement 3
```

(b) Code

Multi-Way Selection

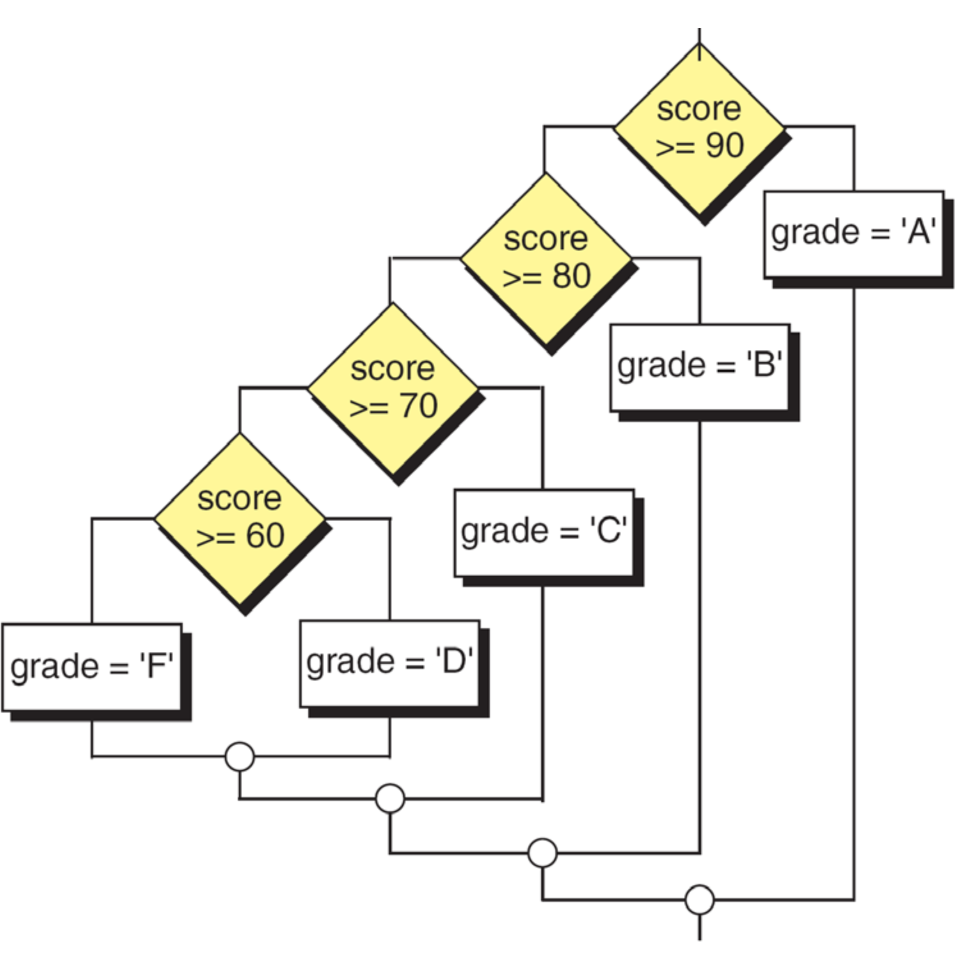
- 여러 가지 선택지 중에서 하나 (혹은 그 이상)을 선택해야 할 때

- 종류
  - else if 구문
  - switch 구문

if ~ else if ~ else

- if…else… 구문의 확장형

```
if( 조건문 )
{
        //조건문1 이 참일 때
}
else if(조건문2)
{
        //조건문1 이 거짓이고 조건문2가 참일 때
}
else
{
        //앞의 모든 조건문이 거짓일 때
}
```



switch 문

- 검사하고자 하는 대상의 값(변수, 함수의 리턴 값)에 여러 가지 선택지가 있을 때

```
switch (검사대상)
{
        case 값1:
                statement1_1;
                statement1_2;
        case 값2 :
                statement2_1;
                statement2_2;
        case 값3 :
                statement3_1;
                statement3_2;
}
```

값1 값2 값3 == 검사대상

break;

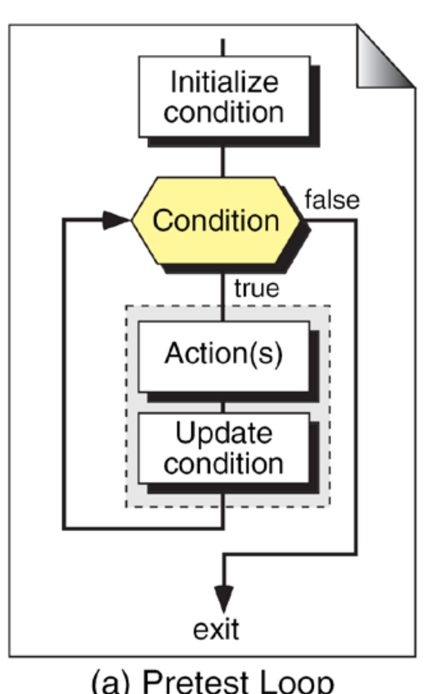
- switch, while, for, do while 같은 조건문 & 반복문을 빠져나가는 명령어
- if 문은 안됨
- 가장 가까운 하나만 빠져나갈 수 있다.
- 사용 용도
  - 특정한 순간에 조건문을 나가고 싶다.
  - 특정한 순간에 반복문을 나가고 싶다.

반복문

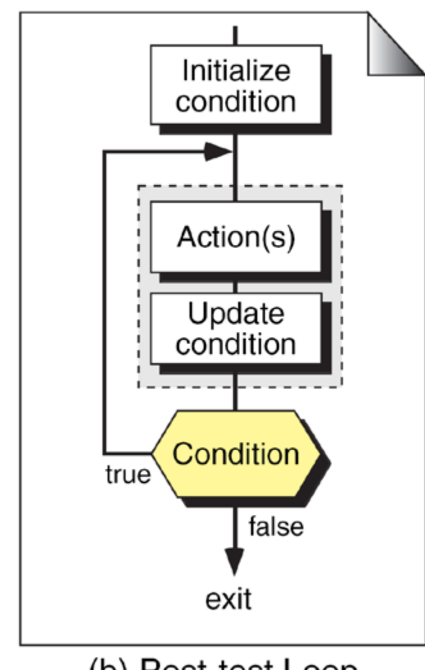
Loop (고리)

- Loop 문은 일련의 작업들을 반복해서 실행하게 하는 것.
  ex. 1~50 까지의 합 구하기, 자동 냉방 장치 시스템
- pre-test loop: 검사하고 실행하기
  while 문, for 문
- post-test loop: 실행하고 검사하기
  do~while 문

- Requirement
  - Initialization
  - Update
  - Condition Check



(a) Pretest Loop



(b) Post-test Loop

for 문

- Pre-test Loop

for (initialization; condition test; update)

{

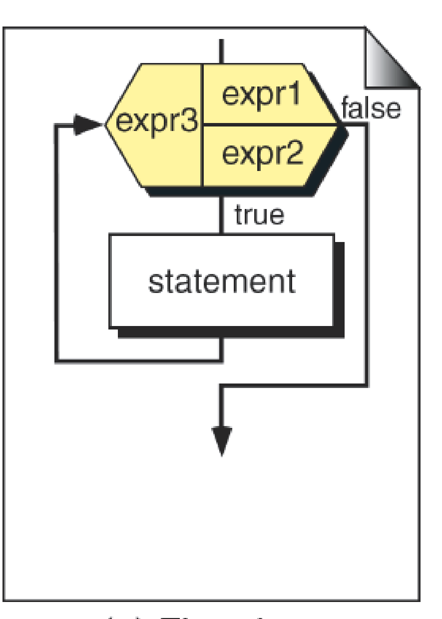
//statement;

//statement;

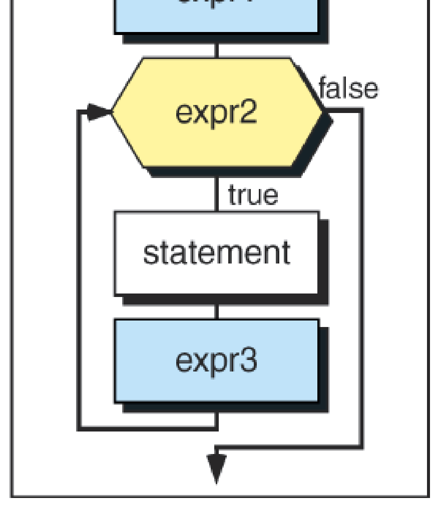
...

}

I -> C (T) -> S -> U -> C (T) -> S -> U -> C (T) -> S -> U -> C (F)-> loop out



(a) Flowchart



(b) Expanded Flowchart

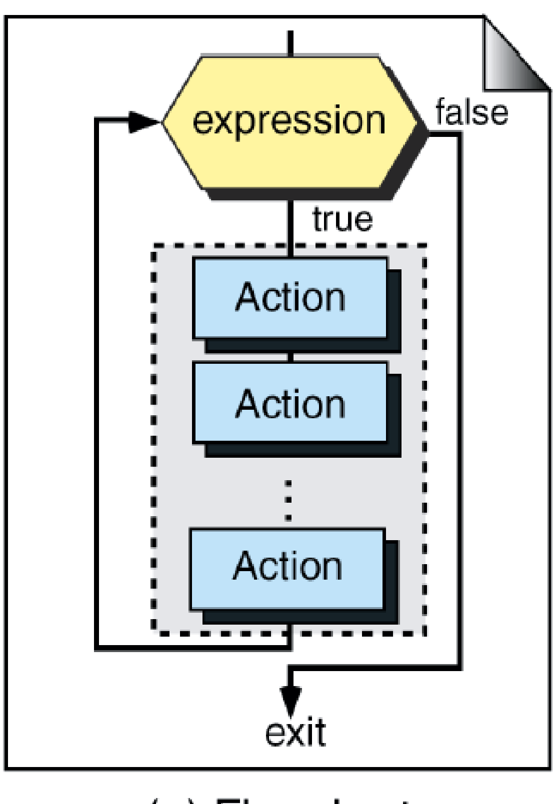
```
for (expr1; expr2; expr3)
    statement
```

while 문

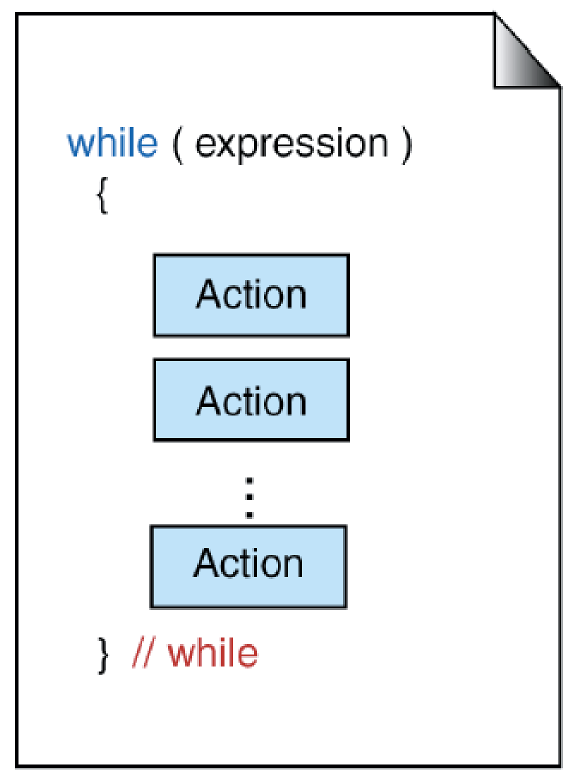
- Pre-test Loop
- 조건이 True인 동안 Statements를 계속 실행

```
while (condition)
{
    //statement;
    //statement;;
        ...
}
```

C -> S -> C -> S -> C -> loop out
T T F



(a) Flowchart



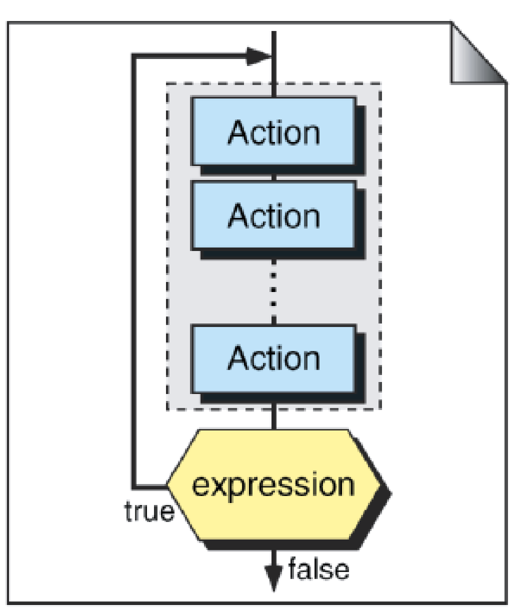
(b) C Language

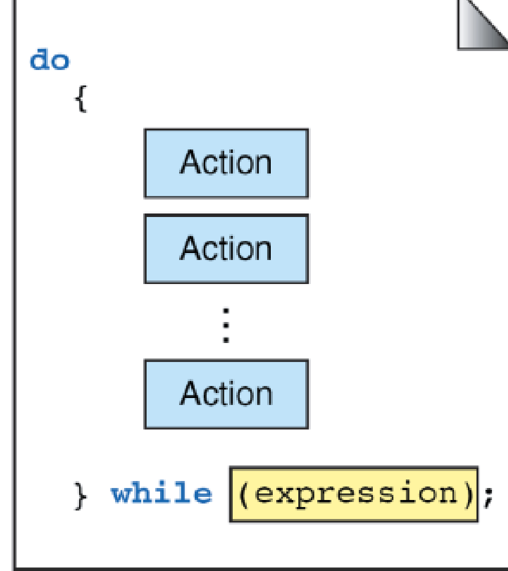
do~while 문

- Post-test Loop
- 조건문이 만족할 동안 Statements를 실행
- 최초 1번은 반드시 실행됨

```
do
{
    //statement;
    //statement;
        ...
} while(condition);
```

S-> C -> S -> C -> S -> C -> loop out





Requirement가 필요 없는 경우

- Requirement들이 필요에 따라 없을 수도 있다.
- Initialization 없는 경우

  반복문을 위한 특별한 변수가 필요 없을 때

- Condition 없는 경우

  그냥 무조건 참. 무한 루프 ex) while()

- Update 없는 경우

  음…Think…

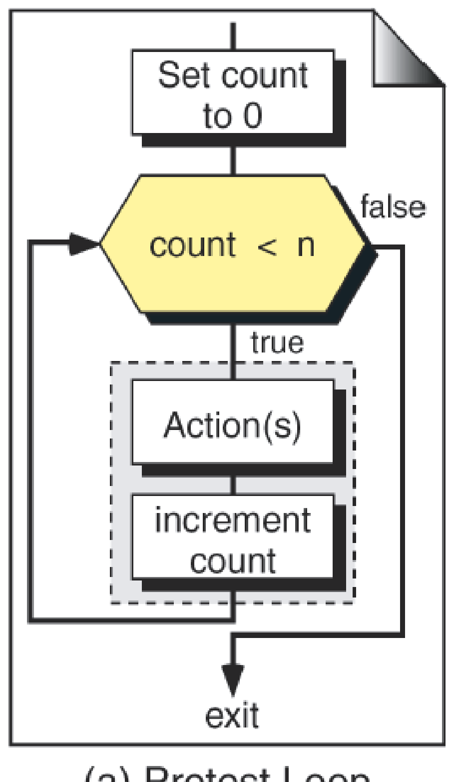
for and *while* as Perpetual Loops

```
1  // A bad loop style
2  for ( ; ; )
3     {
4      …
5      if (condition)
6          break;
7     } // for
```

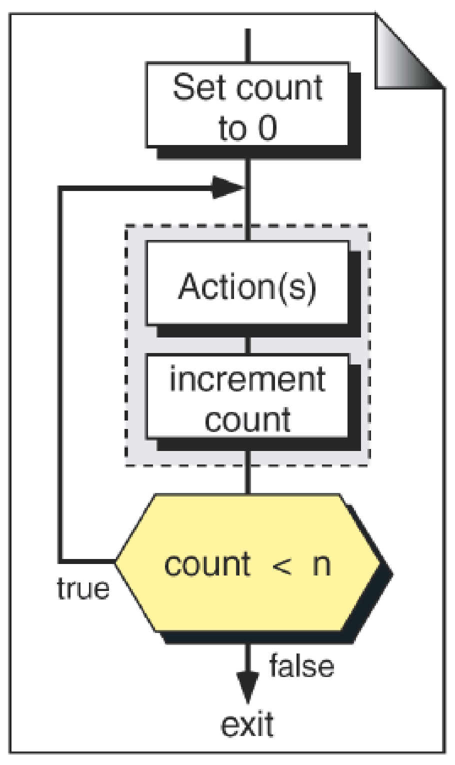
```
// A better loop style
for ( ; !condition ; )
    {
      …
    } // for
```

```
1  while (x)
2     {
3      …
4      if (condition)
5          break;
6      else
7          …
8     } // while
```

```
while  (x && !condition)
    {
     …
     if (!condition)
         …;
    } // while
```



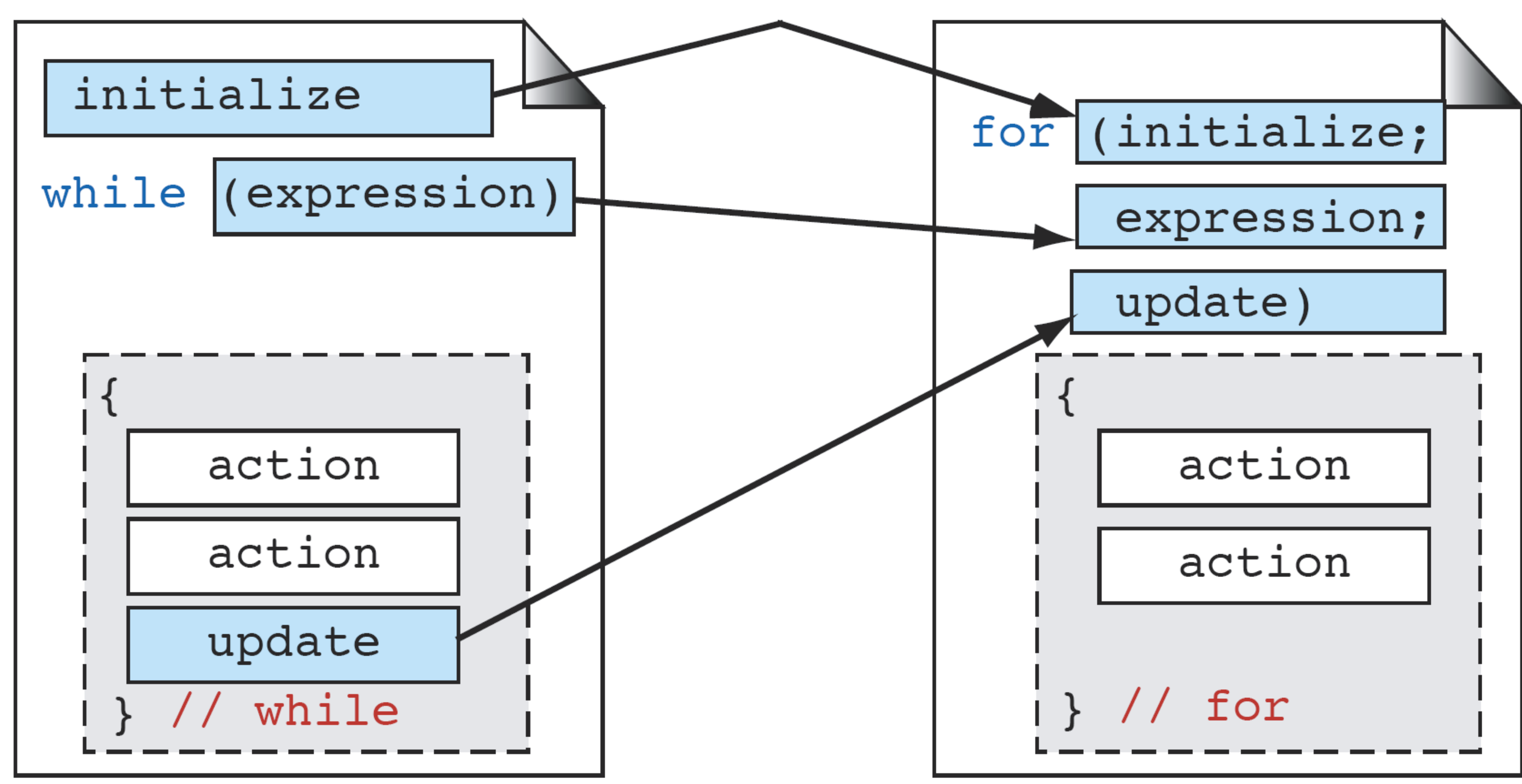
(a) Pretest Loop



(b) Post-test Loop

Pretest Loop		Post-test Loop	
Initialization:	1	Initialization:	1
Number of tests:	$n + 1$	Number of tests:	n
Action executed:	n	Action executed:	n
Updating executed:	n	Updating executed:	n
Minimum iterations:	0	Minimum iterations:	1

for문과 while문 비교



언제 어떤걸 쓰나요

- for: loop 를 몇번 돌아야 하는 지 알고있는 경우 주로 사용
  → "**회만 돌려야겠다."
- While, do~while: 몇번 돌아야 하는 지 모르는 경우 주로 사용
  → "조건 충족 때 까지 돌리고 싶다!!"

- 조금만 고치면 세 개가 다 의미가 같도록 할 수 있기 때문에 사실 원하는 결과가 나오게만 잘 쓰면 되긴함.

for ↔ while ↔ do ~ while

분기문

- Break
  - 가장 가까운 switch문, loop 문 한 개를 나간다.
- Continue
  - loop 문 안에 쓰여서, 바로 Condition 으로 간다.

Break

- 가장 가까운 loop 한 개를 빠져나간다.

```
while (condition)
  {
    …
    for ( …; …; … )
     {
      …
      if (otherCondition)
          break;
      …
     } // for

    // more while processing
    …
  }  // while
```

The *break* statement takes us out of the inner loop (the *for* loop). The *while* loop is still active.

Continue

- for 문에서는 Continue 를 만나면 update -〉 condtion을 거친다.
- while, do~while에서는 Continue 를 만나면 condition만 거친다. 그것밖에 없기도 하고…

```
while (expr)
  {
    ...
   continue;
    ...
  } // while
```

```
do
  {
    ...
   continue;
    ...
  } while (expr);
```

```
for (expr1; expr2; expr3)
  {
    ...
   continue;
    ...
  } // for
```

The Comma Expression

- Complex expression made up of two expressions separated by a comma
- Most often used in for statements

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

```
while (testCount++, loopCount <= 10)
```

```
do
   printf("%3d",    loopCount++);
while (testCount++,  loopCount <= 10);
```

반복문 예제

모양 만들기

1)
0
00
000
0000
00000

2)
00000
0000
000
00
0

3)
0
00
000
0000
00000

4)
00000
0000
000
00
0

5)
0
000
00000

6)
00000
000
0

모양 만들기 1

```
#include <stdio.h>
int main(void) {
    int i=0, j=0, k=5;
    while(i<5) {
        while(j<=i) {
            printf("O");
            j++;
        }
        printf("\n");
        i++;
        j=0;
    }
    return 0;
}
```

```
1)
O
OO
OOO
OOOO
OOOOO
```

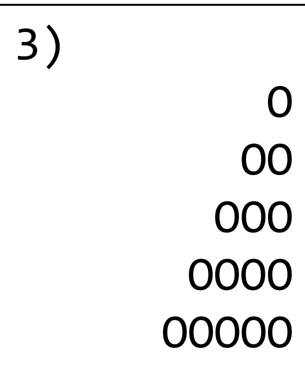
모양 만들기 2

```
#include <stdio.h>
void main() {
    int i=0, j=5, k=5;
    while(i<5) {
        while(j>i) {
            printf("O");
            j--;
        }
        printf("\n");
        i++;
        j=5;
    }
}
```

```
2)
OOOOO
OOOO
OOO
OO
O
```

모양 만들기 3

```
void main() {
  int i=0, j=4, k=0;
  while(i<5) {
        while(j > i) {
                printf(" "); j--;
      }
        while(k <= i) {
                printf("O"); k++;
      }
        i++; j=4; k=0; printf("\n");
        }
}
```



모양 만들기 4

```
void main() {
    int i=0, j=5, k=5;
    while(i<5) {
        while(j<i) {
            printf(" "); j++;
        }
     while(k>i){
         printf("O"); k--;
     }
        printf("\n"); i++, j=0, k=5;
    }
}
```

```
4)
        OOOOO
         OOOO
          OOO
           OO
            O
```

모양 만들기 5

```
void main() {
    int i=0, j, k=0, temp=0, row;
    scanf("%d",&row);
    while(i<row) {
        j=row;
        while(j-1 > i) {
            printf(" "); j--;
        }
     while(k <= temp) {
        printf("O"); k++;
     }
        printf("\n"); i++, k=0, temp+=2;
    }
}
```

5)

3입력

O
OOO
OOOOO

모양 만들기 6

```
void main() {
  int i=0, j, k=0, temp=0, row;
  scanf("%d",&row);
  temp = row*2-1;
  j = row-1;
     while(i<row) {
      while(i>j) {
              printf(" "); j++;
          }
       while(k<temp){
           printf("O"); k++;
       }
         printf("\n");
         i++, k=0,j=0, temp-=2;
    }
}
```

```
6)
   3입력

  OOOOO
   OOO
    O
```

함수

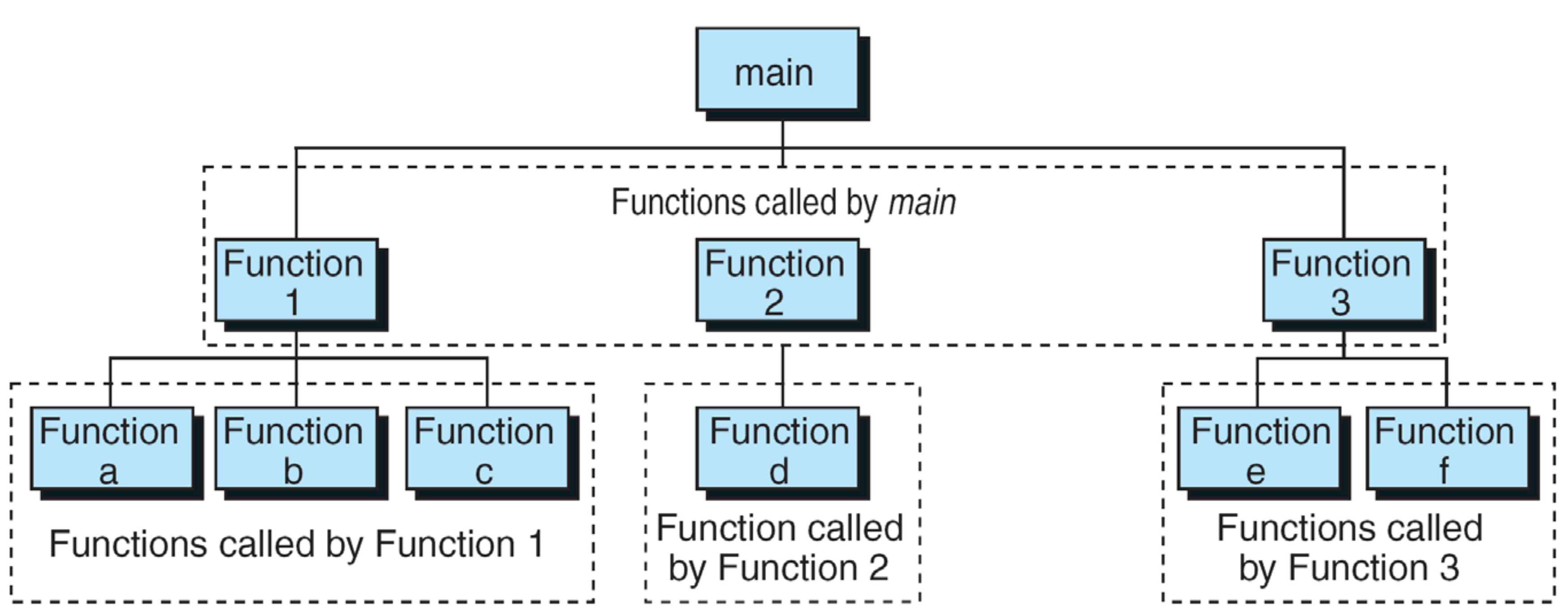
함수

Zero or more data
values can be passed.

Side Effects

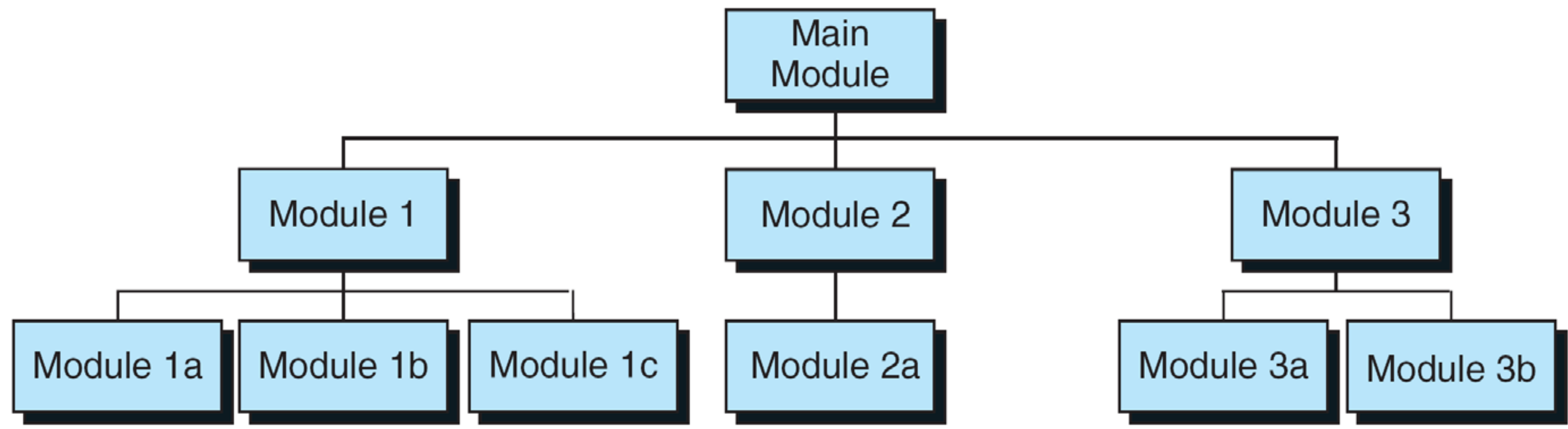
At most one data value
or structure can be returned.

C는 한 개 이상의 함수로 구성된다



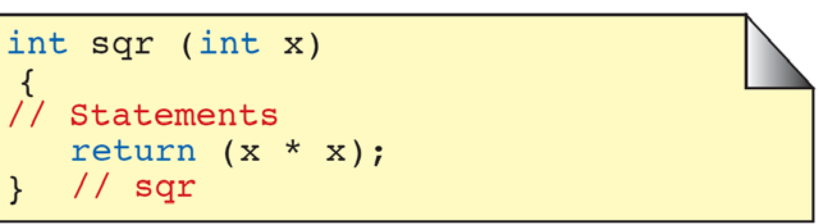
함수를 왜 만들죠

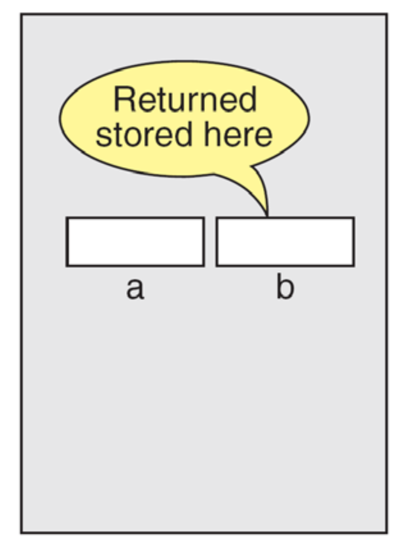
- 함수란 특정한 작업을 수행하도록 독립적으로 작성된 프로그램
- 프로그램에서 반복적으로 수행되는 부분을 함수로 작성하여 필요할 때마다 호출하여 사용
- 코드의 분할과 재사용

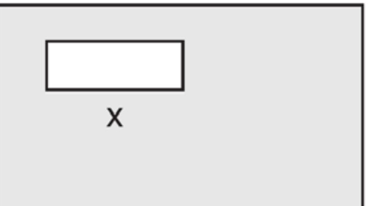


Calling a Function That Returns a Value

```c
// Function Declaration
int sqr (int x);
int main (void)
{
// Local Declarations
   int a;
   int b;
// Statements
   scanf("%d", &a);
   b  =  sqr (a);
   printf("%d squared: %d\n", a,  b);
   return 0;
}  // main
```








```c
// Function Declaration
int multiply (int multiplier, int multiplicand );
int main (void)
{
   int product;
   …
   product = multiply (6, 7);
   …
   return 0;
}  // main
```

42

```c
int multiply (int x, int y)
{
   return x * y;
}  // multiply
```

x 6
y 7

Function Definition

메모리 공간은 함수마다 따로따로

```
int sqr (int x)
 {
// Statements
   return (x * x);
}  // sqr
```

x

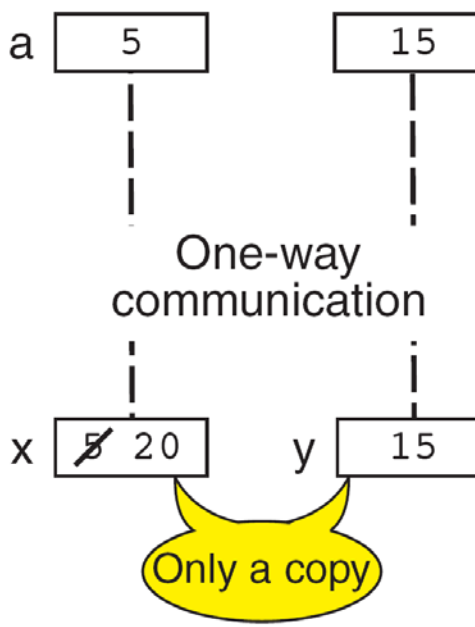

```
double average (int x,int y)
{
   double sum;

   sum = x + y;
   return (sum / 2);

}  // average
```

Call-by-Value

```c
// Function Declaration
void downFun (int x, int y);
int main (void)
{
// Local Definitions
   int a = 5;
// Statements
   downFun (a, 15);
   printf("%d\n", a);
   return 0;
}  // main
```

prints 5

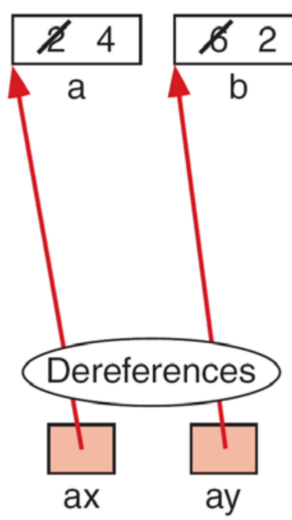
```c
   void downFun (int x, int y)
   {
   // Statements
      x = x + y;
      return;
   }  // downFun
```



Call-by-Reference

```
// Function Declaration
void biFun (int* ax, int* ay);

int main (void)
{
// Local Definitions
   int  a = 2;
   int  b = 6;

// Statements
   …
   biFun (&a, &b);
   …
 return 0;
}  // main
```

```
void biFun (int* ax, int* ay)
{
   *ax = *ax + 2;
   *ay = *ay / *ax;
   return;
}  // biFun
```



SWAP

```
// Function Declarations
void exchange (int* num1, int* num2);
int main (void)
{
// Local Definitions
   int  a;
   int  b;

// Statements
   …
   exchange (&a, &b);
   …
 return 0;
}  // main
```

```
void exchange (int* num1, int* num2)
{
// Local Definitions
   int  hold;

// Statements
   hold   =  *num1;
   *num1  =  *num2;
   *num2  =   hold;
   return;
}  // exchange
```

Scope

```
/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include  <stdio.h>                              Global area
int fun (int a, int b);

  int main (void)
  {
    int    a;                                    main's area
    int    b;
    float y;
    …
      {  // Beginning of nested block
        float a = y / 2;
        float y;                                 Nested block
        float z;                                    area
        …
        z = a * b;
        …
      }  // End of nested block
    …
  }  // End of main

  int fun (int i, int j)
  {
    int a;                                       fun's area
    int y;
    …
  }  // fun
```

- 중괄호로 싸여져 있는 코드 블럭, 코드 범위
- Scope별로 변수를 만들 수 있다.
- Scope 내에서 선언된 변수는 해당 Scope 안에서만 존재한다.
- 자기보다 상위 Scope의 변수는 볼 수 있지만 하위 Scope의 변수는 볼 수 없다.
- 바깥 Scope, 지금 Scope 에 같은 이름의 변수가 있으면 현재 Scope 에 있는 변수를 본다.

오늘 할 것

- 재귀함수(recursive)
- Array + Search
- File I/O

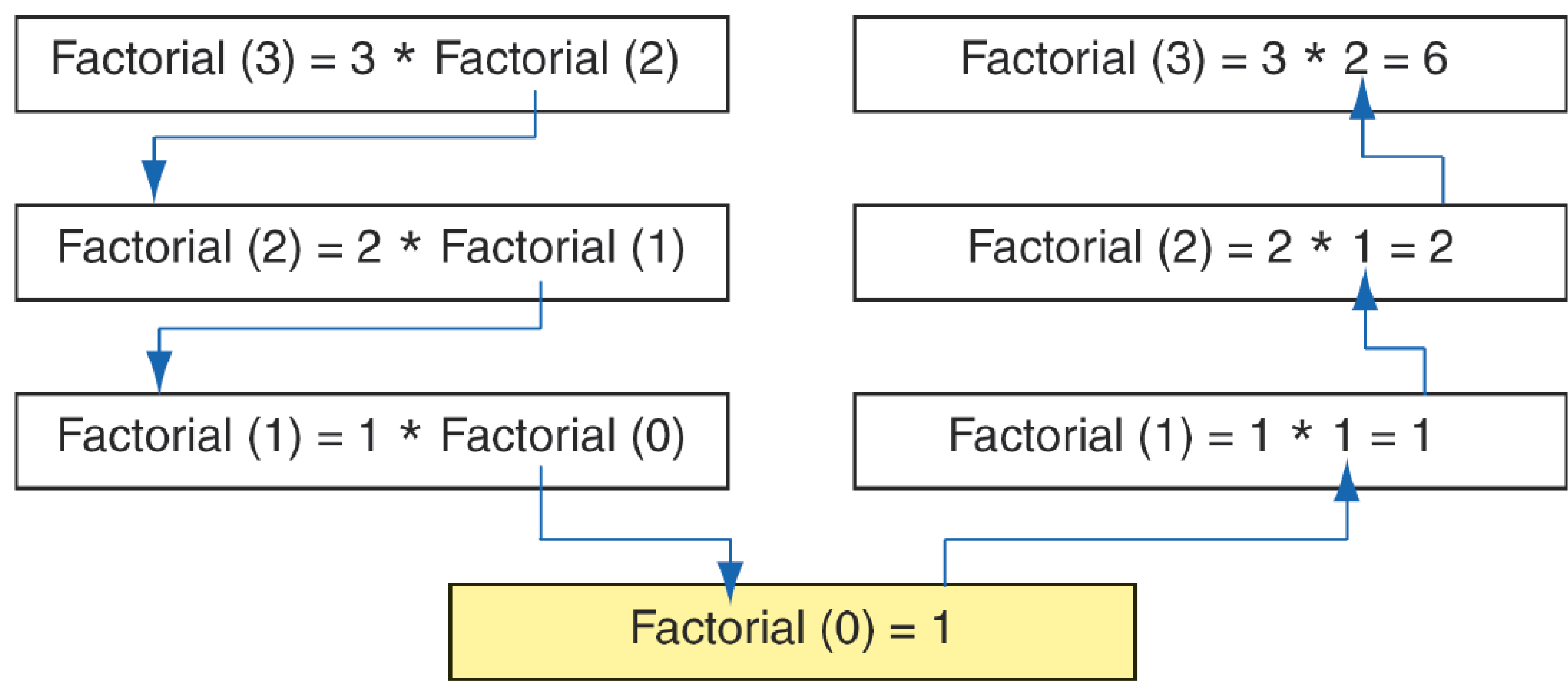
Recursion

Iterative vs Recursive

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

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

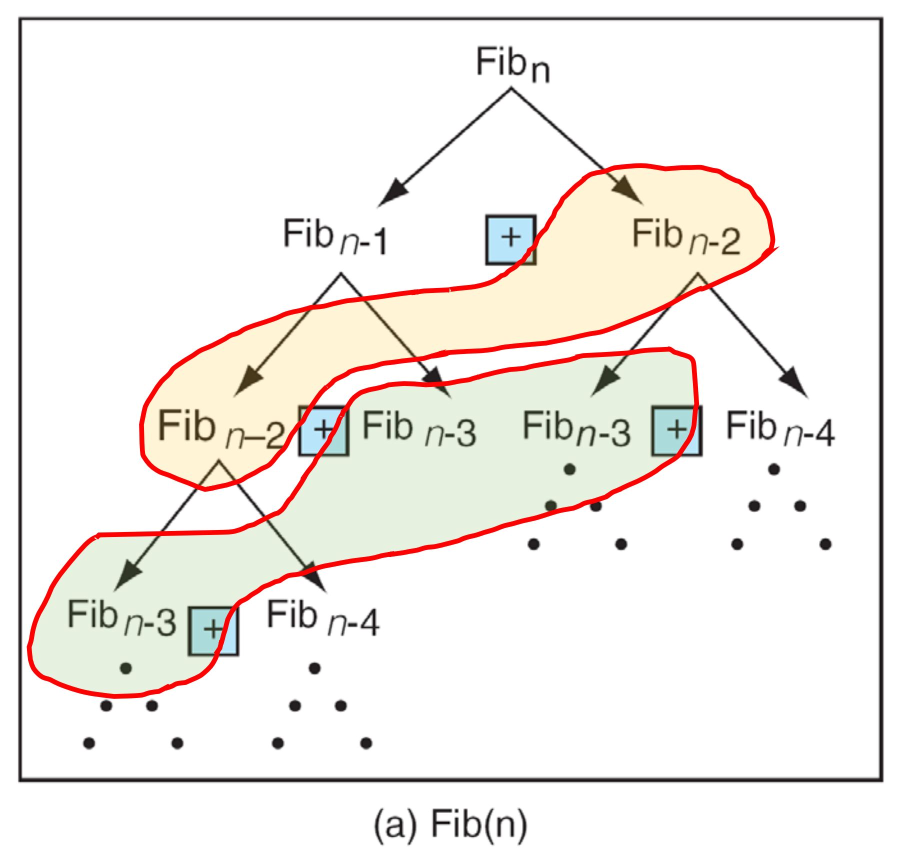
Factorial Recursively



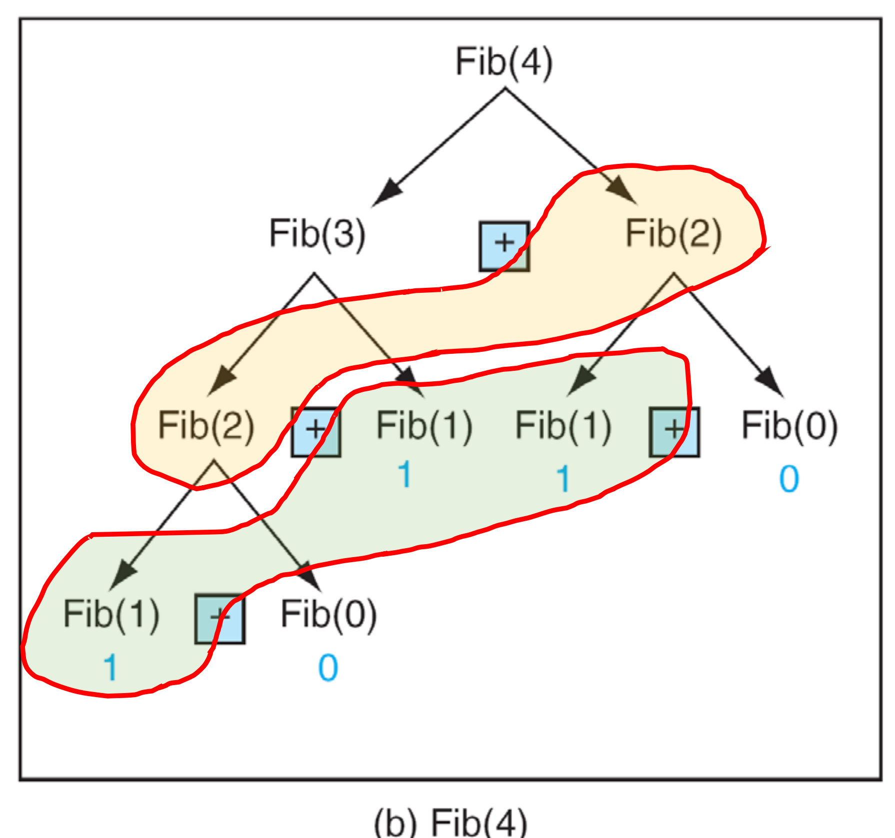
Limitations of Recursion

- Recursive solutions may involve extensive overhead because of function calls
- Each recursive call uses up some of memory allocation
- Possibly duplicate computation, but not always

Duplicate computation on Fibonacci



(a) Fib(n)

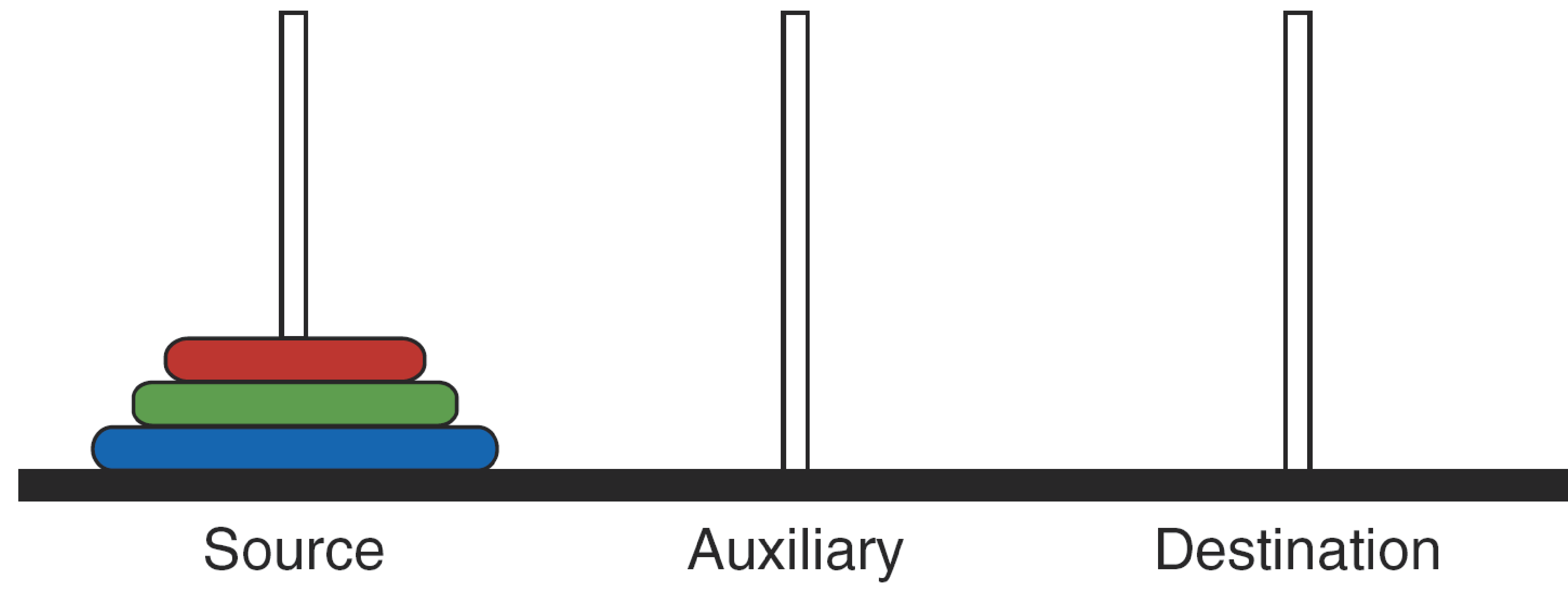


(b) Fib(4)

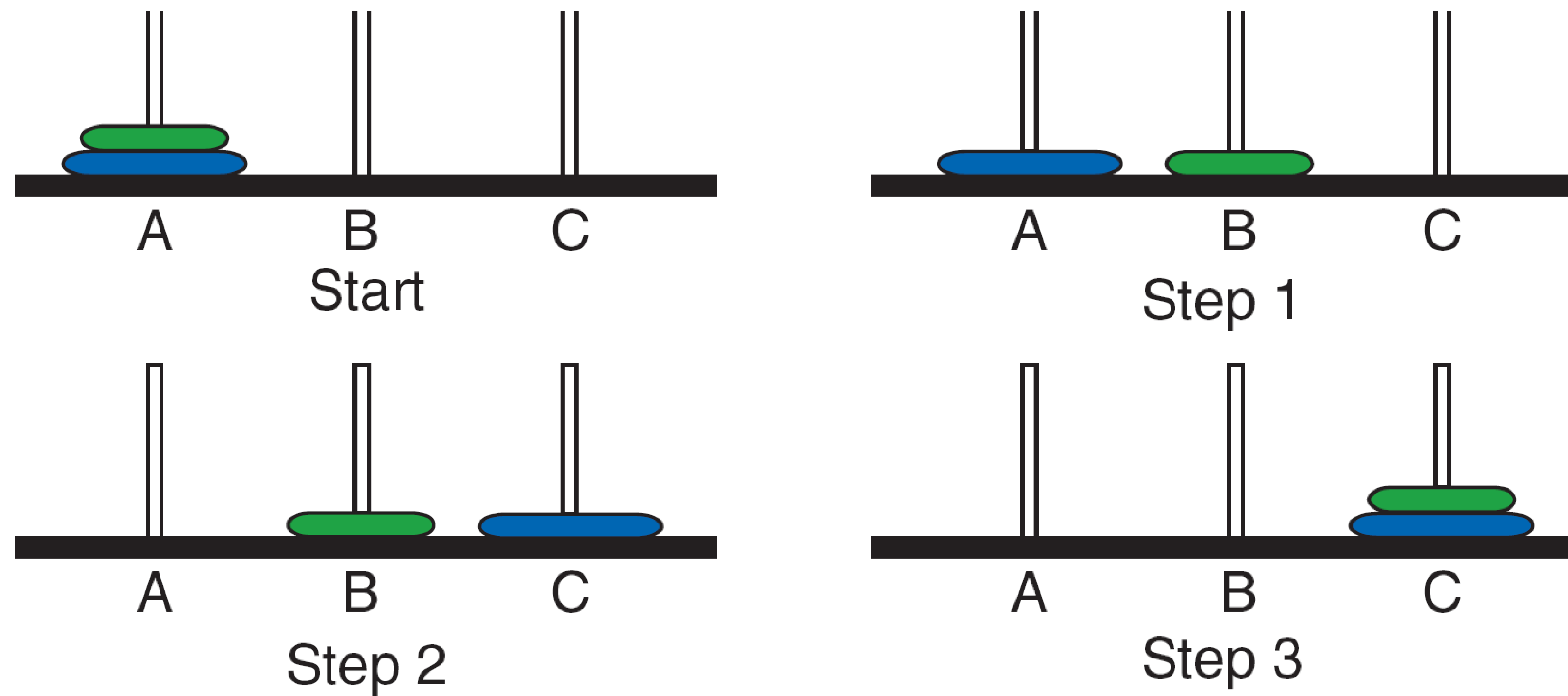
Fibonacci Run Time

No.	Calls	No.	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1,219
5	15	15	1,973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

Tower of Hanoi



Tower of Hanoi



Recursion 풀기

필수조건

- Base Case (n=1)
- n=k → n=k+1?

왜 쓰나요?

- 장점: 문제를 단순하게 풀 수 있다
- 단점: 메모리 소모가 많다

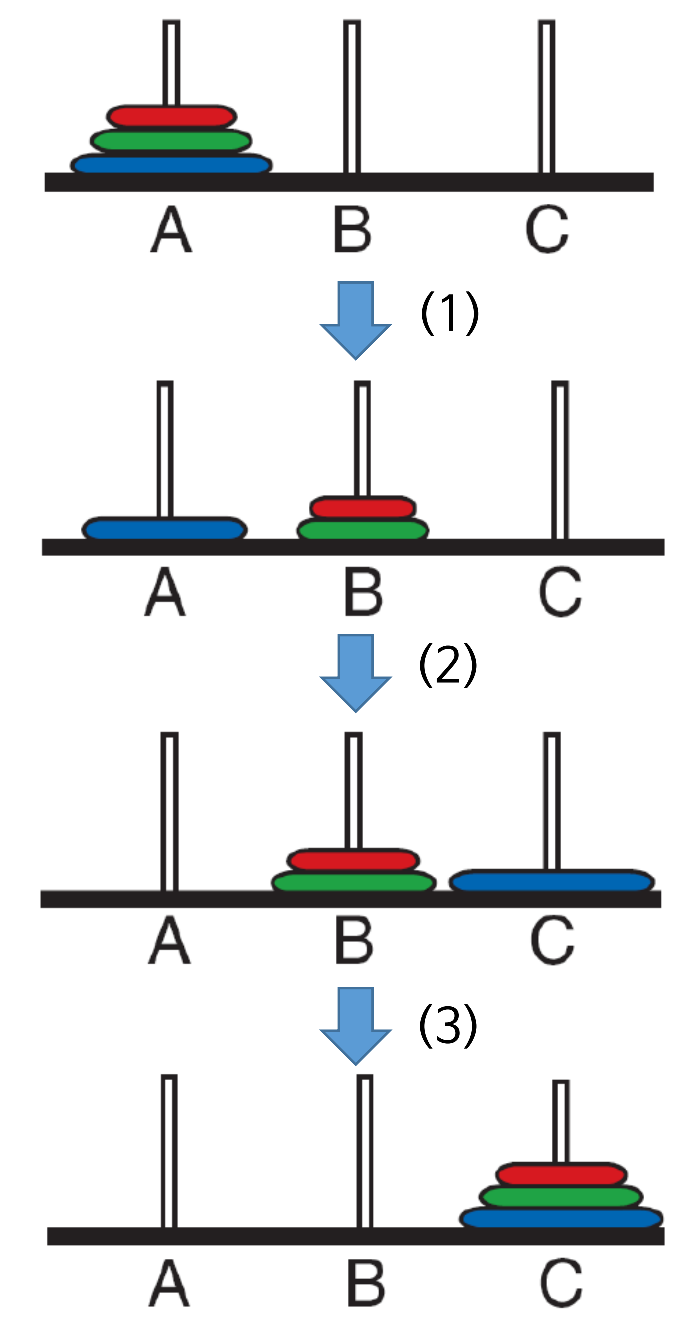
Tower of Hanoi

void tower(int n, char source, char auxiliary, char dest);

source는 시작점, auxiliary는 중간, dest는 목적지

- Base case: if(n==1) source → destination
- else:

```
tower(n-1, source, dest, auxiliary);   (1)
source → destination;                  (2)
tower(n-1, auxiliary, source, dest);   (3)
```

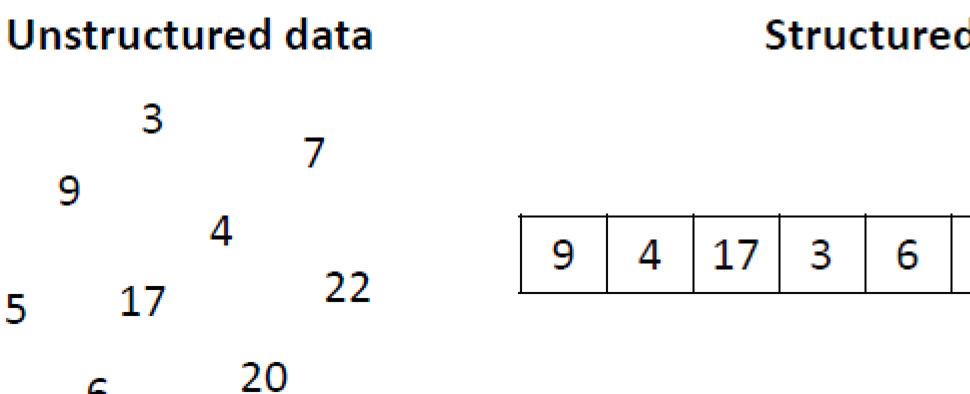


Array 맛보기

자료구조 Data Structure

- A particular way of **storing and organizing data** in a computer so that it can be **used efficiently**.
- 많은 양의 데이터를 효과적으로 관리, 사용하기 위해 (ex. 인터넷 검색)
- 좋은 자료구조는 데이터를 빠르게 검색할 수 있다.

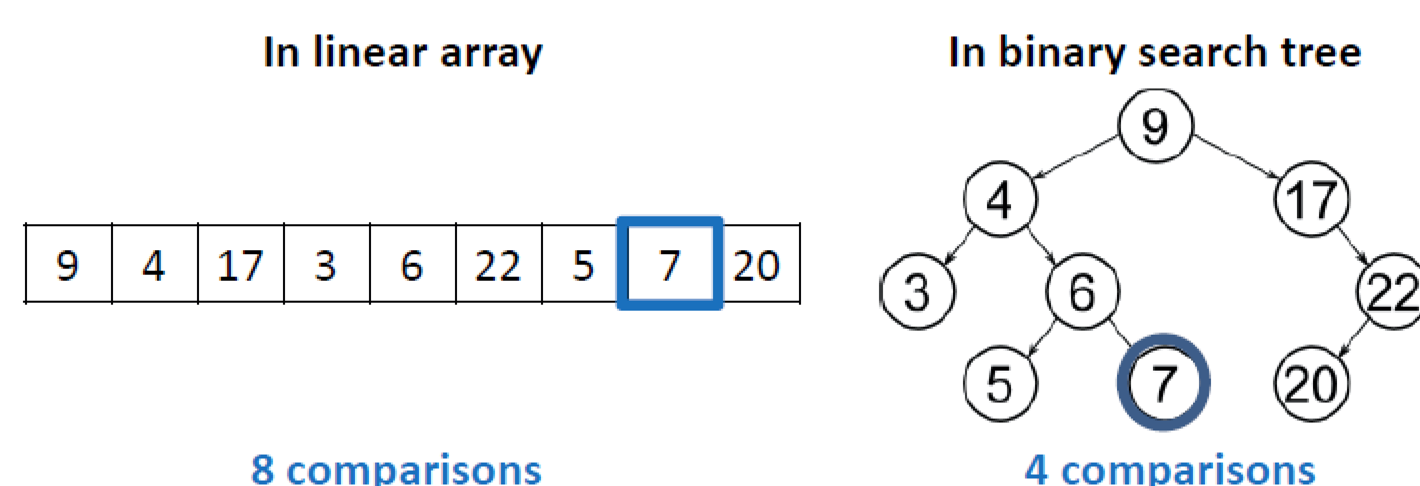
- 종류
  - Matrices
  - Linked lists
  - Priority queues
    - stack, queue, etc.
  - Hash tables
  - Trees and graphs



왜 쓰나요

- 많은 데이터를 예쁘게 잘 저장해서
- 쉽고 빠르게 꺼내 쓰려고
- 데이터가 많아지면 찾는데도 오래 걸린다.

- Example: finding a number from a set of numbers
  - How many comparisons do we need to retrieve 7?



Array

Array가 뭔가요

- 연관성 있는 변수들을 하나로 묶어서 보관하는 데이터 타입

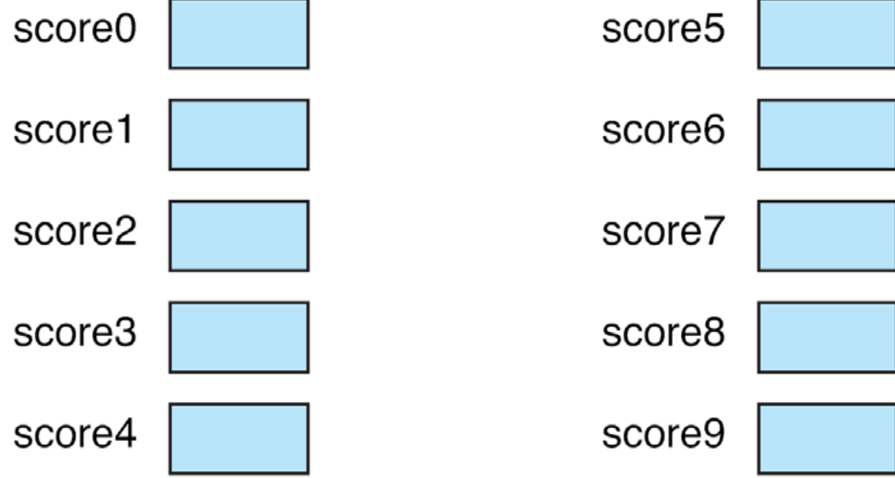
(a) Subscript Format (b) Index Format

Array를 왜 쓰나요?

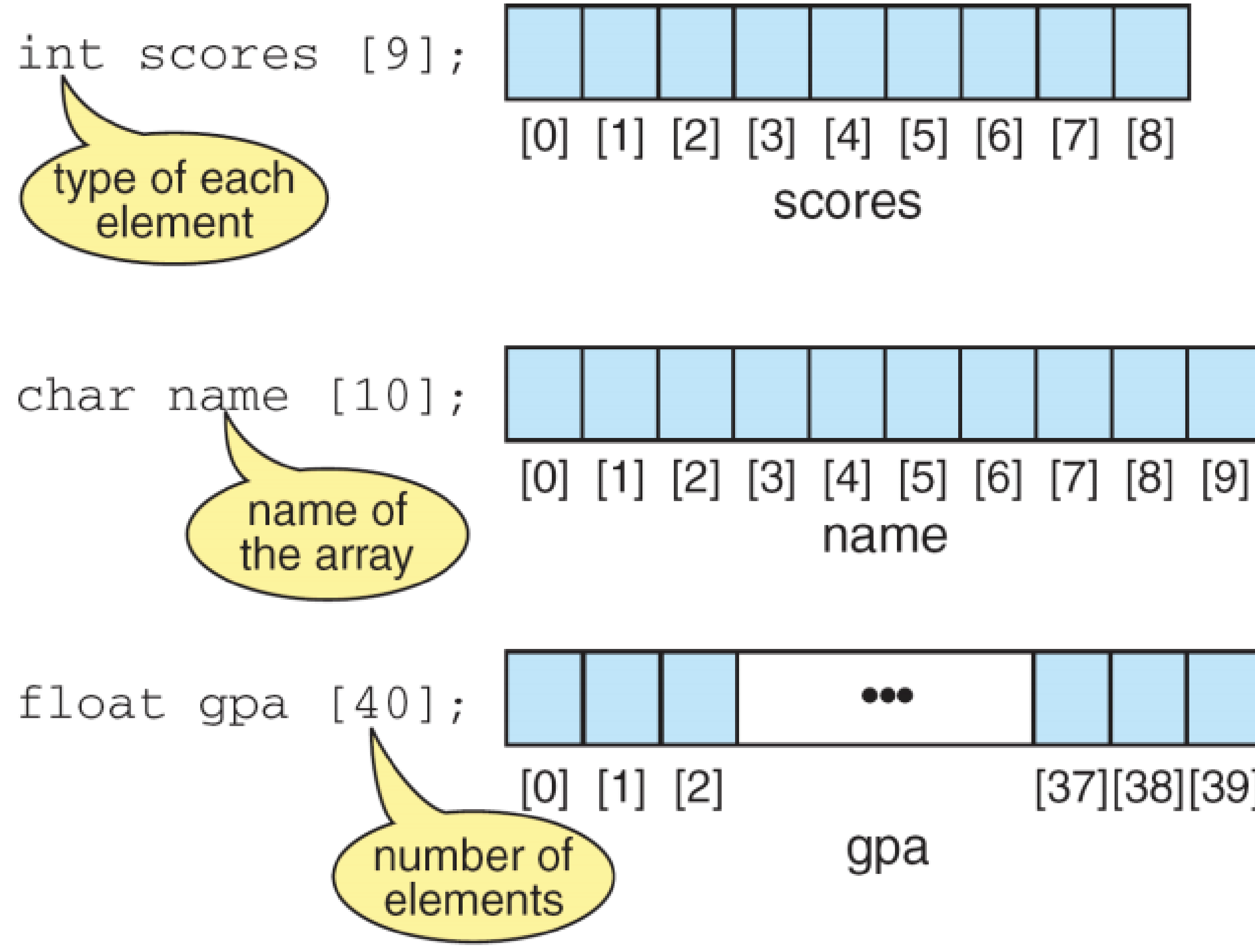
- 여러개의 데이터를 저장해야 할때!
  - 변수를 일일이 만들건가?

- 그렇게 일일이 만들었다고 쳐!
  - 그 여러개에 저장된거를 어떻게 가져올 것인가?

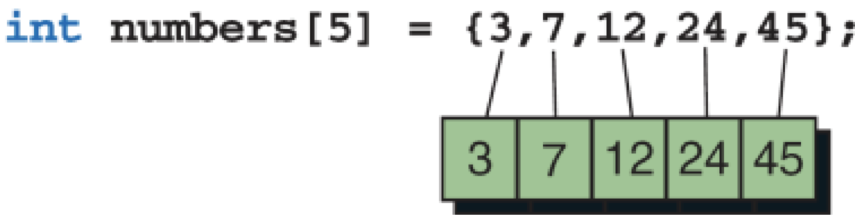
Array!

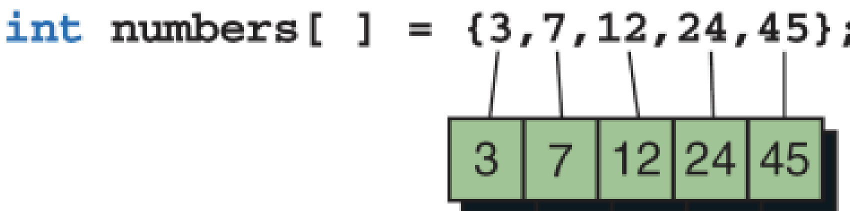


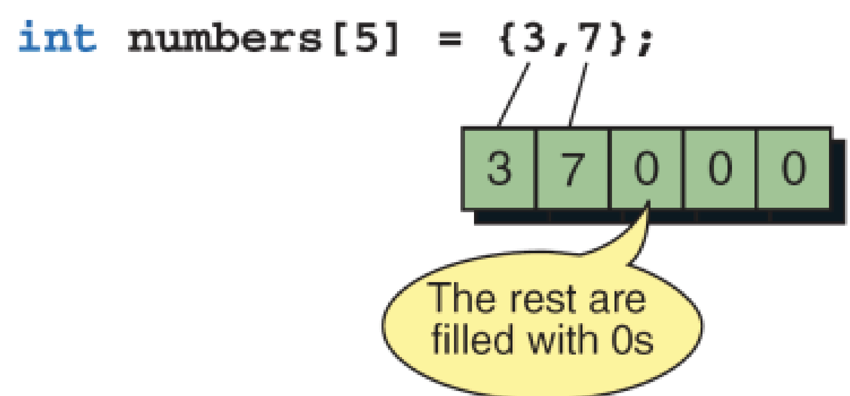
선언

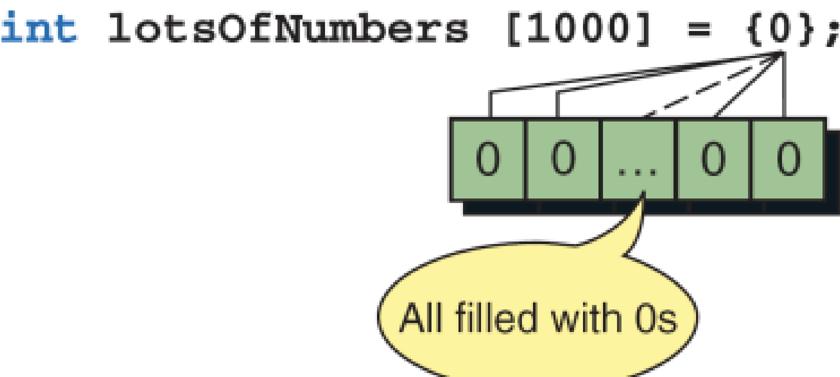


초기화









접근

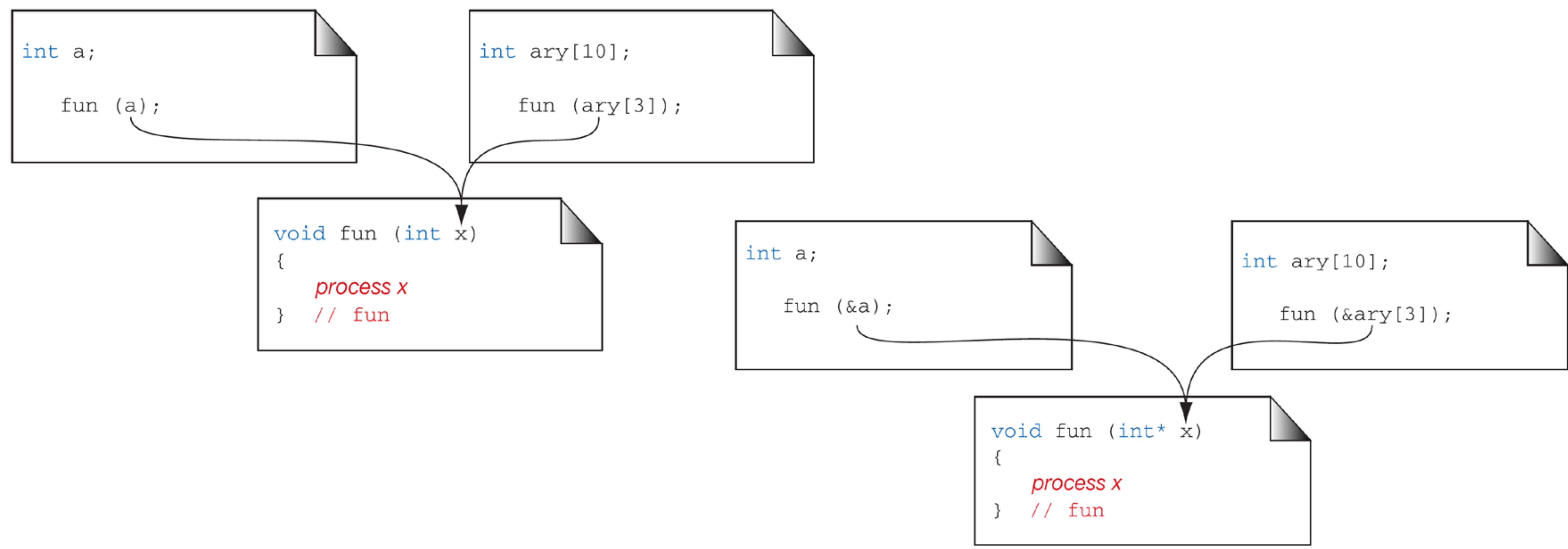
name [index] : 배열은 0부터 시작하는 것 주의

```
for (i = 0; i < 9; i++)
   scanf (“%d”, &scores[i]);

scores[4] = 23;
printf(“%f”,gpa[11]);    //배열 gpa의 ‘12번째’ 실수 출력
```

Element address = array address + (sizeof (element) * index)

함수의 인자로 배열 넘기기

- 똑같이 하면 됩니다.



함수의 인자로 배열 넘기기 #2

- 배열 전체를 넘길때

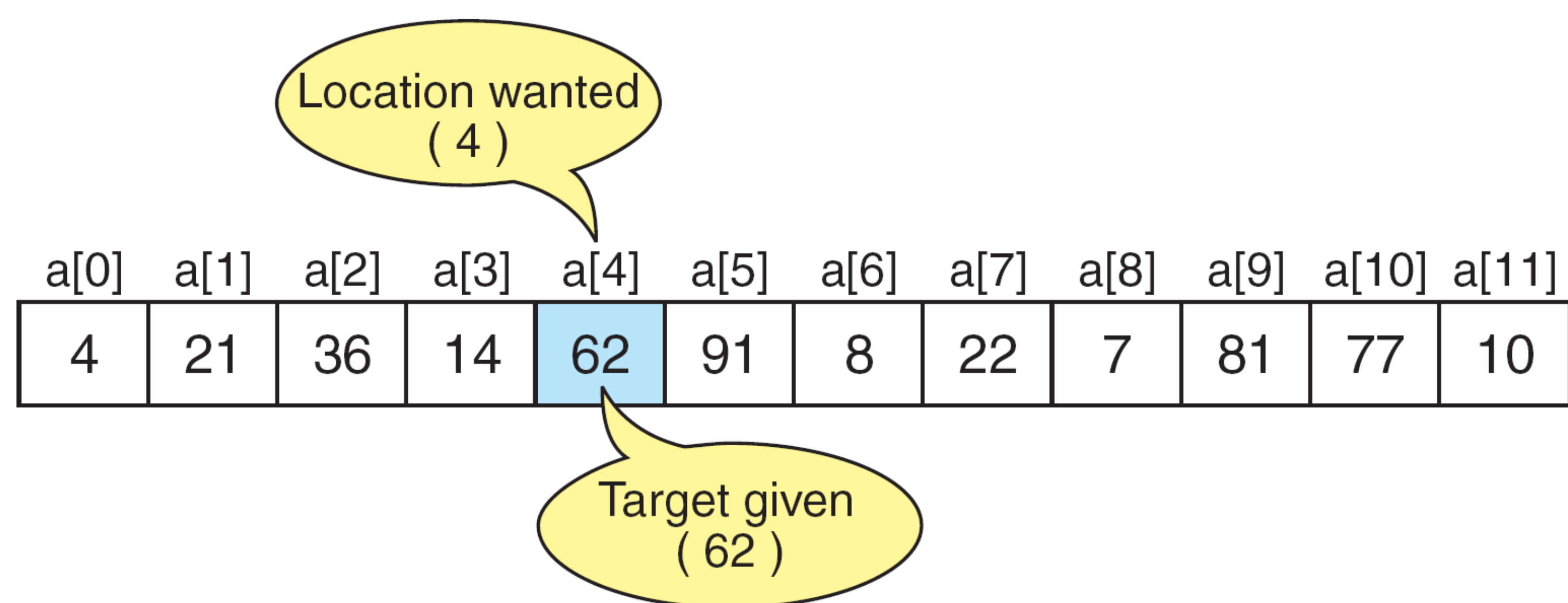
```
int ary[10];

   fun (ary);
```

```
void fun (int fAry[ ])
{
     process x
}   // fun
```

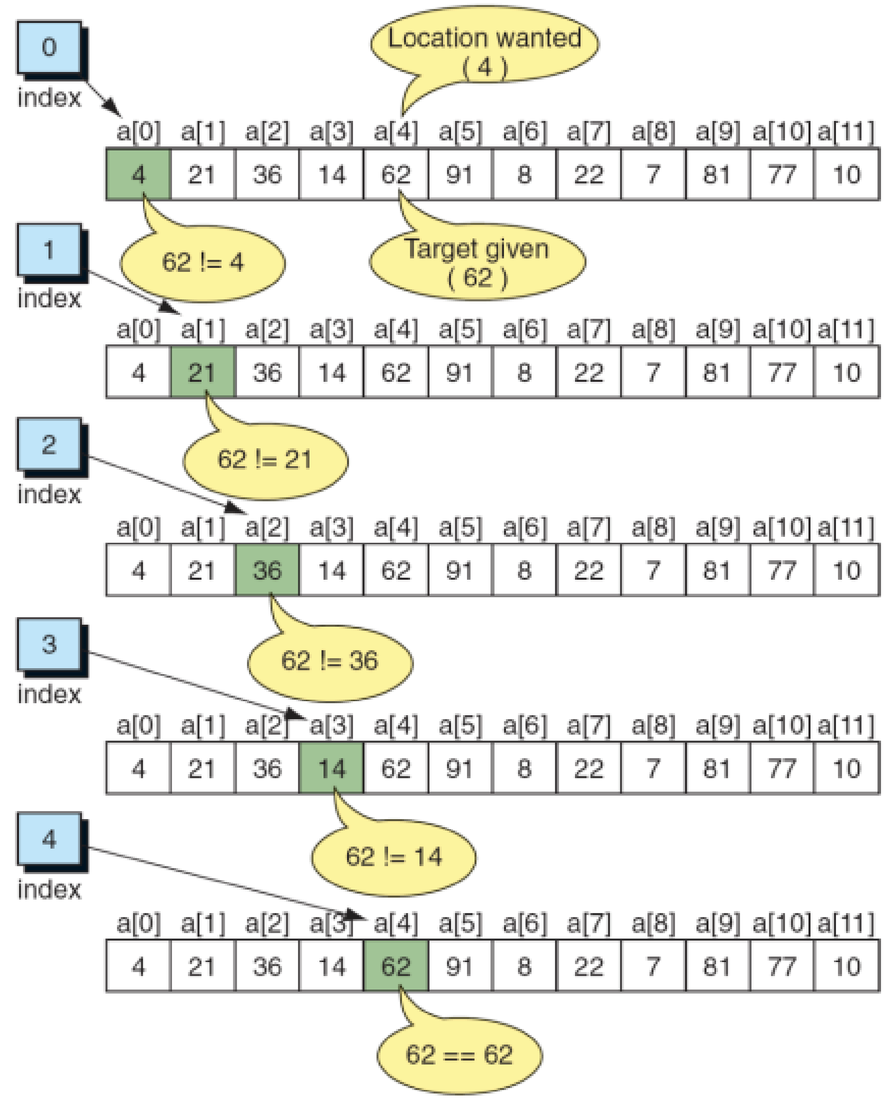
Fixed-size Array

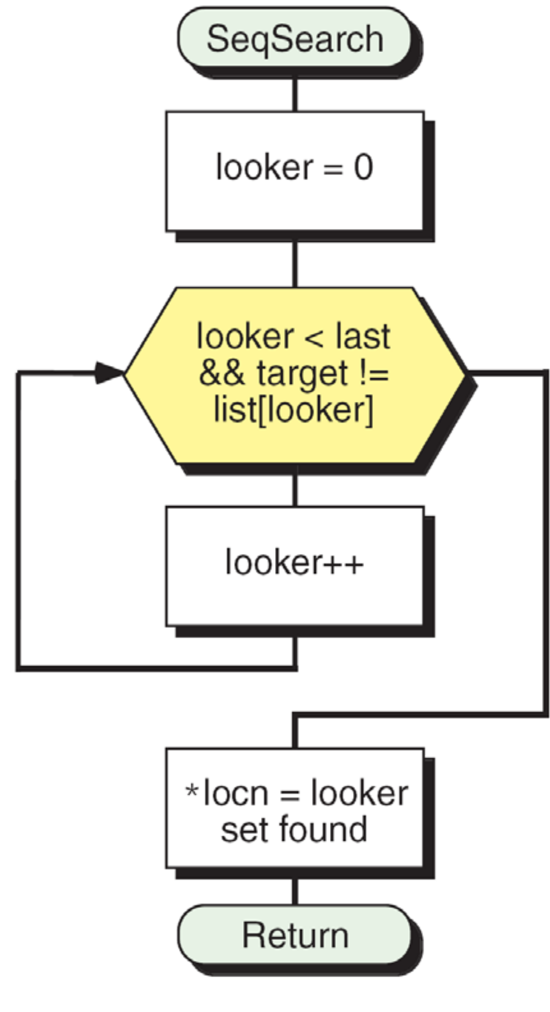
Search

Searching on Array



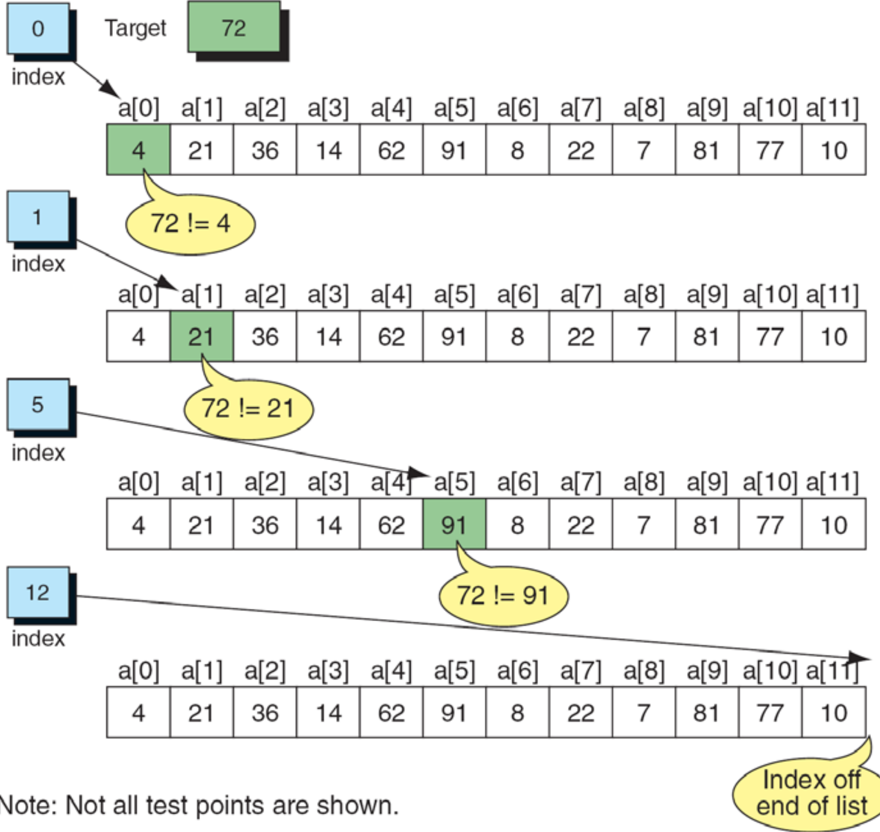
Sequential Search Design





Sequential Search 특징

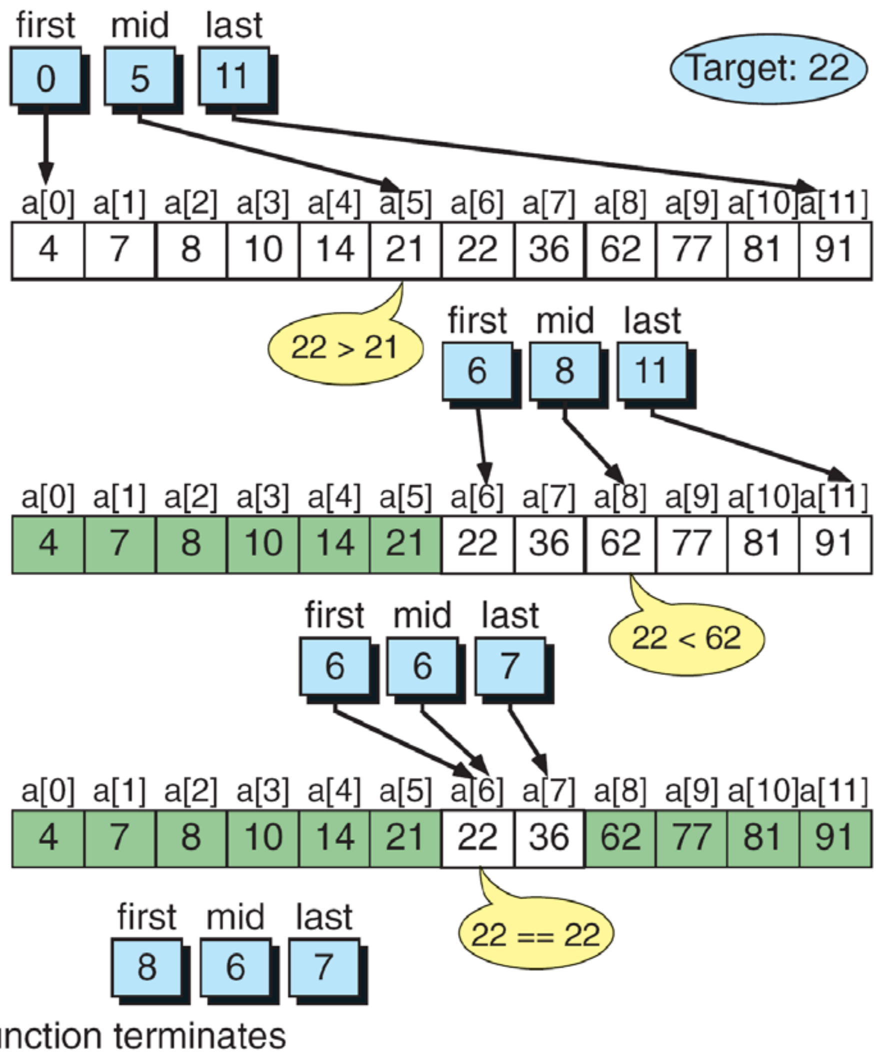
- Unsorted 배열에 대해서는 처음부터 쭉 봐야하기 때문에 비효율적이다.
- Worst Case: O(n)



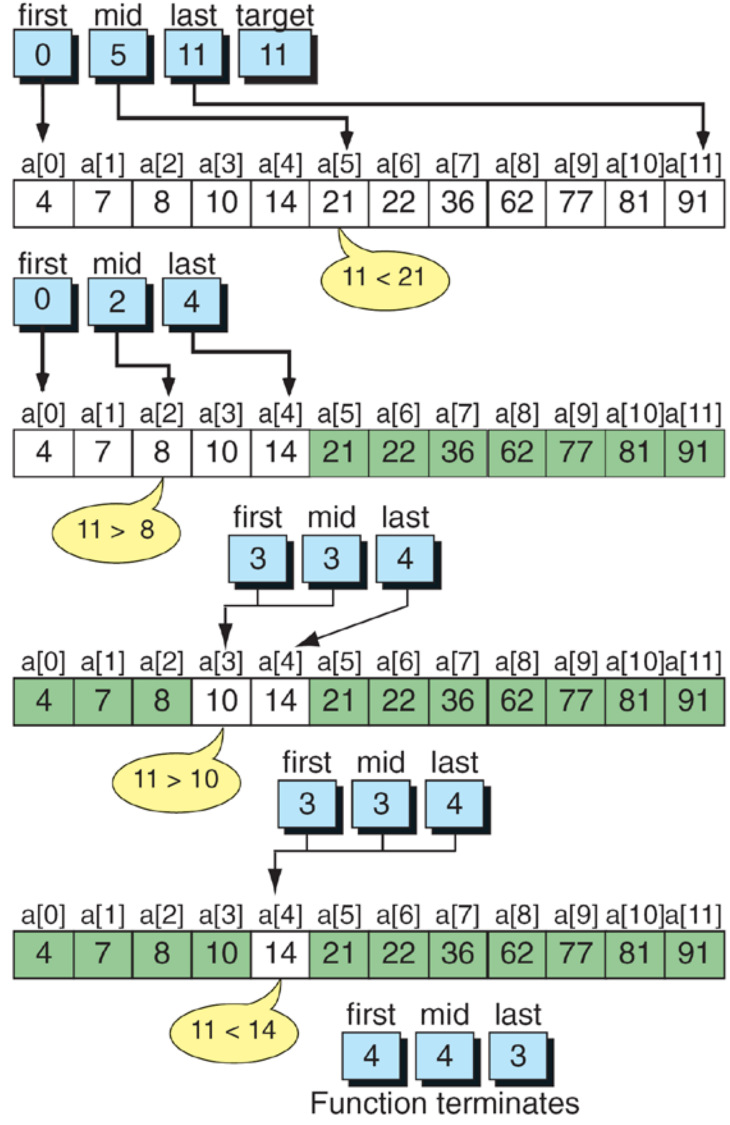
Code

```
1  /* ==================== seqSearch ====================
2     Locate target in an unordered list of size elements.
3        Pre   list must contain at least one item
4              last is index to last element in list
5              target contains the data to be located
6              locn is address for located target index
7        Post  Found: matching index stored in locn
8                     return true (found)
9              Not Found: last stored in locn
10                        return false (not found)
11 */
12 bool seqSearch (int list[], int  last,
13                 int target, int* locn)
14 {
15 // Local Declarations
16    int  looker;
17    bool found;
18
19 // Statements
20    looker = 0;
21    while (looker < last && target != list[looker])
22       looker++;
23
24    *locn = looker;
25    found = (target == list[looker]);
26    return found;
27 } // seqSearch
```

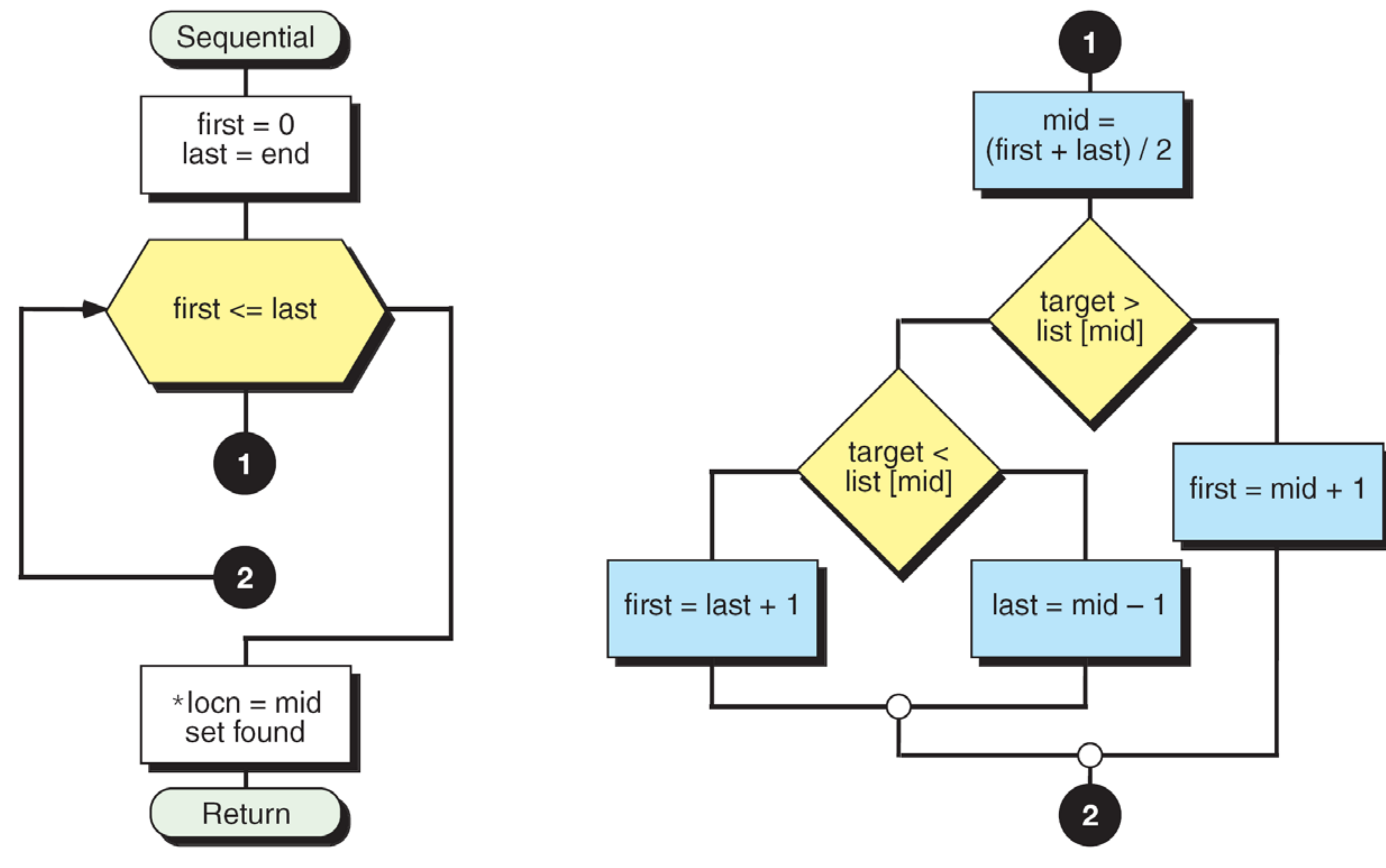
Binary Search



Unsuccessful Binary Search Example



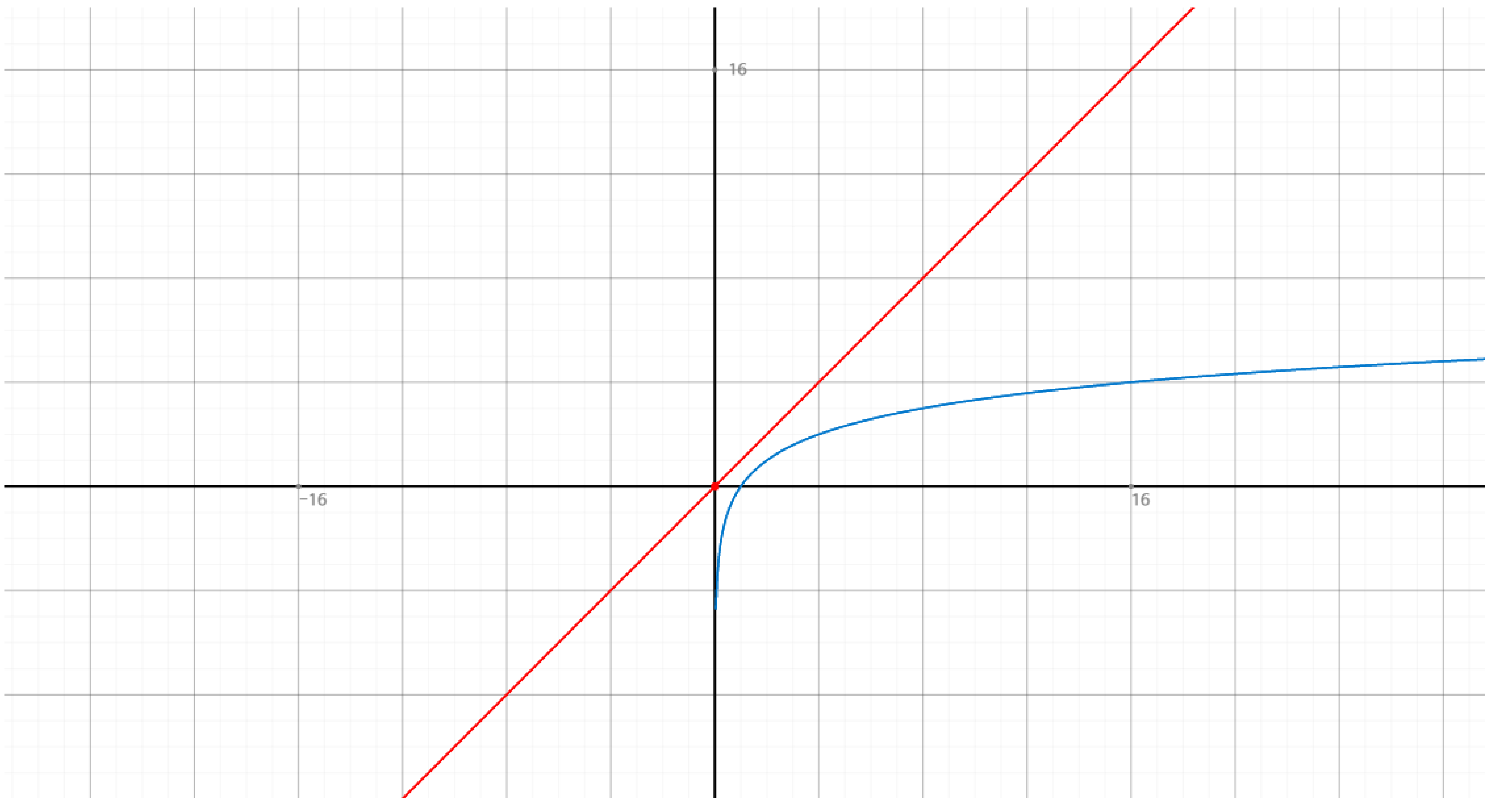
Binary Search Design



Binary Search 특징

- 검색 대상 배열은 sorted 인 상태여야 한다.
  - “정렬된 데이터에 대해서는 이진검색이 빠르다”
- 매 iteration에서 절반(1/2)을 서치 후보에서 제외시킨다. (제외된 거는 볼 필요 없음)
- O(logn)

O(n) vs O($\log_2$n)



Code

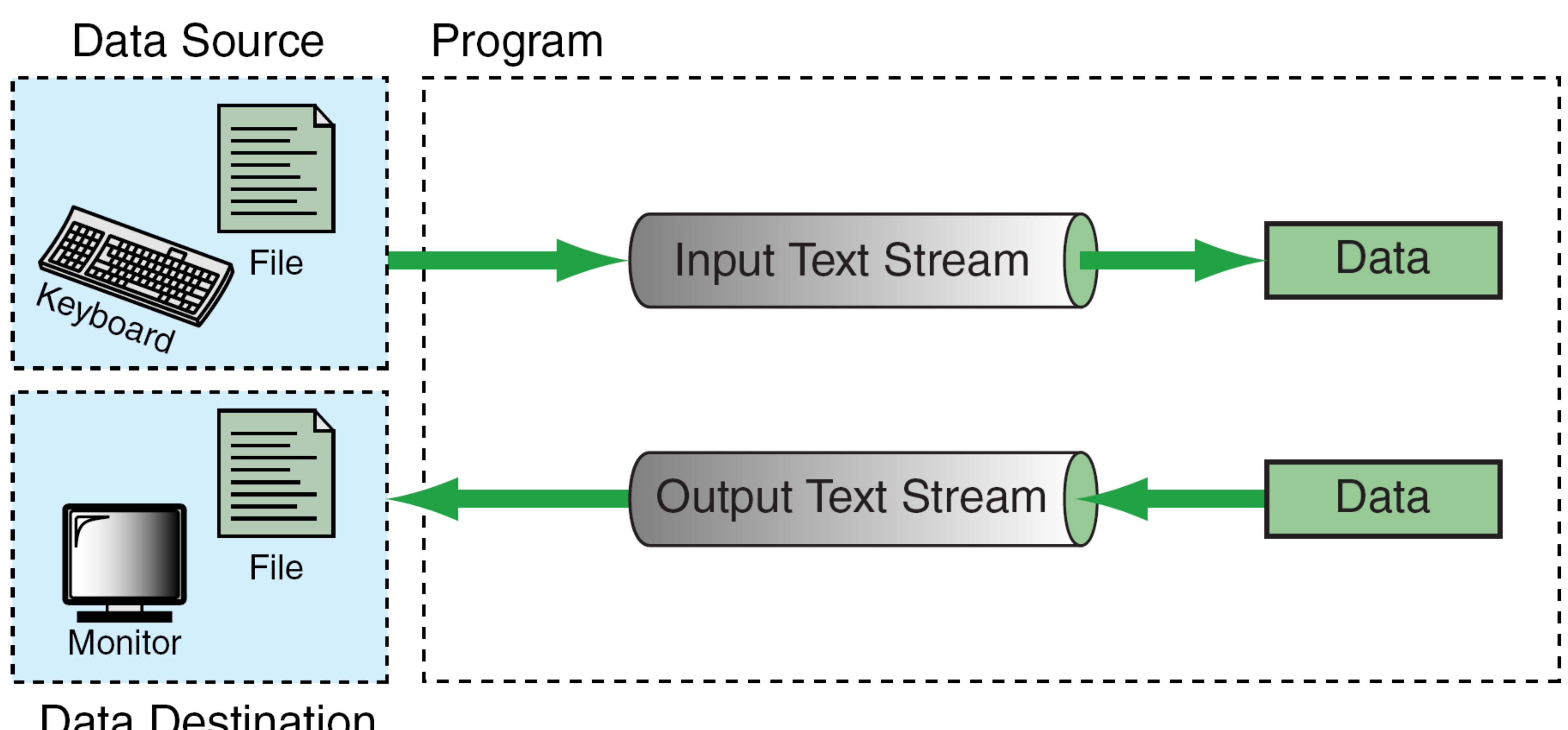
```
1  /* ==================== binarySearch ====================
2     Search an ordered list using Binary Search
3        Pre   list must contain at least one element
4              end is index to the largest element in list
5              target is the value of element being sought
6              locn is address for located target index
7        Post  Found: locn = index to target element
8                     return 1 (found)
9            Not Found: locn = element below or above target
10                       return 0 (not found)
11 */
12 bool binarySearch (int list[], int  end,
13                    int target, int* locn)
14 {
15 // Local Declarations
16    int first;
17    int mid;
18    int last;
19
20 // Statements
21    first = 0;
22    last = end;
23    while (first <= last)
24       {
25        mid = (first + last) / 2;
26        if (target > list[mid])
27           // look in upper half
28           first = mid + 1;
29        else if (target < list[mid])
30           // look in lower half
31           last = mid - 1;
32        else
33           // found equal: force exit
34           first = last + 1;
35       } // end while
36    *locn = mid;
37    return target == list [mid];
38 }  // binarySearch
```

File I/O

뭘 배우나요

- Stream?
- Standard Input, Output functions
- 파일로 부터 입력 받기/파일에 출력하기
  - FILE*, fopen, fclose, fscanf, fprintf
- FLUSH?
- Character I/O
  - getChar, putChar, fgetc, fputc, ungetc

다시보는 Steam 다이어그램



STanDard I/O (stdio.h)

- Standard Input Stream : stdin
  - Scanf 함수가 사용하는 스트림
  - 콘솔(console) 화면에서 키보드를 통한 입력을 받음
- Standard Output Stream : stdout
  - Printf 함수가 사용하는 스트림
  - 콘솔 화면에 출력함

- 일종의 콘솔 화면과 소통하기 위한 빨대(Stream)!

FILE I/O

- 콘솔(console) 화면과의 소통이 아닌 파일(.txt .c etc) 과 소통하기 위한 Steam

```
FILE *infile;
infile = fopen(“anyfile.txt”, “w”);
~ ~
fclose(infile);
```

```
#include <stdio.h>
…
{
int main (void)
  FILE* spData;
  …
  spData = fopen("MYDATA.DAT", "w");
  …
} // main
```

Internal File Variable

External File Name

FILE

spData

MYDATA.DAT
Physical File

Stream

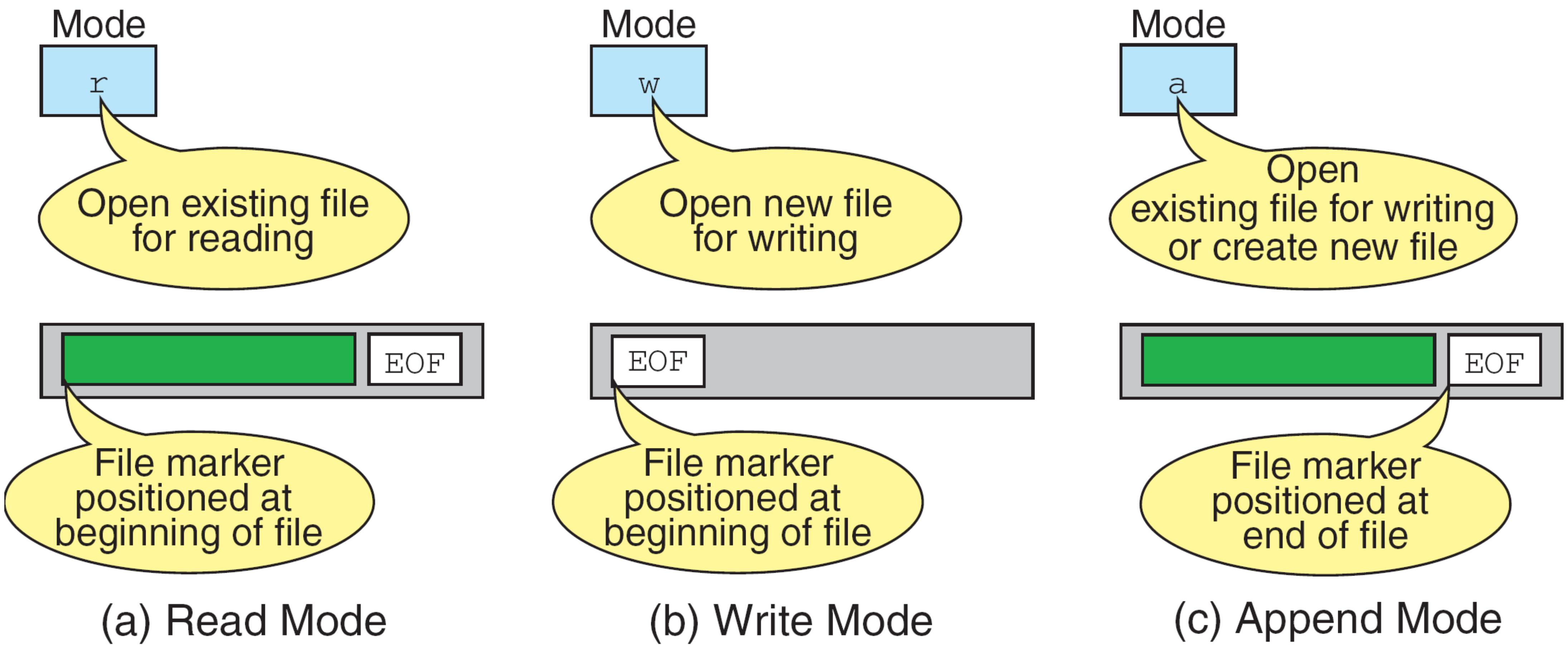
Data

File 사용 options

- r : 읽기 모드 (텍스트 모드)
  - 없으면 널 리턴, 있으면 처음부터 읽기
- w : 쓰기 모드 (텍스트 모드)
  - 기존에 없으면 새로 만들기, 있으면 다 지우고 새로 쓰기
- a : 덧붙이기 모드 (텍스트 모드)
  - 기존에 없으면 새로 만들기, 있으면 제일 뒤에 더하기
- b : 바이너리 모드

```
FILE* infile = fopen("indata.txt", "r");      //읽기로 열기
FILE* outfile = fopen("outdata.txt", "w");    //쓰기로 열기
```

File-Opening Modes



FILE I/O 를 위해 필요한 것

```
//파일을 열기 전에 반드시 파일 포인터 선언
FILE *fp;

//반드시 text.txt 파일이 폴더내에 있어야 한다. (컴파일 된 실행파일이 있는 폴더)
fp = fopen(“test.txt”, “r”);

//파일이 제대로 열렸는지 반드시 체크
if (fp == NULL) { return -1; }

//사용이 끝나면 포인터를 닫아준다.
fclose(fp);
```

대표적 FILE I/O 관련 함수들

```
int fprintf(FILE * out, const char * format, …)
```

Filestream, out 으로 부터 format의 형태로 출력한다.

비교) int printf(const char * format, …)

```
int fscanf(FILE * in, const char * format, …)
```

File stream, in 으로 부터 format의 형태로 입력받는다.

비교) int scanf(const char * format, …)

그외 I/O 관련 함수들

```
int getchar(void)
int putchar(int out_char)

int fgetc(FILE* in)
int fputc (int oneChar, FILE* out)
int ungetc (int oneChar, FILE* Data)
Int fgets (char *str, int num, FILE * in)
Int fputs(const char *str, FILE * out)
```

fprintf

```
FILE * out;
out = fopen(“test.txt”, “w”);
fprintf(out, “%c %d\n”, ‘a’, 22);
…
fclose(out);
```

test.txt

```
1 a 22
~
~
~
~
~
```

fscanf

```
int a;
char b;
FILE * in;

in = fopen(“text.txt”, “r”);
fscanf(in, “%c %d”, &b, &a);
printf(“%c %d\n”, b, a);

fclose(in);
```

FLUSH

- 스트림 버퍼를 비우는 역할을 한다.
- scanf 등 입력받는 함수 사용 시 주의!! 할 점!!
  - 근접한 scanf 두 번 사이에는… 뭔가… 언짢은 일들이…

ex)

```
int a;  char b;
scanf(“%d”, &a);
scanf(“%c”,&b);
printf(“%d %c\n”, a, b);

fflush();
#define FLUSH while(getchar != ‘\n’)
```

예제

- input.txt에 있는 3쌍의 점수를 입력 받아 output.txt에 합계와 평균을 출력
- 점수의 개수는 정해져 있지 않음
- input.txt

```
94 55 78
66 54 70
100 96 95
77 63 88
```

- output.txt

```
234 77.534
210 65.431
294 97.236
223 74.549
```

```
#include <stdio.h>

int main(void) {
        FILE *inFile;
        FILE *outFile;
        int a, b, c;
        int sum;
        float avg;

        inFile = fopen("input.txt", "r");
        outFile = fopen("output.txt", "w");

        if(inFile == NULL) {
                printf(“Input file open error\n");
                return 0;
        }

        while(fscanf(inFile, "%d %d %d", &a, &b, &c) != EOF) {
                sum = a + b + c;
                avg = (float)(sum)/3.0;
                fprintf(outFile, "%d %.3f\n", sum, avg);
        }

        return 0;
}
```

다음시간

- 시험 잘 보세요!
- 질문 있으면 카톡으로 or 청암에서

- 중간고사 끝나고
  Multi-dimentional Array + Sorting, Searching & String