배열과 구조
(Arrays and Structures)

소프트웨어학과

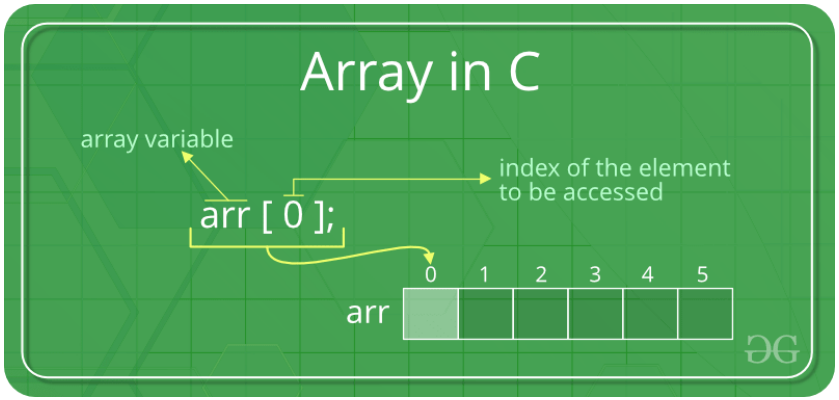
노 서 영

Review – Time Complexity

- **세상을 문제의 집합으로 본다면**
  - 풀 수 있는 문제 (답이 있는 문제) ← 알고리즘의 영역
  - 풀 수 없는 문제 (답이 없는 문제) ← We don't care???
  - 풀 수 있는지 없는지 모르는 문제 (답이 있는지 없는지 모르는 문제) ← Theory of Computation의 영역

- **알고리즘은 풀 수 있는 문제만을 다룬다**
  - 답만 찾으면 되는가?
  - '답'에는 성능이라는 것을 고려해야 한다. ← 유한한 자원
  - 답을 Reasonable time 풀어야 한다 → Polynomial time complexity

배열(Array)



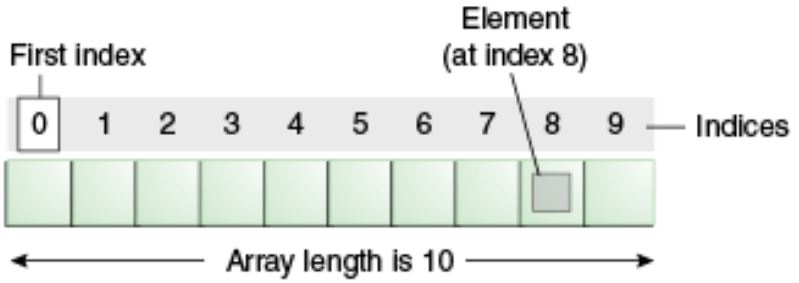
https://www.geeksforgeeks.org/arrays-in-c-cpp/

추상데이터 타입(1)

- **대부분의 프로그래머에게 배열은 단지 '연속된 메모리 위치'**
  - 구현의 관점에서 정의 → 일반적인 관점에서 정의할 필요가 있음

- **배열을 추상데이터 타입 ADT(Abstract Data Type) 관점에서 고려**
- **정의: <index, value>쌍의 집합**
  - set of mappings (or correspondence) between index and values
  - array : i → a_i

ADT의 장점은 배열을 더
일반적인 구조로 설명 →
구현에 독립적으로 설명



구현의 관점에서 배열

추상데이터 타입(2)

ADT Array

object : *index*의 각 값에 대하여 집합 *item*에 속한 한 값이 존재하는
***<index, value>*쌍의 집합.**
index는 일차원 또는 다차원 유한 순서 집합이다.
예를 들면, 1차원의 경우 {0, …., n-1}과
2차원 배열{(0,0), (0,1), (1,1), (1,2), (2.0), (2,1), (2,2)} 등 이다.

functions : 모든 A∈Array, i∈index, x∈item, j, size∈integer

Array **Create**(j, list) ::= **return** j차원의 배열.
list는 i번째 원소가 i번째 차원의
크기인 j-tuple이며 item들은 정의 되지 않음.

item **Retrieve**(A, i) ::= **if** (i∈index)
return 배열 A의 인덱스 i값과 관련된 항목.
else return 에러.

Array **Store**(A, i, x) ::= **if** (i ∈ index) **return** 새로운 쌍 <i,x>가 삽입된 배열 A.
else return 에러.

end Array

Demo: ap1.c

C언어에서의 배열(Arrays in C) (1)

- **C의 일차원 배열:**

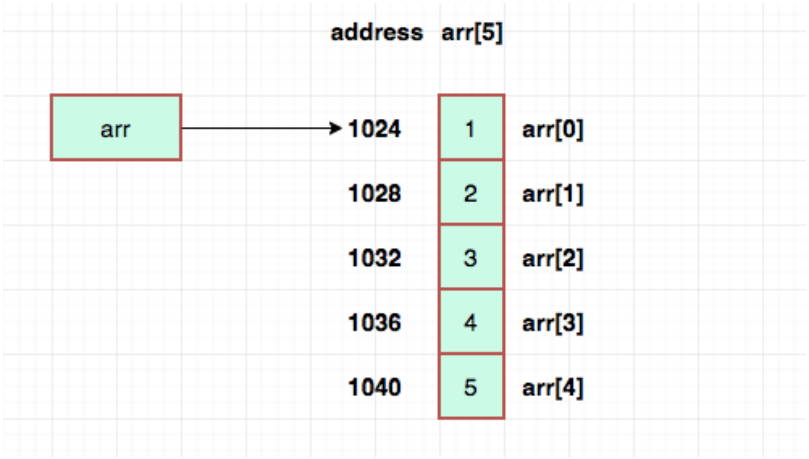
```
int list[5], *plist[5]; /* 5개의 정수 vs. 정수에 대한 5개 포인터*/
```

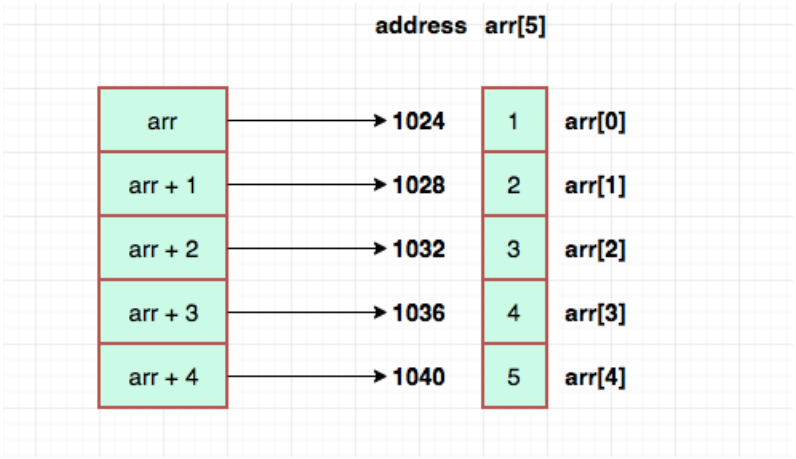
  - 정수 값 : list[0], list[1], list[2], list[3], list[4]
  - 정수포인터 : plist[0], plist[1], plist[2], plist[3], plist[4]

변 수	메모리 주소
list[0]	기본주소 = a
list[1]	a + sizeof(int)
list[2]	a + 2 * sizeof(int)
list[3]	a + 3 * sizeof(int)
list[4]	a + 4 * sizeof(int)

C언어에서의 배열(Arrays in C) (2)

- C에서 index가 0부터 시작하는 이유





Base Position에서의 Offset

https://developerinsider.co/why-does-the-indexing-of-array-start-with-zero-in-c/

https://log2base2.com/c-questions/array/why-array-index-start-with-0.html

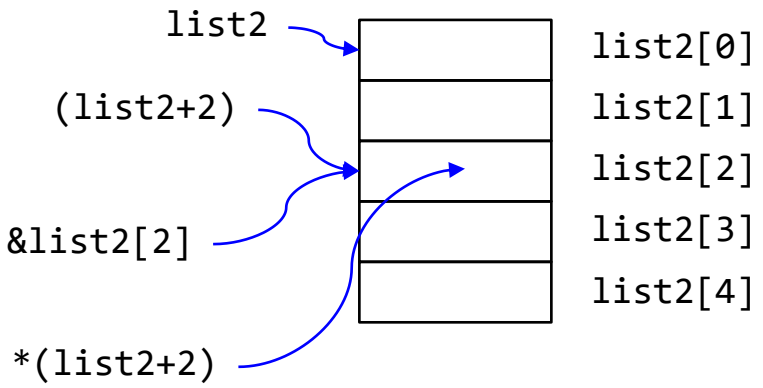
Demo: ap2.c

C언어에서의 배열(Arrays in C) (3)

- **포인터 해석**

  int *list1; int list2[5];

  - list2 = list2[0]를 가리키는 포인터
  - list2 + i = list2[i]를 가리키는 포인터
  - (list2+i) = &list2[i], *(list2+i) = list2[i]



Demo: p2-1.c

C언어에서의 배열(Arrays in C) (4)

- 배열 프로그램의 예 (프로그램 2.1)

```
#define MAX_SIZE 100
float sum(float list[], int);
float input[MAX_SIZE], answer;
int i;
void main(void)
{
        for(i=0; i < MAX_SIZE; i++)
                  input[i] = i;
        answer = sum(input, MAX_SIZE);
        printf("The sum is: %f\n", answer);
}

float sum(float list[], int n)
{
        int i;
        float tempsum = 0;
        for(i = 0; i < n; i++)
                tempsum += list[i];
        return tempsum;
}
```

호출 시 input(=&input[0])은 주소를 담고 있고 → 주소가 sum 함수의 list에 복사

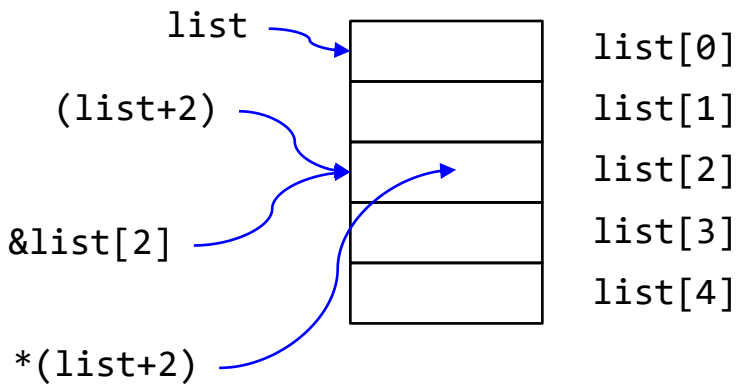
list → call by reference
input의 주소를 list가 받아, input 배열을 list를 통해 접근

n → call by value
값을 그대로 변수 n에 복사
n이 변경 되도 caller의 값은 변하지 않음

C언어에서의 배열(Arrays in C) (5)

- 호출 시 input(=&input[0])의 주소값은 sum함수의 list에 복사

- 역참조(dereference)
  - list[i]가 "="기호 우측 → (list + i)가 가리키는 값 반환
  - list[i]가 "="기호 좌측 → 값을 (list + i) 위치에 저장



C언어에서의 배열(Arrays in C) (6)

Demo: p2-2.c

실습: 함수가 동작하는 Standalone Program 작성

- **예제[일차원 배열의 주소 계산]**
  - `int one[] = {0, 1, 2, 3, 4};`
  - `print1(&one[0], 5)`

```
void print1 (int *ptr, int rows)
{/* print out a one-dimensional array using a pointer */
      int i;
      printf ("Address Contents\n");
      for (i = 0; i < rows; i++)
            printf("%8u%5d\n", ptr + i, *(ptr + i));
      printf("\n");
}
```

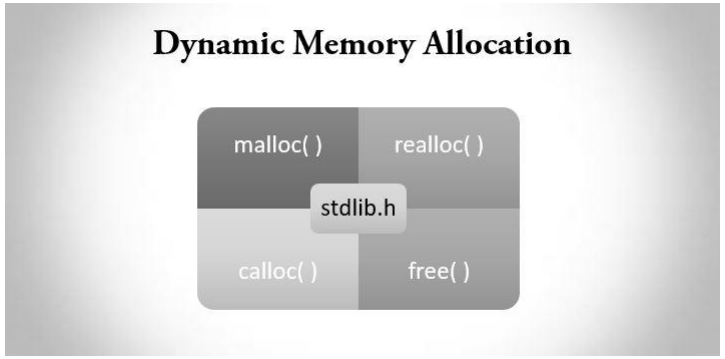
%8u → unsigned integer
%5 → signed integer

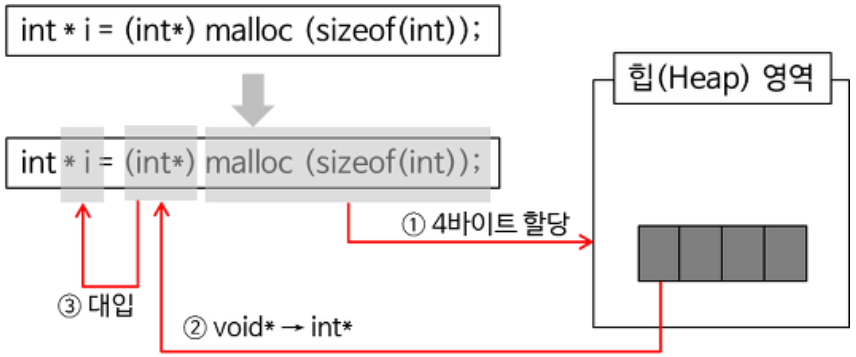
sizeof(int) → 4 bytes

주소	1244868	1244872	1244876	1244880	1244884
내용	0	1	2	3	4

< 1차원 배열의 주소 계산 >

동적으로 할당된 배열
(Dynamically Allocated Arrays)



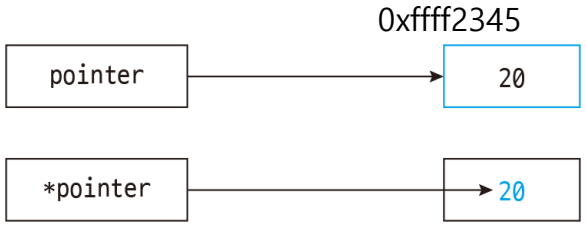


http://www.trytoprogram.com/c-programming/dynamic-memory-allocation-in-c/

https://dsnight.tistory.com/51

C 포인터와 동적 메모리 할당 리뷰(1)
(Pointers and Dynamic Memory Allocations)

- **포인터**
  - C언어에서는 어떤 타입 T에 대해 T의 포인터 타입이 존재
  - 포인터 타입의 실제값은 메모리 주소가 됨
  - 포인터 타입에 사용되는 연산자
    - & : 주소 연산자
    - * : 역참조(dereferencing, 간접 지시) 연산자



(Source) https://dojang.io/mod/page/view.php?id=605

Ex) i(정수 변수), pi(정수에 대한 포인터)

```
int i, *pi
pi = &i;
```

i에 10을 저장하기 위해서는 다음과 같이 할 수 있다.

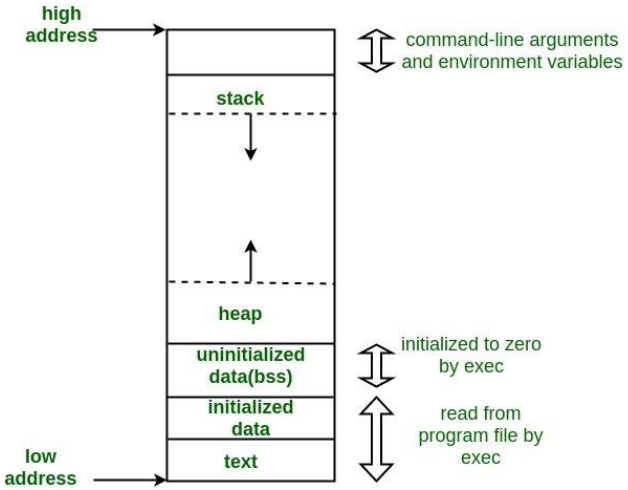
`i = 10;` 또는 `*pi = 10;`

  - 널(null)포인터 : 어떤 객체나 함수도 가리키지 않는다.
    - 정수 0값으로 표현
    - 널포인터에 대한 검사

`int (pi == NULL)` 또는 `if(!pi)`

C 포인터와 동적 메모리 할당 리뷰(2)
(Pointers and Dynamic Memory Allocations)

- **동적 메모리 할당**
  - 프로그램을 작성할 때 얼마나 많은 공간이 필요한지 알 수 없을 때 사용
  - 히프(heap) 기법
  - 새로운 메모리 공간이 필요할 때마다 함수 malloc을 호출해서 필요한 양의공간을 요구

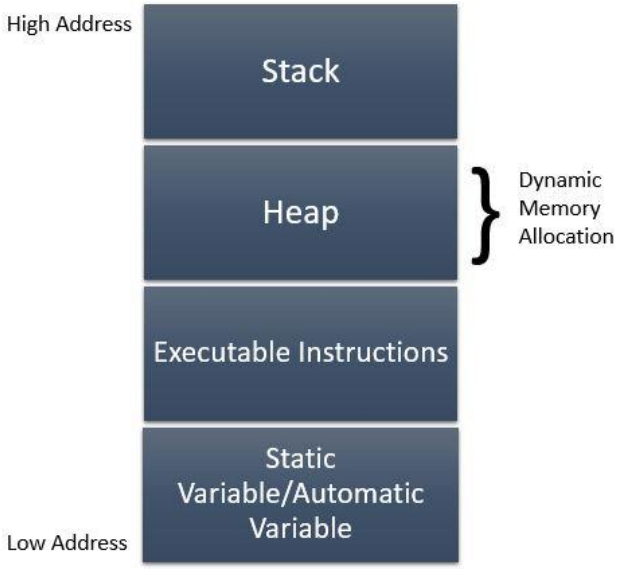


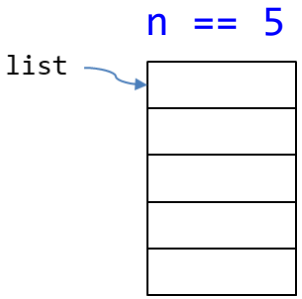

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

메모리 할당과 반환

동적할당의 필요성

- **필요성**
  - MAX_SIZE=100 → MAX_SIZE=1000 ← 컴파일을 다시 해야
  - 배열의 크기가 프로그램 실행시간에 동적으로 변할 경우
  - 배열의 크기를 결정하기 힘들 때, 실행시간을 미루었다가 배열의 크기가 정해지면 동적으로 할당





http://www.trytoprogram.com/c-programming/dynamic-memory-allocation-in-c/

1차원 배열

```
int i, n, *list;
printf(“Enter the number of number to generate: ”);
scanf(“%d”, &n”);
if (n < 1) {
        fprintf(stderr, “Improper value of n \n”);
        exit (EXIT_FAILURE);
}
MALLOC(list, n * sizeof(int));
```

n < 1이거나 정렬할 수의 리스트를 저장 할 메모리가 충분치 않을 때 실패

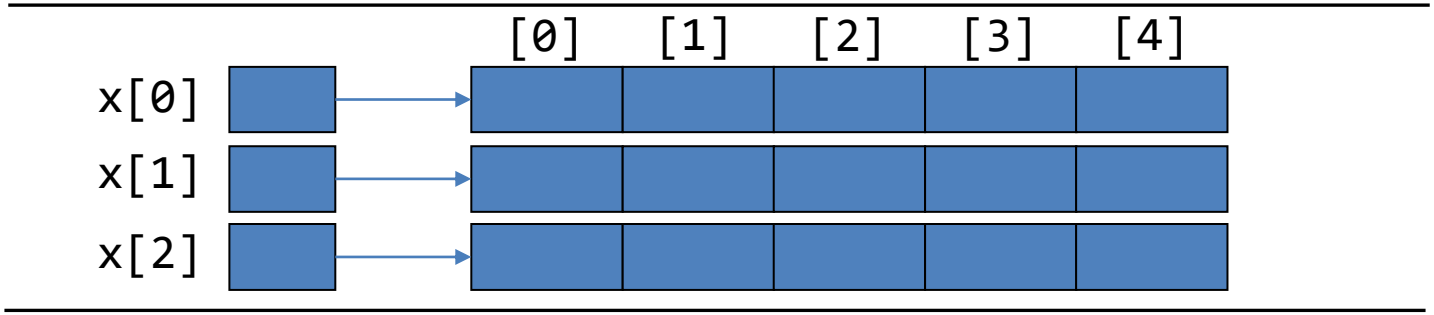
```
#define MALLOC(p, s) \
    if (!((p) = malloc(s))) {\
        fprintf(stderr, “Insufficient memory”); \
        exit(EXIT_FAILURE);
    }
```

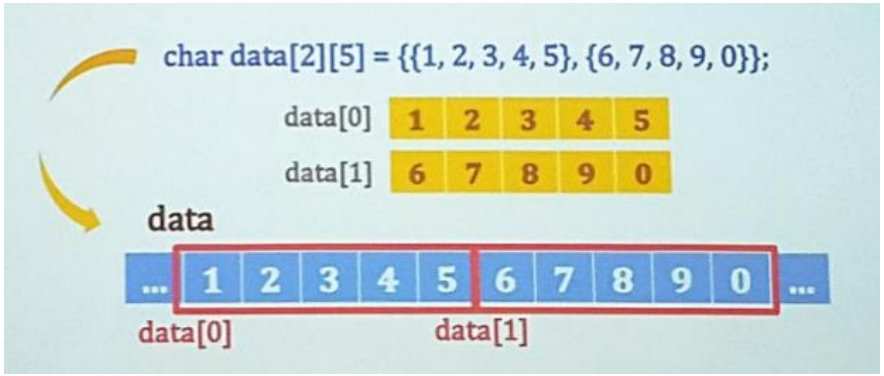
← page 7 MALLOC MACRO 참조

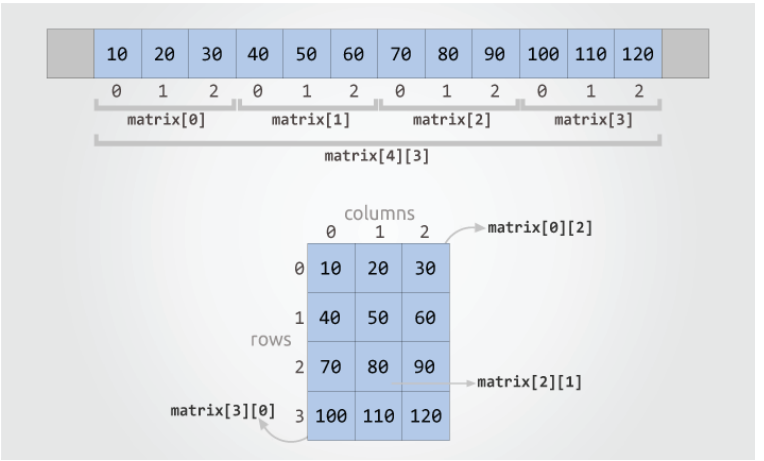
2차원 배열 (1)

- **2차원 배열 → 차원이 커지더라도, 배열의 배열로 생각**
  - 2차원 → 1차원 배열의 1차원 배열
  - 3차원 → 2차원 배열의 1차원 배열

```
int x[3][5]
```





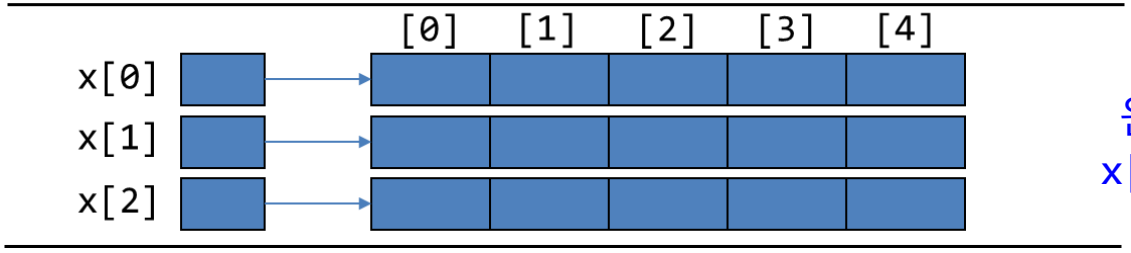


2차원 배열 (2)

Demo: size.c

2차원 배열의 동적 생성

```
/* create a two dimensional rows × cols array */

int **make2dArray(int rows, int cols)
{
        int **x, i;

        /* get memory for row pointers */
        MALLOC (x, rows * sizeof (*x));;        ← 주소를 저장 (8바이트)

        /* get memory for each row */
        for (i = 0; i < rows; i++)
                MALLOC(x[i], cols * sizeof(**x));    ← 값을 저장 (4바이트)
        return x;
}
```



원소 x[i][j]를 찾을 때,
x[i]에 있는 포인터에 접근

2차원 배열 (3)

- **example:**

```
int **myArray;

myArray = make2dArray(5, 10);

myArray[2][2] = 6
```

5×10 2차원 정수 배열에 대한 메모리 할당
이 배열의 [2][2]원소에 정수 6을 지정

```
int** make2dArray(int rows, int cols)
{ /* create a two dimensional rows × cols array */
        int **x, i;

        /* get memory for row pointers */
        MALLOC (x, rows * sizeof (*x));;

        /* get memory for each row */
        for (i = 0; i < rows; i++)
                MALLOC(x[i], cols * sizeof(**x));
        return x;
}
```

2차원 배열 (4)

- **calloc / realloc 함수**
  - calloc : 사용자가 지정한 양의 메모리를 할당 후 0으로 초기화
  - realloc : malloc이나 calloc으로 이미 할당된 메모리 크기 재조정

```
int *x;
x = calloc(n, sizeof(int))
```

MACRO 정의

```
#define CALLOC(p, n, s)\
          if(!((p) = calloc(n, s))) {\
                    fprintf(stderr, "Insufficient memory"):\
                    exit(EXIT_FAILURE):\
          }
```

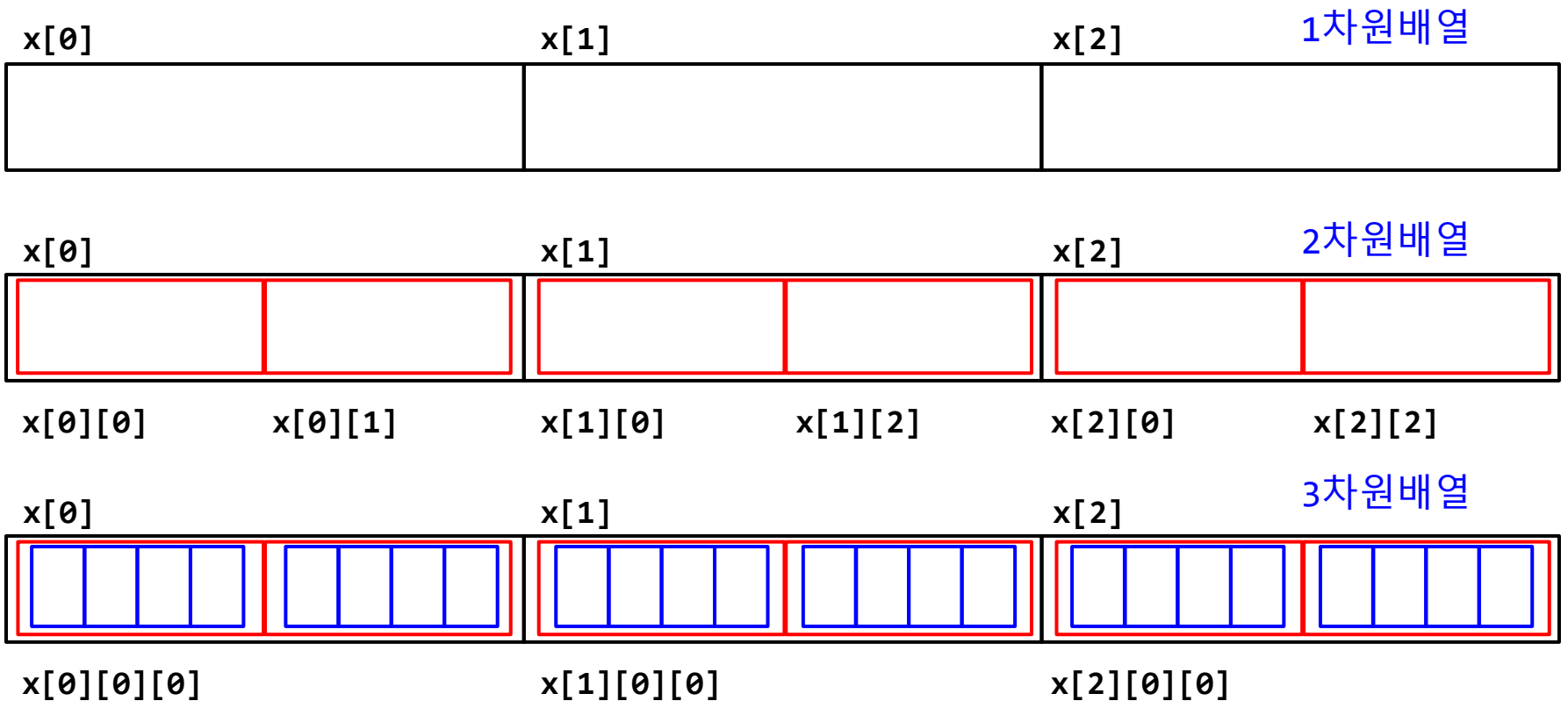
```
#define REALLOC(p, s)\
          if(!((p) = realloc(n, s))) {\
                    fprintf(stderr, "Insufficient memory"):\
                    exit(EXIT_FAILURE):\
          }
```

s > oldSize: s-oldSize 만큼의 임의 값 저장
s < oldSize : 블록에서 oldSize – s 메모리해제

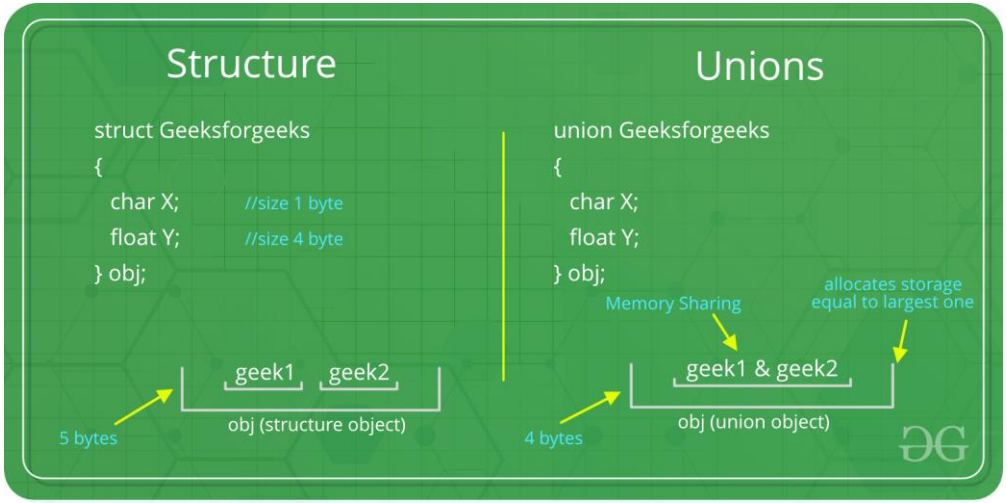
3차원 배열

- 2차원 배열의 1차원 배열

int x[3][2][4]



구조와 유니언
(Structures and Unions)



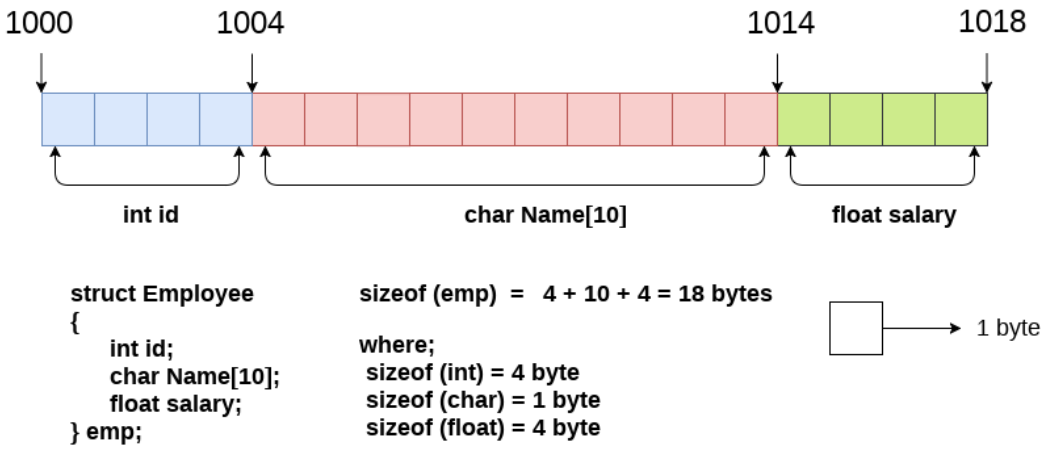
https://www.geeksforgeeks.org/union-c/

구조(structure) (1)

배열은 같은 타입의 데이터 모임

- **struct**
  - 타입이 다른 데이터를 그룹화 (레코드)
  - 데이터 항목의 집단 – 각 항목은 타입과 이름으로 식별

```
struct {
        int id;
        char name[10];
        float salary;
} person;
```



  - 구조의 멤버 연산자

구조의 멤버연산자로 '.' 사용

```
strcpy(person.name, "james");
person.age = 10;
person.salary = 35000;
```

Demo: struct.c

구조(structure) (2)

- **typedef 문장을 사용한 구조 데이터 타입 생성** 2가지 방법

```
typedef struct human_being{
        char name[10];
        int age;
        float salary;
        };
```

```
typedef struct {
        char name[10];
        int age;
        float salary;
        } human_being;
```

- **변수선언**

```
    human_being person1, person2;
    if (strcmp(person1.name, person2.name))
            printf("두사람의 이름은 다르다.\n");
    else
            printf("두사람의 이름은 같다.")
```

  - 전체 구조의 동등성 검사 : ~~if (person1 == person2)~~
  - 구조 치환 : person1 = person2

치환의 구현 (Old C)

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

Demo: struct.c

구조(structure) (3)

- 구조의 동등성 검사

```
#define FALSE 0
#define TRUE 1
```

두 구조체의 값이 동일한지 검사

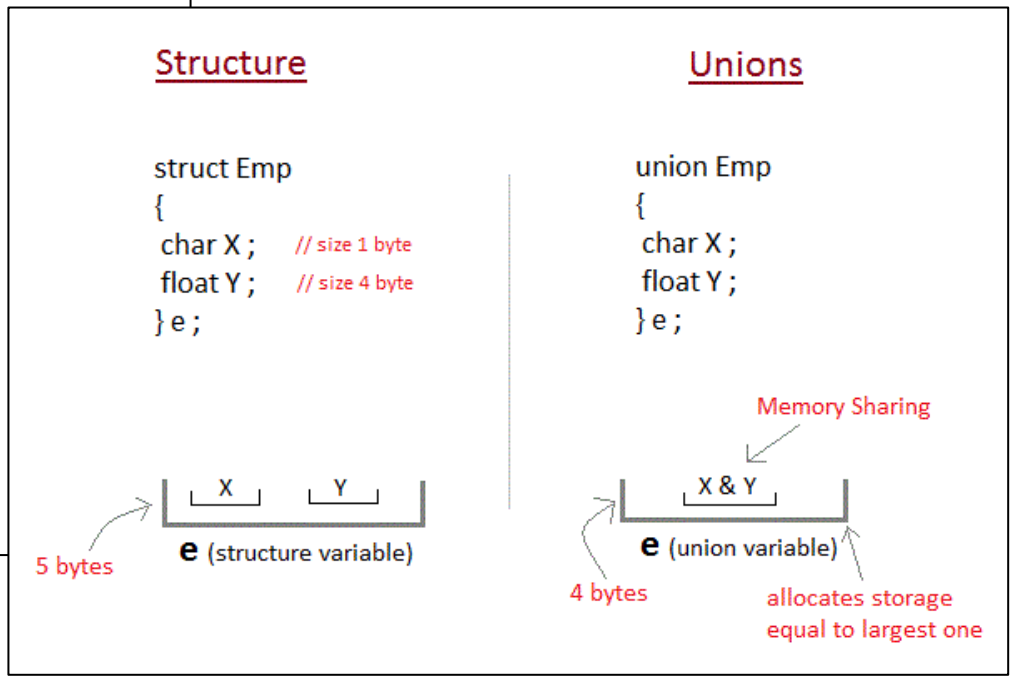
```
int humansEqual (humanBeing person1, humanBeing person2)
{/* 만일 personal과 person2가 동일인이면 TRUE를 반환하고
    그렇지 않으면 FALSE를 반환한다. */

   if (strcmp(person1.name, person2.name))   문자열 비교
          return FALSE;

   if (person1.age != person2.age)
          return FALSE;

   if (person1.salary != person2.salary)
          return FALSE;

   return TRUE;
}
```

유니언(union) (1)

- union의 필드들은 ***메모리 공간을 공유***

```
typedef struct sex_type {
        enum tagField {female, male} sex;
        union {
                int children;
                int beard;
                } u;
        };
typedef struct human_being {
        char name[10];
        int age;
        float salary;
        sex_type sex_info;
        };
human_being person1, person2;
```

메모리 공간 공유




```
person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;

person2.sex_info.sex = female;
person2.sex_info.u.children = 4;
```

C에서 union 필드가 올바르게 사용했는지 여부 검사하지 않음

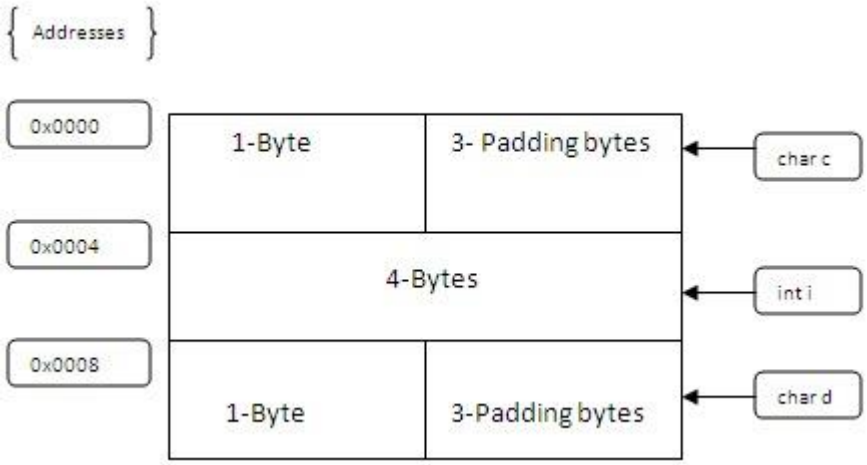
구조의 내부구현

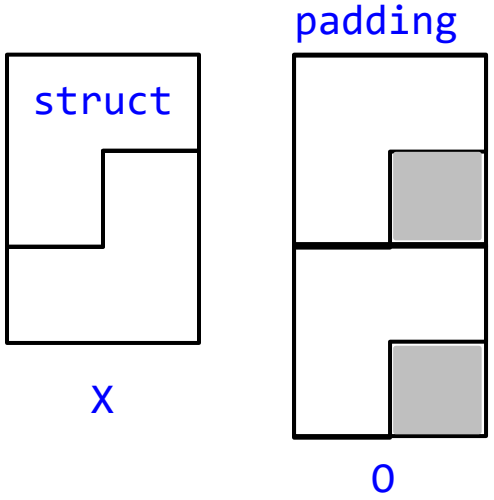
Demo: padding.c

```
struct {int i, j; float a, b;} or
struct {int i; int j; float a; float b;}
```

Compiler 마다
padding 처리가 다름

- 오름차 주소의 위치를 이용하여 저장
- 구조 내 빈 공간을 두거나 채워넣기(padding)를 할 수 있다.
- 구조는 같은 메모리 경계에서 시작하고 끝나야 함
- 짝수바이트거나 4, 8, 16등의 배수가 되는 메모리 경계





자기참조 구조

- 구성요소 중 자신을 가리키는 포인터가 존재하는 구조
- 통상적으로 동적 저장공간 관리 루틴(malloc과 free) 필요

```
typedef struct list {
                    char data;
                    list *link;
                    };
```

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';                    NULL로 초기화
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;                 구조들을 서로 연결
item2.link = &item3;              (item1 → item2 → item3)
```

다항식
(Polynomials)

추상 데이터 타입 (1)

구현에 독립적

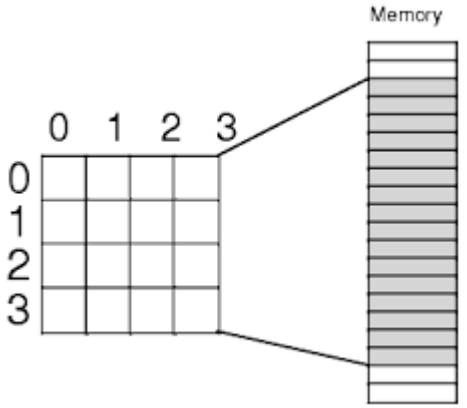
- **추상 데이터 타입(ADT)**
  - 배열을 이용하여 다항식 추상데이터 타입 정의
  - 배열은 자체가 자료구조인 동시에 다른 추상데이터 타입구현에 이용

- **순서 리스트(Ordered list, linear list)**
  - 원소들의 순서가 있는 모임

    $A = (a_1, a_2, \ldots, a_n)$ a_i : atom, n=0 : empty list
    - 한 주일의 요일들(일, 월, 화, 수, 목, 금, 토)
    - 카드 한 벌의 값(ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, j, q, k)
    - 건물의 층(지하실, 로비, 일층, 이층)
    - 미국의 2차 세계 대전 참전 연도(1941, 1942, 1943, 1944, 1945)
  - 연산
    - 리스트의 길이 n의 계산
    - 리스트의 항목을 왼쪽에서 오른쪽 (오른쪽에서 왼쪽)으로 읽기
    - 리스트로부터 i번째 항목을 검색, $0 \le i \le n$
    - 리스트의 i번째 항목을 대체, $0 \le i \le n$
    - 리스트의 i번째 위치에 새로운 항목을 삽입(i번째 위치,$0 \le i \le n$)
    - 리스트의 i번째 항목을 제거(i번째 항목, $0 \le i < n$)

추상 데이터 타입 (2)

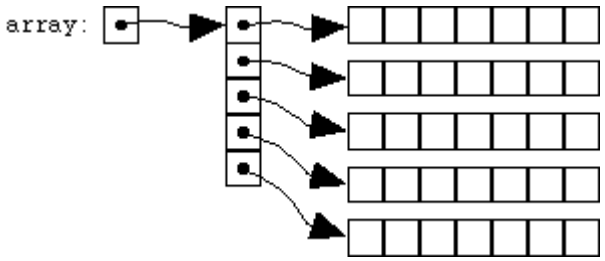
int array[4][4]



4x4 배열의
정적 메모리 할당

- **순서리스트의 표현**
  - 메모리(기억장소) 표현
  - 순차 사상 (sequential mapping)
  - 물리적 인접성(arrays)

- **비순차 사상(non-sequential mapping)**
  - 비연속적 기억장소 위치
  - 리스트 : pointer(links)를 가지는 노드 집합



5x7 배열의
동적 메모리 할당

```
int **array = (int**)malloc(sizeof(int *)*5);

for (int i=0; i < 5; i++)
    array[i] = (int*)malloc(sizeof(int)*7));
```

메모리 주소가 비연속적

추상 데이터 타입 (3)

- **다항식 연산**

$$A(x) = 3x^{20}+2x^{5}+4, \quad B(x) = x^{4}+10x^{3}+3x^{2}+1$$

$$A(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0 x^0$$

ax^e a : coefficient $a_n \neq 0$

e : exponent – unique

x : variable x

- 차수(degree) : 다항식에서 가장 큰 지수

- **다항식 연산**

$A(x) = \sum a_i x^i$와 $B(x) = \sum b_i x^i$ 이 있다고 가정하면

$A(x) + B(x) = \sum (a_i + b_i) x^i$

$A(x) \cdot B(x) = \sum (a_i x_i \cdot (\sum b_j x_j))$

추상 데이터 타입 (4)

ADT =
1) 객체정의
2) 객체에 대한 함수 정의

- **Polynomial 추상 데이터 타입**

Structure Polynomial
Objects : P(x) = a_1x^{e1} + … + $a_nx^{e_n}$: <e_i, a_i>의 순서쌍 집합. 객체정의
여기서 a_i는 Coefficient, e_i는 Exponents. $e_i(\geq 0)$는 정수

Function : 모든 poly, poly1, poly2 ∈ polynomial,
coef ∈ Coefficients, expon ∈ Exponents에 대해

객체에 대한 연산을 할 함수 정의

```
Polynomial Zero()        ::= return 다항식, p(x) = 0

Boolean IsZero(poly)     ::= if(poly) return FALSE
                             else return TRUE

Coefficient Coef(poly, expon)   ::= if(expon∈poly) return 계수
                                    else return 0

Exponent Lead_Exp(poly)         ::= return poly에서 제일 큰 지수

Polynomial Attach(poly, coef, expon) ::= if(expon∈poly) return 오류
                                         else return <coef, exp>항이
                                                     삽입된 다항식 poly
```

추상 데이터 타입 (5)

- **Polynomial 추상 데이터 타입 (계속)**

```
   Polynomial Remove(poly, expon)    ::= if(expon∈poly)
                                          return 지수가 expon인 항이
                                                 삭제된 다항식 poly
                                          else return 오류

   Polynomial SingleMult(poly, coef, expon) ::= return 다항식 poly·coef·x^expon

   Polynomial Add(poly1, poly2)              ::= return 다항식 poly1+poly2

   Polynomial Mult(poly1, poly2)             ::= return 다항식 poly1·poly2

end polynomial
```

객체를 어떻게 정의하느냐에 따라 function의 구현방법이 달라짐

다항식 표현(1)

- 지수들의 내림차순으로 정돈

```
#define MAX_DEGREE 101      /* 다항식의 최대 차수 +1*/     ← a0
typedef struct {
                  int degree;
                  float coef[MAX_DEGREE];
                } polynomial;
```

- a : polynomial 타입, n < MAX_DEGREE

$A(x) = \sum a_i x^i$ 의 표현

a.degree = n, a.coef[i] = a_{n-i}, $0 \le i \le n$ ← 내림차순

A = (n, a_n, a_{n-1}, … , a_1, a_0)

n: degree of A

a: n+1 coefficients

예) $A(x) = x^4+10x^3+3x^2+1$: n = 4

A = (4, 1, 10, 3, 0, 1) : 6 elements

a.degree << MAX_DEGREE 일 경우 → 메모리 낭비 초래

다항식 표현(2)

- **다항식 덧셈(초기버전): padd**

```
/* d = a + b, 여기서 a, b, d는 다항식이다. */
d = Zero();
while (! IsZero(a) && ! IsZero(b)) do {
      switch COMPARE(Lead_Exp(a), Lead_Exp(b)) {
          case -1:
              d = Attach (d, Coef(b, Lead_Exp(b)), Lead_Exp(b));
              b = Remove(b, Lead_Exp(b));
              break;
          case 0:
              sum = Coef(a, Lead_Exp(a)) + Coef(b, Lead_Exp(b));
              if (sum) {
                  Attach(d, sum, Lead_Exp(a));
                  a = Remove(a, Lead_Exp(a));
                  b = Remove(b, Lead_Exp(b));
              }
              break;
          case 1;
              d = Attach(d, Coef(a, Lead_Exp(a)), Lead_Exp(a));
              a = Remove(a, Lead_Exp(a));
      }
}
a 또는 b의 나머지 항을 d에 삽입한다.
```

가장 큰 지수 비교

지수가 다르고 a < b

b항 → d 추가

b에서 현재 지수항 삭제

지수가 같다

계수를 더하고

계수값, a항 → d 추가

a항, b항 제거

지수가 다르고 a > b

다항식의 덧셈(1)

a.degree << MAX_DEGREE → 메모리 낭비 초래

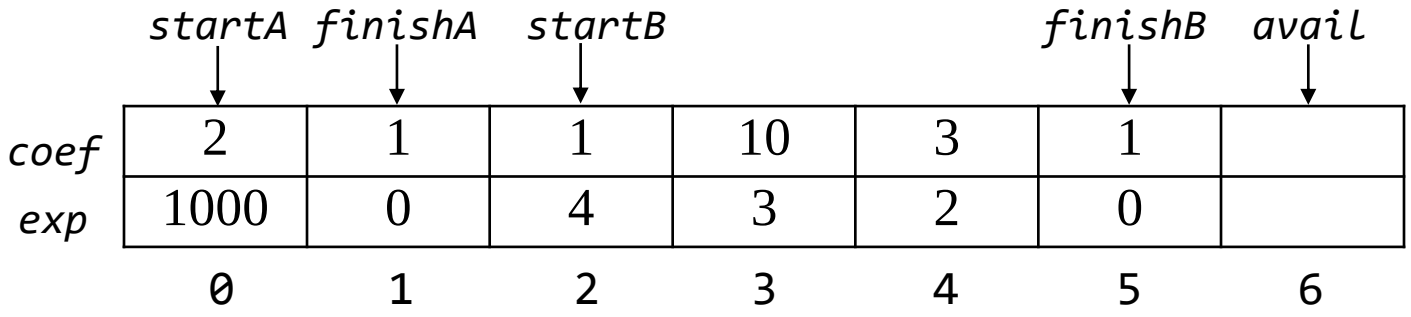
- 공간절약을 위해 모든 다항식을 저장하는 전역 배열 terms 도입

```
MAX_TERMS 100       /* 항 배열의 크기*/
typedef struct {
                    float coef;
                    int expon;
                    } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

$A(x) = 2x^{1000}+1$

$B(x) = x^4+10x^3+3x^2+1$

계수(coef)가 대부분 0인 경우 메모리 절약



A(x) : <startA, finishA>

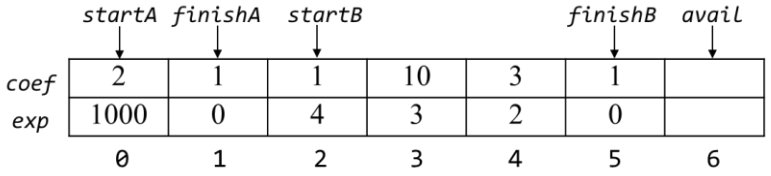
finishA = startA + n - 1

다항식의 덧셈(2)

- **다항식 덧셈: padd – 개선된 버전**

```
void padd(int starta, int finisha, int startb, int finishb, int *startd,
int *finishd)
{  /* A(x)와 B(x)를 더하여 D(x)를 생성한다. */
    float coefficient;
    *startd = avail;
    while (starta <= finisha && startb <= finishb)
        switch (COMPARE(terms[starta].expon, terms[startb].expon))
        {
            case -1:   /* a의 expon이 b의 expon보다 작은 경우 */
                attach(terms[startb].coef, terms[startb].expon);
                startb++; break;
            case 0:    /*지수가 같은 경우 */
                coefficient = terms[starta].coef + terms[startb].coef;
                if(coefficient) attach(coefficient, terms[starta].expon);
                starta++; startb++; break;
            case 1:    /* a의 expon이 b의 expon보다 큰 경우 */
                attach(terms[starta].coef, terms[starta].expon);
                starta++; }
```

각 다항식의 exp, coef 끝까지



avail == startd의 위치

다항식의 덧셈(3)

- **다항식 덧셈: padd – 개선된 버전**

```
        /* A(x)의 나머지 항들을 첨가한다. */
        for(; starta <= finisha; starta++)
                attach(terms[starta].coef, term[starta].expon);
        /* B(x)의 나머지 항들을 첨가한다. */
        for(; starta <= finishb; startb++)
                attach(terms[startb].coef, terms[startb].expon);
        *finishd = avail – 1;
```

< 두 다항식을 더하는 함수 >

```
void attach(float coefficient, int exponent)
{              /* 새 항을 다항식에 첨가한다. */
          if (avail >= MAX_TERMS) {
                  fprintf(stderr, “다항식에 항이 너무 많다.”);
                  exit(1);           }
          terms[avail].coef = coefficient;
          terms[avail++].expon = exponent;
}
```

< 새로운 항을 첨가하는 함수 >

다항식의 덧셈(4)

- **<개선전> padd 알고리즘 분석**
  - O(m*n) → a: n차 다항식, b: m차 다항식

- **<개선후> padd 알고리즘 분석**
  - m, n(>0) ; 각각 A와 B의 0이 아닌 항의 수
  - while 루프 : O(m+n)
    - 각 반복마다 starta나 startb 또는 둘 다 값이 증가
    - 반복 종료→ starta <= finisha && startb <= finishb
    - 반복 횟수 ≤ m+n-1

최악의 경우:
지수가 상호 교차

$A(x) = \sum_{i=0}^{n} x^{2i}$ 와 $\sum_{i=0}^{n} x^{2i+1}$

a: 7, 5, 3, 1
b: 6, 4, 2

  - for 루프 : O(m+n)

avail = MAX_TERMS ? → 불필요한 다항식 제거 후 배열 끝에 가용 공간 생성 → 데이터 이동시간, 각 다항식의 starti, finishi 변경

동적메모리 공간할당으로 공간부족 해결해야 함

희소 행렬
(Sparse Matrices)

추상 데이터 타입(1)

- **행렬 : m x n matrix A = A[MAX_ROWS][MAX_COLS]**
- **희소행렬 (sparse matrix)**
  - 어떤 행렬이 희소행렬인지 결정은 직관적

$$\frac{\text{no. of non-zero elements}}{\text{no. of total elements}} << \text{small}$$

36개의 원소 중 8개만 0이 아님

	col0	col1	col2
row0	-27	3	4
row1	6	82	-2
row2	109	-64	11
row3	12	8	9
row4	48	27	47

희소행렬x

	col0	col1	col2	col3	col4	col5
row0	15	0	0	22	0	-15
row1	0	11	3	0	0	0
row2	0	0	0	-6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

2000개의 0이 아닌 원소를 가지는 1000x1000 행렬을 저장하는 데 필요한 공간?
→ 1,000,000 – 2,000 = 998,000 저장공간낭비

추상 데이터 타입(2)

ADT =
1) 객체정의
2) 객체에 대한 함수 정의

- **희소 행렬의 추상데이터 타입**

```
ADT sparseMatrix
     object: 3원소 쌍 <행, 열, 값>의 집합이다. 여기서, 행과 열은 정수이고
             이 조합은 유일하며, 값은 item집합의 원소이다.

     functions : 모든 a, b∈sparseMatrix, x∈item, i, j, maxCol,
                 maxRow∈index에 대해

     sparseMatrix Create(maxRow, maxCol) ::= return maxItems까지 저장 할 수 있는
                                             sparseMatrix, 여기서 최대 행의
                                             수는 maxRow이고 최대 열의 수는
                                              maxCol이라 할 때
                                             maxItems = maxRow×maxCol이다.

     sparseMatrix Transpose(a)           ::= return 모든 3원소 쌍의 행과 열의
                                             값을 교환하여 얻은 행렬

     sparseMatrix Add(a,b)               ::= if a와 b의 차원이 같으면
                                            return 대응 항들 즉, 똑같은
                                                 행과 열의 값을 가진 항들을
                                                 더해서 만들어진 행렬
                                            else return 에러
```

추상 데이터 타입(3)

```
sparseMatrix Multiply(a,b) ::=    if a의 열의 수와 b의 행의 수가 같으면
                                  return 다음 공식에 따라 a와 b를 곱해서 생성된
                                         행렬 d : d(i, j) = Σ(a[i][k] b[k][j])
                                         여기서 d(i, j)는 (i, j)번째 원소이다.
                                  else return 에러
```

희소 행렬 표현

- 3원소 쌍 <행(row), 열(col), 값(value)>으로 표현
- 전치연산을 효율적으로 표현하기 위해 행을 오름차순으로 조직
  - (0 → n)
- 동일 행의 경우 열 인덱스가 오름차순으로 정렬되도록 조직
- 연산 종료를 보장하기 위해 행과 열의 수와 행렬 내에 있는 0이 아닌 항의 수를 알아야 함 (value != 0인 값만 대상)

행렬의 전치(Transpose) (1)

각 행 i에 대해서
원소 <i, j, 값>을 가져와서
전치 행렬의 원소 <j, i, 값>으로 저장
ex) (0, 0, 15) 는 (0, 0, 15)
(0, 3, 22) 는 (3, 0, 22)
(0, 5, -15) 는 (5, 0, -15)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Transpose of a matrix

	행	열	값
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

	행	열	값
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

6x6 행렬 0이 아닌 값 = 총 8개

행렬의 전치(Transpose) (2)

- **transpose 함수 – simple version**

희소행렬이 아닐 경우

각 행 i에 대해서
원소 <i, j, 값>을 가져와서
전치 행렬의 원소 <j, i, 값>으로 저장

```
for (j = 0; j < columns; j++)
    for(i = 0; i < rows; i++)
        b[j][i] = a[i][j];
```

```
void transpose(term a[], term b[])    /* a를 전치시켜 b를 생성 */
{
    int n, i, j, currentb;
    n = a[0].value;              /* 총 원소 수 */
    b[0].row = a[0].col;         /* b의 행 수 = a의 열 수 */
    b[0].col = a[0].row;         /* b의 열 수 = a의 행 수 */
    b[0].value = n;
    if (n > 0) {                   /* 0이 아닌 행렬 */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)   /* 0번 열부터 순차적으로 */
          for (j = 1; j < n; j++)        /* 현재의 열로부터 원소를 찾는다 */
             if (a[j].col == i) {  /* 현재의 열에 있는 원소를 b에 첨가한다.*/
                   b[currentb].row = a[j].col;
                   b[currentb].col = a[j].row;
                   b[currentb].value = a[j].value;
                   currentb++; }
    }
}
```

	행	열	값
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

a[0].col = 열의 개수
n = 0이 원소의 개수

Time Complexity = O(columns * n), n = # of elements

행렬의 전치(Transpose) (3)

- **transpose 함수 – simple version**

	행	열	값
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

	행	열	값
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

가장 바깥쪽 for loop : 0 → a[0].col

안쪽 for loop의 if : 0, 1, 2 ... 열들을 행으로 b에 순차적으로 저장

Time Complexity = O(columns * n), n = # of elements

만약, # of elements = columns * rows → O(columns^2 * rows)

공간절약 → 시간복잡도의 증가

행렬의 전치(Transpose) (4)

- **fasttranspose 함수 – 개선된 버전**

```
void fasttranspose(term a[], term b[])          /* a를 전치시켜 b를 생성 */
{
        int rowTerms[MAX_COL], startingPose[MAX_COL];
        int i, j, numCol = a[0].col, numTerms = a[0].value;
        b[0].row = numCols; b[0].col = a[0].row;
        b[0].value = numTerms;
        if (numTerms > 0)   { /* 0이 아닌 행렬 */
                for(i = 0; i < numCols; i++)      초기화
  O(columns)            rowTerms[i] = 0;

  O(elements)   for(i = 1; i <= numTerms; i++)
                        rowTerms[a[i].col]++;     열의 위치 카운트

                startingPos[0] = 1;               다음 전치할 위치 저장

  O(columns)    for(i = 1; i < numCols; i++)      offset (jump할 곳)을 누적으로 저장
                        startingPos[i] =
                                startingPos[i-1] + rowTerms[i-1];

  O(elements)   for(i = 1; i <= numTerms; i++) {  startingPos만큼 건너뛰면서 전치
                        j = startingPos[a[i].col]++;
                        b[j].row = a[i].col; b[j].col = a[i].row;
                        b[i].value = a[i].value; }
                }
}                       Time Complexity = O(columns + elements)
```

행렬의 전치(Transpose) (5)

rowTerm[], startingPos[]
저장공간 사용 →
Time Complexity =
O(columns + elements)

- **fasttranspose 함수 – 개선된 버전**

	행	열	값
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

```
void fasttranspose(term a[], term b[])          /* a를 전치시켜 b를 생성 */
{
      int rowTerms[MAX_COL], startingPose[MAX_COL];
      int i, j, numCol = a[0].col, numTerms = a[0].value;
      b[0].row = numCols; b[0].col = a[0].row;
      b[0].value = numTerms;
      if (numTerms > 0)   { /* 0이 아닌 행렬 */
            for(i = 0; i < numCols; i++)      초기화
O(columns)        rowTerms[i] = 0;

O(elements)   for(i = 1; i <= numTerms; i++)
                  rowTerms[a[i].col]++;       열의 위치 카운트

              startingPos[0] = 1;             다음 전치할 위치 저장

O(columns)    for(i = 1; i < numCols; i++)    offset (jump할 곳)을 누적으로 저장
                  startingPos[i] =
                        startingPos[i-1] + rowTerms[i-1];

O(elements)   for(i = 1; i <= numTerms; i++) {  startingPos만큼 건너뛰면서 전치
                  j = startingPos[a[i].col]++;
                  b[j].row = a[i].col; b[j].col = a[i].row;
                  b[i].value = a[i].value; }
            }
}
                  Time Complexity = O(columns + elements)
```

열 → 행으로 바뀔 열의 수

rowTerms[0]	2
rowTerms[1]	1
rowTerms[2]	2
rowTerms[3]	2
...	

저장할 위치
(offset): 누적

startingPos[0]	1
startingPos[1]	3
startingPos[2]	4
startingPos[3]	6
startingPos[4]	8
...	

	행	열	값
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

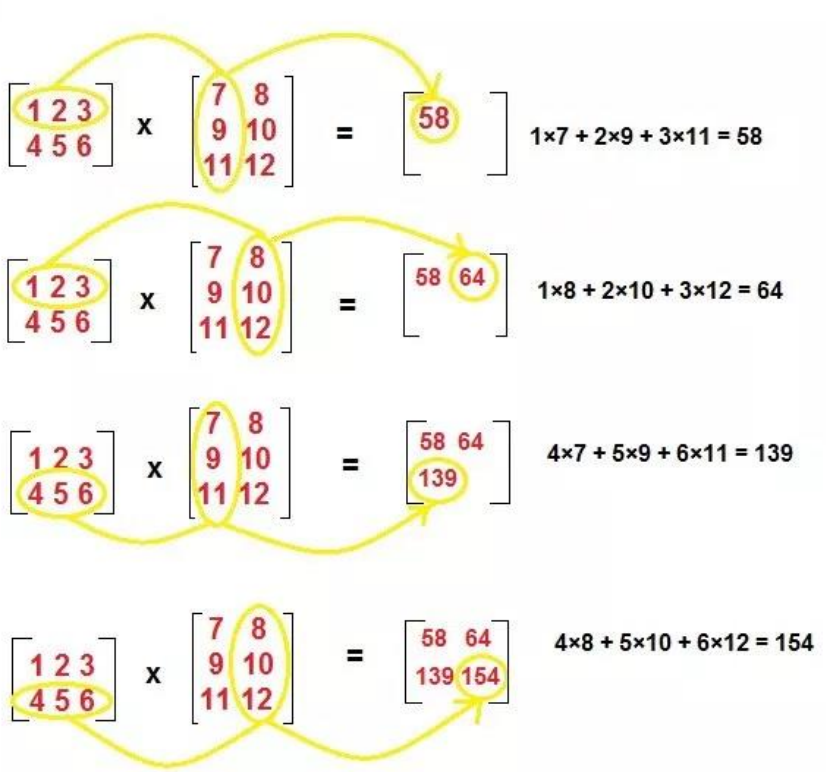
행렬 곱셈 (1)

- mxn행렬 A와 nxp행렬 B가 주어질 때 곱셈 결과 행렬인 D는 mxp 차원을 가지며, 0 ≤ i ≤ m, 0 ≤ j ≤ p에 대해 원소<i, j>는 다음과 같다.

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

< 두 희소 행렬의 곱셈 >



두 희소행렬의 곱셈 결과는 희소행렬이 아니다

행렬 곱셈 (2)

Time Complexity를 구할 때, 증가되는 i, j에 대해 파악

- **희소행렬의 곱셈**

데이터 접근성을 높이기 위해 b 행렬 전치 후 곱

```
void mmult (term a[], term b[], term d[]) {    /* 두 희소 행렬을 곱한다. */
     int i, j, column, totalB = b[0].value, totalD = 0;
     int rowsA = a[0].row, colsA = a[0].col;
     totalA = a[0].value; int colsB = b[0].col;
     int rowBegin = 1, row = a[1].row, sum=0;
     int rowB[MAX_TERMS][3];
     if (colsA != b[0].row) {
               fprintf(stderr, "Incompatible matrices\n");
               exit(1);
     }

     fastTranspose(b, newB); /* b를 전치 */      O(colsB + totalB)

     /* 경계 조건 설정 */
     a[totalA+1].row = rowA;
     newB[totalB+1].row = colsB;
     newB[totalB+1].col = 0;
      column = newB[1].row;

    for (i = 1; i < totalA;) {
          column = newB[1].row;
          for (j = 1; j <= totalB+1;) {        /* a의 행과 b의 열을 곱한다. */
              if (a[i].row != row) {
                    storeSum(d, &totalD, row, column, &sum);
                    i = rowBegin;
                    for (; newB[j].row == column; j++)
                    ;
                    column = newB[j].row;
```

rowTerms[0]	2
rowTerms[1]	1
rowTerms[2]	2
rowTerms[3]	2
...	

행렬 곱셈 (3)

- 희소행렬의 곱셈 (계속)

```
                }
                else if (newB[j].row != column) {
                     storeSum(d, &totalD, row, column, &sum);
                     i = rowBegin;
                     column = newB[j].row;
                }
                else if (newB[j].row != column) {
                     storeSum(d, &totalD, row, column, &sum);
                     i = rowBegin;
                     column = newB[j].row;
                }
                else switch (COMPARE(a[i].col, newB[j].col)) {
                          case -1:        /*a의 다음항으로 이동 */
                               i++; break;
                          case 0:     /*항을 더하고, a와 b를 다음 항으로 이동 */
                               sum += (a[i++].value * newb[j++].value);
                               break;
                          case 1 :         /* b의 다음 항으로 이동 */
                               j++;
                }
           }                    /* for j <= totalB + 1문의 끝 */

           for (; a[i].row == row; i++)
                ;
           rowBegin = i; row = a[i].row;
     }     /* for i <= totalA문의 끝 */

     d[0].row = rowsA;
     d[0].col = colsB; d[0].value = totalD;
}
```

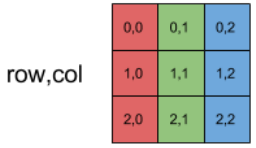
다차원 배열의 표현
(Representations of Multidimensional Arrays)

다차원 배열의 표현(1/3)
(Representation of Multidimensional Arrays)

- 배열의 원소 수 : a[upper$_0$][upper$_1$]….[upper$_{n-1}$]

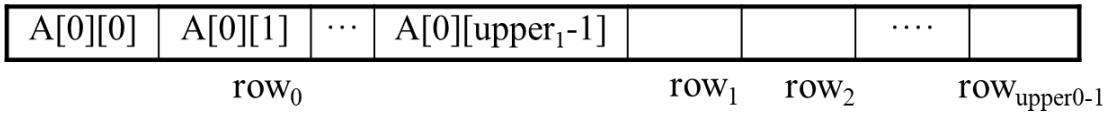
$$\prod_{i=0}^{n-1} upper_i$$

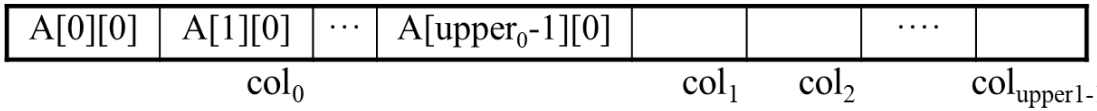
a[10][10][10] → 10x10x10 = 1000



- 다차원 배열 표현 방법
  - 행 우선 순서(row major order)
  - 열 우선 순위(column major order)

행 우선

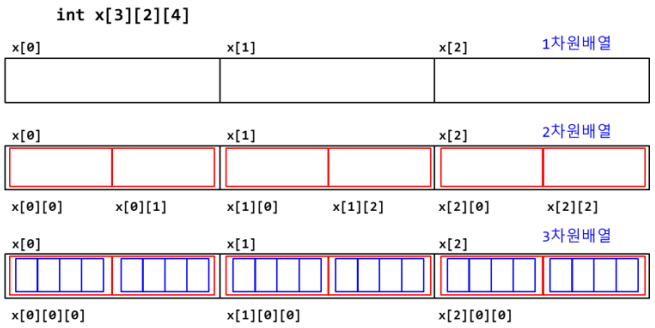




열 우선

다차원 배열의 표현(2/3)
(Representation of Multidimensional Arrays)

Demo: arrayaddr.c



- **주소 계산 (2차원 배열 – 행우선 표현)**
  - α = A[0][0]의 주소
    A[i][0]의 주소 = α + i · $upper_1$
    A[i][j]의 주소 = α + i · $upper_1$ + j

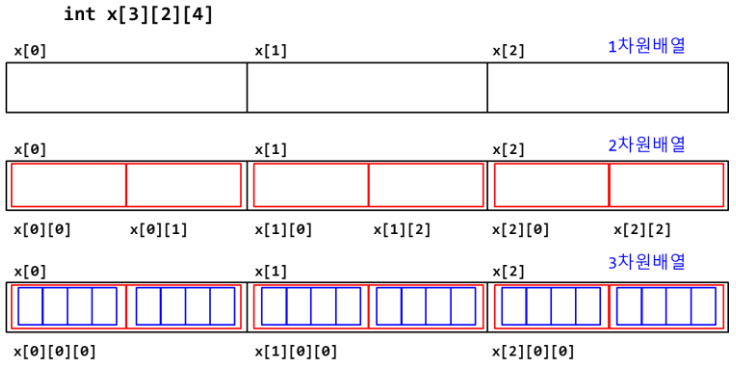
- **주소 계산 (3차원 배열) : A[$upper_0$][$upper_1$][$upper_2$]**
  - 2차원 배열 $upper_1$ x $upper_2$ → $upper_0$개
    A[i][0][0] 주소 = α + i · $upper_1$ · $upper_2$
    A[i][j][k] 주소 = α + i · $upper_1$ · $upper_2$ + j · $upper_2$ + k

- **A[upper0][upper1]…[uppern-1]**
  - α = A[0][0]…[0]의 주소
    $a[i_0][0]…[0]$ 주소 = α + i_0 x $upper_1$ x $upper_2$ x…x $upper_{n-1}$
    $a[i_0][i_1][0]…[0]$ 주소
    = α + i_0 x $upper_1$ x $upper_2$ x…x $upper_{n-1}$ +
    i_1 x $upper_2$ x $upper_3$ x…x $upper_{n-1}$

다차원 배열의 표현(3/3)
(Representation of Multidimensional Arrays)

- **일반화**
  - $a[i_0][i_1]\ldots[i_{n-1}]$ 주소 =

$$\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} + i_1 upper_2 upper_3 \ldots upper_{n-1} + i_2 upper_3 upper_4 \ldots upper_{n-1} + \vdots + i_{n-2} upper_{n-1} + i_{n-1}$$



$$= \sum_{j=0}^{n-1} i_j a_j \quad \text{, 여기에서} \quad \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

스트링 (Strings)

추상 데이터 타입

ADT =
1) 객체정의
2) 객체에 대한 함수 정의

```
ADT String
       object : 0개 이상의 문자들의 유한 집합

function : 모든 s, t ∈ string, i, j, m ∈ 음이 아닌 정수

    string Null(m) ::= NULL로 초기화된 길이가 m인 스트링을 반환

    integer Compare(s, t) ::= if (s와 t가 같으면) return 0
                              else if (s가 t에 선행하면) return -1
                              else return +1

    Boolean IsNull(s) ::= if (Compare(s, NULL)) return FALSE
                          else return TRUE

    Integer Length(s) ::= if (Compare(s, NULL)) s의 문자를 반환
                          else return 0

    string Concat(s, t) ::= if (Compare(s, NULL)) s뒤에 t를 붙인 스트링을 반환
                            else return s

    string Substr(s, i, j) ::= if ((j>0) && (i+j-1)<Length(s))
                                  s에서 i, i+1, i+j-1의 위치에 있는 스트링을 반환
                               else return NULL
```

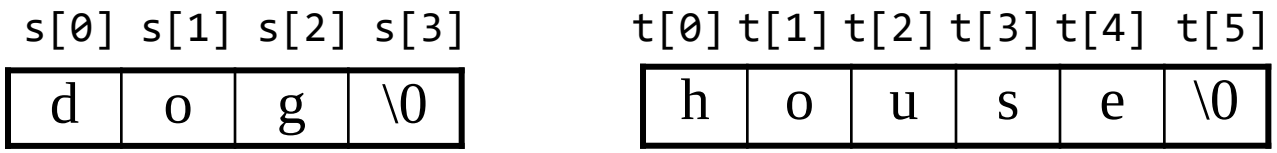
C에서의 스트링 (1)

- 널 문자 '\0'으로 끝나는 문자 배열을 의미

다음과 같은 스트링이 정의되어 있다면,

```
#define MAX_SIZE 100  /* 최대 크기 */
char s[MAX_SIZE]={"dog"};
char t[MAX_SIZE]={"house"};
```

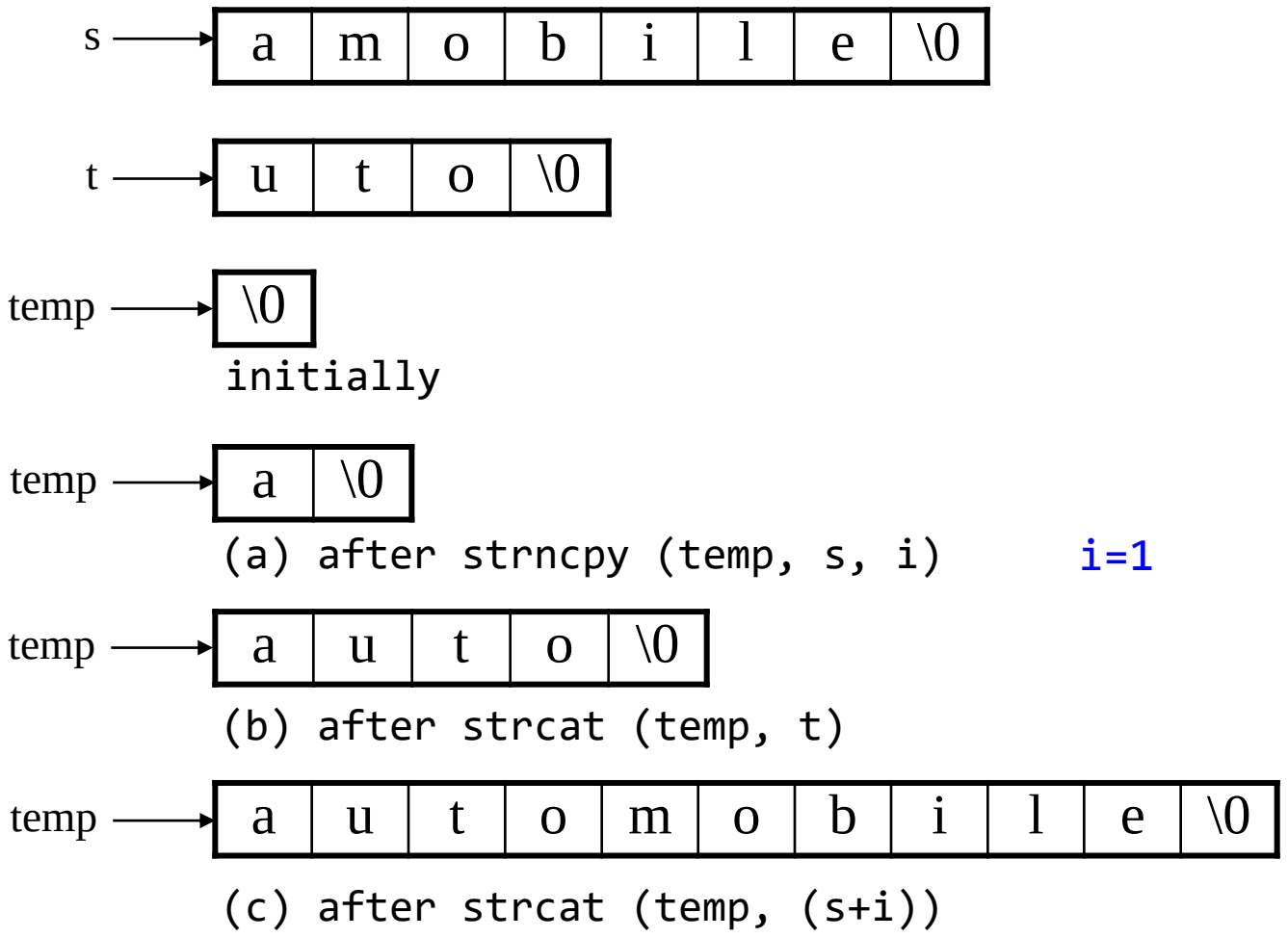
```
char s[]={"dog"};
char t[]={"house"};
```



< C언어에서 스트링 표현 >

C에서의 스트링 (2)

- 스트링 삽입의 예



C에서의 스트링 (3)

- 스트링 삽입 함수

```
void strnins (char *s, char *t, int i)
  {   /* 스트링 s의 i번째 위치에 스트링 t를 삽입 */
            char string[MAX_SIZE], *temp = string;

            if (i < 0 && i > strlen(s) ) {
                     fprintf(strerr, "Position is out of bounds \n");
                     exit(1);
            }
            if (!strlen(s))                    s가 비어있다면, 그대로 복사
                      strcpy(s, t);
            else if (strlen(t)) {
                      strncpy (temp, s, i);    s의 i번째 문자까지 temp에 복사
                      strcat (temp, t);
                      strcat (temp,  (s + i));
                      strcpy (s, temp);
            }
    }
```

패턴매칭(Pattern Matching)(1)

- **좀 더 복잡한 스트링 응용**
  - ‘pat’는 string을 탐색하기 위한 패턴이라고 가정
  - pat과 string 내에 있는지 결정하는 가장 쉬운 방법 → 내장 함수 strstr의 사용
  - ‘pat가’ string 내에 있는지 식별하는 명령문

```
if (t = strstr(string, pat))
    printf(“The string from strstr is : %s\n”, t);
else
    printf(“The pattern was not found with strstr\n”);
```

  - strstr이 패턴 매칭에 아주 적합한 것처럼 보이지만 패턴매칭 구현방법은 다양
  - 한가지 방법: 패턴을 발견할 때 까지 패턴과 문자열을 처음부터 비교해 나가는 것 → O(n*m), 여기에서 n=strlen(pattern), m=strlen(string)

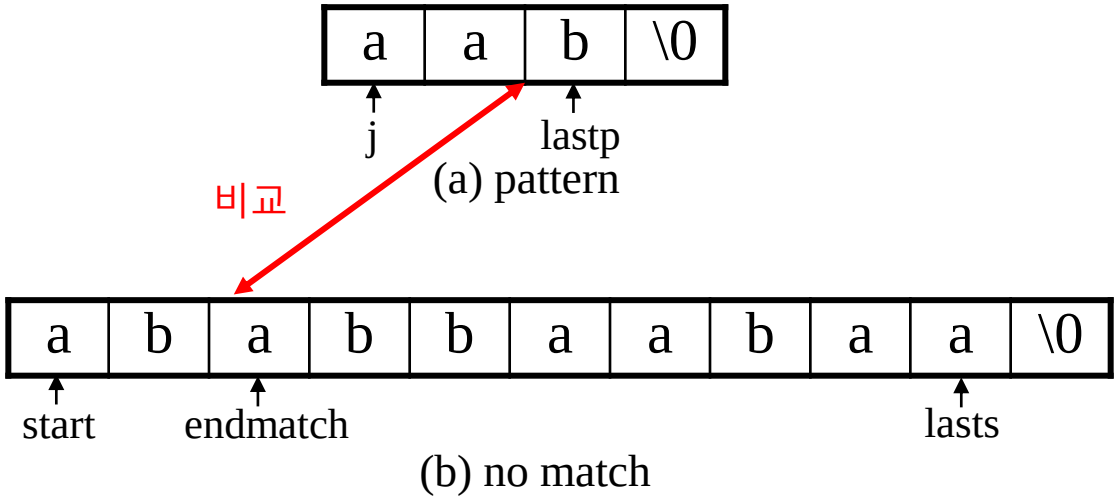
패턴매칭(Pattern Matching)(2)

- **개선된 패턴매칭**
  - strlen(pat)이 string 나머지 문자 수보다 큰 경우 종료 → 성능개선
  - pat와 string의 첫 번째 문자와 마지막 문자를 검사 → 성능개선

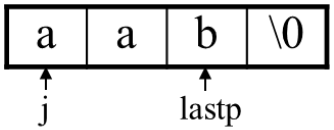
```
int nfind(char *string, char *pat)
{/* 먼저 패턴의 마지막 문자를 매치시켜 본 뒤에, 처음부터 매치시킨다. */
      int i, j, start = 0;
      int lasts = strlen(string) – 1;
      int lastp = strlen(pat) – 1;
      int endmatch = lastp;

      for (i = 0; endmatch <= lasts; endmatch++, start++) {
          if (string[endmatch] == pat[lastp])
              for (j = 0; i = start; j < lastp && start[i] == pat[j]; i++, j++)
                                    ;
          if (j == lastp)
               return start;    /* 성공 */
      }
      return -1;
}
```

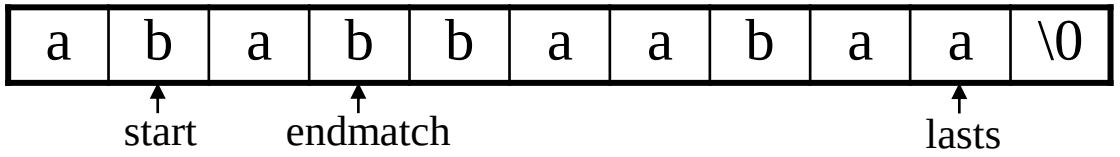
패턴매칭(Pattern Matching)(3)

- **nfind 시뮬레이션**
  - pat = “aab” 이고 string = “ababbaabaa” 으로 가정
  - string과 pat 배열의 끝을 각각 lasts와 lastp 가리키게 함
  - string[endmatch]와 pat[lastp]를 비교
  - 매칭 되면
    - nfind는 매치 되지 않는 경우가 발생하거나
    - pat가 모두 매치될 때까지 두 스트링 이동을 위해 i, j 사용
  - 변수 start는 매치 되지 않을 경우 i를 재설정하기 위해 사용

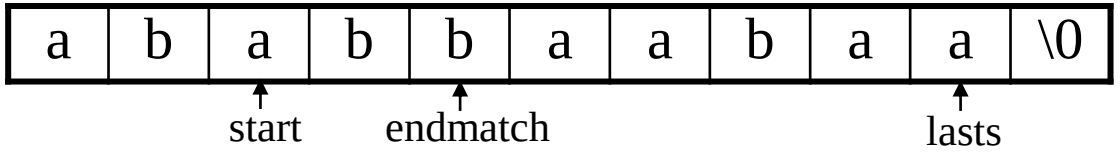


패턴매칭(Pattern Matching)(4)

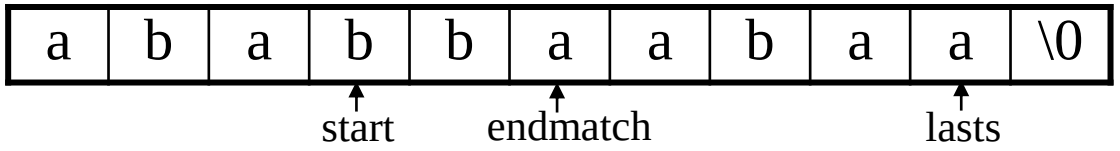




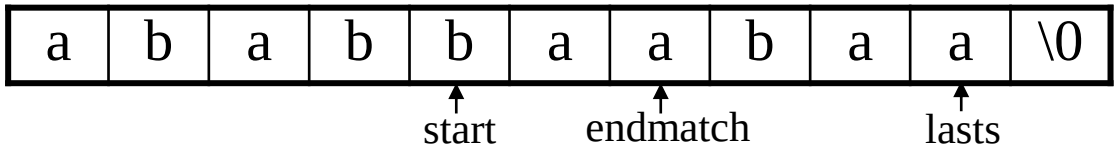
(c) no match



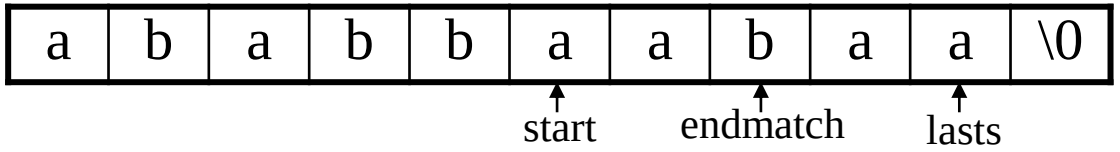
(d) no match



(e) no match



(f) no match



(g) match