

2. 배열 (Arrays)

한국항공대학교 안준선

배열의 개념

- 연속된 메모리 공간을 차지하는 같은 타입(type)의 자료 값들의 집합

0	1	2	3	4	5	6	7	8	9
v0	v1	v2	v3	v4	v5	v6	v7	v8	v9

- 일정 범위의 연속된 정수 인덱스와 해당 값의 쌍들의 집합
 - i번째 원소를 상수 시간에 접근 가능함
 - 삽입 연산의 경우 $O(n)$ 만큼의 시간이 소요됨
 - 할당된 크기(전체 원소의 개수)를 바꾸기 어려움
- 자료구조 정의
 - 원소 : 해당 데이터 타입(T) 원소들의 집합
 - 연산
 - 생성 (create) : $(\langle T \rangle, \text{Int}) \rightarrow \text{Array}^T$
 - 읽어오기 (read, retrieve) : $(\text{Array}^T, \text{Int}) \rightarrow T$
 - 저장 (store) : $(\text{Array}^T, \text{Int}, T) \rightarrow \text{Array}^T$

C에서의 배열 == 포인터

```
<type> arrayname[n];
```

- <type> 타입 원소 n개를 저장한 배열을 생성
- 변수 arrayname은 <type> 타입에 대한 포인터 타입이며 즉, 첫 번째 <type> 타입 원소의 주소값을 가진다.

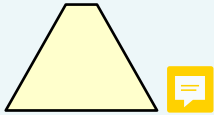
— 배열의 규칙

$\text{<???>}[i] \Leftrightarrow *(<???>+i)$

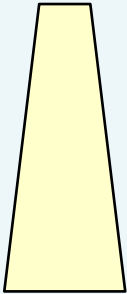
- 인덱스 범위의 제한이 없다 ! ☺ ☹

C에서의 배열: 여러 개의 원소 집합으로 확장

`int x;` // x는 int를 저장하는 변수



`int y[10];` // y는 x같은 것 10개를 저장하는 배열
 // y는 y[0]의 주소(int 포인터)를 가진다.
 // y+1은 y보다 4가 크다.



`int z[5][10];` // z는 y같은 것 5개를 저장한 배열
 // z는 z[0]의 주소를 가진다.
 // z+1은 z보다 40이 크다.
 // z[0]는 z[0][0]의 주소를 가진다.



C에서의 배열

■ 1차원 배열

– C implementation

```
int array[10] = {1,2,3,4,5,6,7,8,9,10};
```

- 연속된 메모리 공간에 저장되며 array는 원소(정수)에 대한 포인터 타입이다.

» array : 1을 가리킴 (array[0] = *(array+0))

» array+2 : 3을 가리킴

■ 2차원 배열

– 배열 원소를 2차원으로 저장 (2개의 첨자가 필요함)

```
int array2[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

- 2차원 배열=1차원 배열의 배열 (1차원 배열에 대한 포인터)
- array2는{1,2,3,4}를 가리킴 (array2[0]== {1,2,3,4})
- array2[0], array2[1], array2[2] 는 각각 정수 4개를 가진 배열임
- array2[1][2] == <{5,6,7,8}>[2] == 7

배열 인자의 전달

- 1차원 배열 인자의 전달: 첫번째 원소의 주소가 전달됨

```
#include <stdio.h>
#define MAX 10
int average(int a[], int n){
    int sum = 0;
    int i;
    for (i = 0; i < n; i++)
        sum += a[i];

    return (sum / n);
}

void main(void) {
    int array[MAX];
    int i;
    printf("\nInput %d integer -> ", MAX);
    for (i = 0; i < MAX; i++)
        scanf("%d", array + i);
    printf("\nAverage of %d integer is %d",
        MAX, average(array, MAX));
}
```

배열 인자의 전달

- 2차원 배열 인자의 전달
 - 포인터(시작 주소)만 받아서는 $a[i][j]$ 를 접근할 수 없다!

int a[][]



Where is $a[1][2]$?

- 각 행의 길이를 알아야 한다.

$$\text{int } a[N][M] \rightarrow \&(a[i][j]) = \&(a[0][0]) + i * M + j$$

배열 인자의 전달

- 2차원 배열 인자의 전달 (계속)
 - 차원의 크기를 명시한다.

```
int sum_matrix(int m[][4], int n) {  
    int i, j, s = 0;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < 4; j++)  
            s = s+m[i][j];  
    return s;  
}  
  
void main(void) {  
    int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};  
    int b[2][4] = {{1,2,3,4},{5,6,7,8}}  
  
    printf("\nSum of a: %d", sum_matrix(a,3));  
    printf("\nSum of b: %d", sum_matrix(b,2));  
}
```


배열 인자의 전달

■ 2차원 배열 인자의 전달 (계속)

- `sum_matrix` 함수의 타입은?

```
int sum_matrix(int *(m[4]), int n)
```

```
int sum_matrix(int (*m)[4], int n)
```



- 다차원인 경우에는 마찬가지로 방법이 사용된다.

```
int func(int t[][4][2]) {    ...    }
```

배열 인자의 전달

■ 일반적인 n차원 배열의 전달

- 첫 번째 원소의 위치와 각 차원의 크기를 인자로 받아 각 원소의 주소를 계산

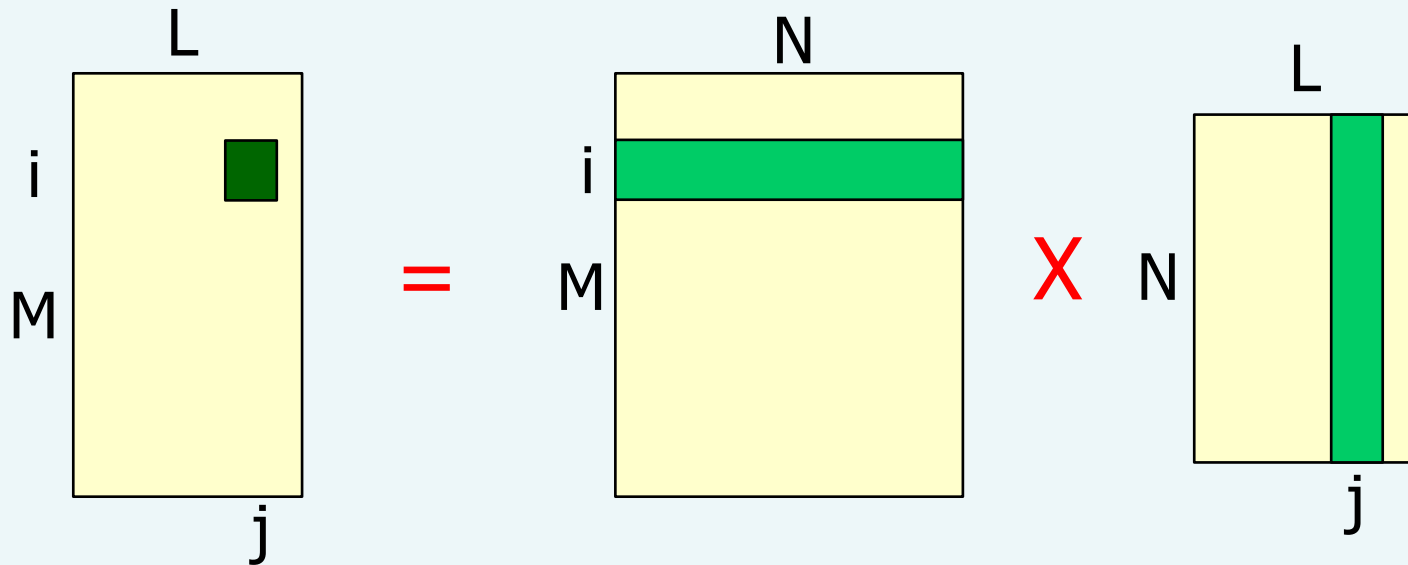
```
int sum_matrix(int *, int n, int l) {
    int i, j, s = 0;
    for (i = 0; i < n; i++)
        for (j = 0; j < l; j++)
            s = s + *(m+(i*l+j));    // m[i*l+j]
    return s;
}
void main(void) {
    int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    printf("\nSum of a: %d", sum_matrix(a,3, 4));  

}
```

- 각 차원의 크기가 $l_1, l_2, l_3, \dots, l_n$ 인 n차원 행렬 a의 원소의 경우

$$\begin{aligned} \&(a[i_1][i_2][i_3] \dots [i_n]) = \&(a[0][0] \dots [0]) + i_1 * (l_2 * l_3 * \dots l_n) \\ &+ i_2 * (l_3 * l_4 * \dots l_n) + \dots + i_{n-1} * l_n + i_n \end{aligned}$$

배열 인자의 전달

■ 행렬의 곱셈



$$\forall i, j: A_{i,j} = \sum_{k=1}^N B_{i,k} \times C_{k,j}$$

배열 인자의 전달

- 행렬의 곱셈

```
/* n == MAX */
void multiply_matrix(int m1[][MAX], int m2[][MAX],
                    int m3[][MAX], int n)
{
    int i, j, k;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            m3[i][j] = 0;
            for (k = 0; k < n; k++)
                m3[i][j] += m1[i][k] * m2[k][j];
        }
}
```

→ Asymptotic Time Complexity : $O(n^3)$

배열 인자의 전달

- 행렬의 곱셈

```
/* m1 :m*n, m2:n*1, m3:m*1 array */
void multiply_matrix(int m1[], int m2[],
                    int m3[], int m, int n, int l)
{
    int i, j, k;

    for (i = 0; i < m; i++)
        for (j = 0; j < l; j++) {
            m3[i*l+j] = 0;
            for (k = 0; k < n; k++)
                m3[i*l+j] += m1[i*n+k]*m2[k*l+j];
        }
}
```

→ Asymptotic Time Complexity : $O(n^3)$

Sparse Matrix (희소 행렬)

■ Sparse Matrix

- 행렬의 0이 아닌 원소가 적은 경우 메모리와 수행 시간 낭비 발생
예)
 - 3차원 행렬로 표현된 교실 안의 공간에서 먼지가 있는 위치
 - 페이스북 친구 관계
 - 각 웹페이지에 포함된 단어 (행:모든 웹페이지, 열: 모든 단어)
- 2차원 배열을 사용한 희소 행렬의 표현
 - 첫 행에는 각 차원의 크기와 0이 아닌 원소의 개수 저장
 - 다음 행부터 0이 아닌 원소만 인덱스 값과 해당 값을 행 우선 순위로 저장
 - M차원 행렬에서 0이 아닌 원소의 개수가 n일 경우 $(M+1) \times (n+1)$ 배열 공간 필요

Sparse Matrix

■ Sparse Matrix

예) $(0,0), (0,4), (1,3), (2,3), (2,5), (3,0), (4,5), (5,1)$ 위치에 각각 1, 2, 3, 4, 5, 6, 7, 8이 저장된 2차원 6x6 배열

0	6	6	8
1	0	0	1
2	0	4	2
3	1	3	3
4	2	3	4
5	2	5	5
6	3	0	6
7	4	5	7
8	5	1	8

Sparse Matrix

- Sparse Matrix에 대한 연산 - 배열 전치(transpose)

```
void transpose(int a[][3], int b[][3]) {
    int n=a[0][1]; int t=a[0][2];
    int col,p;
    int q=1;
    b[0][0]=n; b[0][1]=a[0][0]; b[0][2]=t;
    if (t==0) return;
    for (col=0; col<n; col++)
        for (p=1; p<=t; p++)
            if (a[p][1] == col) {
                b[q][0] = a[p][1];
                b[q][1] = a[p][0];
                b[q][2] = a[p][2];
                q++;
            }
}
```

→ Asymptotic Time Complexity : $O(nt)$
 (which can be $O(n^2m)$)

0	6	6	8
1	0	0	1
2	0	4	2
3	1	3	3
4	2	3	4
5	2	5	5
6	3	0	6
7	4	5	7
8	5	1	8

0	6	6	8
1	0	0	1
2	0	3	6
3	1	5	8
4	3	1	3
5	3	2	4
6	4	0	2
7	5	2	5
8	5	4	7

Sparse Matrix

– transpose 함수의 개선

```
void transpose(int a[][3], int b[][3]) {
    int n=a[0][1]; int terms = a[0][2];
    int *s = (int *)calloc(n,sizeof(int));
    int *t = (int *)calloc(n,sizeof(int));
    int i,j;
    b[0][0]=n; b[0][1]=a[0][0]; b[0][2]=terms;
    for (i=1; i<=terms; i++) s[a[i][1]]++;
    t[0] = 1;
    for (i=1; i<n; i++) t[i]=t[i-1]+s[i-1];
    for (i=1; i<=terms; i++) {
        j = t[a[i][1]];
        b[j][0] = a[i][1];
        b[j][1] = a[i][0];
        b[j][2] = a[i][2];
        t[a[i][1]] = j+1;
    }
}
```

→ Asymptotic Time Complexity : $O(t)$

0	6	6	8	2	0
1	0	0	1	1	1
2	0	4	2	0	2
3	1	3	3	2	3
4	2	3	4	1	4
5	2	5	5	2	5
6	3	0	6		
7	4	5	7		
8	5	1	8		

s

0	6	6	8	1	0
1	0	0	1	3	1
2	0	3	6	4	2
3	1	5	8	4	3
4	3	1	3	6	4
5	3	2	4	7	5
6	4	0	2		
7	5	2	5		
8	5	4	7		

t