C 복습

- calloc, realloc

함수 **calloc()**

- 초기 값을 0으로 하는 동적 할당 함수
  - stdlib.h 헤더 파일에 함수원형 정의

```
void *   calloc(   size_t,   size_t   );
```

할당될 메모리 항목의 갯수이다

할당될 메모리 한 원소의 크기이다.

  - 배열원소의 초기값 0인 int형 배열

```
int *ary;
ary = (int *) calloc( 3, sizeof(int) );
```

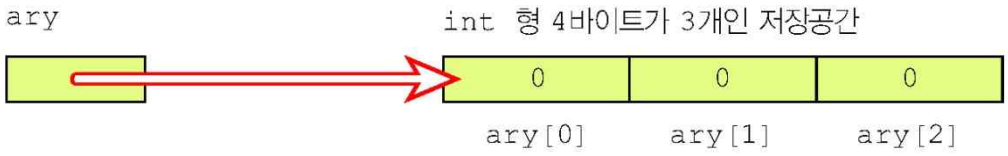


그림 18.5 함수 calloc( )에 의한 저장공간의 할당

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

int main(void)
{
    int *mary, *cary;
    int i = 0;

    printf("malloc()은 초기화를 하지 않는다. >>₩n");
    mary = (int *) malloc( 2*sizeof(int) );
    if (mary == NULL)
    {
        printf("메모리 할당이 문제가 있습니다.₩n");
        exit(EXIT_FAILURE);
    }
    printf("mary 주소는 %#X이다. >>₩n", mary);
    for (i = 0; i < 2; i++)
        printf("mary[%d](%#x) = %d ", i, mary + i, *(mary + i));
    printf("₩n");
    printf("배열에 직접 값을 입력한 후 >>₩n");
    mary[0] = 100; mary[1] = 200;
    for (i = 0; i < 2; i++)
        printf("mary[%d](%#x) = %d ", i, mary + i, *(mary + i));
    printf("₩n₩n");
    free(mary);

    printf("calloc()은 초기화를 0으로 한다. >>₩n");
    cary = (int *) calloc( 2, sizeof(int) );
    if (cary == NULL)
    {
        printf("메모리 할당이 문제가 있습니다.₩n");
        exit(EXIT_FAILURE);
    }
    printf("cary 주소는 %#X이다. >>₩n", cary);
    for (i = 0; i < 2; i++)
        printf("cary[%d](%#x) = %d ", i, cary + i, *(cary + i));
    printf("₩n");
    free(cary);
```

```
C:₩Windows₩system32₩cmd.exe
malloc()은 초기화를 하지 않는다. >>
mary 주소는 0X781460이다. >>
mary[0](0x781460) = -842150451 mary[1](0x781464) = -842150451
배열에 직접 값을 입력한 후 >>
mary[0](0x781460) = 100 mary[1](0x781464) = 200

calloc()은 초기화를 0으로 한다. >>
cary 주소는 0X781460이다. >>
cary[0](0x781460) = 0 cary[1](0x781464) = 0
```

```
malloc()은 초기화를 하지 않는다. >>
mary 주소는 0X781460이다. >>
mary[0](0x781460) = -842150451 mary[1](0x781464) = -842150451
배열에 직접 값을 입력한 후 >>
mary[0](0x781460) = 100 mary[1](0x781464) = 200
```

```
int *mary, *cary;
int i = 0;

printf("malloc()은 초기화를 하지 않는다. >>₩n"
mary = (int *) malloc( 2*sizeof(int) );
if (mary == NULL)
{
    printf("메모리 할당이 문제가 있습니다.₩n");
    exit(EXIT_FAILURE);
}
printf("mary 주소는 %#X이다. >>₩n", mary );
for (i = 0; i < 2; i++)
    printf("mary[%d](%#x) = %d ", i, mary + i  *(mary + i));
printf("₩n");
printf("배열에 직접 값을 입력한 후 >>₩n");
mary[0] = 100  mary[1] = 200;
for (i = 0; i < 2; i++)
    printf("mary[%d](%#x) = %d ", i, mary + i, *(mary + i));
printf("₩n₩n");
free(mary);
```

```c
printf("calloc()은 초기화를 0으로 한다. >>₩n");
cary = (int *) calloc( 2, sizeof(int) );
if (cary == NULL)
{
    printf("메모리 할당이 문제가 있습니다.₩n");
    exit(EXIT_FAILURE);
}
printf("cary 주소는 %#X이다. >>₩n", cary);
for (i = 0; i < 2; i++)
    printf("cary[%d](%#x) = %d ", i, cary + i, *(cary + i));
```

```
calloc()은 초기화를 0으로 한다. >>
cary 주소는 0X781460이다. >>
cary[0](0x781460) = 0 cary[1](0x781464) = 0
```

함수 realloc()

- 이미 확보한 저장공간을 새로운 크기로 변경
  - 기존 영역을 이용하여 변경하거나, 새 영역 할당 후 이전 값을 복사할 수도 있음

```
void *  realloc(void *, size_t);
```

  - 첫 번째 인자 – 변경할 저장공간의 주소
  - 두 번째 인자 - 변경하고 싶은 저장공간의 크기
  - 첫 번째 인자가 NULL이면 함수 malloc()과 같은 기능을 수행

```
int *reary, *cary;
cary = (int *) calloc( 3, sizeof(int) );
```

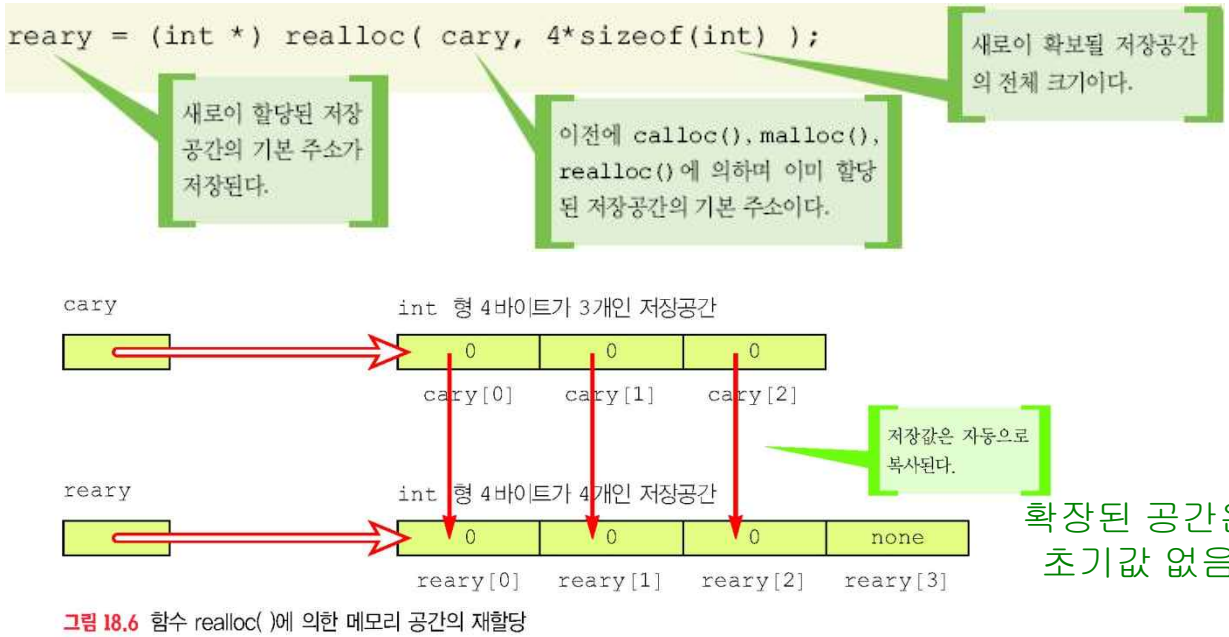


그림 18.6 함수 realloc( )에 의한 메모리 공간의 재할당

확장된 공간은 초기값 없음

```
reary = (int *) realloc( cary, 4*sizeof(int) );
```

새로이 할당된 저장 공간의 기본 주소가 저장된다.

이전에 calloc(), malloc(), realloc()에 의하며 이미 할당된 저장공간의 기본 주소이다.

새로이 확보될 저장공간의 전체 크기이다.

그림 18.6 함수 realloc( )에 의한 메모리 공간의 재할당

```
C:₩windows₩system32₩cmd.exe
calloc()은 초기화를 0으로 한다. >>
cary 주소는 0X401460이다. >>

realloc()은 기존의 값이 남아 있다. >>
realloc() 호출 후에 >>
cary 주소는 0X401460이다. >>
reary 주소는 0X401460이다. >>
cary[0](0X401460) = 10 cary[1](0X401464) = 11
reary[0](0X401460) = 10 reary[1](0X401464) = 11
reary[2](0X401468) = 30 reary[3](0X40146C) = -842150451
reary[4](0X401470) = 40 reary[5](0X401474) = 50
```

```c
/* realloc.c */

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

int main(void) {
    int *reary, *cary;
    int i = 0;

    printf("calloc()은 초기화를 0으로 한다. >>₩n");
    cary = (int *) calloc( 2, sizeof(int) );
    if (cary == NULL)
    {
        printf("메모리 할당이 문제가 있습니다.₩n");
        exit(EXIT_FAILURE);
    }
    printf("cary 주소는 %#X이다. >>₩n₩n", cary);
```

cary (0x401460)

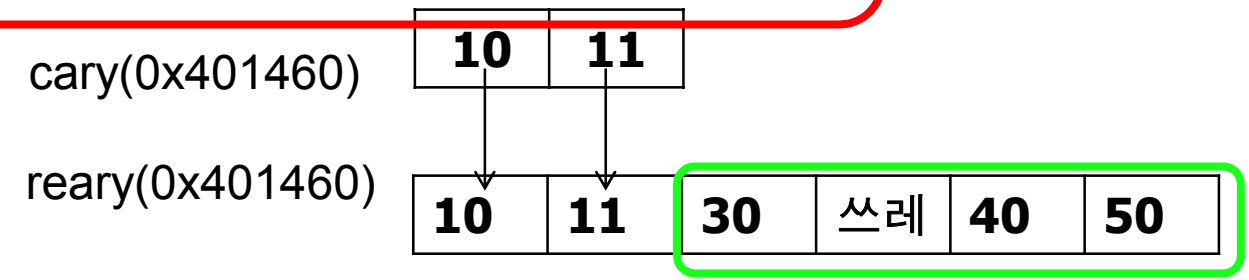
0	0

```
cary[0] = 10; cary[1] = 11;

printf("realloc()은 기존의 값이 남아 있다. >>₩n");
reary = (int *) realloc( cary, 6*sizeof(int) );
if (reary == NULL)
{
    printf("메모리 할당이 문제가 있습니다.₩n");
    exit(EXIT_FAILURE);
}
printf("realloc() 호출 후에 >>₩n");
printf("cary 주소는 %#X이다. >>₩n", cary);
printf("reary 주소는 %#X이다. >>₩n", reary);
reary[2] = 30; //reary[3]에는 저장을 하지 않음
reary[4] = 40; reary[5] = 50;
```

cary(0x401460)	10	11

```
C:₩windows₩system32₩cmd.exe
calloc()은 초기화를 0으로 한다. >>
cary 주소는 0X401460이다. >>

realloc()은 기존의 값이 남아 있다. >>
realloc() 호출 후에 >>
cary 주소는 0X401460이다. >>
reary 주소는 0X401460이다. >>
cary[0](0X401460) = 10 cary[1](0X401464) = 11
reary[0](0X401460) = 10 reary[1](0X401464) = 11
reary[2](0X401468) = 30 reary[3](0X40146C) = -842150451
reary[4](0X401470) = 40 reary[5](0X401474) = 50
```

```
printf("realloc() 호출 후에 >>₩n");
printf("cary 주소는 %#X이다. >>₩n", cary);
printf("reary 주소는 %#X이다. >>₩n", reary);
reary[2] = 30; //reary[3]에는 저장을 하지 않음
reary[4] = 40; reary[5] = 50;
  for (i = 0; i < 2; i++)
      printf("cary[%d](%#X) = %d ", i, cary + i, *(cary + i));
  printf("₩n");
  for (i = 0; i < 6; i++)
  {
      printf("reary[%d](%#X) = %d ", i, reary + i, *(reary + i));
      if (i%2 == 1) printf("₩n");
  }
  free(reary);

  return 0;
```




```
realloc()은 기존의 값이 남아 있다. >>
realloc() 호출 후에 >>
cary 주소는 0X401460이다. >>
reary 주소는 0X401460이다. >>
cary[0](0X401460) = 10 cary[1](0X401464) = 11
reary[0](0X401460) = 10 reary[1](0X401464) = 11
reary[2](0X401468) = 30 reary[3](0X40146C) = -842150451
reary[4](0X401470) = 40 reary[5](0X401474) = 50
```

Arrays and Structures

Chapter 2

Contents

Data structure

- Data structure ≡ data type + storage structure
- Data type
  - How to categorize data objects
  - **Object + operator**
- Abstract Data Type (ADT)
  - Specification of objects and operations independent of their implementation

Performance of a program largely depends on data structures !!

Abstract Data Type (ADT) of Array

- A set of pairs <index, value>
- ADT for array provides operations that
  - retrieves a value
  - stores a value

ADT *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, $\cdots$, n–1} for one dimension, {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.

functions:

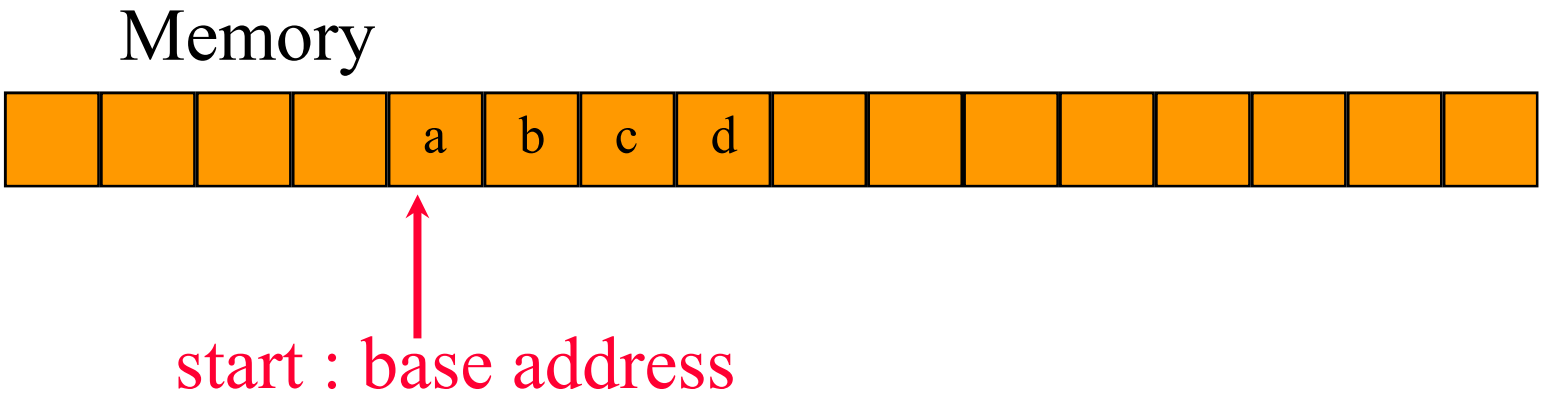
for all $A \in$ *Array*, $i \in$ *index*, $x \in$ *item*, j, *size* $\in$ integer

Array Create(j, *list*)	::=	**return** an array of j dimensions where *list* is a j-tuple whose ith element is the the size of the ith dimension. *Items* are undefined.
Item Retrieve(A, i)	::=	**if** ($i \in$ *index*) **return** the item associated with index value i in array A **else return** error
Array Store(A,i,x)	::=	**if** (i in *index*) **return** an array that is identical to array A except the new pair <i, x> has been inserted **else return** error.

end *Array*

ADT 2.1: Abstract Data Type *Array*

1D Array Representation In C



- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations
- location(x[i]) = start + i

1D Array Addressing

- int one[] = {0,1,2,3,4}; print1(,);

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

Address		Contents
Base address: α	1224	0
α+1*sizeof(int)	1228	1
α+2*sizeof(int)	1232	2
α+3*sizeof(int)	1236	3
α+4*sizeof(int)	1240	4

- Pointer and Arrays

  int *list1, list2[5]; list1 = list2;

  – list2 == &list2[0]

  – list2 + i == &list1[i]

  – *(list1+i) == list2[i]

```c
#include <stdio.h>

void print1(int*, int);

void main()
{
    int one[] = { 0, 1, 2, 3, 4 };
    int size = sizeof(one) / sizeof(int);
    print1(one, size);
}

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

```
Address Contents
11335848    0
11335852    1
11335856    2
11335860    3
11335864    4
```

1D Array Program: examples

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;

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

```
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);
}
```

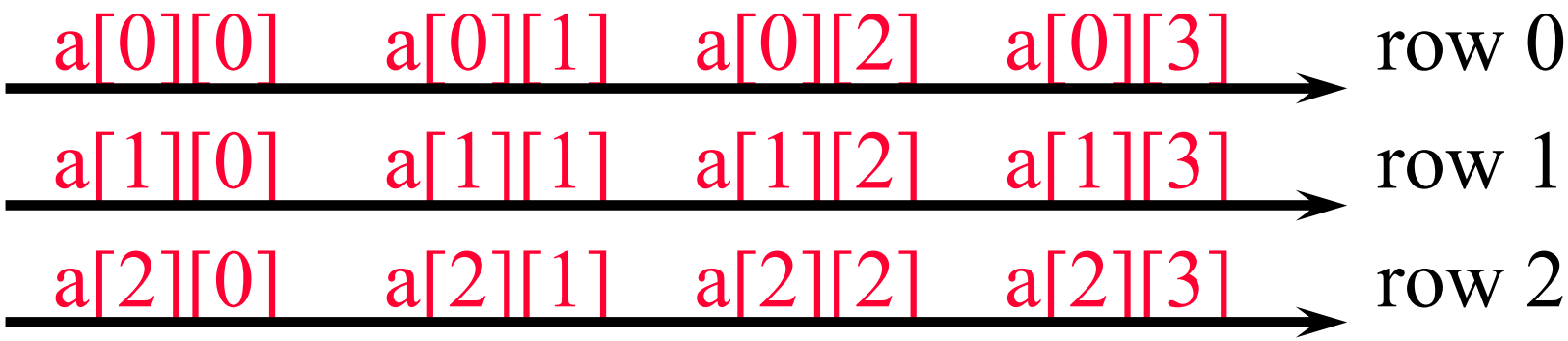
- When sum is called, input (= &input[0]) is stored in a temporary storage
- Dereference (역참조)
  - list[i]가 '=' 우측 : (list+i)가 가리키는 값 반환
  - list[i]가 '=' 좌측 : 값을 (list+i) 위치에 저장
  - (예) list[1]=list[2]

2D Arrays

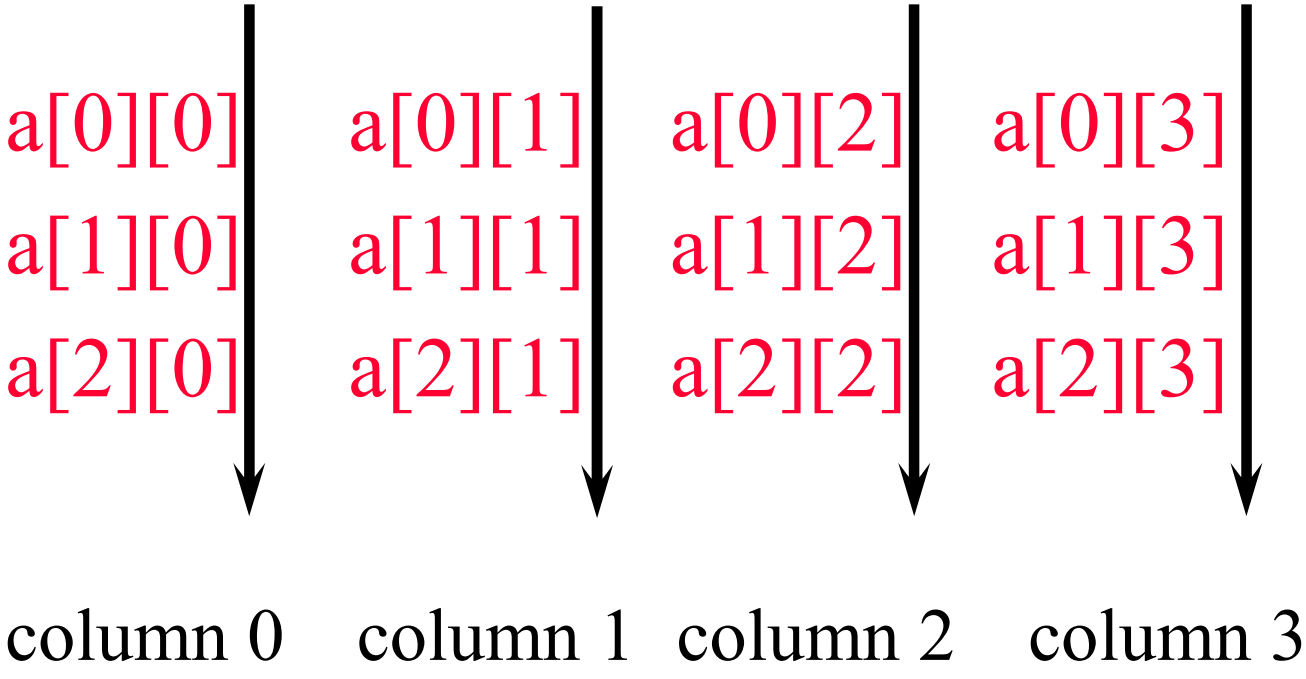
- int a[3][4];
- may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Rows Of A 2D Array



Columns Of A 2D Array



Representation of arrays (1)

- $A[u_1][u_2] \ldots [u_n]$

  - the number of elements :

  - an element $A[i_1][i_2] \ldots [i_n]$ is mapped onto a position in a one-dim C array

- Row major order example

- A[2][3][2][2] : 2*3*2*2 = 24 elements

- stored as A[0][0][0][0], A[0][0][0][1], ..., A[1][2][1][0], A[1][2][1][1]

- translate to locations in the one-dim array

$A[u_1][u_2]\ [u_3]\ [u_4]$

A[0][0][0][0] → position a

A[0][0][0][1] → position a+1

A[i][j][k][m] → position: $a + i*u_2\ u_3\ u_4 + j*u_3\ u_4 + k * u_4 + m$

- int A[2][4][3];
- A[1][2][2]의 경우...
- A[0]
  - A[0][0] 000 001 002
  - A[0][1] 010 011 012
  - A[0][2] 020 021 022
  - A[0][3] 030 031 032
- A[1]
  - A[1][0] 100 101 102
  - A[1][1] 110 111 112
  - A[1][2] 120 121 122
  - A[1][3] 130 131 132

1*4*3+2*3+2

$A[u_1][u_2]\ [u_3]\ [u_4]$

$A[0][0][0][0] \rightarrow$ position a

$A[0][0][0][1] \rightarrow$ position a+1

$A[i][j][k][m] \rightarrow$ position: $a + i*u_2\ u_3\ u_4 + j*u_3\ u_4 + k * u_4 + m$

- A[i] ➔
  - $A[u_2][u_3][u_4]$가 0~ i-1개 있음
- A[][j] ➔
  - $A[u_3][u4]$가 0~ j-1 개 있음
- A[][][k] ➔
  - $A[u_4]$가 0~ k-1개 있음

Representation of arrays (2)

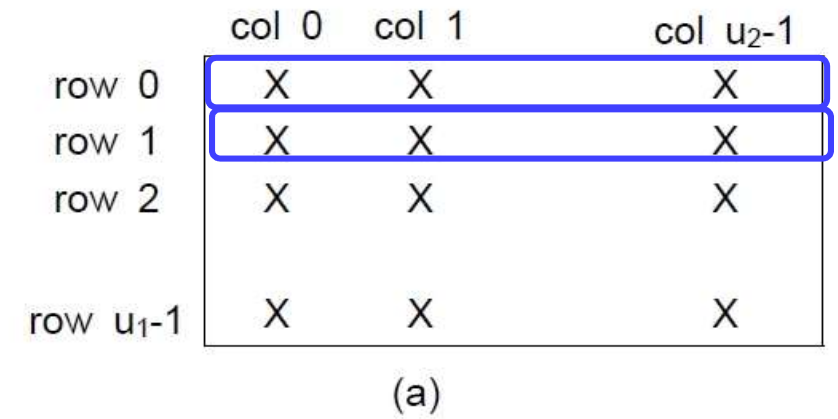
- two-dim array $A[u_1][u_2]$

  let α be the address of A[0][0]

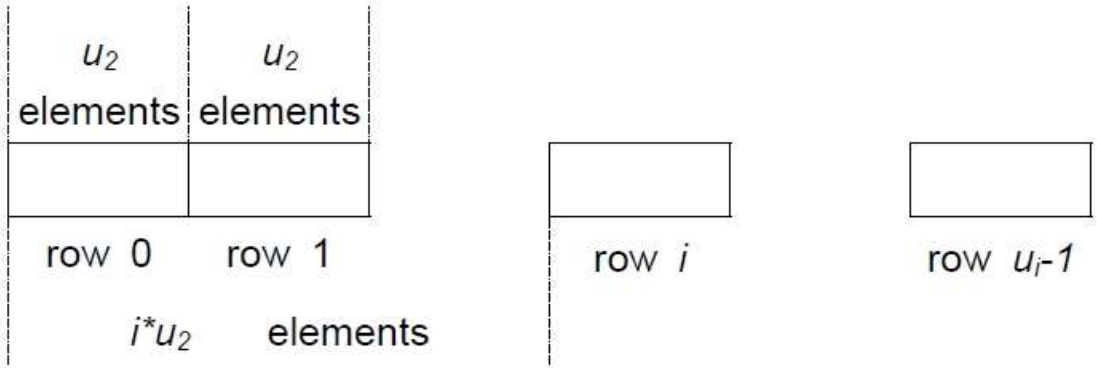
  $A[i][0]$: $\alpha + i*u_2$

  $A[i][j]$: $\alpha + i*u_2 + j$

sizeof(element)=1이라 가정



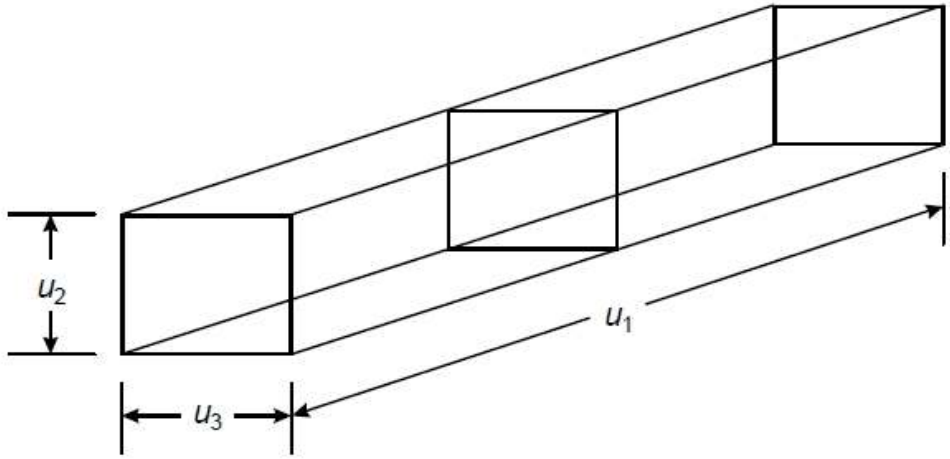
(a)



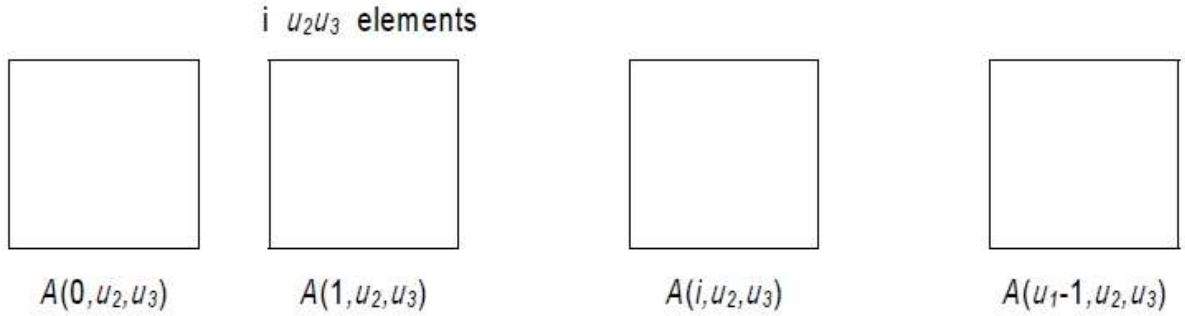
(b)

Representation of arrays (3)

three-dim array $A[u_1][u_2][u_3]$
the address of $A[0][0][0]$: α
$A[i][0][0]$: $\alpha+iu_2u_3$
$A[i][j][k]$: $\alpha+iu_2u_3+ju_3+k$



(a) 3-dimensional array $A[u_1][u_2][u_3]$ regarded as u_1 2-dimensional array



(b) Sequential row major representation of a 3-dimensional array

Representation of arrays (4)

int A[u_0][u_1]…[u_{n-1}]

Repeating in this way the address for $A[i_0][i_1]\ldots[i_{n-1}]$ is:

$$\begin{aligned}
&\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} \\
&\quad + i_1 upper_2 upper_3 \ldots upper_{n-1} \\
&\quad + i_2 upper_3 upper_4 \ldots upper_{n-1} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad + i_{n-2} upper_{n-1} \\
&\quad + i_{n-1}
\end{aligned}$$

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

- int arr[3][4][5][6];

- arr[2][3][1][0]은 &a[0][0][0][0]+ 336 ?
  - int는 4bytes로 가정

  - 2*(4*5*6)+3*(5*6)+1*6+0

Dynamic memory allocation (1)

```
pf = (float *) malloc (sizeof(float));
```

is replaced by

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

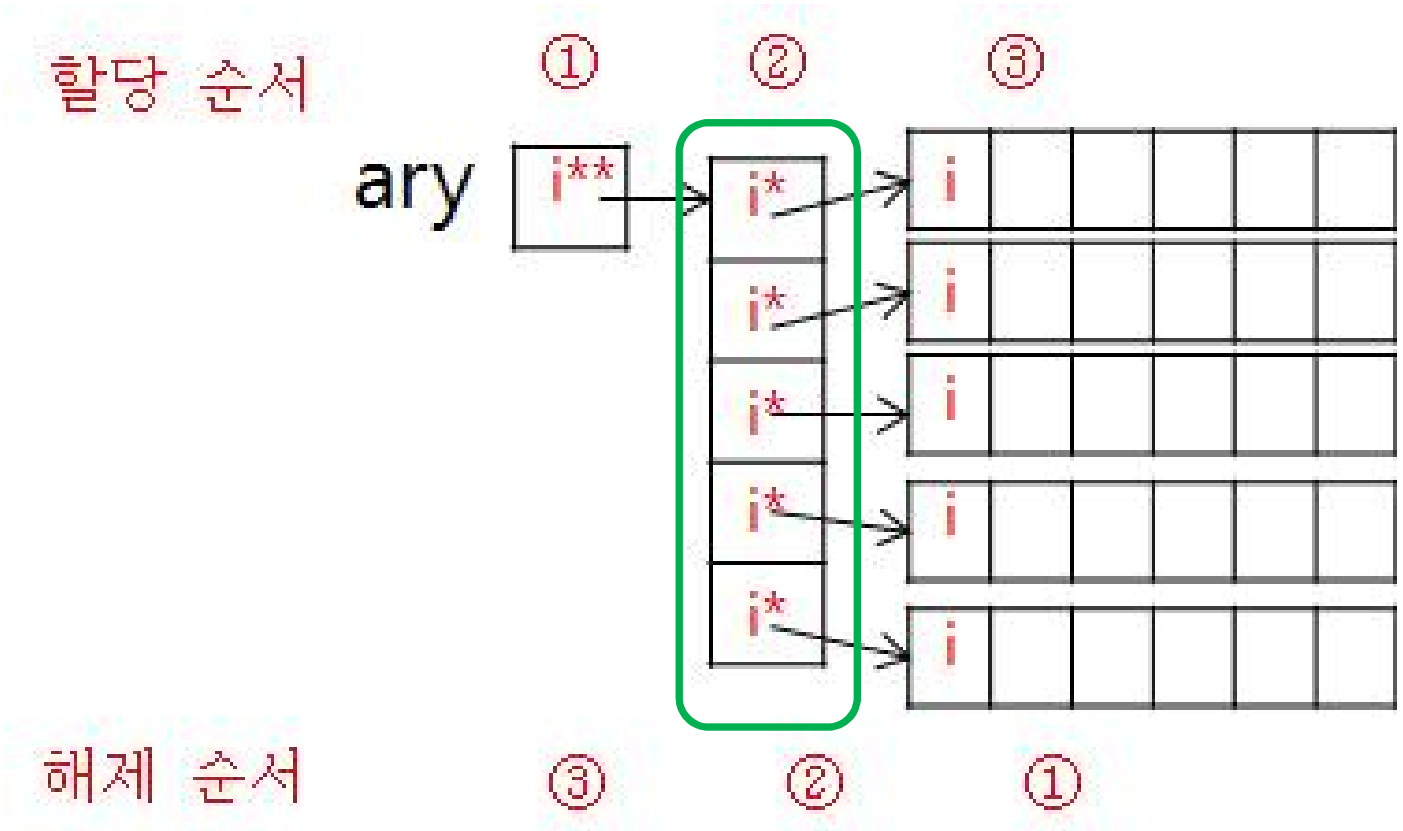
Dynamic memory allocation (2)

- calloc

```
int *x;
x = calloc(n, sizeof(int));
                        // allocated bits are set to 0
```

- realloc

```
realloc(p, s);
            // changes the size of memory block
            // pointed by p to s
```



- int **arr;
- arr=(int**)malloc(sizeof(int*)*5);
- for(int i=0;i<5;i++)
  - arr[i]=(int*)malloc(sizeof(int)*6);

Dynamic memory allocation (3)

```
int **myArray;
myArray = make2dArray(5,10);
myArray[2][4] = 6;
```

```
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;
}
```

Program 2.3: Dynamically create a two-dimensional array

2차원 배열 예제(1)

```
#include <stdlib.h>
#include <stdio.h>
void main()
{
    double **x;
    int i, j, row = 2, col = 3;
    x = (double **)malloc(sizeof(*x)*row);
    printf("sizeof(x) = %d₩n", sizeof(x));
    for (i = 0; i< row; i++) {
        x[i] = (double *)malloc(col * sizeof(**x));
        printf("sizeof(x[%d]) = %d₩n", i, sizeof(x[i]));
        for (j = 0; j < col; j++) {
            x[i][j] = i * 10 + j;
            printf("x[%d][%d] = %f₩n", i, j, x[i][j]);
            printf("sizeof(x[%d][%d]) = %d₩n", i, j, sizeof(x[i][j]));

        }
    }
}
```

double*

double

```
sizeof(x) = 4
sizeof(x[0]) = 4
x[0][0] = 0.000000
sizeof(x[0][0]) = 8
x[0][1] = 1.000000
sizeof(x[0][1]) = 8
x[0][2] = 2.000000
sizeof(x[0][2]) = 8
sizeof(x[1]) = 4
x[1][0] = 10.000000
sizeof(x[1][0]) = 8
x[1][1] = 11.000000
sizeof(x[1][1]) = 8
x[1][2] = 12.000000
sizeof(x[1][2]) = 8
```

2차원 배열 예제(2)

```
sizeof(x) = 4   x = 51729952    sizeof(y)=48  y = 15727296
sizeof(x[0]) = 4   x[0]= 51748776   sizeof(y[0]) = 24   y[0] = 15727296
x[0][0] = 0.000000
x[0][1] = 1.000000
x[0][2] = 2.000000
sizeof(x[1]) = 4   x[1]= 51748848   sizeof(y[1]) = 24   y[1] = 15727320
x[1][0] = 10.000000
x[1][1] = 11.000000
x[1][2] = 12.000000
```

```c
#include <stdlib.h>
#include <stdio.h>
void main()
{
    double **x, y[2][3];
    int i, j, row = 2, col = 3;
    x = (double **)malloc(sizeof(*x)*row);
    printf("sizeof(x) = %d   x = %d    sizeof(y)=%d  y = %d₩n",
        sizeof(x), x, sizeof(y), y);
    for (i = 0; i< row; i++) {
        x[i] = (double *)malloc(col * sizeof(**x));
        printf("sizeof(x[%d]) = %d   x[%d]= %d   sizeof(y[%d]) = %d   y[%d] = %d₩n",
            i, sizeof(x[i]), i, x[i], i, sizeof(y[i]), i, y[i]);
        for (j = 0; j < col; j++) {
            x[i][j] = i * 10 + j;
            printf("x[%d][%d] = %f₩n", i, j, x[i][j]);
        }
    }
}
```

y[0]과 y[1]의 차이는 24.
x[0]과 x[1]의 차이는 일정치 않음.

```c
double **x, y[2][3];
int i, j, row = 2, col = 3;
x = (double **)malloc(sizeof(*x)*row);
printf("sizeof(x) = %d   x = %d    sizeof(y)=%d  y = %d₩n",
    sizeof(x), x, sizeof(y), y);
for (i = 0; i< row; i++) {
    x[i] = (double *)malloc(col * sizeof(**x));
```

```
sizeof(x) = 4    x = 51729952     sizeof(y)=48   y = 15727296
sizeof(x[0]) = 4    x[0]= 51748776    sizeof(y[0]) = 24    y[0] = 15727296

sizeof(x[1]) = 4    x[1]= 51748848    sizeof(y[1]) = 24    y[1] = 15727320
```

Structures

- A structure is a collection of data items
  - Each item is identified as to its type and name

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

  - Examples of the use of the structure member operator

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

Create your own structure using *typedef*

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

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

- Variable declaration
  - humanBeing person1, person2;
- Equality check
  - if (person1 == person2)
- Replacement
  - person1 = person2
  - strcpy(person1.name, person2.name);
    person1.age = person2.age;
    person1.salary = person2.salary;

Usage of structure: example

```
int humansEqual(humanBeing person1,
                humanBeing person2)
{/* return TRUE if person1 and person2 are the same human
    being otherwise return FALSE */
   if (strcmp(person1.name, person2.name))
      return FALSE;
   if (person1.age != person2.age)
      return FALSE;
   if (person1.salary != person2.salary)
      return FALSE;
   return TRUE;
}
```

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

```
humanBeing person1, person2;

if (humansEqual(person1, person2))
   printf("The same people\n");
else
   printf("Different people\n");
```

Embedding a structure within structure

```
typedef struct {
        int month;
        int day;
        int year;
} date ;
typedef struct {
      char name[10];
      int age;
      float salary;
      date dob;
} humanBeing;
```

```
humanBeing person1;
```

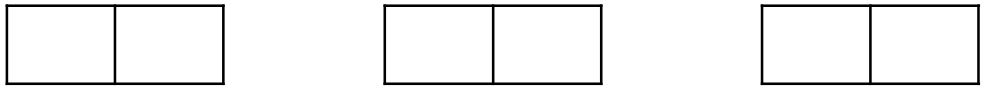
```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

Self-Referential Structures

- A self-referential structure is one in which one or more of its components is **a pointer to itself**.

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

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



```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
item1.link = &item2;   ⇨  link the structures
item2.link = &item3;      (item1→ item2→ item3)
```

Polynomials

- $A(x) = 3x^{20} + 2x^5 + 4$
- $B(x) = x^4 + 10x^3 + 3x^2 + 1$
- A polynomial $P(x) = a_1 x^{e1} + \cdots + a_n x^{en}$ can be considered as an ordered (linear) list.

Polynomials

- Ordered list (linear list): an ordered set of data items.

ex) Days-of-week

(Mon,	Tue,	Wed,	Thu,	Fri,	Sat,	Sun)	: *list*
1st	2nd	3rd	4th	5th	6th	7th	: *order*

- There can be an empty list
- Operations on ordered list

i . finding the length
ii . reading from right to left (or left to right)
iii . retrieve i-th element, 0≤ i<n
iv . update i-th element's value, 0 ≤ i<n
v . insertion (i번째 위치, 0 ≤ i < n) : - i, i+1, ..., n-1 -> i+1, i+2, ..., n
vi . deletion (i번째 항목, 0 ≤ i<n) : - i+1, ..., n-1 -> i, i+1, ..., n-2

before after

ADT of Polynomial

ADT *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all *poly*, *poly*1, *poly*2 ∈ *Polynomial*, *coef* ∈ *Coefficients*, *expon* ∈ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 · *poly*2

end *Polynomial*

ADT 2.2: Abstract data type *Polynomial*

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

ADT *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all *poly*, *poly*1, *poly*2 ∈ *Polynomial*, *coef* ∈ *Coefficients*, *expon* ∈ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted

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

Polynomial Remove(*poly*, *expon*) ::= **if** (*expon* ∈ *poly*)
return the polynomial *poly* with the term whose exponent is *expon* deleted
else return error

Polynomial SingleMult(*poly*, *coef*, *expon*) ::= **return** the polynomial $poly \cdot coef \cdot x^{expon}$

Polynomial Add(*poly*1, *poly*2) ::= **return** the polynomial *poly*1 + *poly*2

Polynomial Mult(*poly*1, *poly*2) ::= **return** the polynomial *poly*1 · *poly*2

Polynomial Representation

- Principle
  - Unique exponents are arranged in decreasing order

Polynomial Representation(1)

- Use *typedef* to creat the type *polynomial*

$$A(x) = 2x^{1000} + 1 \text{ and } B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$$

```
#define MAX_DEGREE 101 /*Max degree of polynomial+1*/
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

a.coef[i]: 차수i의 계수

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

- leads to a very simple algorithms for many of the operations on polynomials
- Wastes computer memory

$A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$

```
#define MAX_DEGREE 101 /*Max degree of polynomial+1*/
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

- polynomial A, B;
- A.coef[0], A.coef[1], ..., A.coef[1000]
- B.coef[0], A.coef[1], ..., B.coef[4]

Polynomial Representation(2)

- Store only non-zero exponents
- All polynomials are represented in a single array called terms

```
#define MAX_TERMS 100    /* size of terms array */
typedef struct {
        float coef;
        int expon;
    } polynomial;

polynomial terms[MAX_TERMS];
int avail = 0;
int starta, finisha
```

Polynomial Repre

$A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$

```
#define MAX_TERMS 100    /* size c
typedef struct {
    float coef;
    int expon;
} polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
int starta, finisha
```

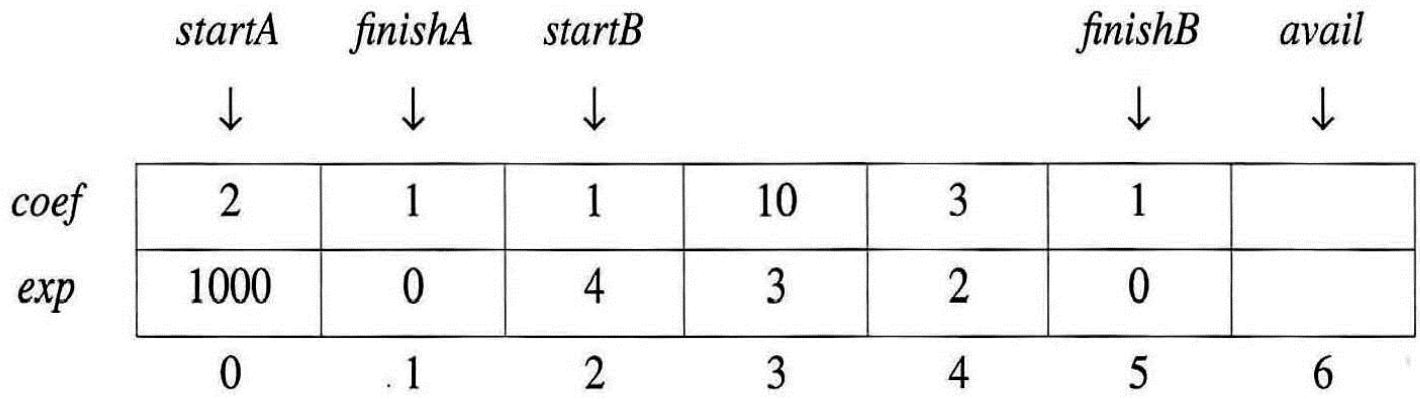


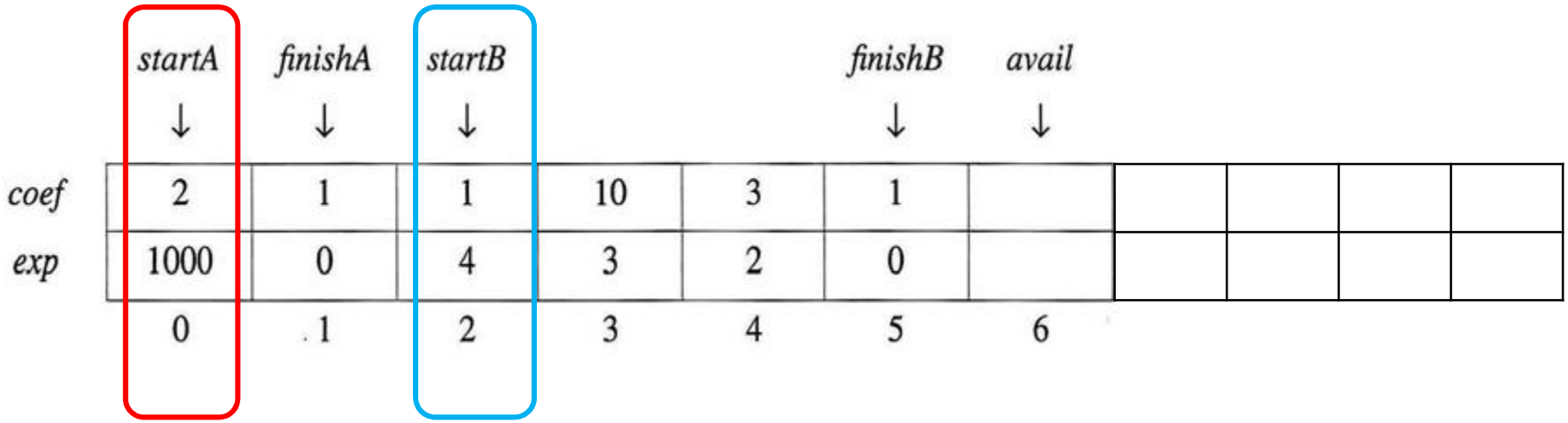
Figure 2.3: Array representation of two polynomials

Polynomial Addition

Ex. $A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$

➔ **D(x) = A(x) + B (x)**

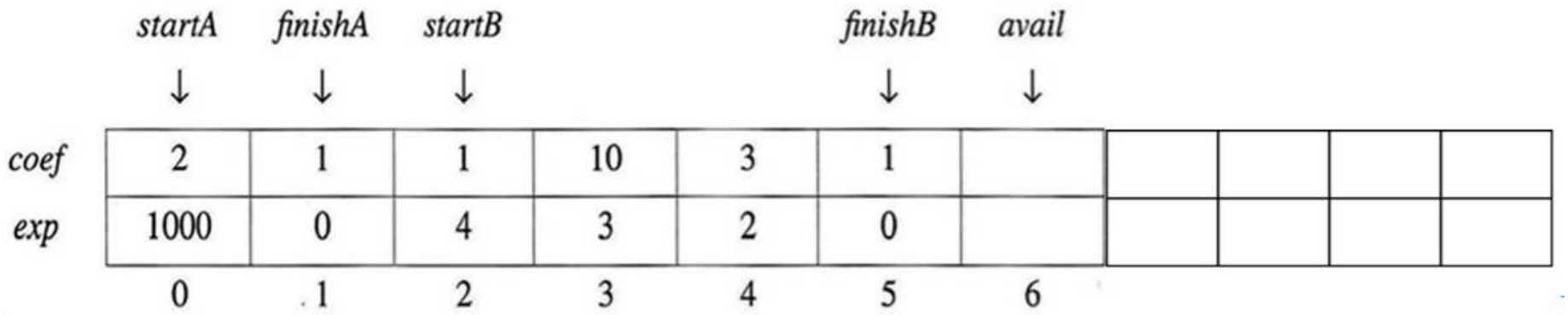
$= 2x^{1000}+x^4+10x^3+3x^2+2$



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

Polynomial Addition (전체)

```
/* add A(x) and B(x) to obtain D(x) */
void padd(int startA, int finishA, int startB,
          int finishB, int *startD, int *finishD){
  float coefficint;
  *startD = avail;
  while (startA <= finishA && startB<= finishB)
    switch(COMPARE(terms[startA].expon,
                   terms[startB].expon)){
      case -1:  /* a expon < b expon */
          attach(terms[startB].coef,
                 terms[startB].expon);
          startB++;
          break;
      case 0:  /* equal exponents */
          coefficient = terms[startA].coef +
                        terms[startB].coef;
          if(coefficient)
            attach(coefficient, terms[startA].expon);
          startA++;
          startB++;
          break;
      case 1:   /* a expon > b expon  */
          attach(terms[startA].coef,
                 terms[startA].expon);
          startA++;
  }
  /* add in remaining terms of A(x) */
  for( ; startA <= finishA; startA++)
    attach(terms[startA].coef, terms[startA].expon);
  /* add in remaining terms of B(x) */
  for( ; startB <= finishB; startB++)
    attach(terms[startB].coef, terms[startB].expon);
  *finishD = avail -1;
}
```

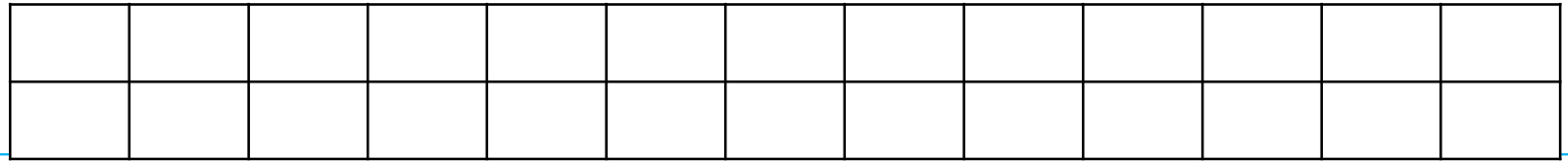



```
/* add A(x) and B(x) to obtain D(x) */
void padd(int startA, int finishA, int startB, int finishB, int *startD, int *finishD){
```

```
int main(){

    int sa=0, fa=1, sb=2, fb=5;
    int sd, fd;
    padd(sa, fa, sb, fb, &sd, &fd);
    …
```

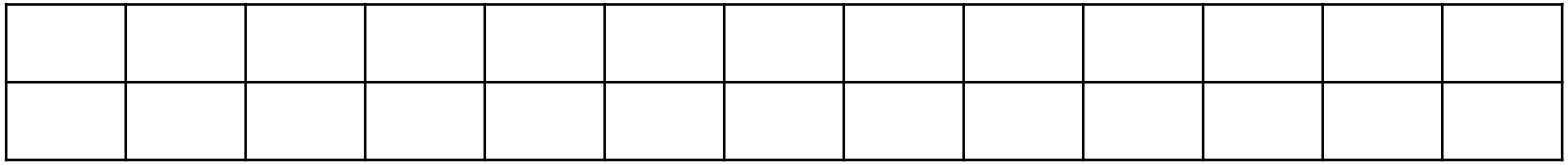
```
#define MAX_TERMS 100    /* size c
typedef struct {
        float coef;
        int expon;
    } polynomial;

polynomial terms[MAX_TERMS];
int avail = 0;
int starta, finisha
```

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



```
void padd(int startA, int finishA, int startB, int finishB, int *startD, int *finishD){
    float coefficint;
    *startD = avail;
    while (startA <= finishA && startB<= finishB)
        switch(COMPARE(terms[startA].expon,  terms[startB].expon)){
           case -1:  /* a expon < b expon */
                    attach(terms[startB].coef,    terms[startB].expon);
                    startB++;  break;
           case 0:  /* equal exponents */
                    coefficient = terms[startA].coef +  terms[startB].coef;
                    if(coefficient)
                          attach(coefficient, terms[startA].expon);
                           startA++;     startB++;        break;
        case 1:   /* a expon > b expon  */
                          attach(terms[startA].coef,  terms[startA].expon);
                          startA++;
    }
```

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



```
/* add in remaining terms of A(x) */
    for( ; startA <= finishA; startA++)
        attach(terms[startA].coef, terms[startA].expon);
   /* add in remaining terms of B(x) */
    for( ; startB <= finishB; startB++)
        attach(terms[startB].coef, terms[startB].expon);
    *finishD = avail -1;
}
```

```
void attach(float coefficient, int exponent)
{ /* add a new term to the polynomial */
      if (avail >= MAX_TERMS) {
          fprintf(stderr, "Too manay terms in the polynomial₩n");
          exit(EXIT_FAILURE);
      }
      terms[avail].coef = coefficient;
      tems[avail++].expon = exponent;
}
```

Sparse Matrices

- Standard representation of a matrix:

  A[MAX_ROWS][MAX_COLS]

	col0	col1	Col2
Row0	-27	3	4
Row1	6	82	-2
Row2	109	-64	11
Row3	12	8	9
row4	48	27	47

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

- Sparse matrix : m× n matrix A

  such that $\frac{\text{no. of non} - \text{zero elements}}{m \times n} << 1$

Sparse Matrix Representation

- Use an array of triples to represent a sparse matrix.
  - <row, column, value> : 3-tuples (triples)
    - no. of rows & columns
    - no. of non-zero elements
    - ordering (column major or row major)

SparseMatrix Create(*maxRow*, *maxCol*) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

Sparse Matrix Representation: example

A[6][6]

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

```
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

Row-major order

	row	col	val
a[0]	**6**	**6**	**8**
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Transposing a Matrix

- 행렬 전치
  - 어떤 행렬의 행과 열을 바꾼 것

Transposing a Matrix

```
for j← 1 to n do
  for i←1 to m do
        B(j, i) ← A(i, j)
  end
End
```

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

좀 더 효율적인 방법은 없을까?

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

	0	1	2	3	4	5
startingPos						

Transposing a Matrix: program(1)

```
void transpose(term a[], term b[])
{/* b is set to the transpose of a */
   int n,i,j, currentb;
   n = a[0].value;        /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

time complexity: ***O(columns· elements)***

Program 2.8: Transpose of a sparse matrix

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]			
[1]			
[2]			
[3]			
[4]			
[5]			
[6]			
[7]			
[8]			

```
void transpose(term a[], term b[])
{/* b is set to the transpose of a */
   int n,i,j, currentb;
   n = a[0].value;        /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
```

i=0

i=1

i=2

i=3

i=4

i=5

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]			
[2]			
[3]			
[4]			
[5]			
[6]			
[7]			
[8]			

```
if (n > 0 )  { /* non zero
   currentb = 1;
   for (i = 0; i < a[0].col; i++)
   /* transpose by the columns in a */
      for (j = 1; j <= n; j++)
      /* find elements from the current column */
         if (a[j].col == i) {
         /* element is in current column, add it to b */
            b[currentb].row = a[j].col;
            b[currentb].col = a[j].row;
            b[currentb].value = a[j].value;
            currentb++;
         }
}
```

Transposing a Matrix: program(2)

```
void fastTranspose(term a[], term b[])
{/* the transpose of a is placed in b */
  int rowTerms[MAX_COL], startingPos[MAX_COL];
  int i,j, numCols = a[0].col, numTerms = a[0].value;
  b[0].row = numCols;  b[0].col = a[0].row;
  b[0].value = numTerms;
  if (numTerms > 0) { /* nonzero matrix */
    for (i = 0; i < numCols; i++)
      rowTerms[i] = 0;
    for (i = 1; i <= numTerms; i++)
      rowTerms[a[i].col]++;
    startingPos[0] = 1;
    for (i = 1; i < numCols; i++)
      startingPos[i] =
                  startingPos[i-1] + rowTerms[i-1];
    for (i = 1; i <= numTerms; i++) {
      j = startingPos[a[i].col]++;
      b[j].row = a[i].col;   b[j].col = a[i].row;
      b[j].value = a[i].value;
    }
  }
}
```

time complexity: ***O(columns+ elements)***

Program 2.9: Fast transpose of a sparse matrix

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]			
[2]			
[3]			
[4]			
[5]			
[6]			
[7]			
[8]			

```
int rowTerms[MAX_COL], startingPos[MAX_COL];
int i,j, numCols = a[0].col, numTerms = a[0].value;
b[0].row = numCols;  b[0].col = a[0].row;
b[0].value = numTerms;
if (numTerms > 0) { /* nonzero matrix */
```

rowTerms

0	1	2	3	4	5

startingPos

0	1	2	3	4	5

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

numTerms=8
numCols=6

```
for (i = 0; i < numCols; i++)
   rowTerms[i] = 0;
for (i = 1; i <= numTerms; i++)
   rowTerms[a[i].col]++;
```

rowTerms

0	1	2	3	4	5

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

rowTerms

0	1	2	3	4	5
2	1	2	2	0	1

numCols=6

```
startingPos[0] = 1;
for (i = 1; i < numCols; i++)
   startingPos[i] =
                startingPos[i-1] + rowTerms[i-1];
```

startingPos

0	1	2	3	4	5

i=1
i=2
i=3
i=4
i=5
i=6
i=7
i=8

numTerms=8

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6	6	8
[1]			
[2]			
[3]			
[4]			
[5]			
[6]			
[7]			
[8]			

```
for (i = 1; i <= numTerms; i++) {
   j = startingPos[a[i].col]++;
   b[j].row = a[i].col;   b[j].col = a[i].row;
   b[j].value = a[i].value;
}
```

rowTerms

0	1	2	3	4	5
2	1	2	2	0	1

startingPos

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

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

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

```
for (i = 1; i <= numTerms; i++) {
   j = startingPos[a[i].col]++;
   b[j].row = a[i].col;   b[j].col = a[i].row;
   b[j].value = a[i].value;
}
```

```
for (i = 1; i <= numTerms; i++)
   rowTerms[a[i].col]++;
startingPos[0] = 1;
```

numTerms=8

numCols=6

rowTerms

0	1	2	3	4	5
2	1	2	2	0	1

```
startingPos[0] = 1;
for (i = 1; i < numCols; i++)
   startingPos[i] =
            startingPos[i-1] + rowTerms[i-1];
```

startingPos

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

	row	col	val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	val
b[0]	6		
[1]	0		
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

```
for (i = 1; i <= numTerms; i++) {
   j = startingPos[a[i].col]++;
   b[j].row = a[i].col;   b[j].col = a[i].row;
   b[j].value = a[i].value;
}
```

	row	col	val
b[0]			
b[1]			
b[2]			
b[3]			
b[4]			
b[5]			
b[6]			
b[7]			
b[8]			

```
for (i = 1; i <= numTerms; i++)
   rowTerms[a[i].col]++;
startingPos[0] = 1;
```

numTerms=8
numCols=6

rowTerms

0	1	2	3	4	5

```
startingPos[0] = 1;
for (i = 1; i < numCols; i++)
   startingPos[i] =
            startingPos[i-1] + rowTerms[i-1];
```

startingPos

0	1	2	3	4	5