Arrays and Structures

서승현 교수
Prof. Anes Seung-Hyun Seo
(seosh77@hanyang.ac.kr)

Division of Electrical Engineering
Hanyang University, ERICA Campus

2.1 ARRAYS

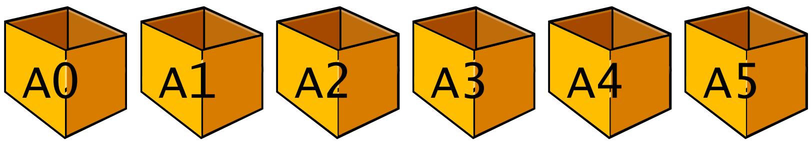
Abstract Data Type

- **Abstract data type: data type organized by**
  - **Specifications** of objects
    - Requirements/properties of objects
  - **Specifications** of operations on the objects
    - Description of what the function does.
    - Names, arguments, result of each functions
  - → "What a data type can do."

- **Abstract data type does not include**
  - Representation of objects
  - Implementation of operations
  - → "How it is don is hidden."

Arrays

- Used when creating multiple data of the same data type (variables of the same type)
  - int A0, A1, A2, A3, …,A5;



  - **int A[6];**

- **Using arrays in iterative code allows efficient programming**
  - Ex) Program to get maximum value: What if there was no array?

```
tmp=score[0];
for(i=1;i<n;i++){
            if( score[i] > tmp )
                        tmp = score[i];
}
```

2.1.1 The Abstract Data Type -Arrays

- **Many programmers view an array as "a consecutive set of memory locations"**
  - Unfortunately, although an array is usually implemented as a consecutive set of memory locations, it is not always the case.
  - It's possible to confuse the data structure of the array and its implementation of the memory.

- **Intuitively(직관적으로), an array is a set of pairs, <index, value>**
  - set of mappings (or correspondence)

    between index and values

    $array : I \Rightarrow a_i$
  - **Array**: mapping from **index** to **element**

A[0]	A[1]	A[2]	…	A[n-1]

< Array A of size n >

2.1.1 The Abstract Data Type -ADT 2.1: ADT Array

ADT Array

object : A set of pairs <index, value> where for each value of index there is a value from the set item.
Index: a finite ordered set of one or more dimensions, for example, {0, …., n-1} for one dimension(1차원배열), {(0,0), (0,1), (1,1), (1,2), (2.0), (2,1), (2,2)} for two dimensions(2차원배열), etc.

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

Array create(j, list) ::= **return** an array of j dimensions where list is a j-tuple whose ith element is the size of the ith dimension. Items are undefined.

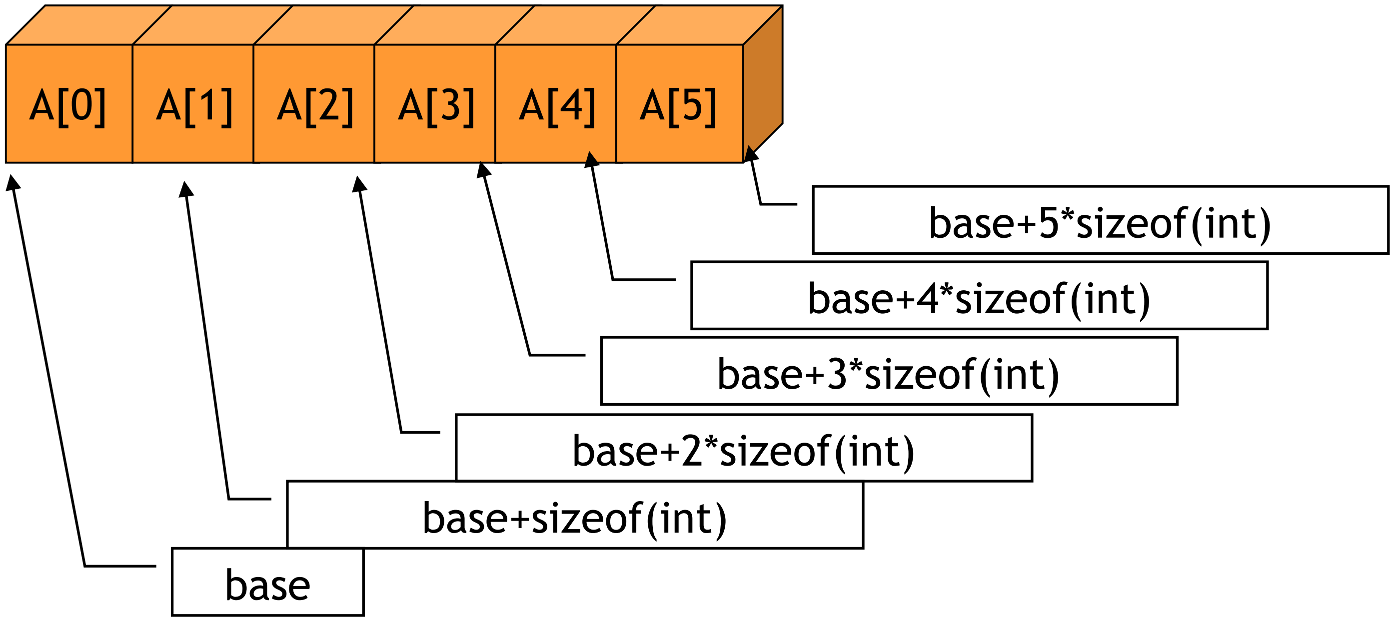
Item Retrieve(A, i) ::= **if** (i∈index) **return** the item associated with index value I in array A.
else return error.

Array Store(A, i, x) ::= **if** (i ∈ index)
return an array that is identical to array A except the new pair <i,x> has been inserted
else return error

end Array

2.1.2 Arrays in C

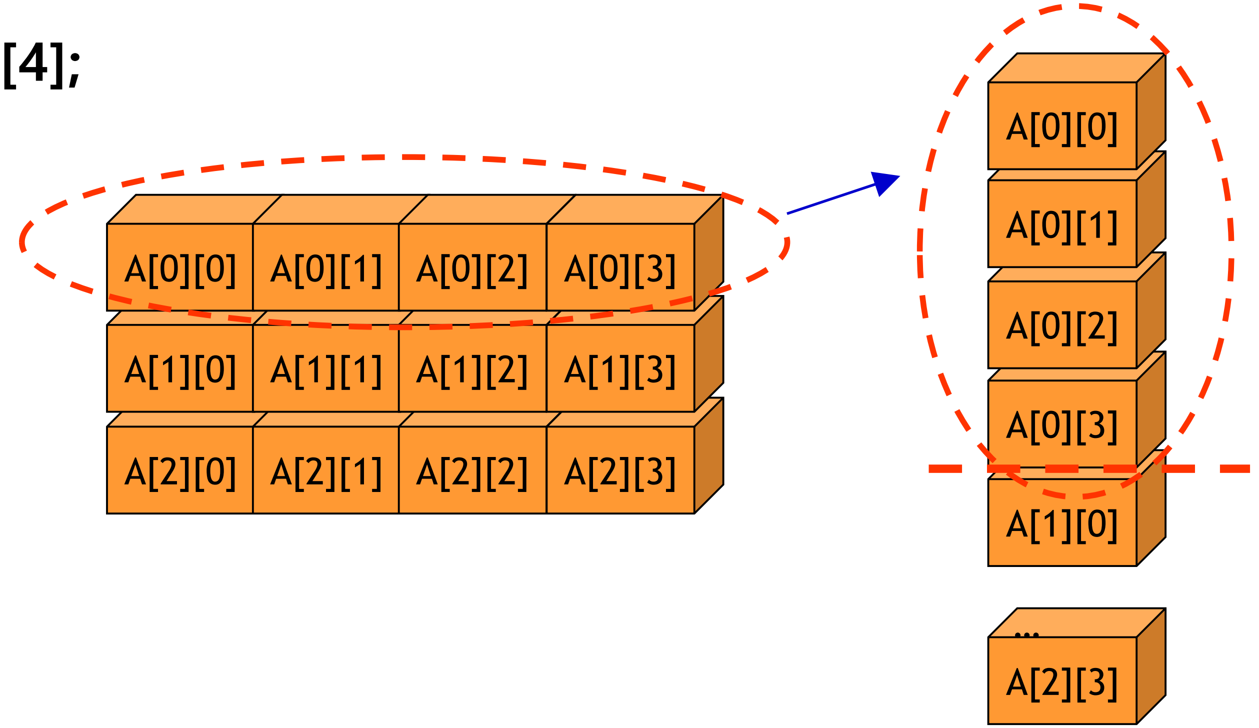
- int A[6];



- Array index starts at 0.
  - base: base address(기본 주소) in memory.
  - Since arrays are implemented at consecutive locations in memory, the address of A [0] is base.

2.1.2 Arrays in C

- **int A[3][4];**



The Location of Array in physical memory

2.1.2 Arrays in C

- One-dimensional arrays

int list[5], *plist[5];

- The first array defines five integers: list[0], list[1], list[2], list[3], list[4]
- The second array defines five integer pointers : plist[0], plist[1], plist[2], plist[3], plist[4]

Variable	Memory address
list[0]	Base address = a
list[1]	a + sizeof(int)
list[2]	a + 2 ·sizeof(int)
list[3]	a + 3 ·sizeof(int)
list[4]	a + 4 ·sizeof(int)

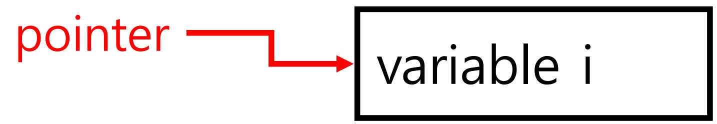
- Example

int *list1; int list2[5];

- The variable list1 and list2 are both integer pointers, but in the second case, five memory locations for holding integers have been reserved.
- list2 is a pointer to list2[0] (list2=list2[0]을 가리키는 포인터)
- list2 + i is a pointer to list2[i] (list2+i= list2[i]를 가리키는 포인터)
- (list2+i) = &list2[i], *(list2+i) = list2[i]

C Review - Pointers

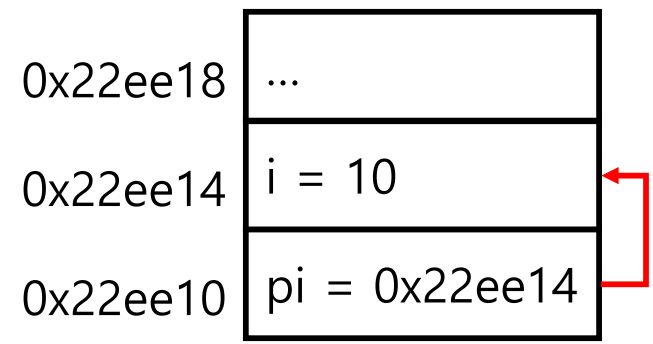
- **Pointer: A data type whose value is used to refer to ("points to") another value stored elsewhere in the computer memory**



- **Pointer-related operators**
  - Address operator &
  - Dereferencing operator *

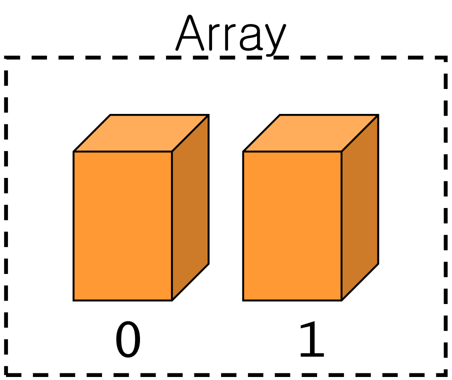
```
int i = 0;              // variable
int *pi = NULL;         // pointer

pi = &i;        // &i: address of i
i = 10;         // store 10 to variable i
*pi = 10;       // store 10 to memory location pointed by pi
```

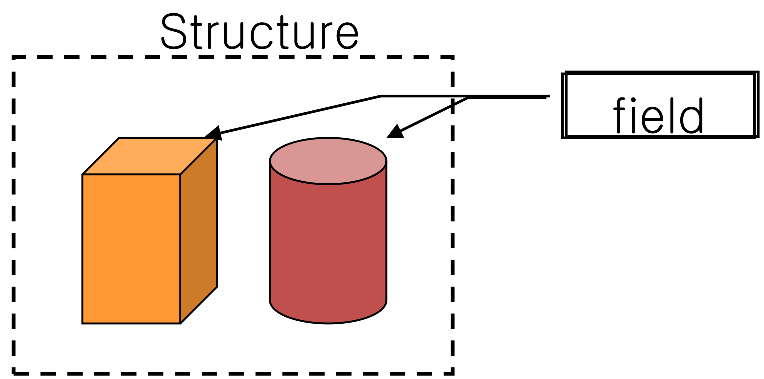


2.3 Structures

- Structure(구조체): The method to group different types of data in a way that permits the data to vary in type.
- Array(배열): The method to group data of the same type (or collections of data of the same type)




```
char carray[100];
```

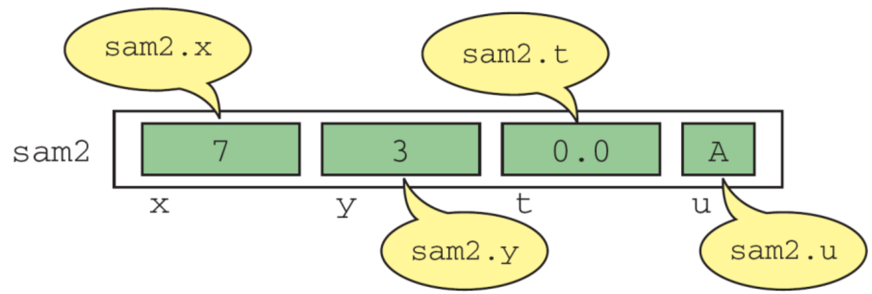



```
struct example {
        char cfield;
        int ifield;
        float ffield;
        double dfield;
};
struct example s1;
```

2.3 Structures

- **Structure: collection of related elements, possibly of different types**
  - Structure declaration

```
typedef struct
   {
     int   x;
     int   y;
     float t;
     char  u;
   } SAMPLE;
```



  - Variable declaration

SAMPLE sam2; // structure variable
SAMPLE *pSam = &sam2; // pointer to structure SAMPLE

2.3 Structures

A new data type that did not exist in the existing C data type.

■ We can create **a structure data type** by using the **typedef** statement.

- ▶ The structure concept evolved to become a C ++ class.

[ex]

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

or

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

2.3 Structures

■ structure : struct

- A collection of data items – each item is identified by its type and name.
- Ex) creates a structure variable whose name is ***person*** and that has three fields

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

- char name[10]; — A character array
- int age; — An integer value representing the age of the person
- float salary; — A float value representing the salary of the individual

■ The structure member operator :

- We may assign values to fields using *structure member operator* " . "
- The item can be accessed from outside by using " . "

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

2.3 Structures

- **human_being :** the name of the type defined by the structure definition

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

- Equality check of the entire structure : if (person1 == person2)
- Structure Assignment : **person1 = person2**
  - The value of every field of the structure of person 2 is assigned as the value of the corresponding field person 1.

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

2.3 Structures

■ Assignment of structure variable (O)

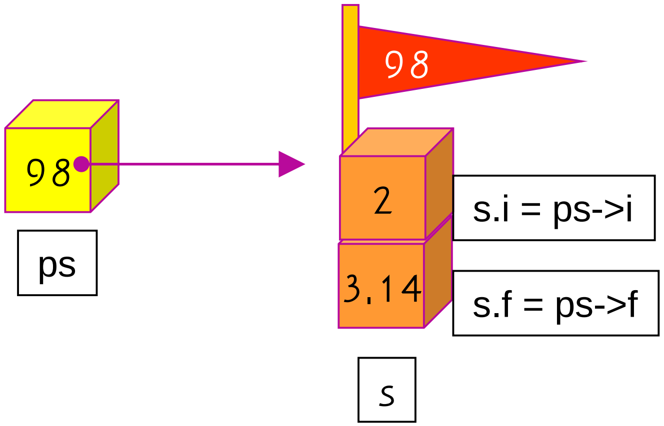
```
struct person {
		char name[10];
		int age;
		float height;
		};
main()
{
		person a, b;
		b = a;
}
```

■ Comparison between structure variables (X)

```
main()
{
		if( a > b )
				printf("a가 b보다 나이가 많음");
}
```

2.3 Structures - Pointer to structure

- An operator that accesses the elements (members) of a structure : ->



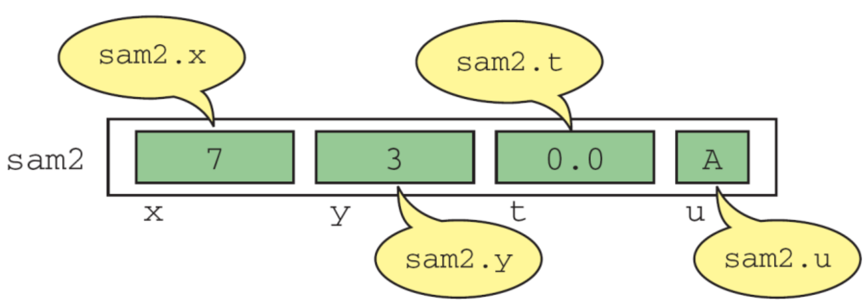

```
main()
{
        struct {
                int i;
                float f;
        } s, *ps;
        ps = &s;
        ps->i = 2;
        ps->f = 3.14;
}
```

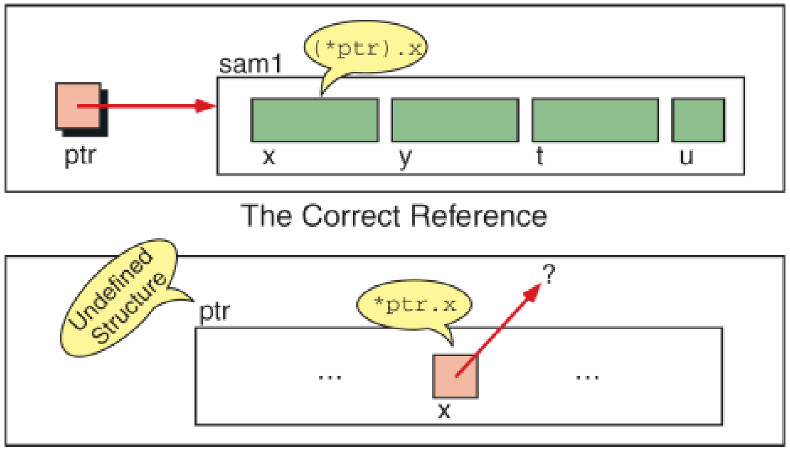
ps->i 는 (*ps).i와 동일한 효과를 가짐

2.3 Structures

- **Referencing individual fields**
  - **Direct selection operator (.)**
    sam2.x, sam2.y, …
  - **Indirect selection operator (->)**
    - (*pointerName).fieldName ≡ pointerName->fieldName

    pSam->x, pSam->y, pSam->t, pSam->u





2.3.2 Unions

■ Union

- A union declaration is similar to a structure, but the fields of a union must share their memory space.
- Only one field of the union is "active" at any given time.

```
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;
```

2.3.2 Unions

– Assigning values to person1 and person2 as:

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

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

- We first place a value in the tag field to determine which field in the union is active.
- Then, place a value in the proper field of the union.
- Once the value of sexInfo.sex was male, we would enter a TRUE or a FALSE in the sexInfo.u.beard field.

2.3.4 Self-Referential Structures

*** One in which one or more of its components is a pointer to itself.**
(구성요소 중 자신을 가리키는 포인터가 존재하는 구조)

- Self-referential structures usually require dynamic storage management routines (malloc and free).

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

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

item1.link = &item2;    "connecting structures to each other"
item2.link = &item3;    (item1 → item2 → item3)
```

2.4 Polynomials

Two example polynomials are:

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

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

ax^e a : coefficient $a_n \neq 0$
e : exponent – unique
x : variable x

− degree(차수) : The largest exponent of a polynomial
* Coefficients that are zero are not displayed.

– Ex) Two polynomials, $A(x) = \sum a_i x^i$ and $B(x) = \sum b_i x^i$

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

$A(x) \cdot B(x) = \sum (a_i x^i \cdot (\sum b_j x^j))$

2.4 Polynomials

■ Abstract Data Type: Polynomial

Structure Polynomial
Objects : $P(x) = a_1x^{e1} + \ldots + a_nx^{e_n}$: a set of ordered pairs of <e_i, a_i> where a_i in Coefficient, e_i in Exponents. $e_i \geq 0$ are integers.
Function : for all poly, poly1, poly2 ∈ 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 0
Exponent Lead_Exp(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, exp> inserted

2.4 Polynomials

Polynomial Remove(poly, expon)	::= **if**(expon $\in$ 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·coef·x^{expon}
Polynomial Add(poly1, poly2)	::= **return** the polynomial poly1+poly2
Polynomial Mult(poly1, poly2)	::= **return** the polynomial poly1·poly2

end polynomial

2.4 Polynomials

- The general form of a polynomial

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

- To process polynomials in a program, we need a data structure for polynomials

  ➔ What data structure is convenient and efficient when performing polynomial addition, subtraction, multiplication, and division operations?

- Two ways to use Arrays as a data structure for polynomials
  1) Put all items(e.g. coefficients) of a polynomial in an array.
  2) Put non-zero terms(e.g. coefficient) in an array of the polynomial.

2.4 Polynomials

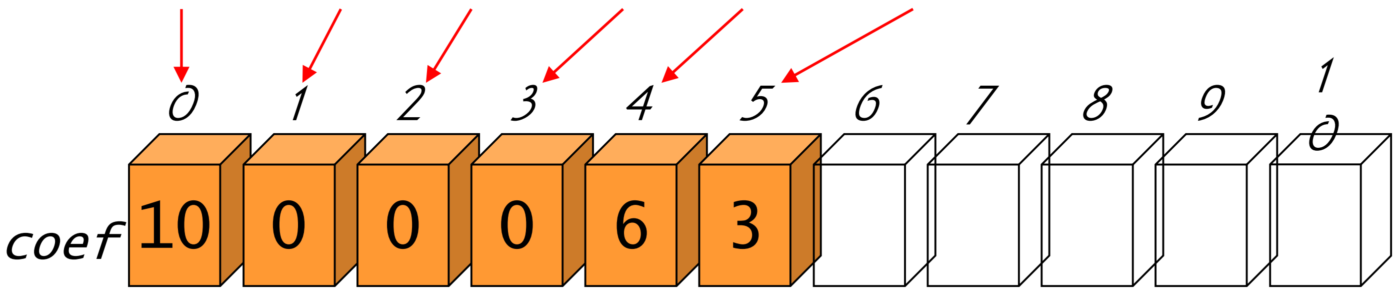
■ Polynomial Representation (I)

- ▶ Arrange in descending order of exponents(지수들의 내림차순으로 정돈)

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

Put coefficients for all items as an array.
Express one polynomial as one array.

$$10x^5 + 0x^4 + 0x^3 + 0x^2 + 6x^1 + 3x^0$$




```
typedef struct {
            int degree;
            float coef[MAX_DEGREE];
} polynomial;
polynomial a = { 5, {10, 0, 0, 0, 6, 3} };
```

2.4 Polynomials

■ Polynomial Representation (I)

- Advantage: Simplified polynomial operations
- Disadvantage: If most of the coefficients are zero, there is a lot of wasted space.

- Program example <Polynomial Addition>

```
#include <stdio.h>
#define MAX(a,b) (((a)>(b))?(a):(b))
#define MAX_DEGREE 101
typedef struct {                        // Declare polynomial struct type
        int degree;                     // Order(degree) of a polynomial
        float coef[MAX_DEGREE];         // Coefficients of a polynomial
} polynomial;
```

2.4 Polynomials

```
// C = A+B, where A and B are polynomials.
polynomial poly_add1(polynomial A, polynomial B)
{
        polynomial C;       // a Polynomial which contains the result of A+B
        int Apos=0, Bpos=0, Cpos=0;     // array index
        int degree_a=A.degree;
        int degree_b=B.degree;
        C.degree = MAX(A.degree, B.degree); // degree of a polynomial C
        while( Apos<=A.degree && Bpos<=B.degree ){
                if( degree_a > degree_b ){ // degree of terms in A > degree
                                           // of terms in B
                 C.coef[Cpos++]= A.coef[Apos++];
                 degree_a--;
                }
```

2.4 P

```
        else if( degree_a == degree_b ){ // degree of terms in A == degree of
                                         // terms in B
              C.coef[Cpos++]=A.coef[Apos++]+B.coef[Bpos++];
              degree_a--; degree_b--;
             }
        else {                           // degree of terms in B > degree of
                                         // terms in A
              C.coef[Cpos++]= B.coef[Bpos++];
              degree_b--;
             }
        }
        return C;
}

main()
{
        polynomial a = { 5, {3, 6, 0, 0, 0, 10} };
        polynomial b = { 4, {7, 0, 5, 0, 1} };
        polynomial c;
        c = poly_add1(a, b);
}
```

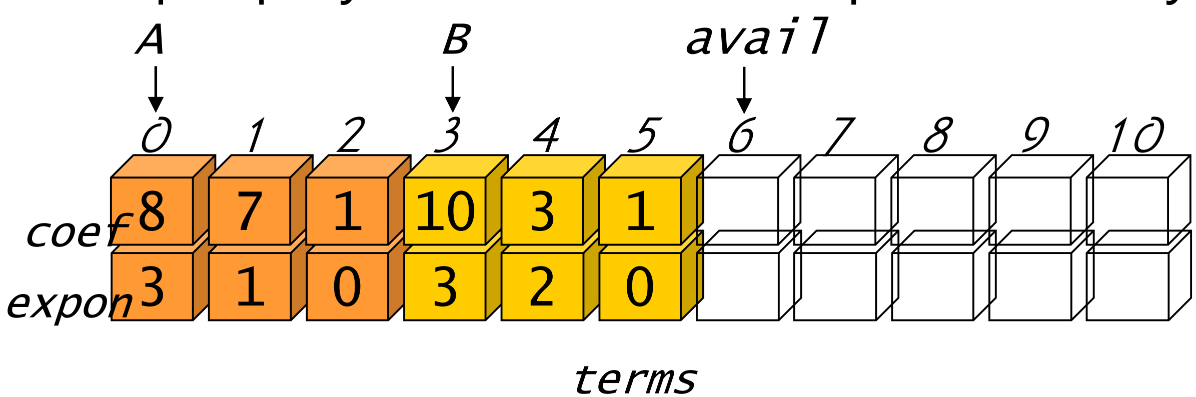
2.4 Polynomials

■ Polynomial Representation (II)

- ▶ Put non-zero terms of a polynomial in an array.
- ▶ Save to an array in (coefficient, degree) format
  - (ex) $10x^5+6x+3$ -> ((10,5), (6,1), (3,0))

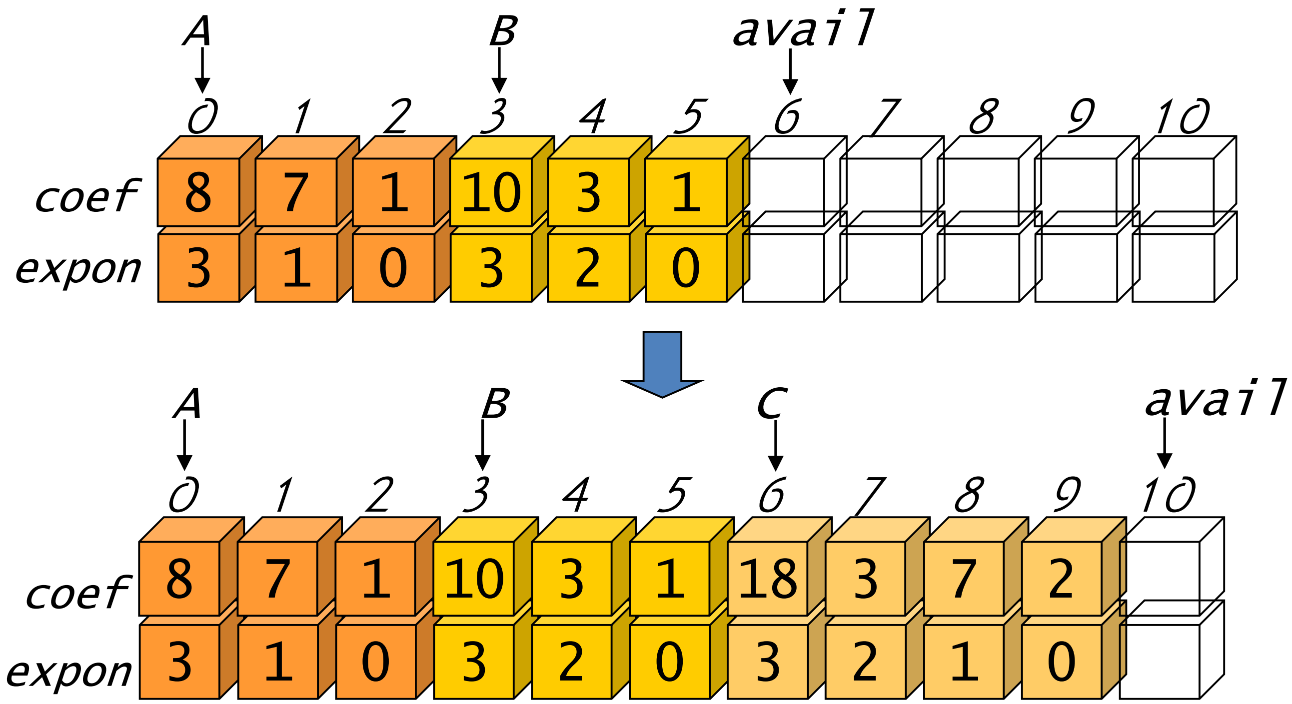
```
struct {
        float coef;
        int expon;
} terms[MAX_TERMS]={ {10,5}, {6,1}, {3,0} };
```

- ▶ Multiple polynomials can be represented by one array.



2.4 Polynomials

- Advantage: Efficient use of memory space
- Disadvantage: Implementation of polynomial operations is complicated.
  - (EX) Polynomial addition $A=8x^3+7x+1$, $B=10x^3+3x^2+1$, $C=A+B$



2.4 Polynomials

```
#define MAX_TERMS 101
struct {
            float coef;
            int expon;
} terms[MAX_TERMS]={ {8,3}, {7,1}, {1,0}, {10,3}, {3,2},{1,0} };
int avail=6;
// compare between a and b
char compare(int a, int b)
{
            if( a>b ) return '>';
            else if( a==b ) return '=';
            else return '<';
}
```

2.4 Polynomials

```
// 새로운 항을 다항식에 추가한다.
void attach(float coef, int expon)
{
        if( avail>MAX_TERMS ){
                fprintf(stderr, "항의 개수가 너무 많음₩n");
                exit(1);
        }
        terms[avail].coef=coef;
        terms[avail++].expon=expon;
}
```

2.4 Polynomials

```
// C = A + B
poly_add2(int As, int Ae, int Bs, int Be, int *Cs, int *Ce)
{
        float tempcoef;
        *Cs = avail;
        while( As <= Ae && Bs <= Be )
         switch(compare(terms[As].expon,terms[Bs].expon)){
         case '>':   // A의 차수 > B의 차수
                     attach(terms[As].coef, terms[As].expon);
                     As++;                              break;
         case '=':   // A의 차수 == B의 차수
                     tempcoef = terms[As].coef + terms[Bs].coef;
                     if( tempcoef )
                      attach(tempcoef,terms[As].expon);
                     As++; Bs++;                        break;
         case '<':   // A의 차수 < B의 차수
                     attach(terms[Bs].coef, terms[Bs].expon);
                     Bs++;                              break;
        }
```

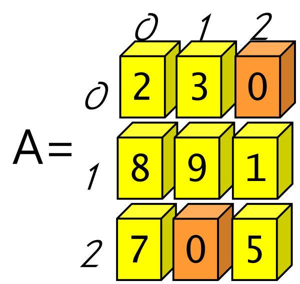
2.4 Polynomials

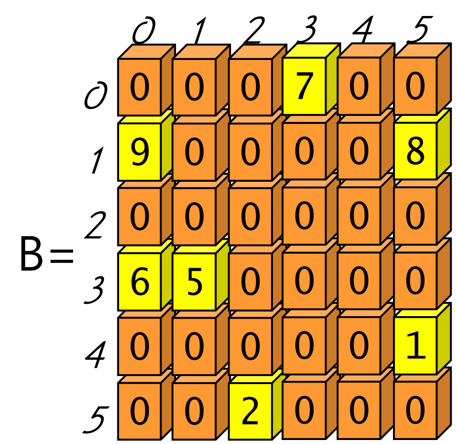
```
// A의 나머지 항들을 이동함
        for(;As<=Ae;As++)
                attach(terms[As].coef, terms[As].expon);
        // B의 나머지 항들을 이동함
        for(;Bs<=Be;Bs++)
                attach(terms[Bs].coef, terms[Bs].expon);
        *Ce = avail -1;
}
//
void main()
{
        int Cs, Ce;
        poly_add2(0,2,3,5,&Cs,&Ce);
}
```

2.5 Sparse Matrix

- **Method to store all elements of a sparse matrix by using a 2-dimensional array**
  - Advantage: a simple implementation of matrix operations
  - Disadvantage: Memory space is wasted in a sparse matrix where most terms are zero

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$





2.5 Sparse Matrix

```
#include <stdio.h>
#define ROWS 3
#define COLS 3
// 희소 행렬 덧셈 함수
void sparse_matrix_add1(int A[ROWS][COLS],
                        int B[ROWS][COLS], int C[ROWS][COLS]) // C=A+B
{
        int r,c;
        for(r=0;r<ROWS;r++)
                for(c=0;c<COLS;c++)
                        C[r][c] = A[r][c] + B[r][c];
}
```

2.5 Sparse Matrix

```
main()
{
        int array1[ROWS][COLS] = {      { 2,3,0 },
                                                { 8,9,1 },
                                                { 7,0,5 } };
        int array2[ROWS][COLS] = {      { 1,0,0 },
                                                { 1,0,0 },
                                                { 1,0,0 } };
        int array3[ROWS][COLS];
        sparse_matrix_add1(array1,array2,array3);
}
```

2.5 Sparse Matrix (Method II)

- **A method to store only nonzero elements**
  - Advantage: Efficient use of memory
  - Disadvantage: The implementation of various matrix operations becomes complicated.

$$A=\begin{bmatrix}2 & 3 & 0\\ 8 & 9 & 1\\ 7 & 0 & 5\end{bmatrix} \quad B=\begin{bmatrix}0 & 0 & 0 & 7 & 0 & 0\\ 9 & 0 & 0 & 0 & 0 & 8\\ 0 & 0 & 0 & 0 & 0 & 0\\ 6 & 5 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 2 & 0 & 0 & 0\end{bmatrix}$$

A=

	행	열	값
0	0	0	2
1	0	1	3
2	1	0	8
3	1	1	9
4	1	2	1
5	2	0	7
6	2	2	5

B=

	행	열	값
0	0	3	7
1	1	0	9
2	1	5	8
3	3	0	6
4	3	1	5
5	4	5	1
6	5	2	2

2.5 Sparse Matrix (Method II)

```
#define ROWS 3
#define COLS 3
#define MAX_TERMS 10
typedef struct {
        int row;
        int col;
        int value;
} element;
typedef struct SparseMatrix {
        element data[MAX_TERMS];
        int rows;   // 행의 개수
        int cols;   // 열의 개수
        int terms;  // 항의 개수
} SparseMatrix;
```

2.5 Sparse Matrix (Method II)

```
// 희소 행렬 덧셈 함수
// c = a + b
SparseMatrix sparse_matrix_add2(SparseMatrix a, SparseMatrix b)
{
        SparseMatrix c;
        int ca=0, cb=0, cc=0; // 각 배열의 항목을 가리키는 인덱스
        // 배열 a와 배열 b의 크기가 같은지를 확인
        if( a.rows != b.rows || a.cols != b.cols ){
                fprintf(stderr,"희소행렬 크기에러₩n");
                exit(1);
        }
        c.rows = a.rows;
        c.cols = a.cols;
        c.terms = 0;
```

```
while( ca < a.terms && cb < b.terms ){
        // 각 항목의 순차적인 번호를 계산한다.
                int inda = a.data[ca].row * a.cols + a.data[ca].col;
                int indb = b.data[cb].row * b.cols + b.data[cb].col;
        if( inda < indb) {
                // a 배열 항목이 앞에 있으면
                c.data[cc++] = a.data[ca++];
        }
        else if( inda == indb ){
                // a와 b가 같은 위치
                if( (a.data[ca].value+b.data[cb].value)!=0){
                   c.data[cc].row = a.data[ca].row;
                   c.data[cc].col = a.data[ca].col;
                   c.data[cc++].value = a.data[ca++].value +
                                        b.data[cb++].value;
                }
                else {
                   ca++; cb++;
                }
        }
        else   // b 배열 항목이 앞에 있음
                c.data[cc++] = b.data[cb++];
}
```

2.5 Sparse Matrix (Method II)

```
// 배열 a와 b에 남아 있는 항들을 배열 c로 옮긴다.
        for(; ca < a.terms; )
                c.data[cc++] = a.data[ca++];
        for(; cb < b.terms; )
                c.data[cc++] = b.data[cb++];
        c.terms = cc;
        return c;
}
// 주함수
main()
{
        SparseMatrix m1 = {  {{ 1,1,5 },{ 2,2,9 }}, 3,3,2 };
        SparseMatrix m2 = {  {{ 0,0,5 },{ 2,2,9 }}, 3,3,2 };
        SparseMatrix m3;
        m3 = sparse_matrix_add2(m1, m2);
}
```