Data Structures

Lecture 3: Array, Structure, and Pointer

Dongbo Min

Department of Computer Science and Engineering

Ewha Womans University, Korea

E-mail: dbmin@ewha.ac.kr

Array

- Array
  - Creating multiple variables of the same type

    int A0, A1, A2, A3, …,A9;
    → int A[10];

- Efficient programming in iterative code

  Ex) What if there is no array, when seeking maximum value?

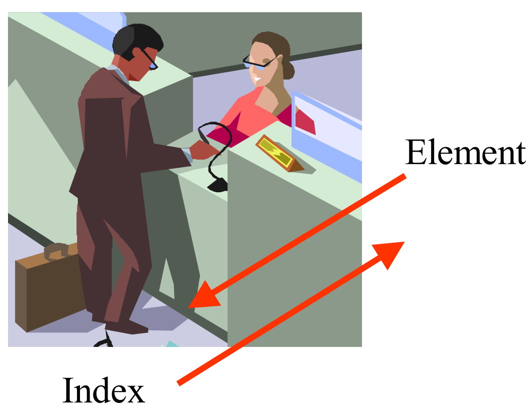
```
tmp=score[0];
for(i=1;i<n;i++){
        if( score[i] > tmp )
                tmp = score[i];
}
```

Array

- Array: a set of pairs of <index, element>
- For a given index, the corresponding element is matched.

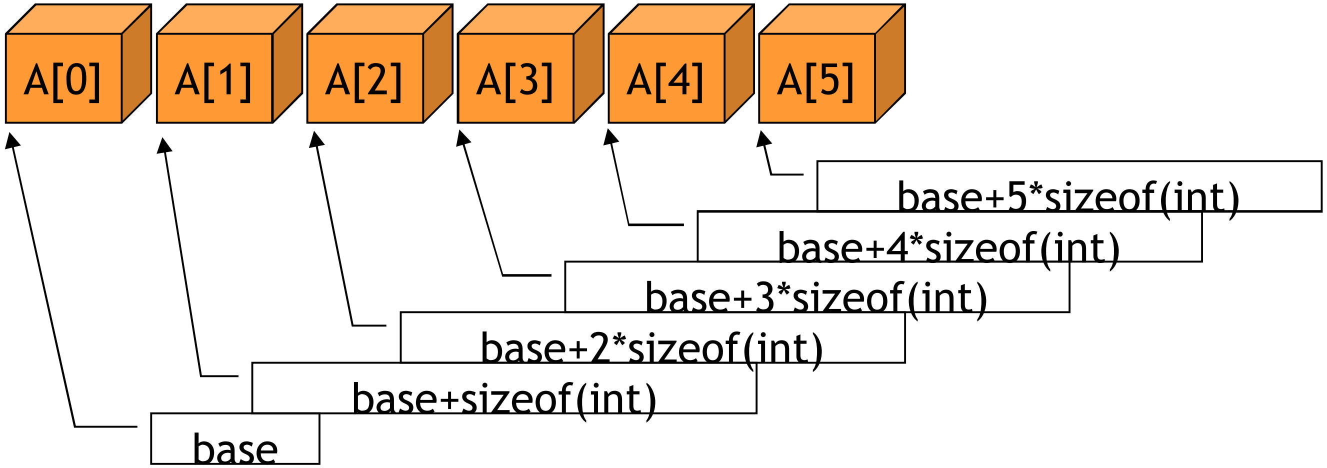
```
Array ADT

Object: a set of pairs of <index, element>
Operation:
 ▪ create(n) ::= Create the array with n elements
 ▪ retrieve(A, i) ::= Return ith element in the array 'A'
 ▪ store(A, i, item) ::= Save 'item' at the ith element of the array 'A'
```



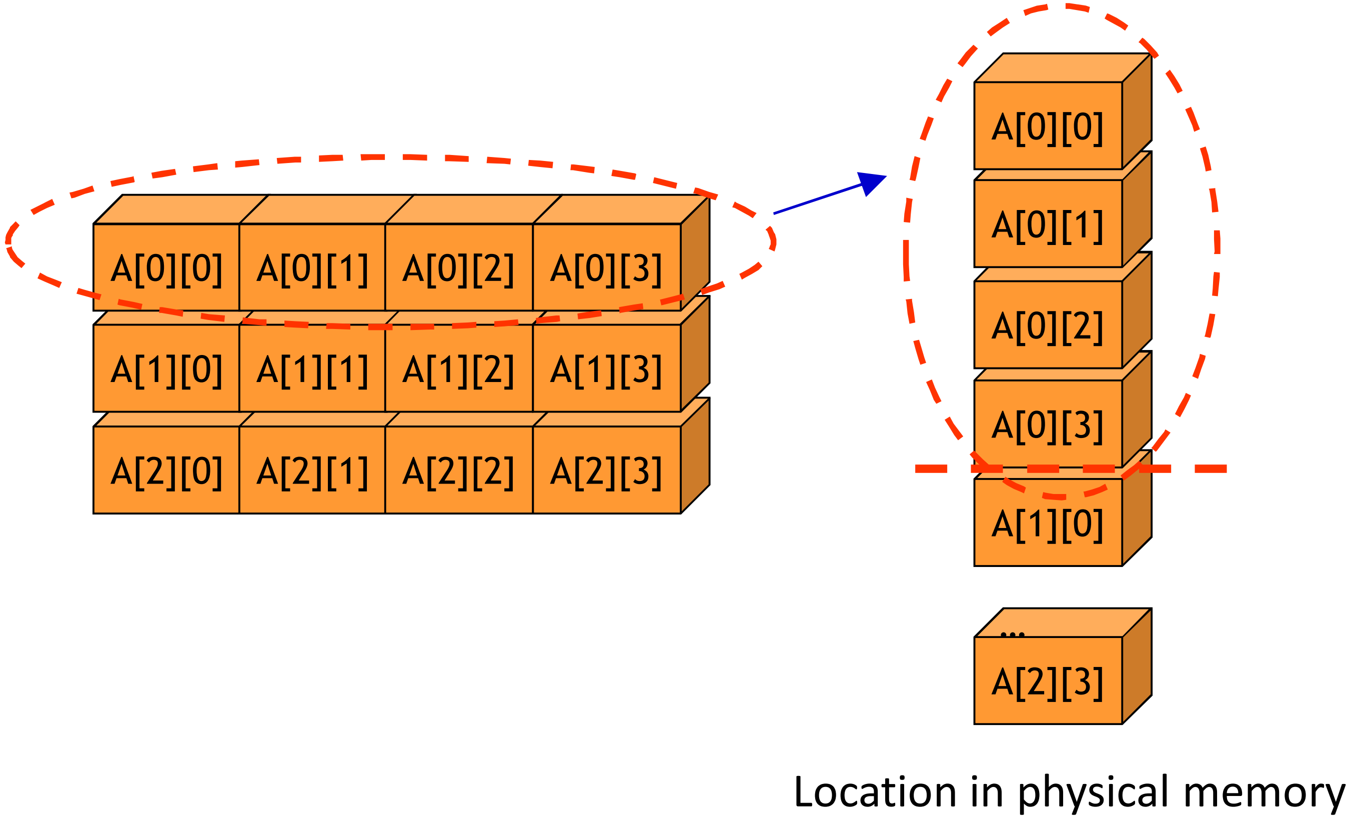
1D Array

- int A[6];



2D Array

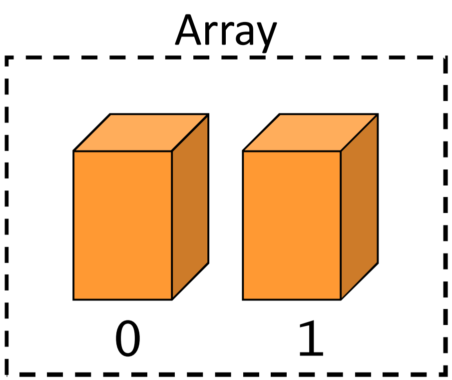
- int A[3][4];



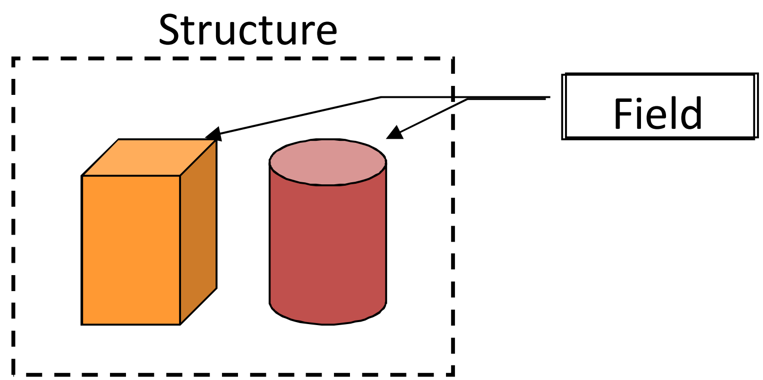
Structure

Array
How to group data of the *same* type

Structure
How to group data of *different* types




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




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

Example of Structure

- Declaration of structure & generation of structure variable

```
struct person {
        char name[10];     // Character array
        int age;           // Integer representing an age
        float height;      // Real value representing a height
};
struct person a;           // Generation of structure variable
```

- Declaration of structure & generation of structure variable using ‘typedef’

```
typedef struct person {
        char name[10];     // Character array
        int age;           // Integer representing an age
        float height;      // Real value representing a height
} person;
person a;                  // Generation of structure variable
```

Assignment and Comparison in Structure

- Assignment of structure variable: O

```
struct person {
          char name[10];     // Character array
          int age;           // Integer representing an age
          float height;      // Real value representing a height
};
main()
{
          person a, b;                                          O
          b = a;
}
```

- Comparison between structure variables: X

```
main()
{
          if( a > b )                                           X
                    printf("a is older than b");
}
```

Self-Referential Structure

- Self-referential structure
  - Has one or more pointers to itself in the field
  - Is often used in linked lists or trees

```
typedef struct ListNode {
        char    data[10];
        struct  ListNode *link;
} ListNode;
```

Structure Array

```
#define MAX_STUDENTS 200
#define MAX_NAME 100

typedef struct {
        int month;
        int date;
}  BirthdayType;

typedef struct {
        char name[MAX_NAME];
        BirthdayType birthday;
}  StudentType;

StudentType students[MAX_STUDENTS];

void main()
{
        strcpy(students[0].name, “HongGilDong”);
        students[0].birthday.month = 10;
        students[0].birthday.date = 28;
}
```

Applications of Arrays: Polynomials

- General form of a polynomial

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

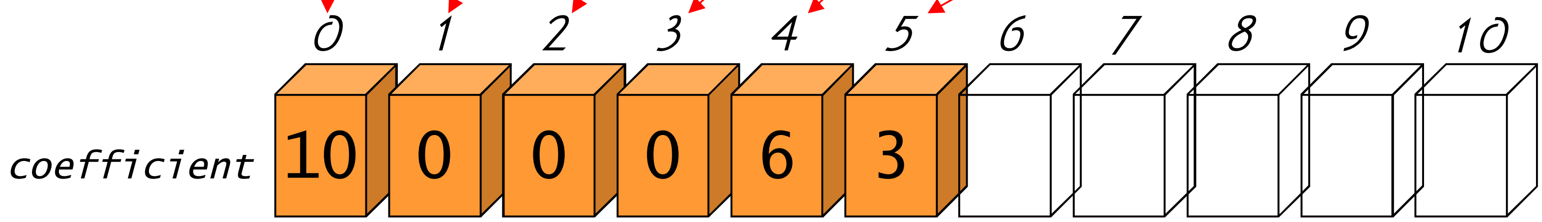
- Data structure for polynomials
  - Which data structures are efficient when handling the addition, subtraction, multiplication, and division of polynomials in a program?

- Two ways to use arrays
  1. Store all terms of a polynomial in an array
  2. Store only non-zero terms of a polynomial in an array

Polynomial Representation in Arrays (1)

- Store coefficients for all orders 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} };
```

Polynomial Representation in Arrays (1)

- Store all terms of a polynomial in an array
  - Pros: Simplified polynomial operations
  - Cons: It causes wasteful space, when most of the coefficients are zero.

Ex) Addition operation of polynomial

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

Polynomial Representation in Arrays (1)

```
// C = A+B, where A and B are polynomials.
polynomial poly_add1(polynomial A, polynomial B)
{
       polynomial C;                         // Result
       int Apos=0, Bpos=0, Cpos=0;           // Array index variables
       int degree_a=A.degree;
       int degree_b=B.degree;
       C.degree = MAX(A.degree, B.degree);   // Result polynomial order
       while( Apos<=A.degree && Bpos<=B.degree ){
              if( degree_a > degree_b ){  // A term > B term
               C.coef[Cpos++]= A.coef[Apos++];
               degree_a--;
              }
```

Polynomial Representation in Arrays (1)

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

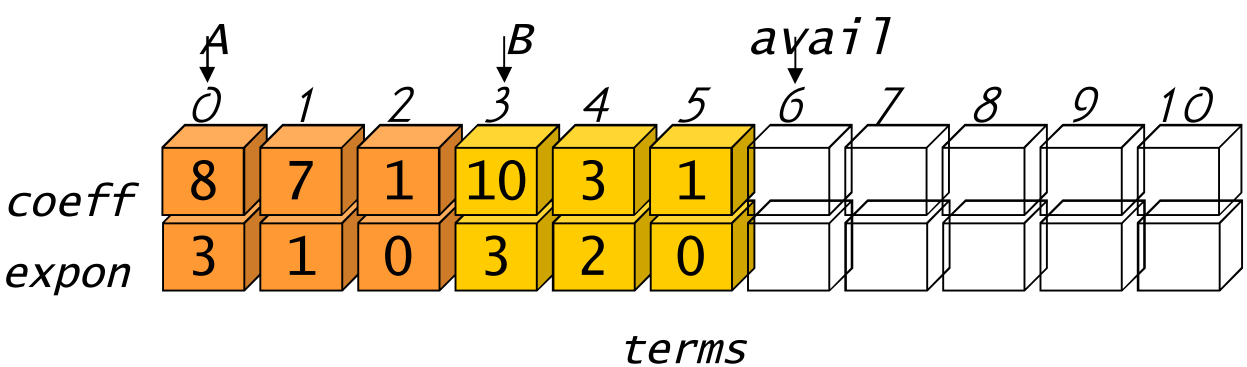
Polynomial Representation in Arrays (2)

- Store only non-zero terms of a polynomial in an array
  - Store (Coefficients, orders) in an array

    Ex) $10x^5+6x+3$ -> ((10,5), (6,1), (3,0))

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

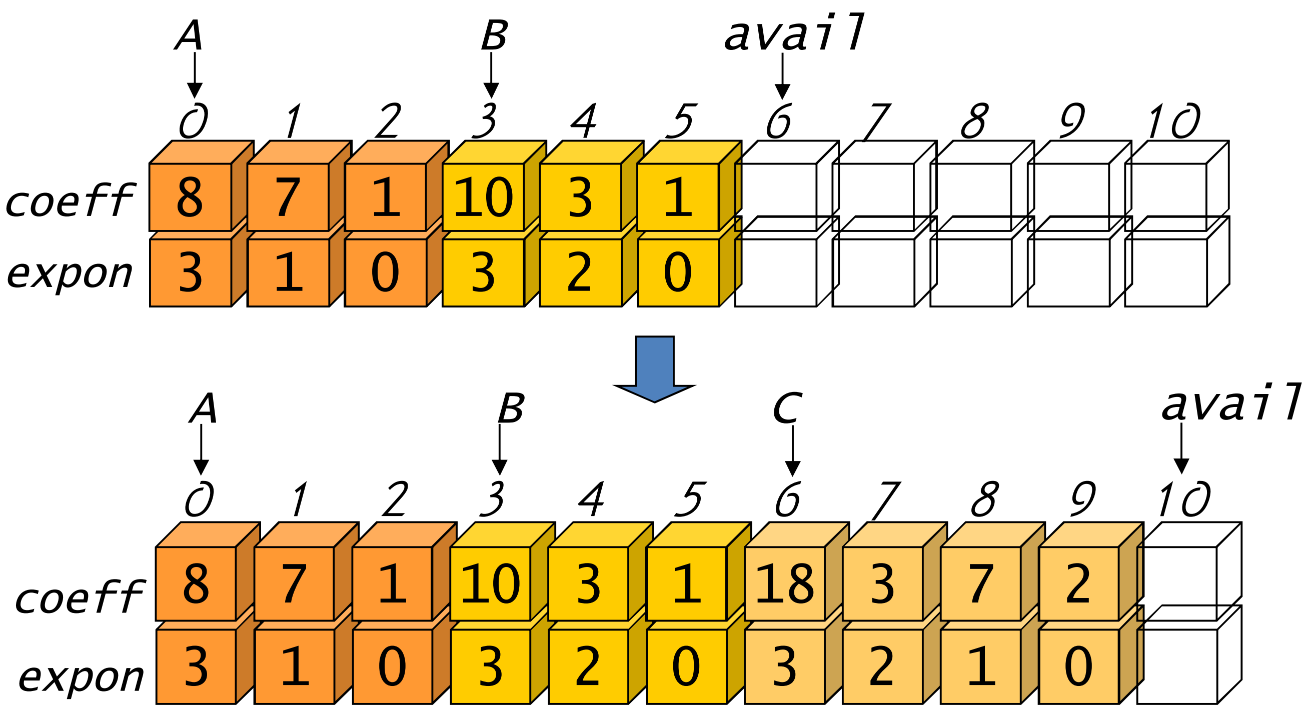
  - Multiple polynomials can be represented by an array.



Polynomial Representation in Arrays (2)

- Pros: Efficient use of memory space
- Cons: Polynomial operations are complex

Ex) $A=8x^3+7x+1$, $B=10x^3+3x^2+1$, $C=A+B$



Polynomial Representation in Arrays (2)

```c
#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 two integers
char compare(int a, int b)
{
        if( a>b ) return '>';
        else if( a==b ) return '=';
        else return '<';
}
```

Polynomial Representation in Arrays (2)

```c
// Add a new term to the polynomial.
void attach(float coef, int expon)
{
        if( avail>MAX_TERMS ){
                fprintf(stderr, " Too many terms \n");
                exit(1);
        }
        terms[avail].coef=coef;
        terms[avail++].expon=expon;
}
```

Polynomial Representation in Arrays (2)

```
// 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 '>':          // A term > B term
                 attach(terms[As].coef, terms[As].expon);
                 As++;                              break;
         case '=':          // A term == B term
                 tempcoef = terms[As].coef + terms[Bs].coef;
                 if( tempcoef )
                  attach(tempcoef,terms[As].expon);
                 As++; Bs++;                                  break;
         case '<':          // A term < B term
                 attach(terms[Bs].coef, terms[Bs].expon);
                 Bs++;                              break;
        }
```

Polynomial Representation in Arrays (2)

```
        // Copy and paste of remaining terms of A
        for(;As<=Ae;As++)
                attach(terms[As].coef, terms[As].expon);
        // Copy and paste of remaining terms of A
        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);
}
```

Sparse Matrix

- Two ways to represent a matrix using arrays
  1. How to store all elements in a 2D array
  2. How to store only non-zero elements

- Sparse matrix
  – Matrix where most terms are zero

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

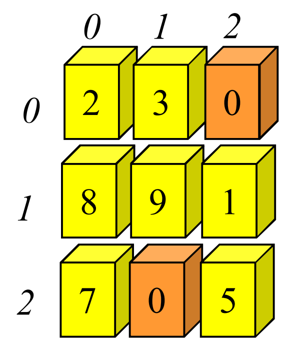
Dense matrix

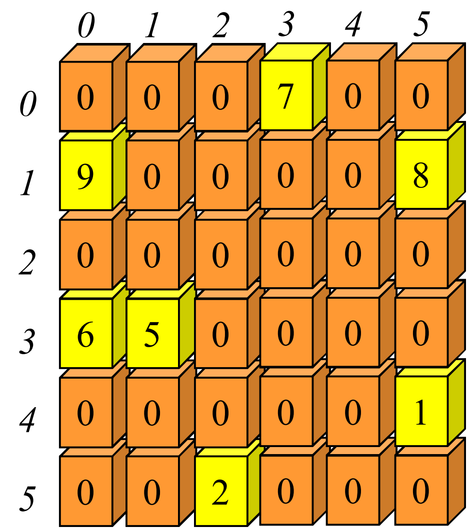
Sparse matrix

Sparse Matrix Representation (1)

- How to store all elements in a 2D array
  - Pros: Matrix operations can be implemented simply.
  - Cons: Memory is wasted when most terms are zero

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

A=


B=


Sparse Matrix Representation (1)

- Addition in sparse matrix

```
#include <stdio.h>
#define ROWS 3
#define COLS 3
// Addition
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];
}
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);
}
```

Sparse Matrix Representation (2)

- How to store only non-zero elements
  - Pros: Memory is saved for sparse matrices
  - Cons: Complex implementation of matrix operations

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

A=

	row	*column*	*value*
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=

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

Sparse Matrix Representation (2)

```
#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;          // row number
        int cols;          // column number
        int terms;         // element number
} SparseMatrix;
```

Sparse Matrix Representation (2)

```
// Addition
// c = a + b
SparseMatrix sparse_matrix_add2(SparseMatrix a, SparseMatrix b)
{
        SparseMatrix c;
        int ca=0, cb=0, cc=0; // Index indicating terms in each array
        // Check if array a and array b are the same size.
        if( a.rows != b.rows || a.cols != b.cols ){
                fprintf(stderr," Size error in Sparse matrix \n");
                exit(1);
        }
        c.rows = a.rows;
        c.cols = a.cols;
        c.terms = 0;
```

```
while( ca < a.terms && cb < b.terms ){
            // Compute the index of each item.
            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 the array ‘a’ entry is in front
            if( inda < indb) {
                    c.data[cc++] = a.data[ca++];
            }
            // If ‘a’ and ‘b’ are the same location
            else if( inda == indb ){
                    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   // If the array ‘b’ entry is in front
                    c.data[cc++] = b.data[cb++];
    }
```

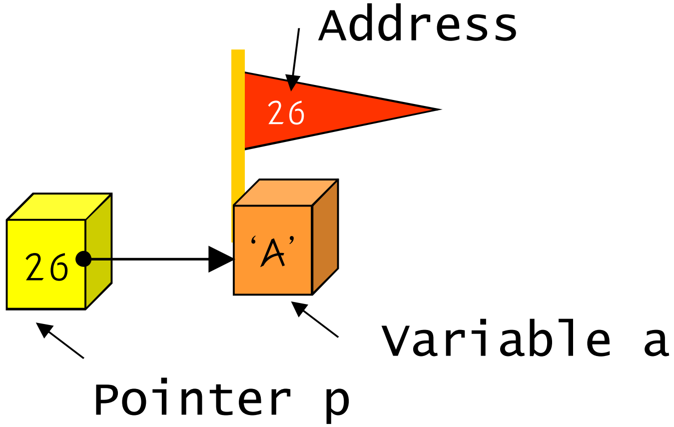
Sparse Matrix Representation (2)

```
// Copy and paste the remaining terms in ‘a’ and ‘b’ into ‘c’.
        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;
}
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);
}
```

Pointer

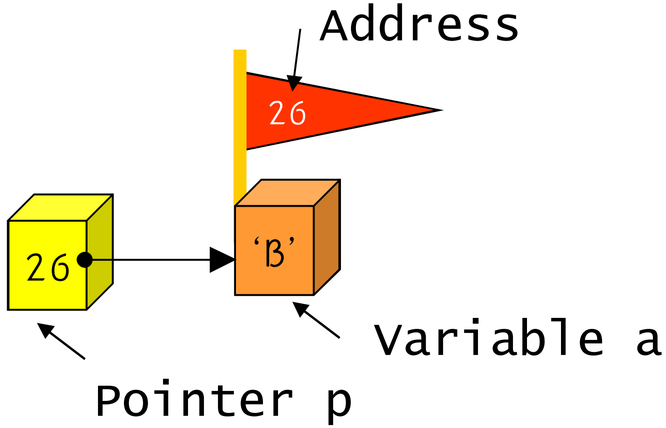
- Pointer
  - Variable with the address of another variable

```
char a='A';
char *p;
p = &a;
```



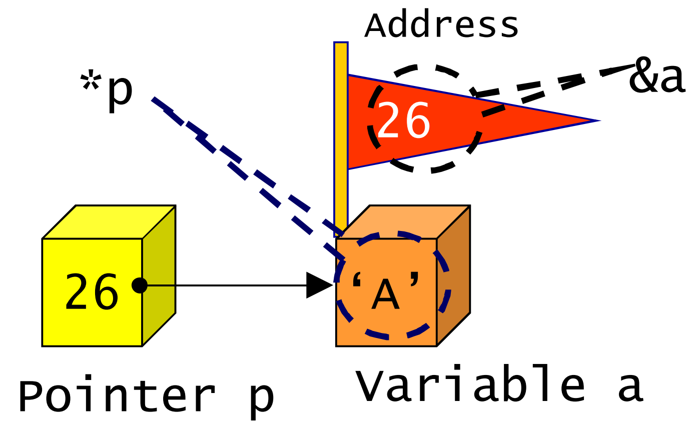
  - To change the pointer: use * operator

```
*p= 'B';
```



Operation in Pointer

- '&': return variable address
- '*' : return the contents of the pointer




```
int a;      // Declare an integer variable
int *p;     // Declare an integer pointer
int **pp;   // Declare a pointer of an integer pointer
p = &a;     // Connect variable a and pointer p
pp = &p;    // Connect pointer p and pointer's pointer pp
```

Pointer Type

```
void *p;              // pointer to point to null
int *pi;              // pointer to an integer variable
float *pf;            // pointer to a real variable
char *pc;             // pointer to a character variable
int **pp;             // pointer to a pointer
struct test *ps;      // pointer to a structure of test type
void (*f)(int);       // pointer to a function f with 'int' parameter
```

- Pointer casting: It is possible to cast whenever necessary
  - Void pointer can be changed to different type of pointer.

```
void *p;
pi=(int *) p;
```

Pointer as Function Parameter

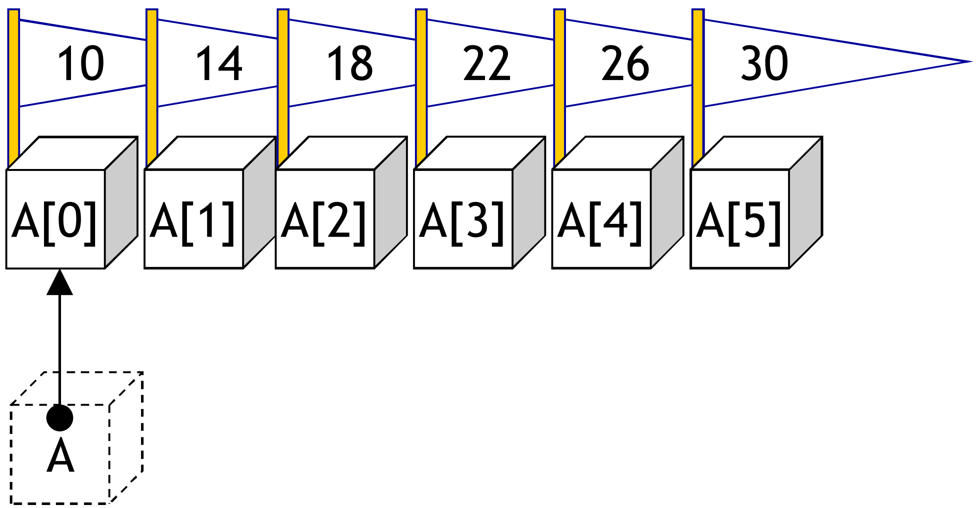
- It is possible to change the value of external variable by using pointer passed as parameter in function

```
void swap(int *px, int *py)
{
        int tmp;
        tmp = *px;
        *px = *py;
        *py = tmp;
}
main()
{
        int a=1,b=2;
        printf("Before swap: a=%d, b=%d\n", a,b);
        swap(&a, &b);
        printf("After swap: a=%d, b=%d\n", a,b);
}
```

```
Before swap: a=1, b=2
After swap: a=2, b=1
```

Array and Pointer

- Array name = pointer
  - The compiler replaces the array name with the first address in the array.



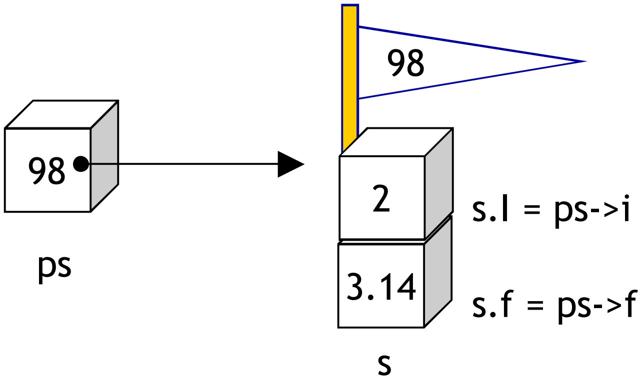
Array and Pointer

```c
#include <stdio.h>
#define ROWS 3
// C=A+B
void vec_add1(int A[ROWS], int B[ROWS], int C[ROWS])
{
          int r,c;
          for(r=0;r<ROWS;r++)
                    C[r] = A[r] + B[r];
}
main()
{
          int array1[ROWS] = { 2,3,0 };
          int array2[ROWS] = { 1,0,0 };
          int array3[ROWS];
          vec_add1(array1,array2,array3);
}
```

```c
#include <stdio.h>
#define ROWS 3
// C=A+B
void vec_add1(int *A, int *B, int *C)
{
          int r,c;
          for(r=0;r<ROWS;r++)
                    C[r] = A[r] + B[r];
}
main()
{
          int array1[ROWS] = { 2,3,0 };
          int array2[ROWS] = { 1,0,0 };
          int array3[ROWS];
          vec_add1(array1,array2,array3);
}
```

Structure Pointer

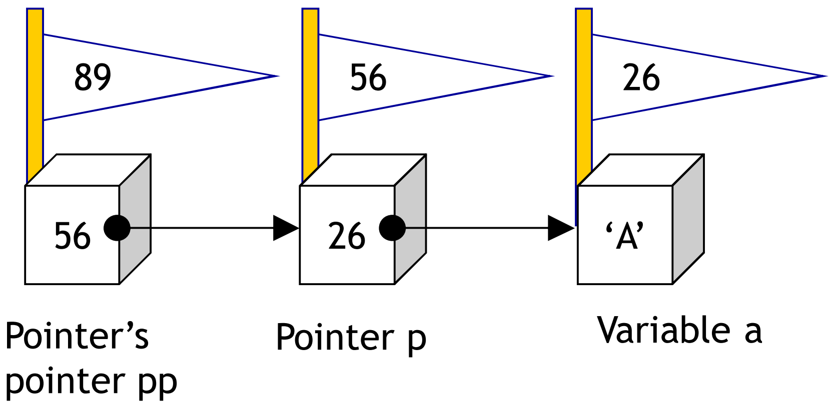
- '->': accesses elements of a structure




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

```
(*ps).i = 2;
(*ps).f = 3.14;
```

Pointer of Pointer

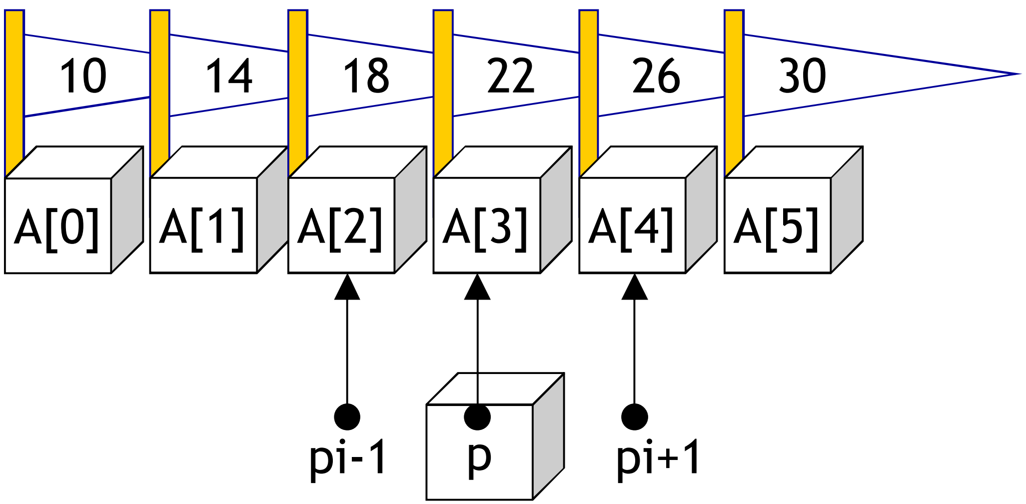



```
char a;          // Declare a character variable
char *p;         // Declare a character pointer
char **pp;       // Declare a pointer of a character pointer
a = 'A';
p = &a;          // Connect variable a and pointer p
pp = &p;         // Connect pointer p and pointer's pointer pp
```

Pointer Operation

```
int A[6], int *pi;
pi = &A[3];
pi+1 = ?
pi-1 = ?
```

pi	// pointer
pi + 1	// object after the object pointed to by pointer p
pi - 1	// object before the object pointed to by pointer p



(Review) Data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Pointer Operation

```
p        // pointer
*p       // the value pointed to by the pointer
*p++     // Get the value pointed to by the pointer, then increment the pointer by one.
*p--     // Get the value pointed to by the pointer, then decrement the pointer by one.
(*p) ++  // Increment the value pointed to by the pointer.
```

```c
int a1 = 10;
int *p;
p = &a1;

printf("%d\n", p);

int b = (*p)++;

printf("%d\n", p);
printf("%d\n", *p);
printf("%d\n", b);
```

⇨

5896696
5896696
11
10

```c
int a1 = 10;
int *p;
p = &a1;

printf("%d\n", p);

int b = *p++;

printf("%d\n", p);
printf("%d\n", *p);
printf("%d\n", b);
```

⇨

5896696
5896700
-868996460
10

Pointer Operation

- Set to NULL when the pointer is pointing to nothing.

```
int * pi = NULL;
```

- Do not use it when it is not initialized.

```
main()
{
        char *pc;        // Pointer pi is not initialized
        *pc = 'E';       // Not recommended.
}
```

- Use explicit type conversion when converting between pointer types

```
int *pi;
float *pf;
pf = (float *)pi;
```

Dynamic Memory Allocation

- How a program allocates memory
  - Static memory allocation
  - Dynamic memory allocation

- *Static* memory allocation
  - The memory size is fixed before the program starts, and it cannot be changed during execution.
  - An input, larger than the size initially determined, will not be processed.
  - A smaller input will waste the remaining memory.

  Ex) int buffer [100];   char name [] = "data structure";

- *Dynamic* memory allocation
  - To allocate memory during program execution
  - Allocate, use, and return as much as you need
  - Very efficient use of memory

Dynamic Memory Allocation

- Example code for dynamic memory allocation

```
main()
{
    int *pi;
    pi = (int *)malloc(sizeof(int)); // Dynamic Memory Allocation
    ...
    ...                              // Use dynamic memory
    ...
    free(pi);                        // Release dynamic memory
}
```

- Related library functions

```
malloc(size)    // memory allocation
free(ptr)       // deallocate memory
sizeof(var)     // return the size of the variable or type (in bytes)
```

Dynamic Memory Allocation

- 'malloc(int size)'
  - allocate the memory block of size bytes

```
(char *)malloc(100) ;                // Allocate 100 bytes
(int *)malloc(sizeof(int)); // Allocate memory for a single integer
(struct Book *)malloc(sizeof(struct Book))    // Allocate memory for a single structure
```

- 'free (void ptr)'
  - Releases the allocated memory block pointed to by ptr

- 'sizeof(var)'
  - Returns the size of a variable or type (in bytes)

```
size_t  i = sizeof( int );                              // 4
struct AlignDepends {
    char c;
    int i;
};
size_t size = sizeof(struct AlignDepends);              // 8
int  array[] = { 1, 2, 3, 4, 5 };
size_t  sizearr = sizeof( array ) / sizeof( array[0] );   // 20/4=5
```

Dynamic Memory Allocation

```
struct Example {
        int number;
        char name[10];
};
void main()
{
        struct Example *p;

        p = (struct Example *)malloc(2 * sizeof(struct Example));
        if (p == NULL) {
                fprintf(stderr, "can't allocate memory\n");
                exit(1);
        }
        p->number = 1;
        strcpy_s(p->name, " Park");
        (p + 1)->number = 2;
        strcpy_s((p+1)->name, "Kim");

        printf_s("%d    %s\n", p->number, p->name);
        printf_s("%d    %s\n", (p+1)->number, (p+1)->name);
        free(p);
}
```

```
strcpy_s(p[0].name, "Park");
```

```
strcpy_s(p[1].name, "Kim");
```

Memory Allocation of 2D Array

```
void main()
{
	int row = 3;
	int col = 3;

	int **m2 = (int **)malloc(sizeof(int *)*row);
	for (int i = 0; i<row; i++)
		m2[i] = (int *)malloc(sizeof(int)*col);

	int count = 0;
	for (int i = 0; i < row; i++)
	{
		for (int j = 0; j < col; j++)
		{
			m2[i][j] = ++count;
			printf_s("%d\n", m2[i][j]);
		}
	}
}
```

Memory Allocation of 2D Array

```
void main()
{
	int row = 3;
	int col = 3;

	int **m2 = (int **)malloc(sizeof(int *)*row);
	for (int i = 0; i<row; i++)
		m2[i] = (int *)malloc(sizeof(int)*col);

	int count = 0;
	for (int i = 0; i < row; i++)
	{
		for (int j = 0; j < col; j++)
		{
			*(*(m2 + i) + j) = ++count;
			printf_s("%d\n", *(*(m2 + i) + j));
		}
	}
}
```

```
int count = 0;
for (int i = 0; i < row; i++)
{
	int *tmp = *(m2 + i);
	for (int j = 0; j < col; j++)
	{
		tmp[j] = ++count;
		printf_s("%d\n", tmp[j]);
	}
}
```

m2=500	*(m2) =100	*(m2)+1 =104	*(m2)+2 =108
m2+1=504	*(m2+1) =200	*(m2+1)+1 =204	*(m2+1)+2 =208
m2+2=508	*(m2+2) =300	*(m2+2)+1 =304	*(m2+2)+2 =308

Memory Deallocation of 2D Array

- Memory Deallocation of 2D Array

```
if (m2 != NULL)
{
        free(m2[0]);
        free(m2);
        m2 = NULL;
}
```

Q: What about 2D array of double type?
Q: What about 3D array and 4D array?