Contents

Chapter 2

2.1 Arrays

2.2 Dynamically Allocated Array

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays

2.1.1. The Abstract Data Type

※ Create (2, (3, 4))
3행 4열의 2차원 배열 생성

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, ⋯ , 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*

2.1.2 Arrays in C

❖ one-dimensional array

int list[5];

list[0] [4]

i	i	i	i	i

※ i stands for int
i* stands for int pointer

int *plist[5];

plist[0] [4]

i*	i*	i*	i*	i*

cf) int (*ary)[5] ,배열포인터

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

KUT

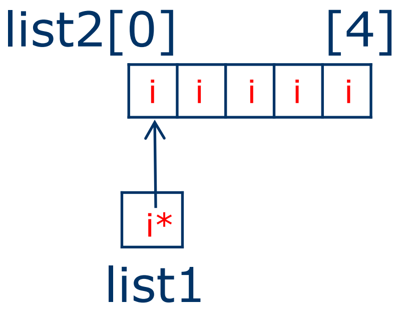
one-dimensional array & pointer

❖ **interpretations of pointers: list1, list2**

int *list1, list2[5];

list1 = list2; //(variable, constant)

```
list1 == &list2[0]
list1 + i == &list2[i]
*(list1+i) == list2[i]
```

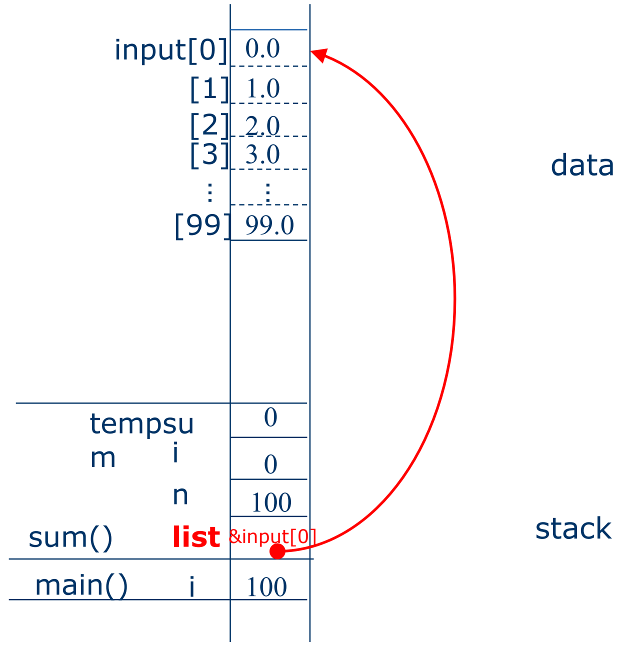


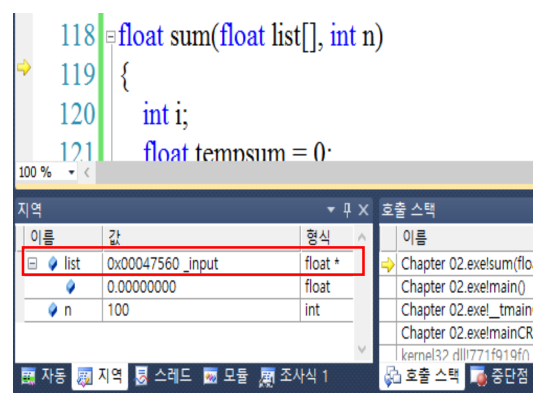
```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
void main(void)
{
   int i;
   for (i = 0; i < MAX_SIZE; i++)
      input[i] = i;
   answer = sum(input, MAX_SIZE);
   printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
   int i;
   float tempsum =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

array parameter

float * list → pointer parameter

Program 2.1: Example array program

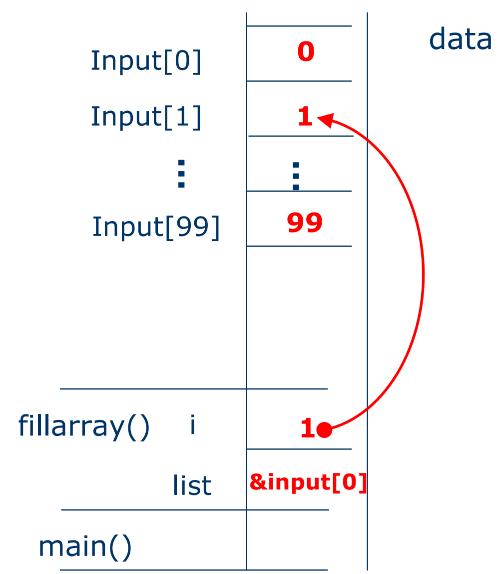




```c
#include <stdio.h>
#define MAX_SIZE 100
float sum(float[], int);
void fillarray(float[], int);
float input[MAX_SIZE], answer;
void main(void)
{
    fillarray(input, MAX_SIZE);
    answer = sum(input, MAX_SIZE);
    printf("The sum is: %f\n", answer);
}
void fillarray(float list[], int n)
{
    int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

■ **left-side of equal sign**

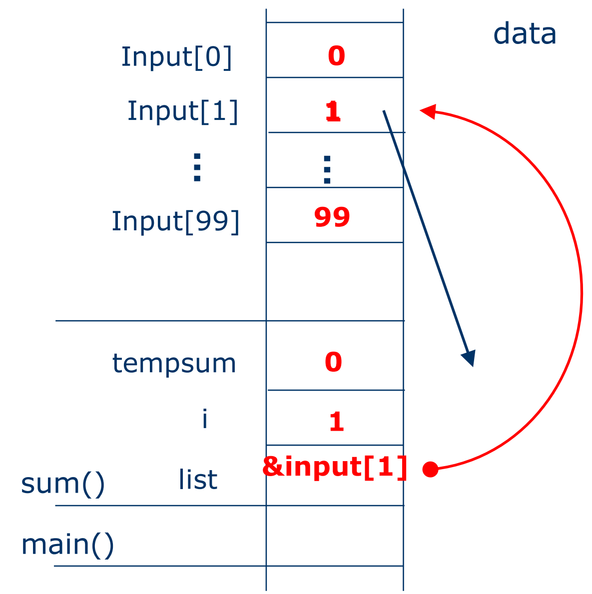
- the value produced on the right-hand side is stored in the location (*list+i*)



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

■ right-side of equal sign

- the value pointed at by (*list+i*) is returned



KUN

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

In C, array parameters have their values altered, despite the fact that the parameter passing is done using *call-by-value*.

2.1.2 Arrays in C

int one[] = {0, 1, 2, 3, 4};
print1(&one[0], 5);

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

Program 2.2: One-dimensional array accessed by address

Address	Contents
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

2.2 Dynamically Allocated Arrays

2.2.1 One-dimensional Arrays

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

i nt i,n,*list;
printf("Enter the number of numbers to generate: ");
scanf_s ( "%d", &n);
if( n < 1 ) {
fprintf(stderr, "Improper value of n\n");
e x it(EXIT_FAILURE);
MALLOC(list, n * sizeof(int));
```

2.2.2 Two-Dimensional Arrays

- **A multidimensional array in C**
  - *Array-of-arrays* representation

    int x[3][5];

x[i] : a pointer to zeroth element of row i of the array

x[i][j] : an element accessed by the address,

x[i]+j*sizeof(int)

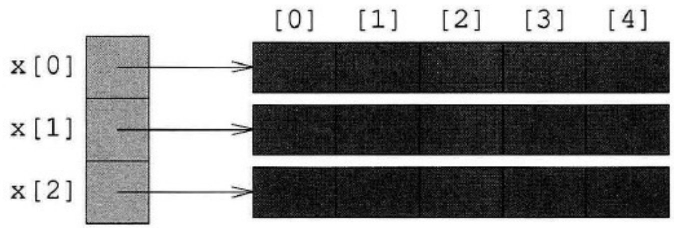
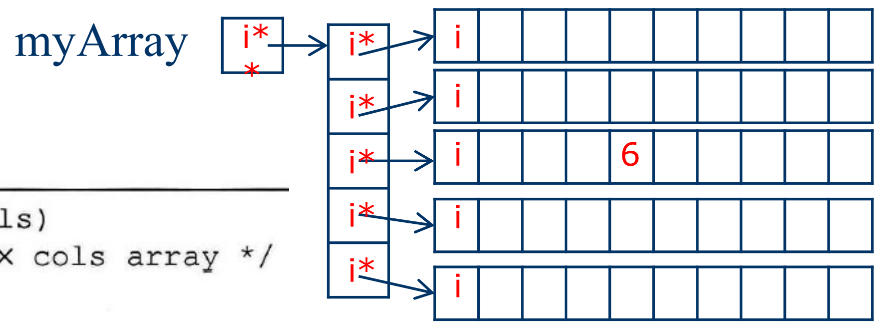


Figure 2.2: Array-of-arrays representation

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;
}
```

포인터배열

Program 2.3: Dynamically create a two-dimensional array

int *arr2[10] : 포인터 배열, arr2는 상수
int (*arr2)[10] : 배열 포인터, arr2는 [10]을 가르키는 포인터 변수

정적 Array

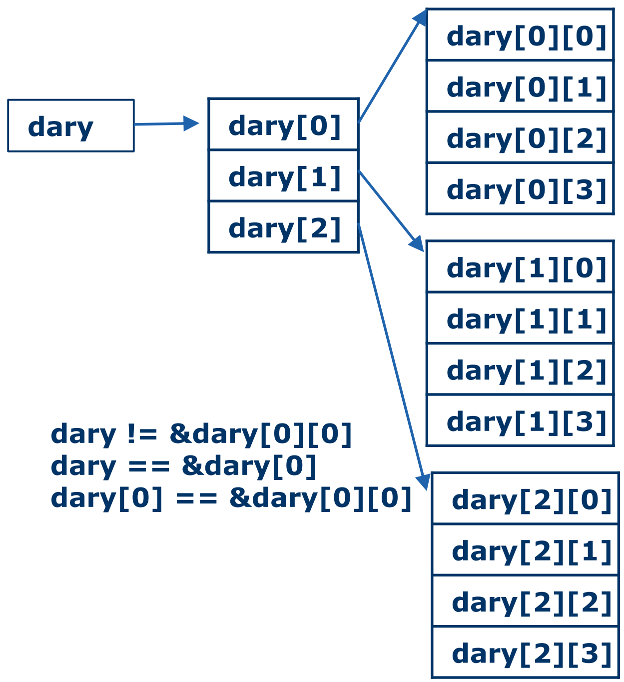
int sary[3][4]

동적 Array

int **dary
dary는 동적으로 할당 받은 2차원 배열[3][4]의 시작 주소를 가지고있는 포인터임

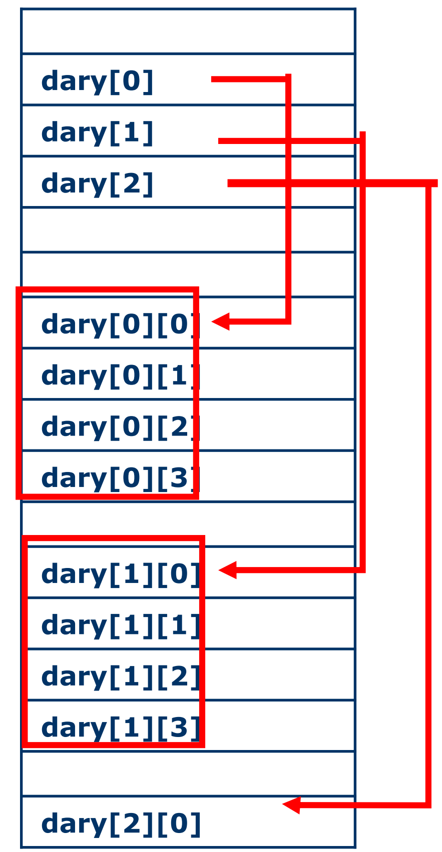
sary

sary ==
&sary[0][0]
== sary[0]



dary != &dary[0][0]
dary == &dary[0]
dary[0] == &dary[0][0]

dary

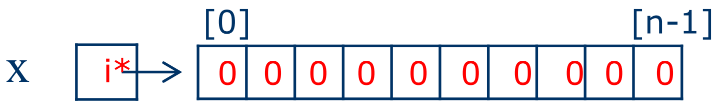


calloc

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

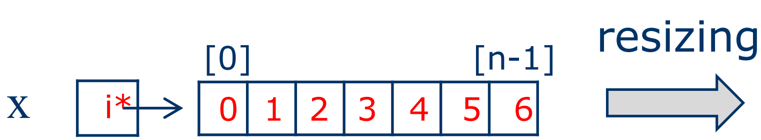
Write clean robust programs

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

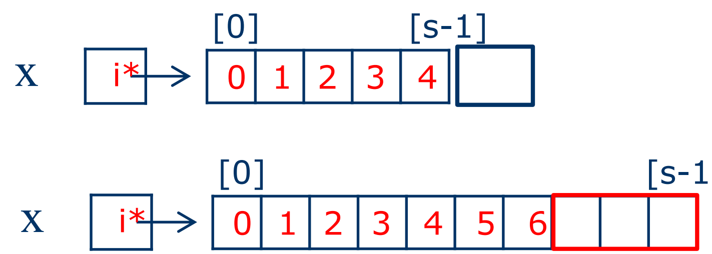


realloc

```
/* changes the size of memory block pointed by x to
s*sizeof(int) */
int *old, *x, s;
old = x;
if ( (x = (int *)realloc(x, s*sizeof(int))) == NULL ){
        free(old);
        exit(EXIT_FAILURE);
}
…
free(x);
```



재 할당이 적을 경우: 메모리 반납



재 할당이 클 경우: 메모리 추가할당

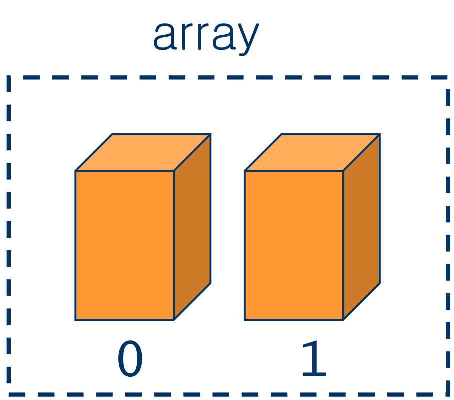
realloc

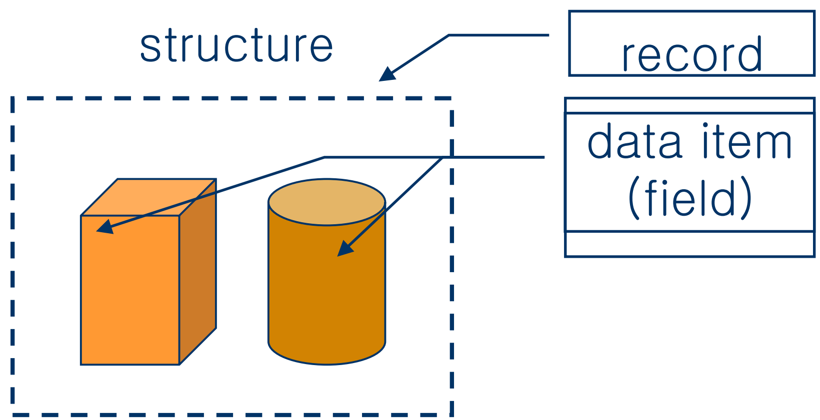
```
#define REALLOC(p,s)\
if (! ( (p) = realloc (p, s))) {\
     fprintf(stderr, "Insufficient memory");\
     exit(EXIT_FAILURE);\
}
```

- **Correct error.**

```
#define REALLOC(o, p, s) \
          if (!((p) = realloc(o, s))) {\
                    free(o);\
                    fprintf(stderr, "Insufficient memory"); \
                    exit(EXIT_FAILURE);\
       }
   ...
REALLOC(old, x, s*sizeof(int));//x = realloc(x, s*sizeof(int))
```

2.3.1 Structures






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

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

2.3 Structures and Union

2.3.1 Structures

- **Called a *record***
- **Collection of data items**
  - Each item is identified as to its type and name

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

    - man is a variable
      - strcpy_s(man.name, sizeof(name), “kim”);
      - man.age = 10;
  - Structure member operator : dot(.)
    - struct person a, b;
    - a and b are variables.

KUN

2.3 Structures and Union

- **Using the typedef statement**

```
typedef structure {
    char name[10]
    int age;
    float salary;
} human;

human person1, person2; // Declaration of variables
if(strcmp(person1.nae, person2.name))
    printf("The two people do not have the same name\n");
else
    printf("The two people have the same name\n");
```

•

KUN

2.3 Structures and Union

- **Structure assignment : person1=person2;**
  - in ANSI C, OK!
    - However, don't use the assignment operation when the structure has a pointer to a memory space. Why?
- **Check of equality or inequality : if(person1==person2)**
  - cannot be checked directly

2.3 Structures and Union

❖ strcpy_s format

- errno_t strcpy_s(char *dest, rsize_t dest_size, const char *src);
    - Return Value
    - Zero if successful; otherwise, an error.

```
#include <stdio.h>      // for printf
#include <string.h>     // for strcpy_s

int main(void)
{
    char stringBuffer[80];
    strcpy_s(stringBuffer, sizeof(stringBuffer), "Hello world from ");
    printf("stringBuffer = %s\n", stringBuffer);
}
```

2.3 Structures and Union

strcmp format

- int strcmp (const char* str1, const char* str2);
  - return 0 : str1 and str2 are equal
  - return >0 : str1 is greater than str2
  - return <0 : str2 is lower than str2

```
#include <stdio.h>
#include <string.h>
int main() {
        char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
        int result;
        // comparing strings str1 and str2
        result = strcmp(str1, str2);
        printf("strcmp(str1, str2) = %d\n", result);
        // comparing strings str1 and str3
        result = strcmp(str1, str3);
        printf("strcmp(str1, str3) = %d\n", result);
        return 0;
}
```

2.3 Structures and Union

❖ Check of equality or inequality(cont’)

```
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;
}
```

2.3 Structures and Union

- **A structure within a structure**

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

```
humanBeing person1, person2;
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2.3.2 Unions

The fields share their memory space

Only one field is "active" at any given time.

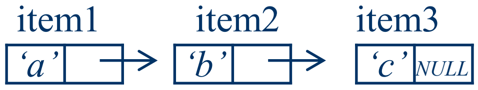
```
typedef struct {
        enum tagField {female, male) sex;
        union {
        int children;
        int beard;
        } u;
} sexType;
```

```
typedef struct {
    char name[IO];
    int age;
    float salary;
    date dob;
    sexType sexinfo;
} humanBeing;
humanBeing personl, person2;
personl.sexinfo.sex =male;
personl.sexinfo.u.beard = FALSE;
person2.sexinfo.sex = female;
person2.sexinfo.u.children = 4;
```

2.3.4 Self-Referential Structures

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

```
typedef struct list {
      char data;
      list *link ;
} ;
list iteml, item2, item3;
iteml.data 'a';
item2 .data 'b';
item3.data 'c';
iteml.link = &item2;
item2.link = &item3;
 item3.link  = NULL;
```



2.4.1 The Abstract Data Type

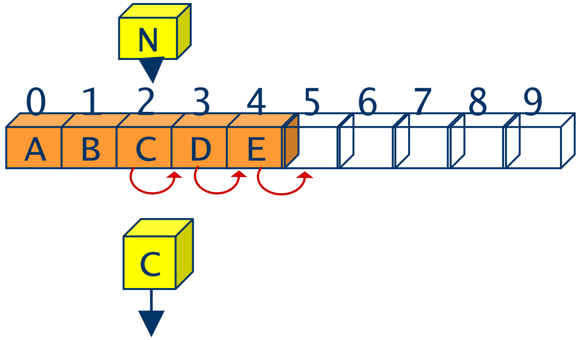
- ***Ordered list*** **or** ***linear list***
  - an ordered set of data items
- ex) Days-of-week

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

  - denote as ($item_0$, $item_1$, …, $item_{n-1}$)
  - empty list : ( )

Implementation of Ordered List

- **Array**
  - associate the list element, *item*$_i$, with the array index *i*
  - *sequential mapping*
  - retrieve, replace an item, or find the length of a list, in constant time
  - problems in insertion and deletion
    - sequential mapping forces us to move items
- **Linked List**
  - *Non-sequential mapping*
  - Chapter 4



2.4 Polynomials

- **Manipulation of symbolic polynomials**
  - $A(x) = 3x^{20} + 2x^5 + 4$, $B(x) = x^4 + 10x^3 + 3x^2 + 1$
- **degree : the largest exponent of a polynomial**
- **assumption: unique exponents arranged in decreasing order**
  - When $A(x) = \Sigma a_i x^i$ and $B(x) = \Sigma b_i x^i$,
  - $A(x) + B(x) = \Sigma (a_i + b_i) x^i$
  - $A(x)\, B(x) = \Sigma (a_i x^i\, \Sigma (b_j x^j))$

2.4 Polynomials

operations on ordered list

- find the length n
- read the items in a list from right to left (or left to right)
- retrieve ith item, $0 \le i < n$
- replace ith item's value, $0 \le i < n$
- insert ith position, $0 \le i < n$
- delete ith item, $0 \le i < n$

2.4 Polynomials

ADT *Polynomial* is

objects: $p(x) = a_1x^{e_1} + \cdots + a_nx^{e_n}$; 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 $poly, poly1, poly2 \in Polynomial$, $coef \in Coefficients$, $expon \in Exponents$

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** ($expon \in poly$) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** ($expon \in poly$) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
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 $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial $poly1 + poly2$
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial $poly1 \cdot poly2$

end *Polynomial*

ADT 2.2: Abstract data type *Polynomial*

Representation of polynomials in C

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

- $A(x) = \Sigma_{i=0}^{n} a_i x^i$ **would be represented as :**
  - a.degree = n
  - a. coef [i] = a_{n-i}, $n < MAX\_DEGREE$
- ※ *a.coef[i] is the coefficient of* x^{n-i}

Representation of polynomials in C

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

1000

2	0	0	0	0	0	....	0	0	0	0	0	0	0	1

4

1	10	3	0	1

Initial version of *padd* function (cont')

$D(x) = 0$
$A(x) = 2x^{1000} + 2x^3$
$B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$

(step1)
$D(x) = 2x^{1000}$
$A(x) = 2x^3$
$B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$

(step2)
$D(x) = 2x^{1000} + x^4$
$A(x) = 2x^3$
$B(x) = 10\,x^3 + 3\,x^2 + 1$

(step3)
$D(x) = 2x^{1000} + x^4 + 12x^3$
$A(x) = 0$
$B(x) = 3\,x^2 + 1$

(step4)
$D(x) = 2x^{1000} + x^4 + 12x^3$
$+3\,x^2 + 1$
$A(x) = 0$
$B(x) = 0$

2.4.2 Polynomial Representation

```
#define COMPARE(x, y) ( ((x) < (y)) ? -1 : ((x) == (y)) ? 0: 1 )  ※ p.12
```

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero()
while (! IsZero(a) && ! IsZero(b)) do {
   switch COMPARE(LeadExp(a), LeadExp(b)) {
      case -1: d =
         Attach(d,Coef(b,LeadExp(b)),LeadExp(b));
         b = Remove(b,LeadExp(b));
         break;
      case 0: sum = Coef( a, LeadExp(a))
                    + Coef(b, LeadExp(b));
         if (sum) {
            Attach(d,sum,LeadExp(a));
            a = Remove(a,LeadExp(a));
            b = Remove(b,LeadExp(b));
            }
         break;
      case 1: d =
         Attach(d,Coef(a,LeadExp(a)),LeadExp(a));
         a = Remove(a,LeadExp(a));
   }
}
insert any remaining terms of a or b into d
```

Program 2.5: Initial version of *padd* function

Representation of polynomials in C(cont')

```
MAX-TERMS 100 /*size of terms array*/
typedef struct {
      float coef;
      int expon;
} polynomial;
polynomial terms[MAX-TERMS);
int avail = 0;
```

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

coef	2	1
exp	1000	0
	0	1

Representation of polynomials in C(cont')

Using one array to represent two polynominals

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

	startA	*finishA*	*startB*			*finishB*	*avail*
	↓	↓	↓			↓	↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

iterations $\leq m+n-1$

$\leq m$

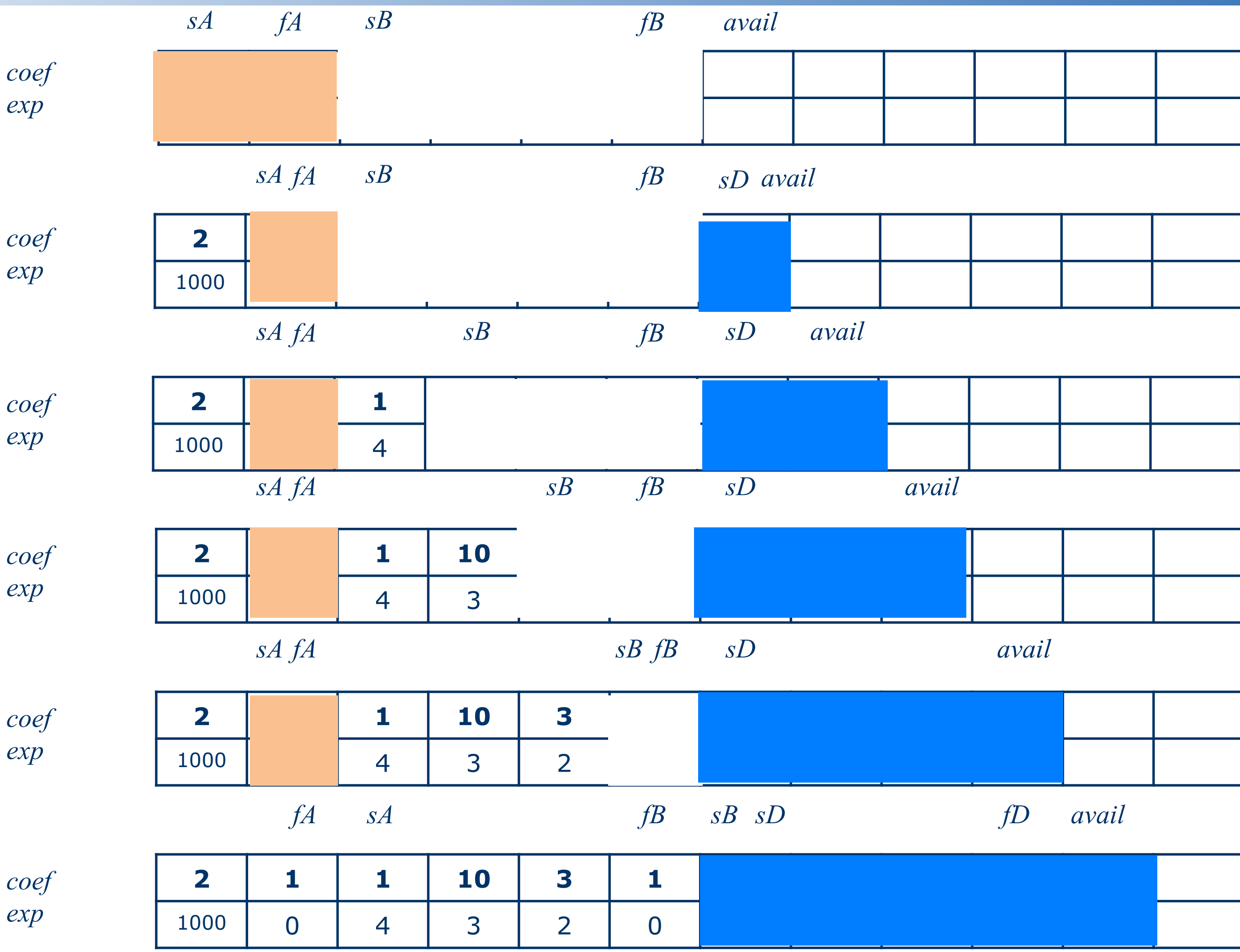
$\leq n$

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

Program 2.6: Function to add two polynomials

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

Program 2.7: Function to add a new term



sA fA sB fB avail
coef
exp
sA fA sB fB sD avail
coef
exp
2
1000
sA fA sB fB sD avail
coef
exp
2 1
1000 4
sA fA sB fB sD avail
coef
exp
2 1 10
1000 4 3
sA fA sB fB sD avail
coef
exp
2 1 10 3
1000 4 3 2
fA sA fB sB sD fD avail
coef
exp
2 1 1 10 3 1
1000 0 4 3 2 0

2.4.3 Polynomial Addition

Analysis of *padd*

- Let *m* and *n* be the number of nonzero terms in A and B, respectively.
- ① If *m*>0 and *n*>0, **while loop**
  - each iteration : O(1)
  - The iteration terminates when either *startA* or *startB* exceeds *finishA* or *finishB*, respectively
  - The number of iterations is bounded by $m+n-1$
    - the worst case : ex) $a(x) = x^6+x^4+x^2+x^0$, $b(x) = x^7+x^5+x^3+x^1$
- ② The remaining **two for loops** → O($m+n$)
- ①&② → **O($m+n$)**