CHAPTER 2

ARRAYS AND STRUCTURES

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C /2nd Edition”,
Silicon Press, 2008.

Arrays

Array: a set of index and value

data structure

For each , there is a associated with that index.

representation (possible)

implemented by using consecutive memory.

Structure *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 ∈ 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 *i*th element is the size of the *i*th 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* 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

Arrays in C

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

list[5]:

list[0], list[1], list[2], list[3], list[4]

*plist[5]:

plist[0], plist[1], plist[2], plist[3], plist[4]

implementation of 1-D array

list[0]   base address = α

list[1]

list[2]

list[3]

list[4]

Arrays in C (*Continued*)

Compare int *list1 and int list2[5] in C.

Same: list1 and list2 are .

Difference: list2 reserves .

Notations:

list2 - a pointer to list2[0]

(list2 + i) - a pointer to list2[i] =

*(list2 + i) =

Example: 1-dimension array addressing

int one[] = {0, 1, 2, 3, 4};

Goal: print out address and value

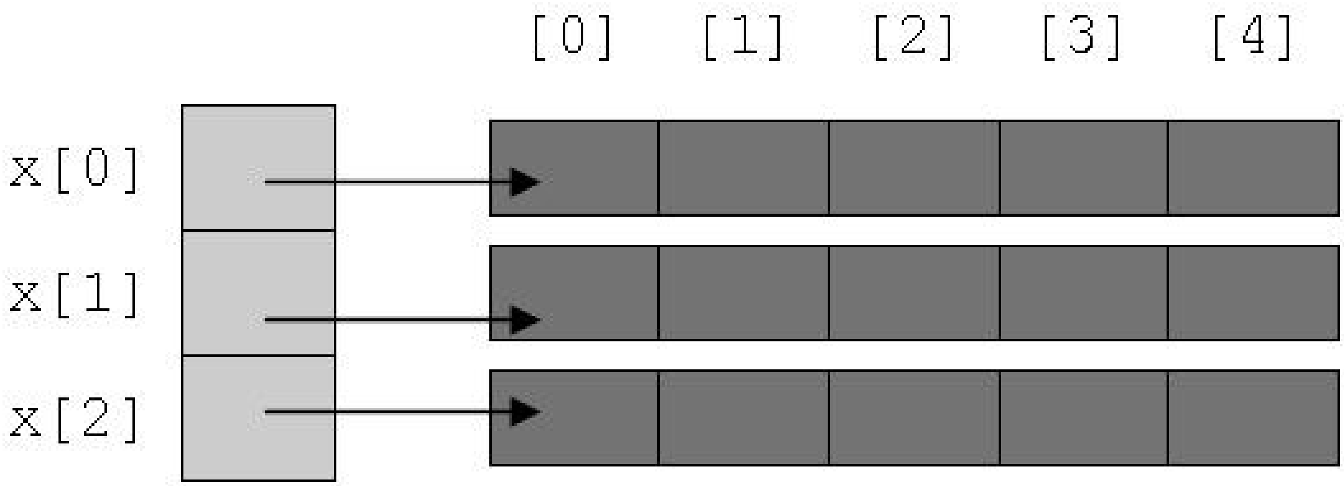
```
void print1(int *ptr, int rows)
{
 /* print out a one-dimensional array using a pointer */
        int i;
        printf("Address Contents\n");
        for (i=0; i < rows; i++)
                printf("%8u%5d\n", ptr+i, *(ptr+i));
        printf("\n");
}
```

call print1(&one[0], 5)

Address	Contents
1228	0
	1
	2
	3
	4

Two-dimension array

Int x[3][5];



Structures (records)

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

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

Create structure data type

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

human_being person1, person2;
```

If (person1 == person2) ?

```
int humansEqual (humanBeing person1, humanBeing person2)
{
   if (strcmp(person1.name, person2.name))
         return FALSE;
   if (person1.age != person2.age)
         return FALSE;
   if (person1.salary != person2.salary)
         return FALSE;
   return TRUE;
}
```

```
if (humansEqual( person1, person2 ))
```

A structure within a structure

```
typedef struct {
        int month;
        int day;
        int year;
        } date;

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

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

Unions

Similar to struct, but only one field is active.

Example: Add fields for male and female.

```
typedef struct sex_type {
        enum tag_field {female, male} sex;
        union {
                int children;
                int beard;
                } u;
        };
typedef struct human_being {
        char name[10];
        int age;
        float salary;
        date dob;
        sex_type sex_info;
        }
```

```
human_being person1, person2;
person1.sex_info.sex=male;
person1.sex_info.u.beard=FALSE;
```

Self-Referential Structures

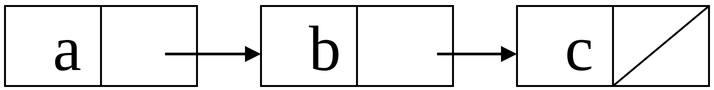
One or more of its components is a pointer to itself.

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

Construct a list with three nodes

malloc: obtain a node

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



Ordered List Examples

ordered (linear) list: (item1, item2, item3, …, item*n*)

- (MONDAY, TUEDSAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAYY, SUNDAY)
- (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)
- (1941, 1942, 1943, 1944, 1945)
- (a_1, a_2, a_3, …, a_{n-1}, a_n)

Polynomials $A(X)=3X^{20}+2X^5+4$, $B(X)=X^4+10X^3+3X^2+1$

Structure *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$; 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*, *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 Lead_Exp(*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(*poly1*, *poly2*) ::= **return** the polynomial $poly1 + poly2$

Polynomial Mult(*poly1*, *poly2*) ::= **return** the polynomial $poly1 \cdot poly2$

End *Polynomial*

Polynomial Addition

data structure 1:

```
#define MAX_DEGREE 101
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

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

```
case 1: d =
        Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
        a = Remove(a, Lead_Exp(a));
      }
  }
insert any remaining terms of a or b into d
```

***Program 2.4 :**Initial version of *padd* function(p.62)

$\boxed{\text{Data structure 2:}}$ use one global array to store all polynomials

$A(X)=2X^{1000}+1$

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

***Figure 2.2:** Array representation of two polynomials (p.63)

	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

storage requirements: start, finish, 2*(finish-start+1)

nonparse: twice as much as (1)
when all the items are nonzero

```
MAX_TERMS 100 /* size of terms array */
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

*(p.62)

Add two polynomials: D = A + B

```
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;
}
```

Analysis: O(n+m)
where n (m) is the number of nonzeros in A(B).

```c
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(1);
      }
      terms[avail].coef  = coefficient;
      terms[avail++].expon = exponent;
}
```

Problem: Compaction is required
when polynomials that are no longer needed.
(data movement takes time.)

Sparse Matrix

$$\begin{array}{cc} & \begin{array}{ccc} \text{col 1} & \text{col 2} & \text{col 3} \end{array} \\ \begin{array}{c} \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5} \end{array} & \begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix} \end{array}$$

5*3

$$\begin{array}{cc} & \begin{array}{cccccc} \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} & \text{col6} \end{array} \\ \begin{array}{c} \text{row0} \\ \text{row1} \\ \text{row2} \\ \text{row3} \\ \text{row4} \\ \text{row5} \end{array} & \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

6*6

15/15

8/36

sparse matrix
data structure?

SPARSE MATRIX ABSTRACT DATA TYPE

Structure *Sparse_Matrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item.*

functions:

for all $a, b \in Sparse\_Matrix$, $x \in item$, $i, j, max\_col, max\_row \in index$

Sparse_Marix Create(*max_row*, *max_col*) ::=

return a *Sparse_matrix* that can hold up to $max\_items = max\_row \times max\_col$ and whose maximum row size is *max_row* and whose maximum column size is *max_col.*

Sparse_Matrix Transpose(*a*) ::=

return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=

if the dimensions of a and b are the same

return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.

else return error

Sparse_Matrix Multiply(*a*, *b*) ::=

if number of columns in a equals number of rows in **b**

return the matrix *d* produced by multiplying a by *b* according to the formula: *d* [*i*] [*j*] = Σ(a[i][k]•b[k][j]) where *d (i, j)* is the *(i,j)*th element

else return error.

(1) Represented by a two-dimensional array.
Sparse matrix wastes space.

(2) Each element is characterized by <row, col, value>.

# of rows (columns)

# of nonzero terms

	row	col	value
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

(a)

transpose $\longrightarrow$

	row	col	value
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

(b)

row, column in ascending order

Sparse_matrix Create(max_row, max_col) ::=

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

# of rows (columns)
# of nonzero terms

* (P.69)

Transpose a Matrix

(1) for each row i

take element <i, j, value> and store it
in element <j, i, value> of the transpose.

difficulty: where to put <j, i, value>

(0, 0, 15) ====> (0, 0, 15)
(0, 3, 22) ====> (3, 0, 22)
(0, 5, -15) ====> (5, 0, -15)
(1, 1, 11) ====> (1, 1, 11)

Move elements down very often.

(2) For all elements in column j,

place element <i, j, value> in element <j, i, value>

Sparse Matrix Multiplication

Definition: $[D]_{m*p}=[A]_{m*n}*[B]_{n*p}$

Procedure: Fix a row of A and find all elements in column j of B for j=0, 1, …, p-1.

Alternative 1. Scan all of B to find all elements in j.

Alternative 2. Compute the transpose of B. (Put all column elements consecutively)

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

***Figure 2.5:**Multiplication of two sparse matrices (p.73)

$$
\begin{bmatrix}
X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & X & X & 0 & 0 & 0 & 0 & 0 \\
X & & & & & . & & & & \\
. & & & & & & . & & & \\
. & & & & & & & X & 0 & 0 \\
. & & & & & & & & X & 0 \\
X & X & X & X & X & X & X & X & X & X
\end{bmatrix}
$$

$$
\begin{bmatrix}
X & X & X & X & X & X & X & X & X & X \\
0 & X & X & X & X & X & X & X & X & X \\
0 & 0 & X & X & X & X & X & X & X & X \\
0 & 0 & 0 & X & X & X & X & X & X & X \\
0 & 0 & 0 & 0 & X & X & X & X & X & X \\
0 & & & & & . & & & & \\
. & & & & & & . & & & \\
. & & & & & & & X & X & X \\
. & & & & & & & & X & X \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X
\end{bmatrix}
$$