Arrays and Structures

All source credit to:
All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C",
Computer Science Press, 1992.

Arrays

Array: a set of index and homogenous value

data structure

For each index, there is a value 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

*__Structure 2.1:__ Abstract Data Type *Array* (p.50)

Arrays in C

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

list[5]: five integers

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

*plist[5]: five pointers to integers

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

implementation of 1-D array

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*size(int)

Arrays in C *(Continued)*

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

Same: list1 and list2 are pointers.
Difference: list2 reserves five locations.

Notations:

list2 - a pointer to list2[0]
(list2 + i) - a pointer to list2[i] (&list2[i])
*(list2 + i) - list2[i]

Example: 1-dimension array addressing

Goal: print out address and value

```
void print1()
{
        int i;   int one[5] = {0, 1, 2, 3, 4};
int *ptr = one;
printf(“Address Contents\n”);
for (i=0; i < 5; i++)
{       printf("%u %d\n", ptr+i, *(ptr+i));
        printf("%u %d\n", one+i, one[i]);
        printf("%u %d\n", one+i, *(one+i));
}
printf("\n");
}
```

call print1()

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

*__Figure 2.1:__ One- dimensional array addressing (p.53)

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;
```

Unions

Similar to struct, but only one field is active.

Example: Add fields for male and female.

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

```
human_being person1, person2;
person1.gender_info. gender =male;
person1. gender_info.u.beard=FALSE;
```

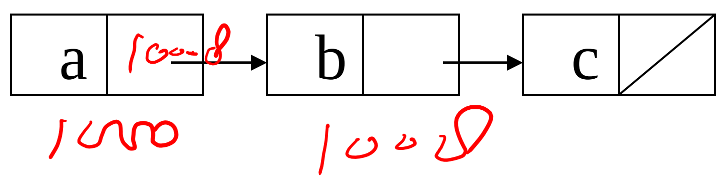
Self-Referential Structures

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

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

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node

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



```
#include <stdio.h>
#include <stdlib.h>
struct Node {
        int data;
        struct Node* next;};

// This function prints contents of linked list starting
// from the given node
void printList(struct Node n)
{ struct Node *p;
   p=&n;
        while ( p!=NULL) {
                printf(" %d ", p->data);
                p = p->next;
                    }
}

// Driver's code
int main()
{       struct Node head;
        struct Node second;
        struct Node third;
        // allocate 3 nodes in the heap
        //head = (struct Node*)malloc(sizeof(struct Node));
        //second = (struct Node*)malloc(sizeof(struct Node));
        //third = (struct Node*)malloc(sizeof(struct Node));
        head.data = 1; // assign data in first node
        head.next = &second; // Link first node with second
        second.data = 2; // assign data to second node
        second.next = &third;
        third.data = 3; // assign data to third node
        third.next = NULL;
        // Function call
        printList(head);
        return 0;       }
```

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)
- (1941, 1942, 1943, 1944, 1945)
- (a_1, a_2, a_3, …, a_{n-1}, a_n)

Operations on Ordered List

- Find the length, n , of the list.
- Read the items from left to right (or right to left).
- Retrieve the i’th element.
- Store a new value into the i’th position.
- Insert a new element at the position i , causing elements numbered i, i+1, …, n to become numbered i+1, i+2, …, n+1
- Delete the element at position i , causing elements numbered i+1, …, n to become numbered i, i+1, …, n-1

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*ϵ*Polynomial*, *coef* ϵ *Coefficients*, *expon*ϵ*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* ϵ *poly*) **return** its coefficient **else return** Zero
Exponent Lead_Exp(*poly*)	::= **return** the largest exponent in *poly*
Polynomial Attach(*poly,coef, expon*)	::= **if** (*expon* ϵ *poly*) **return** error **else return** the polynomial poly with the term <*coef, expon*> inserted

Polynomial Remove(*poly, expon*) ::= **if** (*expon* ϵ *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*

***Structure 2.2:**Abstract data type *Polynomial* (p.61)

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
```

advantage: easy implementation
disadvantage: waste space when sparse

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

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

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

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).

***Program 2.5:** Function to add two polynomial (p.64)

```
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;
}
```

***Program 2.6:**Function to add anew term (p.65)

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

Sparse Matrix

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

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

(a) 15/15

(b) 8/36

<u>*Figure 2.3:Two matrices</u>

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: $d[i][j] = \sum (a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i,j)th element
else return error

*** Structure 2.3:** Abstract data type Sparse-Matrix (p.68)

(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

	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

***Figure 2.4:** Sparse matrix and its transpose stored as triples (p.69)

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>

```
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 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 */
```

columns

elements

```
b[currentb].row = a[j].col;
b[currentb].col  = a[j].row;
b[currentb].value = a[j].value;
currentb++
    }
  }
}
```

* **Program 2.7:** Transpose of a sparse matrix (p.71)

Scan the array “columns” times.
The array has “elements” elements.

==> O(columns*elements)

Discussion: compared with 2-D array representation

O(columns*elements) vs. O(columns*rows)

elements --> columns * rows when nonsparse
O(columns*columns*rows)

Problem: Scan the array “columns” times.

Solution:

Determine the number of elements in each column of the original matrix.

==>

Determine the starting positions of each row in the transpose matrix.

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

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

```
void fast_transpose(term a[ ], term b[ ])
{
/* the transpose of a is placed in b */
  int row_terms[MAX_COL], starting_pos[MAX_COL];
  int i, j, num_cols = a[0].col, num_terms = a[0].value;
  b[0].row = num_cols; b[0].col = a[0].row;
  b[0].value = num_terms;
  if (num_terms > 0){ /*nonzero matrix*/
           for (i = 0; i < num_cols; i++)
columns          row_terms[i] = 0;
           for (i = 1; i  <= num_terms; i++)
elements         row_term [a[i].col]++
           starting_pos[0] = 1;
           for (i =1; i < num_cols; i++)
columns          starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
```

elements

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

***Program 2.8:**Fast transpose of a sparse matrix

Compared with 2-D array representation

O(columns+elements) vs. O(columns*rows)

elements --> columns * rows

O(columns+elements) --> O(columns*rows)

Cost: Additional row_terms and starting_pos arrays are required.

Let the two arrays row_terms and starting_pos be shared.

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)

- For example: Consider 2 matrices:

- Row Col Val   Row Col Val
- 1   2   10   1   1   2
- 1   3   12   1   2   5
- 2   1   1   2   2   1
- 2   3   2   3   1   8
- The resulting matrix after multiplication will be obtained as follows:

- Transpose of second matrix:

- Row Col Val   Row Col Val
- 1   2   10   1   1   2
- 1   3   12   1   3   8
- 2   1   1   2   1   5
- 2   3   2   2   2   1

- Summation of multiplied values:

- result[1][1] = A[1][3]*B[1][3] = 12*8 = 96
- result[1][2] = A[1][2]*B[2][2] = 10*1 = 10
- result[2][1] = A[2][1]*B[1][1] + A[2][3]*B[1][3] = 2*1 + 2*8 = 18
- result[2][2] = A[2][1]*B[2][1] = 1*5 = 5

- Hence the final resultant matrix will be:
- Row Col Val
- 1   1   96
- 1   2   10
- 2   1   18
- 2   2   5

```
void mmult (term a[ ], term b[ ], term d[ ] )
/* multiply two sparse matrices */
{
   int i, j, column, totalb = b[].value, totald = 0;
   int rows_a = a[0].row, cols_a = a[0].col,
   totala = a[0].value; int cols_b = b[0].col,
   int row_begin = 1, row = a[1].row, sum =0;
   int new_b[MAX_TERMS][3];
   if (cols_a != b[0].row){
         fprintf (stderr, “Incompatible matrices\n”);
         exit (1);
}
```

```
fast_transpose(b, new_b);
/* set boundary condition */
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;
new_b[totalb+1].col = 0;
for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1;) {
    /* mutiply row of a by column of b */
    if (a[i].row != row)  {
        storesum(d, &totald, row, column, &sum);
        i = row_begin;
        for (; new_b[j].row == column; j++) ;
        column =new_b[j].row }
    if (new_b[j].row != column)  {
        storesum(d, &totald, row, column, &sum);
        i = row_begin;
        column = new_b[j].row;}
```

cols_b + totalb

at most rows_a times

```
            else switch (COMPARE (a[i].col, new_b[j].col)) {
                case -1: /* go to next term in a */
                        i++; break;
                case 0: /* add terms, go to next term in a and b */
                        sum += (a[i++].value * new_b[j++].value);
                        break;
                 case 1: /* advance to next term in b*/
                        j++
              }
          } /* end of for j <= totalb+1 */
          for (; a[i].row == row; i++)
              ;
          row_begin = i; row = a[i].row;
        } /* end of for i <=totala */
        d[0].row = rows_a;
        d[0].col = cols_b; d[0].value = totald;
}
```

***Praogram 2.9:** Sparse matrix multiplication (p.75)

```
void storesum(term d[ ], int *totald, int row, int column,
                                                    int *sum)
{
/* if *sum != 0, then it along with its row and column
    position is stored as the *totald+1 entry in d */
    if (*sum)
      if (*totald < MAX_TERMS) {
        d[++*totald].row = row;
        d[*totald].col = column;
        d[*totald].value = *sum;
          sum=0;
      }
     else {
       fprintf(stderr, "Numbers of terms in product
                                          exceed %d\n", MAX_TERMS);
  exit(1);
  }
}
```

Program 2.10: storsum function

Analyzing the algorithm

cols_b * termsrow_1 + totalb +
cols_b * termsrow_2 + totalb +
… +
cols_b * termsrow_p + totalb
= cols_b * (termsrow_1 + termsrow_2 + … + termsrow_p) + rows_a * totalb
= cols_b * totala + row_a * totalb

O(cols_b * totala + rows_a * totalb)

Compared with matrix multiplication using array

```
for (i =0; i < rows_a; i++)
    for (j=0; j < cols_b; j++) {
        sum =0;
        for (k=0; k < cols_a; k++)
            sum += (a[i][k] *b[k][j]);
        d[i][j] =sum;
    }
```

O(rows_a * cols_a * cols_b) vs.
O(cols_b * total_a + rows_a * total_b)

optimal case: total_a < rows_a * cols_a
total_b < cols_a * cols_b

worse case: total_a --> rows_a * cols_a, or
total_b --> cols_a * cols_b