CHAPTER - 6
ARRAYS

CHAPTER 6

ARRAYS

ARRAYS IN COMPUTERS **(Reading)**
DECLARATION OF ARRAYS
INITIALIZATION OF ARRAY
PRINT ASTERISKS IN DIAMOND SHAPE
ASTERISKS OUTPUT AND ANALYSIS
IMPROVED ASTERISKS
ARRAY MANIPULATIONS
MULTIDIMENSIONAL ARRAYS
ELEMENTS OF MULTIDIMENSIONAL ARRAYS **(Reading)**
INITIALIZATION OF MULTIDIMENSIONAL ARRAYS
GENERAL FORMAT OF MULTIDIMENSIONAL ARRAYS **(Reading)**
TIC-TAC-TOE GAME

ARRAYS

DECLARATION OF ARRAYS

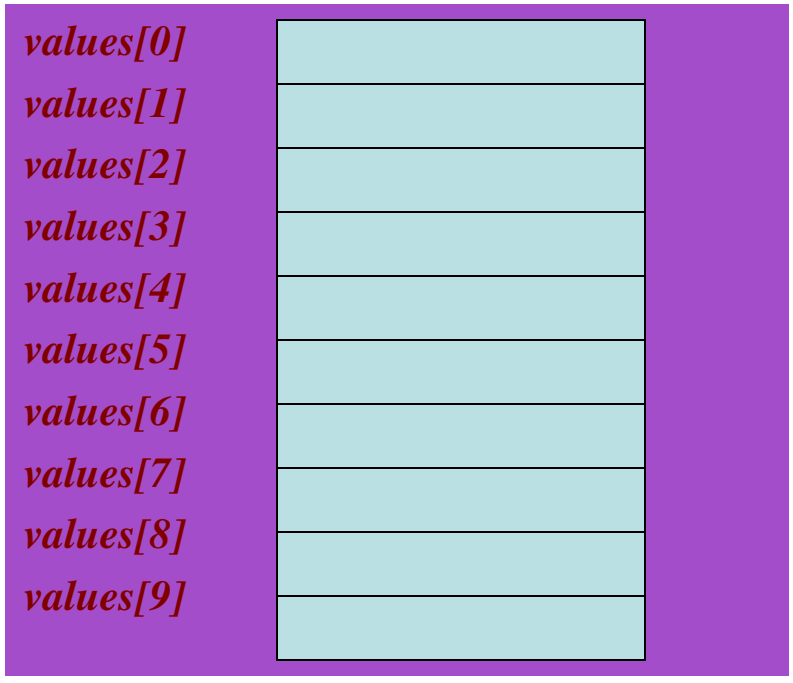
- Motivation to create an array is to avoid declaring many data variables of the same type.
- Contiguous allocation of memory space helps looping the statements and manipulating with array subscript.
- Pointers can access data elements by dereferencing.
- In C the array elements index or subscript begins with zero.

syntax:

```
data type variable-name[arraysize];
```

examples:

```
float height[50];
grades [100] = 95;
for (i =0; i < 100; ++i)
    sum = sum + grades [i];
```



ARRAYS

INITIALIZATION OF ARRAY

- Arrays can be initialized at declaration time or during execution.
- If the array size is not given at declaration, it assumes the size from the number of initialized values.
- Character arrays are not character strings. Character strings are terminated by a null character ('\0')

syntax:

```
data type array_name[size] = {list of values};
```

examples:

```
int number[3] = {0,0,0};
int counter[] = {1,1,1,1};
char  name [] = {"Rajainder Yeldandi"};
for (i =0; i < 4; ++i)
    counter[i] = i;
```

ARRAYS

EXAMPLE OF ARRAY INITIALIZATION

```
/ #include < stdio.h >
void main ( )
{
      int a[50], n, count_neg = 0, count_pos = 0, i;
      printf ("Enter the size of the array\n");
      scanf ("%d", &n);
      printf ("\n Enter the elements of the array\n");
      for ( i = 0; i < n; i++)
              scanf ("%d", &a[i]);
      for (i = 0; i <  n; i++)
      {
         if (a[i] < 0)
            count_neg++;
         else
            count_pos++;
      }
      printf ("\n There are %d negative
                numbers  in the array\n",
                         count_neg);
      printf ("There are %d positive numbers
                in       the array\n\n",
                         count_pos);
      return;
}
```

ARRAYS

OUTPUT OF ARRAY INITIALIZATION

Output of the program:

Enter the size of the array

20

Enter the elements of the array

10

20

30

40

50

60

70

80

90

100

110

120

130

140

150

160

170

180

190

200

There are 0 negative numbers in the array

There are 20 positive numbers in the array

ARRAYS

PRINTING ASTERISKS IN DIAMOND SHAPE

```
#include <stdio.h>
#define LENGTH  15
void main ()
{
      int i, j, k;
      for (i = 1;  i <= LENGTH / 2 + 1;  i++)
      {
              for (j = 1;  j <= (LENGTH / 2) - i + 1;  j++)
                  printf (" ");
              for (k = 1;  k <= i * 2 - 1;  k++)
                  printf ("*");
              printf ("\n");
      }
```

```
   for (i = 1;  i <= LENGTH / 2;  i++)
   {
         for (j = 1;  j <= i;  j++)
               printf (" ");
         for (k = 1;  k <= LENGTH - 2 * i
                                        k++)
               printf ("*");

          printf ("\n");
     }
     return;
}   /*  end of main function  */
```

ARRAYS

ASTERISKS OUTPUT AND ANALYSIS

output of the above program:

```
        *
       ***
      *****
     *******
    *********
   ***********
  *************
 ***************
  *************
   ***********
    *********
     *******
      *****
       ***
        *
```

The above program uses two outer loops to print the upper half and the lower half of the asterisks in diamond shape. It is comparatively a little easier algorithm to print with two loops with the aid of incrementing or decrementing a counter for the loop iterations. Using a single outer loop is a little more challenging and needs some imagination to play with the varying number of iterations.

One should try to improve the code after the program accomplishes the intended task. It is best to write an efficient code to start with. Writing efficient algorithm needs motivation to improve the program with less number of loop iterations and minimum lines of code will be easier to maintain.

ARRAYS

IMPROVED ASTERISKS

```
#include <stdio.h>
#include <MATH.h>
#define LENGTH 15
void main ()
{
      int i, j, k, d, spaces, asterisks;
      for (i = 1, d = LENGTH; i <= LENGTH; i++, d--)
      {
            for (j = 1; j <= fabs (d - i)/2; j++)   printf (" ");
            spaces = --j;
            asterisks = LENGTH - 2 * spaces;
            for (k = 1; k <= asterisks; k++)   printf ("*");
             printf ("\n \n");
      }
      return;
}   /* end of main function */
```

The improved asterisks program works with a single outer loop instead of two outer loops in the previous case. The improvement is done with a simple manipulation of loop iterations variables spaces and asterisks.

Iteration	1	2	3	4	5	6	7	8
Spaces	7	6	5	4	3	2	1	0
Asterisks	1	3	5	7	9	11	13	15

Iteration	9	10	11	12	13	14	15	
Spaces	1	2	3	4	5	6	7	
Asterisks	13	11	9	7	5	3	1	

output of the above program:

Same as the previous program

ARRAYS

ARRAY MANIPULATIONS

```
#define  ASIZE  12
int main()      /* sorts an array into ascending order */
{
      int  ia[ASIZE] = {2, 92, 8, 42, 68, 87, 35, 13, 72, 4,
      29, 51};
      int i, pass, hold, value, size = ASIZE;
      int n, searchkey, found = -1, temp, j = ASIZE - 1;
      printf ("\n\n Data items in original order: \n");
      /* code to print the array elements */
      for (i = 0;  i <= ASIZE - 1;  i++)
          printf ("%4d", ia[i]);
      /* code to reverse the existing array */
      for (i = 0;  i < size / 2;  i++, j --)
      {
          temp = ia[i];
          ia[i]  = ia[j];
          ia[j]  = temp;
      } /* end of reverse for loop */
```

```
printf ("\n\n Data items in
                reversed order: \n");
for (i = 0;  i <= ASIZE - 1;  i++)
    printf ("%4d", ia[i]);
printf ("\n\n Sorting the reversed
                array: \n");
for (i = 0;  i <= ASIZE - 1;  i++)
    printf ("%4d", ia[i]);
printf ("\n\n Sorting the reversed
                array: \n");
```

ARRAYS

ARRAY MANIPULATIONS

```
/* code to sort the array */
for (pass = 1;  pass <= ASIZE - 1;
                                    pass++)
    for (i = 0;  i <= ASIZE - 2;  i++)
        if (ia[i] > ia[i + 1])
    {
            hold     =  ia[i];
            ia[i]     =  ia[i + 1];
            ia[i +1]   =  hold;
     }
printf ("\n\n Data items in ascending order
                                        \n");
for (i = 0;  i <= ASIZE - 1;  i++)
    printf ("%4d", ia[i]);
printf  ("\n \n");
```

```
// code to search the array for a given element
printf  ("\n please enter the value to be searched:
                                        ");
scanf ("%d", &searchkey);
for (n = 0; n <= size - 1;  n++)
     if (ia[n] == searchkey)
     {
        value = ia[n];
        found = n;
      }
if (found != -1)
   printf  ("\n Found the value %d in the array
            at subscript: %d \n", value, found);
else
   printf  ("\n Not found the value in the
                                    array\n");
return 0;
} /* end of main routine
```

ARRAYS

ARRAY MANIPULATIONS

Output of the program:

Data items in original order:
2 92 8 42 68 87 35 13 72 4 29 51
Data items in reversed order:
51 29 4 72 13 35 87 68 42 8 92 2
Sorting the reversed array:
Data items in ascending order
2 4 8 13 29 35 42 51 68 72 87 92
please enter the value to be searched: 42
Found the value 42 in the array at subscript: 6

ARRAYS

MULTIDIMENSIONAL ARRAYS

- Applications often need to store and manipulate two dimensional data
- Matrices and tables can be assumed as a two dimensional array
- One subscript denotes the row and the other the column
- Multidimensional arrays have more than two dimensions

two dimensional array syntax:

```
data_type array_name [row_size] [column_size];
```

multdimensional array syntax:

```
data_type multi_array [row] [column] [width];
```

two dimensional initialization:

```
int table[2][3]={0,0,0,1,1,1};
int table[2][3]={{0,0,0},{1,1,1}}
```

multidimensional array example:

```
int survey[3][5][12];
float table[5][4][5][3];
```

ARRAYS

TWO DIMENSIONAL ARRAY EXAMPLE

```
#include< stdio.h >
#include< conio.h >
void main()
{
    int a[10][10], b[10][10], c[10][10], i, j, m, n, p, q;
    printf ("enter the order of the matrix\n");
    scanf ("%d %d", &p, &q);
    if (m == p && n == q)
    {
      printf ("matrix can be added\n");
     printf ("enter the elements of the matrix a");
     for (i =0;  i < m;  i++)
         for ( j = 0;  j < n;  j++)
             scanf ("%d", &a [i][j]);
    printf ("enter the elements of the matrix b");
    for (i = 0;  i < p;  i++)
       for (j = 0;  j < q;  j++)
           scanf ("%d",  &b[i][j]);
     printf ("the sum of the matrix
                        a and b is");
     for (i = 0;  i < m;  i++)
         for (j = 0;  j < n;  j++)
             c[i][j] = a [i][j] + b [i][j];
     for (i = 0;  i < m;  i++)
     {
           for (j = 0;  j < n;  j++)
              printf ("%d\t",
                              &a[i][j]);
           printf ("\n");
     }  /* end of inner for loop  */
    }   /*   end of if statement  */
}  /* end of main    */
```

ARRAYS

TIC-TACK-TOE

Tic-Tack-Toe Game Strategy:

This is 2 dim. array game between two players marking ‘x’ and ‘0’ in the array cells.

The first player can win if the second player does not use a perfect defense strategy.

The second player can rarely win unless playing against an inexperienced opponent or unless the opponent makes a mistake.

The other player can be forced to defend early game plays made by the holder of the center cell, usually resulting in a tie.

Initial setting of the grid:

Rows	Col 0	Col 1	Col 2	Col 3
Row0	--	--	--	--
Row1	--	--	--	--
Row2	--	--	--	--
Row3	--	--	--	--

result of a tie game:

Rows	Col 0	Col 1	Col 2	Col 3
Row0	**X**	**O**	**X**	**O**
Row1	**X**	**O**	**O**	**X**
Row2	**X**	**X**	**O**	**O**
Row3	**O**	**X**	**O**	**X**

ARRAYS

TIC-TAC-TOE ALGORTHM

Main program:

define a variable board of type Board of DIM size and initialize it.

show the empty board to start the game

while (neither PLAYER1 nor PLAYER2 has won,
and the board is not full)

{

get player PLAYER1 to make a move

if (PLAYER1 has not won and the board is not full)

get player PLAYER2 to make a move

}

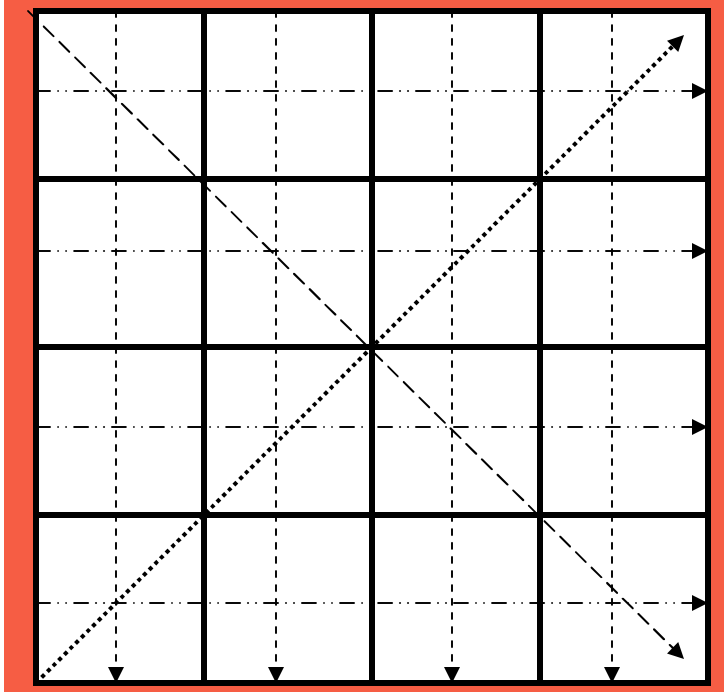
if the loop is over, either a player has won or the board is full,

if (player PLAYER1 has won), write PLAYER1 message

else if (player PLAYER2 has won),
write PLAYER2 message

else the board must be full and the game tied.
write the draw message

Grid information to be verified :



ARRAYS

TIC-TAC-TOE ALGORTHM

Row full:

Checks whether the row contains DIM pieces of type player.

Function returns 1 (true) if there are DIM pieces of type player in row, or 0 (false) otherwise.

Column full:

Checks whether the column contains DIM pieces of type player.

Function returns 1 (true) if there are DIM pieces of type player in column, or 0 (false) otherwise.

Any Row full:

Checks whether any of the DIM rows are full by using the Row Full.

Any Column full:

Checks whether any of the DIM columns are full by using the Column Full.

Major Diagonal full:

Checks whether the major diagonal is full.

Minor Diagonal full:

Checks whether the minor diagonal is full.

Player has won:

Checks if the player has won. Returns 1 if player has DIM pieces in a row, horizontally, vertically, or diagonally.

Board full:

Checks if there are any empty spots left on the board.

Move:

Prompts the player to enter an integer for the row and for the column.