Lecture 5A: Arrays

Content

I. Definition of Array

- Array is a collection of a **fixed number** of **elements** of **the same data type**
  - Common identifier (name)
  - **Stored sequentially in the memory (contiguous allocation)**

- It is a powerful data structure **for grouping alike variables for easy access**

- Although C and MATLAB arrays are based on the same computer science concept, the syntax is very different

All variables in MATLAB are arrays

```
Vect = [ 4   7   2   6   5 ]; % create a 1-D array with 5 elements
numel(Vect)   % obtain the number of elements in the array vect
X = Vect(2); % get the 2nd element and assign it to a scalar
```

Arrays in C programs must be declared before they can be used

```
int vect[5] = { 4, 7, 2, 6, 5 }; /*declare a 1-D array with 5 elements*/
int x = vect[0]; /* get the 1st element (its index is 0) and assign it to a variable */
```

The first array element has index 0

II. 1-D Array: Declaration

Syntax:

```
/* declaration of an array without initialization */
type arrayName[ numberOfElements ];

/* declaration of an array with initialization */
type arrayName[ numberOfElements ] = {list of values};
```

OPTIONAL

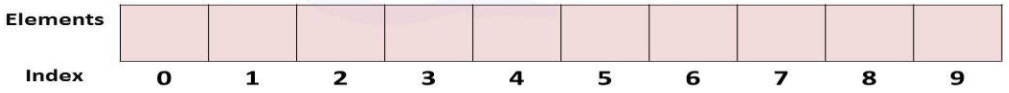
`type` – **any basic data type**
`int, double, float`
`char, short` …

`arrayName`
a valid C identifier

`numberOfElements`:
A numeric or a symbolic constant

```
int marks[10];
```

int - Array Data Type
Marks - Array Name
10 - Array Size



Examples:

```
#define SIZE 32
  .   .   .   .
float amplitude[SIZE];                    /* declared without initialization */
char letters[3] = {'A', 'E', 'I'};        /* declared and initialized */
int numbers[] = {6, 35, 128};             /* declared and initialized, array size
                                             is determined automatically [3] */
```

✓

✗ `int number [];`

…Continued

- Arrays must be declared with **a fixed number of elements** according to C89 version but not the latest


```
/*-----Declaration----*/
int itemsInStock[5];          /* size is a numeric constant */
                              /* OK, but not convenient */
```

or

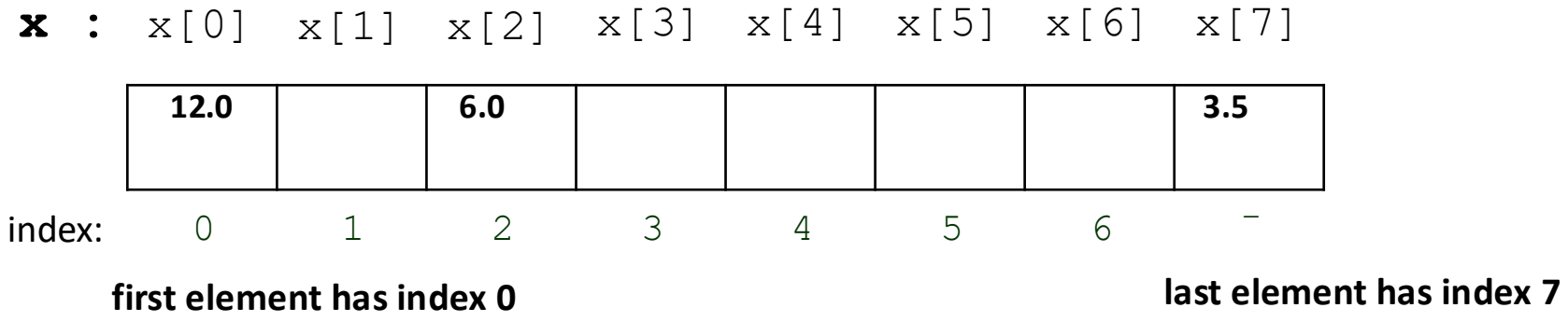
```
#define SIZE   5              /* easy to change when needed */
.    .    .
int itemsInStock[SIZE];       /* size is a symbolic constant*/
```

```
int size = 5;
int itemsInStock[size];       /* size is not a constant */
                              /* not supported by C89 */
```

1-D Array Access

- Individual variables in an array are called **elements**
- Elements in an array are accessed via **indexing**
- To access an individual element you need to specify its index

```
double x[8];  /* array declaration. 8 is the size */

 .  .   .  .
 x[0] = 12.0;  /* access an element. 0 is an index */
 x[2] = 6.0;   /* access an element. 2 is an index */
 .  .   .
 x[7] = 3.5;   /* access an element. 7 is an index */
```



first element has index 0

last element has index 7

…Continued

index can be a constant, a variable, or an expression of integral data type

Syntax:

```
arrayName[ index ]
```

Examples:

```
sum   += x[7];          /* index is a numeric constant */
temp   = x[i];          /* index is an integer variable */
x[i+1] = y[2*j - 1];    /* index is produced by an expression */
sum   += x[3.5];        /* Error ! Index must be an integer */
```

Cautions

- index must be an integral number (`int, short etc.`)
- index must range from `0` to the `numberOfelements - 1` for **correct access** `to your array elements`
- **C does not check the value of the index**
- any invalid index **may** cause **unexpected outcomes**

…Continued

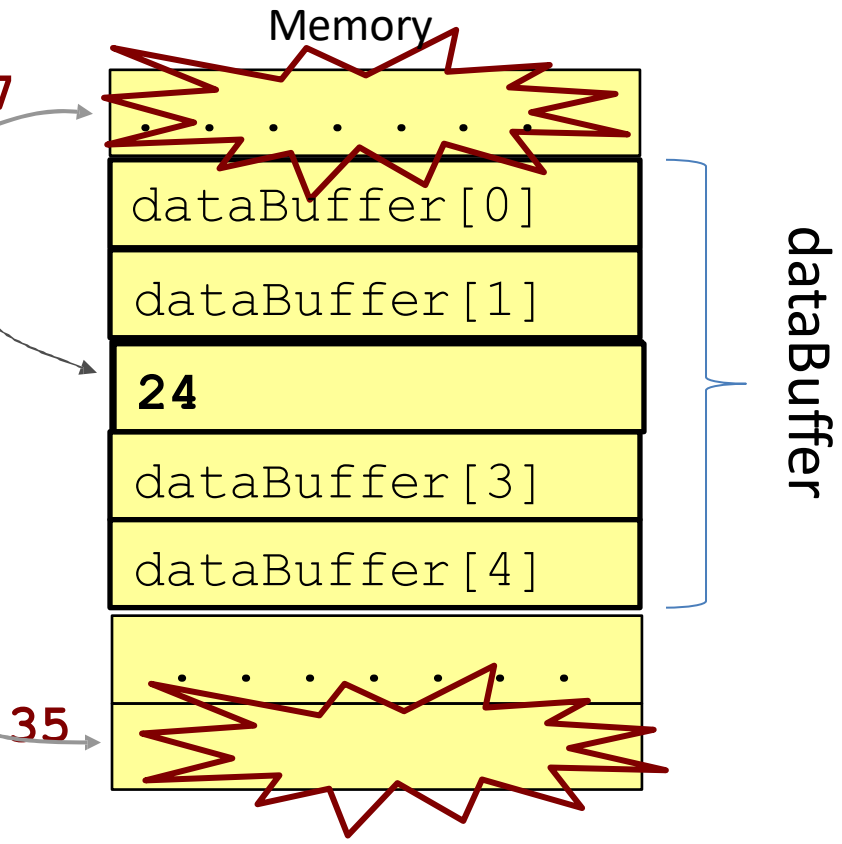
There is **NO Index range checking in C**

```
int dataBuffer[5]; /* array to store 5 integer numbers */
```

```
int dataBuffer[-1] = 17;
dataBuffer[2] = 24;
dataBuffer[6] = 35;
```

C does not check whether the index is within the valid range or not

- **No error or warning messages**
- **Can corrupt other variables**
- **Unexpected behaviour at run time or program crash: `segmentation fault`**



1-D Array Manipulation

```
double x[8];
```

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	–54.5

Statement	Explanation
`printf("%.1f", x[0]);`	Displays the value of `x[0]`, which is `16.0`.
`x[3] = 25.0;`	Stores the value `25.0` in `x[3]`.
`sum = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]`, which is `28.0` in the variable `sum`.
`sum += x[2];`	Adds `x[2]` to `sum`. The new `sum` is `34.0`.
`x[3] += 1.0;`	Adds `1.0` to `x[3]`. The new `x[3]` is `26.0`.
`x[2] = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` in `x[2]`. The new `x[2]` is `28.0`.

Array x

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	28.0	26.0	2.5	12.0	14.0	–54.5

Using Loops for Sequential Access

- To access the array elements **sequentially**, we can use **loops**
- Example: the following array `square` will be used to store the squares of the integers: 0 through 10 (e.g., square[0] is 0 , square[10] is 100)

```
#define SIZE 11
…..
int main() {
int square[SIZE], i;

for (i = 0; i < SIZE; ++i)
     square[i] = i * i;

}
```

Array square

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	1	4	9	16	25	36	49	64	81	100

Unsupported Operations

Caution: you can't use the array name to

- a constant to all the array elements

✗
```
float salePrice[SIZE];
salePrice = 15.0;
```

- one array content to another, even if they match completely in type and size

```
int inData[4]={2, 4, 8, 3}, outData[4];
```

✗
```
outData = inData;
```

- compare two array contents through the array names

✗
```
if( outData == inData )
```

- For any of the above operations, use **Loops** to go through the elements of the array

III. Arrays and Functions

- Function parameters:

  1. **Individual array elements: e**lements are passed the same way as simple data types (their **values** are copied); use arrayName[Index of element]

  2. **Complete Array**: **user array name only. The array is passed by reference** (their values are not copied, but the address of the array is)**. The called function can access the array directly**

     - Besides the array you have to pass the array **`size`** to the called function, to access the valid range of the array

- Return value:

  – C does not allow functions to return a value of the type **`array`**

```
double[] createArray( int size);  /* Error */
```

If you need to return an array, add it in the function parameters

```
void createArray( int size, int data[]);
```

…Continued

Example:

1. Function prototype

```
float findSmallest( float marks[], int size);
```

Empty [ ] indicate that this function parameter is an array

2. Function call: **pass the whole array**

```
float examMarks[50];
baseNumber = findSmallest( examMarks, 50);
```

the array name without brackets

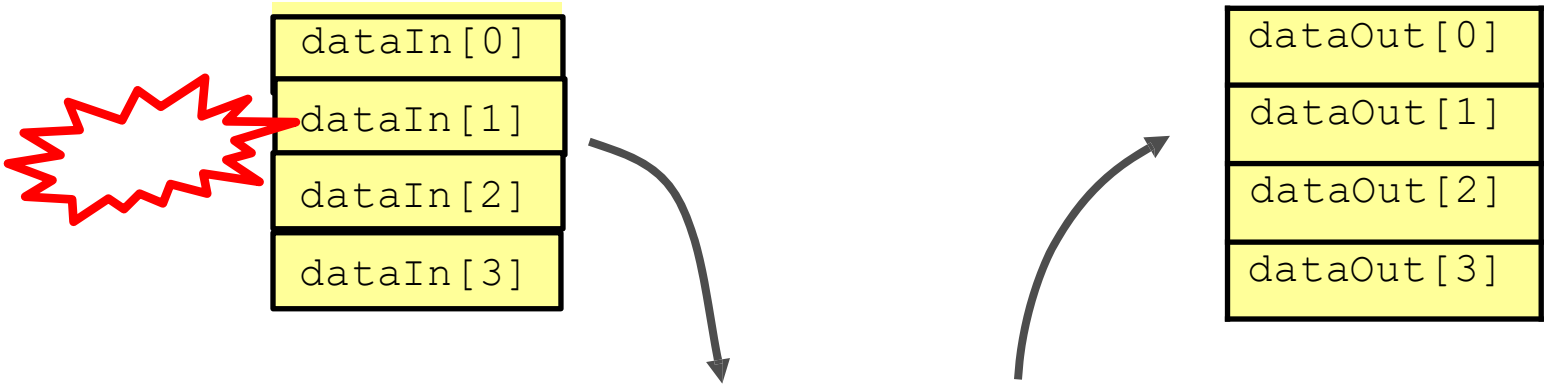
3. Function definition

```
float findSmallest( float marks[], int size )
{
  float min = marks[0];
  int i;

  for( i=1; i < size; i++ )
     if( min > marks[i] ) min = marks[i];

  return min;
}
```

Marks points to examMarks, see slide 17

…Continued

Since the function has full access to the actual array data, it may mistakenly change it




```
void addVector( int dataIn[], int dataOut[], int size )
{
  for(int i=0; i < size; i++) {
     dataOut[i] = dataIn[i] + dataIn[i]%2;
     if (dataOut[i] == 0) dataIn[i]++;
  }
  return ;
}
```

A bug: the source data array will be corrupted

Meant to be: `dataOut[i]++`

...Continued

To **prevent accidental modification of arrays**, pass them as **const**

```
int itemsIn[250];
int itemsOut[250];
itemsIn[5] = 128;
addVector( itemsIn, itemsOut, 250 );
```

```
void addVector( const int dataIn[], int dataOut[], int sz )
{                          Can't be modified        Can be modified
  for(int i=0; i < sz; i++) {
     dataOut[i] = dataIn[i] + dataIn[i]%2;
     if (dataOut[i] == 0) dataIn[i]++;
  }
  return ;
}
```

As dataIn[] can't be modified, this attempt will be detected by the compiler

Compiler error:

read only location

…Continued

```
#include <stdio.h>

int sum( const int data[], int size );

int main(void)
{
    int array[5] = {1,2,3,4,5};
    int total;
    /* call the sum() function */
    total = sum( array, 5 );
    printf("Total is %d\n", total);
    return (0);
}

/* function definition */
int sum( const int data[], int size )
{
  .  .  .
   return theSum;
}
```

If the array is not to be modified, declare it as **const**

specify only the array name, no brackets

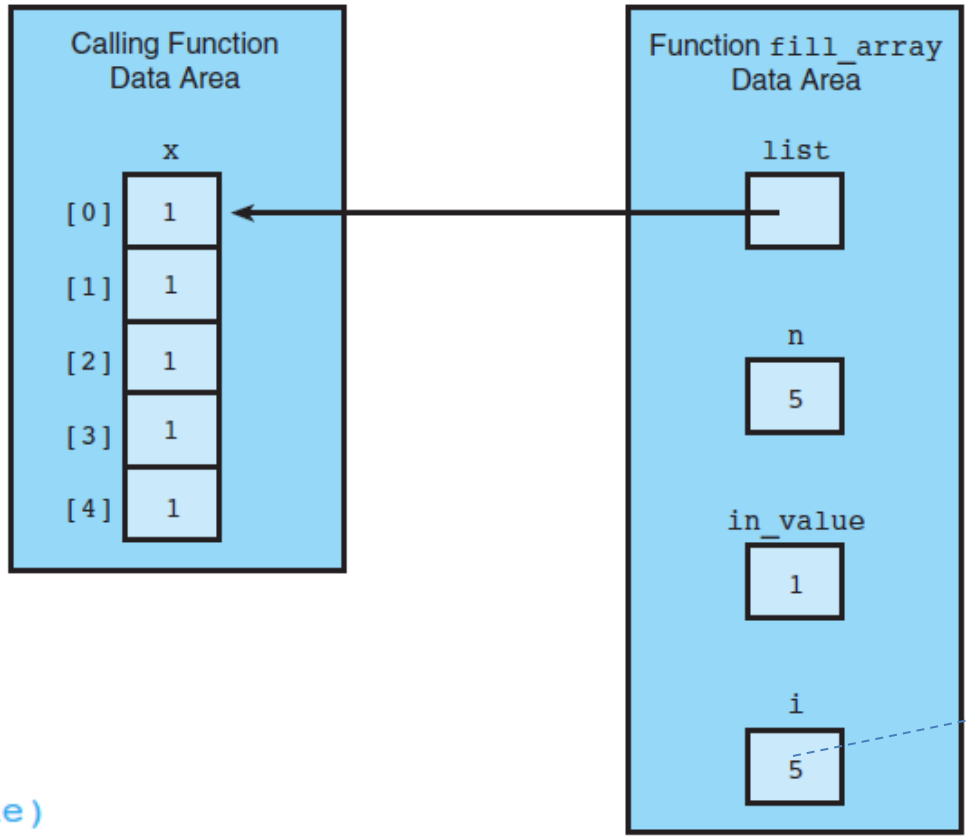
If the array is not to be modified, declare it as **const**

Example 1: Pass by Reference

A function that stores the same value (`in_value`) in all elements of the array corresponding to its formal array parameter list .

```
1.  /*
2.   * Sets all elements of its array parameter to in_value.
3.   * Pre: n and in_value are defined.
4.   * Post: list[i] = in_value, for 0 <= i < n.
5.   */
6.  void
7.  fill_array (int list[],    /* output - list of n integers                */
8.              int n,         /* input - number of list elements            */
9.              int in_value)  /* input - initial value                      */
10. {
11.
12.       int i;              /* array subscript and loop control           */
13.
14.       for  (i = 0; i < n; ++i)
15.           list[i] = in_value;
16. }
```

…Continued




```
6.  void
7.  fill_array (int list[],
8.                int n,
9.                int in_value)
```

The function's local variable

If `x` is an array with five type `int` elements, the function call

`fill_array(x, 5, 1);` stores the value of `1` in the five elements of array `x`.

The **array** x is passed **by reference**, whereas *in_value* and *i* are **passed by value**

Example 2: max of an array

Write a function `get_max` to find the largest value in an array. It uses the variable `list` as an array input parameter.

```
1.  /*
2.   * Returns the largest of the first n values in array list
3.   * Pre: First n elements of array list are defined and n > 0
4.   */
5.  int
6.  get_max(const int list[], /* input - list of n integers                    */
7.          int       n)      /* input - number of list elements to examine   */
8.  {
9.        int i,
10.           cur_large;      /* largest value so far                         */
11.
12.       /* Initial array element is largest so far.                         */
13.       cur_large = list[0];
14.
15.       /* Compare each remaining list element to the largest so far;
16.          save the larger                                                  */
17.       for  (i = 1; i < n; ++i)
18.           if (list[i] > cur_large)
19.                 cur_large = list[i];
20.
21.       return (cur_large);
22. }
```

Function call example:`x_large = get_max(x, 5);`

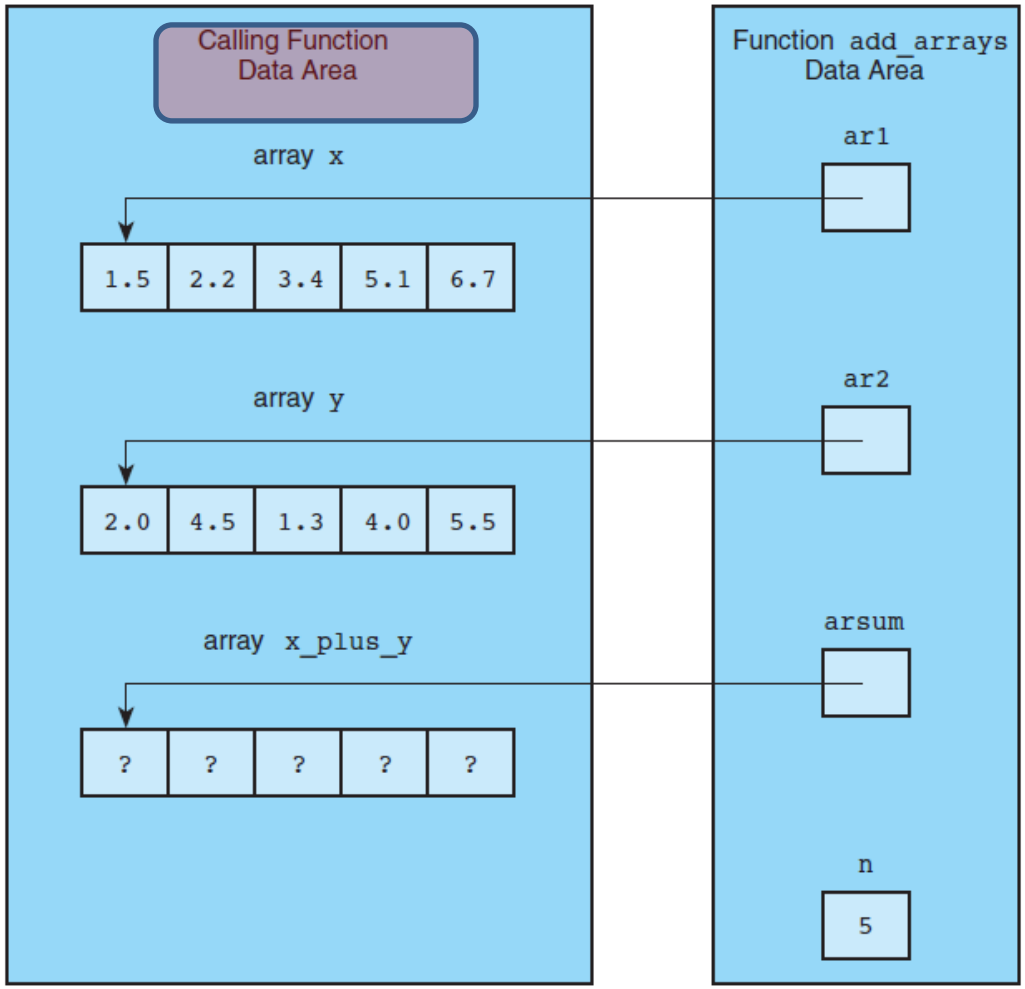
Example 3: return the sum of 2 arrays

Write a function add_arrays to add two arrays.

```
1.  /*
2.   * Adds corresponding elements of arrays ar1 and ar2, storing the result in
3.   * arsum. Processes first n elements only.
4.   * Pre: First n elements of ar1 and ar2 are defined. arsum's corresponding
5.   *      actual argument has a declared size >= n (n >= 0)
6.   */
7.  void
8.  add_arrays(const double ar1[],    /* input -                              */
9.             const double ar2[],    /* arrays being added                   */
10.            double       arsum[],  /* output - sum of corresponding
11.                                       elements of ar1 and ar2             */
12.            int          n)        /* input - number of element
13.                                       pairs summed                        */
14. {
15.       int i;
16.
17.       /* Adds corresponding elements of ar1 and ar2                       */
18.       for  (i = 0; i < n; ++i)
19.           arsum[i] = ar1[i] + ar2[i];
20. }
```

…Continued

```
Example function call:
add_arrays(x, y, x_plus_y, 5);
```




```
void
add_arrays(const double ar1[],     /
           const double ar2[],     /
           double       arsum[],   /

           int          n)         /
```

Formal parameters `ar1 , ar2` , and `arsum` of the function point to the **actual parameter** arrays in the calling functions x, y, and x_plus_y (**pass by reference**) unlike `n` which is **passed by value**.

Example 4: search an array

The function `search` finds a target value in an array

```
1.  #define NOT_FOUND -1    /* Value returned by search function if target not
2.                             found                                          */
3.
4.  /*
5.   * Searches for target item in first n elements of array arr
6.   * Returns index of target or NOT_FOUND
7.   * Pre:  target and first n elements of array arr are defined and n>=0
8.   */
9.  int
10. search(const int arr[],   /* input - array to search                      */
11.        int       target,  /* input - value searched for                  */
12.        int       n)       /* input - number of elements to search        */
13. {
14.       int i,
15.           found = 0,      /* whether or not target has been found        */
16.           where;          /* index where target found or NOT_FOUND       */
17.
18.       /* Compares each element to target                                 */
19.       i = 0;
20.       while (!found && i < n) {
21.           if (arr[i] == target)
22.                 found = 1;
23.           else
24.                 ++i;
25.       }
26.
27.       /* Returns index of element matching target or NOT_FOUND           */
28.       if (found)
29.             where = i;
30.       else
31.             where = NOT_FOUND;
32.
33.       return (where);
34. }
```

If array `ids` is declared in the calling function, the assignment statement:

```
index = search(ids, 4902, ID_SIZE);
```

calls function `search` to search the first `ID_SIZE` elements of array `ids` for the target `ID 4902` . The index of the first occurrence of `4902` is saved in index. If `4902` is not found, then index is set to `-1` .

Example 5: statistical computation using arrays

Write a program that computes the mean and standard deviation of an array of data and displays the difference between each value and the mean.

```
Enter 8 numbers separated by blanks or <return>s
> 16  12  6  8  2.5  12  14  -54.5
The mean is 2.00.
The standard deviation is 21.75.

Table of differences between data values and mean
Index       Item      Difference
  0        16.00          14.00
  1        12.00          10.00
  2         6.00           4.00
  3         8.00           6.00
  4         2.50           0.50
  5        12.00          10.00
  6        14.00          12.00
  7       -54.50         -56.50
```

...Continued

```c
#include <stdio.h>
#include <math.h>

#define MAX_ITEM 8    /* maximum number of items in list of data         */

int
main(void)
{
     double x[MAX_ITEM],    /* data list                                 */
            mean,           /* mean (average) of the data                */
            st_dev,         /* standard deviation of the data            */
            sum,            /* sum of the data                           */
            sum_sqr;        /* sum of the squares of the data            */
     int    i;

       /* Gets the data                                                  */
       printf("Enter %d numbers separated by blanks or <return>s\n> ",
              MAX_ITEM);
       for   (i = 0; i < MAX_ITEM; ++i)
           scanf("%lf", &x[i]);

       /* Computes the sum and the sum of the squares of all data        */
       sum = 0;
       sum_sqr = 0;
       for   (i = 0;   i < MAX_ITEM; ++i) {
             sum += x[i];
             sum_sqr += x[i] * x[i];
       /* Computes and prints the mean and standard deviation            */
       mean = sum / MAX_ITEM;
       st_dev = sqrt(sum_sqr / MAX_ITEM - mean * mean);
       printf("The mean is %.2f.\n", mean);
       printf("The standard deviation is %.2f.\n", st_dev);

       /* Displays the difference between each item and the mean         */
       printf("\nTable of differences between data values and mean\n");
       printf("Index       Item      Difference\n");
       for  (i = 0; i < MAX_ITEM; ++i)
           printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ', x[i] - mean);

       return (0);
}
```

Standard deviation

$$\sqrt{\mathrm{E}[X^2] - (\mathrm{E}[X])^2}$$

IV. 2-D Arrays

- Created by defining two separate element numbers: **number of rows** and **number of columns**
- Declaration

```
type name[numOfRows][numOfColumns];
```

- *Example:* an array of integers with 5 rows and 4 columns

```
int codeTable[5][4];
```

- **Accessing elements:** to access an element of a 2-D array you need to specify both the row and the column indexes

Examples:

```
int x = 2, y = 1, z = 0;
codeTable[1][2] = 128;
codeTable[x][y] = z + codeTable[1][2];
codeTable[3] = 0;    /* Error. This is a 2-D array */
```

2D Array Initialisation

- You can use **either,**

1. Grouping of braces **to initialise each row of elements**

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

myArray[0][1] returns 1

myArray[1][2] returns 3

0	1	2
1	2	3
2	3	4

2. Use **nested loop** to access each element via its pair of indices:

```
for (int x = 0; x <=2; x++) {   // row index
  for (int y = 0; y <=2, y++) //    column index
      myArray[x][y] = ( x + y); // adds values to element
```

- If you're initalising the array elements to 0, `int myArray[3][3] = {{0}};` works too!

2D Array Access

- Use nested loops to iterate through each dimension of the array

```
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
int i, j;

    for ( i = 0; i < 5; i++ ) // row
      for ( j = 0; j < 2; j++ ) // column
          printf("a[%d][%d] = %d\n", i,j, a[i][j] );
```

2D Arrays and Functions

The number of columns must be a constant

- Function prototype

```
double findSmallest( double matrix[][3], int rows);
```

- Function call

```
double data2D[25][3];
.   .   .
minNumber = findSmallest( data2D, 25);
```

This function can process 2D arrays only with 3 columns

- Function definition

```
double findSmallest( double matrix[][3], int rows )
{
  double min = matrix[0][0];
  for(i=0; i < rows; i++)
  for(j=0; j < 3; j++)
     if(min > matrix[i][j]) min = matrix[i][j];
  return min;
}
```

V. Multidimensional Arrays

- Multidimensional arrays are supported where every dimension should be associated with a [SIZE] in your array declaration

- A simple example of a 3D array: considering storing a video.
  - Each frame of the video is a 2D image with intensity values: image (i,j).
  - To have multiple frames, we can use Mat[Height][Witdth][Depth] to store "Depth" frames, where each frame is a 2 D array of dimension Height*Width.

- 4D arrays is an array of 3D arrays… and so forth

Appendix: sizeof() function

- Some of inbuilt functions are used to calculate length of array such as **sizeof()** function
- Example:

```
int arr[] = {1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14};
printf("Num of elements: %d, sizeof(arr)/sizeof(arr[0]);
```

Prints: Number of elements: 11

sizeof(arr) by itself returns size of array in bytes, in this case 44 (int = 4 bytes).

sizeof(arr[0])returns memory size of element. So 44/4 = 11

Example

```
int myArray[5] = {1, 3, 5, 7, 9}, search, c, size;
size = sizeof(myArray)/sizeof(myArray[0]);
printf("Enter value to find \n");
scanf("%d", &search);
for (c = 0; c<size; c++)
   {
   if (myArray[c] == search)
       printf("Its in positions %d", c + 1);
   }
```