

## ARRAYS

Arrays are the simplest and most common type of structured data.

An array contains a group of items that are all the same type, placed contiguously in memory, and that are directly accessed through the use of the array name and index or subscript.

### THE NEED FOR ARRAYS

If you have a collection of similar data elements you may find it inconvenient to give each one a unique variable name.

Consider the situation that requires the temperatures for each day of a week to be read in and stored for processing later on in the program. If each day's temperature had a unique variable name, the input routine could be :

```
#define DAYS_IN_WEEK 7

float Sunday_Temp,
      Monday_Temp,
      Tuesday_Temp,
      Wednesday_Temp,
      Thursday_Temp,
      Friday_Temp,
      Saturday_Temp;

printf("\nPlease enter Sunday's temperature : ");
scanf("%f", &Sunday_Temp);
printf("\nPlease enter Monday's temperature : ");
scanf("%f", &Monday_Temp);
printf("\nPlease enter Tuesday's temperature : ");
scanf("%f", &Tuesday_Temp);
printf("\nPlease enter Wednesday's temperature : ");
scanf("%f", &Wednesday_Temp);
printf("\nPlease enter Thursday's temperature : ");
scanf("%f", &Thursday_Temp);
printf("\nPlease enter Friday's temperature : ");
scanf("%f", &Friday_Temp);
printf("\nPlease enter Saturday's temperature : ");
scanf("%f", &Saturday_Temp);
```

If the average temperature for the week was required, the expression could be :

$$(\text{Sunday\_Temp} + \text{Monday\_Temp} + \dots + \text{Saturday\_Temp}) / \text{DAYS\_IN\_WEEK}$$

This is a rather cumbersome way of implementing the requirement; it is repetitive, time consuming and unwieldy. Imagine using this implementation technique for each day in a month!!

A convenient way to refer to such collections of similar data elements is to use an array.

## USING ARRAYS

The use of arrays provides a way to refer to individual items in a collection by using the same variable name but differing subscripts or indexes. The individual components of the array are called elements i.e. the individual storage locations to hold each occurrence of the data.

### Array Declaration

Like all identifiers, arrays need to be declared so that the compiler will know:

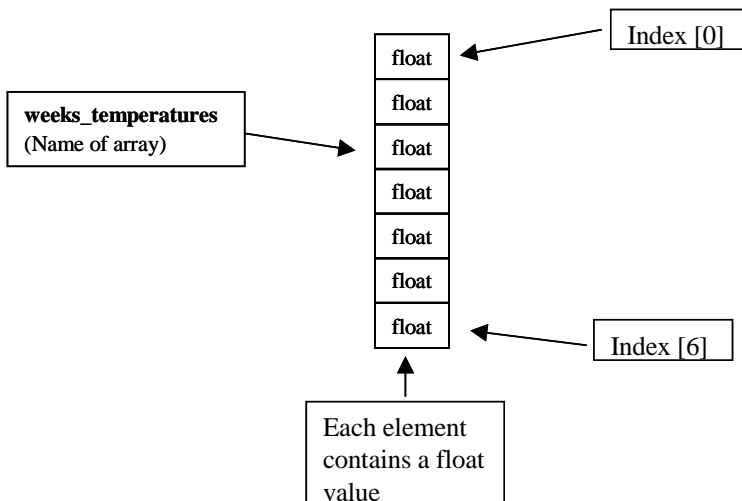
- The name of the array.
- The size of the array (i.e. the number of elements).
- The data type held.

Example :

```
float weeks_temperatures [7];
```

where : The brackets [ ] indicate an array whose name is *weeks\_temperatures*.  
The numerical value 7 is the number of elements in the array; thus it indicates its size.  
The data type held in each element is of type *float*.

The following diagram shows a symbolic representation of this array



### Referring To Individual Elements

Reference to an individual element of an array can be made via the use of the array name followed by the index or subscript value (i.e. the value in square brackets)..

Thus the element of the array with index or subscript 2 (i.e. the 3<sup>rd</sup> element) would be referred to as:

```
weeks_temperatures [2]
```

Variable of data type

- NOTE:** 1. The number in square brackets has a different meaning when referring to an array element than it does when declaring the array when the number specifies the size of an array
2. The index or subscript value can be a countable variable (e.g. integer or character) if required.

### Entering Data Into The Array

Data is entered into the array via reference to the array name and index or subscript number.

For example the following section of code would place the data in the array weeks\_temperatures :

```

/***** NAMED CONSTANT *****/
#define DAYS_IN_WEEK 7

/***** GLOBAL VARIABLES *****/
float weeks_temperatures [DAYS_IN_WEEK];

int day; /*Loop control for days in the week*/

/***** MAIN FUNCTION *****/
void main (void)
{
    Input_Temps ( );
}
/***** OTHER FUNCTIONS *****/
/**** INPUT TEMPERATURES FOR WEEK FUNCTION *****/
void Input_Temps (void)
{
    for (day = 1: day <= DAYS_IN_WEEK; day++ )
    {
        printf("\nPlease enter temperature for day %d : ", day);
        scanf("%f", &weeks_temperatures [day - 1] );
    }
}

```

### Reading Data From An Array

As for entering the data into the array- via the array name and index or subscript value.

For example, the following function will calculate the average the week's temperature.

```

void Average_Temp (void)
{
    total = 0;

    for (day = 1: day <= DAYS_IN_WEEK; day++ )
    {
        total = total + weeks_temperatures [day - 1] ;
    }
    average = total / DAYS_IN_WEEK;
    printf("\nAverage temperature for week is %5.2f", average);
}

```

## Declaring An Array With An ‘Unknown Size’

This can be simply addressed by declaring the size to be ample for the requirements using typical declarations as before.

A better solution is dynamic allocation when the actual size of the array is defined at run time i.e. when the program actually runs. This can be set-up using pointers and the routine malloc ( ).

## Bounds Checking

In C there are no checks on the bounds of the array i.e. the range of the index or subscript values. In other words, it is possible to write data into a location outside the array. This can lead to an unpredictable result, to say the least, and there will be no error message from the system to indicate that this has happened.

Thus it is imperative that the programmer does not allow this to happen by implementing checks on the array bounds in the program.

## Initialising Arrays

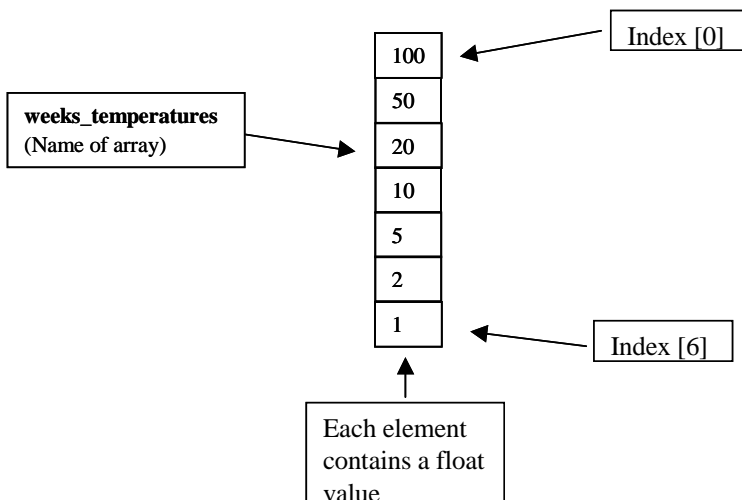
This is analogous to initialising a simple variable such that the program is compiled with specific values in the array

Consider a requirement to calculate the number of coins required in the change given in pounds i.e. the operator types in the change required in pounds and the program replies with the required makeup of the change; always giving the maximum highest coin value.

The coins value required in this program could be held in a preset array as a number of pennies.

```
#define LIMIT 7  
  
int change_table[LIMIT] = { 100, 50, 20, 10, 5, 2, 1 };
```

where the list of values is enclosed by braces, and the values are separated by commas. The values are assigned in turn to the elements of the array as follows:



## STORAGE CLASSES AND ARRAY INITIALISATION

### ARRAY SIZE AND INITIALISATION

C will allow the size of the array to be omitted by the programmer. For example :

```
int change_table[] = { 100, 50, 20, 10, 5, 2, 1 };
```

This is possible because the compiler will count the number of values in the initialisation list and fix that as the array size.

But what happens if you state the array size but supply less initialisation values ?  
e.g.

```
float temperature_bands[3] = { 26, 20 };
```

In this case, the extra elements will be initialised to a value of zero.

As you would expect, the compiler will complain if the number of values in the initialisation list is more than the declared size of the array e.g.

```
int error[5] = { 1, 2, 3, 4, 5, 6 };
```

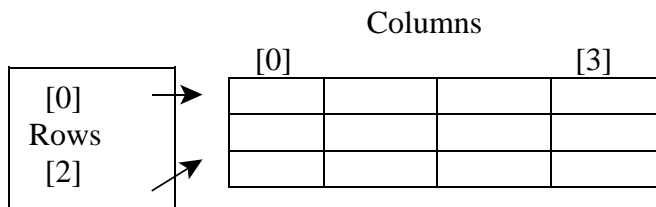
### MORE THAN ONE DIMENSION

The arrays considered so far have had only one dimension. It's also possible for them to have two or more dimensions. This allows them to model or emulate multidimensional objects, such as graph paper with rows and columns, or the computer display screen itself.

Only two dimensional arrays will be considered.

### Two Dimensional Arrays

These can be considered as a matrix with rows and columns. A schematic representation of a 3 by 4 matrix could be :



If the array is to hold real numbers, a typical declaration is :

```
#define MAX_ROWS 3
#define MAX_COLUMNS 4

float example_array [MAX_ROWS] [MAX_COLUMNS];
```

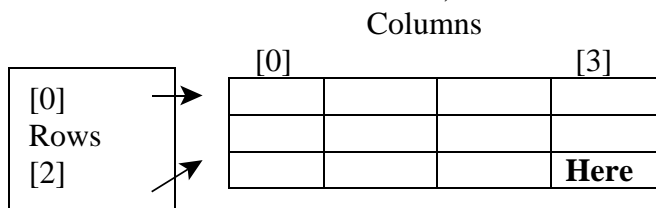
Notice that this array now has two subscripts:  
 The first is the row number and  
 The second is the column number..

In this example the row index has a range of 0 to 2, with the column index range being 0 to 3.

### Referring To Individual Elements

Reference to any individual element of the array can be made by the use of both the required index or subscript values.

Thus access to the element of row 2, column 3 is :



example\_array [2][3]

### Initialising Two-Dimensional Arrays

This is a similar concept to that of initialising a one-dimensional array.

Imagine that a 4 by 4 hexadecimal keypad looks like :

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>
<b>8</b>	<b>9</b>	<b>A</b>	<b>B</b>
<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>

A two-dimensional array could be initialised to hold the ASCII values for these keys as follows:

```
#define MAX_ROWS 4
#define MAX_COLUMNS 4

char keypad [MAX_ROWS] [MAX_COLUMNS] =
  { { '0', '1', '2', '3' },
    { '4', '5', '6', '7' },
    { '8', '9', 'A', 'B' },
    { 'C', 'D', 'E', 'F' } };
```