Declaration of Arrays

```
type name[size];
```

Accessing the Elements of an Array

/* Set each element of the array to -1 */

```
int i, marks[10];
for(i = 0; i < 10; i++) {
      marks[i] = -1;
}
```

Calculating th Address of Array Elements

/* Address of data element*/

```
A[k] = BA(A) + w(k - lower_bound);
```

where `A` is the array, `k` is the index of the element that we want to calculate. `BA` is the **base address** of the array `A`, and `w` is the size of one element in memory.

Example

Given an array `int marks[] = { 99, 67, 78, 56, 88, 90, 34, 85 }`, calcuate the address of `marks[4]` if the `base address = 1000`.

$$\begin{aligned} \texttt{marks}[4] &= 1000 + 2(4 - 0) \\ &= 1000 + 2(4) \\ &= 1008 \end{aligned}$$

Calculating the Length of an Array

```
Length = upper_bound - lower_bound + 1
```

where `upper_bound` is the index of the last element and `lower_bound` is the index of the first element in the array.

Initializing Arrays during Declaration

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

In C we write

```
int marks[5] = { 90, 82, 78, 95, 88 };
```

where `list of values` is a comma separated list.

Inputting Values from the Keyboard

```
int i, marks[10];
for( i = 0; i < 10; i++ ) {
scanf("%d", &marks[i]);
}
```

Assigning Values to Individual Elements

/* Code to copy an array at the individual element level */

```
int i, array1[10], array2[10];
array1[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for( i = 0; i < 10; i++ ) {
       array2[i] = array1[i];
}
```

/* Code for filling an array with even numbers */

```
int i, array[10];
for( i = 0; i < 10; i++ ) {
       array[i] = i * 2;
}
```

Passing Arrays To Functions

Passing Data Values

/* Passes value of individual array element to a function*/

```
int main() {
    int arr[5] = { 1, 2, 3, 4, 5 };
    fun(arr[3]);
}
```

/* Called function */

```
void func(int num) {
     printf("%d", num);
}
```

Passing Addresses

/* Passes address of individual array element to a function*/

```
int main() {
    int arr[5] = { 1, 2, 3, 4, 5 };
    fun(&arr[3]);
}
```

/* Called function */

```
void func(int num) {
     printf("%d", *num);
}
```

Passing the Entire Array

/* Passes entire array to a function */

```
int main() {
    int arr[5] = { 1, 2, 3, 4, 5 };
    fun(arr);
}
```

/* Called function */

```
void func(int arr[5]) {
     int i;
     for(i = 0; i < 5; i++) {
           printf("%d", arr[i]);
}
```

a function that accepts an array can declare the formal parameter in either of the two following ways

`func(int arr[]);` or

`func(int *arr);`

a function that accepts an array as a parameter, the declaration should look like this

`func(int arr[], int n);` or

`func(int *arr, int n);`

Declaring Two-dimensional Arrays

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

Calculating the Address of a 2-Dimensional Array

/* Column major order */

```
Address(A[I][J]) = Base_Address + w{ M (J - 1) + (I - 1) }
```

/* Row major order */

```
Address(A[I][J]) = Base_Address + w{ N (I - 1) + (J - 1) }
```

where `w` is the number of bytes required to store one element, `N` is the number of columns, `M` is the number of rows, and `I` and `J` are subscripts of the array element.

Example

Consider a 20 × 5 two-dimensional array `marks` which has its `base address = 1000` and size of an element = 2. Now compute the address of the element, `marks[18][4]` assuming that the elements are stored in row major order.

$$\begin{aligned}
\texttt{Address(A[I][J])} &= Base\_Address + w\{N(I-1)+(J-1)\}\\
\texttt{Address(marks[18][4])} &= 1000 + 2\{5(18-1)+(4-1)\}\\
&= 1000 + 2\{5(17)+(4-1)\}\\
&= 1000 + 2\{5(17)+(3)\}\\
&= 1000 + 2(88)\}\\
&= 1000 + 176\\
&= 1176
\end{aligned}$$

Passing Two-Dimensional Arrays To Functions

Passing a Row

/* Passing a row of a 2D array to a function*/

```
int main() {
    int arr[2][3] = ({ 1, 2, 3 }, {4, 5, 6 });
    func(arr[1]);
}
```

/* Called Function*/

```
void func(int arr[]) {
     int i;
     for(i = 0; i < 3; i++)
        printf("%d" arr[i] * 10);
}
```

Pointers and Three-Dimensional Arrays

/* Declaring a pointer to aone-dimmensional array*/

```
int arr[] = { 1, 2, 3, 4, 5 };
int *parr;
parr = arr;
```

/* Declaring a pointer to a two-dimmensional array*/

```
int arr[2][2] = {{ 1, 2 }, { 3, 4 }};
int (*parr)[2];
parr = arr;
```

/* Declaring a pointer to a three-dimmensional array*/

```
int arr[2][2][2] = { 1, 2, 3, 4, 5, 6, 7, 8 };
int (*parr)[2][2];
parr = arr;
```

we can access an element of a three-dimensional array by writing

```
arr[i][j][k] = *(*(*(arr + i) + j + k)
```

Points To Remember

- An array is a collection of elements of the same data type.

- The elements of an array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).

- The index specifies an offset from the beginning of the array to the element being referenced.

- Declaring an array means specifying three parameters: data type, name, and its size.

- The length of an array is given by the number of elements stored in it.

- There is no single function that can operate on all the elements of an array. To access all the elements, we must use a loop.

- The name of an array is a symbolic reference to the address of the first byte of the array. Therefore, whenever we use the array name, we are actually referring to the first byte of that array.

- C considers a two-dimensional array as an array of one-dimensional arrays.

- A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second subscript denotes the column of the array.

- Using two-dimensional arrays, we can perform the different operations on matrices: transpose, addition, subtraction, multiplication.

- A multi-dimensional array is an array of arrays. Like we have one index in a one-dimensional array, two indices in a two-dimensional array, in the same way we have n indices in an n-dimensional or multi-dimensional array. Conversely, an n-dimensional array is specified using n indices.

- Multi-dimensional arrays can be stored in either row major order or column major order.

- Sparse matrix is a matrix that has large number of elements with a zero value.

- There are two types of sparse matrices. In the first type, all the elements above the main diagonal have a zero value. This type of sparse matrix is called a lower-triangular matrix. In the second type, all the elements below the main diagonal have a zero value. This type of sparse matrix is called an upper-triangular matrix.

- There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of sparse matrix is called a tridiagonal matrix.