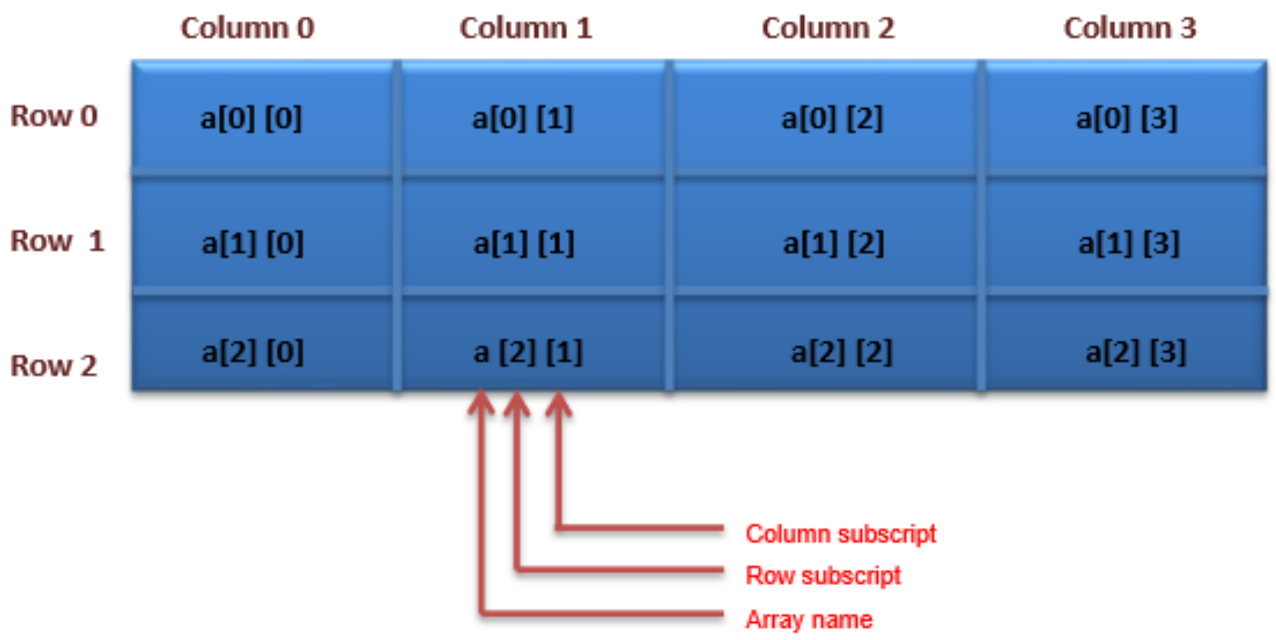
MULTI-DIMENSIONAL ARRAYS

Arrays with two dimensions (i.e. subscripts) often represent tables of values of information arranged in rows and columns.

- The identity a particular table element, we must specify two subscripts.
- By convention, the first identifies the element's row and the second identifies the element's column.
- Arrays that require two subscripts to identify a particular element are called two-dimensional arrays or 2-D arrays.
- Arrays with two or more dimensions are known as multidimensional arrays and can have more than two dimensions.

The following figure illustrates a two-dimensional array, **a.** The array contains three rows and four columns, so it is a 3-by-4 array. In general, an array with **m** rows and **n** columns is called an **m-by-n array**.



Every element in array **a** is identified by an element name of the form **a[i][j]**. **a** is the name of the array and **i** and **j** are the subscripts that uniquely identify each element in **a.**

EXAMPLE

- Lines a-c declare thee arrays, each with two rows and three columns.

- The declaration of array1 (line a) provides six initialisers in the two sub lists. The first sub list initialises row 0 of the array to the values 1, 2 and 3, the second sub list initialises row 1 of the array to the values 4, 5 and 6.

- If the braces around each sub-list are removed from the array1 initialiser list, the compiler initialises the elements of row 0 followed by the elements of row 1, yielding the same result.

- The declaration of array2 (line b) provides only five initialisers.

- The initialisers are assigned to row 0, then row 1. Any elements that do not have an explicit initialiser are initialised to zero, so array2[1][2] is initialised to zero.

- The declaration of array3 (line c) provides three initialisers in two sub lists.

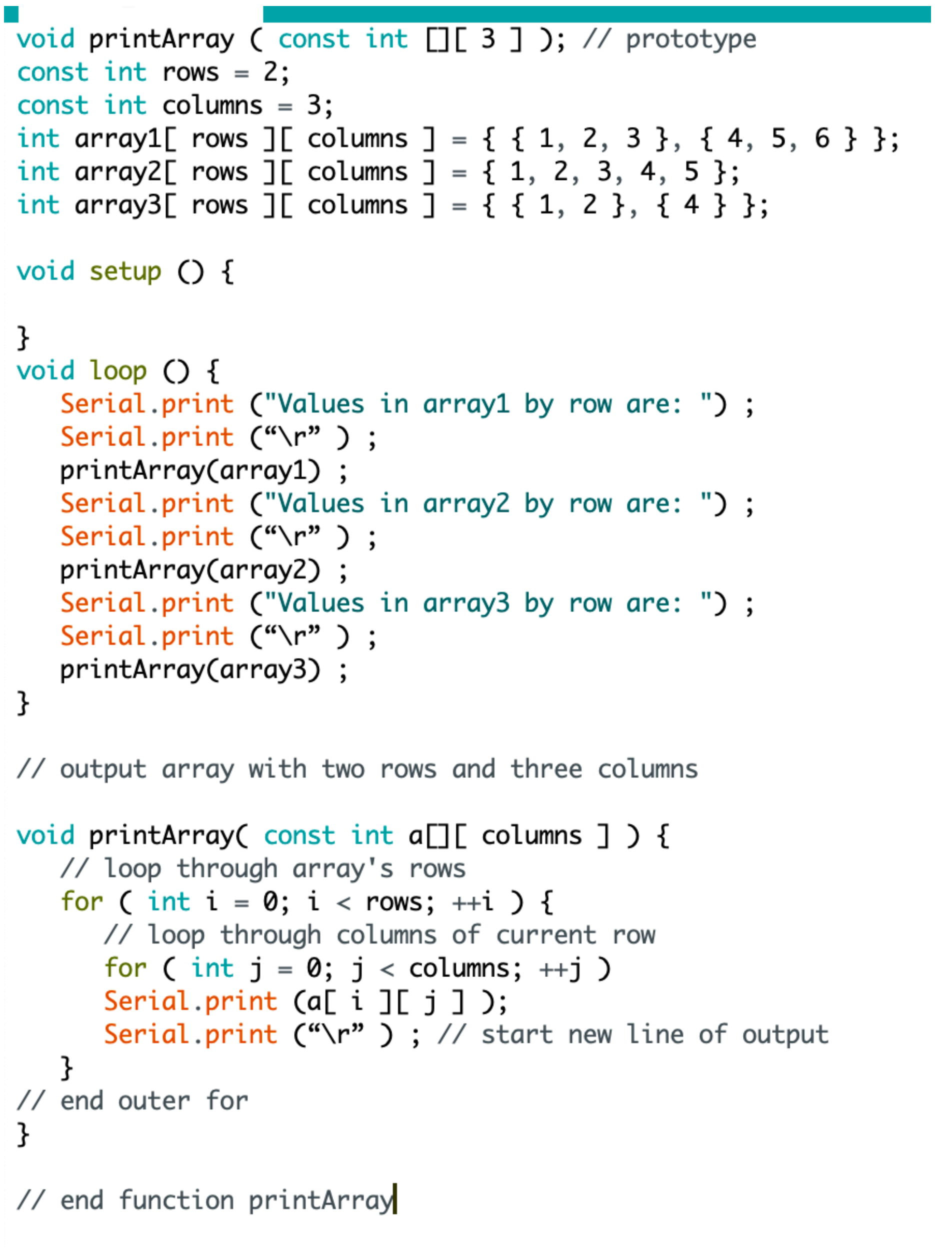
- The sub list for row 0 explicitly initialises the first two elements of row 0 to 1 and 2, the third element is simplicity initialised to zero.

- The program calls function printArray to output each array's elements. Notice that the function prototype (line k) specify the parameter **const int a[ ][columns]**.

- When a function receives a one-dimensional array as an argument, the array brackets are empty in the functions parameter.

- The size of two-dimensional arrays; s first dimension (i.e., the number of rows) is not required either, but all the subsequent dimension sizes are required. The compiler uses these sizes to determine the locations in memory of elements in multidimensional arrays.

- All array elements are stored consecutively in memory, regardless of the number of dimensions. In a two-dimensional array, row 0 is stored in memory followed by row 1.



```
void printArray ( const int [][ 3 ] ); // prototype
const int rows = 2;
const int columns = 3;
int array1[ rows ][ columns ] = { { 1, 2, 3 }, { 4, 5, 6 } };
int array2[ rows ][ columns ] = { 1, 2, 3, 4, 5 };
int array3[ rows ][ columns ] = { { 1, 2 }, { 4 } };

void setup () {

}
void loop () {
   Serial.print ("Values in array1 by row are: ") ;
   Serial.print (“\r” ) ;
   printArray(array1) ;
   Serial.print ("Values in array2 by row are: ") ;
   Serial.print (“\r” ) ;
   printArray(array2) ;
   Serial.print ("Values in array3 by row are: ") ;
   Serial.print (“\r” ) ;
   printArray(array3) ;
}

// output array with two rows and three columns

void printArray( const int a[][ columns ] ) {
   // loop through array's rows
   for ( int i = 0; i < rows; ++i ) {
      // loop through columns of current row
      for ( int j = 0; j < columns; ++j )
      Serial.print (a[ i ][ j ] );
      Serial.print (“\r” ) ; // start new line of output
   }
// end outer for
}

// end function printArray
```

<u>RESULT</u>

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

NOTE - Each row is a one-dimensional array. To locate an element in a particular row, the function must know exactly how many elements are in each row so it can skip the proper number of memory locations when accessing the array. Thus, when accessing a[1][2], the function knows to skip row 0's three elements in memory to get to row 1. Then, the function accesses element 2 of that row. Many common array manipulations use FOR statements.

EXAMPLE

The following **FOR** statement sets all the elements in row 2 of array a.

```
for ( int column = 0; column < 4; ++column )
        a[ 2 ][ column ] = 0;
```

The **FOR** statement varies only the second subscript (i.e., the column subscript). The preceding **FOR** statement is equivalent to the following assignment statements -

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

The following **Nested FOR** statement determines the total of all the elements in array a -

```
total = 0;
for ( int row = 0; row < 3; ++row )
for ( int column = 0; column < 4; ++column )
total += a[ row ][ column ];
```