1 Overview

Arrays are data structures that store sets of data items under a single name declaration. Each element of the array must be of the same type. The elements can be accessed via an index (a non-negative integer).

2 Defining Arrays

An array is declared like any other variable but included is the number of items in the array. Each element, is accessed by an index (or subscript), beginning with the zero-ith element. So, for example, if you need to store a set of 5 integer grades, you can declare an array like the following:

```c
int grades[5];
```

where the array will have from 0-4 elements: grades[0], grades[1], grades[2], grades[3], & grades[4] – each one containing an integer. If you need to access the 3rd item, you would access grades[2] or grades[n-1] where n is the total number of elements in the array.

An individual array element can be used anywhere that a normal variable can be used. For example, if you want to assign the 3rd element the value of 85, then you would write

```c
grades[2] = 85;
```

Or, if you have an integer variable called g and you want to assign to it the value of the 5th element, you would write

```c
g = grades[4];
```

which then gives g the value of whatever that 5th element is in the array.

This construct makes it much easier for handling large data sets. With a for loop, for example, you can easily access each item in a large array. If you’re trying to find the lowest one, or add up all the values, it is much easier to implement using an array. The following loop will sum up all the values in an array of 100 items:

```c
for ( i = 0; i < 100; ++i)
    sum += grades[i];
```

Remember to start with zero or you will miss the first item.

3 Initializing Arrays

The elements in an array can be initialized in the following way:

```c
int counters[5] = { 0, 0, 0, 1, 5 };
```


An array of characters can be initialized in a similar manner:

```c
```

It’s not necessary to completely initialize the entire array. If the first few are initialized, the remaining are set to zero (0). Consider the following:

```c
float sample_data[500] = { 100.0, 300.0, 500.5 };
```
The first 3 elements are initialized to the values provided; the remaining 497 elements will be set to zero (0). If all the elements need to be initialized to something other than zero (0), the best way to do that is to use a for loop.

```c
int array_values[10];
int i;

for ( i = 0; i < 10; ++i )
array_values[i] = i * i;
```

This for loop initializes each element to the square of the element number (subscript number), so the array will contain 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 after initialization.

C allows you to define an array without specifying the number of elements, which works fine as long as you initialize every element of the array when it is defined.

```c
char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

will implicitly dimension the array to 6 elements.

You can also use the index numbers when initializing. So if you know specific elements need to be initialized to a certain value, you can do the following:

```c
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

Because the largest index number specified above is 99, sample_data will be set to contain 100 elements, the remaining elements initialized to zero (0).

### 4 Multidimensional Arrays

The C language allows arrays of any dimension to be defined. For example, here is a 2-dimensional array, a matrix:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>5</td>
<td>-3</td>
<td>17</td>
<td>82</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>-7</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>32</td>
<td>1</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

In this matrix, \( M[i][j] \) refers to the element in the \( i \)th row and \( j \)th column. So, \( M[2][1] \) refers to the element with the value of 32; \( M[0][4] \) refers to the element with the value of 82; \( M[2][3] \) refers to the element with the value 0, and so on.

The statement `int sum = M[0][2] + M[2][4];` would result in `sum` being equal to 11 (-3 + 14).

Two-dimensional arrays are declared the same way as one-dimensional arrays. So, \( int M[4][5]; \) would declare a 2-dimensional array consisting of 4 rows and 5 columns, for a total of 20 elements. Each position in the array is defined to contain an integer value.

Initializing a 2-dimensional array, like the one above, can be done like the following:

```c
int M[4][5] = {
    { 10,  5, -3, 17, 82 },
    { 9,   0,  0,  8, -7 },
    { 32, 20,  1,  0, 14 },
    { 0,   0,  8,  7,  6 }
};
```
or, it could have been initialized this way:

```c
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, 7, -7, 32, 20, 1, 0, 14, 0, 0, 8, 7, 6 }
```

As with 1-dimensional arrays, it is not required that the entire array be initialized. A statement such as the following only initializes the first 3 elements of each row; the remaining values are initialized to zero (0) (the inner pairs of braces are required in this case or else these values would fill up the first couple two rows + the first 2 values of the 3rd row, and the remaining elements would be set to zero (0)).

```c
int M[4][5] = {
  { 10,  5, -3 },
  { 9,  0,  0 },
  { 32, 20,  1 },
  {  0,  0,  8 }
};
```

Subscripts can also be used to initialize specific elements (the remaining are set to zero (0)):

```c
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

### 5 Variable-length Arrays

The C language allows you to declare arrays of a variable size. The following example prompts the user for the number of values, and the that number is used to declare an array of that length.

```c
printf (" How many Fibonacci numbers do you want ( between 1 and 75)? ");
scanf ("%i", &numFibs);
unsigned long long int Fibonacci[numFibs];
```

This is called a variable length array because the size is specified by a variable and not by a constant expression.

### 6 Functions & Arrays (chapter 8)

As with ordinary variables, it is also possible to pass the value of an array element, or an entire array, as an argument to a function.

The following is an example of passing an array element:

```c
sq_root_result = squareRoot (averages[i]);
```

where the i'th element of the array called averages is passed to the squareRoot function and the result is assigned to the variable called sq_root_result.

Passing an entire array as an argument is different – only the array name is required without any subscripts. For example, if you have an array called gradeScores that has been declared to have 100 elements:

```c
int gradeScores [100];
```

it may be passed as an argument to a function:

```c
minimumGrade = minimum (gradeScores);
```
and the function to which it is passed must be expecting an entire array to be passed as an argument. So the `minimum` function might look like the following:

```c
int minimum ( int values[100] )
{
    ...
    return minValue;
}
```

Unlike other arguments passed to functions, when an entire array is passed as an argument to a function, any changes made to the formal parameter array by the function are actually made to the original array passed to the function. Therefore, when the function returns, these changes still remain in effect. (When an array is passed as an argument, the function gets passed information describing where in the computer’s memory the array is located.) This only applies to entire arrays, not individual elements of an array, whose values are copied into the formal parameters and therefore cannot be permanently changed by the function.

With multi-dimensional arrays, the same applies – you can pass the entire array by just using the name of the array as the argument, and any changes made to the formal parameter array inside the function makes permanent changes to the original array.

When declaring a 2-dimensional array, the number of rows in the array can be omitted, but the declaration must contain the number of columns in the array. So the following declarations are valid:

```c
int array_values [100][50];
int array_values2 [ ][50];
```

The following 2 declarations are not valid:

```c
int array_values [100][ ];
int array_values2 [ ][ ];
```

You can also have functions that accept variable-length multi-dimensional arrays as arguments. For example:

```c
void displayMatrix ( int nRows, int nCols, int matrix[nRows][nCols] )
{
    ...
}
```

7 Sorting Arrays

Sorting an array will further illustrate how the values of the array are changed when the entire array is passed as an argument to a function.

Sorting algorithms have always received much attention from computer scientists because it is an operation that is so commonly performed. There are many sophisticated algorithms that sort in the least amount of time using as little of memory as possible.

Program 8.12 in the book illustrates a straightforward way to sort the values in an array in ascending order. The algorithm for a sort of array `a` with `n` elements follows:

**Step 1:** set `i` to 0
**Step 2:** set `j` to `i + 1`
**Step 3:** if `a[i] > a[j]`, exchange their values
**Step 4:** set `j` to `j + 1`; if `j < n`, go to step 3
**Step 5:** set `i` to `i + 1`; if `i < n-1`, go to Step 2
**Step 6:** `a` is now sorted in ascending order