Arrays in C - part 1

An array is a fixed number of data items that are all of the same type.

The data items in an array are referred to as elements. The elements in an array are all of type int, or of type long, or all of any type you choose.

Array declaration:

The array declaration is similar to a declaration for a normal variable that contains a single value, except that you've placed a number between square brackets [] following the name.

The syntax for declaring an array in C is as follows:

```
datatype arrayName[arraySize];
```

- **datatype**: The data type of the elements that the array will hold.
- **arrayName**: The name of the array.
- **arraySize**: The number of elements in the array. It must be a constant expression (literal or defined constant).

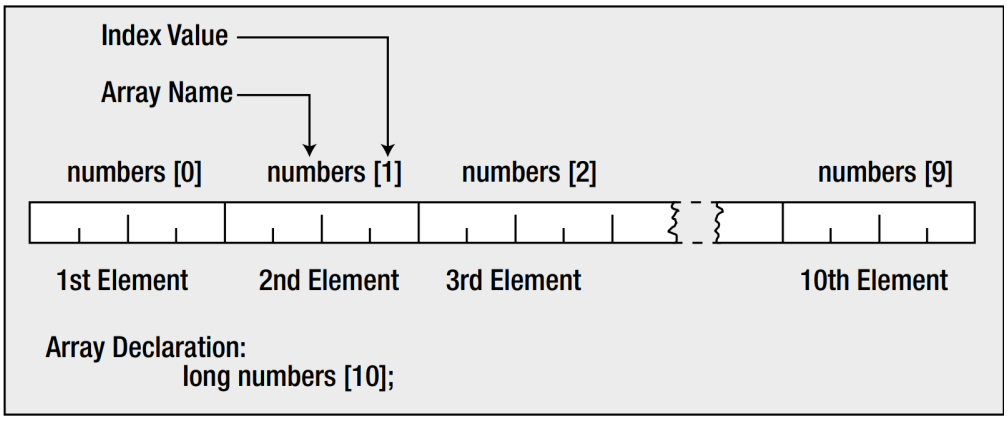
Here is an example of array declaration:

```
long numbers[10];
```

Accessing Array:

Each of the data items stored in an array is accessed by the same name; in the previous statement the array name is numbers.

You select a particular element by using an **index value** between square brackets following the array name. **Index values are sequential integers that start from zero, and 0 is the index value for the first array element**. The index values for the elements in the numbers array run from 0 to 9, so the index value 0 refers to the first element and the index value 9 refers to the last element. Therefore, you access the elements in the numbers array as numbers[0], numbers[1], numbers[2], and so on, up to numbers[9]. You can see in figure below.



You can specify an index for an array element by an expression in the square brackets following the array name. The expression must result in an integer value that corresponds to one of the possible index values.

When you use an expression, the only constraints are that it must produce an integer result, and the result must be a legal index value for the array.

Here is an example to calculate averages with arrays:

```c
#include <stdio.h>

int main(void)
{
    int grades[10];                     // Array storing 10 values
    unsigned int count = 10;            // Number of values to be read
    long sum = 0L;                      // Sum of the numbers
    float average = 0.0f;               // Average of the numbers
    printf("\nEnter the 10 grades:\n"); // Prompt for the input
    // Read the ten numbers to be averaged
    for (unsigned int i = 0; i < count; ++i)
    {
        printf("%2u> ", i + 1);
        scanf("%d", &grades[i]); // Read a grade
        sum += grades[i];        // Add it to sum
    }
    average = (float)sum / count; // Calculate the average
    printf("\nAverage of the ten grades entered is: %.2f\n", average);
    return 0;
}
```

output:

```
Enter the 10 grades:
 1> 75
 2> 85
 3> 84
 4> 82
 5> 65
 6> 95
```

```
 7> 89
 8> 86
 9> 79
10> 78

Average of the ten grades entered is: 81.80
```

In this example:

- Read the values in to the array
- Add the array elements
- calculate average and print the result.

Initializing an Array:

To initialize the elements of an array, you **just specify the list of initial values between braces and separate them by commas** in the declaration.

```
double values[5] = { 1.5, 2.5, 3.5, 4.5, 5.5 };
```

This declares the values array with five elements. The elements are initialized with values[0] having the value 1.5, value[1] having the initial value 2.5, and so on.

To initialize the whole array, there must be one value for each element. **If there are fewer initializing values than elements, the elements without initializing values will be set to 0**. Thus, if you write:

```
double values[5] = { 1.5, 2.5, 3.5 };
```

the first three elements will be initialized with the values between braces, and the last two elements will be initialized with 0.

Knowing that the compiler will supply zeroes for elements for which you don't provide, an initial value offers an easy way **to initialize an entire array to zero**. You just need to **supply one element with the value of 0**:

```
double values[5] = {0.0};
```

The entire array will then be initialized with 0.0.

If you put more initializing values than there are array elements, you'll get an error message from the compiler. However, you can omit the size of the array when you specify a list of initial values. In this case, the compiler will assume that the number of elements is the number of values in the list:

```
int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

The size of the array is determined by the number of initial values in the list, so the primes array will have ten elements.

Example below illustrates array Initialization:

```c
#include <stdio.h>

int main() {
    // Initializing an array of integers at the time of declaration
    int numbers1[5] = {1, 2, 3, 4, 5};

    // Initializing an array of integers with 0
    int numbers2[5] = {0};

    // Initializing an array of characters at the time of declaration
    char numbers3[] = {10, 20, 30, 40, 50, 60, 70};

    // Display the initialized arrays
    printf("Initialized Array 'numbers1': ");
    for (int i = 0; i < 5; ++i) {
        printf("%d ", numbers1[i]);
    }

    printf(" \nInitialized Array 'numbers2': ");
    for (int i = 0; i < 5; ++i) {
        printf("%d ", numbers2[i]);
    }

    printf(" \nInitialized Array 'numbers3': ");
    for (int i = 0; i < sizeof(numbers3); ++i) {
        printf("%d ", numbers3[i]);
    }
    return 0;
}
```

output:

```
Initialized Array 'numbers1': 1 2 3 4 5
Initialized Array 'numbers2': 0 0 0 0 0
Initialized Array 'numbers3': 10 20 30 40 50 60 70
```

array out-of-bounds access:

Note that if you use an expression for an index value that's outside the legal range for the array, the program won't work properly. The compiler can't check for this, so your program will still compile. You'll pick up a junk value from somewhere so that the results are incorrect and may leads to runtime errors.

It is therefore most important to ensure that your array indexes are always within bounds.

Here's an example illustrating array out-of-bounds access:

```c
#include <stdio.h>

int main() {
    int numbers[5] = {10, 20, 30, 40, 50};

    // Attempt to access an element outside the valid range
    int value = numbers[6];

    // Print the accessed value (this behavior is undefined)
    printf("Value at index 6: %d\n", value);

    return 0;
}
```

output:

```
Value at index 6: 4194432
```

You can see that value at index 6 is garbage value(4194432).