L5 - Arrays

December 14, 2019

1 Arrays

An array allows us to represent many different variables using a single identifier.

Upon creating an array, we need to decide what datatype the array contains. All elements in the array must have the same type. Individual array elements are accessed using the array's name and the position of the element in the array (its *index*).

Indices range from **0** to **length - 1**: for example, an array of size 10 has indices ranging from 0 to 9. In C, an index greater than the capacity will *access memory outside the array* - which is usually a bug, and will **not cause a compiler error in C!**

```
In [ ]: // don't run this
        #include <stdio.h>
        #include <stdlib.h>

        int main(){
            int itemsList[4];

            for ( i = 0; i < 4; i++ ){
            itemsList[i] = itemsList[i+1];
            }
        }
```

This is wrong, because the for loop must stop at **3**, not 4. This is a very common error.

As in the above code, an array index can be an *expression*, not just a literal integer. This can even be a function call, so long as the function returns an integer.

```
In [ ]: // don't run this
        #include <stdio.h>
        #include <stdlib.h>

        int main(){
            int i = 3;
            samples[i] = 5.2; // initialize fourth element

            i = 1;
            samples[i]
        }
```

A very common task is to iterate through the first n elements in an array, doing something with each element.

In [16]:
```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    double samples[100];

    srand(3); // random seed

    for (int i = 0; i < 100; i++){
        samples[i] = rand() % 100 + 1; // "random" number generator
    }

    // calculate the average
    double sum;
    double average;

    for (int k = 0; k < 100; k++){
        sum = sum + samples[k];
    }

    average = sum / 100;

    printf("%3.4f\n", average);

    return EXIT_SUCCESS;
}
```

```
53.2600
```

We can take that average feature out and plonk it into a function. Note that if we are passing an array into a function, we will pass the array *by reference*, not by *value*. This means that if we modify the array inside the function's *local scope*, the values in the function *will be modified in the program's global scope as well*.

In [15]:
```c
#include <stdio.h>
#include <stdlib.h>

double average_samples( double x[], int n ){
    double sum = 0;
    for ( int i = 0; i < n; i++ ){
        sum  = sum + x[i];
    }
    return sum / n; // note we don't need a variable for the average
}
```

```
int main(){
    double samples[100];

    srand(3); // random seed

    for (int i = 0; i < 100; i++){
        samples[i] = rand() % 100 + 1; // "random" number generator
    }
    printf("%f\n", average_samples(samples,100));
    return EXIT_SUCCESS;
}
```

```
53.260000
```

Notice that we do not need to specify the array's capacity in the function declaration; the function will apply equally to all possible arrays of doubles. We give the function the number of elements it should process as a second integer parameter, n, as there is **no way to determine how many elements in the array have been initialized.**

Note that we can pass a single array element by using the subscript:

```
In [ ]: // don't run this
        result = myFunction( samples[i] );
```

When we do this; this element is now passed into `myFunction` **by value** and `myFunction` will **not** change the value of `samples[i]` in the global scope!

1.1 Functions Changing Arrays

```
In [18]: #include <stdio.h>
         #include <stdlib.h>

         void init_array( int arr[], int n, int initial ){
             for ( int i = 0; i < n; i++ ){
                 arr[i] = initial;
             }
         }

         int main(){

             int numbers[10];
             init_array(numbers, 5, 0);

             return EXIT_SUCCESS;
         }
```

1.2 Calculating Array Capacity

The C function `sizeof()` returns the amount of memory, in bytes, used for an object in memory. The following code calculates the capacity of an array by comparing the total memory occupied

by an array with the memory occupied by a single element.

```
In [20]: #include <stdio.h>
         #include <stdlib.h>

         void init_array( int arr[], int n, int initial ){
             for ( int i = 0; i < n; i++ ){
                 arr[i] = initial;
             }
         }

         int main(){
             int capacity;
             int numbers[10];

             init_array(numbers, 5, 10);

             capacity = sizeof(numbers) / sizeof(numbers[0]);

             printf("%d\n", capacity);

             return EXIT_SUCCESS;
         }
```

```
10
```

1.2.1 This does not work

If we pass an array into the function by reference, in the function scope we will end up with a *pointer* to the array, *not the entire array itself*.

```
In [ ]: #include <stdio.h>
        #include <stdlib.h>

        void init_array( int arr[], int n, int initial ){

            int n = sizeof(arr) / sizeof(arr[0]); // since arr is a pointer, n isn't the lengt

            for ( int i = 0; i < n; i++ ){
                arr[i] = initial;
            }
        }
```

1.3 2-dimensional Array

Arrays in C can also be 2-dimensional. The following code generates a 10x10 identity matrix:

In [1]:
```c
#include <stdio.h>
#include <stdlib.h>

void main(void){

    int matrix[10][10] = {0}; // a 10x10 matrix in 2d array, initialized to zero

    // generate the identity matrix
    for (int row = 0; row < 10; row++)
    {
        matrix[row][row] = 1;
    }

    // print the matrix
    for (int row = 0; row < 10; row++ )
    {
        for (int col = 0; col < 10; col++ )
        {
            printf("%2d ", matrix[row][col]);
        }
        printf("\n");
    }
}
```

```
 1  0  0  0  0  0  0  0  0  0
 0  1  0  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0
 0  0  0  1  0  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0  0  1
```