# Chapter 6 - Arrays

A **array** is a set of values where each value is identified and referenced by a number (called an index). The nice thing about arrays is that they can be made up of any type of element, including basic types like ints and doubles, but all the values in an array have to have the same type.

When you declare an array, you have to determine the number of elements in the array. Otherwise the declaration looks similar to other variable types:

int c[4];

double values[10];

Syntactically, array variables look like other C variables except that they are followed by [NUMBER\_OF\_ELEMENTS], the number of elements in the array enclosed in square brackets. The first line in our example, int c[4]; is of the type "array of integers" and creates a array of four integers named c. The second line, double values[10]; has the type "array of doubles" and creates an array of 10 doubles.

C allows you to to initialize the element values of an array immediately after you have declared it. The values for the individual elements must be enclosed in curly brackets {} and separated by comma, as in the following example:

int c[4] = {0, 0, 0, 0};

This statement creates an array of four elements and initializes all of them to zero. This syntax is only legal at initialization time. Later in your program you can only assign values for the array element by element.

Imagine a set of boxes with large number in them and small numbers written on the box. The large numbers inside the boxes are the values of the **elements** in the array. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new array, without initializing, the arrays elements typically contain arbitrary values and you must initialize them to a meaningful value before using them.

## Accessing elements

The [] operator allows us to read and write the individual elements of an array. The indices start at zero, so c[0] refers to the first element of the array, and c[1] refers to the second element. You can use the [] operator anywhere in an expression:

c[0] = 7;

c[1] = c[0] \* 2;

c[2]++;

c[3] -= 60;

All of these are legal assignment statements.

By now you should have noticed that the four elements of this array are numbered from 0 to 3, which means that there is no element with the index 4.

Nevertheless, it is a common error to go beyond the bounds of an array. In safer languages such as Java, this will cause an error and most likely the program quits. C does not check array boundaries, so your program can go on accessing memory locations beyond the array itself, as if they where part of the array. This is most likely wrong and can cause very severe bugs in your program.

**It is necessary that you, as a programmer, make sure that your code correctly observes array boundaries!**

You can use any expression as an index, as long as it has type int. One of the most common ways to index an array is with a loop variable. For example:

int i = 0;

while (i < 4)

{

printf ("%i\n", c[i]);

i++;

}

This is a standard while loop that counts from 0 up to 4, and when the loop variable i is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when i is 0, 1, 2 and 3.

Each time through the loop we use i as an index into the array, printing the ith element. This type of array traversal is very common. Arrays and loops go together like fava beans and a nice Chianti.

## Copying arrays

Arrays can be a very convenient solution for a number of problems, like storing and processing large sets of data.

However, there is very little that C does automatically for you. For example you can not set all the elements of an array at the same time and you can not assign one array to the other, even if they are identical in type and number of elements.

double a[3] = {1.0, 1.0, 1.0};

double b[3];

a = 0.0; /\* Wrong! \*/

b = a; /\* Wrong! \*/

In order to set all of the elements of an array to some value, you must do so element by element. To copy the contents of one array to another, you must again do so, by copying each element from one array to the other.

int i = 0;

while (i < 3)

{

b[i] = a[i];

i++;

}

## for loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it.

This type of loop is so common that there is an alternate loop statement, called for, that expresses it more concisely. The general syntax looks like this:

for (INITIALIZER; CONDITION; INCREMENTOR)

{

BODY

}

This statement is exactly equivalent to

INITIALIZER;

while (CONDITION)

{

BODY

INCREMENTOR

}

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read.

int i;

for (i = 0; i < 4; i++)

{

printf("%i\n", c[i]);

}

## Array length

C does not provide us with a convenient way to determine the actual length of an array. Knowing the size of an array would be convenient when we are looping through all elements of the array and need to stop with the last element.

In order to determine the array length we could use the sizeof() operator, that calculates the size of data types in bytes. Most data types in C use more than one byte to store their values, therefore it becomes necessary to divide the byte-count for the array by the byte-count for a single element to establish the number of elements in the array.

sizeof(ARRAY)/sizeof(ARRAY\_ELEMENT)

It is a good idea to use this value as the upper bound of a loop, rather than a constant. That way, if the size of the array changes, you won't have to go through the program changing all the loops; they will work correctly for any size array.

int i, length;

length = sizeof (c) / sizeof (c[0]);

for (i = 0; i < length; i++)

{

printf("%i\n", c[i]);

}

The last time the body of the loop gets executed, the value of i is length - 1, which is the index of the last element. When i is equal to length, the condition fails and the body is not executed, which is a good thing, since it would access a memory location that is not part of the array.

## Arrays and random

The numbers generated by rand() are supposed to be distributed uniformly. That means that each value in the range should be equally likely. If we count the number of times each value appears, it should be roughly the same for all values, provided that we generate a large number of values.

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

## Array of random numbers

The first step is to generate a large number of random values and store them in a array. By "large number," of course, I mean 20. It's always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following function takes three arguments, an array of integers, the size of the array and an upper bound for the random values. It fills the array of ints with random values between 0 and upperBound-1.

void RandomizeArray (int array[], int length, int upperBound)

{

int i;

for (i = 0; i < length; i++)

{

array[i] = rand() % upperBound;

}

}

The return type is void, which means that this function does not return any value to the calling function. To test this function, it is convenient to have a function that outputs the contents of a array.

void PrintArray (int array[], int length)

{

int i;

for (i = 0; i < length; i++)

{

printf ("%i ", array[i]);

}

}

The following code generates an array filled with random values and outputs it:

int r\_array[20];

int upperBound = 10;

int length = sizeof(r\_array) / sizeof(r\_array[0]);

RandomizeArray (r\_array, length, upperBound);

PrintArray (r\_array, length);

On my machine the output is:

3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect each digit to appear the same number of times---twice each. In fact, the number 6 appears five times, and the numbers 4 and 8 never appear at all.

Do these results mean the values are not really uniform? It's hard to tell. With so few values, the chances are slim that we would get exactly what we expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that count the number of times each value appears, and then see what happens when we increase the number of elements in our array

## Passing an array to a function

You probably have noticed that our RandomizeArray() function looked a bit unusual. We pass an array to this function and expect to get a a randomized array back. Nevertheless, we have declared it to be a void function, and miraculously the function appears to have altered the array.

This behaviour goes against everything what I have said about the use of variables in functions so far. C typically uses the so called **call-by-value** evaluation of expressions. If you pass a value to a function it gets copied from the calling function to a variable in the called function. The same is true if the function returns a value. Changes to the internal variable in the called function do not affect the external values of the calling function.

When we pass an array to a function this behavior changes to something called **call-by-reference** evaluation. C does not copy the array to an internal array -- it rather generates a reference to the original array and any operation in the called function directly affects the original array. This is also the reason why we do not have to return anything from our function. The changes have already taken place.

Call by reference also makes it necessary to supply the length of the array to the called function, since invoking the sizeof operator in the called function would determine the size of the reference and not the original array.

We will further discuss call by reference and call by value in Section [Pointers and Addresses](#Pointers_and_Addresses), Section [Call by value](#Call_by_value) and [Call by reference](#Call_by_reference).

## Counting

A good approach to problems like this is to think of simple functions that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. This approach is sometimes called **bottom-up design**.

Of course, it is not easy to know ahead of time which functions are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

In our current example we want to examine a potentially large set of elements and count the number of times a certain value appears. You can think of this program as an example of a pattern called ``traverse and count.'' The elements of this pattern are:

* A set or container that can be traversed, like a string or a array.
* A test that you can apply to each element in the container.
* A counter that keeps track of how many elements pass the test.

In this case, I have a function in mind called HowMany() that counts the number of elements in a array that are equal to a given value. The parameters are the array, the length of the array and the integer value we are looking for. The return value is the number of times the value appears.

int HowMany (int array[], int length, int value)

{

int i;

int count = 0;

for (i=0; i < length; i++)

{

if (array[i] == value) count++;

}

return count;

}

## Checking the other values

HowMany() only counts the occurrences of a particular value, and we are interested in seeing how many times each value appears. We can solve that problem with a loop:

int i;

int r\_array[20];

int upperBound = 10;

int length = sizeof(r\_array) / sizeof(r\_array[0]);

RandomizeArray(r\_array, length, upperBound);

printf ("value\tHowMany\n");

for (i = 0; i < upperBound; i++)

{

printf("%i\t%i\n", i, HowMany(r\_array, length, i));

}

This code uses the loop variable as an argument to HowMany(), in order to check each value between 0 and 9, in order. The result is:

value HowMany

0 2

1 1

2 3

3 3

4 0

5 2

6 5

7 2

8 0

9 2

Again, it is hard to tell if the digits are really appearing equally often. If we increase the size of the array to 100,000 we get the following:

value HowMany

0 10130

1 10072

2 9990

3 9842

4 10174

5 9930

6 10059

7 9954

8 9891

9 9958

In each case, the number of appearances is within about 1\% of the expected value (10,000), so we conclude that the random numbers are probably uniform.

## A histogram

It is often useful to take the data from the previous tables and store them for later access, rather than just print them. What we need is a way to store 10 integers. We could create 10 integer variables with names like howManyOnes, howManyTwos, etc. But that would require a lot of typing, and it would be a real pain later if we decided to change the range of values.

A better solution is to use a array with length 10. That way we can create all ten storage locations at once and we can access them using indices, rather than ten different names. Here's how:

int i;

int upperBound = 10;

int r\_array[100000];

int histogram[upperBound];

int r\_array\_length = sizeof(r\_array) / sizeof(r\_array[0]);

RandomizeArray(r\_array, r\_array\_length, upperBound);

for (i = 0; i < upperBound; i++)

{

int count = HowMany(r\_array, length, i);

histogram[i] = count;

}

I called the array **histogram** because that's a statistical term for a array of numbers that counts the number of appearances of a range of values.

## A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it calls HowMany(), it traverses the entire array. In this example we have to traverse the array ten times!

It would be better to make a single pass through the array. For each value in the array we could find the corresponding counter and increment it. In other words, we can use the value from the array as an index into the histogram. Here's what that looks like:

int upperBound = 10;

int histogram[upperBound] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

for (i = 0; i < r\_array\_length; i++)

{

int index = r\_array[i];

histogram[index]++;

}

The second line initializes the elements of the histogram to zeroes. That way, when we use the increment operator (++) inside the loop, we know we are starting from zero. Forgetting to initialize counters is a common error.

As an exercise, encapsulate this code in a function called Histogram() that takes an array and the range of values in the array (in this case 0 through 10) as two parameters min and max. You should pass a second array to the function where a histogram of the values in the array can be stored.

## Partially filled arrays

In C, an array's sized is fixed at the moment it is declared. But what if we don't know how much space is needed when an array is declared. For example, we may want to store the names of students who enrolled in a course, but we don't know how many students there will be until they register. Or we may want to store some numbers that are entered by the user, but we don't know how many numbers there will be until they stop entering them. In these cases, we can use a **partially filled array**.

A partially filled array is an array that has some empty slots that are not used for storing data. For example, if we declare an array of size 10, but only use 5 elements to store some data, then we have a partially filled array with 5 empty slots. The advantage of using a partially filled array is that we can allocate more space than we need at first, and then fill it up later as needed. The disadvantage is that we may waste some memory if we allocate too much space than we need.

To work with partially filled arrays in C, we need two things: a variable that defines the maximum size of the array (its capacity), and a local variable that keeps track of how many elements are actually used in the array (usually called size). The size variable also allows us to ensure that we don't exceed the capacity of the array when adding new elements. All the values are stored contiguously in the array starting at the zero index. This means we can use the size variable to find the next open location. We can assign a value at that location, then increase the size.

#define CAPACITY 10 //the maximum size of the array

int arr[CAPACITY]; // Declare an int array with the max capacity

int size = 0; //the current size of the filled part

//get some values from the user using a user-terminated loop (or sentinel loop)

int num;

puts("Enter some numbers (enter -1 to stop):");

scanf("%d", &num);

//while the user enters a valid number

//and we have room in the array

while (num != -1 && size < CAPACITY)

{

arr[size] = num;

size++;

puts("Enter some numbers (enter -1 to stop):");

scanf("%d", &num); // Read another number from input

}

## Glossary

**array:**

A named collection of values, where all the values have the same type, and each value is identified by an index.

**element:**

One of the values in a array. The [] operator selects elements of a array.

**index:**

An integer variable or value used to indicate an element of a array.

**increment:**

Increase the value of a variable by one. The increment operator in C is ++.

**decrement:**

Decrease the value of a variable by one. The decrement operator in C is --.

**bottom-up design:**

A method of program development that starts by writing small, useful functions and then assembling them into larger solutions.

**histogram:**

A array of integers where each integer counts the number of values that fall into a certain range.

Permission is granted to copy, distribute, transmit and adapt this work un- der the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national License: https://creativecommons.org/licenses/by-nc/4.0/. If you are interested in distributing a commercial version of this work, please contact the author(s). The LATEX source and code for this book is available from: https://github.com/mkpenta/ThinkC