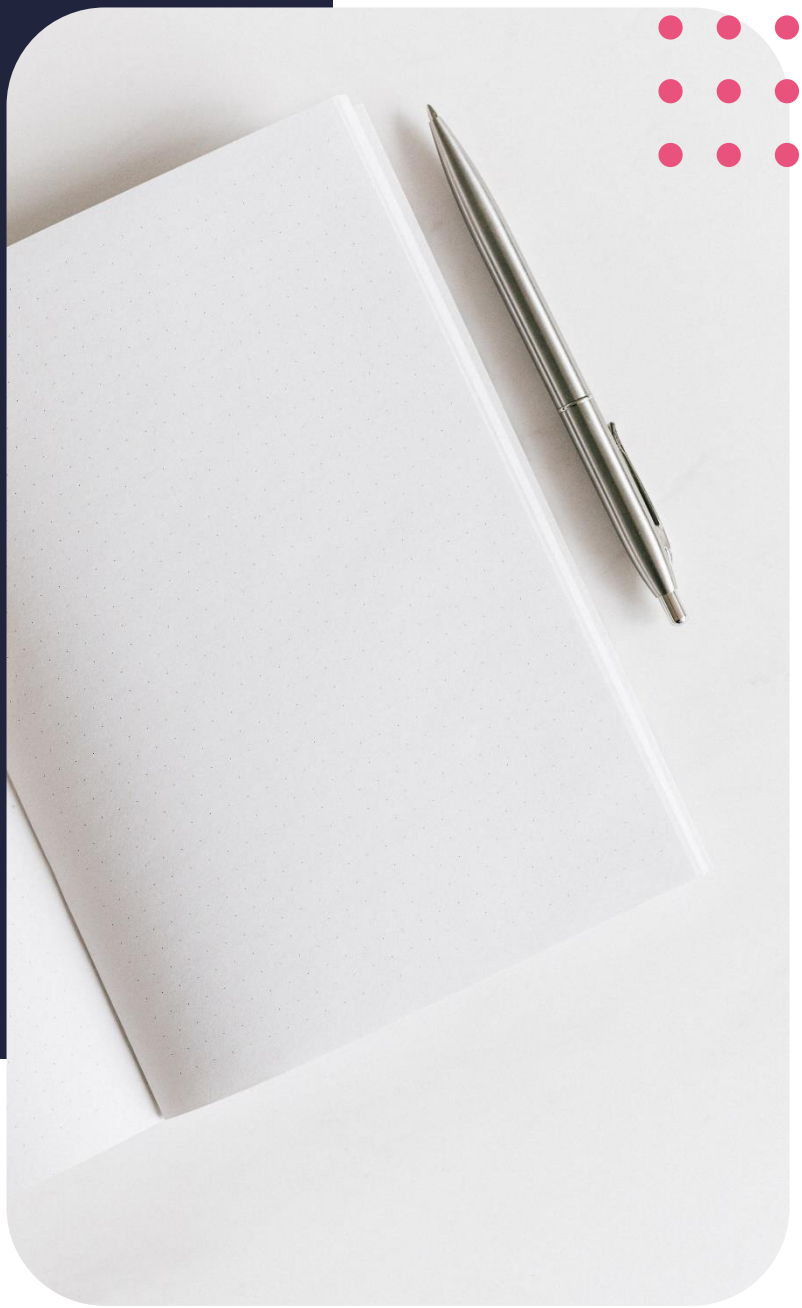
Diploma in Computer Science

Arrays

Contents



Lesson outcomes

By the end of this lesson, you should be able to:

- Explain what an array is
- Identify where an array can be used
- Explore data manipulation using arrays
- Apply array use guidelines in programming

Arrays?

Arrays are amongst some of the oldest and most respected tools in programming, used by almost every program in existence. They provide an easy way of manipulating data and given the vast amounts of data that we process each second, an array only makes sense for the programmer most of the time. In the early days, array indexing was originally done by self-modifying code, then later using memory segmentation around the 1960s. Around the 1970s, high-level languages started appearing, and had support for arrays as we know them today.

Why do we even need arrays?

What are we even talking about? Well, an array is data structure consisting of a collection of elements, which are either values or variables. Why do we this? Imagine you have 1000 items, say, the list of Fibonacci numbers that we generated in lesson 2. How would you store them? Hmm, the first thing that might come up in your mind is creating a thousand variables, but you'd probably die of exhaustion before you finish typing out the code. It's also not a very readable way of writing code. In this case, that is where 2 really nifty C features come in. You'd define an array that can hold 1000 elements, create a for loop that iterates through the 1000 elements, and voila! Problem solved!

Arrays are also used in other important data structures, such as linked lists, hash tables, search trees, queues, stacks and strings, as we shall see soon. Arrays are so important that processors are often optimised for array operations. Another key feature of arrays is that you can compile the indices at runtime. This is how you can use a few lines of code to manipulate an array which contains thousands of elements. It would really be wasteful to do this using regular variables as it will just increase the size of the programme and that affects the program's system footprint. Everything also used for dynamic memory allocation within the programs using

emulation. In mathematics, that is a used to implement affecters in matches is and other types of tables. A lot of databases typically use an array to store records.

The first majority of the data that we use in our day to day lives relies on searching and sorting. In fact, for you to search do you take efficiently needs to be sorted. This functionality is almost entirely dependent on arrays, as they make these operations much more efficient than working with separate variables. If you go back to module one you will see that searching and sorting algorithms that we looked at actually used arrays, even in the diagrams that we used! We just didn't mention it at that point because it wouldn't make much sense. If you are to proceed with computer science, you will see that most of these algorithms will use arrays, so it is important that you fully understand how arrays work.

Array syntax

There are a number of ways of declaring arrays but they all pretty much have the same result. The first way is to declare an array and immediately assign elements to it. The syntax looks like this:

```
Data_type arrayname[]={element1, element2, ….element n};
```

Here we haven't specified the size but the compiler knows that we have added n elements and so creates an array of size n.

The 2nd way is to declare an array and specify the number of elements n.

```
Data_type arrayname[n]={element1, element2, ….element n};
```

Where n refers to the size of the array

If you do not want to declare array elements right away, then you can specify the number of elements in the square within the square brackets. This will create an empty array which you will then assign values at a later time, Such as when the program is running and you ask for input from the user, input from a file, or storing results of a calculation or an operation. The syntax will look like this:

```
Data_type arrayname[n];
```

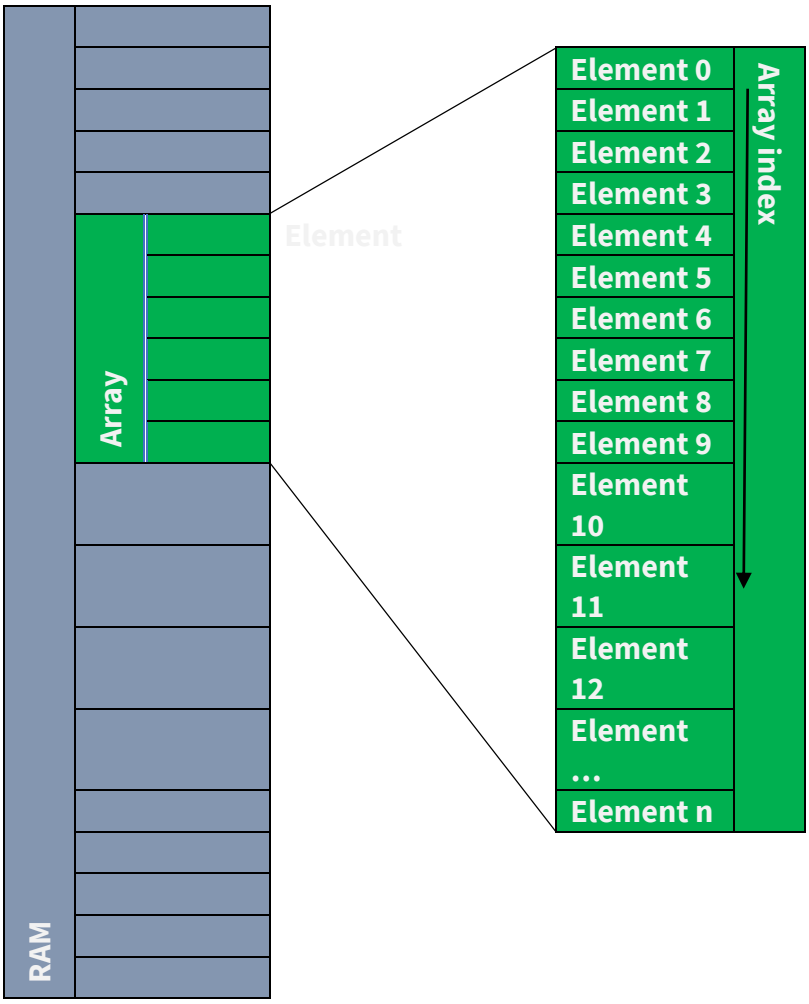
Here also, n refers to the size of the array.

The data in an array is typically related, which only makes sense since the array's data type determines what kind of data you can store in the array. This follows that arrays are classified as homogenous data structures because they store elements of the same type.

You may be wondering how you retrieve the array elements after storing them, wait, how do we even store elements and keep track of the exact position in the array? Well, this is where the array's best friend comes in, the for loop. Remember we established that the for loop is best for counting; it allows us to run through the array and access each of the address spaces. Notice

here we use the phrase address space. This is because arrays are Contiguous block of memory, each with its own address. They are grouped together and accessed using the same variable name, but then the index Refers to each of the elements is a subunit of the variable. This means that the elements can be accessed individually but are part of the same data structure. The idea is to store items in such a way that they are one entity but cannot be accessed in one go. An example is the set of marks for a class of 50. Ideally, the marks make sense when stored in one location, but aren't really much use if they are poured out all at one. Rather if the marks are accessed one at a time, it allows them to be processed and produce much more meaningful data.

To get a better idea of how this works, let's take a look at the representation of an array in memory.



The array is a block of memory, as we previously mentioned, which is referenced by one variable name. The index allows you to access each block of memory as though it were a separate variable, which in principle it is. The array has a data type, for example

Int ages {8}; carries the data type int. This means that each of the indexed locations are as large as the size of int on your PC. On my PC, the Size of Int is 4 bytes, so the array ages will occupy digital bytes of memory. This is one thing you have to be careful about because it is very easy to waste memory using arrays. You should only ever declare the size of array that fits the approximate number of elements that you need to store. For example, if you want to store 1000 elements, there is no need to create an area that can hold 2000 elements. The other 1000 elements will never be used, so it is wasted memory. From the C99 standard onward, C supports dynamic array allocation.

We should also note here that once you declare the size and type of an array and compile the programme you cannot change these attributes. Another common error that occurs when working with arrays is referencing out of bounds of the array. For example, if we go back to our 1000 element array, we can only reference elements at index 0 up to index 1000. If you attempt to reference element 1012, your compiler may generate an error. Sometimes the error goes unnoticed right up until the time you run your program. Just like using an uninitialized variable this cause random unexpected results. Which brings us to rule number 1: never attempt to access elements outside the bounds of the array. But hey, wait a minute, did we just say counting starts at index zero? Well, this is because of what we said in lesson 1. In computing and computer science, zero counts.

The amazing power of arrays comes from their efficiency to access values. This is because off the fact that they are stored in contiguous memory.

Array operations

Look at the many interesting things that you can do using arrays. To assign a value to an array we use the index. Typically, you would initialise in the area then that would mean it would have data in it already. If you do not initialise an array, then at some point an operation in your programme needs to to add data to the error. If you are a adding data to the complete array sequentially, you would use a for loop does start from the index 0 right up to the last element in the array. You do not necessarily need to add items sequentially to the array. Using an index means that you can randomly access any one of the elements using the index number. If for example you want to assign a value to the 6th position in an array named ages, then the syntax of the statement would look like this:

```
Ages[6] = 21;
```

This code snippet would assign the value 21 to the 6th position in the array. This demonstrates one of the clever capabilities of arrays.

Just like normal variables you can perform the regular arithmetic operations addition, subtraction, multiplication, division and modulus. And it doesn't even change one bit! The only thing you need to be wary of is making sure that the area index is always correct. When dealing with Arrays it's pretty easy to mix up index numbers and ending up doing calculations on the wrong elements. It is pretty easy to write a programme that will wreak a whole lot of havoc, especially if your index is being passed by another variable. To get a clearer picture of what we're trying to get at here let's look at this code snippet.

```
  #include<stdio.h>

Int main()
{
 Int Size, i, a[10], b[10];
 Int Addition[10], Subtraction[10], Multiplication[10], Module[10];
 Float Division[10];

 Printf("\n Please Enter the Size of the Array\n");
 Scanf("%d", &Size);

 Printf("\nplease Enter the First Array Elements\n");
 For(i = 0; i < Size; i++)
  {
      Scanf("%d", &a[i]);
  }

 Printf("\n Please Enter the Second Array Elements\n");
 For(i = 0; i < Size; i ++)
  {
      Scanf("%d", &b[i]);
  }

 For(i = 0; i < Size; i ++)
  {
      Addition [i]= a[i] + b[i];
      Subtraction [i]= a[i] - b[i];
      Multiplication [i]= a[i] * b[i];
      Division [i] = a[i] / b[i];
      Module [i] = a[i] % b[i];
  }

 Printf("\n Add\t Sub\t Multi\t Div\t Mod");
 For(i = 0; i < Size; i++)
  {
      Printf("\n%d \t ", Addition[i]);
      Printf("%d \t ", Subtraction[i]);
      Printf("%d \t ", Multiplication[i]);
      Printf("%.2f \t ", Division[i]);
      Printf("%d \t ", Module[i]);
  }

  Return 0;
```

```
}
```

Here we have to clear quite a number of arrays, but all of them are all of the same size which is 10. We have integer type arrays a comma B, addition, subtraction, multiplication, and module. You recall from our lesson on data types that when you are performing division int is not the best data type of choice because it will throw away any fractional part of the result and this results in wild in accuracies. For this reason, we declared the area that is going to store division results as float to preserve the decimal part of the result. To aid the operation that we want to perform we have also declared variables size and I. The variable size is used to determine the number of elements that we are going to read from the user. The variable I is used as a counter variable for the for loop. Within Reed airy elements from the user, since, as you may have noticed, when we declared our arrays, we did not initialise them. The first batch of the array elements that we read will be stored in the array a. The second batch of elements that we read will be stored in array b. The programme then enters at 3rd for loop which will perform addition by taking an element from the first array a, and add it's to the corresponding element from array b. The same is done for subtraction multiplication division and modulus. Using the formatting marks that we learned you know input and output lesson; the programme then displays the output in the form of a table. And there you have it one simple programme that demonstrate the power of arrays!

Just like normal variables, you can also perform logical operations on arrays. Again, you just need to keep note off the index number, and this cannot be stressed enough. You will find that most of the time when you have problems with arrays, it has to do with the index number. In my experience, paying extra attention when you are declaring and initialising your arrays, you save yourself a lot of trouble. One little problem that you might see popping up when you are dealing with arrays is that errors often arise not at compile time but at runtime. You can't simply rely on the compiler to find errors for you as some of them might slip right past it! You simply just have to keep your logic in check.

Character strings

We have already talked about strings and previous lessons and if it looked at the syntax and usage. You may have been wondering wave and never actually used strings in any of our programs. Well, it only makes sense to take a closer look at strings after looking at arrays. But wait why? Isn't a string a data type just like any other data type? Why should it be closely associated with arrays?

Strings=arrays

Well surprise! Surprise! In see a string is a sequence of characters terminated with a null character which is a\ 0. This sequence of characters is actually in array! If you look closely at the syntax of the declaration of a string you will see that it uses the exact same square brackets that we used to define an array! It is actually exactly the same, left ear and right ear!

String syntax

Let's revisit this syntax of a string. Let's say we want to create a string of characters called names. The syntax would look like this:

```
Char names[5];
```

You record that when you are declaring a string you have to specify the length of the string. You record that with said a character is represented by one byte. This means that in this array each of the elements is exactly 1 byte. There are four ways in which you can initialise a string and this is exactly the same as when you declare a regular array.

In the first method, you put the elements of the string in quotes. Here you do not need to specify the number of elements as the compiler will know that when it encounters a" it depends the string with a\ 0 to indicate the end.

```
Char c[] = "abcd";
```

In the second method, you do specify the length of the string, but the compiler knows that the string can be expanded to the number of characters that you specify in the square brackets.

```
Char names[30] = "abcd";
```

You can also separate the Array elements by in closing each of them in single quotes and separating each choosing a comma then appending the end of the array what is a\ 0. The whole thing is enclosed in curly brackets. You then don't really need to specify the number of elements in the string.

```
Char names[] = {'a', 'b', 'c', 'd', '\0'};
```

You can also do it in exactly the same way in our previous example but then specify the number of elements in the string.

```
Char names[5] = {'a', 'b', 'c', 'd', '\0'};
```

Keep in mind that if you specify the number of elements and the string you cannot add any

more than the ones that you add at declaration. Another point to note is that when you explicitly initialise a character airy by listing all of its characters separately using single quotes and comma, then you need to append the array with a\ 0. If you do not add the\ 0, the compiler will throw an error let's say for example you have declared a character array named names, if you declare it like this

```
Char names[6] = "Samuell";     // won't work
```

```
Char names[6];
Str = "Samuel";    // won't work
```

It's pretty easy to see that this code snippet violates the syntax that we learned about earlier in the first part of this lesson.

When reading strings from the user, that is using the scan function, the programme will read the sequence of characters that the user enters until it encounters white space in the form of a new line, a space or a tab, then terminates the read operation. This is kind of a problem if you want to read a string that contains a space such as your name and surname for example if you want to read the string Samuel Masuka, only the first part of the string, Samuel is rea, and the programme completely ignores all of the characters that come after the space. Luckily, there is a way around this as C supports a format specification known as the edit set conversion code:

```
%[ ... ]
```

This allows you to read a line that is a variety of characters including white spaces. Voila! Problem solved! It is pretty interesting that in spite of all the rules and strict syntax that come with programming languages, your own create creativity and ability to solve problems means the sky is pretty much the limit. There are so many ways to get around the limitations of programming languages.

String operations

We can't talk about strings without talking about the string.h header file. Strings themselves can quickly become Kimber some to deal with and that is where this header file comes in. By the way you can go into a header file and explore the functions that it contains. This is not only limited to the string.h header file but all header files contained in the C standard library. This is actually standard procedure when you are working on a new project and identify the header files that you need to accomplish your task. Let's now look at the more commonly used string functions. This table shows string functions and their purpose.

strcat	Concatenate two strings
Strchr	String scanning operation
Strcmp	Compare two strings
Strcpy	Copy a string
Strlen	Get string length
Strncat	Concatenate one string with part of another
Strncmp	Compare parts of two strings
Strncpy	Copy part of a string
Strrchr	String scanning operation

Strcat

Don't be fooled by the name of this string it's not that complicated at all! The string concatenation function simply takes the contents of a given string say string2, and joins the two the end of another given string that will name string1. The string concatenation function overrides them now character symbol\ 0. The general syntax looks like this:

```
Char *strcat(char *destination, const char *source)
```

Strchr

At the string scanning function is used to find certain characters in a string. Given a character, the function runs through the string and returns the location of the given character in the string function. If the character is not found in the string, then the function will return null. Fun fact this function used to be called index, because it returns the index of a given character in the string array. Its general syntax looks like this:

```
Char *strchr(const char *str, int c)
```

The string scanning function takes 2 parameters, str, which is the string to be scanned, and C, which is the character that just searched for in the string array.

Strcmp

The string compare function will compare two given strings character by character. If the first character in each of the strings is the same the next character in each of the strings are compared until the function encounters two different strings or a null character. If the two given strings are equal the function returns 0. The function will return a negative value if the ASCII value of the first unmatched character is less than the second. The function will return a positive integer value if the ASCII value of the unmatched character is greater than the 2nd.

The general syntax of the string copy function looks like this:

```
Char *strcpy(char *restrict s1, const char *restrict s2);
```

In this function S1 stands for the first string that is going to be compared against the second string denoted by S2. In this comparison the null byte, \ 0, is included in the comparison. This means that strings can have the same length in terms of characters, but if the actual string array is longer, even though it doesn't contain any subsequent characters, the two strings are considered unequal.

A simple fun way in which you can test out this function is creating a programme that asks for a password from the user and checks if the password is correct against a provided password and allows the user to log in if the password is correct.

Strcpy

The string copy function is probably the simplest of them all. The function takes the contents of a given string say s1, and copies them over into another string array, say s2. The general syntax of the string copy function looks like this:

```
Char *strcpy(char *restrict s1, const char *restrict s2);
```

But, here be Dragons! The function does not return any value that indicates if there is an error caused by the strings not being equal , that is including the termination symbol\ 0. If you copy a string onto S2 that is larger then the size of s2, the programme will simply overwrite past the length of the string array, into the adjacent memory, and this typically causes the programme to crash. Besides what we discussed in our lesson on scope, this is one of the thing that can be exploited by hackers to gain entry into the system,. They simply locate an area to overflow in your system, overwrites the adjacent memory with desired information, in theyre in! Again, C comes to your rescue! It's just typically always better to use the `strncpy()` function instead of the `strcpy()` function to prevent and handled buffer overflow.

Strlen

The string length function is rather simple to. The function computes the number of bytes in a string, excluding the terminating null byte \ 0. The function then returns the number of bytes in the string. The basic syntax of the function looks like this:

```
Size_t strlen(const char *s);
```

Strncat

This function does exactly what the string concatenation function does except it does it in a slightly different way. Let's take a look at the syntax

```
Char *strncat(char *restrict s1, const char *restrict s2, size_t n);
```

You will notice that we have introduced a variable n. So instead of simply joining the two strings, it only takes the first string and joins it to the first end characters in the second string.

Strncmp

Another function that works with bits and pieces of a string as the alternate string compare function which has basic syntax that looks like this:

```
Int strncmp(const char *s1, const char *s2, size_t n);
```

Again you will see that we have introduced a variable denoted by n. The function will only compare the first n bytes off the first string to the second string.

Strncpy

This function does exactly what the string copy function does , except that, like the last two that we looked at , it copies only the first n bytes of the string. The basic syntax of this function looks like this:

```
Char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

The last of the commonly used string functions that we're going to look at is the string scan function . Unlike the first one that we looked at this function will scan the string but instead of returning the first occurrence of the character indicated by C it will return the last location of the character indicated by C. Its basic syntax looks like this:

```
Char *strrchr(const char *s, int c);
```

We have covered the basic set of functions used when manipulating strings. As you may have noticed, you have to be extra careful when working with these functions since the error handling capabilities are limited. A lot of the errors will only come up at runtime and will wreak havoc in your program. The one thing you have to be wary of most is the correspondence of the lengths of the strings that you're working with. It is a lot better to have left over unused array indices then to have the string overflow and cause a runtime error.

Memchr	Find a byte in memory
Memcmp	Compare bytes in memory
Memcpy	Copy bytes in memory
Memmove	Copy bytes in memory with overlapping areas
Memset	Set bytes in memory

Strcoll	Compare bytes according to a locale-specific collating sequence
Strcspn	Get the length of a complementary substring
Strerror	Get error message
Strpbrk	Scan a string for a byte
Strspn	Get the length of a substring
Strstr	Find a substring
Strtok	Split a string into tokens
Strxfrm	Transform string

This table shows the functions that exist in sea but are not as commonly used as the ones we just covered. Feel free to run through this table and explore the functions of the stated functions. In fact, we are going to use some of them in our final module when we are covering advanced see concepts.

Array dimensions

All along we have been declaring arrays that appear linear. We've also already discovered that arrays are quite powerful and allow you to do so much with data. What if I told you that there was an even better way of using arrays? We can introduce another dimension to arrays to make the most dimension arrays.

Single dimension arrays

All the arrays that we have discussed so far are 1 dimensional array. This means that if we think of the way the data is arranged, it is essentially just one long chain.

Multi dimension arrays

An easy way of picturing multidimensional arrays is envisioning a table. A table consists of rows and columns. Now, in 2 dimensional arrays the first part of the declaration that is in square brackets represents the table rows, and the second part of the declaration represents the table columns. The basic syntax offered 2-dimensional array looks like this:

```
Data_type array_name[x][y];
```

Now if you remember from mathematics, the x value of a table represents the rows, while the y value of the table represents the columns. Let's look at the example in the code snippet

```
Int twod_array[10][20];
```

Sneak peek: in today's demo we are going to use a 2-dimensional array, but I will save the rest

for later when we get to the demo.

It doesn't end there! You can keep adding more dimensions to an array! The syntax is as easy as just adding more square brackets that contained the size of that part of the array. If for example you want to create a 3 dimensional array, the the basic syntax would look like this:

```
Data_type array_name[x][y][z];
```

Here we have added a third dimension which takes on the value of z.

Multidimensional arrays are very useful in mathematics and computer graphics. If you want to represent a screen for example , which has a resolution of say 1080p, it would mean that the screen has 1920 rows and 1080 columns. If you represent this in a 2 dimensional array that can hold one byte each, it means that you can display 8 bits of colour in each of the pixels on the screen. In mathematics , using a 3 dimensional array, you can represent 3 dimensional graphs that are used to model 3 dimensional shapes. There are countless things that you can achieve using arrays..

Demo

No, the time that we've all been waiting for! In today's demo we are going to create multiplication tables using a 2-dimensional array. I'm sure we all remember multiplication tables from our primary school days when we were first being taught about multiplication. The programme itself is fairly simple. It uses a 2-dimensional array to display the results and uses a for loop to carry out the calculations and store the results in the relevant memory locations within the array. It uses another for loop to retrieve the results from the array and display them on the screen. Enough of the chit chat let's get started!

Conclusion

In this lesson we looked at one of the most powerful tools used in programming languages, arrays we looked at the various ways in which you can manipulate data using arrays and also looked at the loopholes that may cause your program to work correctly when using arrays. We also looked at the string epic a bit closer and establish that it is actually an area that is just terminated by a special symbol. We also looked at the application of arrays in mathematics and other fields of science come and how they implemented in a lot of the software that is used in daily life. Discovered that arrays can also be used in multiple dimensions to give the programme at even more versatility to produce powerful programs. Is one of the bonuses in computer science, we looked at how an array is represented in memory, something that you normally don't come across when learning programming. In our next lesson, we expand a bit more on arrays and explain a very important concept in computer science namely pointers. Buckle up and please try to read ahead as most people usually find this topic a bit tricky.

References

Programiz.com. (2021). *C Multidimensional Arrays (2d and 3d Array)*. [online] Available at: https://www.programiz.com/c-programming/c-multi-dimensional-arrays

Codingame. (2018). *Coding Games and Programming Challenges to Code Better*. [online] Available at: https://www.codingame.com/playgrounds/14213/how-to-play-with-strings-in-c/array-of-c-string

Overiq.com. (2021). *Array of Strings in C*. [online] Available at: https://overiq.com/c-programming-101/array-of-strings-in-c/

Esa.int. (2020). *Array arithmetic*. [online] Available at: https://xmm-tools.cosmos.esa.int/external/sas/current/doc/selectlib/node26.html

Harvard.edu. (2011). *Logical Operators*. [online] Available at: https://r.iq.harvard.edu/docs/zelig/3.5.3/Logical_Operators.html

https://www.facebook.com/sureshbasinasetty (2015). *C Program to Perform Arithmetic Operations on Arrays*. [online] Tutorial Gateway. Available at: https://www.tutorialgateway.org/c-program-to-perform-arithmetic-operations-on-arrays/

Codesdope. (2021). *Codesdope : Array in C : Learn declaration, 2D array, pointer to array, to iterate over array, etc.* [online] Available at: https://www.codesdope.com/c-array/

Lmu.edu. (2021). *C*. [online] Available at: https://cs.lmu.edu/~ray/notes/c/

Cppreference.com. (2020). *Array declaration - cppreference.com*. [online] Available at: https://en.cppreference.com/w/cpp/language/array

Sciencedirect.com. (2016). *Memory Array - an overview | sciencedirect Topics*. [online] Available at: https://www.sciencedirect.com/topics/computer-science/memory-array#:

Poly.edu. (2021). *Arrays in Memory*. [online] Available at: http://cis.poly.edu/~mleung/CS1114/s09/ch07/memory.htm