# ARRAYS AND STRINGS

Sercan Külcü | 21.06.2023

# Contents

Contents 1

1 Declaring and initializing arrays 2

2 Manipulating arrays: accessing, modifying, and traversing 4

2.1 Accessing Array Elements 4

2.2 Modifying Array Elements 4

2.3 Traversing Arrays 5

2.4 Common Array Operations 5

2.5 Multidimensional Arrays 6

3 String manipulation and string library functions 7

3.1 Understanding Strings 7

3.2 String Library Functions 7

3.3 Additional String Library Functions 10

3.4 String Manipulation Techniques 10

# 1 Declaring and initializing arrays

In this chapter, we will delve into the fundamentals of declaring and initializing arrays. Arrays are essential data structures that allow us to store and manipulate multiple values of the same data type. By the end of this chapter, you'll have a solid understanding of how to declare and initialize arrays, unlocking a powerful tool in your programming arsenal.

An array is a collection of elements of the same data type, arranged in a contiguous block of memory. Each element in an array is identified by its index, which represents its position in the array. Arrays provide us with a convenient way to work with multiple values efficiently.

To declare an array in C, we need to specify its data type and size. The syntax for declaring an array is as follows:

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

Here, datatype represents the data type of the elements in the array, arrayName is the name we choose for our array, and arraySize defines the number of elements the array can hold. It's important to note that the arraySize must be a positive integer value.

For example, to declare an array of integers named myArray that can hold 10 elements, we would write:

```
int myArray[10];
```

After declaring an array, we can initialize it by assigning values to its elements. There are two common ways to initialize an array: explicitly and implicitly.

1. Explicit Initialization

Explicit initialization involves providing a set of initial values for the array elements at the time of declaration. We enclose the initial values within curly braces {}, separated by commas. The number of initial values provided must match the size of the array.

For instance, let's initialize our myArray with some values:

```
int myArray[5] = {10, 20, 30, 40, 50};
```

In this example, we explicitly initialize myArray with 5 elements, where each element is assigned a specific value.

2. Implicit Initialization

In C, arrays that are not explicitly initialized will be implicitly initialized with default values based on their data type. For example, arrays of integers will have all their elements set to 0, while arrays of characters will have their elements initialized to the null character ('\0').

Consider the following declaration:

float floatArray[3];

In this case, the elements of floatArray will be implicitly initialized to 0.0.

To access individual elements of an array, we use the array name followed by the index of the element in square brackets. The index ranges from 0 to arraySize - 1, inclusive.

For example, to access the third element of our myArray, we would write:

int element = myArray[2];

The value of element will be 30 since arrays are zero-indexed, meaning the first element is at index 0, the second element at index 1, and so on.

# 2 Manipulating arrays: accessing, modifying, and traversing

In this chapter, we will delve deeper into manipulating arrays by exploring techniques for accessing, modifying, and traversing array elements. By the end of this chapter, you'll have a solid understanding of how to work with arrays effectively and efficiently.

## 2.1 Accessing Array Elements

Accessing individual elements of an array allows us to retrieve or manipulate specific values. As a quick recap, to access an element, we use the array name followed by the index of the element in square brackets.

Let's consider the following example:

int myArray[5] = {10, 20, 30, 40, 50};

int element = myArray[2];

In this case, we access the third element of myArray using the index 2 and assign its value (30) to the variable element. Remember, array indices start from 0, so the first element is at index 0, the second at index 1, and so on.

## 2.2 Modifying Array Elements

Arrays are mutable, which means we can change the values of individual elements. To modify an element, we access it using its index and assign a new value.

For instance, let's modify the second element of myArray from our previous example:

myArray[1] = 75;

After executing this line, the value of the second element will change from 20 to 75. Feel free to experiment with different indices and values to update specific elements as needed.

## 2.3 Traversing Arrays

Traversing an array involves visiting each element systematically. It allows us to perform operations on every element, such as printing values, calculating totals, or searching for specific elements.

The most common way to traverse an array is by using a loop. Let's explore a simple example using a for loop:

```
int myArray[5] = {10, 20, 30, 40, 50};

int i;



for (i = 0; i < 5; i++) {

   printf("%d ", myArray[i]);

}
```

In this code snippet, we use a loop to iterate over each element of myArray. The loop variable i starts from 0 and increments by 1 until it reaches the array size minus 1 (i.e., 5 - 1 = 4). During each iteration, we print the value of the current element using the format specifier %d in the printf() function.

When you run this code, you will see the output: 10 20 30 40 50. This demonstrates how we can traverse an array and perform operations on each element systematically.

## 2.4 Common Array Operations

Manipulating arrays involves more than just accessing and modifying elements. Here are some common operations you might encounter when working with arrays:

- Finding the minimum or maximum element: Iterate through the array, comparing each element with a variable that keeps track of the minimum or maximum value encountered so far.
- Calculating the sum or average: Traverse the array, adding each element to a variable that accumulates the sum. To find the average, divide the sum by the number of elements.

- Searching for an element: Traverse the array and compare each element with the target value. If a match is found, you can take appropriate action (e.g., print a message or perform further operations).
- Reversing the array: Swap elements from the beginning and end of the array, gradually moving towards the middle, until the entire array is reversed.

## 2.5 Multidimensional Arrays

So far, we have been working with one-dimensional arrays, where elements are arranged in a single line. However, arrays can also have multiple dimensions, forming matrices or tables of values.

A multidimensional array is essentially an array of arrays. It can have two or more dimensions, allowing you to organize data in a grid-like structure. To declare a multidimensional array, you specify the size for each dimension.

Let's consider a simple example of a 2D array, often referred to as a matrix:

```
int matrix[3][3] = {

   {1, 2, 3},

   {4, 5, 6},

   {7, 8, 9}

};
```

In this case, we declare a 2D array named matrix with dimensions 3x3. We provide initial values enclosed in nested curly braces. Each row represents a subarray, and the elements within each row are separated by commas.

To access or modify elements in a multidimensional array, we use multiple indices. For example, to access the element in the second row and third column of matrix, we use matrix[1][2], where the first index represents the row and the second index represents the column.

# 3 String manipulation and string library functions

In this chapter, we will dive into the fascinating world of string manipulation and explore the powerful string library functions available in C. Strings are an essential part of many programs, allowing us to work with textual data. By the end of this chapter, you'll have a solid understanding of how to manipulate strings and utilize the string library functions effectively.

## 3.1 Understanding Strings

In C, a string is a sequence of characters terminated by the null character ('\0'). Strings are represented as arrays of characters, where each character occupies one element of the array. The last element of a string array is always the null character, indicating the end of the string.

For example, let's declare and initialize a string named message:

```
char message[] = "Hello, World!";
```

In this case, the string "Hello, World!" is stored in the message array, which automatically includes the null character at the end.

## 3.2 String Library Functions

C provides a rich set of string library functions that simplify common string manipulation tasks. These functions are declared in the <string.h> header file and can be used by including this file in your program.

Let's explore some of the most commonly used string library functions:

The strlen() function returns the length of a string by counting the number of characters in the string until it reaches the null character. Here's an example:

```
#include <string.h>


char message[] = "Hello, World!";
```

```
int length = strlen(message);
```

In this example, the strlen() function calculates the length of the string stored in message and assigns it to the variable length. The value of length will be 13, as there are 12 characters in the string "Hello, World!" plus the null character.

The strcpy() function copies the contents of one string into another. It takes two arguments: the destination string and the source string. Here's an example:

```
#include <string.h>


char source[] = "Hello";

char destination[10];


strcpy(destination, source);
```

After executing this code, the destination string will contain a copy of the source string, which is "Hello".

The strncpy() function is similar to strcpy(), but it allows you to specify the maximum number of characters to be copied. This helps prevent buffer overflow errors. For example:

```
#include <string.h>


char source[] = "Hello";

char destination[10];


strncpy(destination, source, 5);

destination[5] = '\0';  // Adding null character explicitly to terminate the string
```

In this case, only the first 5 characters of source are copied into destination, resulting in the string "Hello".

The strcat() function appends one string to the end of another. It takes two arguments: the destination string and the source string to be appended. Here's an example:

```
#include <string.h>


char destination[20] = "Hello";

char source[] = ", World!";


strcat(destination, source);
```

After executing this code, the destination string will be "Hello, World!".

Similar to strcpy(), the strncat() function allows you to specify the maximum number of characters to be appended. This helps prevent buffer overflow errors.

The strcmp() function compares two strings lexicographically and returns an integer value based on the result. It takes two arguments: the first string and the second string to compare. The return value of strcmp() indicates the relationship between the two strings. It returns 0 if the strings are equal, a negative value if the first string is lexicographically less than the second string, and a positive value if the first string is lexicographically greater than the second string.

```
#include <string.h>


char str1[] = "apple";

char str2[] = "banana";

int result = strcmp(str1, str2);
```

In this example, the strcmp() function compares the strings "apple" and "banana". The value of result will be a negative value since "apple" is lexicographically less than "banana".

The strncmp() function is similar to strcmp(), but it allows you to specify the maximum number of characters to compare. This is useful when comparing substrings or when working with strings of different lengths.

## 3.3 Additional String Library Functions

In addition to the functions mentioned above, there are many other useful string library functions available in C. Here are a few examples:

- strchr(): Searches for a specific character in a string and returns a pointer to its first occurrence.
- strstr(): Searches for a substring within a string and returns a pointer to its first occurrence.
- strtok(): Splits a string into tokens based on a delimiter and returns a pointer to the next token.
- sprintf(): Formats and stores a series of characters into a string.

## 3.4 String Manipulation Techniques

In addition to the string library functions, you can perform various string manipulation techniques using loops, conditional statements, and character-based operations. Some common techniques include:

- Looping through a string character by character and performing operations on each character.
- Changing the case of characters (e.g., converting lowercase to uppercase or vice versa).
- Reversing a string by swapping characters from the beginning and end.
- Removing or replacing specific characters within a string.

These techniques require a good understanding of loops, conditional statements, and character manipulation in C. With practice and experimentation, you'll gain confidence in manipulating strings using these techniques.