CHAPTER 6

Arrays

Introduction

This chapter will introduce readers to array and string handling, two essential topics in programming for efficiently organizing and managing data. It will begin with a clear definition of arrays, explaining how they function as collections of elements stored under a single variable name, with each element accessible by an index. The chapter will explore the different types of arrays – such as one-dimensional, two-dimensional, and multi-dimensional arrays – and explain their specific use cases and benefits in data management. Next, the chapter will focus on string handling, covering the basics of creating, accessing, and modifying strings. Key string manipulation techniques, including concatenation, slicing, searching, and formatting, will be discussed to help readers work effectively with text data. Through these topics, readers will gain the foundational skills for handling structured data and manipulating textual information within their programs.

Structure

The chapter covers the following topics:

* Definition and types of arrays
* String handling

Objectives

The objective of this chapter is to introduce readers to the fundamental concepts and applications of arrays and string handling in programming. By exploring the definition and the different types of arrays - including one-dimensional, two-dimensional, and multi-dimensional arrays - readers will learn how to efficiently store and access collections of data under a single variable name, thereby enhancing data organization and manipulation capabilities. The chapter also aims to provide a solid foundation in string handling, including essential operations such as creation, modification, concatenation, and searching within strings. These objectives are designed to equip readers with practical skills for effectively managing structured and textual data, which are critical to building robust and efficient programs.

Definition and types of arrays

An array is a data structure in programming that allows the storage of multiple values of the same type under a single variable name. Each value in an array is identified by an index or a key, which makes it easy to access and manipulate the elements. Arrays provide a way to organize data in a linear fashion, enabling efficient access to elements through their indices. They are particularly useful for handling collections of data that are related, such as lists of numbers, characters, or other objects. In C programming, arrays are defined with a specific type and size, where the size determines the number of elements the array can hold. The elements of an array are stored in contiguous memory locations, allowing for efficient data access. For example, an integer array can store multiple integers, and each element can be accessed using its index, with the first element typically starting at index 0.

An example of array declaration in C is as follows:

int numbers[5]; // Declares an integer array of size 5

In this declaration, *numbers* is the name of the array, and it can store up to five integers.

Types of arrays

Arrays can be classified into several types based on their dimensions and the nature of their elements. The most common types of arrays include:

* **One-dimensional arrays:** A one-dimensional array, commonly known as an array, is the simplest form of data structure that consists of a sequence of elements stored in a single row, where each element is identified by its unique index. It provides a straightforward way to store and manage a collection of data of the same type, such as a list of integers, characters, or floating-point numbers. The index of the array starts from 0, allowing for easy access and manipulation of the elements in a linear fashion. This type of array is useful for handling lists or sequences of related data efficiently.
* **Example:**

int grades[4] = {90, 85, 88, 92}; // A one-dimensional array of integers

In this example, *grades* is a one-dimensional array containing four integer values.

* **Multi-dimensional arrays:** Multi-dimensional arrays are arrays with more than one dimension, enabling the storage of data in a structured format like grids or tables. The most common type is the two-dimensional array, which is visualized as a matrix with rows and columns, where each element is identified by two indices, one for the row and one for the column. This type of array is particularly useful for representing tabular data, such as spreadsheets or matrices in mathematics, making it ideal for applications that require the handling of complex, organized datasets.
* **Example:**

int matrix[3][4]; // A two-dimensional array with 3 rows and 4 columns

Here, *matrix* is a two-dimensional array capable of storing integers in a grid of three rows and four columns. You can access an element using two indices: one for the row and one for the column (e.g., matrix[1][2] accesses the element in the second row and third column).

* **Dynamic arrays:** Dynamic arrays are arrays whose size can be determined during runtime rather than at compile time. This allows for greater flexibility, especially when the number of elements is not known beforehand. In C, dynamic arrays are created using pointers and dynamic memory allocation functions such as malloc, calloc, or realloc.
* **Example:**

int \*dynamicArray; int size = 5;

dynamicArray = (int \*)malloc(size \* sizeof(int)); // Allocates memory for 5 integers

In this example, a dynamic array is created to hold five integers, and the memory for it is allocated at runtime.

* **Jagged arrays:** Jagged arrays (or an *array of arrays*) are arrays where each row can have a different number of columns. Unlike multi-dimensional arrays, where each row has the same length, jagged arrays allow for variable-length rows.
* **Example:**

int \*jaggedArray[3]; // Array of three integer pointers

jaggedArray[0] = (int \*)malloc(2 \* sizeof(int)); // First row has 2 elements jaggedArray[1] = (int \*)malloc(3 \* sizeof(int)); // Second row has 3 elements

jaggedArray[2] = (int \*)malloc(1 \* sizeof(int)); // Third row has 1 element

Here, jaggedArray consists of three rows, each with a different number of integer elements, showcasing the flexibility of jagged arrays.

Arrays are a fundamental data structure in programming, providing a way to store and manage collections of related data efficiently. Understanding the different types of arrays, including one-dimensional, multi-dimensional, dynamic, and jagged arrays, allows programmers to select the most suitable array type for their specific needs. This knowledge is essential for effective data manipulation and organization in various programming tasks.

Initialization and processing an array

Initializing an array in C refers to the process of assigning values to the array elements when it is declared. There are various ways to initialize an array, either during or after declaration. If not initialized, the array will contain garbage values, except for arrays with static or global scope, which are automatically initialized to zero.

Initializing an array during declaration

You can initialize an array at the time of its declaration by listing the values inside curly braces {}. The number of elements in the list must not exceed the declared size of the array. If fewer values are provided, the remaining elements will be automatically initialized to zero.

**Example:**

int numbers[5] = {10, 20, 30, 40, 50}; // Explicitly initializes all 5 elements

In this example, the array numbers are initialized with five elements. Each value is directly assigned to the corresponding position. Alternatively, if you neglect the size but provide initialization values, the compiler will automatically determine the size based on the number of values provided.

**Example**:

int numbers[] = {10, 20, 30}; // Array of size 3

Initialization includes the following actions:

* **Partial initialization**: In the case of partial initialization, only the specified elements will be initialized, and the rest will automatically be set to zero.
* **Example**:

int numbers[5] = {10, 20}; // Initializes first two elements; others will be 0

* **Uninitialized array:** If you declare an array without initializing it, it will contain undefined (garbage) values.
* **Example:**

int numbers[5]; // Elements will have garbage values

Processing an array

Processing an array involves accessing, modifying, and performing operations on the array elements. This is done using loops since arrays are indexed, and looping constructs allow you to iterate through each element easily.

* **Accessing array elements:** You can access the elements of an array using the index of the array. The index starts from 0 and goes up to (size - 1). For instance, in an array number[5], the first element is numbers[0], and the last element is numbers[4].
* **Example**:

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

printf("%d", numbers[2]); // Outputs 30

* **Modifying array elements:** You can modify any element in the array by assigning a new value to the corresponding index.
* **Example:**

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

numbers[3] = 100; // Changes the 4th element from 40 to 100

* **Looping through an array:** To process all elements of an array, you can use looping constructs like for, while, or do-while. Typically, loops are the most commonly used for iterating through arrays.
* **Example:**

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

// Loop to print all elements for(int i = 0; i < 5; i++) {

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

}

This loop accesses each element of the array numbers using the index i, and prints the elements one by one.

Basic operations processing arrays

You can perform several operations on arrays, such as summing all the elements, finding the largest or smallest element, and more. Here is how you can process the elements to perform basic tasks:

* **Summing array elements:**

int numbers[5] = {10, 20, 30, 40, 50}; int sum = 0;

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

sum += numbers[i]; // Adds each element to sum

}

printf("Sum = %d", sum); // Outputs: Sum = 150

* **Finding maximum or minimum element:**

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

int max = numbers[0]; // Assume first element is largest for(int i = 1; i < 5; i++) {

if(numbers[i] > max) {

max = numbers[i]; // Update max if larger element is found

}

}

printf("Maximum element = %d", max); // Outputs: Maximum element = 50

In this example, the array is processed to find the largest element by comparing each element to the current maximum.

Multidimensional array initialization and processing

For multidimensional arrays, initialization and processing follow the same principles as one-dimensional arrays, but with more indices.

**Example**:

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

for(int i = 0; i < 2; i++) { for(int j = 0; j < 3; j++) {

printf("%d ", matrix[i][j]); // Accesses each element in the 2D array

}

}

This code initializes a 2D array and then processes it using nested loops to access each element. Initializing and processing arrays is a fundamental task in C programming. Arrays can be initialized in several ways, and processing them usually involves accessing elements via loops. Understanding how to initialize, modify, and process arrays efficiently allows programmers to manage collections of data more effectively in their applications.

String handling

In C, strings are arrays of characters terminated by a null character ('\0'). A string is essentially a sequence of characters stored in a contiguous block of memory. Since C does not have a dedicated string data type, strings are handled using arrays of char type. String handling in C includes tasks such as declaring strings, initializing them, manipulating them using standard library functions, and performing operations like concatenation, comparison, and length determination.

Declaring and initializing strings

You can declare and initialize strings in two ways: by specifying the size of the character array or by using string literals. The definition for both are as follows:

* **Declaration:** A string is declared as a character array:

char str[20]; // Declares a character array of size 20

* **Initialization:** Strings can be initialized using string literals or by specifying individual characters.
* **Using a string literal:**

char str[20] = "Hello, World!"; // Initializes with a string literal

* **Using individual characters:**

char str[5] = {'H', 'e', 'l', 'l', 'o', '\0'}; // Explicitly initializing each character

In the above examples, the string is automatically terminated by the null character ('\0'), which marks the end of the string in memory.

Input and output of strings

You can read and print strings using standard input/output functions like scanf, gets, printf, and puts:

* **Using scanf and printf:**

char name[50];

scanf("%s", name); // Reads a single word (without spaces) printf("Hello, %s!", name); // Prints the string

Note that scanf stops reading the string when it encounters a space. To handle strings with spaces, you can use gets (although fgets is preferred for safety reasons).

* **Using fgets and puts:**

char sentence[100];

fgets(sentence, 100, stdin); // Reads a line of input including spaces puts(sentence); // Outputs the string

Common string handling functions

C provides several standard library functions for performing operations on strings declared in the <string.h> header file. These include functions for finding the length of a string, copying, concatenating, comparing, and more:

* **strlen():** Find the length of a string. This function returns the number of characters in the string, excluding the null terminator.

char str[] = "Hello";

int len = strlen(str); // len will be 5

* **strcpy():** Copy one string to another. This **function** copies the content of one string into another.

char src[] = "Source"; char dest[20];

strcpy(dest, src); // Copies "Source" into dest

* **strcat():** Concatenate two strings.This function appends one string to the end of another.

char str1[20] = "Hello"; char str2[] = " World";

strcat(str1, str2); // str1 now contains "Hello World"

* **strcmp():** Compare two strings. This function compares two strings lexicographically. It returns:
* 0 if the strings are equal
* A positive value if the first string is greater
* A negative value if the first string is smaller

char str1[] = "Apple"; char str2[] = "Orange";

int result = strcmp(str1, str2); // Will return a negative value as "Apple" is smaller than "Orange"

* **strncpy() and strncat():** Secure versions of copying and concatenation. These functions allow you to copy/concatenate a limited number of characters, providing safer alternatives to strcpy and strcat.

char src[] = "Hello"; char dest[10];

strncpy(dest, src, 3); // Copies only "Hel" into dest

Processing strings

Strings can be processed using loops, just like arrays. Each character in the string can be accessed and manipulated using its index.

**Example**: Counting vowels in a String:

char str[] = "Hello World"; int vowels = 0;

for (int i = 0; str[i] != '\0'; i++) {

if (str[i] == 'a' || str[i] == 'e' || str[i] == 'i' || str[i] == 'o' || str[i] == 'u' ||

str[i] == 'A' || str[i] == 'E' || str[i] == 'I' || str[i] == 'O' || str[i] == 'U') { vowels++;

}

}

printf("Number of vowels: %d", vowels);

This example shows how to loop through each character in a string and perform a specific operation (counting vowels).

String manipulation example

Consider the following example that demonstrates multiple string operations:

#include <stdio.h>

#include <string.h>

int main() {

char firstName[50], lastName[50], fullName[100];

// Input first and last name

printf("Enter first name: ");

scanf("%s", firstName);

printf("Enter last name: ");

scanf("%s", lastName);

// Concatenate first and last name

strcpy(fullName, firstName);

strcat(fullName, " ");

strcat(fullName, lastName);

// Display the full name

printf("Full Name: %s\n", fullName);

// Find the length of the full name

printf("Length of Full Name: %lu\n", strlen(fullName));

return 0;

}

In this example, the user inputs a first and last name, which are then concatenated into a full name, and the length of the full name is displayed using strlen().

Conclusion

In C programming, an array is a collection of elements of the same data type stored in contiguous memory locations. Arrays are defined by specifying a data type and the number of elements. They come in different types: one-dimensional arrays (e.g., int arr[10] for a list of integers), two-dimensional arrays (e.g., int matrix[3][3] for tabular data), and multidimensional arrays for complex data structures. Arrays can be initialized at the time of declaration or assigned values during execution. Processing arrays involves iterating over each element to perform tasks like summing values, finding minimums or maximums, and searching or sorting. String handling in C uses character arrays, where a string is an array of characters terminated by a null character (\0). C provides functions in the <string.h> library, such as strcpy, strcat, strlen, and strcmp, to simplify common string operations. In the upcoming chapter, we will explore structures and unions, which allow grouping of different data types under one name for efficient handling. We'll delve into pointers, a powerful feature in C for direct memory access and manipulation, and examine their relationship with arrays, enabling dynamic data handling. The chapter also covers dynamic memory allocation, critical for optimizing memory usage in programs. We'll discuss pointers and strings, showcasing efficient string manipulation and memory handling. Moving to data files, we'll learn about opening and closing files, performing essential I/O operations on files, and techniques for reading and writing data, laying the foundation for file-based data management in applications.

Exercises

* Write a program that initializes an array of integers with values 1 to 5 and then prints these values using a loop.
* Create a program to input five numbers into an array and calculate their average by accessing each element.
* Write a program that declares a 3x3 integer array and fills it with values from the user, then displays it in a matrix form.
* Implement a function that takes an array and its size as arguments and returns the sum of its elements.
* Write a program that finds the minimum and maximum values in a given array of integers.
* Create a program that searches for a specific value in an array and outputs the index if found or a message if not found.
* Write a function to sort an array of integers in ascending order using the Bubble Sort algorithm.
* Use the strlen function to find and print the length of a user-input string.
* Create a program that takes two strings as input and concatenates them using the strcat function.
* Write a program that checks if a string is a palindrome (reads the same forwards and backwards) using array indexing.