Lab 10: Strings & 2D Arrays

*Practice with strings which are implemented as null terminated character arrays and Practice with 2D Arrays.*

# Strings

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are also called strings. Hence strings are actually one-dimensional array of characters terminated by a **null** character '\0'.

## Syntax and Initializing Strings

In C, string can be initialized in different number of ways:

|  |
| --- |
| **char c[]="abcd";** |
| **char c[5]="abcd";** |
| **char c[]={'a','b','c','d','\0'};** |
| **char c[5]={'a','b','c','d','\0'};** |

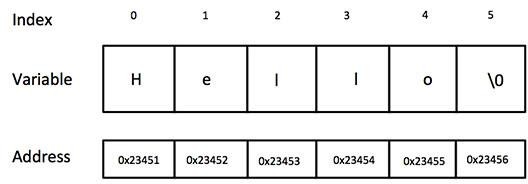
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows:

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C:



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string:

## Example 1:

#include <stdio.h> void main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; printf("Greeting message: %s\n", greeting ); getch();

}

When the above code is compiled and executed, it produces the following result:

Greeting message: Hello

## String Manipulations In C Programming Using Library Functions

Strings are often needed to be manipulated by programmer according to the need of a problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. To solve this, the C supports a large number of string handling functions.

There are numerous functions defined in "string.h" header file. Few commonly used string handling functions are discussed below:

|  |  |
| --- | --- |
| **S.N.** | **Function & Purpose** |
| **1** | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| **2** | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| **3** | **strlen(s1);**  Returns the length of string s1. |
| **4** | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |

Student Name: Roll No: Section:

The following example uses some of the above-mentioned functions:

## Example 2:

#include <stdio.h> #include <string.h> void main () {

char str1[12] = "Hello"; char str2[12] = "World"; char str3[12];

int len ;

/\* copy str1 into str3 \*/ strcpy(str3, str1);

printf("strcpy( str3, str1) : %s\n", str3 );

/\* concatenates str1 and str2 \*/ strcat( str1, str2);

printf("strcat( str1, str2): %s\n", str1 );

/\* total lenghth of str1 after concatenation \*/ len = strlen(str1);

printf("strlen(str1) : %d\n", len );

getch();

}

When the above code is compiled and executed, it produces the following result:

strcpy( str3, str1) : Hello strcat( str1, str2): HelloWorld strlen(str1) : 10

# Using puts and gets functions

Scanf( ) is not capable of receiving multi-word strings. Therefore names such as ‘Dennis Ritchie’ would be unacceptable. The way to get around this limitation is by using the function gets( ). Functions gets() and puts() are two string functions to take string input from user and display string respectively The usage of functions gets( ) and its counterpart puts( ) is shown below:

Student Name: Roll No:

#### Syntax: gets (string name); puts (string name);

Section:

Though, gets() and puts() function handle string, both these functions are defined in "stdio.h" header file.

**Example 3:**

#include<stdio.h>

void main(){

char name[30]; printf("Enter name: ");

gets(name);

//Function to read string from user.

printf("Name: ");

puts(name);

//Function to display string.

getch();

}

**Example 4:**

#include<string.h> main( ) {

char name[25] ;

printf ( "Enter your full name " ) ; gets ( name ) ;

puts ( "Hello!" ) ; puts ( name ) ;

}

**Example 5:**

#include<string.h> main( ) {

char name[25] ;

printf ( "Enter your full name " ) ; scasf ( “%s”, &name ) ;

puts ( "Hello!" ) ;

printf (“%s”, name ) ; }

**Multidimensional Arrays**

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

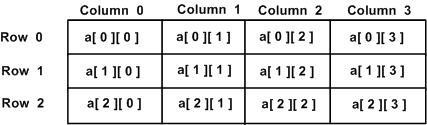
type name[size1][size2]...[sizeN];

## Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows:

type arrayName [ x ][ y ];

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows:



Thus, every element in the array **a** is identified by an element name of the form **a[ i ][ j ]**, where **'a'** is the name of the array, and **'i'** and **'j'** are the subscripts that uniquely identify each element in **'a'**.

## Syntax

The definition of a two-dimensional array takes the form:

# data\_type identifier[ r ][ c ] = init;

where ***r*** is the number of rows in the array and ***c*** is the number of columns. ***r*** and ***c*** are integer constants or constant integer expressions. The total number of elements in the array is ***r \* c***. ***init*** is a braces-enclosed, comma-separated list of initial values. The assignment operator together with ***init*** are optional. If we add an initialization list, we may omit the value

of ***r***. If ***r \* c*** exceeds the number of initial values, the compiler initializes the remaining elements to **0**. If we omit the initialization list, we must specify ***r***. We must always specify ***c***. **For example:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **int** | **a[4][5]** | **=** | **{11,** | **12,** | **13,** | **14,** | **15,** |  |
|  |  |  | **21,** | **22,** | **23,** | **24,** | **25,** |
|  |  |  | **31,** | **32,** | **33,** | **34,** | **35,** |
|  |  |  | **41,** | **42,** | **43,** | **44,** | **45};** |

To improve clarity, we may enclose each subset of initial values for each row in additional braces:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **int** | **a[4][5]** | **=** | **{{11,** | **12,** | **13,** | **14,** | **15},** |  |
|  |  |  | **{21,** | **22,** | **23,** | **24,** | **25},** |
|  |  |  | **{31,** | **32,** | **33,** | **34,** | **35},** |
|  |  |  | **{41,** | **42,** | **43,** | **44,** | **45}};** |

## Initializing Two-Dimensional Arrays

We have divided the concept into two different types:



Initializing

2D Array

Row wise

Assignment

Combine

Assignment

## Method 1: Initializing all Elements Row-wise

For initializing 2D Array we can need to assign values to each element of an array using the below syntax.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| int a[3][2] = { | { | 1 | , | 4 | }, |
|  | { | 5 | , | 2 | }, |
| }; | { | 6 | , | 5 | } |

Consider the below program:

## Example 6:

6

#include<stdio.h>

void main() { int i, j;

int a[3][2] = { { 1, 4 },

{ 5, 2 },

{ 6, 5 }};

**for** (i = 0; i < 3; i++) {

**for** (j = 0; j < 2; j++) { printf("%d ", a[i][j]);

}

printf("\n");

}

**getch()**;

}

#### Output :

|  |  |
| --- | --- |
| 1 | 4 |
| 5 | 2 |
| 6 | 5 |

We have declared an array of size 3 X 2, It contain overall 6 elements.

Row 1 : { 1 , 4 },

Row 2 : { 5 , 2 },

Row 3 : { 6 , 5 }

We have initialized each row independently

a[0][0] = 1

a[0][1] = 4

**Method 2 : Combine and Initializing 2D Array**

Initialize all Array elements but initialization is much straight forward. All values are assigned sequentially and row-wise

int a[3][2] = {1 , 4 , 5 , 2 , 6 , 5 };

Consider the below example program:

## Example7 :

#include <stdio.h>

void main() { int i, j;

int a[3][2] = { 1, 4, 5, 2, 6, 5 };

**for** (i = 0; i < 3; i++) {

**for** (j = 0; j < 2; j++) { printf("%d ", a[i][j]);

}

printf("\n");

}

**getch()**;

}

Output will be same as that of the one above it.

## Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

int val = a[2][3];

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array :

## Example 8:

#include <stdio.h> void main () {

/\* an array with 5 rows and 2 columns\*/

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

int i, j;

/\* output each array element's value \*/ for ( i = 0; i < 5; i++ ) {

for ( j = 0; j < 2; j++ ) {

printf("a[%d][%d] = %d\n", i,j, a[i][j] );

} } getch();}

When the above code is compiled and executed, it produces the following result:

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

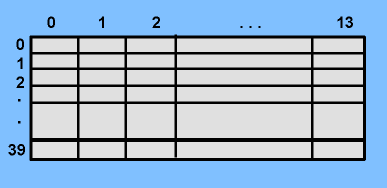
a[3][0]: 3

a[3][1]: 6

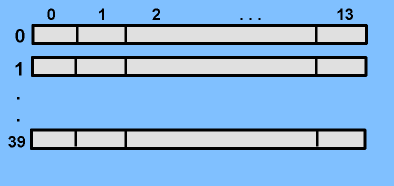
a[4][0]: 4

a[4][1]: 8

### Manipulation of a two-dimensional array requires the manipulation of two indices. When the two-dimensional array *labScores* is declared, enough storage is set aside to hold a table containing 40 rows and 14 columns for a total of 40 \* 14 = 560 integer values. To access one particular value, we must specify the row and column. The row index ranges from 0 to MAX\_STUDENTS-1 (39) and the column index ranges from 0 to MAX\_LABS-1 (13). Thus the table can be visualized as:



This two-dimensional array may also be visualized as a one-dimensional array of arrays. An alternative view of



this array would be like the above diagram.

This two-dimensional array may be viewed as a one-dimensional array having 40 elements where each element is an array of 14 values.

In our *labScores* example, suppose we wish to indicate that the second student (corresponding to row 1) made a 90 on lab 10 (corresponding to column 9). We might use the statement:

**labScores [1][9] = 90;**

### Array indices may be integer constants (as in the above example), variables, or expressions. They should be within the bounds of the array.

:

Roll No:

Section:

**Student Tasks:**

#### Task 1:

Write a program that accepts one string and then find the length of that string and print it.

#### Task 2:

Write a program that takes input in a string (char array) and ‘find’ a particular character and ‘replace’ that character. The program should replace all the occurrences of that particular character with the new character.

For example:

Array: “I love programming” find character: ‘m’

replace with character: ‘n’

then the string should become “I love progranning”

#### Task 3:

Fifteen numbers are entered from the keyboard into an array with 3 rows and 5 columns. The number to be searched is entered through the keyboard by the user. Write a program to find if the number to be searched is present in the array and if it is present, display the number of times it appears in the array.

#### Task 4:

Write a program to pick up the largest number and sum of all the integers from any 5 x 5 matrix.

#### Task 5:

Write a program to obtain transpose of a 4 x 4 matrix. The transpose of a matrix is obtained by exchanging the elements of each row with the elements of the corresponding column

#### Task 6:

Write a program to add two 6 x 6 matrices. Add result in third array and print this array.

#### Task 7:

Write a program to multiply any two 3 x 3 matrices. Add result in third array and print this array.

#### Task 8:

In a bitmap file black is represented by 1 and white by 0. Create a 8X8 pattern and initialize using following data: 1 1 1 0 0 1 1 1

1 0 0 0 0 0 1 0

1 1 1 1 1 1 0 0

1 1 1 0 0 0 1 0

1 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1

0 0 0 0 1 1 1 1

11