**ARRAY**

**Definition**:

Recall that a variable is an identifier that is used to represent a single data item. The data item must be assigned to the variable at some point in the program which could be changed to different data items at various places within the program. But the data type associated with the variable cannot change.

The ***array*** is another kind of variable. An array is an identifier that refers to a ***collection*** of data items using the same name. The data items must all be of the same type (e.g., all integers, all characters, etc.). The individual data items are represented by their corresponding ***array-elements.***

**Example:**

Using variables to hold the values of temperature readings of a room for week requires creating seven different variable names, each representing each day. Your first option is to declare seven different variables of the same type as follows:

***float t1, t2, t3, t4, t5, t6, t7****;*

Note that the data items may have different readings but the data collected are of the same type.

There are difficulties working with the above declaration, for example, you cannot use a loop construct to manipulate the content of variables. This becomes convenient when you introduce the use of an array.

**Declaring an array variable:**

*The general format of declaring an array variable in C language is as follows:*

datatype arrayRefVar[arraySize];

Example*:*

*float t[7] ; // Variable* ***t*** *(a 7-element array)**holds a reference to the array elements.*

*//This declaration creates space in memory for an array of 7 elements with each*

*//of type float*

Note that the size of the array or the number of elements must be known and set before you can start using the array. It makes sense to allocate more space (which may not be used) than needed if you are not sure of the number of elements to be processed because once the size is set, it cannot be changed.

**Array Index:**

Items of an array could be described as *objects* or *elements* of the array and are numbered consecutively 0, 1, 2, 3,…These numbers are called *index values* or *subscripts* of the array. The collection of objects/elements is given one variable name such as ***t*** (or ***temperature***), while each is seen as a with subscript 0, 1, 2, 3,… and generally expressed as follows:

t = {*t*0, t1, t2, … *tn-1*}

The subscripts locate the element’s position within the array, thereby giving *direct access* into the array. If the name of the array is **t**, then t[0] is the name of the element that is in position 0 (the first element), t[1] is the name of the element that is in position 1 (the second element), *etc.* The subscript associated with each element is shown in square braces.For *an* n-element array, the subscripts always range from *0* to n-1.

Furthermore, an array is seen as a series of adjacent storage compartments that are numbered by their index values. The diagram actually represents a region of the computer’s memory because an array is always stored this way with its elements in a contiguous sequence.

|  |  |
| --- | --- |
| float t[7]; | |
| t[0] | ***31.0*** |
| t[1] | ***33.4*** |
| t[2] | ***32.5*** |
| t[3] | ***39.0*** |
| t[4] | ***37.5*** |
| t[5] | ***35.0*** |
| t[6] | ***33.4*** |

Fourth Array Element; at index 3

Element value

Fig. An array is seen as a series of adjacent storage compartments in the memory

A good analogy is to say that an array name such as **‘t’** above is like a street name while array index represents each house number which uniquely identifies or locates each house. Array indices or *subscripts* are usually contiguous positive integers and C language usually starts indexing from zero (0).

t

0

1

2

3

4

5

Fig. Array analogy

Therefore:

t[0] implies the first element

t[6] implies the seventh / last element

t[7] is invalid; there is no such element

If all elements are known before hand, declaration, initialization and allocation can be done as follows with each element separated by comma:

***float t[7 ] = {31.0, 33.4, 32.5, 39.0, 37.5, 35.0, 33.4 } or***

***float t[ ] = {31.0, 33.4, 32.5, 39.0, 37.5, 35.0, 33.4 }*** *// note that there is no size specification, //since the elements are known and assigned from start, the number of elements is automatically //known by this declaration*

Array could be categorized as e.g. integer arrays, character arrays, etc or one-dimensional, arrays, multi-dimensional arrays.

**char-type array declaration and use:**

char-type array (often called a one-dimensional ***character*** array). This type of array is generally used to represent a string. Since the array is one-dimensional, there will be a single ***subscript /*** ***index*** whose value refers to individual array elements. Note that an n-character string will require an **(n+1**)-element array, because of the null character **(\0)**that is automatically placed at the end of the string.

Suppose that the string **"Babcock"** is to be stored in a one-dimensional character array called

**school .** Since **"Babcock"** contains 7 characters, **school** will be an 8-element array. The following representation happens:

school[0] represents the letter B

school[1] represents the letter a…

school[7] represents the null character signifying the end of the string(last or the 8th element).

Table: showing array school description

|  |  |  |  |
| --- | --- | --- | --- |
| ***Element Number*** | ***Subscript value*** | ***Array Element***  ***(as indexed)*** | ***Corresponding Data Item (String Character)*** |
| 1 | 0 | school[0] | B |
| 2 | 1 | school[1] | a |
| 3 | 2 | school[2] | b |
| 4 | 3 | school[3] | c |
| 5 | 4 | school[4] | o |
| 6 | 5 | school[5] | c |
| 7 | 6 | school[6] | k |
| 8 | 7 | school[7] | \0 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | B | a | b | c | o | c | k | \0 |
| Subscript: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Fig. An 8-element character array

A character-type array can be initialized within a declaration as follows:

***char school[8] = “Babcock”;***

Or without an explicit size specification (the square brackets are empty):

***char school[ ] = "Babcock";***

Both declarations cause ***school*** to be an 8-element character array. The first 7 elements will represent the 7 characters of the word ***Babcock***, and the 8th element represents the null character (\0) which is automatically added at the end of the string.

Similarly, the following is acceptable:

char zone[] = {'N', 'O', 'R', 'T', 'H'};

char zone[] = "NORTH";

This statement is equivalent to the preceding statement, except that C adds the character '\0', called the *null terminator*, to indicate the end of the string, as shown below. Recall that a character that begins with the back slash symbol (\) is an escape character.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 'N' | 'O' | 'R' | 'T' | 'H' | '\0' |
| zone[0] | zone[1] | zone[2] | zone[3] | zone[4] | zone[5] |

*Fig: Arrangement of characters on array or list memory*

The following are acceptable one-dimensional array declarations:

* float t[]= {1.0, 3.4, 2.5, 9.0, 7.5, 5.0, 4.0 };
* char color[3] = { ‘R’, 'E', ‘D' } ;
* char color[4] = "RED”;
* char color[] = "RED";
* int digits[10] = {1, 2, 3, 4, *5 , 6, 7, 8, 9, 10};*
* int digits[] = {1, 2, 3, 4, *5 , 6, 7, 8, 9, 10};*

*Spot the difference especially between the 2nd and the 3rd.*

Note that for one-dimensional array, there will be a single *subscript / index* whosevalue refers to individual array elements.

Example using array to store string:

Consider storing string “LAGOS ” in a one-dimensional character array named state. LAGOS will be stored in 6 element array as state[6]:

char state[ ] = “LAGOS”; //acceptable

char state[6 ] = “LAGOS”; // acceptable

char state[5] = { 'L', 'A', 'G', 'O', 'S'} ;

char state[5] = “LAGOS”;//this truncates the end of string character

char state[20 ] = “LAGOS”;// too large waste space, extra spaces may be filled with zeros or meaningless characters

**Using Array:**

**Example 1**

*/ \* converting word from lowercase text to uppercase \*/*

*#include <stdio.h>*

*#include <ctype.h>*

*#define SIZE 80*

*void main ( )*

*{*

*char word[SIZE];*

*i n t count ;*

*// read in text*

*f o r (count = 0; count < SIZE; ++count)*

*word[count] = getchar();*

*// display the text in upper case*

*f o r (count = 0; count < SIZE; ++count)*

*putchar(toupper(word[count]));*

*}*

**Example 2**

**//**Summing All Elements

//Use a variable named total to store the sum. Initially total is 0. Add each element in the //array to total using a loop as follows:

int myList[4] = {1, 2, 3, 4};  
double total = 0;

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

{

total = total + myList[i];

}

**Example 3**

Consider the following initialization:

float myList[4] = {1.9, 2.9, 3.4, 3.5};

This shorthand notation is equivalent to the following statements:

float myList[4];

myList[0] = 1.9;

myList[1] = 2.9;

myList[2] = 3.4;

myList[3] = 3.5;

**Example 4**

Consider the following array definitions and their interpretations.

**int digits[6]** = **{3, 4, 3};**

**float x[5]** = **{-0.3, 0** , **0.25};**

Table: showing array element by subscripts; the elements without values are assigned automatic values of zeros.

|  |  |
| --- | --- |
| **digits[0] = 3** | **x[0] = -0.3** |
| **digits[1] = 4** | **x[1] = 0** |
| **digits[2] = 3** | **x[2] = 0.25** |
| **digits[3] = 0** | **x[3] = 0** |
| **digits[4] = 0** | **x[4] = 0** |
| **digits[5] = 0** |  |

**Example 5**

/\* calculate the average of n numbers,

then compute the deviation of each number about the average \*/

#include <stdio.h>

void main ( )

{

int n , count;

float avg, sum = 0;

float list[100] ;

/\* read a value for n \*/

printf ( “\nHow many numbers do you want to average? ') ;

scanf ("%d", &n) ;

/\* read in the numbers and calculate their sum \*/

for (count = 0; count < n; ++count) {

printf ("i= %d x = ", count + 1);

scanf ( "%f ”, &list[count ] ) ;

sum += list[count];

}

/\* calculate and display the average \*/

avg = sum / n;

printf("\nThe average is %5.2f\n\n", avg);

**}**

/\* calculate and display the deviations about the average \*/

f o r (count = 0; count < n; ++count) {

d = list[count] - avg;

printf( " i = %d x = %5.2f d = %5.2f\n”, count + 1, list [count], d);

**}**

**Passing an Array To a Function**

An entire array can be passed to a function as an argument in a way different from how a variable is passed. To pass an array to a function, the array name must appear by itself, without brackets or subscripts, **as** an actual argument within the function call. It must be declared as **an** array within the formal argument declarations. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets showing its an array. The size of the array is not specified within the formal argument declaration:

float average(int a, float **x[ ] ) ;** / \* function prototype \* /

main( )

{

int n; /\* variable declaration\*/

float avg ; / \* variable declaration \*/

float list[100]; /\* array definition \*/

. . . . .

avg = average(n, list) ;// array passed to function average; brackets or subscripts not required

. . . . .

**}**

**float average(int a, float x [ ])** /\* **function definition** \*/

{

……

}

Note that the function prototype could have been written without argument names, **as**

**float average(int, float[]);** /\* **function declaration** \*/

Arguments are passed to a function ***by value*** when the arguments are ordinary variables but when an array is passed to a function, however, the values of the array elements ***are not*** passed to the function. Rather, the array name is interpreted **as** the ***address*** of the first array element (i.e., the address of the memory location containing the first array element). This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a ***pointer*** to the first array element. Therefore, arguments that are passed in this manner are said to be passed ***by reference*** rather than ***by value***. So, ***if an array element is altered within the function, the alteration will be recognized in the calling portion of the program*** (actually, throughout the entire scope **of** the array). That is the change made is permanent. But for pass by value only a copy of the data is sent to the function and any change within that function does not affect the original copy where the data item is created.

**Example 1:**

#include <stdio.h>

void modify(int a[]); /\* function prototype \* /

main ( )

{

int count, a[3]; /\* array definition \*/

printf(“\nFrom main, before calling the function:\n”);

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

a[count] = count + 1;

printf(“a[%d] = %d\n”, count, a[count]);

}

modify(a);

printf('\nFrom main, after calling the function:\n");

for (count = 0; count <=2; ++count)

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

}

void modify (int a[ ] ) /\* function definition\*/

{

i n t count;

printf("\nFrom the function, a f t e r modifying the values:\n”);

f o r (count = 0; count <= 2; ++count) {

a[count] = -9;

p r i n t f ( " a [ % d ] = %d\n", count, a[count]);

**}**

}

From main, before calling the function:

a[0] = 1

a [1] = **2**

a[2] = **3**

From the function, after modifying the values:

a[0] = -9

a [l] = -9

a[2] = -9

From main, after calling the function:

a[0] = -9

a [l] = -9

a[2] = -9

**Example 2:**

**C**onsider a variation of the previous program. The present program includes the use of a

global variable, and the transfer of both a local variable and an array to the function.

#include <stdio.h>

int a = 1; /\* global variable \*/

void modify ( int b, int c[ ]) ; /\* function prototype \*/

main( )

{

int b = 2; /\* local variable *\*I*

int count, c[ 3 ] ; /\* array d e f i n i t i o n \*/

p r i n t f ( " \ n F r o m main, before c a l l i n g the f u n c t i o n : \ n " ) ;

p r i n t f ("a = %d b = %d\n", a, b);

f o r (count = 0; count <= **2;** ++count) {

c[count] = 10 \* (count + 1);

p r i n t f ("c[%d] = %d\n", count, c[count]);

modify(b, c); /\* function access *\*/*

printf("\nFrom main, a f t e r c a l l i n g the function:\n'));

p r i n t f ("a = %d b = %d\n", a, b);

f o r (count = 0; count <=2; ++count)

printf(“c[%d] = %d\n”, count, c[count]);

}

void modify (int **b,** int c[ ]) /\* function d e f i n i t i o n \*/

{

int count;

printf("\nFrom the function, after modifying the values:\n');

a = -999;

**b** = **-999;**

p r i n t f ("a = %d b = %d\n”, a, b);

f o r (count = 0; count <= 2; ++count) {

c[count] = -9;

p r i n t f ("c[%d] = %d\n', count, c[count]);

return;

}

**When the program is executed, the following output is generated.**

From main, before c a l l i n g the function:

a = l b = 2

c[0] = 10

c [1] = 20

c[2] = 30

From the function, after modifying the values:

a = -999 b = -999

c[0] = -9

**c [1]** = -9

c[2] = -9

From main, a f t e r c a l l i n g the function:

a = -999 b = 2

c[0] = -9

c [0] = -9

c[2] = -9

We now see that the value of **a** and the elements of **c** are altered within main as a result of the changes that were made in **modify**. However, the change made to **b** is confined to the function, as expected. The ability to alter an **array** globally within a function provides a convenient mechanism for moving multiple data items back and forth between the function and the calling portion of the program. Simply pass the array to the function and then alter its elements within the function. Or, if the original array must be preserved, copy the array (element-by-element) within the calling portion of the program, pass the copy to the function, and perform the alterations. You should exercise some caution in altering an array within a function, however, since it is very easy to unintentionally alter the array outside of the function.

**Review Questions**

1. What conditions must be satisfied by all of the elements of any given array?
2. When passing an array to a function, how must the array argument be written? How is the corresponding formal argument written?
3. When passing an argument to a function, what is the difference between passing by value and passing by reference?
4. If an array is passed to a function and several of its elements are altered within the function, are these changes recognized in the calling portion of the program
5. Describe the array that is defined in each of the following statements. Indicate what values are assigned to the individual array elements:

float c[8] = {2., 5., 3., -4., 12., 12., O., 8.);

float c[8] = {2., 5., 3., -4.);

int z[12] = (0, 0, *8,* 0, 0, 6);

char flag[4] = { ' T ' , ' R ‘ , ' U ' , 'E'};

char flag[5]= { ' T ' , ' R ‘ , ' U ' , 'E'};

char flag[] = "TRUE';

1. Write an appropriate array definition for each of the following problem situations.
2. Define a one-dimensional, 12-element integer array called c. Assign the values 1, 4, 7, 10, . . . ,34 to the array elements.
3. Define a one-dimensional character array called point. Assign the string "NORTH" to the array elements. End the string with the null character.
4. Describe the output generated by the following programs

#include <stdio.h>

main ( )

{

int a, b = 0;

int c[10] = (1, 2, 3, 4, *5 ,* 6, 7, *8,* 9, *0);*

for (a = 0; a < 10; ++a)

b += c[a];

printf ( "%d", b) ;

}

1. Write a C program that will enter a line of text, store it in an array and then display it backwards. Allow the length of the line to be unspecified (terminated by pressing the Enter key), but assume that it will not exceed 80 characters.

Test the program with any line of text of your own choosing.

Note: Array passing operates by reference, therefore any change made by the receiving method is permanent- that is global; beyond the scope of the method.

**READING AND WRITING LINE OF TEXT**

***Using scanf and printf:***

Functions as **scanf** and **printf** can be used to read in a line of text and then write it back out, just **as** it was entered. This can work for a line of text that contains a variety of characters, including whitespace characters. The string will be entered from the keyboard, and will terminate with a newline character (i.e., the string will end when the user presses the **Enter** key).

//program to demonstrate the reading and writing of a line of text

#include <stdio.h>

main( ) /\* read and w r i t e a l i n e o f t e x t \*/

{

char line[80] ;

scanf(" % [ ^ \ n ] " , line) ;

printf ( **"%s"**, line ) ;

**}**

Example of such input string is:

*Eureka, I have found it!*

***Using gets and puts functions:***

We can alternatively use functions gets and puts to achieve the same task as above.

The **gets** and **puts** functions offer simple alternative to the use of **scanf** and **printf** for reading and displaying strings, as illustrated below.

On the other hand, the **scanf** and **printf** functions in the earlier program can be expanded to include additional data items, whereas the present program cannot.

//using gets and puts functions

#include <stdio.h>

main( ) // read and write a line of text

{

char line[80] ;

gets(1ine);

puts(1ine);

**}**

**Further Example:**

Write a simple C program that can allow user to input his/her name from the keyboard with the aid of an array of character. The program should be able to:

1. Ask for the name
2. Allow input of name
3. Output the name

Hint: The program should output the firstname and lastname and there must be a space in-between.

**ANSWER** *//Program to input and print my name*

*#include<stdio.h>*

*int main()*

*{*

*char name[14];*

*printf("Enter your name ");*

*gets(name);*

*printf("My name is ");*

*puts(name);*

*return 0;*

*}*

**FURTHER READING**

Read on your own about Pointers