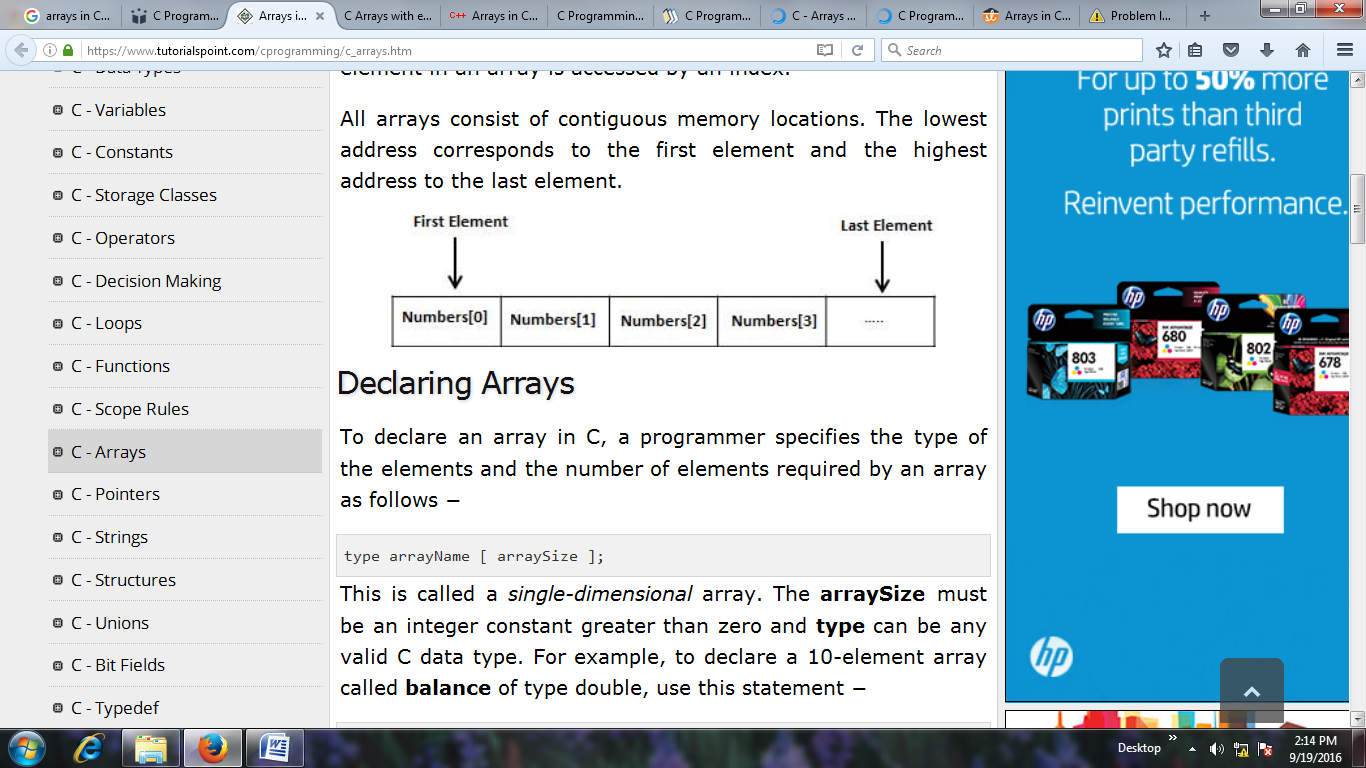
**ARRAYS**

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, technically speaking we can define an array as a collection of variables of the same type.

In C programming, one of the frequently arising problems is to handle similar types of data. For example: If the user wants to store marks of 100 students. This can be done by creating 100 variables individually but, this process is rather tedious and impractical. This type of problem can be handled in C programming using arrays.

An array is a sequence of data item of homogeneous value (same type).



C Array is a collection of variables belongings to the same data type. You can store group of data of same data type in an array.

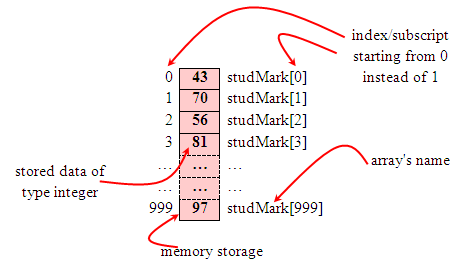
* Array might be belonging to any of the data types
* Array size must be a constant value.
* Always, Contiguous (adjacent) memory locations are used to store array elements in memory.
* It is a best practice to initialize an array to zero or null while declaring.

**Example:**

* By using an array, we just declare like this,

int studMark[1000];

* This will reserve 1000 contiguous memory locations for storing the students’ marks.
* Graphically, this can be depicted as in the following figure.



**Other Examples:**

* Examples of the one-dimensional array declarations,

int      xNum[20], yNum[50];

float    fPrice[10];

char     chLetter[70];

* The first example declares two arrays named xNum and yNum of type int.  Array xNum can store up to 20 integer numbers while yNum can store up to 50 numbers.
* The second line declares the array fPrice of type float.  It can store up to 10 floating-point values.
* fYield is basic variable which shows array type can be declared together with basic type provided the type is similar.
* The third line declares the array chLetter of type char.  It can store a string up to 69 characters.
* Why 69 instead of 70? Remember, a string has a null terminating character (\0) at the end, so we must reserve for it.

**Arrays are of two types:**

1. One dimensional array
2. Multi dimensional array
   * Two dimensional array
   * Three dimensional array
   * four dimensional array etc…

**Declaration of one-dimensional array**

data\_type array\_name[array\_size];

**For example**:

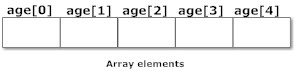
int age[5];

Here, the name of array is *age*. The size of array is 5,i.e., there are 5 items(elements) of array *age*. All element in an array are of the same type (int, in this case).

**Array elements**

Size of array defines the number of elements in an array. Each element of array can be accessed and used by user according to the need of program. For example:

int age[5];



Note that, the first element is numbered 0 and so  on.

Here, the size of array *age* is 5 times the size of int because there are 5 elements.

Suppose, the starting address of age[0] is 2120d and the size of int be 4 bytes. Then, the next address (address of a[1]) will be 2124d, address of a[2] will be 2128d and so on.

**Initialization of one-dimensional array**

Arrays can be initialized at declaration time in  this source code as:

int age[5]={2,4,34,3,4};

It is not necessary to define the size of arrays during initialization.

int age[]={2,4,34,3,4};

In this case, the compiler determines the size of array by calculating the number of elements of an array.



**Accessing array elements**

In C programming, arrays can be accessed and treated like variables in C.

**Examples:**

scanf("%d",&age[2]);

/\* statement to insert value in the third element of array age[]. \*/

scanf("%d",&age[i]);

/\* Statement to insert value in (i+1)th element of array age[]. \*/

/\* Because, the first element of array is age[0], second is age[1], ith is age[i-1] and (i+1)th is age[i]. \*/

printf("%d",age[0]);

/\* statement to print first element of an array. \*/

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

/\* statement to print (i+1)th element of an array. \*/

**A Sample Program**

**Example 1:**

/\* C program to accept and display elements of arrays \*/

#include <stdio.h>

void main()

{

int marks[5],i;

printf("Enter the marks of 5 students: ");

for(i=0;i<n;i++)

scanf("%d",&marks[i]);

printf(“\The marks which you have entered are:”);

for(i=0;i<n;i++)

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

}

**Example 2:**

/\* C program to find the sum of marks of n students using arrays \*/

#include <stdio.h>

void main(){

int marks[10],i,n,sum=0;

printf("Enter number of students: ");

scanf("%d",&n);

for(i=0;i<n; i++)

{

printf("Enter marks of student%d: ",i+1);

scanf("%d",&marks[i]);

sum+=marks[i]; //this means sum=sum+marks[i]

}

printf("Sum= %d",sum);

}

**Output**

Enter number of students: 3

Enter marks of student1: 12

Enter marks of student2: 31

Enter marks of student3: 2

sum=45

**Important thing to remember in C arrays**

Suppose, you declared the array of 10 students. For example: arr[10]. You can use array members from arr[0] to arr[9]. But, what if you want to use element arr[10], arr[13] etc. Compiler may not show error using these elements but, may cause fatal error during program execution.

Array can also be a collection of sequence of characters. Whenever array contains characters/strings as its elements, it is called as String Array.

Example for C string:

* char string[20] = {‘f’, ’r’, ‘e’, ‘s’, ‘h’, ‘2’, ‘r’, ‘e’, ‘f’, ’r’, ‘e’, ‘s’, ‘h’, ‘\0’};

OR

* char string[20] = “fresh2refresh”;

OR

* char string []    = “fresh2refresh”;

Difference between above declarations are, when we declare char as “string[20]”, 20 bytes of memory space is allocated for holding the string value. When we declare char as “string[]”, memory space will be allocated as per the requirement during execution of the program.

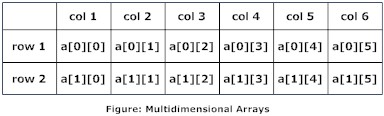
**Multidimensional Array**

C programming language allows programmer to create arrays of arrays known as multidimensional arrays. For example:

float a[2][6];

Here, a is an array of two dimension, which is an example of multidimensional array.

For better understanding of multidimensional arrays, array elements of above example can be thinked of as below:



**Initializing a 2-Dimensional Array**

How do we initialize a two-dimensional array? As simple as this...

int stud[4][2] = { { 1234, 56 }, { 1212, 33 }, { 1434, 80 }, { 1312, 78 } } ;

or even this would work...

int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;

of course with a corresponding loss in readability.

It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations,

int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;

int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;

are perfectly acceptable,

whereas,

int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ; int arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;

would never work.

**Initialization Of three-dimensional Array**

double cprogram[3][2][4]={

{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},

{{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},

{{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}

};

Suppose there is a multidimensional array arr[i][j][k][m]. Then this array can hold i\*j\*k\*m numbers of data.

Similarly, the array of any dimension can be initialized in C programming.

### Example of Multidimensional Array In C

**/\*Write a C program to find sum of two matrix of order 2\*2 using multidimensional arrays where, elements of matrix are entered by user.\*/**

#include <stdio.h>

void main(){

float a[2][2], b[2][2], c[2][2];

int i,j;

printf("Enter the elements of 1st matrix\n");

/\* Reading two dimensional Array with the help of two for loop. If there was an array of 'n' dimension, 'n' numbers of loops are needed for inserting data to array.\*/

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

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

printf("Enter a%d%d: ",i+1,j+1);

scanf("%f",&a[i][j]);

}

printf("Enter the elements of 2nd matrix\n");

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

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

printf("Enter b%d%d: ",i+1,j+1);

scanf("%f",&b[i][j]);

}

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

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

/\* Writing the elements of multidimensional array using loop. \*/

c[i][j]=a[i][j]+b[i][j]; /\* Sum of corresponding elements of two arrays. \*/

}

printf("\nSum Of Matrix:");

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

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

printf("%.1f\t",c[i][j]);

if(j==1) /\* To display matrix sum in order. \*/

printf("\n");

}

}

**Output:**

Enter the elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter the elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2 0.5

-0.9 25.0

**Pointers and 2-Dimensional Arrays**

The C language embodies an unusual but powerful capability—it can treat parts of arrays as arrays. More specifically, each row of a two-dimensional array can be thought of as a one-dimensional array. This is a very important fact if we wish to access array elements of a two-dimensional array using pointers.

Thus, the declaration,

int s[5][2] ;

can be thought of as setting up an array of 5 elements, each of which is a one-dimensional array containing 2 integers. We refer to an element of a one-dimensional array using a single subscript.

Similarly, if we can imagine **s** to be a one-dimensional array then we can refer to its zeroth element as **s[0]**, the next element as **s[1]** and so on. More specifically, **s[0]** gives the address of the zeroth one-dimensional array, **s[1]** gives the address of the first one-dimensional array and so on. This fact can be demonstrated by the following program.

**Example:**

/\* Demo: 2-D array is an array of arrays \*/ \

void main( )

{

int s[4][2] = { { 1234, 56 }, { 1212, 33 }, { 1434, 80 }, { 1312, 78 } } ;

int i ;

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

printf ( "\nAddress of %d th 1-D array = %u", i, s[i] );

}

And here is the output...

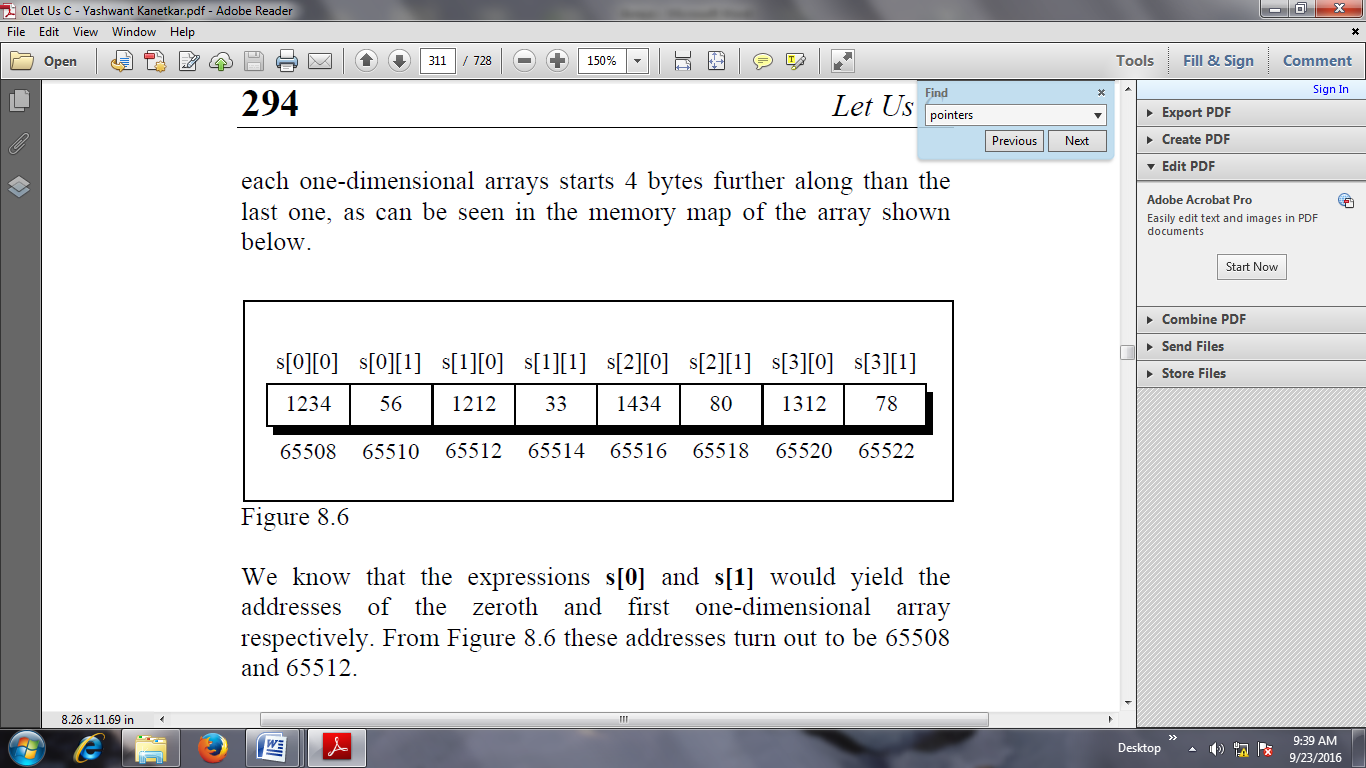
Address of 0 th 1-D array = 65508

Address of 1 th 1-D array = 65512

Address of 2 th 1-D array = 65516

Address of 3 th 1-D array = 65520

Let’s figure out how the program works. The compiler knows that **s** is an array containing 4 one-dimensional arrays, each containing 2 integers. Each one-dimensional array occupies 4 bytes (two bytes for each integer). These one-dimensional arrays are placed linearly (zeroth 1-D array followed by first 1-D array, etc.). Hence each one-dimensional arrays starts 4 bytes further along than the last one, as can be seen in the memory map of the array shown below.



We know that the expressions **s[0]** and **s[1]** would yield the addresses of the zeroth and first one-dimensional array respectively. From Figure 8.6 these addresses turn out to be 65508 and 65512.

Now, we have been able to reach each one-dimensional array. What remains is to be able to refer to individual elements of a one-dimensional array. Suppose we want to refer to the element **s[2][1]** using pointers.

We know (from the above program) that **s[2]** would give the address 65516, the address of the second one-dimensional array. Obviously ( 65516 + 1 ) would give the address 65518. Or **( s[2] + 1 )** would give the address 65518. And the value at this address can be obtained by using the value at address operator, saying **\*( s[2] + 1 )**.

But, we have already studied while learning one-dimensional arrays that **num[i]** is same as **\*( num + i )**. Similarly, **\*( s[2] + 1 )** is same as, **\*( \*( s + 2 ) + 1 )**.

Thus, all the following expressions refer to the same element,

s[2][1] OR

\* ( s[2] + 1 ) OR

\* ( \* ( s + 2 ) + 1 )

Using these concepts the following program prints out each element of a two-dimensional array using pointer notation.

**Example:**

/\* Pointer notation to access 2-D array elements \*/

main( )

{

int s[4][2] = { { 1234, 56 }, { 1212, 33 }, { 1434, 80 }, { 1312, 78 } } ;

int i, j ;

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

{

printf ( "\n" ) ;

for ( j = 0 ; j <= 1 ; j++ )

printf ( "%d ", \*( \*( s + i ) + j ) ) ;

}

}

And here is the output...

1234 56

1212 33

1434 80 1312 78

**Passing Array to function**

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

## Way-1

Formal parameters as a sized array −

void myFunction(int param[10]) {

.

.

.

}

**Example:**

#include<stdio.h>

#include<conio.h>

**void** modify(**int** b[3]);

**void** main()

{

**int** arr[3] = {1,2,3};

modify(arr);

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

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

getch();

}

**void** modify(**int** a[3])

{

**int** i;

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

a[i] = a[i]\*a[i];

}

**Output :**

1 4 9

## Way-2

Formal parameters as an unsized array −

void myFunction(int param[]) {

.

.

.

}

## Example:

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows −

#include <stdio.h>

float average(float a[]);

int main(){

float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};

avg=average(c);

printf("Average age=%.2f",avg);

return 0;

}

float average(float a[]){

int i;

float avg, sum=0.0;

for(i=0;i<6;++i){

sum+=a[i];

}

avg =(sum/6);

return avg;

}

**Output**

4

## Way-3

Formal parameters as a pointer −

void myFunction(int \*param) {

.

.

.

}

## Example:

#include <stdio.h>

/\* function declaration \*/

double getAverage(int \*arr, int size);

void main ()

{

/\* an int array with 5 elements \*/

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

/\* pass pointer to the array as an argument \*/

avg = getAverage( balance, 5 ) ;

/\* output the returned value \*/

printf("Average value is: %f\n", avg );

}

double getAverage(int \*arr, int size)

{ int i, sum = 0;

double avg;

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

sum += arr[i];

}

avg = (double)sum / size;

return avg;

}

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

Average value is: 214.40000

**Program to pass a single element of an array to function**

#include <stdio.h>

void display(int a)

{

printf("%d",a);

}

void main(){

int c[]={2,3,4};

display(c[2]); //Passing array element c[2] only.

}

## Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

### Example:

#include<stdio.h>

void Function(int c[2][2]);

void main()

{

int c[2][2],i,j;

printf("Enter 4 numbers:\n");

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

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

scanf("%d",&c[i][j]);

}

Function(c); /\* passing multi-dimensional array to function \*/

}

void Function(int c[2][2]){

/\* Instead to above line, void Function(int c[][2]){ is also valid \*/

int i,j;

printf("Displaying:\n");

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

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

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

}

**Output**

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5

# Returning Array from a Function

We can return a pointer to the base address(address of first element of array) of array from a function like any basic data type. However, C programming language does not allow to return whole array from a function.  
  
We should not return base pointer of a local array declared inside a function because as soon as control returns from a function all local variables gets destroyed. If we want to return a local array then we should declare it as a static variable so that it retains it's value after function call.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example −

int \* myFunction() {

.

.

.

}

**Example:**

|  |
| --- |
| #include <stdio.h>  #include <conio.h>    /\* This function returns an array of N even numbers \*/  int\* getEvenNumbers(int N)  {      /\* Declaration of a static local integer array \*/      static int evenNumberArray[20];      int i, even = 2;        for(i=0; i<N; i++)  {          evenNumberArray[i] = even;          even += 2;      }      /\* Returning base address of evenNumberArray array\*/      return evenNumberArray;  }    void main()  {     int \*array, counter;     array = getEvenNumbers(10);     printf("Even Numbers\n");     for(counter=0; counter<10; counter++){         printf("%d\n", array[counter]);     }       getch();  } |

Program Output

Even Numbers

2

4

6

8

10

12

14

16

18

20

**Advantages of arrays:**

 It is capable of storing many elements at a time

 It allows random accessing of elements i.e. any element of the array can be randomly accessed using indexes.

**Disadvantages:**

 Predetermining the size of the array is a must.

 There is a chance of memory wastage or shortage.

 To delete one element in the array, we need to traverse throughout the array.