One and Two-Dimensional Arrays and Pointers in C

http://cs-fundamentals.com/c-programming/arrays-in-c.php

Arrays in C Programming

Arrays in C or in any other programming language are container types that contain a finite number of homogeneous elements. Elements of an array are stored sequentially in memory and are accessed by their indices. Arrays in C language are static type and thence the size of array cannot be altered at runtime, but modern languages like Java, implement arrays as objects and give programmers facility to resize them at runtime. Arrays become useful storage containers when the size of the list is known beforehand.

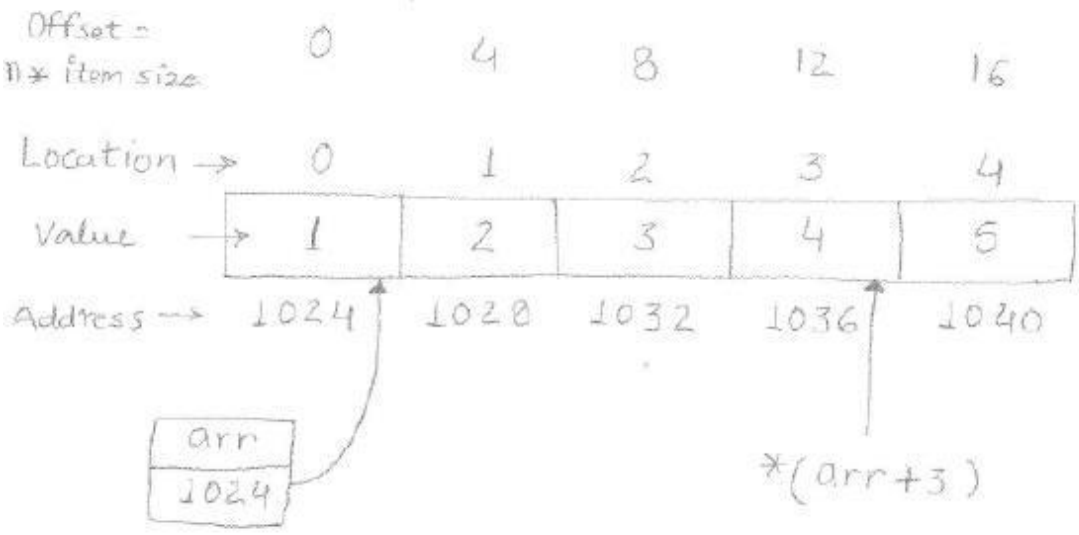
One Dimensional Arrays in C

Array name in C language behaves like a constant pointer and represents the base address of the array. It points to the first element of the array which is located at 0^{th} index. As array name serves like a constant pointer, it cannot be changed during the course of program execution. To demonstrate how arrays in C are defined, and accessed by the item index? How they are accessed by the help of pointers? Let's go through the following program and the description.

```c
/* Program: arrayDemo.c */
#include <stdio.h>
int main ()
{
  int arr[5] = {1, 2, 3, 4, 5}; // array of 5 integers
  for (int i = 0; i < 5; i++)
  {
    printf("value at %d location is: %d\n", i, arr[i]);
  }
  return 0;
}
```

Array `arr` defined in above program contains 5 integer values stored at indices from 0 to 4. Array index starts from zero that's why index of the last element will be size of array minus one, that is, `arr[4]` in this case. Below is the pictorial representation of `arr`, where the array `arr` points to first element at location 0. This element has address 1024 that is the base address of the array and represented by the array name.

Pictorial representation of array `arr`



Elements stored in an array are accessed by following the syntax "arrayName[index]" e.g., `arr[3]` yields 4 in above example. This strategy computes the base address of the desired element by combining the base address of the array with the index of desired element. Every array element takes a fixed number of bytes, which is known at compile time and this is decided by the type of array. In above example array is of type integer thus consumes 4 bytes (size of integer) per element. So the address of any n^{th} element in the array using 0 based indexing will be at an offset of *(n * element_size)* bytes from the base address of the array. You can devise the following formula to compute n^{th} element address.

```
nth element address = base address of array + (n * size of element in bytes)
```

The square bracket ([]) syntax to access array elements deals with address computation at its own. It takes the index that you wish to access, multiplies it with the element size, adds this resulting offset to the base address of the array, and finally dereferences the resulting pointer to get the desired element.

Now that you understand the computation of offset and then address of the element you want to access. Any element in the valid range of array can also be accessed by adding the index of that element to the base address and it computes the same offset as it was computed by square bracket syntax. But it leaves the result as a pointer and you have to dereference it at your own.

It should be clear from above passage of text that `(arr + 3)` is a pointer to the integer `arr[3]`, `(arr + 3)` is of type `int*` while `arr[3]` is of type `int`. The two expressions only differ by whether the pointer is dereferenced or not. Thus the expression `(arr + 3)` is exactly equivalent to the expression `&arr[3]`. In fact those two probably compile to exactly the same code. They both represent a pointer to the element at index 3. Any square bracket ([]) syntax expression can be written with the + syntax instead. We just need to add in the pointer dereference. Conclusively, `arr[3]` is exactly equivalent to `*(arr + 3)`.

Pay attention to the following lines to understand correct syntax of dereferencing. You can also write `*(arr + 3)` to `*(3 + arr)`, this is perfectly right and acceptable as addition is commutative. Likewise, can you write `arr[3]` to `3[arr]`? Yes, you can. But on the contrary `[arr]3` or `[3]arr` is not correct and will result into syntax error, as `(arr + 3)*` and `(3 + arr)*` are not valid expressions. The reason is dereference operator should be placed before the address yielded by the expression not after the address.

Why Array Index in C Starts From Zero?

The short answer is, it depends on language design whether the start index should be zero or any other positive integer of your choice. in Fortran, when an array is declared with `integer a(10)` (an array of 10 integer elements), the index starts from 1 and ends at 10. However, this behavior can be overridden using a statement like, `integer a(0:9)`, declaring an array with indices from 0 to 9.
But In C language we do not have the freedom to start the array index from any other number than zero, and language strictly sticks to zero as start index of array. It is because in C the name of an array is a pointer, which is a reference to a memory location. Therefore, an expression `*(arr + n)` or `arr[n]` locates an element n-locations away from the starting location because the index is used as an offset. Likewise, the first element of the array is exactly contained by the memory location that array refers (0 elements away), so it should be denoted as `*(arr + 0)` or `*(arr)` or `arr[0]`.
C programming language has been designed this way, so indexing from 0 is inherent to the language.

Two-Dimensional Arrays in C

A two dimensional array (will be written 2-D hereafter) can be imagined as a matrix or table of rows and columns or as an array of one dimensional arrays. Following is a small program `twoDimArrayDemo.c` that declares a 2-D array of 4x3 (4 rows and 3 columns) and prints its elements.

```c
/* Program: twoDimArrayDemo.c */
#include <stdio.h>
#define ROWS 4
#define COLS 3
int main ()
{
    // declare 4x3 array
    int matrix[ROWS][COLS] = {{1, 2, 3},
                              {4, 5, 6},
                              {7, 8, 9},
                              {10, 11, 12}};

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

The array elements in above program are stored in memory row after row sequentially; assuming array is stored in row major order. As you know array name behaves like a constant pointer and points to the very first element of the array. The same is true for 2-D arrays, array name matrix serves as a constant pointer, and points to the first element of the first row. Array elements within valid range of matrix can be accessed by following different syntaxes.

```
matrix[0][0] = *(*(matrix))
matrix[i][j] = *((*(matrix)) + (i * COLS + j))
matrix[i][j] = *(*(matrix + i) + j)
matrix[i][j] = *(matrix[i] + j)
matrix[i][j] = (*(matrix + i))[j]
&matrix[i][j] = ((*(matrix)) + (i * COLS + j))
```

Passing Two-Dimensional Array to a Function in C

Passing 2-D array to a function seems tricky when you think it to pass as a pointer because a pointer to an array and pointer to a pointer (double pointer) are two different things. If you are passing a two dimensional array to a function, you should either use square bracket syntax or pointer to an array syntax but not double pointer. Why should not you use double pointer to access array elements will be described in next section. Let's rewrite `twoDimArrayDemo.c` as `twoDimArrayDemoPtrVer.c` and demonstrate passing 2-D array to function for printing array.

```c
/* Program: twoDimArrayDemoPtrVer.c */
#include <stdio.h>
#define ROWS 4
#define COLS 3

void array_of_arrays_ver(int arr[][COLS]); /* prototype */
void ptr_to_array_ver(int (*arr)[COLS]); /* prototype */

int main ()
{
    // declare 4x3 array
    int matrix[ROWS][COLS] = {{1, 2, 3},
                              {4, 5, 6},
                              {7, 8, 9},
                              {10, 11, 12}};

    printf("Printing Array Elements by Array of Arrays Version Function: \n");
    array_of_arrays_ver(matrix);

    printf("Printing Array Elements by Pointer to Array Version Function: \n");
    ptr_to_array_ver(matrix);
    return 0;
}

void array_of_arrays_ver(int arr[][COLS])
{
     int i, j;
     for (i = 0; i < ROWS; i++)
     {
          for (j = 0; j < COLS; j++)
          {
              printf("%d\t", arr[i][j]);
          }
          printf("\n");
     }
}
```

```c
void ptr_to_array_ver(int (*arr)[COLS])
{
    int i, j;
    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            printf("%d\t", (*arr)[j]);
        }
        arr++;
        printf("\n");
    }
}
```

```
OUTPUT
======
Printing Array Elements by Array of Arrays Version Function:
1       2       3
4       5       6
7       8       9
10      11      12
Printing Array Elements by Pointer to Array Version Function:
1       2       3
4       5       6
7       8       9
10      11      12
```

In `twoDimArrayDemoPtrVer.c` you see two ways of passing 2-D array to a function. In first function `array_of_arrays_ver` *array of arrays* approach is used, while is second function `ptr_to_array_ver` *pointer to array* approach is used.

Another very important point to note is the called function does not allocate space for the array and it does not need to know the overall size, so the number of rows can be omitted. Space is not allocated because called function does not create a local copy of the array rather it uses the original one that has been passed to it. The width of the array is still important because number of elements contained by one row has to be told to compiler in order to increment the pointer to point the next row. So the column dimension `COLS` must be specified.

Double Pointer and Two Dimensional Arrays in C

While trying to access a 2-D array by a double pointer compiler may not prevent you from doing so but you would not get expected results in some situations. Let's have a look at the following program `twoDimArrayDblPtrVer.c` and its generated output.

```c
/* Program: twoDimArrayDblPtrVer.c */
#include <stdio.h>
#define ROWS 4
#define COLS 3
int main ()
{
    // matrix of 4 rows and 3 columns
    int matrix[ROWS][COLS] = {{1, 2, 3},
                              {4, 5, 6},
                              {7, 8, 9},
                              {10, 11, 12}};
    int** pmat = (int **)matrix;

    printf("&matrix[0][0] = %u\n", &matrix[0][0]);
    printf("&pmat[0][0] = %u\n", &pmat[0][0]);
    return 0;
}
```

```
OUTPUT
======
&matrix[0][0] = 1245016
&pmat[0][0] = 1
```

It happens because two dimensional array `matrix` and double pointer `pmat` are different types and using them in similar fashion points to different locations in memory. The following piece of code aborts the program with a "memory access violation" error.

```c
printf("pmat[0][0] = %u\n", pmat[0][0]);
```

And, you might have guessed why this access violation error is? Because the same memory location is being dereferenced two times, that's why this access violation error. To conclude, arrays in C are containers for homogeneous elements. Array elements can be accessed and modified with help of the pointers.

Last Word

In this tutorial we talked of one dimensional arrays in C, why array index in C starts from zero, two dimensional arrays, passing 2-D array to a function, and double pointer and two dimensional arrays. Hope you have enjoyed reading this tutorial. Please do write us if you have any suggestion/comment or come across any error on this page. Thanks for reading!