Advanced Data Representation in C

(Arrays, Structures, Unions and the use of Pointers with them)

Manik Chand Patnaik (M.Tech.)
Lecturer, (C.S.E), Roland Institute of Technology.
Wednesday, November 29 – 2006.

Introduction

We have handled all sorts of variables, integer, long, double, char so on and so forth. These data types are called the primitive datatypes in C. Now it is time to deal with some of the advanced data representations. These advanced datatypes are derivatives of the primitive datatypes only and are not completely new ones. The datatypes created by using the datatypes already present as primitive datatypes are referred to as derived data types. You should have a clear understanding of the datatypes† before we proceed.

Arrays

The first derived datatype in our discussion list is an array. An array is a linear data structure which is a collection of data elements of the same datatype each having exactly the same size and under a common name. Each data element of the array can be uniquely referenced using its index number along with the name of the array.

Generally speaking, arrays are of two types:

1. Unidimensional arrays (also called Linear arrays)
2. Multidimensional arrays

Declaration

Syntax:

<datatype> <arrayname>[<dimension1>][<dimension2>] ... so on..;

e.g. int arr1[10]; will create a unidimensional array of 10 integer size and int arr2[5][2]; would create a two dimensional array of 10 integer size (5 rows and 2 columns matrix).

Referencing each element

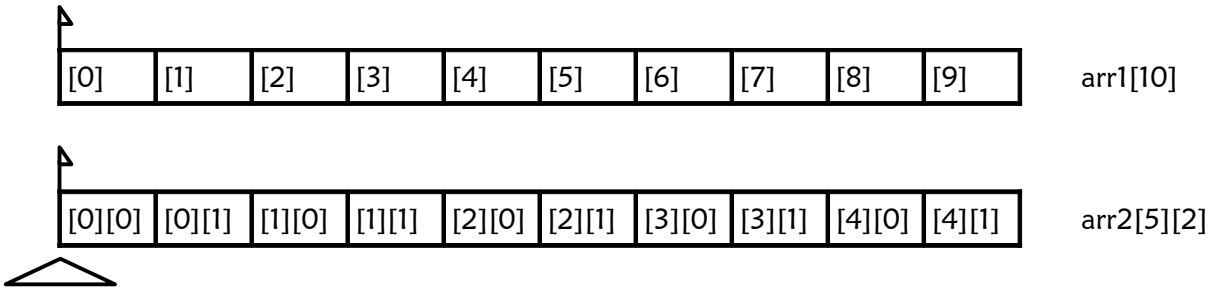
Each element of the array is referenced by the array name and the subscript number a.k.a the index. The index is the logical dimension-wise offset of the element from the base$. To refer to the first element we would write arr1[0] and to refer to the last element we would write arr1[4]. We will handle them as our normal variables.

---

† Here is a quick recap:
Datatypes are of Four types,
1. Primitive - Your integer, character, float etc inbuilt data types.
2. Derived - The array, structure(an aggregate datatype), pointer and function.
3. User Defined - Alias created by users for existing datatypes.
4. Null - The empty type.

$ Offset is the concept of distance. How distant is the element from the base of the array. The first element would be placed at the base whereas the second element would be at distance 1 from the base. So it is why the array index/subscript values start from zero rather than starting from one.

There is no implementational difference between a uni or multi dimensional array. Both are implemented as a linear data structure†, there is no tabular layout although a multidimensional array is an array of arrays. The following diagram is enough to show it. When we will handle it with a pointer it would be clearer to understand (Refer and relate with the example in Pointer arithmetic).



You already know from the discussions in the class that array name itself is like a pointer‡ let's create a pointer for the array arr2.

Just create a pointer of the same datatype as that of the array and assign as follows:

```
int arr2[5][2]; /*creating the array*/
int *ptr;       /*creating the pointer*/
ptr=arr2;       /*assigning the base address of the array*/
```

This can very well can be written in a single line.

```
int arr2[5][2],*ptr=arr2;      /*don't put like this int *ptr=arr2,arr2[5][2]; it will tell that
                                 arr2 is undeclared.*/
```

I have written them in different lines for clarity sake. The line on assignment of address can also can take a different form:

```
ptr=&arr2[0][0];       /*assigning base address of the first element of the array*/
```

Both methods are same and bear the same result. Now how do we refer to the elements of the array using the pointer? It's by using dereferencing$^{\mu}$. To point to the first element, the answer is obvious, *ptr. Now we can use *ptr to store anything at the first element of the array arr2.

```
*ptr=20;       /*storing 20 at arr2[0][0]. We can directly do it as arr2[0][0]=20; As the
                 address of arr2 is stored at ptr which is exactly the same as arr2[0][0]*/
```

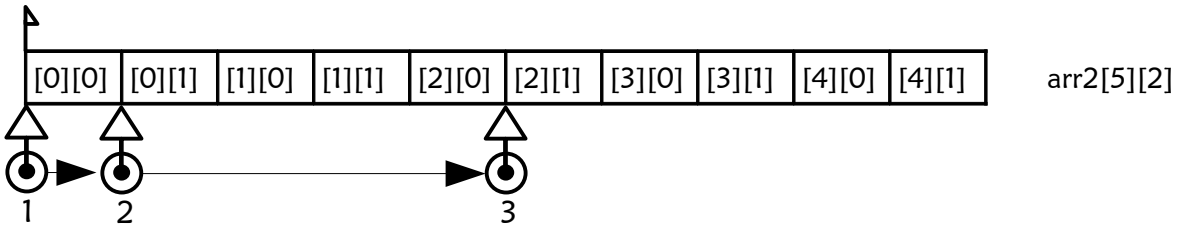
† This is a concept to be understood. There is no actual matrix like data allocation in memory because as I always point out, "memory is always linear". How? It is a series of addresses linked to actual hardware and the addresses are a series of unsigned numbers; There is nothing of a tabular fashion. You plug in another bank of memory on the motherboard and it will be added up to the current continuing list. You won't refer it to memory xyz from bank 2. Got the point?

‡ Array name is not actually a pointer type. There are glaring differences. You cannot use dereferencing by putting a star before array name. You cannot assign another address to it because it is a constant not a variable like the ordinary pointers. Equally you cannot put subscripts in a pointer to an array to handle the array elements like you do with an array name.

μ Dereferencing uses the indirection operator. Both are the same concept with two different names. Indirection as well as dereference is pointing to the value which is stored at the address of the current variable. In this case the pointer we are using.

Pointer Arithmetic

Pointer arithmetic is a very interesting concept. We can add and subtract integer values to a pointer. Let's continue with our previous example of arr2. You know that ptr is pointing to the first element of the two dimensional array arr2. You can shift the pointing location of ptr to the second element by just incrementing the pointer. e.g. ptr++; You can further move it to arr2[2][1] (marked by number 3 in diagram) by just adding 4 to it. e.g. ptr+=4;

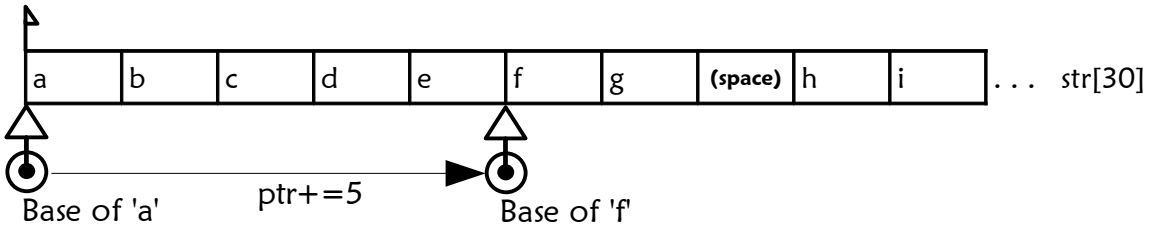


Moving the pointer along the array is called traversal. The traversal along this arr2[5][2] two dimensional array is analogous to traversal along a unidimensional 10 element integer array because the memory map of a unidimensional 10 element array is analogous with a two dimensional 5X2 or 2X5 matrix representation of array.

Carefully watch the following block of code:

```
char str[30],*ptr;
ptr=str;                /* pointer assigned to string a.k.a. character array */
printf("Enter a string :");
scanf("%[^\n]",str);    /* caret-backslash-n to accept characters till enter is pressed */
ptr+=5;                 /* moving pointer forward 5 places in the string */
printf("%s",ptr);       /* See the output carefully and understand it through the diagram*/

/* Output:
abcdefg hijklm nopq     <- ptr was pointing to the base of 'a' initially. After moving 5
fg hijklm nopq          <- characters ahead it is now pointing to base of 'f'. printf has started
                        <- to print the string starting from 'f'. See the similarity of handling of
                        <- string and a pointer here along with the pointer arithmetic.
*/
```



Exactly in the same manner we can subtract whole numbers from pointers too and the arithmetic operation will be similarly done depending on the datatype. For addition of n the pointer will point forward n*sizeof(ptr) bytes and for a subtraction of n, the same number of bytes will be referenced backwords.

This is the simplest representation of pointer arithmetic.