Unit-2 Arrays, Pointers and Structures

2.1 Array and Pointer.

2.2 Structure and Pointer.

- 2.2.1 Structure pointer.
- 2.2.2 Self-referential pointer.

2.3 Dynamic Memory Allocation: malloc, calloc, realloc, free.

Practical Works

2.1 Write program to illustrate memory allocation dynamically.

2.1 Array and Pointer.

- Array

Array is the collection of similar data items that can be represented by a single variable name. The individual data items in an array are called elements. A data item is stored in memory in one or more adjacent storage locations depending upon its type.

If it contains *int* data type, all the data of the array must be of *int*, if *float* then all data must be of *float*. A specific element of an array is accessed by an index. Each array element is accessed by specifying an array name followed by subscript/ index [enclosed in square bracket]. So, array is also called as subscripted/indexed variable.

Subscript/ index is used to denote size of array.

For example,

int b[5], where *int* is data type of array, *b* is name of array and 5 is size of 5 elements of array and each subscript/ index must be a non-negative number.

- Array Declaration

In C program, all the variables must be declared before its use. So, an array must be declared so that compiler will understand the type of the

variable and array size in the first line after the *main().*

Syntax:

1. datatype array name [size];
2. datatype array name [size] = {list of elements};
3. datatype array name [ ] = {list of elements};

- Types of Array

1. Single Dimensional Array

The array which contains only one subscript is called single dimensional array.

Syntax:

datatype *array name* [size];

2. Multi Dimensional Array

The array which contains more than one subscript is called multi dimensional array.

Syntax:

datatype *array name* [size][size]……[size];

- Array Initialization

Each array element can be initialized, when an array is declared. The initial

values must appear in the order in which they will be assigned to the individual array elements enclosed in braces and separated by commas.

Examples:

1. Single Dimensional Array

int k[10)= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float p[4] ={1.3, 2.4, 3.5, 4.6};
char color [5] = { 'R', 'G', 'B', 'Y', 'P');

When a list of elements are shorter than the number of array elements to be initialized, the remaining elements are initialized to zero.

int k[5] = {1,2,3};
The elements k[3] and k[4] will have value 0 'zero'.
The array size can be omitted if you initialize the array elements.
int k[ ] = {1, 2, 3, 4};
Now, the size of array k is 4.

2. Two Dimensional Array

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

Here, value [0][0] will take 1, value [0][1] will take 2 and so on value [1][2] will take 6.

In the above example, the elements are grouped in separate braces. We can also initialize the above array as

int value [2][3] = {1, 2, 3, 4, 5, 6}.

Here, the values will be assigned in the order they appear.

- Pointers

A pointer is a variable that stores the address of another variable. In other words, pointer is a variable that refers or points to another variable. The variable that the pointer refers to is also known as its pointee.

Some of the uses of pointers are:

1. In passing the argument by reference.
2. In passing array as arguments.
3. In processing arrays more efficiently.
   - 4. In creating data structure such as linked lists.
   - 5. In reducing length and complexity of program and increase execution speed.

- Address(&) and Indirection(*) operator

Pointer uses two operators **'&'** and **'*'**. **'&'** (ampersand) is a unary operator that returns a memory address of a variable. **'*'** (asterisk) is also a unary operator that returns value stored at the memory location stored in a pointer variable. **'&'** is also called as **"address"** or **"the**

address of" operator. **'*'** is also called as **"indirection"** or **"the value at address"** operator. Accessing an object through a pointer is called dereferencing.

- Declaration of Pointer

A pointer variable is declared like other variables except an operator '*' precedes the variable name.

Syntax:

Data type * variable name;

where, **data type** represents the **type of variable** which the **pointer is pointing to** and **variable name** represents the **name of the pointer variable.**

Example:

int *p;

where, 'p' is a variable of type "pointer-to-int". In other words, 'p' is a pointer variable that points to the value of type 'int'.

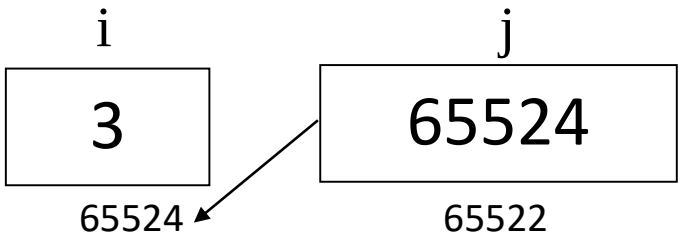


Figure: 'j' is the pointer of 'i' as it holds the address of 'i'

- Relationship between arrays and pointers

In C, any operations that can be achieved by array subscripting can also be done with pointers. Pointers are difficult concept to understand for beginners but are faster in execution than array.

A pointer variable can take different addresses as values where as an array name is an address or pointer which is fixed. An array name always points to the first element of the array.

Let **arr** be a one-dimensional array, then the address of the first array element can be expressed as either **&arr[0]** or simply **arr.** In the same way, the address of the second element of the array can be written as either **&arr[1]** or as **(arr+1).** It means the address of array element **(i+1)** can be expressed as either **&arr[i]** or as **(arr+i).** The dereference as ***arr, *(arr+1),…..,*(arr+i)** is same as accessing array elements as **arr[0],arr[1], ……arr[i].** The relation of pointer and array is presented in the following tables.

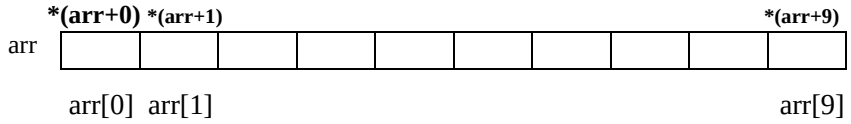
Address of array elements	**Equivalent pointer notation**
&arr[0]	arr
&arr[1]	(arr+1)
…….	…….
&arr[i]	(arr+i)
Value of array elements	**Equivalent dereference notation**
arr[0]	*arr
arr[1]	*(arr+1)
…….	…….
arr[i]	*(arr+i)

Following example illustrates the concept.

The declaration

int arr[10];

defines an array of size 10, that is a block of 10 consecutive variables of type **'int'** named as **arr[0], arr[1] ….. arr[9].**



If **'parr'** is a pointer to an integer, declared as **int * parr;**

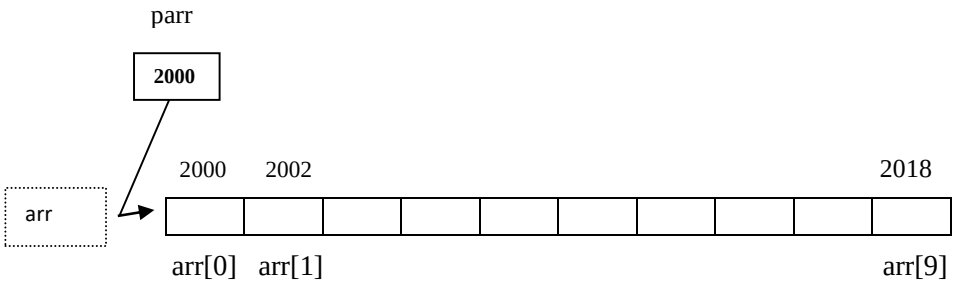
Then the address of array can be assigned as

parr = &arr[o];

or simply,

parr = arr;

because the array name contains the address of the element zero (first element). This mechanism is represented pictorially as



Now, the assignment **x=*parr;** // same as **x=*arr;**

will copy the contents of arr**[0]** into **x**.

Similarly,

x=*(parr + 1); // same as **x=*(arr+1);**

will copy the contents of **arr[1]** into **x** and

x=*(parr + i); // same as **x=*(arr+i);**

will copy the contents of **arr[i]** into **x**.

After assigning the address of array **'arr'** to pointer **'parr'**, the **'parr'** can use subscripting to access the element of the array **'arr'** as **parr[0]** same as **arr[0]**, **parr[1]** same as **parr[1]** and **parr[i]** same as **arr[i].**

- Differences between array and pointer

Array	Pointer
1. An array name is a constant pointer to the first element of the array	1. A pointer variable can point to different memory locations.
2. Declaration ***int x[10];***	2. Declaration ***int *p;***
3. Address can be accessed by ***&x[i]*** as ***&x[0]*** ,***&x[1]*** till ***&x[9]***	3. Address can be accessed by ***p+i*** as ***p+1***,***p+2*** till ***p+9.***
4. Value can be accessed by ***x[i]*** as ***x[0],x[1]*** till ***x[9].***	4. Value can be accessed by ****(p+i)*** as ****p,*(p+1)*** till ****p+9.***
5. Array is easier to understand but works slow.	5. Pointer is difficult to understand but works fast.

2.2 Structure and Pointer

2.2.1 Structure pointer.

Relationship between structures and pointers

Consider the following declaration

```
struct inventory
        {
                char name [30];
                int number;
                float price;
        } product [2],*ptr;
```

The statement declares the *product* as an *array* of two elements, each of type *struct inventory* and *ptr* as a *pointer* to data objects of the type *struct inventory.*

The assignment

ptr = product;

would assign address of *zeroth* element of *product* to *ptr*. *That* is the pointer will now point to product [0]. It's member can be accessed by using following notation.

ptr→ name
ptr→ number
ptr→ price

Symbol → is called arrow operator

2.2.2 Self-referential pointer

Self-referential structures are those structures that have one or more pointers which point to the same type of structure, as their member. In other words, structures pointing to the same type of structures are self-referential in nature

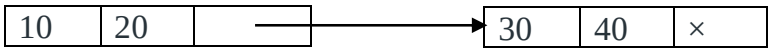
Example:

```
struct node
        {
                int data1;
                char data2;
                struct node *link;
        }
int main( )
        {
                struct node ob;
                return 0;
        }
```

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer. An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Self-Referential Structure with Single Link:

These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



ob1 **ob2**

2.3 Dynamic Memory Allocation: malloc(), calloc(), realloc(), free()

Consider the following array declaration

int age [20];

It reserves memory space for 20 integer numbers. Sometimes, problems arise with this declaration. If the user needs to work on more than 20 values the allocated memory will not be sufficient. Besides, if the user needs to work on less than 20 values, the allocated memory will be unused. This type of allocating memory at compile time is called static memory allocation.

To overcome this limitation, the memory for array elements can be allocated at the run time. The first address of the allocated memory is assigned to the pointer variable which can subsequently be used as array.

The memory allocation at runtime is called dynamic memory allocation.

1. Malloc()

Syntax:

ptr=(data_type*)malloc(size_of_block);

For example:

p=(int*) malloc(n*sizeof(int));

pnew=(struct node*)malloc(sizeof(struct node))

The n is the number of variables.

If n=100 then pointer p variable occupies 200 bytes of memory.

100*2= 200; occupies 2 bytes for a single integer variable.

2. Calloc()

Syntax:

ptr=(data_type*calloc(number_of blocksize_of_each_block);

For example:

x=(int*) calloc(5,10*sizeof(int));

or

x=(int*)calloc(5,20);

5 blocks of size 20 bytes.

3. Realloc()

Syntax:

ptr=realloc(ptr,newsize);

For example:

pnew=realloc(pnew,20);

4.Free()

Syntax:

free(ptr);

For example:

free(pnew);

-0-