KIET Group of Institutions, Ghaziabad

Department of Computer Applications

(An ISO – 9001: 2015 Certified & 'A' Grade accredited Institution by NAAC)

Problem Solving Using C

KCA 102: Session 2020-21

Unit 3	**Lecture 1**	**Representation, notations, declaration and initialization of one dimensional array**

What is Array?

An Array is a collection of similar datatype of elements addressed by a common variable name.

OR

An Array is a collection of 2 or more adjacent memory location containing same type of data.

Point to Remember:

1. This memory location is called array elements
2. First dimensional array is mathematically called vector

Advantages of Array:

1. Use of less line of code as it creates a single array of multiple elements.

2. Easy access to all the elements.

3. Traversal through the array becomes easy using a single loop.

4. Random access of elements using array index.

5. Sorting becomes easy as it can be accomplished by writing less line of code.

Disadvantages of Array:

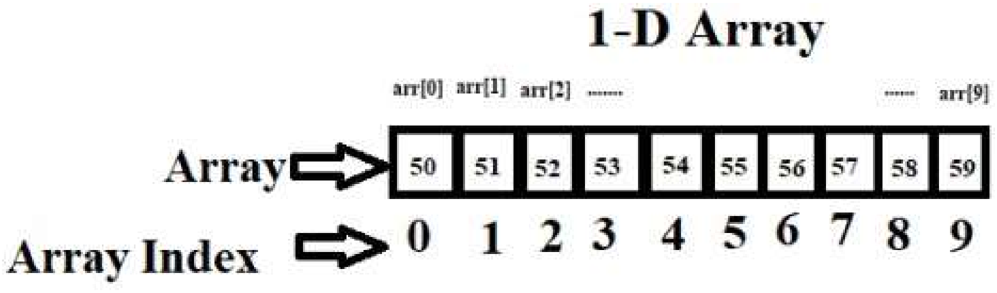
1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.

2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Types of Array:

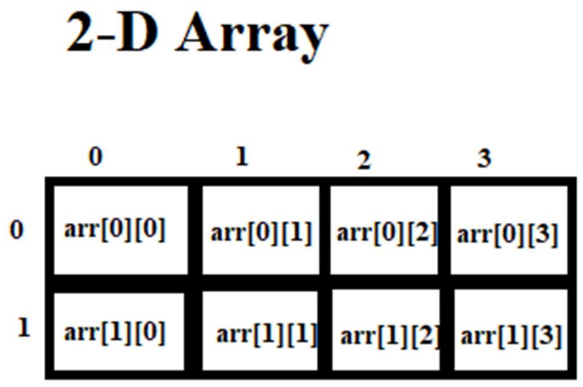
1**. 1-D Array**

A one dimensional arrays store records in one dimension. As it can store only one record of any entity.



2. Multi-Dimensional Array

A 2-D array can hold two record of an entity.



Declaration of Array:

The syntax is:

data_type array_name[size];

Point to Remember:

1. here, the "data_type" is the type of data used by the array
2. "array_name" is the name of the array variable
3. "size" is the number of elements contain in the array

For Example:

int mark[50];

The above declaration instructs the C-Compiler to store or to associate 50 memory cells with the name of mark having the int datatype.

Array Representation and Notation:

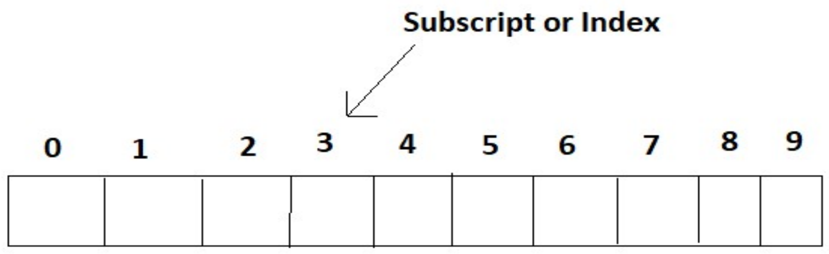
In array representation and notation subscript play great role, the array subscript is the counting value of the array which is always started from zero.

For Example:

If a user declare

int marks[10];

the subscript value of the above array is



The element inside the subscript value is known as the array element.

Initialization of an Array:

Like the normal variable, we can also initialize the array elements.

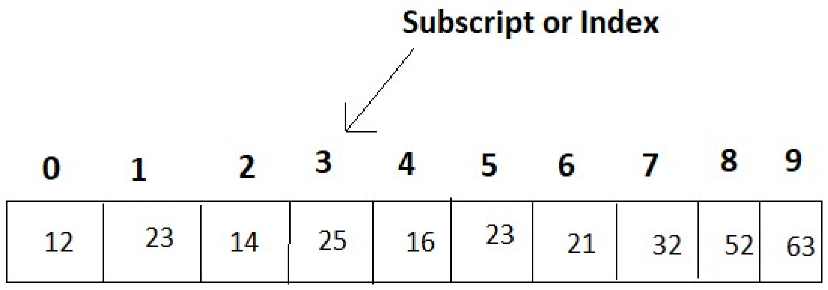
For Example:

```
int marks[10]={12,23,14,25,16,23,21,32,52,63};
                              OR
int marks[]={12,23,14,25,16,23,21,32,52,63};
                              OR
Int marks[10];
 marks[0]=12;
 marks[1]=23;
 marks[2]=14;
 marks[3]=25;
 marks[4]=16;
 marks[5]=23;
 marks[6]=21;
 marks[7]=32;
 marks[8]=52;
 marks[9]=63;
```

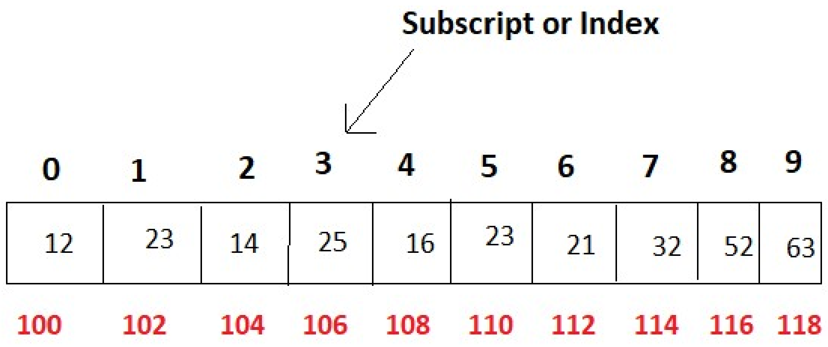
Point to be remember:

1. Till the array elements are not given any specific values, they are supposed to contain garbage values.
2. If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

memory representation of above array marks will be as:



Memory representation of above array marks with memory location will be as:



Unit 3	**Lecture 2**	**Accessing and manipulating array elements**

Accessing Array Value:

We can access array value by using it index value like as given below:

```
int marks[4]={4,5,6,7};
```

For accessing single value:

```
printf("%d",marks[0]);
```

For accessing all value using for loop:

```
for(int i=0;i<4;i++)
{
        printf("%d",marks[i]);
}
```

Manipulating Array Value:

Array manipulating means accessing, deleting, modify and inserting values in array. To understand it completely, let's take an example to get the sum of 10 value using array:

```
int main()
{
         int numbers[11];  //Array Declaration
         numbers[11]=0;   //Initializing array last value with 0
         int i=0;
         printf("Enter 10 value=>");
         for(i=0;i<10;i++)
         {
         scanf("%d",&numbers[i]);   //Inserting 10 values in array
         }
         for(i=0;i<10;i++)
         {
         numbers[11]=numbers[11]+numbers[i];
         //Accessing and manipulating values in array
         }
         printf("Sum of 10 number is=%d",numbers[11]);
         //Displaying result from last position of array
         return 0;
}
```

Unit 3	**Lecture 3**	**Introduction of two and multidimensional array**

2D Array:

2 Dimensional array is an array which store the data in row wise and column wise. Sometimes in programming it is require to store data in a tabular format, so that on that time 2D Array is very much helpful to store data in tabular format.

The 2D array is also called table.

Declaration of 2D Array:

Syntax for it is:

```
data_type array_name [rowsize][columnsize];
```

Here, data_type is the type of data which is used in the array & array_name is the name of the array used by the user, rowsize and columnsize is the size of the row & column.

For Example:

```
int marks[3][4];
```

Here, in the above declaration, the C-Compiler instruct to store a table in the name of marks with data type "int" having the 3 rows & 4 columns.

Initialization of 2D Array:

We can initialize 2D array as:

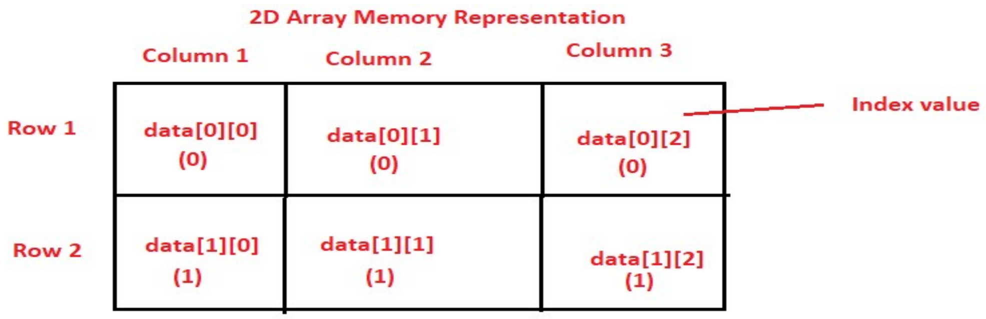
```
int data[2][3]={0,0,0,1,1,1};
```

Here, in the above example the 1st row contains 0 and 2nd row contains 1.

We can also initialize the above initialization in the following format also:

```
int data[2][3]={
                {0,0,0},
                {1,1,1}
            };
```

Here, the rows are separated by braces.



Accessing and Manipulation of 2D Array:

Let's take an example to understand accessing and manipulation of data in 2D Arrays

```
int main() {
        int data[2][3];
        //2D Array declaration
         int i=0,j=0;
        printf("Enter data in 2 Row and 3 Column");
        //Starting for loop for getting input in 2D Array by Keyboard
        for(i=0;i<2;i++)
        {
                for(j=0;j<3;j++)
                {
                        scanf("%d",&data[i][j]);
                        //Scanf to enter values from keyboard to 2D Array
                }
        }
        printf("Data inserted by you is=\n");
        //Starting for loop for displaying valued of 2D Array on screen
        for(i=0;i<2;i++)
        {
                for(j=0;j<3;j++)
                {
                        printf("%d ",data[i][j]);
                }
                printf("\n");
        }

        return 0;
}
```

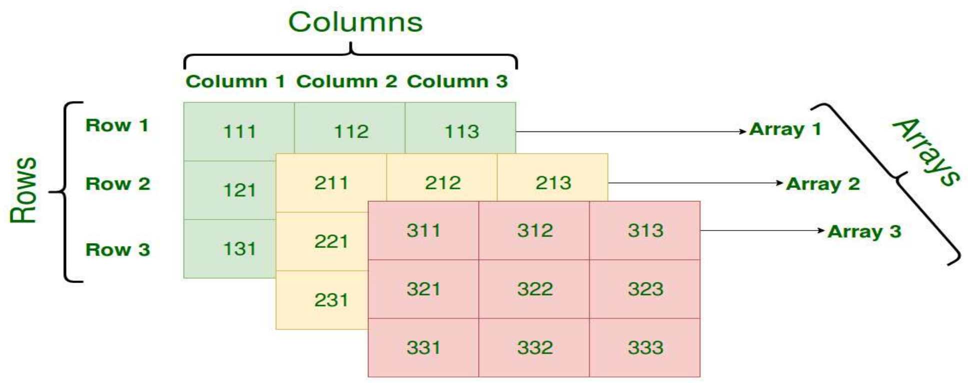
Multi-Dimensional Arrays:

As we declare 2D array, in same manner we can declare multi-dimensional or n-dimensional arrays also;

For Example:

```
int data[2][3][4];
int data[2][4]……..[n];
```

Memory Representation for Multi-Dimensional Array:



Unit 3	Lecture 4	Introduction and characteristics of & and * operator

Reference operator (&):

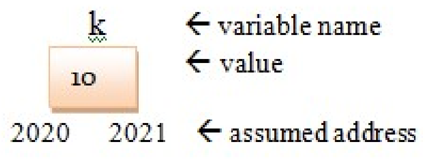
This referencing operator is also called address operator, which gives the address of a variable, in which location the variable is resided in the memory.

Syntax:

&variable-name;

Example:

int k=10;



```
printf("\n The address of k=%u", &k);

printf("\n The value of k=%d", k);
```

Output:

The address of k=2020
The value of k=10

Point to be Remember:

1. Here the expression '&k' gives the address of 'k' in which it is stored, the format string %u or %p is used to print address.
2. Here the expression '&k' is pronounced as "address of k", the space for 'k' at run time may be allocated anywhere in the RAM, and to get the address of 'k', reference operator is used.
3. The address of a variable may not be a fixed location like 2020 above, and is randomly given by the computer.

De-reference operator (*):

This operator is used to get the value at a given address.

For example:

```
int k=10;

print("%d",*&k);

//*&k -> k -> 10
```

Output:

```
10;
```

The reference operator (&) gives the address of 'k', whereas de-reference operator (*) gives value in that address, so finally it is 'k' value 10.

The action of 'de-referencing operator' is just contrary to the action of referencing operator, this is also called 'indirection operator' as it changes the direction of control from location to another location within the program's memory to access the required data.

Used in Pointer Declaration:

This operator also used in C for creating pointer variable, the pointer variable should proceed by called "value at address operator". It returns the value stored at particular address. It is also called an indirection operator (symbol *).

Example:

```
Int v;  //Normal variable declaration
int * p; //Pointer variable declaration
p=&v;   //passing address of variable "v" into point variable "p"
```

Unit 3	Lecture 5	Pointer type, declaration, assignment and pointer arithmetic

What is Pointer?

Pointer is a special kind of variable used to store address of memory location through which indirect memory accessing can be performed.

When variables are declared memory is allocated to each variable. C provides data manipulation with addresses of variables therefore execution time is reduced. Such concept is possible with special data type called pointer. A pointer is a variable which holds the address of another variable or identifier this allows indirect access of data.

We know that memory is a collection of bits, in which eight-bits constitute one byte. Each byte has its own unique location number called its "address". To store this address, we need a special kind of variable called "pointer variable".

To work with pointers, we have two unary operators

1. **Reference operator (&)**
2. **De-reference operator (*)**

The '**&**' and '*****' operators work together for referencing and de-referencing.

Pointer Declaration:

Syntax of pointer declaration is:

datatype *pointer_name;

For Example:

int *p; // pointer to integer variable

Here are a few examples:

```
float *fp;     // pointer to float variable
```

```
double *dp;     // pointer to double variable
char *cp;       // pointer to char variable
```

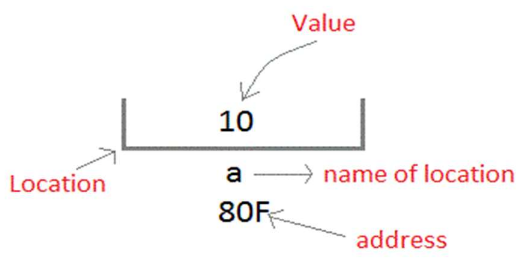
Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

For Example:

Let us assume that system has allocated memory location 80F for a variable a.

int a = 10;



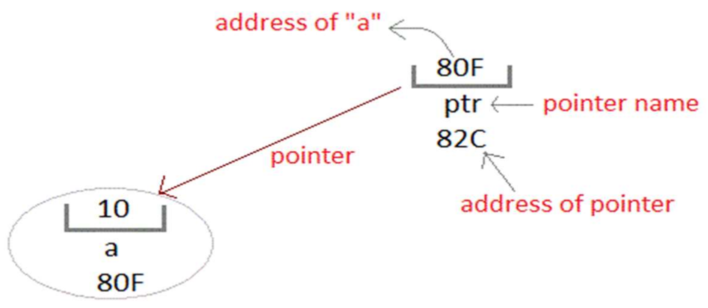
We can access the value 10 either by using the variable name "a" or by using its address 80F.

The question is how we can access a variable using its address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.

For Example:

```
int a=10;
int *ptr=&a;
```



Benefits of using pointers:

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

Initialization of Pointer variable:

Pointer Initialization is the process of assigning address of a variable to a pointer variable. Pointer variable can only contain address of a variable of the same data type. In C language address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>
 void main()
{
        int a = 10;
        int *ptr;      //pointer declaration
       ptr = &a;      //pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>
void main()
{
        float a;
       int *ptr;
       ptr = &a;      // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a NULL pointer.

```
#include <stdio.h>
int main()
{
int *ptr = NULL;
return 0;
}
```

Points to be Remember While Using Pointers:

1. While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand &.
3. The pointer variable stores the address of a variable. The declaration int *a doesn't mean that a is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as 'value at'.

Let's understand it by using C program:

```
#include <stdio.h>

int main()

{

        int i = 10;     // normal integer variable storing value 10

        int *a;     // since '*' is used, hence it's a pointer variable

  // '&' returns the address of the variable 'i' which is stored in the pointer variable 'a'

        a = &i;

 // below, address of variable 'i', which is stored by a pointer variable 'a' is displayed

        printf("Address of variable i is %u\n", a);

//   below, '*a' is read as 'value at a' which is 10

        printf("Value at the address, which is stored by pointer variable a is %d\n", *a);

        return 0;

}
```

Output:

```
Address of variable i is 2686728 (The address may vary)

Value at an address, which is stored by pointer variable a is 10
```

Pointer Arithmetic:

An integer operand can be used with a pointer to move it to a point / refer to some other address in the memory.

In a **16-bit** machine, size of all types of pointer, be it **int***, **float***, **char*** or **double*** is always **2 bytes**. But when we perform any arithmetic function like increment on a pointer, changes occur as per the size of their primitive data type.

Consider an int type ptr as follows:

int * mptr ;

Assume that it is allotted the memory address 65494 increment value by 1 as follows.

```
mptr ++;          or          ++ mptr;
mptr = mptr +1;   or          mptr + =1 ;
```

++ and -- operators are used to increment, decrement a ptr and commonly used to move the ptr to next location. Now the pointer will refer to the next location in the memory with address 65496.

C language automatically adds the size of int type (2 bytes) to move the ptr to next memory location.

mptr = mtpr + 1
= 65494 + 1 * sizeof (int)
= 65494 + 1 * 2

Similarly, an integer can be added or subtract to move the pointer to any location in **RAM**. But the resultant address is dependent on the size of data type of **ptr**.

The step in which the ptr is increased or reduced is called scale factor. Scale factor is nothing but the size of data type used in a computer.

We know that, the size of

float = 4
char = 1
double = 8 and so on

Example:

```
float *xp;     (assume its address = 63498)
xp = xp + 5;
```

Will more the ptr to the address 63518

$\Rightarrow$ 63498 + 5 * size of (float)
$\Rightarrow$ 63498 + 5*4
$\Rightarrow$ 63498+20
$\Rightarrow$ 63518

Rules for pointer operation:

The following rules apply when performing operations on pointer variables

1. A pointer variable can be assigned to address of another variable.
2. A pointer variable can be assigned the values of other pointer variables.
3. A pointer variable can be initialized with NULL or 0 values.
4. A pointer variable can be prefixed or post fixed with increment and decrement operator.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers points to the same array, one pointer variable can be subtracted from another.

7. When two pointers points to the objects of same data types, they can be compared using relational
8. A pointer variable cannot be multiple by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address.

Unit 3	Lecture 6	Call by reference, passing pointer to a functions

In C, we can pass parameters to a function either by pointers or by reference. In both the cases, we get the same result.

Call by Reference:

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will affect the original variable.

For Example: Swapping two numbers using Pointer

```
        #include <stdio.h>

        void swap(int *a, int *b);

        int main()

        {

                int m = 10, n = 20;

                printf("m = %d\n", m);

                printf("n = %d\n\n", n);

                swap(&m, &n);    //passing m and n to the swap function

                printf("After Swapping:\n\n");

                printf("m = %d\n", m);

                printf("n = %d", n);

                return 0;

        }

//pointer 'a' and 'b' holds and points to the address of 'm' and 'n'

void swap(int  *a, int  *b)

        {

                int temp;

                temp = *a;

                *a = *b;
                *b = temp;

        }
```

Output:

```
        m = 10

        n = 20

        After Swapping:

        m = 20

        n = 10
```

Passing Pointer to Functions:

In pointer to functions, we do same as do in call by reference but in place of passing parameter address we pass pointer as parameter to function. This is also known as **pass by pointer**. When a function is called by passing pointer, any change made to the pointer variable will affect the original variable same as reflected in call by reference.

For Example: Swapping two numbers using Pointer

```
        #include <stdio.h>

        void swap(int *a, int *b);

        int main()

        {

                int m = 10, n = 20;

                int *mm, *nn;

                *mm=&m;

                *nn=&n;

                printf("m = %d\n", m);

                printf("n = %d\n\n", n);

                swap(mm, nn);    //passing address of m and n to the swap function

                printf("After Swapping:\n\n");

                printf("m = %d\n", m);

                printf("n = %d", n);

                return 0;

        }

// pointer 'a' and 'b' holds and points to the address of 'm' and 'n'

void swap(int *a, int *b)

        {
                int temp;
                temp = *a;
                *a = *b;
                 *b = temp;
        }
```

Output:

```
        m = 10
        n = 20
        After Swapping:
        m = 20
        n = 10
```

Functions returning Pointer variables:

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function don't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>
int* larger(int*, int*);
void main()
{
        int a = 15;
        int b = 92;
        int *p;
        p = larger(&a, &b);
        printf("%d is larger",*p);
}
int* larger(int *x, int *y)
{
        If (*x > *y)
        return x;
        else
        return y;
}
```

Output:

```
92 is larger
```

Safe ways to return a valid Pointer:

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.
2. Or, use static **local variables** inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available throughout the program.

Unit 3	**Lecture 7**	**Array of pointers, pointer to functions, pointer to pointer and arrays of pointers**

Array of Pointers:

Array of Pointers is also called array pointer. It is just like normal array but in place of storing normal value it stores address of variable of same type.

Syntax:

```
int *arr[4];
```

Let understand it with proper example:

```
int main()
{
        int *arr[4];
        int a=20,b=45,c=67,d=78;
        arr[0]=&a;
        arr[1]=&b;
        arr[2]=&c;
        arr[3]=&d;
        printf("Value of Array of Pointer are\n");
        for(int i=0;i<4;i++)
        {
        printf("%d",*(arr[i]));
        }
}
```

Pointer to Array:

It is pointer variable that point to an array is called Pointer to Array, and then we can use that pointer to access the array elements.

Let's have an example to understand it:

```
int main()

{

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

        int I, *j;

        j=a;

        printf("Values of arrays are =\n");

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

        {

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

        }

}
```

Pointer to Pointer:

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator (*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

```
int ***p2
```

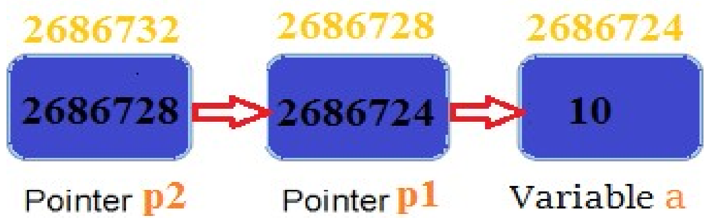
Let's understand it by example:

```
#include <stdio.h>

int main() {

         int  a = 10;

         int  *p1;      //this can store the address of variable a

        int  **p2;

        /* this can store the address of pointer variable p1 only.

          It cannot store the address of variable 'a'
*/
p1 = &a;
p2 = &p1;
printf("Address of a = %u\n", &a);
printf("Address of p1 = %u\n", &p1);
printf("Address of p2 = %u\n\n", &p2);
// below print statement will give the address of 'a'
printf("Value at the address stored by p2 = %u\n", *p2);
printf("Value at the address stored by p1 = %d\n\n", *p1);
printf("Value of **p2 = %d\n", **p2); //read this *(*p2)
/*This is not allowed, it will give a compile time error-
        p2 = &a;
        printf("%u", p2);
*/
return 0;
}
```

Output:

```
Address of a = 2686724
Address of p1 = 2686728
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
Value of **p2 = 10
```

Explanation of the above program:



1. p1 pointer variable can only hold the address of the variable a (i.e Number of indirection operator (*)-1 variable). Similarly, p2 variable can only hold the address of variable p1. It cannot hold the address of variable a.

2. *p2 gives us the value at an address stored by the p2 pointer. p2 stores the address of p1 pointer and value at the address of p1 is the address of variable a. Thus, *p2 prints address of a.
3. **p2 can be read as *(*p2). Hence, it gives us the value stored at the address *p2. From above statement, you know *p2 means the address of variable a. Hence, the value at the address *p2 is 10. Thus, **p2 prints 10.

Pointer to functions:

In C programing, It is also possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
type (*pointer-name)(parameter);
```

Here is an example:

```
int (*sum)();   //legal declaration of pointer to function

int *sum();     //This is wrong declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);

int (*s)(int, int);

s = sum;
```

Here's is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```
s (10, 20);
```

Let's understand Pointer to Function with example:

```
#include <stdio.h>

int sum(int x, int y)

{

        return x+y;

}

int main( )

{

        int (*fp)(int, int);

        fp = sum;

        int s = fp(10, 15);

        printf("Sum is %d", s);

        return 0;

}
```

Output:

```
25
```

Let's understand complicated Function Pointer example:

You will find a lot of complex function pointer examples around, lets see one such example and try to understand it.

```
void *(*foo) (int*);
```

It appears complex but it is very simple. In this case (*foo) is a pointer to the function, whose argument is of int* type and return type is void*.

Unit 3	Lecture 8	Introduction, initializing strings, Accessing string elements

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

Declaring and Initializing a string variables:

There are different ways to initialize a character array or string variable.

```
char name[10] = {'L','e','s','s','o','n','s','\0'};     // valid initialization
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

```
char name[13] = "StudyTonight";      // valid character array initialization

char name[]    = "StudyTonight";      //it also valid
```

Difference between above declarations are, when we declare char as "name[13]", 13 bytes of memory space is allocated for holding the string value.

And, when we declare char as "name[]", memory space will be allocated as per the requirement during execution of the program.

To understand it clearly, let's take an another example:

```
char msg[5]="Welcome to C";
```

//compiler raises an error because the size of array is not enough for string

```
char msg[20]="Hello";
```

//remaining 14 locations automatically filled with null characters

Accessing String Elements:

Accessing individual character in a string is just as accessing elements in the array. The index of the character should be specified as array subscript.

For Example:

```
char c[]="Hello";
```

The expression 'c[0]' accesses the first character 'H'

'c[1]' accesses the third character 'e',……….

Similarly remaining elements are accessed.

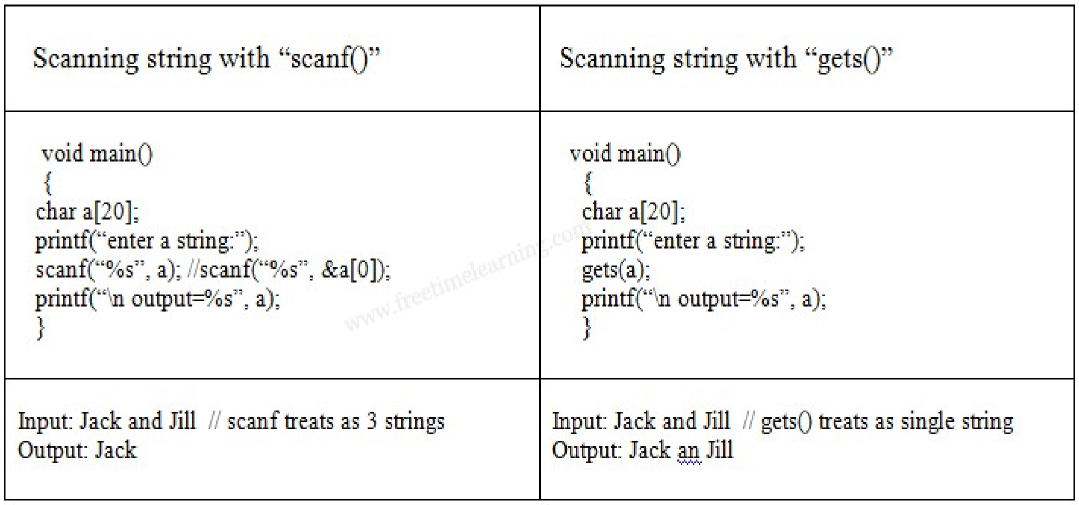
Accessing String Input and Output:

The following are the input and output functions of strings in c

1. **Input functions: scanf(), gets()**
2. **Output functions: printf(), puts()**

The **scanf()** and **printf()** are generic **i/o** functions that they support all built-in data types such as int, float, long, double, strings,..etc. But **gets()** and **puts()** are specialized to scan and print only string data. There is a little difference between **scanf()** and **gets()**, while reading string from keyboard, the **scanf()** accepts character by character from keyboard until either a new line ('**\n**') or blank space is found, whichever comes earlier. Whereas "**gets()**" accepts until a newline is found. That is it accepts white spaces & tab also, these input functions append a null character at end of string, the formatted string %s is used in **printf()** and **scanf()**.

For Example:



Scanning string with "scanf()"	Scanning string with "gets()"

void main()
{
char a[20];
printf("enter a string:");
scanf("%s", a); //scanf("%s", &a[0]);
printf("\n output=%s", a);
}
	void main()
	{
	char a[20];
	printf("enter a string:");
	gets(a);
	printf("\n output=%s", a);
	}

Input: Jack and Jill // scanf treats as 3 strings Output: Jack	Input: Jack and Jill // gets() treats as single string Output: Jack an Jill

The **scanf()** function consider the jack and Jill as 3 strings, whereas **gets()** considers as single string. In case to scan total string using **scanf()**, then it should be **scanf("%s%s%s", a,b,c);** here **a,b,c** are three arrays.

The **printf()** and **puts()** is work in similar way. All **I/O** functions take first byte address (base address of array) as argument and prints the given string using pointer.

Let's understand it by an example:

```
#include<stdio.h>

#include<string.h>

int main()

{
        char name[30];

        printf("Enter name: ");

        gets(name);     //Function to read string from user.

        printf("Name: ");

        puts(name);    //Function to display string.

        return 0;

}
```

Output:

```
        Enter name: C Programming in KIET
        Name: C Programming in KIET
```

Unit 3	**Lecture 9**	**Array of strings, Passing strings to functions, String functions.**

Array of Strings:

A string is a 1-D array of characters, so an array of strings is a 2-D array of characters. Just like we can create a 2-D array of int, float etc; we can also create a 2-D array of character or array of strings. Here is how we can declare a 2-D array of characters.

```
char string_arr[3][10] = {

                    {'K', 'I', 'E', 'T', '\0'},

                    {'M', 'C', 'A','\0'},

                     {'D', 'E', 'A', 'N','\0'}

                 };
```

It is important to end each 1-D array by the null character, otherwise, it will be just an array of characters. We can't use them as strings.

We can also declare it as shown below, this is equivalent to above one:

```
        char string_arr[3][10] = {

                            "KIET",

                            "MCA",

                            "DEAN"

                        };
```

Same can be declare using pointer like as:

```
         char *string_arr[10] = {

                            "KIET",

                            "MCA",
                                        "DEAN"
                                    };
```

Let's understand it by an example:

```
int main()
{
        int i;
        char name[2][8] = {
                                "KIET",
                                "MCA"
                        };
        for (i = 0; i < 2; i++)
        {
                printf("String = %s \n", name[i]);
        }
        return 0;
}
```

Output:

```
        String = KIET
        String = MCA
```

Passing Strings to Functions:

As strings are character arrays, so we can pass strings to function in a same way we pass an array to a function.

Let's understand it by an example:

```
#include<stdio.h>
void printStr(char str[])
 {
      printf("String is : %s",str);
 }
int main()
 {
      // declare and initialize string
```

```
    char str[] = "String to Functions in C";
   // print string by passing string to a different function
    printStr(str);
    return 0;
}
```

Output:

```
    String is: String to Function in C
```

String Functions:

C language supports a large number of string handling functions that can be used to carry out many string manipulations operations. These functions are packaged in "string.h" library. Hence, you must include "string.h" header file in your programs to use these functions.

The following are the most commonly used string handling functions.

1. **strlen( ) Function:**

strlen( ) function is used to find the length of a character string.

Example:

```
int  n;
char st[20] = "Bangalore";
n = strlen(st);
```

- This will return the length of the string 9 which is assigned to an integer variable n.
- Note that the null character "**\0**" available at the end of a string is not counted.

2. **strcpy( ) Function :**

strcpy( ) function copies contents of one string into another string. Syntax for strcpy function is given below.

Syntax: **char * strcpy (char * destination, const char * source);**

Example:

strcpy (str1, str2) – It copies contents of str2 into str1.

strcpy (str2, str1) – It copies contents of str1 into str2.

If destination string length is less than source string, entire source string value won't be copied into destination string.

Example:

```
char  city[15];
strcpy(city, "BANGALORE") ;
```

This will assign the string "BANGALORE" to the character variable city.

3. **strcat( ) Function :**

strcat( ) function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for **strcat( )** function is given below.

Syntax: char * strcat (char * destination, const char * source);

Example:

strcat (str2, str1); - str1 is concatenated at the end of str2.

strcat (str1, str2); - str2 is concatenated at the end of str1.

• As you know, each string in C is ended up with null character ('**\0**').

• In **strcat( )** operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after **strcat( )** operation.

4. **strcmp( ) Function :**

strcmp( ) function in C compares two given strings and returns zero if they are same. If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value.

Syntax : int strcmp (const char * str1, const char * str2);

strcmp( ) function is case sensitive. i.e., "A" and "a" are treated as different characters.

Example :

```
char city[20] = "Madras";
char town[20] = "Mangalore";
strcmp(city, town);
```

This will return an integer value "-10‟ which is the difference in the ASCII values of the first mismatching letters "D‟ and "N‟.

* Note that the integer value obtained as the difference may be assigned to an integer variable as follows:

```
int n;
n = strcmp(city, town);
```