MODULE 3

Syllabus

CO3: Develop programs using Pointers to solve problems more efficiently.

M3.01	Explain the concept of pointers and operations on pointers with examples	2	Understanding
M3.02	Illustrate the advantage of passing pointers to functions	1	Understanding
M3.03	Explain dynamic memory allocation concepts in C	1	Understanding
M3.04	Explain the relationship of arrays and pointers	2	Understanding
M3.05	Develop programs for single and multi-dimensional arrays using pointers.	4	Applying

Contents: Pointers – Fundamentals – declaration, Initialization, accessing of pointer variables -Pointer arithmetic – Passing pointers to Functions – dynamic memory allocation - Arrays and Pointers - Strings and Pointers – Array of Pointers.

POINTERS

Pointers-Fundamentals-declaration,initialization,accessing of pointer,pointer arithemetic

---

Definition: **The pointer in C is a special type of variable which stores the address of another variable**.
Advantages of using Pointers in C:

- Improves the performance.
- Frequently used in arrays, structures, graphs and trees.
- Less number of lines needed in a code.
- Useful in accessing any memory location.
- Used in Dynamic Memory Allocation.

Declaration of pointer variable:

The pointer in C language can be declared using * (asterisk symbol) before the name of the variable.. It is also known as indirection operation used to dereference a pointer.
The general form,

```
datatype *pointername;
```

Where ***datatype*** specifies the data type of the object that the pointer can point to.
Eg,

```
int *a;               //pointer to integer
char *c;              //pointer to char
```

Initialization of Pointer Variable

The process of assigning the address of a variable to a pointer variable is known as initialization. All uninitialized pointers have some garbage value so it should be initialized.
Eg,

```
int a=10;
int *p;                  //declaration
p=&a;                    //initialization
```

Here declaration tells the compiler that “p” is a pointer to an integer and initialization assign address of “a” to pointer “p”.

Accessing Data through Pointers

A variable can be accessed through its pointer using unary operator ’*’. Also known as indirection operator or dereferencing operator.

Consider the following example:

```
#include<stdio.h>
void main()
{
int n = 10;
int *p;
p = &n;
printf("Address of n=%u\n",&n);
printf("Value of n=%d\n",n);
printf("Address of p=%u\n",&p);
printf("Value of p=%u\n",p);
printf("Value of *p=%d\n",*p); // *p = value of n;
}
```

OUTPUT

```
Address of n=6487628
Value of n=10
Address of p=6487616
Value of p=6487628
Value of *p=10
```

Each variable has two properties: address and value. Address of a variable is the memory location allocated for that variable. Value of a variable is the value stored in the memory location . The address of a variable can be derived using address of (&) operator. In the example above, the address of n is 6487628 and value is 10.

Variable name	Address(**&n**)	Value of n(**n)**
n	**6487628**	***10***

When p is assigned with &n, p stores the address of n. Thus p points to n. In order to retrieve the value of n, we can make use of *p.

Pointer name	Address of p(**&p**)	Value of p(**p**)	***P**
p	6487616	**6487628**	***10***

NULL Pointer

If you don't have any address to be specified in the pointer at the time of declaration, you can Initialize pointer variable with NULL.A pointer that is not assigned any value, but NULL, is known as the NULL pointer.
It means pointer doesn't point to any valid memory location.
General form,

```
datatype *pointer_name=NULL;
```

eg,
int *p=NULL;

Pointer Arithmetic

1. Increment
2. Decrement
3. Addition
4. Subtraction

1.Increment pointer(++):If we increment a pointer by 1, the pointer will start pointing to the immediate next location. Since the value of the pointer will ***get increased by the size of the data type to which the pointer is pointing***.
We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array.

```
new_address= current_address + 1 * size_of(data type)
```

Eg,
For 64-bit int variable, it will be incremented by 4 bytes

```
ptr++;                    //equals ptr = ptr+1
```

2.Decrement pointer(--):Decrementing a pointer in C simply means to decrease the pointer value step by step to point to the previous location.

new_address= current_address -1 * size_of(data type)

eg,

For 64-bit int variable, it will be decremented by 4 bytes

ptr - -;   //equals ptr = ptr-1;

3.Pointer Addition(+):Addition in pointer in C simply means to add a value to the pointer value.

Next Location = Current Location + (n * size_of(data type))
'n'=value To Add

Eg,

For 64-bit int variable, it will be incremented by ***n*4 bytes***

ptr=ptr+3;   //equals ptr=ptr+3*4 bytes

4.Pointer subtraction(-):subtraction pointer in C simply means to subtract a value to the pointer value.

Next Location = Current Location - (n * size_of(data type))
'n'=value To Subtract

Eg,

For 64-bit int variable, it will be decremented by ***n*4 bytes***

ptr=ptr-2;   //equals ptr=ptr-2*4 bytes

Passing pointers to function

PASS BY REFERENCEIn C programming, it is also possible to pass addresses as arguments to functions. To accept these addresses in the function definition, we can use pointers. In this method, the address of the actual parameters is passed to formal parameters. So any change in formal parameters will be reflected in the actual parameters.

Eg,

Consider the program to swap two numbers using pass by reference method,

```
#include<stdio.h>
        Formal parameters
                ↓
int swap(int *a,int *b)
{
int temp=*a;
*a= *b;
*b=temp;
}

void main()
{
int x,y;
printf("\nEnter the numbers:");
scanf("%d%d",&x,&y);
printf("\nBefore swapping : x=%d\ty=%d\n",x,y);
   actual parameters
        ↓
swap(&x,&y);
printf("\nAfter swapping : x=%d\ty=%d",x,y);
}
```

OUTPUT

```
Enter the numbers:10 20
Before swapping: x=10 y=20
After swapping: x=20 y=10
```

Differences between pass by value and pass by reference

Pass by Value	**Pass by Reference**
In call by value, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.	In call by reference, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within from the called function.

In call by value, actual arguments will remain safe, they cannot be modified accidentally.	In call by reference, alteration to actual arguments is possible within called function.

Arrays and pointers

1-D array

When an array is declared,the compiler allocates a block of contiguous memory locations to suit the number of elements.The base address is the location of first element (index 0) of an array.When we initialize a pointer with array,then pointer points to the base address of the array.
Eg,
To use a pointer to an array, and then use that pointer to access the array elements

```
#include<stdio.h>
void main()
{
  int a[3] = {1, 2, 3};
  int *p;
  p = a;   //pointer points to base address of array a.(p=&a[0])
  for (int i = 0; i < 3; i++)
  {
    printf("%d \t", *p);       //pointer prints value stored in the address
    p++;                       //increment pointer by 1 to get address of next element
  }
 return 0;
}
```

Output:

```
1        2        3
```

***(a+i) is same as a[i]**

Eg,
*(a+0)=a[0]=1
*(a+1)=a[1]=2
*(a+2)=a[2]=3

2-D array

Let's see how to make a pointer point to a multidimensional array. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0] element.

***(*(a + i) + j)**

The following program demonstrates how to access values and address of elements of a 2-D array using pointer notation.

```c
#include<stdio.h>
int main()
{
    int arr[3][4] = {
                        {11,22,33,44},
                        {55,66,77,88},
                        {11,66,77,44}
                    };

    int i, j;

    for(i = 0; i < 3; i++)
    {
        printf("Address of %d th array %u \n",i , *(arr + i));
        for(j = 0; j < 4; j++)
        {
             printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j)
);
        }
        printf("\n\n");
    }

   return 0;
}
```

Expected Output:

```
Address of 0 th array 2686736
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44

Address of 1 th array 2686752
arr[1][0]=55
arr[1][1]=66
arr[1][2]=77
arr[1][3]=88

Address of 2 th array 2686768
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44
```

Strings and pointers

Eg,

char str[7] = "Hello";
Char *ptr=str;

The above code creates a string str and stores its address in the pointer variable ptr. The pointer ptr now points to the first character of the string "Hello".
The content of the string can be printed using

```
while(*ptr!='\0')
{
        printf("%c",*ptr);
        ptr++;
}
```

And also

```
*ptr=H
*(ptr+1)=e
*(ptr+2)=l
…………
…………
```

Array of pointers

When we want to point at multiple variables or memories of the same data type in a C program, we use an array of pointers.

Declaration of an Array of Pointers in C

An array of pointers can be declared just like we declare the arrays of char, float, int, etc. The syntax for declaring an array of pointers would be:

```
data_type *pointer_array [array_size];
```

Now, let us take a look at an example for the same,
int *ary[55]
This one is an array of a total of 55 pointers. In simple words, this array is capable of holding the addresses a total of 55 integer variables. Think of it like this- the ary[0] will hold the address of one integer variable, then the ary[1] will hold the address of the other integer variable, and so on.

Eg,
Let us take a look at a program that demonstrates how one can use an array of pointers in C:

```
#include<stdio.h>
#define SIZE 10
int main()
{
int *arr[3];
int p = 40, q = 60, r = 90, i;
arr[0] = &p;
arr[1] = &q;
arr[2] = &r;
for(i = 0; i < 3; i++)
{
printf("For the Address = %d\t the Value would be = %d\n", arr[i], *arr[i]);
}
return 0;
}
```

Output

```
For the Address = 387130656 the Value would be = 40
For the Address = 387130660 the Value would be = 60
For the Address = 387130664 the Value would be = 90
```

Dynamic memory allocation

In Dynamic Memory Allocation, memory is allocated at run time, that can be modified while executing program.**<stdlib.h>** header file to facilitate dynamic memory allocation in C

Advantages of Dynamic Memory allocation

1. This allocation method has no memory wastage.
2. The memory allocation is done at run time.
3. Memory size can be changed based on the requirements of the dynamic memory allocation.
4. If memory is not required, it can be freed.

Methods used for Dynamic memory allocation:

Method	**Syntax**	**Uses**
malloc() **Memory allocation**	**ptr = (cast_type*) malloc(number*byte_size)** Eg, **ptr = (int*) malloc(100 * sizeof(int));** Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.	**To allocate a single block of requested memory**
calloc() **contiguous allocation**	**ptr = (cast_type*) calloc(number, byte_size)** Eg, **ptr = (float*) calloc(25, sizeof(float));**	**To allocate multiple block of requested memory**

	This statement allocates contiguous space in memory for 25 elements each with the size of the float.	
realloc()	**ptr= realloc(P, new_size)** **Eg,** **realloc(ptr,200*sizeof(int))**	**To reallocate the memory occupied by malloc() or calloc() function**
free()	**free(ptr)** Eg, **free(ptr)**	**To free the dynamically allocated memory. de-allocate the memory**

malloc()	calloc()
1.It is a function that creates one block of memory of a fixed size.	1.It is a function that assigns more than one block of memory to a single variable.
2.It only takes one argumemt	2.It takes two arguments.
3.It is faster than calloc	3.It is slower than malloc()