UNIT 7
POINTERS

1. Introduction

Pointer is a variable or data structure that contains the memory address or location of another variable. Variables contain the values and pointer variables contain the address of variables that has the value. The normal variable directly accesses their own values whereas a pointer provides indirectly access the values of the variable whose address it stores. Referencing a value through a pointer is called indirection or dereference.

The asterisk (*) operator, also known as unary or indirection operator, is used before the pointer variable name and it operates only on the pointer variable.

2. Advantages of Pointer

- The pointer enables us to access a variable that is defined outside the function.
- It reduces the length and complexity of a program.
- It allows passing variables arrays, functions, strings and structures as function arguments.
- It supports dynamic memory allocations and de-allocation.
- Pointer can return more than one value.

3. Pointer Declaration and Initialization

A pointer variable is declared with an asterisk (*) operator before the variable name. This operator is called pointer or indirection or dereference operator. The type-specifier determines that what kind of variable the pointer variable points to.

Syntax

data-type *pointer_variable_name;

Every pointer contains garbage values before assignment of some valid address. The uninitialized pointer is called the bad pointer. Pointer variables should be initialized to 0, null or an address. No other constant can be initialized to a pointer variable. Pointer variable of a particular data type can hold only the address of the variable of same data type.

4. Pointer operator

C provides two operators for pointer implementation, which are inverse of each other. They are:

4.1 * (Asterisk) operator

It is also called indirection or de-referencing operator. It represents value at address. It returns the value of the variable to which its operand points.

Syntax: data_type *pointer_variable_name;

Example: int   *myptr;

4.2 & (Ampersand) operator

It is also called the address of operator. It is a unary operator that returns the address of its operand.

Syntax: pointer_variable=&general_variable

Example: mypter=&x;

5. Pointer assignment

A pointer is a variable data type and hence the general rule to assign value to the pointer is same as that of any other variable data type. For example:

```
int  x,  y;
int  *ptr1,*ptr2;
```

Operation	**Description**
ptr1=&x;	The memory address of variables x is assigned to the pointer variable ptr1.
y=*ptr1;	The contents of the pointer variables is assigned to the variable y (i.e. value of x), not the memory address

ptr2=ptr1;	Address of ptr1 is assigned to the ptr2.

Representation	**Description**
int *p	The pointer variable p is integer type pointer. In other word, p is capable to hold the value of integer type variable.
int *p[10]	p is a 10-element array of pointers to integer quantities.
int (*p)[10]	p is a pointer to an array of 10 integers. The parenthesis ( ) is necessary here, otherwise p will be an array of 10 integer pointers.
int (*p) (void)	p is a function that return a pointer to an integer quality.
int *p (char *a)	p is a function that accept an argument which is a pointer to a character and returns a pointer to an integer quality.

<u>Examples showing the use of pointer:</u>

Example1:

```
#include  <stdio.h>
#include  <conio.h>
void  main()
{
int  x=10;
int  *ptr;
ptr=&x;
printf("\n  Address  of  x  =  %u",  &x);
printf("\n  Address  of  x  =%u",ptr);
printf("\n  Address  of  ptr  =  %u",  &ptr); 5
printf("\n  Value  of  x  =  %d",  x);
printf("\n  Value  of  x  =  %d",  *ptr);
printf("\n  Value  of  ptr  =  %u",  ptr);
getch();
clrscr();
}
```

Output:

```
Address  of  x  =  2686684
Address  of  x  =  2686684
Address  of  ptr  =  2686680
Value  of  x  =  10
Value  of  x  =  10
Value  of  ptr  =  2686684
```

Example2:

WAP to assign a character variable to the pointer and to display the contents of the pointer.

```
#include<stdio.h>
#include  <conio.h>
void  main()
{
char  x,y;
char  *ptr;
x='c';  /*assignment  of  character*/
ptr=&x;
y=*ptr;
printf("Value  of  x  =  %c",x);
printf("\nPointer  value  ptr  =  %c",y);
getch();
clrscr();
}
```

Output:

```
Value  of  x  =  c
Pointer  value  ptr  =  c
```

6. Types of pointers

6.1 Void pointer

A pointer that points any data type (int or float or char or double) is called the void pointer. Using void pointer the pointed data cannot be referenced directly. Type casting or assignment must be used to change the void pointer to a concrete data type to which we can refer.

For example:

```
#include <stdio.h>
#include <conio.h>
void main ()
{
int a = 10;
double b = 4.5;
void *ptr;
ptr = &a;
printf ("a = %d", *((int *)ptr));
ptr = &b;
printf ("\nb = %lf", *((double *)ptr));
getch();
clrscr();
}
```

Output:

```
a = 10
b = 4.500000
```

6.2 Null pointer

A null pointer is a special pointer value that points nowhere or nothing. We can define the null pointer using predefined constant NULL, which is defined in the header files: stdio.h, stdlib.h, string.h.

For example: int *ptr = NULL;

6.3 Double pointer (pointer to pointer or double indirection or multiple indirection)

C allows the use of pointers that point to other pointers, and these, in turn, point to data. For this operation, we need to add asterisk (*) for each level of reference.

For example:

```
#include <stdio.h>
#include <conio.h>
void main ()
{
        int a = 10;
        int *p;
        int **q;
        p = &a;
        q = &p;
        printf ("a=%d", *p);
        printf ("\na=%d", **q);
        getch();
        clrscr();
}
```

Output:

```
a=10
a=10
```

7. Pointer Arithmetic

As a pointer holds the memory address of variable, some arithmetic operations can be performed with pointers. C and C++ support four arithmetic operators that can be used with pointers, such as:

Pointer arithmetic	**Symbol**

Addition	+
Subtraction	-
Increment	++
Decrement	--

Pointer is variables that hold that memory address of another variable. They are not integers, but they can be displayed as unsigned integers. According to data type declared to the pointer variable, if arithmetic operation is done then values (contents) will be incremented or decremented as per the data type chosen.

Let p = &a[0] = 1001 i.e. Base address of an array

int type pointer
(2-byte space)

p++ = p+1 = 1001 + 2 = 1003 i.e. Address of next element
p = p+5 = 1001 + 10 = 1011 i.e. Address of 5th integer type element.

float type pointer
(4-byte space)

p++ = 1001 + 4 = 1005 i.e. Address of next element
p = p+5 = 1001 + 20 = 1021 i.e. Address of 5th float type element.

char type pointer
(1-byte space)

p++ = 1001 + 1 = 1002 i.e. Address of next element
p = p+5 = 1001 + 5 = 1006 i.e. Address of 5th char type element.

Note: Same operation is done for decrement.

Invalid pointer operation

- Addition of two pointer
- Addition or subtraction of float or double data type to or from a pointer.
- Multiplication of pointer with a constant.
- Division of two pointer with a constant

Illegal operations: p1/p2 p1*p2 p1+p2 p1/5

Example 1: **WAP to increment the pointer's address and display it.**

```
#include <stdio.h>
#include <conio.h>
void main()
{
	int value=20,*ptr;
	ptr=&value;
	printf("ptr=%u",ptr);
	ptr++;
	printf("\nptr=%u",ptr);
	getch();
	clrscr();
}
```

Output:

```
ptr=1310548
ptr=1310550
```

Example 2: WAP to show the arithmetic operation on pointers.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        float value=12.5,*ptr;
        int x,*ptr1,*ptr2;
        ptr=&value;
        printf("\nMemory address=%u",ptr);
        ptr++;
        printf("\nMemory address after increment=%u",ptr);
        ptr--;
        printf("\nMemory adderss after decrement=%u\n",ptr);
        x=101;
        ptr1=&x;
        printf("ptr1=%u",ptr1);
        ptr2=ptr1+6;
        printf("\nCotents of x=%d",*ptr1);
        printf("\nAddress of ptr1=%d",&ptr1);
        printf("\nAddress of (ptr2=ptr1+6)=%d",&ptr2); getch();
        clrscr();
}
```

Example-3:

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int x,y,*ptr;
        x=10;
        ptr=&x;
        printf("Value of x=%d and pointer=%d",x,*ptr);
        y=++ *ptr;
        printf("\nValue of y=%d and pointer=%d",y,*ptr); getch();
        clrscr();
}
```

Output:

```
Value of x=10 and pointer=10
Value of y=11 and pointer=11
```

Example-4:

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int x, y, *x_pointer, temp;
        temp=3;
        x=5*(temp+5);
        x_pointer=&temp;
        y=6*(*x_pointer+5);
        printf("x=%d",x);
        printf("\ny=%d",y);
        getch();
clrscr();
}
```

Output:

```
x =40
y =48
```

8. Pointers and Functions

Pointers are very much used in function declaration. Sometimes only with a pointer a complex function can be easily represented and accessed. The use of the pointers in a function definition may be classified into two groups they are call by value and call by reference.

Call by value

```
#include  <stdio.h>
#include  <conio.h>
void  swap  (int,  int);
void  main()
{
  int  x=100,y=20;
printf("Values   before swap\n");
printf("x=%d  y=%d",x,y);
swap(x,  y);
getch();
clrscr();
}
void  swap(int  a,int  b)
{
int  temp;
temp=a;
a=b;
b=temp;
printf("\nValues  after swap\n");
printf("x=%d  y=%d",a,b);
}
```

Call by Reference

```
#include  <stdio.h>
#include  <conio.h>
void  swap  (int  *x,  int  *y);
void  main()
{
int  x=100,y=20;
printf("Values   before swap\n");
printf("x=%d  y=%d",x,y);
swap(&x,&y);
printf("\nValuesafter swap\n");
printf("x=%d  y=%d",x,y);
getch();
clrscr();
}
void  swap(int  *a,int  *b)
{
int  temp;
temp=*a;
*a=*b;
*b=temp;
}
```

Output of both:

```
Values  before  swap
x=100  y=20
Values  after  swap
x=20  y=100
```

In CALL BY VALUE method, the called function creates its own copies of the original values sent to it. Any changes, that are made, occur on the called function's copy of values and are not reflected back to the calling function.
In CALL BY REFERENCE method, the called function accesses and works with the original values using their references. Any changes, that occur, take place on the original values and are reflected back to the calling code.

Returning multiple values from function using pointer

The use of return() function, we can return only one value but it is unable to return more than one value at a time. This operation can be possible using pointer in the function by "call by reference" method.

For example:

```
#include <stdio.h>
#include <conio.h>
void fun (int radious, float *area, float *perimeter);
void main()
{
        int radious;
        float area, perimeter;
        printf("Enter radious of circle:");
        scanf ("%d", &radious);
        fun(radious, &area,&perimeter);
        printf("\nArea = %f", area);
        printf("\nPerimeter=%f",perimeter);
        getch();
        clrscr();
}
void fun(int r, float *a, float *p)
{
        *a=3.14*r*r;
        *p=2*3.14*r;
}
```

9. Pointers and Arrays

Array is used to store the similar data items in contiguous memory locations under single name. Pointer addressing is in the form of absolute addressing. Exact location of the elements can be accessed directly by assigning the starting location of the array to the pointer variable. The pointer variable is incremented to find the next element.

Accessing various elements in array :

1. For 1D array: a[i] = *(p+i) = *(i+p) are same and *a =*(a+0) is 0th element of a
2. For 2D array: a[i][j] = *(*(p +i)+j)

Here in two dimensional array,
p=pointer to first row
p+i=pointer to ith row
*(p+i)=pointer to first element in ith row
*(p+i)+j=pointer to jth element in ith row
((p+i)+j)=value stored in the cell (i,j) i.e value stored in ith row and jth column

Example-1: A program to access the array element using pointer.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int Array[5]={1,2,3,4,5};
int i;
for(i=0;i<5;i++)
{
printf("%d\t",*(Array+i));
}
getch();
clrscr();
}
```

Output: 1	2	3	4	5

Example-2: A program to access the array element and particular memory location using pointer.

```
#include  <stdio.h>
#include  <conio.h>
void  main()
{
 int  x[5]={10,20,30,40,50},  i,  *p;
p  =  &x[0];
for(i=0;i<5;i++)
{
printf("\nArray    element = %d\t  and  Memory  location =  %u", *p,  p);
p = p+1;
}
getch();
clrscr();
}
```

Example 3:

```
#include  <stdio.h>
#include  <conio.h>
void  main(  )
{
int  num[  ]  =  {  1,  2,  3,  4,  5}  ; int  i  ;
for  (  i  =  0  ;  i  <=  4  ;  i++  )
{
printf  (  "\naddress  =  %u  ",  &num[i]  )  ;
printf ("element =          %d       %d       %d ",  num[i], *(num+i), *(i+num)) ;
}
getch();
clrscr();
}
```

Output:

```
Address = 65512 Element = 1 1 1
Address = 65514 Element = 2 2 2
Address = 65516 Element = 3 3 3
Address = 65518 Element = 4 4 4
Address = 65520 Element = 5 5 5
```

Example 4: WAP to display the contents of 2-Dimensional array using pointer arithmetic.

```
#include  <stdio.h>
#include  <conio.h>
void  main()
{
int  a[2][3]={ {11,12,13}, {14,15,16} },i, j, n, temp;
printf("Contents  of  the  array\n");
 for(i=0;i<=1;i++)
{
for(j=0;j<=2;j++)
{
temp=*(*(a+i)+j);  //*(*(a+i)+j)  =  a[i][j]
printf("%d\t",temp);
}
printf("\n");
}
getch();
clrscr();
}
```

9.1 Passing arrays using pointers

As we passed the variables onto the function, using pointer arrays can be passed as arguments to functions in the same way and their elements can be accessed by the function. However, it's more common to use pointer notation instead of array notation when arrays are passed to the function.

Example:

```
#include <stdio.h>
#include <conio.h>
void fun(int *ptr);
void main()
{
        int Array[5]={10,12,14,16,18}, i;
        fun(Array);
for( i=0; i<5; i++)
{
        printf("Array[%d]:%d\n", i, Array[i]);
}
getch();
clrscr();
}
void fun(int *p)
{
int j;
for(j=0;j<5;j++)
{
*p=(*p)*2;
p++;
}
}
```

Output:

```
Array[0]  =  20
Array[1]  =  24
Array[2]  =  28
Array[3]  =  32
Array[4]  =  36
```

10. Pointers and strings

Strings are one-dimensional arrays of type "char". In C, a string is terminated by NULL or '\0'. String constants are written in double quotes. Pointers can be used to access the string.

Example 1: **WAP to input any string and display it.**

```
#include <stdio.h>
#include <conio.h>
void main()
```

```
{
char  s[10],*ptr;
int i;
printf("Enter  a  word:");
scanf("%s",s);
ptr=s;
printf("\nOutput:");
for(i=0;i<=10;i++)
{
printf("%c",*ptr);
ptr++;
}
getch();
clrscr();
}
```

Same program using function

```
#include  <stdio.h>
#include  <conio.h>
void  strfunc(char  *s);
void  main()
{
char  s[10];
printf("Enter  a  word:");
scanf("%s",s);
printf("\nOutput:");
strfunc (s);
getch();
clrscr();
}
void  strfunc (char  *ptr)
{
int i;
for(i=0;i<=10;i++)
{
printf("%c",*ptr);
ptr++;
}
}
```

Example 2:

```
#include  <stdio.h>
#include  <conio.h>
void  main()
{
char  str1[]="This  is  a  string  stored  in  an  array";
char  *str2="This  is  string  stored  using  pointer";
printf("%s",str1);
printf("\n%s",str2);
str2+=5;              //str2+=5  is  same  as  str2=str2+5
printf("\n%s",str2);
getch();
clrscr();
}
```

Output:

```
This is a string stored in an array
This is string stored using pointer
 is string stored using pointer
```

11. Dynamic Memory Allocation (DMA)

The process of allocating memory at run time is known as dynamic memory allocation. For example: suppose the size of an array has been fixed to 100. If we just enter 10 elements only then there will be wastage of 90 memory location. In this case the use of DMA is more memory efficient that dynamically allocates and frees the memory at execution time. There are four library functions: malloc(), calloc(), free(), and realloc() for memory management. These functions are defined within header file stdlib.h. The function calloc() and malloc() used to create the space for array, structure, and union dynamically. The difference in malloc and calloc is that malloc does not set the memory to zero whereas calloc sets allocated memory to zero. Generally, the malloc() takes less time.

11.1 malloc ()

The term malloc stands for memory allocation. It is dynamic memory allocation which allocates the memory and returns a pointer to it. It initializes garbage value and allocates memory as a single contiguous block. If a single contiguous block cannot be allocated then malloc would fail. malloc take one argument and allocate bytes of memory. We do not need to free the memory explicitly while using malloc. Variables used with malloc are destroyed automatically. The malloc() returns a void pointer so we need to convert into specific datatype.

<u>Syntax</u>:

pointervariable= (cast-type *) malloc(size_in_bytes);

OR

pointervariable= (cast-type *) malloc (number of elements * size of each element in bytes);

This function returns a pointer to the allocated memory, or NULL if the request fails.

Example: Write a program to dynamically allocate memory for the array elements using malloc() function and read and display the array elements

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
int i, n;
int *a;
printf("Number of elements to be entered:");
scanf("%d",&n);
a = (int*)malloc(n*sizeof(int));          //a=(int *)malloc(n*2);since size of “int” is 2 bytes
printf("\nEnter %d numbers:\n",n);
for( i=0 ; i < n ; i++ )
{
scanf("%d",&a[i]);
}
printf("The numbers entered are: ");
for( i=0 ; i < n ; i++ )
{
printf("%d ",a[i]);
}
getch();
clrscr();
}
```

Output

```
Number of elements to be entered:4
Enter 4 numbers:
25
13
14
21
The numbers entered are: 25 13 14 21
```

11.2 calloc()

The term calloc stands for contiguous allocation. It is similar to malloc but difference that it initializes zero. It allocates memory which may/may not be contiguous. It takes two argument (calloc(no.of.var, size of each var) and allocate block of memory. The calloc will not fail if memory can be allocated in non-contiguous blocks when a single contiguous block cannot be allocated. But we need to free the memory explicitly while variables are created using calloc.

Syntax:
Pointervariable= (cast-type *) calloc(number_of_blocks, size_of_each_block_in_bytes);

This function returns a pointer to the allocated memory, or NULL if the request fails.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main()
{
int i, n;
int *a;
printf("Number of elements to be entered:");
scanf("%d",&n);
a = (int*)calloc(n, sizeof(int));      //a=(int *)calloc(n,2);since size of “int” is 2 bytes
printf("\nEnter %d numbers:\n",n);
for( i=0 ; i < n ; i++ )
{
scanf("%d",&a[i]);
}
printf("The numbers entered are: ");
for( i=0 ; i < n ; i++ )
{
printf("%d ",a[i]);
}
getch();
clrscr();
}
```

Output:

```
Number of elements to be entered:3
Enter 3 numbers:
22
55
14
The numbers entered are: 22 55 14
```

11.3 free ()

This build in function frees previously allocated space by calloc(), malloc() or realloc() functions. The memory dynamically allocated is not returned to the system until the programmer returns the memory explicitly. This can be

done using free() function. Thus, the free() function is used to release the space when it is not required. **<u>Syntax</u>**:
free(ptr); where, ptr be any pointer to a memory block created already.

11.4 realloc ()

This function is used to modify the size of previously allocated space. Sometimes, the previously allocated space is not sufficient; we need additional space and sometimes the allocated memory is much larger than necessary. In both situations, we can change the memory size already allocated with the help of function realloc ().

Syntax:

```
ptr = malloc (size);      //Original allocation of memory
ptr = realloc (ptr, newsize);   //Reallocation of space
```

WAP to convert the string to uppercase letters using pointer.

```
#include<stdio.h>
#include<conio.h>
void uppercase(char *);
void main()
{
        char str[11]="engineering";
        int i;
        uppercase(str);
        for(i=0;i<=10;i++)
        {
        printf("%c", str[i]);
        }
        getch();
}
void uppercase(char *p)
{
        int i;
        for(i=0;i<=10;i++)
        {
        *p=toupper(*p);
        p++;
        }
}
```

12. Declaration of function that returns pointer

Following is the function declaration syntax that will return pointer.

returnType *functionName(param list);

```
#include <stdio.h>
// function declaration
int *getMax(int *, int *);
int main(void)
{
  // integer variables
  int x = 100;
  int y = 200;

  // pointer variable
  int *max = NULL;
    max = getMax(&x, &y);
  // print the greater value
  printf("Max value: %d\n", *max);
  return 0;
}

// function definition
int *getMax(int *m, int *n)
{
  if (*m > *n)
{
    return m;
  }
  else
{
    return n;
  }

}
```