7. POINTER

Memory is important area where data/information are stored. As computers have primary memory also known as RAM (Random Access Memory). RAM holds currently running program along with data variables. All variables used in a program reside in memory when the program is executed. Computer memory RAM is divided into various of small units locations. Each location is represented by some unique number known as memory address. Each memory location is represented by some unique number which is known as byte. A char data is one byte in size and hence needs one memory location, similarly integer data is two bytes in size so it requires two memory locations. The smaller unit of memory address is byte(0 or 1).

Each byte in memory is accessed with unique address. An address is an integer labeling bytes in memory. The integer is always positive values that range from zero to some positive integer constant corresponding to location in memory. Thus a computer having 1 GB has 1024*1024*1024 i.e. 1073741824 bytes. The memory address 65524 represents bytes in memory and it can store data of one byte.

Each variable in C program is assigned space in memory. When a variable is declared, it tells computer to hold in memory in appropriate size variable. According to type of variable declared, the required memory locations are reserved. int a=30; float b; char c. here a reserve 2 bytes similarly char c holds 1 bytes.

WAP to display memory location reserved by variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a;  a=20;
        printf("\n the address of a is:\t%u",&a);
        printf("\n the value of a is: \t%d",a);
        getch();
}
```

Here &a denote address of variable. The variables takes two bytes data in memory. As it's of type is int. It takes two memory area locations, although it takes a block of two bytes memory consisting two different addresses, this memory is accessed by first address. The first address is only shown which known as base address. Again control string %u is used to display address of variables as value of address is always positive number.

WAP to illustrate address reserved by different data types

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a=10,b=20;float c=10.5;  char d='M';
        clrscr();
        printf("\n The base address of a\t%u",&a);
        printf("\n The base address of b\t%u",&b);
        printf("\n The base address of c\t%u",&c);
        printf("\n The base address of d\t%u",&d);
        getch();
}
```

Output:

The base address of a	**1045064**
The base address of b	**1045060**
The base address of c	**1045056**
The base address of d	**1045055**

7.1. Introduction to Pointer

A pointer is a variable that contains a memory address of another variable. Normally a pointer variable is declared some way like other variables in C. So that it will work only with address of another variable. Just as integer variable can hold only integer, a character variable can hold only character type, each pointer variables can point only to one specific type (int ,float, char).

Pointer can have name that is legal for other variable and it is declared in some fashion like other variable but while declaration we preceded by * (asterisk/indirection operator) in front of variable name.

For Eg:

- ***int num;*** Here num is normal variable of integer it can store only integer value.
- ***int *p;*** It signifies p is pointer variable it can store address of integer variable.
- The address of float variable cannot be stored in p. Eg: ***int *p; float num; p=&num*** is not allowed

7.1.1. Pointer Declaration

Pointer variable can be declared as:

data type *variable name;

For Eg:

int *s; char *p;

The first s is an integer pointer and it tells compiler that holds address of any int variables. In same way char pointer that store address of any char variable.

WAP to demonstrate pointer variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int v=10,*p; //*p is pointer variable
        p=&v;  //the address of variable v is assigned to p
        printf("\n address of v is\t%u",&v); // display address of v directly
        printf("\n the address v is\t%u",p); // display address of v indirectly
        printf("\n value of v is\t%d",v);  // display value directly
        printf("\n value of address v is \t%d",*p); //display value of p indirectly
        printf("\n address of p=%u",&p); // display address of *p
        getch();
}
```

7.1.2. Indirection or De-reference Operator

The * operator is used inform of a variable, is called pointer or indirection or de-reference operator. Normal variable provides direct access to their own values where as a pointer provides indirect access to variables. The indirection operator (*) is used in two distinct ways within pointers deceleration. When the pointer is de-referenced in form of variable the indirection operator indicates the values at the memory location stored in pointer. The compiler knows automatically which operator to call based on the context of is used.

7.1.3. Address Operator

Ampersand (&) operator is known as address operator thus “&a” denotes address of normal to pointer variable.

7.1.4. Initialization Pointers

Address of some variable can be assigned to a pointer variable at the time of declaration of pointer variable. ***int num; int * ptr=#*** These two statements are equivalent to following statements. ***int num; int *p;p=#***

7.1.5. Bad Pointer

When a pointer variable is first declared it does not have a valid address, the pointer is uninitialized or bad. The de-reference operation on a bad pointer is a serious run time error. Each pointer must have be assigned a valid address before it can support de-reference operators. Before that the pointer is bad and must not be used.

In fact every pointer contains garbage value before assignment of some valid address. Correct code override the garbage value with a correct reference to an address and thereafter the pointer work fine. There is nothing automatic that gives a pointer valid address.

7.1.6. Void Pointer

A void pointer is special type of pointer. It can point to any data type from an integer float or string of characters. Using void pointer the pointer data can be referenced directly. Type casting or assignment must be used to change void pointer to concrete data type to which we can refer.

WAP to illustrate the void pointer

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a=40; double b=4.5;
        void *v;
        clrscr();
        v=&a;
        printf("\n a=%d",*(int*)v);/* not simple*/
        v=&b;
        printf("\nb=%lf",*((double*)v));
        getch();
}
```

7.1.7. Null Pointer

A null pointer is a special pointer value that points nowhere or nothing.

To test a pointer for “NULL” before inspect value pointed to code such as the following can be used.

```
If(ptr!=NULL) printf(“value =%d”,*ptr);
```

7.2. Pointer to Pointer (Double Pointer)

C allows use of pointer that point to other pointer and theses in turn, point to data for pointer to do that we only need to add asterisk for each level of reference.

```
int a=30;
int *p;
int **q;
p=&a;
q=&p;
```

**q is double pointer it requires double pointing to refer to variable address.

Double pointer variable requires two label of indirection while assigning the value we use double address to q variable. As both *p and **q display 30 if they are pointing to the same address.

7.2.1. Arrays of Pointers

An array of pointers can be declared as

```
data type *pointer (size); int *p [5];
```

This declares an array of 5 elements each of which points to an integer, the first pointer is called p [0];

Initially these pointers are uninitialized and they can be used as int a=10,b=100,c=1000; p[0]=&a; p[1]=&a; p[2]=&c;

7.2.2. Relationship Between 1d Array and Pointer

Array name by itself is an address of pointer. It points to address of first element.

Thus if x is one dimensional array then address of first array element can be expressed as either &x [0] or simple x [0]. In general the address of the array element i+1 can be expressed as either &x+i. In the expression x+i, x represents array name whose elements may be integers, characters, float, etc, and I represents integer quantity.

Thus x+i specifies address that is a certain number of memory calls beyond the address of first element. Thus x[i] and *(x+i) both represent same content of that address.

WAP to display elements with their address using name as pointer.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int x[5]={20,40,6,8,100},k;
        clrscr();
        printf("\n aray elements\t elements \t address");
        for(k=0;k<5;k++)
                printf("\ns[%d]\t%d\t%u",k,*(x+k),x+k);
        getch();
}
```

Similarly,

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int x[5]={20,40,6,8,100},k;
        clrscr();
        printf("\n aray elements\t elements\t address");
        for(k=0;k<5;k++)
                printf("\ns[%d]\t%d\t%u",k,x[k],&x[k]);
        getch();
}
```

From the above it can be concluded that in case of array &x[k] is same as x+k and x[k] is same as *(x+k).

7.2.3. One Dimensional Array

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[5]={1,2,3,4,5};
        int i;
        for(i=0;i<5;i++)
        {
                printf("add is\t%d\tand \t value \t%d\n",&a[i],a[i]);/*for Normal
                printf("add is\t%d\tand \t value \t%d\n",(a+i),*(a+i));/*for pointer type
                printf("\n");
        }
        getch();
}
```

7.2.4. Pointer and 2d Arrays

Multidimensional array can also be represented with an equivalent pointer single dimensional array. A two dimensional array is actual a collecting of one dimensional arrays, each indicating a row. It is stored in memory of row form.

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
	int *b[3][3],i,j;
	clrscr();
	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{
			b[i][j]=&a[i][j];
			printf("The value is%d and addressed\n",*b[i][j],&*b[i][j]);
		}
	}
	getch();
}
```

It is possible to access two-dimensional array elements using pointers same way in one dimensional array. Each row of the two dimensional array is treated as a one dimensional array. Each row of two dimensional array as a pointer to a group of contiguous one dimensional array.

Syntax: ***data type (*pt variable) [size2]; instated data type array [size1] [size2];***

Suppose x is two dimensional integer array having 4 row and 5 columns. We can declare x as int (*x) [4];

Rather than int x [4] [5]; it can be concluded that x pointer to first row.

Here in first declaration x is defined to be a pointer to a group of continuous one dimensional element integer array. The x points to first 5 elements array which is actual first row of two dimensional arrays. Similarly x+1 points to the second row 5 elements which is second row of the two dimensional array.

It can be illustrated as:

*x ([0][0])	*x+1 ([0][1])	*x+2 ([0][2])	*x+3 ([0][3])	*x+4 ([0][4])
*(x+1) ([1][0])	*(x+1)+1 ([1] [1])	*(x+1)+2 ([1] [2])	*(x+1)+3 ([1] [3])	*(x+1)+4 ([1] [4])
*(x+2) ([2][0])	*(x+2)+2 ([2] [1])	*(x+2)+2 ([2] [2])	*(x+2)+3 ([2] [3])	*(x+2)+4 ([2] [4])
*(x+3) ([3][0])	*(x+3)+3 ([3] [1])	*(x+3)+2 ([3] [2])	*(x+3)+3 ([3] [3])	*(x+3)+4 ([3] [4])

Two Dimensional Array

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
	int i,j;
	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{
```

```
                        printf("add is\t%d\tand \t value \t%d\n",&a[i][j],a[i][j]);/*for normal
                        printf("add is\t%d\tand \t value \t%d\n",*((a+i)+j),*(*(a+i)+j));/*for
                                pointer
                        printf("\n");
                }
        }
        getch();
}
```

Illustrate 2d array represents in memory.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int p[2][3]={{1,2,3},{4,5,6}};
        clrscr();
        printf("\n%u and%u",p,*(p+1));
        printf("\n*p=%u\t*(p+1)%u",*p,*(p+1));
        printf("\n(*(p+0)+1)%u\t*(*(p+1)+1)%u",*(p+0)+1,*(p+1)+1);
        printf("\n(*(p+0)+1)%u\t*(*(p+1)+1)%u",*(*(p+0)+1),*(*(p+1)+1));
        getch();
}
```

WAP to add to add two m*n matrices using pointer.

```
#include<stdio.h>
#include<conio.h>
#define m 2
#define n 3
void main()
{
        int(*a)[n],(*b)[n],*(sum)[n],i,j;
        clrscr();
        printf("firtst matrix");
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        scanf("%d",*(a+i)+j);
        printf("second matrix");
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        scanf("%d",*(b+i)+j);
        printf("sum is");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        *(*(sum+i)+j)=*(*(a+i)+j)+ *(*(b+i)+j);
                        printf("\t%d",*(*sum+i)+j);
                }
                printf("\n");
        }
        getch();
}
```

WAP to multiply matrices of order m*n and p*q using pointer

```
#include<stdio.h>
#include<conio.h>
#define m 2
#define n 3
#define p 3
#define q 2
void main()
{
        int(*matrix1)[n],(*matrix2)[q],pr[m][q],i,j,k;
        int sum=0;
        clrscr();
        printf("firtst matrix");
        for(i=0;i<m;i++)       for(j=0;j<n;j++)
        scanf("%d",(*(matrix1+i)+j));
        printf("second matrix");
        for(i=0;i<p;i++)       for(j=0;j<q;j++)  scanf("%d",*(matrix2+i)+j);
for(i=0;i<m;i++)  for(j=0;j<p;j++) {  for(k=0;k<n;k++)
sum+=*(*(matrix1+i)+k)**(*(matrix2+k)+j);
*(*(pr+i)+j)=sum;
sum=0;}
printf("\nproduct are\n:");
   for(i=0;i<m;i++){     for(j=0;j<q;j++)
printf("\t%d",(*(*pr+i)+j));
printf("\n");  }
getch(); }
```

7.5. Pointer Operations

To illustrate operations let us consider following declaration of ordinary variable and pointer variables.

int a, b; float c; int *p1,*p2;   float*f;

1. A pointer variable can be assigned the address of ordinary variable. For example p1=&a; p2=&b, f=&c;
2. Content of one pointer can be assigned to other pointer provide they point to same data type. p1=p2; where both of them are pointer type as above.
3. Integer data can be added to pointer variable, p1+2;
4. One pointer can be subtracted from other pointer they must point same elements of array

```
void main(){
                int a[]={1,2,3,4,5},*pf,*p1;
                p1=a;  pf=a+2;     printf("%d",pf-p1);
            }
```

5. The two pointer variables cannot be compared both pointers point to objects of the same data type If(p1>p2){}//invalid
6. There is no sense in ascending an integer to pointer variable. p1=100;p2=200; //invalid
7. Two pointer variables cannot be multiplied together.p1*p2;
8. A pointer variable cannot be multiplied by constant. p1*3;
9. A null value can be assigned to a pointer variable.(p1=NULL)

String and Pointer

As strings are arrays and arrays are closed connected with pointers, we can say that string and pointers are closely related for example: char name [5] =”Shyam”;

As string variable name is an array of 5 characters. It is a pointer to first character of string and can be used to access and manipulate characters of the string. When a pointer to char is pointed in format of string, it will start to print pointer character and then successive characters until end of string is reached.

Dynamic Memory Allocation (DMA)

The process of allocating and freeing memory at run time is known as Dynamic Memory Allocation. This reserves the memory required by the program and returns this valuable resources to the system then reserved space is utilized for variable dynamically.

Through arrays can be used for data storage, they are of fixed size. The programmer must know size of array or data in advance while writing programs. A variable cannot be used to define size of array while declaring an array. In most situations it is not possible to know the size for memory required until run time. For example the size of the array used to store marks of students is fixed. Suppose the size of array used to stored name in character is 100. The program wont work if the number of students is more than 100. If number of students is less than 100 say 10. The 10 memory locations are used and rest 90 locations are reserved but not used, this is wastage of memory occurs. In these situations DMA will very be useful.

Since an array name is actual pointer to first element within the array, it is possible to define array as pointer variable rather than general array. While defining conventional array system reserve fixed block of memory at the beginning of program design which is inefficient but this does not occur if the array is represented in terms of a pointer variable. The use of a pointer variable to represent an array requires some type of initial memory assignment before array elements are processed. This is known as Dynamic Memory Allocation. At execution time a program the compiler request more memory to be free memory. Thus DMA refers allocating and freeing memory at execution time or run time for better memory management.

There are four library functions malloc() calloc(),free() and realloc() for memory useful management. These functions are defined with header file stdlib.h and alloc.h.

malloc () : It allocates requested size of bytes and returns a pointer to first byte of the allocated space.

Syntax:

ptr=data type* malloc (size);

Here ptr is a pointer type data. The malloc () returns a pointer to an area of memory with size specified in size.

eg: s= (int*) malloc (100*size of (int));

A memory space equivalent to 100 times the size of a integer bytes is reserved and the address of the first bytes of allocated memory

calloc () : The function calloc () provides access to the C memory heap which is available for dynamics allocation of variable sized block of memory. Unlike malloc (), function calloc () accepts two arguments no of block and size of each block. The first parameter specifies the number of items to allocate and size of each block specifies size of each items. The function calloc () allocates multiple blocks of storage each of same size and then sets all bytes to zero. The difference between them is that calloc initializes all bytes in the allocated block to zero. Thus it is normally used for requesting memory space at run time for storing derived data types such as and structure. s= (int*) calloc (5, 10*size of (int));

In above statement allocates continuous space for 5 blocks, each of size 20 bytes. We can store 5 arrays each of 10 elements of integer types.

free (): This built-in function free up previously allocated space by calloc, malloc and realloc functions. The memory dynamically allocated is not return to the system until the programmer returns the memories explicitly. This can be done using free() function. This function is used to release the space when it is not required/reserved. free (ptr); Here ptr is pointer to a memory block which must already created.

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

This function allocates the new space size to the pointer variable ptr and returns a pointer to the first byte of the new memory block of memory fails to allocate it return null.

WAP to illustrate the use of realloc() function

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
        char*name;
        clrscr();
        name=(char*) malloc(11);
        strcpy(name,"Happy Singh");
        printf("\nname=%s",name);
        name=(char*) realloc(name,23);
        strcpy(name,"Mr. Happy Singh Sardar");
        printf("\nname=%s",name);
        getch();
}
```

Application of Pointer

There are number of applications of pointer

1. Pointer is widely used in dynamic memory allocation. The functions used for allocation of memory at run time that return the address of allocated memory.
2. Pointer can be used to pass information back and forth between a function and its reference point. Pointer provides a way to return multiple values form arguments to function by reference address.
3. Pointers provide an alternative way to access individual array elements. They used to manipulate arrays more easily by using pointer nested of using array themselves.
4. They increase execution speed as they refer particular address directly.
5. They are used to create complex data structure such as linked list, trees, and graphs.

Passing Pointer to Function

A pointer can be passed to function as an argument. Passing pointer means passing address variable instead of value. As address is passed in this mechanism is also known as call by reference. When pointer is passed to function while calling, formal argument of function must be compatible with the passing pointer. If integer pointer is being passed formal argument in function must be pointer of type. If value in the passed address is changed within function definition, the value of actual variable also changed.

WAP to illustrarate user passing pointer to function

```
#include<stdio.h>
#include<conio.h>

void mul(int *m)
{
        *m=*m+10;
}

void main()
{
        int marks;
        clrscr();
        printf("the actual marks is");
        scanf("%d", &marks);
        mul(&marks);
        printf("\n the numbers are %d", marks);
        getch();
}
```