POINTERS

Pointer is a variable which stores address of another variable.

Declaration

int * p; /* p is a pointer to an int */

***** - value at address

& - address of the variable

int a=10;

int *p ;

p = &a ;

Here the value of variable a is 10; The statement p=&a will store the address of a into pointer variable p. Now p is a pointer to an integer variable a.

```
main()
{
  int a=10;
  int *p;
  p=&a;
  printf ("\n a= %d",a);
  printf ("\n *p= %d",*p);
}
```

Output : 10 10

Consider following example.

int *p;

float *f;

char *c;

Here p is a pointer to an integer variable. It will point to any integer variable only. F is a pointer to float variable and c is a pointer to character variable.

Accessing value through a pointer

Following program shows how to access a value of any variable through the pointer variable.

```
void main()
{
  int * p;
  int a=100;
  printf (" Value of a= %d" ,*p);
  *p=200;
  printf ("\n Value of a= %d ", a);
  printf("\n Value of a= %d ", *p);
}
```

Output :

```
100
200
200
```

In above program value of **a** can be displayed using pointer p also. If we change *p=200, then indirectly a will be changed to 200.

Array and Pointers

Consider arrary s as follows. P is a pointer to the array s.

```
Int  s[5] = {10,20,30,40,50};
int  *p;
p=s;
```

Above statement will store the address of first element of the array s, as shown in following figure.

Now u can perform any operation on array using the pointer p1. Following program shows displaying array values using pointer.

```
#include<stdio.h>
void main()
{
  int s[5] ={10,20,30,40,50};
  int *p;
  p=s;
  printf("\n Array Values are .");

  for(int i=0; i<5;i++)
  {
    printf("\n%d",*p);
    p++;
  }
}
Output :  10
          20
          30
          40
          50
```

Here the statement p++ will each time point to the next element in the array.

Pointer Arithmetic : Address Arithmetic

When we use the any arithmetic operator on any pointer variable then it will act differently as compared to any simple variable.

e.g. int *p;

p++; this statement will increment the pointer variable p as p = p+2; Here +2 because the memory required to store the integer value is 2;

e.q. float *f;

f++; // now f=f+4 because the memory requirement of the float variable is 4 and soon.

Pointer to Pointer

When a pointer variable will store address of another pointer variable then it is called as pointer to pointer.

e.g. int a=100;

int * p1;

p1=&a;

int * p2;

p2= & p1;

Here p2 is a pointer to pointer variable, since it stores the address of another pointer variable p1.

Pointers and Function

There are two types of function calls

1) Call by value
2) Call by reference

In the first method the value of each actual argument in the calling function is copied into corresponding formal arguments of the called function.

```
#include<stdio.h>
void main()
{
   int a=10,b=20;
   printf("Values of a & b before swapping : %d %d",a,b);
   swap(a,b);
   printf("Values of a & b After swapping : %d %d",a,b);
}
void swap (int p, int q)
{
   int temp;
   temp=p;
   p=q;
   q=temp;
}
Output:
Values of a & b before swapping : 10 20
```

```
Values of a & b After swapping : 10 20
```

Note that the values of a and b remains unchanged even after exchanging the values of p and q.

In the second method call by reference, the address of actual arguments are passed to the function swap and therefore the values of the variables a and b are exchanged.

```
#include<stdio.h>

void swap (int *,int *);
void main()
{
   int a=10,b=20;
   printf("\nValues of a & b before swapping : %d %d",a,b);
   swap(&a,&b);
   printf("\nValues of a & b After swapping : %d %d",a,b);
}
void swap (int * p, int * q)
{
   int temp;
   temp=*p;
   *p=*q;
   *q=temp;
}

Output:
Values of a & b before swapping : 10 20
Values of a & b After swapping : 20 10
```

Dynamic memory allocation

- Dynamic memory allocation allows your program to obtain more memory space while running, or to release it if it's not required.
- Dynamic memory allocation allows you to manually handle memory space for your program.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration.

Def : A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
void main ()
{
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
}
```

When the above code is compiled and executed, it produces the following result –

The value of ptr is 0