# POINTERS

## INTRODUCTION:

int x =20;

when complier executes the above statement in the program, then 2 bytes of memory will be registered to variable x and its value 20 is stored in it. That is, whatever the values we assign to variables in program will be stored some where in the memory .

Memory of a computer is divided into equal partitions where each partition is called as **STORAGE CELL**. Each storage cell will have a unique id called as **ADDRESS**.

EX: Each student of a particular class will have a unique id called as ROLL NUMBER. That is no two students of class will have same roll number. It is called as uniqueness. Similarly in memory also each storage cell will have a unique id called as address.

There are 2 ways of accessing the values of variables:-

1. **Directly manipulating their values.**
   Ex: int x=20;
   X=x+2; /* gives value of x as 22 */
   X=x*x /* gives the value of x as 400 */
2. **Accessing the values of variables using their addresses.**
   IN ORDER TO ACCESS THE VALUES OF VARIABLES USUNG THEIR ADDRESS WE HAVE TO USE THE CONCEPT OF **POINTERS.**

**Definition:** pointer is a variable which stores the address of another variable.

**Difference between pointer variable and normal variable is :-**

- Normal variable stores a specific value.
  Ex: int x=20;
  Char c =’p’;
  float f=23.54; etc.,,,,
- Pointer variable sores the address of another variable.

  **Syntax for declaring a pointer:**

  datatype *pointer_name;

Ex: **int *k;** this declaration indicates to compiler that k is a pointer variable and it stores address of another integer variable.

**Sample program-1**

```
int main()
{
  int  x=20;
  int  *k;
  k=&x;
  printf("%u", k);    /* prints address of x…..format specifier is %u */
  printf("%d", *k);  /* prints value of x */
}
```

In the above program:-

| variable | value | address |
|---|---|---|
| X (normal variable) | 20 | 65524 (assume) |
| K (pointer variable) | 65524 | 65538 (assume) |

**Operators used with pointers:**

- Address operator ( & )
- Indirection operator ( * )

**Address operator** gives the address of variable and **Indirection operator** gives the value of the variable that the pointer is pointing to. **( refer sample program-1)**. Indirection operator is also called as dereferencing operator.

**printf("%d", *k);** printing the value of x indirectly by using the address of x is known as **POINTER DEREFERENCING**.

**ADVANTAGES OF POINTERS:**

- Increases the execution speed of a program.
- Variables defined outside the function can be accessed using pointers.
- We can return more than one value to calling function by using pointers.

**OPERATIONS ON POINTERS**

A limited set of arithmetic operations can be performed on pointers:

- A pointer can be incremented or decremented.

- An integer can be added to pointer.
- An integer can be subtracted from a pointer.
- One pointer can be subtracted from another pointer.

Since, pointers stores the address of another variable whatever the changes we make to pointers will affect the address that is stored in pointer variable.

Ex: In case of a integer variable, if increment / decrement operation is performed on pointer variable, then address in that pointer is incremented by 2 bytes since, length of one integer variable is 2 bytes.

**Sample program-2**

```
int main()
{
        int  x=20;
        int  *k;
        k=&x;   /* let address of x =65524  and value of k=65524*/
        printf("%u",k);
        k++;
        printf("%u",k);  /* now k=65526 */
}
```

**Sample program-3**

```
int main()
{
        int  x=20;
        int  *k1,*k2,*k3;
        k1=&x;   /* let address of x =65524  and value of k1=65524*/
        printf("%u",k1);
        k2=k1+2;
```

```
        printf("%u",k);  /* now k2=65528 */
        k3=k1-4;
        printf("%u",k);  /* now k=65516 */
}
```

The expression **k2=k1+2;** results in the incrementation of address in k1 by 4 bytes. This is done as follows:-

**Length of datatype * integer** that is, length of pointer variable k2 is integer (length of integer = 2 bytes) and integer is 2.

Similarly,

The expression **k3=k1-4;** results in the decrementation of address in k1 by 8 bytes. This is done as follows:-

2*4=8 bytes

**Sample program-4** (subtraction of pointers)

```
int main()
{
  int  a[5]={ 10,20,30,40,50 };
  int  *p,*q;
  p=&a[1];
  q=&a[3];
  printf("%d",*q-*p);
  printf("%u",q-p);
}
```

Explanation:

| a[0] | a[1] | a[2] | a[3] | a[4] |
|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 |
| 1000 | 1002 | 1004 | 1006 | 1008 |

From the above program :-

P=1002 and q=1006

*q-*p = 40-20=20

q-p= 2 but not 4

because, pointer subtraction results in the number of bytes separating he corresponding elements of array **or** pointer subtraction is the result of subtraction of index elements of array i.e., a[3]-a[1]=3-1=2

**operations that are not possible on pointers:-**

- Addition of 2 pointers
- Multiplication and division operations cannot be performed on pointers.

Because addition of 2 pointers means, adding 2 addresses but the resultant address may no exist similarly for multiplication and division also the resultant address may not exist in memory.

**POINTERS AND FUNCTION ARGUMENTS**

**Call-by-value:**

```
int swap(int a,int b);

int main()

{

  int x=10,y=20;

  swap(x,y);

  printf("x=%d   y=%d",  x, y);  /* x=10   y=20 */

}

int swap(int a, int b)

{

   int  t;
```

```
  t=a;
  a=b;
  b=t;
printf("a=%d    b=%d",a,b);    /* a=20   b=10 */
}
```

In the above program we are passing a copy of the actual arguments to called function. So whatever the changes we make to the formal arguments those changes will not apply to actual arguments.

**Call-by-reference:**

```
int swap(int *a,int *b);
int main()
{
 int x=10,y=20;
 swap(&x,&y);
 printf("x=%d   y=%d",  x, y);  /* x=20   y=10 */
}
int swap(int *a, int *b)
{
  int  t;
   t=*a;
  *a=*b;
  *b=t;
printf("a=%d    b=%d",a,b);    /* a=20   b=10 */
}
```

In the above program we are the addresses of the actual arguments to called function. So whatever the changes we make to the formal arguments those changes will apply to actual arguments because both a and x refer same memory location and ; both b and y refer same memory location.

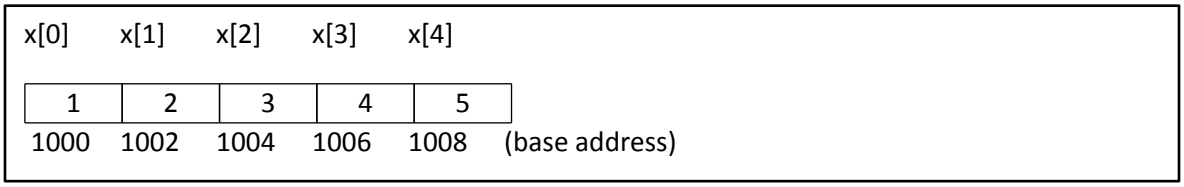
## RELATIONSHIP BETWEEN POINTERS AND ARRAYS:-

An array is a collection of elements, which belongs to similar data type. When an array is declared, the compiler allocates memory, depending upon the length of data type of array and its size. The array is allocated with base address and from there, all elements of array are stored in continuous locations.

The base address is the location of first element of the array, i.e, element at '0'index. Along with this, the compiler defines array name as CONSTANT POINTER to the first element.

Ex:- int x[5]={ 1,2,3,4,5};

Let base address of x is 1000. Therefore elements are stored as: -




The array name 'x' is defined as constant pointer, pointing to the first element x[0].

X = &x[0] = 1000.

If we declare 'p1' as pointer, then we can make 'p1' as pointer to the array 'x[5]' by the following assignment expression:-

| | | |
|---|---|---|
| Int *p1; | (or) | int *p1; |
| P1= &x[0]; | | p1=x; |

Now, every value of array can be accessed by using (++) increment operator along with pointer 'p1'. i.e,

| | |
|---|---|
| P1 = &x[0] = 1000 | *p1 = x[0] = 1 |
| P1+1 = &x[1] = 1002 | *(p1+1) = x[1] = 2 |

P1+2 = &x[2] = 1004 *(p1+2)= x[2] = 3

P1+3 = &x[3] = 1006 *(p1+3) = x[3] = 4

P1+4 = &x[4] = 1008 *(p1+4)= x[4] = 5

The array name itself can be treated as pointer and used in pointer arithmetic i.e., arithmetic operations.

Ex: *(p1+3) is same as *(x+3) both refers element of location x[3];

Pointers can be subscripted same as arrays, i.e., *(p1+3) = *(x+3) = p1[3] refers to x[3].

An array name is always a constant pointer; it always points to the beginning of the array.

The expression X+ = 3 alid because, it attempts to modify the value of array name, with pointer arithmetic.

**Sample program-5**

```
int main()

{

int x[5]={1,2,3,4,5}, *p1,i;

p1=x;

printf("elements of array are");

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

{

printf("x[%d] = %d \n",i,x[i]);

}

printf("pointer subscript notation");

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

{

printf("p1[%d] = %d \n",i,p1[i]);  }  }
```

## ARRAY OF POINTERS

- Array is a collection of elements of similar data type
- Pointer is a variable, which stores, the address of another variable.
- Therefore, Array of pointers means, an array which contains address of variables.
- The address present, in the array of pointers, can be addresses of variables (or) array elements.

**Syntax for declaring array of pointers is:**

Datatype *array_name[size];

Ex: int *k[5]; indicates that array contains addresses of integer variables.

```
int main()
{
int *k[2];
int a=20,b=30;
k[0]=&a;
k[1]=&b;
for(i=0;i<2;i++)
{
printf("%d",*(k[i]));
}
}
```

output: 20 30

## POINTER TO A POINTER:

Like an ordinary variable, a pointer variable also has an address. We cannot store the address of a pointer variable, using ordinary point variable. To store the address of a pointer variable, we need to use **pointer to a pointer**,i.e., A pointer variable, which can store the address of another pointer variable.

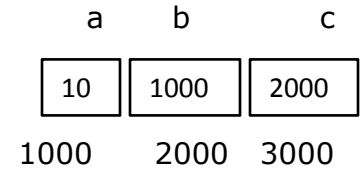
**Syntax**: datatype **pointer_name;

Ex: int =**k;

This declaration creates a pointer to a pointer ‘k’ , which stores the address of another integer pointer.

//**program to demonstrate pointer to pointer:**

```
int main()
{
int a=10, *b,**c;
b=&a;c=&b;
printf("address of variable a= %u",b);
printf("address of variable b= %u",c);
}
```




```
int main()
{
int a=10, *b,**c;
b=&a;c=&b;
printf("%u" , b );   →   1000  (address of a)
printf("%u" , *c );  →   1000  (value at address stored in c)
printf("%u" , &b );  →   2000  (address of b)
printf("%u" , &c );  →   3000  (address of c)
printf("%u" , *(*a) );   →   10   (value stored at address of a)
printf("%u" , *b );  →   10   (value stored at address of b)
printf("%u" , **c );     →   10   (*(*(&b)))
}
```

**(*(*(&b)))** → **(&b) –**address of ‘b’ = 2000

**(*(&b)) –** value stored at address of ‘b’ = 1000

**(*(*(&b))) - *(1000) = *(&a) = *(b) = 10**

It is always better to use normal pointer, because, when ever compiler executes the expression, a [10], it internally, converts it into *(a+10). In case of normal pointers, conversion does not take place, therefore program runs faster.