# POINTERS IN C

Rahul Deshwal
*Department of Computer Science and Engineering,*
*Dronacharya College of Engineering, Gurgaon*

**Abstract- This paper consists of all the basic information regarding usage of pointers in C language; its usage and implementation.**

## I. INTRODUCTION

**Pointers**

Pointer are a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers.

C uses ***pointers*** a lot.

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

C uses pointers explicitly with:

- Arrays,
- Structures,
- Functions.

**NOTE:** Pointers are perhaps the most difficult part of C to understand. C's implementation is slightly different DIFFERENT from other languages.

**What is a Pointer?**

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The ***unary*** or ***monadic*** operator **&** gives the ``address of a variable''.

The ***indirection*** or dereference operator ***** gives the ``contents of an object ***pointed to*** by a pointer''.

To declare a pointer to a variable do:

```
int *pointer;
```

Consider the effect of the following code:

```
int x = 1, y = 2;
int *ip;

ip = &x;
y = *ip;


x = ip;
*ip = 3;
```

It is worth considering what is going on at the ***machine level*** in memory to fully understand how pointer work. Consider Fig. 9.1. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000. **Note** A pointer is a variable and thus its values need to be stored somewhere. It is the nature of the pointers value that is ***new***.

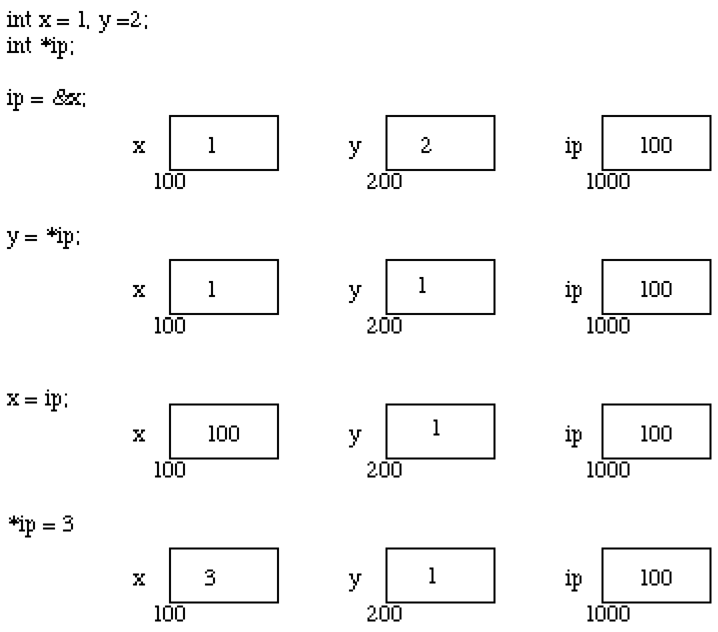

**Fig. 9.1 Pointer, Variables and Memory** Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a ***pointer to an integer*** and is assigned to the address of x(&x). So ip gets loaded with the value 100.

Next y gets assigned to the ***contents of*** ip. In this example ip currently ***points*** to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1.

We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly **legal** (although not all that common) to assign the current value of ip to x. The value of ip at this instant is 100.

Finally we can assign a value to the contents of a pointer (*ip).

## II. POINTER AND FUNCTIONS

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.
When C passes arguments to functions it passes them by value.
There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. Other languages do this (***e.g.*** var parameters in PASCAL). C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.
The best way to study this is to look at an example where we must be able to receive changed parameters.
Let us try and write a function to swap variables around?
The usual function ***call***:

```
   swap(a, b)   WON'T WORK.
```

Pointers provide the solution: ***Pass the address of the variables to the functions and access address of function.***
Thus our function call in our program would look like this:

```
   swap(&a, &b)
```

The Code to swap is fairly straightforward:

```
   void swap(int *px, int *py)

                                   { int temp;

                                 temp = *px;
                                 /* contents of pointer */

                                 *px = *py;
                                 *py = temp;
                                 }
```

We can return pointer from functions. A common example is when passing back structures. ***e.g.***:

```
typedef struct {float x,y,z;} COORD;

                                 main()

                                  {  COORD p1,
*coord_fn();
                                /* declare fn to return ptr of
                                 COORD type */

                                 ....
                                 p1 = *coord_fn(...);
                                 /* assign contents of
address returned */
                                 ....
                                 }

  COORD *coord_fn(...)

                                  {  COORD p;

                                 .....
                                 p = ....;
                                 /* assign structure values */

                                 return &p;
                                 /* return address of p */
                                 }
```

Here we return a pointer whose contents are immediately ***unwrapped*** into a variable. We must do this straight away as the variable we pointed to was local to a function that has now finished. This means that the address space is free and can be overwritten. It will not have been overwritten straight after the function ha squit though so this is perfectly safe.

## III. POINTERS AND ARRAYS

Pointers and arrays are very closely linked in C.
Hint: think of array elements arranged in consecutive memory locations.
Consider the following:

```
  int a[10], x;
                                 int *pa;

                                 pa = &a[0];  /* pa pointer to
address of a[0] */

                                 x = *pa;
                                 /* x = contents of pa (a[0]
in this case) */
```

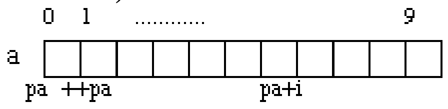

**Fig. 9.3 Arrays and Pointers**
To get somewhere in the array (Fig. 9.3) using a pointer we could do:

```
  pa + i ≡ a[i]
```

**Arrays of Pointers**
We can have arrays of pointers since pointers are variables.
Example use:
***Sort lines of text of different length***.

***Arrays of Pointers*** are a data representation that will cope efficiently and conveniently with variable length text lines.
How can we do this?:

- Store lines end-to-end in one big char array (Fig. 9.4). \n will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using strcmp() standard library function.
- If 2 lines are out of order -- swap pointer in pointer array (not text).

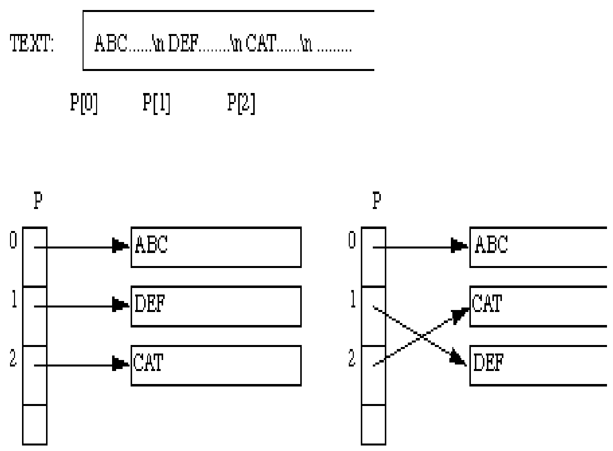

**Fig. 9.4 Arrays of Pointers (String Sorting Example)**
This eliminates:

- complicated storage management.
- high overheads of moving lines.

**Multidimensional arrays and pointers**
We should think of multidimensional arrays in a different way in C:
***A 2D array is really a 1D array, each of whose elements is itself an array***
Hence
a[n][m] notation.
Array elements are stored row by row.
When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant.
The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.
Considerint a[5][35] to be passed in a function:
We can do:

```
f(int a[][35]) {.....}
```

or even:

```
f(int (*a)[35]) {.....}
```

We need parenthesis (*a) since [] have a higher precedence than *
So:
int (*a)[35]; declares a pointer to an array of 35 ints.
int *a[35]; declares an array of 35 pointers to ints.
Now lets look at the (subtle) difference between pointers and arrays. Strings are a common application of this.
Consider:

```
char *name[10];
char Aname[10][20];
```

We can legally do name[3][4] and Aname[3][4] in C.
However

- Aname is a true 200 element 2D char array.
- access elements via
  ***20*row + col + base_address***
  in memory.
- name has 10 pointer elements.

REFERENCE

Let us c – yashvant p.kanetkar