# POINTERS IN C

# PART I

Dear Friends,

**Module – 1**

## Introduction

Pointers play an important role in the C language and it seems to be a new concept for the beginners. Pointers are simple to use, provided the basic concept is understood properly. Programs can be written efficiently and compactly with the help of pointers but its careless use courses unexpected errors and difficulties in the program execution. We have divided the topic in to two sessions.

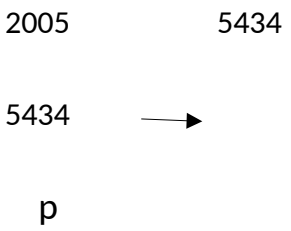
Before coming to pointers let us see the data representation in computer memory. The smallest piece of data in a computer memory is known as a bit, which represents 0 or 1. Bits are grouped in to bytes. Each byte in the computer memory is called a memory location or memory cell. Computer memory consists of series of a series of consecutive memory locations. The number attached to a memory location is called the memory address of that location.

The number of bytes required for storing the value of each data type is system dependent. When more than one memory locations are used for holding a value, the starting address of that storage is considered as the address of that value. A data type is associated with a memory location referred to by a variable in a C program. Based on the data type, the compiler interprets the content of the memory location allotted for the variable. Generally one byte of memory is used for character type data storage , 2 bytes for integer type data storage, 4 bytes for float data storage and 8 bytes for double type data storage.

**What is a pointer?**

A pointer value or pointer is a data object that refers to a memory location, which is an address. It resides in the memory location, which is represented by the pointer variable. Thus a pointer variable may contain the address of another variable or any valid address in the computer memory. That's why a pointer is also called an address variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code.

Observe the content of the pointer variable p in the memory location in the figure shown in screen. It holds a valid address 5434 from the memory where a data object may be stored.




Representation of a pointer variable

**Module – 2**

## Understanding pointers

Consider the declaration,

int i = 5;

This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i** with this memory location.

(c) Store the value 5 at this location.

We can understand more through the following program:

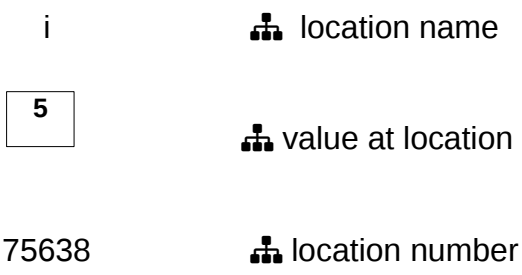
```
main( )
{
        int  i = 5 ;
        printf ( "\nValue of i = %d", i ) ;
        printf ( "\nAddress of i = %u", &i ) ;
}
```

The output of the above program would be:

Value of i = 5

Address of i = 75638

We may represent **i**'s location in memory by the following memory map.




We can see that the computer has selected memory location 75638 as the place to store the value 5. The location number 75638 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.

In the above printf( ) statement. '&' used in this statement is '**address of**' operator. The expression **&i** returns the address of the variable **i**, which in this case it is 75638. Since 75638 represents an address, there is no question of a sign being associated with it.

Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer. We have been using the '&' operator all the time in the scanf( ) statement.
The other pointer operator available in C is '*****', called '**value at address**' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

The indirection operator * is used to access the value pointed by its operand. The operand must be a pointer variable or a pointer expression. If p is a pointer variable, then *p is and expression refereeing to a data object. The actual address in which the data object is stored is represented by a pointer variable. The stored data item is accessed indirectly using the expression *p. Actually the name indirection is taken from low level programming to refer to the indirect addressing mode. The direct value of the pointer variable, say p, is an address, whereas *p is the indirect value of p.

Consider the following statements for the illustration of the expressions using * and values.

```
int num = 15, *p;
p=&num
```

From the example it is clear that *&num is equivalent to num and also *p is equivalent to num as the address of num is assigned to p. The three expressions num, *&num and *p represent the same storage where the value 15 is stored. It is not necessary to enclose &num within parenthesis in *&num, since unary operators associate from right to left. It is also possible to have expressions like &*&num, &*&*&num, *&*&*&num etc.

Look at the output of the following program:

```
main( )
{
```

```
    int i = 5 ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

Address of i =75638

Value of i = 3

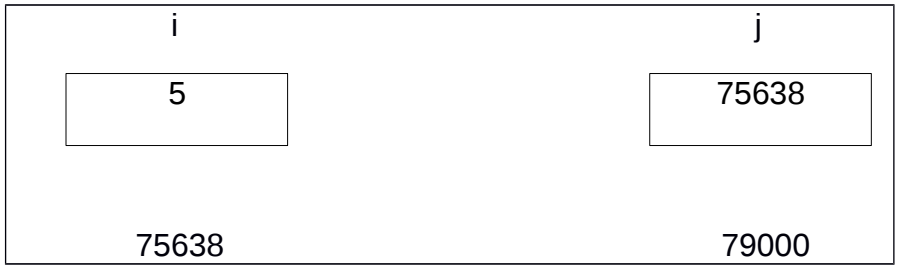
Value of i = 3

Note that printing the value of ***( &i )*** is same as printing the value of **i**.

The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying,

j = &i ;

But remember that **j** is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (**i** in this case). Since **j** is a variable the compiler must provide it space in the memory. Once again, the following memory map would illustrate the contents of **i** and **j**.




As you can see, **i**'s value is 5 and **j**'s value is **i**'s address. So we need to declare j as a pointer variable so as to use in the program. The general format for declaration of pointer variable is

data_type * var1, *var2,……*varn;

Where data_type refers to the data type of the value stored in the address given by the pointer variable and var1,var2,…..varn are pointer variables.

For example in the previous example, we can't use **j** in a program without declaring it. And since **j** is a variable that contains the address of **i**, it is declared as,

int *j ;

This declaration tells the compiler that **j** will be used to store the address of an integer value. In other words **j** points to an integer. How do we justify the usage of ***** in the declaration,

int *j ;

Let us go by the meaning of *****. It stands for 'value at address'. Thus, **int *j** would mean, the value at the address contained in **j** is an **int**.
Here is a program that demonstrates the relationships we have been discussing.

```
#include<stdio.h>
main( )
{
        int i = 5 ;
        int *j ;
        j = &i ;
        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nAddress of i = %u", j ) ;
        printf ( "\nAddress of j = %u", &j ) ;
        printf ( "\nValue of j = %u", j ) ;
        printf ( "\nValue of i = %d", i ) ;
```

```
        printf ( "\nValue of i = %d", *( &i ) ) ;
        printf ( "\nValue of i = %d", *j ) ;
}
```

The output of the above program would be:

```
Address of i = 75638
Address of i = 75638
Address of j = 79000
Value of j = 75638
Value of i = 5
Value of i = 5
Value of i = 5
```

Work through the above program carefully, Everything we say about C pointers from here onwards will depend on your understanding these expressions thoroughly.

We will now see some valid pointer variable declarations

```
int     *ptr;
```

Here ptr is pointer variable to store the address of an integer value.

```
float *fraction, *epsilon;
```

Here fraction and epsilon are pointer variables which points to float values

```
short   *sptr;
```

Here sptr is a pointer to a short integer.

```
char    *cptr;
```

Here cptr is a pointer to a character and

```
long    *lptr;
```

Here lptr is a pointer to a long integer.

It is also possible to combine the pointer variables and other ordinary variables of the same data type in a single declaration as shown in the screen.

int a, *ap;
Here a is an integer variable and ap is a pointer variable which points to a integer value.

float *xp, x, y;
Here xp is pointer variable which points to a float value, x and y are simply float variables.

char c1,c2,*c1p,*c2p;
Here c1, c2 are character variables and c1p and c2p are pointer variables which points to char values.

User defined pointer variables can be created with typedef statements. The following example show how a typedef statement can be used to declare a pointer variable num pointing to int data type (int *num)

typedef int NUMBER; OR typedef int * NUMBER;
NUMBER *num; NUMBER num;

**Module – 3**

## Incrementation and decrementation of pointers

It is possible to increment and decrement pointer variables using ++ and - - operators respectively. A pointer expression may be preceded by and or followed by ++ or - -. In

such cases, the operators &, * and ++ and – are involved. While evaluating these expressions, it is very important to remember the operator precedence and associativity of the operations. The operators &, *, ++ and -- have same precedence level and the evaluation is carried out from right to left. Consider the statements,

```
int *p, x=20;
p=&x;
++ *p;
*p++;
```

According to the first two statements, the address of the variable x is assigned to the pointer variable p. Since x=20, the value of *p is also 20. The statement ++*p; has two operators ++ and *. Since both the operators have the same precedence level, associativity is applied resulting in right to left evaluation. Hence the evaluation increments *p by 1 ( that is 20+1=21).

The statement *p++; is also evaluated from right to left. Since a post increment operator is used, the effect of the increment is available only after the execution of *p++;. Hence for the evaluation of *p, the value of p before increment is used yielding the output 21. After executing the statement *p++; the value of p is p+1 (p is incremented by 2bytes as 2bytes are assumed to store an int value). The comiler takes care of the scaling factor based on the data type it points to while incrementing the pointer. If a pointer points to float, then incrementing it gives an offset of 4 bytes.

The expression,

++*p*2 increments what p points to by 1 and the incremented value is multiplied by 2.

Now let us summarize the permissible and not permissible pointer operations.

**Pointer operations – permissible**

- A pointer can have the address of an ordinary variable and its content(ptr=&v).
- The content of a pointer can be copied to another pointer variable provided both pointers are of same type.(ptr=ptr1)
- A pointer variable can be assigned a null(zero) value. (e.g ptr=NULL, where

NULL is a symbolic constant that represents the value 0).

- An integer quantity can be added to or subtracted from a pointer variable. (e.g ptr+1, ++ptr, ptr-3, ptr- - etc).

**Pointer operations – not permissible**

- Two pointer variables cannot be added.
- Pointer variable cannot be multiplied by a constant.
- Two pointer variables cannot be divided or multiplied.

**Module – 4**

## Use of pointers in arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

We know that the declaration

int a[10];

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9] as shown in the figure on the screen.

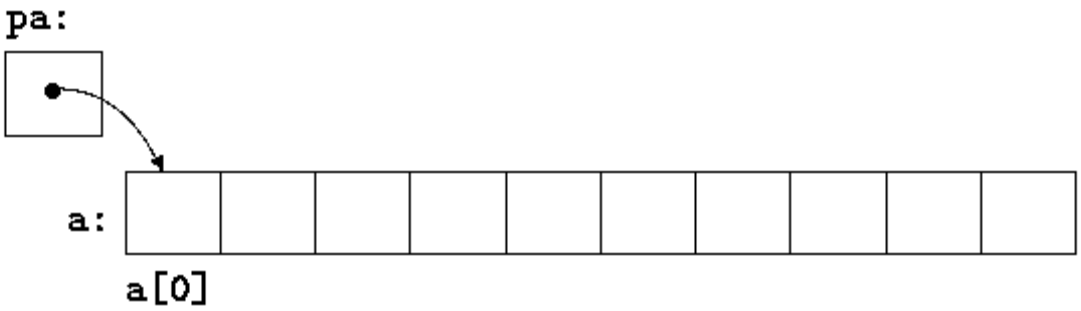
The notation **a[i]** refers to the **i-th** element of the array. If **pa** is a pointer to an integer, declared as

int *pa;

then the assignment

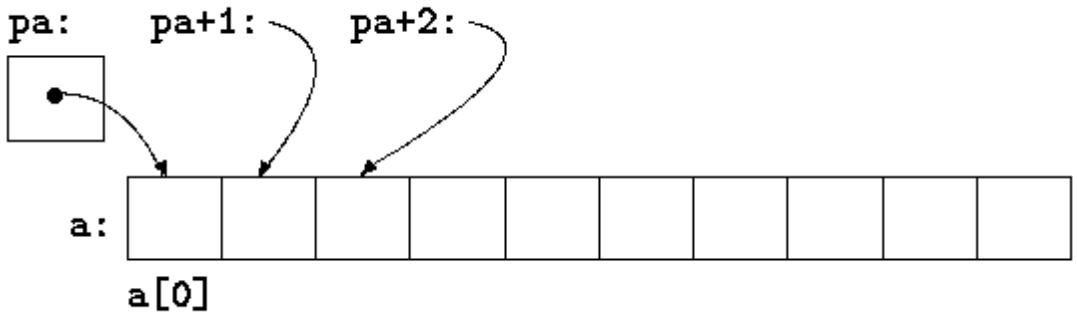
pa = &a[0];

Sets **pa** to point to element zero of **a**; that is, **pa** contains the address of a[0] as shown in the screen.




Now the assignment
x = *pa;
will copy the contents of a[0] into x.
If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. Thus, if pa points to a[0],

*(pa+1)
refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].




These remarks are true regardless of the type or size of the variables in the array a. The meaning of ``adding 1 to a pointer,'' and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

pa = &a[0];

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment pa=&a[0] can also be written as

pa = a;

Rather more surprising, at first sight, is the fact that a reference to a[i] can also be written as *(a+i). In evaluating a[i], C converts it to *(a+i) immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. As the other side of this coin, if pa is a pointer, expressions might use it with a subscript; pa[i] is identical to *(pa+i). In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; constructions like a=pa and a++ are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write a program which computes the length of a string.

```
/* strlen: return length of string s */
int strlen(char *s)
{
        int n;
        for (n = 0; *s != '\0', s++)
                n++;
        return n;
}
```

## Summary

Let us now summarize the points that we discussed in the session.

1. C uses bytes as the basic units of memory. The bytes requires to store a value depend on the data type of the object used. Each byte is numbered by an address for its reference. It is possible to directly access the addresses in C by using pointers.
2. A pointer is a valid address, which is stored in a pointer variable.
3. The pointer type and the memory that it refers should be same. That is to hold

the address of type integer memory we have to create an integer pointer.

4. There are two unary operations that are exclusively used in connection with pointers. They are
    a. Address of operator , denoted by & (ampersand) and
    b. Indirection or dereferencing operator, denoted by * (asterisk)

   Each of the above operators must precede its operand. Pointer expressions are formed by using these operators & and * with their respective operands.

5. The address of operator & returns the address of its operand. The operand must be a named region of storage (like integer variable, array variable, pointer variable etc.) for which a value may be assigned. It cannot be a constant or expression or register type variable.

6. We can use the pointer to retrieve data from memory or to store data into the memory to which it points to. Both operations are classified as pointer dereferencing. It is also called as indirection operator '*'.

```
int num = 5, new, *ptr;
ptr = &num;                    /* ptr holds the address of variable num*/
new = *ptr;                    /* copies the content of num to new using
                                  pointer */
```

7. After assigning proper values to the pointer variables, it is possible to increment or decrement the pointer variables using ++ and – operators respectively.
8. The array variable represented by a subscript expression may also be written using a pointer expression. The representation of the pointer expression helps in faster and easier execution.
9. Character arrays may also be manipulated using pointers. String constants return a pointer that can be used in pointer expressios.

Please try to completely solve this **assignment.**

1. What are pointers a? Are they integers? Explain with an example.

2. How is a pointer variable declared and what are the operators exclusively used with pointers?
3. Is the name of an array a pointer? Why is only the address of the first element of an array necessary in order to access the whole array?
4. How are single and two dimensional arrays represented using pointers? Explain.
5. Write a program using pointers to read in an array of integers and print the elements in reverse order.
6. Write a program to concatenate two strings using pointers.

Here is some books for your **Reference**


1. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. Yashavant Kanetkar; Let us C, BPB Publications, New Delhi.
3. Greg Perry, Absolute Beginners' guide to C, SAMS publishing, April 1994.


Hope you start enjoying the lessons of C and interested in writing C programs. There are many more other features of pointers in C are remaining to explore. So let us wait for the coming sessions and till then bye._