Pointers in C

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer .

some of the valid pointer declarations –

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

C pointer Address operators

There are two important operators which are highly required, if you are working with the pointers. Without these operators, we cannot work with the pointers.

The operators are:

The * Operator (Dereference Operator or Indirection Operator)
The & Operator (Address Of Operator)

1) The * Operator (Dereference Operator or Indirection Operator)
"Dereference Operator" or "Indirection Operator denoted by asterisk character (*), * is a unary operator which performs two operations with the pointer (which is used for two purposes with the pointers).

To declare a pointer
To access the stored value of the memory (location) pointed by the pointer

A) To declare a pointer Consider the syntax

data_type *pointer_variable_name;
Let suppose, if we declare a pointer ptrX to store the memory address of an integer variable; then the pointer declaration will be int *ptrX;

B) To access the stored value of the memory (location) pointed by the pointer
Consider the syntax

*pointer_variable_name;
Let suppose, if there is a pointer variable ptrX which is pointing to the address of an integer variable x; then to access the value of x, *ptrX will be used.

2) The & Operator (Address Of Operator)

The "Address Of" Operator denoted by the ampersand character (&), & is a unary operator, which returns the address of a variable.

After declaration of a pointer variable, we need to initialize the pointer with the valid memory address; to get the memory address of a variable Address Of" (&) Operator is used.

How to Use Pointers?

(a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using **unary operator *** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include <stdio.h>

int main ()
 {

   int  var = 20;    /* actual variable declaration */
   int  *ip;         /* pointer variable declaration */

   ip = &var;        /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

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

```
Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of *ip variable: 20
```

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. 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>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

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

```
The value of ptr is 0
```

An array of pointers

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer –

```
int *ptr[MAX];
```

It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows –

```
#include <stdio.h>

const int MAX = 3;

int main ()
 {

   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];

   for ( i = 0; i < MAX; i++)
 {
     ptr[i] = &var[i]; /* assign the address of integer. */
   }

   for ( i = 0; i < MAX; i++) {
     printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }

   return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows −

```
#include <stdio.h>

const int MAX = 4;

int main ()
 {

  char *names[] = {
    "ram",
    "hari",
    "krishna",
    "murli"
  };

  int i = 0;

  for ( i = 0; i < MAX; i++) {
    printf("Value of names[%d] = %s\n", i, names[i] );
  }

  return 0;
}
```

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

```
Value of names[0] = ram
Value of names[1] = hari
Value of names[2] = krishna
Value of names[3] = murli
```

What is static memory allocation and dynamic memory allocation?

Static Memory Allocation: Memory is allocated for the declared variable by the compiler. The address can be obtained by using 'address of' operator and can be assigned to a pointer. The memory is allocated during compile time. Since most of the declared variables have static memory, this kind of assigning the address of a variable to a pointer is known as static memory allocation.

Dynamic Memory Allocation: Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables.
C Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

C malloc() method
"malloc" or "memory allocation" method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```
For Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

C calloc() method
"calloc" or "contiguous allocation" method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

Syntax:

ptr = (cast-type*)calloc(n, element-size);
For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

C free() method
“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

free(ptr);

C realloc() method
“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.

Syntax:

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.