POINTERS IN C PROGRAMMING

A pointer is a variable in C that points to a memory location. It does not directly contain a value like int or float but just a memory direction. One can reserve some memory on the direction marked by the pointer and use it to store some values on it, later on these values are indirectly accesed through the pointer variable.

A pointer is declared as follows :

```
    int *ptr1;
```

And may be used as in the example to point to a variable of the same type as the pointer:

```
    int i;
    ptr1 = &i;
```

Here you have to understand the role played by characters '*' and '&' in front of variables. The character '*' in front of the variable is used to declare a pointer variable. The character '*' is also used in front of a pointer variable to access and retrieve the value contained by a pointer variable, not the address (i.e. the address is designated by the name itself without the '*'). The character '&' is used to access the memory address of any variable (note that every variable is stored somewhere in memory). In the above example, you are declaring an integer pointer "ptr1" and an integer variable "i". Then you are forcing the pointer ptr1 to point to the address of the variable "i". Now both "*ptr1" and "i" refer to the same memory address. Consider the example below to understand the concept.

Program 10.1

```
#include <stdio.h>
main()
{
  int *ptr1;
  int i;
  ptr1 = &i;
  i = 3;
  printf("The value of i is %d\n", i);
  printf("The pointer ptr1 contains the value %d\n",*ptr1);
}
```

The above program yields the result

```
The value of i is 3
The pointer ptr1 contains the value 3
```

This illustrates the fact that the pointer ptr1 is pointing to the address of the variable i. Hence '&' is referred to as the *address of* operator.

Functions using Pointers

You know that a function can return only one value. If you want a function to alter more than one variable, you can pass pointers to the function to achieve this. The function will not alter the pointer but the contents of the pointer.

Consider the following example where you are passing two integer pointers iptr1 and iptr2 to a function named *swap*.

Program 10.2

```
#include <stdio.h>
void swap(int *iptr1, int *iptr2)
{
  int temp;
  temp = *iptr2;
  *iptr2 = *iptr1;
  *iptr1 = temp;
}
main()
{
  int x = 10;
  int y = 20;
  swap(&x,&y);
  printf("Value of iptr1 is %d", *iptr1);
  printf("Value of iptr2 is %d", *iptr2);
}
```

Note: Whenever you are calling a function which expects a pointer as one of its arguments, you have to pass the address of a variable to it.

The built in function *scanf()* expects the second argument to be a pointer to a variable. Consider the statement

```
scanf("%f",&Total);
```

Here you are passing the address of the variable *Total*, if you just pass the variable *Total*, then a core dump occurs when you try to execute this program. Basically this is a memory fault on a UNIX system.

```
scanf("%f",Total); // Results in core dump error !
```

Pointers and Arrays

In C, pointers and arrays are closely related. Pointers are very useful in array manipulation. Declaration of a pointer to an array is just like an integer pointer or a float pointer.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

Here the pointer contains the address of the first element of the array. C provides another alternate way for pointing to the first element of an array, ptr = arr;

Note: It is important to remember that the name of an array not followed by a subscript, is a pointer to the first element in the array.

Once a pointer has been set to point to an element of an array, it is possible to use the increment ++ and decrement -- operator to point to subsequent or previous elements of the array, respectively. But incrementing or decrementing the pointer to access a memory location beyond an array's bound produces a runtime error, and your program may crash or may overwrite other data or code sections of your program. So it is the responsibility of the programmer to use the pointer to an array wisely.

One more factor to consider is the size of the data type that the pointer points to. Suppose an integer pointer is pointing to an integer array; when you increment the pointer, the pointer points to the next element of the array. But in reality the pointer will contain an address which is typically four bytes greater than the address of the first element of the array.

Array Transformation rule

If *arr1* is an array, then the expression *arr1 + 1* is the address of the second element of the array, regardless of arr1's data type. We can now use the indirection operator * in front of the variable to retrieve the value stored at this location. Thus

```
    *(arr1 + 1)
```

gives the value stored in the array's second element. Parentheses are required because the indirection operator * has a higher precedence over the addition operator. So we can use this transformation rule to convert any array reference to its equivalent pointer expression.

arr1[0] is equivalent to *(arr1 + 0)

arr1[1] is equivalent to *(arr1 + 1)

arr1[2] is equivalent to *(arr1 + 2)

Without the parentheses the expression *arr1 + 1 evaluates to something totally different, as this retrieves the value stored in the array's first element and adds 1 to it.

Memory Allocation and pointers

Array definition results in a block of memory being reserved by the operating system at the begining of program execution, this does not happen if an array is represented by a pointer variable. The use of a pointer variable requires a programmer to do some type of memory assignment before the array elements are utilized. This is called *Dynamic memory allocation* . The *malloc()* standard C library function is used for this purpose.

Program 10.3

```
#include <stdio.h>
main()
{
  int *x;
  float *y;
  x =(int *) malloc(10 * sizeof(int));
  /* reserves memory for 10 integers */
  y = (double *) malloc(10 * sizeof(double));
```

```
   /* reserves memory for 10 floating point numbers */
 }
```

Note: The type cast preceding the *malloc()* function must be same as the data type of the pointer variable.

Difference between arrays and pointers

There are some key differences between arrays and pointers. A pointer's value can be changed to point to some other memory location. But the pointer represented by an array name cannot be changed. It is treated as a constant.

```
float TotAmt[10];


TotAmt  ++; // illegal statement
TotAmt -= 1; // illegal statement
```

One more point to remember is that array names are initialized to point to the first element in the array whereas pointers are uninitialized when declared. They have to be explicitly initialized before usage otherwise a run time error will occur.

Source : http://www.peoi.org/Courses/Coursesen/cprog/frame10.html