Features of Pointers In C

Subhash khalkho & Sunny kumar
Department of IT, 3rd sem DCE, Gurgaon Haryana
subhash.16938@ggnindia.dronachary.info ; sunny.16941@ggnindia.dronacharya.info

ABSTRACT:

This paper discusses about the most powerful and yet most dangerous and effective tool in C language i.e, pointers. Pointers are an extremely powerful programming tool in C, they can make programming much easier to execute and design and help improve your program's efficiency and even allow you to handle unlimited stock of data, using a pointers is the one way to have a function to modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle nearly unlimited stock of data. This research includes all the meritorious features of pointers along with the common errors made while using a pointers.

INTRODUCTION:

A pointer is a kind of variable that holds up memory address that is usually a location of another variable present in computer memory. The pointer is one of the strongest and most significant tools of C programming languages, before using a pointer the programmer must know how to use and implement the pointer in order to make a perfect and successful program. There are certain reasons why pointers are considered to be the most effective tool in C:

i. The pointer provides a way by which the memory location of a given variable can be directly accessed and after that it can be manipulated.

ii. The pointers support the feature of dynamic allocation of memory i.e, allocation of memory during run time.

iii. The pointers help in improving the efficiency of a program.

REFERENCE OPERATOR (&):

Whenever a variable is declared in a program a storage location in the internal memory is made available by the compiler. Now the address of the variable can be obtained by '&', an address operator. This operator when applied to a given variable gives the present memory address of that variable

If var is a variable then, &var is the address in memory.

Example to demonstrate the use of reference operator in C programming

```
int main( )
{
  int var=5;
  printf("Value: %d\n",var);
  printf("Address:   %d",&var);      //Note, the ampersand(&) before var.
  return 0;
}
```

Output

Value: 5 ; Address: 2686778

Note: You may obtain different value of address while using this code. In above source code, value 5 is stored in the memory location 2686778. var is just the name given to that location.

scanf("%d",&var);

INDIRECTION OPERATOR (*) :

If address of a variable is known then this operator provides contents of that variable.

For example:

```
main( )
{       int x=5;
        printf("\n value of x is = %d" , x);
        printf("\n Address of x = %u", &x);
        printf("\n Value at address %d = %d, *(&x));
}
```

Output

Values of x = 15 ; Address of x = 2712; Value at address 2712 = 15

POINTER VARIABLES: Pointers variables are the special types of variables that hold memory address rather than data, that is, a variable that holds address value is called a pointer variable or simply a pointer.

DECLARATION OF POINTER

data_type * pointer_variable_name;

int *p;

Above statement defines, *p* as pointer variable of type int.

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it

Example to demonstrate pointers

```
int main( )
{
        int *pc,c;
        c=22;
        printf("Address of c:%d\n",&c);
        printf("Value of c:%d\n\n",c);
        pc=&c;
        printf("Address of pointer pc:%d\n",pc);
        printf("Content of pointer pc:%d\n\n",*pc);
        c=11;
        printf("Address of pointer pc:%d\n",pc);
        printf("Content of pointer pc:%d\n\n",*pc);
        *pc=2;
        printf("Address of c:%d\n",&c);
        printf("Value of c:%d\n\n",c);
        return 0;
}
```

Output:
```
Address of c: 2686784
Value of c: 22
Address of pointer pc: 2686784
Content of pointer pc: 22
Address of pointer pc: 2686784
Content of pointer pc: 11
Address of c: 2686784
Value of c: 2
```

Explanation of program and figure

1. Code int *pc, p; creates a pointer *pc* and a variable *c*. Pointer *pc* points to some address and that address has garbage value. Similarly, variable *c* also has garbage value at this point.
2. Code c=22; makes the value of c equal to 22, i.e.,22 is stored in the memory location of variable *c*.
3. Code pc=&c; makes pointer, point to address of c. Note that, &c is the address of variable *c* (because *c* is normal variable) and *pc* is the address of *pc* (because pc is the pointer variable). Since the address

of pc and address of c is same, *pc (value of pointer pc) will be equal to the value of *c*.

4. Code c=11; makes the value of *c*, 11. Since, pointer *pc* is pointing to address of *c*. Value of **pc* will also be 11.
5. Code *pc=2; change the contents of the memory location pointed by pointer *pc* to change to 2. Since address of pointer *pc* is same as address of *c*, value of *c* also changes to 2.

POINTER AND FUNCTION:

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. C uses pointers explicitly to do this. The best way to study this is to look at an example where we must be able to receive changed parameters.

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( );

              p1 = *coord_fn( );

        }
  COORD *coord_fn( )
  {      COORD p;
          p = ....;
          return &p;
      }
```

POINTERS AND ARRAY:

Pointers and arrays are very closely linked in C. In C the name of array is a pointer that actually has the base address of the array. In real it is a pointer to the first element of array.

Hint: think of array elements arranged in consecutive memory locations.

Consider the following:

```
int a[10], x;
int *pa;
 pa = &a[0];
 x = *pa;
```

Note: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more subtle in its link between arrays and pointers.

For example we can just type

```
   pa = a;
```

instead of

```
   pa = &a[0]
```

and

a[i] can be written as *(a + i).

i.e. &a[i] ≡a + i.

We also express pointer addressing like this:

pa[i] ≡*(pa + i).

However pointers and arrays are different:

- A pointer is a variable. We can do pa = a and pa++.
- An Array is not a variable. a = pa and a++ ARE ILLEGAL.

This stuff is very important. Make sure you understand it. We will see a lot more of this.

We can now understand how arrays are passed to functions.

When an array is passed to a function what is actually passed is its initial elements location in memory.

So:

`strlen(s) ≡strlen(&s[0])`

This is why we declare the function:

`int strlen(char s[ ]);`

An equivalent declaration is : `int strlen(char *s);`

since `char s[ ]` ≡`char *s`.

`strlen()` is a *standard library* function that returns the length of a string. Let's look at how we may write a function:

```
int strlen(char *s)
        { char *p = s;

while (*p != `\0');

    p++;

return p-s;
        }
```

Now let's write a function to copy a string to another string. `strcpy()` is a standard library function that does this.

```
void strcpy(char *s, char *t)
        { while ( (*s++ = *t++) != `\0');}
```

This uses pointers and assignment by value.

Very Neat!!

NOTE: Uses of Null statements with `while`.

ARRAYS OF POINTERS:

Like any other array we can have arrays of pointers since pointers are variables

Declaration of array of pointer:

Data type *variable[ ];

Example of array of pointer:

Char *name[30];

{ram,

raman,

raj,

raju};

The above example declares an array of pointer named as 'name', each element of name points to a string.

The advantage of having array of pointer to string is that we can manipulate the strings conveniently.

For example, the string raman can be copied to to a pointer ptr without using strcpy() function by following statement:

Char *ptr;

Ptr=name[1];

POINTERS AND STRUCTURES:

These are fairly straight forward and are easily defined. Consider the following:

```
struct COORD {float x,y,z;} pt;
        struct COORD *pt_ptr;

pt_ptr = &pt; /* assigns pointer to pt */
```

the arrow operator lets us access a member of the structure pointed to by a pointer.*i.e.*:

```
pt_ptr -> x = 1.0;
pt_ptr -> y = pt_ptr -> y - 3.0;
```

Example to demonstrate use of arrow operator:

```
#include<stdio.h>
Main()
{
        Struct item
                {       char code[5];
                        int qty;
                        float cost;
                       };
Struct item item_rec;
Struct item*ptr;

printf("\n Enter the data for an item");
printf("\n code:");
scanf(" %s", &item_rec.code);
printf("\n qty:");
scanf("%d", &item_rec.qty);
printf("\n cost:");
pcanf("%f", &item_rec.cost);
ptr=&item_rec;
printf("\n The data for the item..");
printf("\n code: %s", ptr -> code);
printf("\n qty : %d", ptr-> qty);
printf("\n cost :5.2f', ptr-> cost);
}
```

CONCLUSION:

The use of pointers is a powerful tool and it is one of the strongest feature of C, but at the same time it is dangerous. In our opinion the difficulty lies not in understanding the subject as much as it does in remembering how to use pointers. If time passes and you don't program in C often, you tend to forget how it works. By the use of our imaginary methods we are better suited at remembering how to use pointers, and we hope it will help you as well. Perhaps you can make up your own way of remembering. In reality all variables are pointers. What makes the difference between when we call it a variable and when we call it a pointer, is what the pointer is actually pointing at. We can point at a memory location, and we then call it a "pointer", or we can point at the value located inside a memory location, and we then call it a variable.

REFFERENCE:

1. Ghiya, R., & Hendren, L. J. (1996, January). Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 1-15). ACM.

2. Reek, K. A. (1997). *Pointers on C.* Addison-Wesley Longman Publishing Co., Inc..

3. Séméria, L., & De Micheli, G. (1998, November). SpC: synthesis of pointers in C application of pointer analysis to the behavioral synthesis from C. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on* (pp. 340-346). IEEE.

4. Colosimo, C., Albanese, A., Hughes, A. J., de Bruin, V. M., & Lees, A. J. (1995). Some specific clinical features differentiate multiple system atrophy (striatonigral variety) from Parkinson's disease. *Archives of neurology*, *52*(3), 294-298.

5. Hughes, A. J., Colosimo, C., Kleedorfer, B., Daniel, S. E., & Lees, A. J. (1992). The dopaminergic response in multiple system atrophy. *Journal of Neurology, Neurosurgery & Psychiatry*, *55*(11), 1009-1013.