International Journal Of Engineering Research & Management Technology
ISSN: 2348-4039
July- 2014 Volume 1, Issue-4
Email: editor@ijermt.org www.ijermt.org

Core Study about Pointers and Memory Allocation in C Language

Riya Jain
Department Of Information Technology
Dronacharya College of Engineering
Gurgaon, Haryana

Divyanshu Chhabra
Department Of Information Technology
Dronacharya College of Engineering
Gurgaon, Haryana

ABSTRACT –
In recent years the I.T and C.S.E sector has bloomed rapidly and globally because of which most people have started inclining towards one or more programming language. This document gives a brief overview of pointers and memory. It explains how do they work and how to use them. Efforts have been put to make others understand the basic concepts of pointers and memory through all the other major programming techniques. For each topic there is a combination of discussion, sample C code, and drawings.

INTRODUCTION –
The paper has been written in order to ensure that the readers get to know the Irresistible power of pointers and how effective they can be while programming. The whole work has been divided into 4 sections. In the first section we will enumerate the basic pointers and a few Operators. The second section is going to be about the memory allocation and deal location. Third Section is going to be about the reference parameters which combine the previous two sections showing how a function can use reference parameters to communicate back to its caller. The last section will be about the heap memory which will deal with topics such as heap allocation deal location, memory leaks etc.

Section I
Basic Pointers

History of Pointers: Harold Lawson is credited with the 1964 invention of the pointer. In 2000, Lawson was presented the Computer Pioneer Award by the IEEE "[f]or inventing the pointer variable and introducing this concept into PL/I, thus providing for the first time, the capability to flexibly treat linked lists in a general-purpose high level language.

POINTER:
A **pointer** is a programming language object whose value refers directly to or points to another value stored somewhere in the computer memory using its address. For high-level programming languages, pointers very effectively take the place of general purpose registers in low-level languages such as assembly language or machine code, but in available memory. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer. A pointer is a simple, more concrete implementation of the more abstract reference data type. Several languages support some type of pointer, although some have more restrictions on their use than others. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number.

The actual data type of the value of all pointers, whether integer, float, character, or anything else, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.
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. In this statement the asterisk is being used to designate a variable as a pointer. Following are some of the valid pointer declarations:

```
int *i; //Pointer to an integer
char *ch; //Pointer to a character
Double *dbl; //Pointer to a double value
```

Dereference Operator: The dereference operator, also called **indirection operator** denoted by "*", is a unary operator found in many languages that include pointer variables. It operates on a pointer variable, and returns a l-value equivalent to the value at the pointer address. This is known as "dereferencing" the pointer. Following code can explain it well:

```
int x=0;
int *ptr=&x;  // & is the dereferencing operator
*ptr=10;      //Value of x=10
```

When the dereference operation is used correctly, it's simple. It just accesses the value of the pointed. The only restriction is that the pointer must have a pointee for the deference to access. Almost all bugs in pointer code involve violating that one restriction. A pointer must be assigned a pointed before dereference operations works.

NULL POINTER: The constant NULL is basically used to "point to nothing". NULL is equal to the integer Constant 0, so NULL can play the role of a Boolean false. It is convenient to have a well defined pointer value which represents the idea that a pointer does not have a pointee. For example:

```
#include<stdio.h>
int main()
{
int *ptr=NULL;
printf("Value of pointer is:");
printf("%d", ptr”);
return 0;
}
```

Output of the above code will be 0.
This is so because the pointer ptr is initialized with NULL. This gives the result to be constant 0.

Shallow and Deep Copying: Sharing can enable communication between two functions. One function passes a pointer to the value to another function both functions can access the value, but the value of interest itself is not copied. This communication is called "shallow". This is so because instead of making and sending a large copy of the value of interest, a small pointer is sent and the value of interest is shared. The recipient needs to understand that they have a shallow copy, so they know not to change or delete it since it is shared. Moreover, the alternative where a complete copy is made and sent is known as a "deep" copy. Deep copies are simpler, since each function can change their copy without interfering with the other copy. But deep copies run slower because of all the copying.

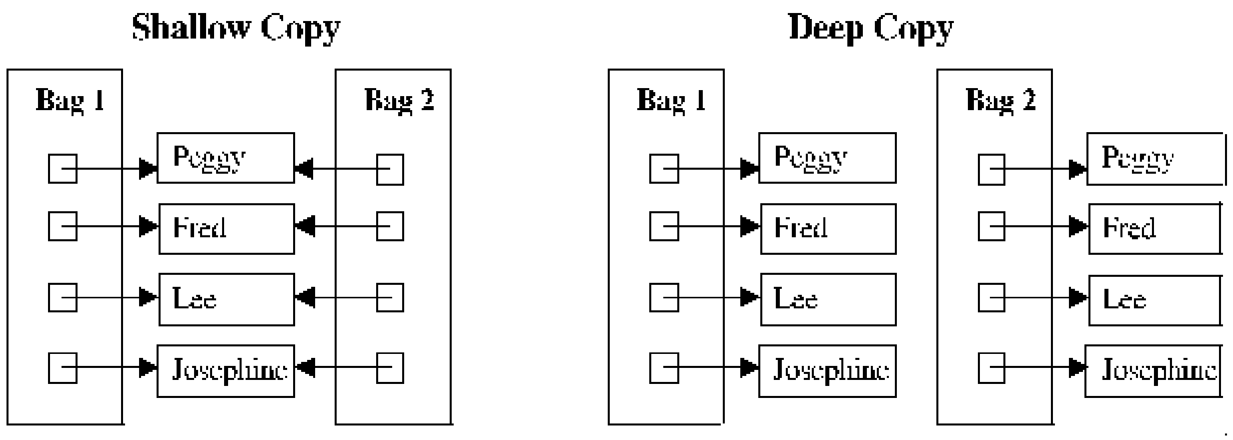

Figure 1

Section II
Memory Allocation and Deal location
Variables represent storage space in the computer's memory. Each variable presents a convenient name like length or sum in the source code. **C dynamic memory allocation** refers to performing manual memory management for dynamic memory allocation in the C programming language by a group of functions in the standard library, namely malloc, realloc, calloc and free.
Dynamic allocation is one of the ways of using memo. To accomplish this in C, the malloc function is used and the "new" keyword is used for C++. Both of them perform an allocation of a contiguous block of memory, malloc taking the size as parameter. Example :

```
int *ptr=new int;
int *ptr=(int*)malloc(sizeof(int));      //sizeof is used for portability
```

When this memory is no longer required, then the keyword free is used to deallocate the memory so that it can be used for other purposes.

```
free(ptr);                //Deal locates memory
```

Local Memory:
The most common variables that are used are "local" variables within functions such as the Variables num and answer in the following function. All of the local variables and Parameters taken together are called its "local storage" or just its "locals". Example:

```
// Local storage example
int Square(int num) {
int answer;
answer = num * num;
return answer;
}
```

The variables are called "local" to capture the idea that their lifetime is tied to the Function where they are declared. Whenever the function runs, its local variables are Allocated. When the function exits, its locals are then deal located. For the above example, that

Means that when the Square () function is called, local storage is allocated for num and result. When the function finally exits, its local storage is deal located.

Local variables are also called "automatic" variables as their allocation and deallocation is done automatically as a part of the mechanism. All variables declared within a block of code are automatic by default, but this can be made explicit with the keyword "auto". Using the storage class register instead of auto is a hint to the compiler to cache the variable in a processor register. Other than not allowing the referencing operator (&) to be used on the variable or any of its subcomponents, the compiler is free to ignore any hint.

In C++, the constructor of automatic variables is called when the execution arrives at the place of declaration. Their destructor is called when it reaches the end of the given program block. This feature is often used to manage resource allocation and deallocation, like opening and then automatically closing files or freeing up memory.

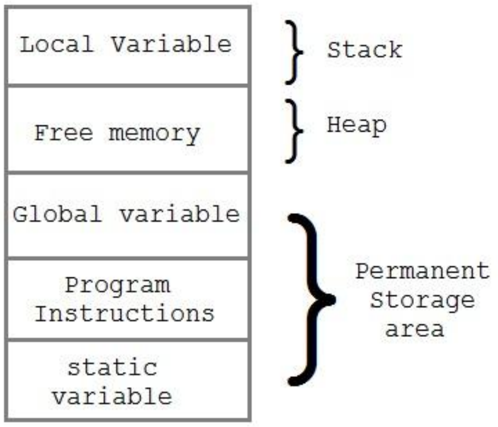

Figure 2

Now, let us take a look at the basic difference between malloc() and calloc() functions:

malloc()	calloc()
1. malloc() initializes the allocated memory with garbage value.	1. calloc() initializes the allocated memory with value 0.
2. Number of arguments taken here is 1.	2. Number of arguments taken here is 2.
3. Syntax: (cast_type*)malloc(size_in_bytes);	3. Syntax: (cast_type*)calloc(blocks,size_of_block);

Section III
Reference Parameters

In the simplest "pass by value" scheme, each function has separate, local memory and the parameters are copied from the caller to the callee when the function is called. Now, an alternative to using a return statement for getting back values from functions is to pass parameters by reference. It is seen that arrays are always passed in this way.

A parameter which is passed by value results in its value being copied into a local (automatic) variable within the called function's block at the point of call. Like all automatic variables this is lost at the end of the function's execution and any changes made to its value within the code of the function are also lost.

Parameters which are passed by reference, however, can be thought of having their addresses passed to the called function and so the called function can use this to change the contents of the variable in the calling function. We have already seen that "scanf" requires parameters to be passed in this way and that we have to use an ampersand (&) in front of the parameter name. The ampersand tells the compiler to pass over the memory address of the variable, and not its value.
Let us have a look at an example to examine a better difference between the usage of "pass by value" and "pass by reference" methods:
Pass by value:

```
void pass(int i) {
  i = 0;  /*  modifies the copy of the argument received by the function  */
}

int main()
{
  int k=10;
  pass(k);
  /*  k still equals 10, No change has occured  */
}
```

Pass by reference:

```
void passref(int *j) {
  *j = 0;
}

int main()
{
  int k=10;
  passref(&k);
  /*  k now equals 0 ,value has changed */
}
```

Now the question arises why do we need to pass the parameters by references? There are numerous advantages of passing by reference. It allows us to have the function change the value of the argument, which is sometimes very useful. Because a copy of the argument is not made, it is fast, even when used with large structures or classes. We can pass by const reference to avoid any unintentional changes. We can return multiple values from a function.
As every coin has two faces, pass by reference has some disadvantages as well. Because a non-const reference cannot be made to a literal or an expression, reference arguments must be normal variables. Also, it can be hard to tell whether a parameter passed by reference is meant to be input, output, or say both. It’s impossible to tell from the function call that the argument may change. An argument passed by value and passed by reference appears to be the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

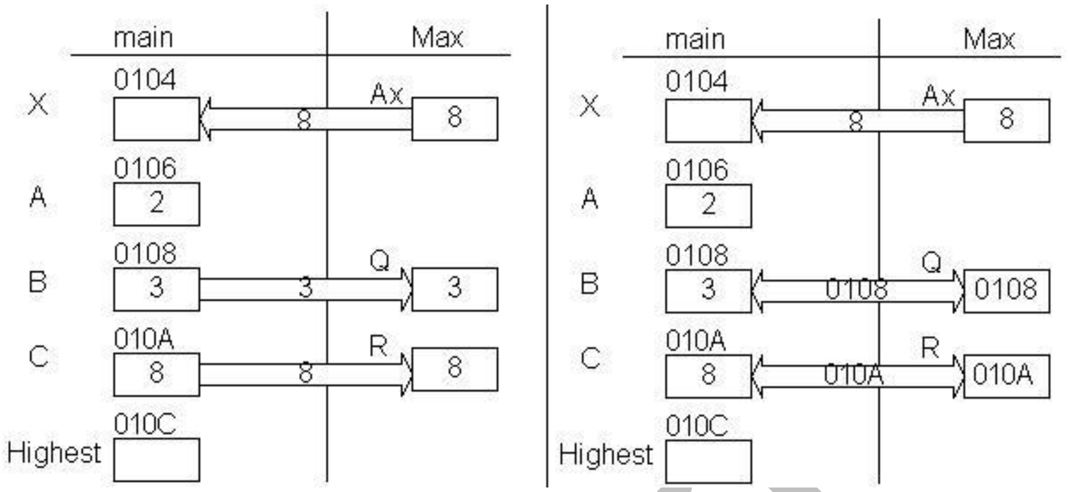

Figure 3

Is The & Always Necessary?
When passing by reference, the caller does not always need to use operator & to compute a new Pointer to the value of interest sometimes the caller already has a pointer to the value of Interest and so no new pointer computation is required. The pointer to the value of Interest can be passed through unchanged.
For example, suppose B() is changed so it calls a C() function which adds 2 to the value Of interest:

```
// Takes the value of interest by reference and adds 2.
void C(int* ref) {
*ref= *ref + 2;
}
// Adds 1 to the value of interest, and calls C().
void B(int *ref) {
*ref = *ref + 1; // add 1 to value of interest as before
C(ref);          /* no & required. We already have
                    a pointer to the value of interest, so
                    it can be passed through directly. */
}
```

Section IV
Heap Memory
"Heap" memory, also known as "dynamic" memory, is an alternative to local stack memory. The **heap** is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory and is comparatively larger. To allocate memory on the heap, you must use malloc() or calloc(), which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using free() to deal locate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside and won't be available to other processes.

The heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap.
Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

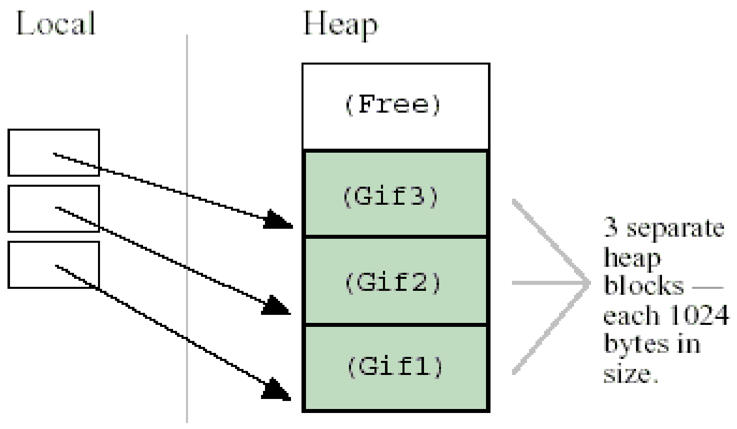

Figure 4

The following program shows how you can create and use a heap that uses regular memory

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
  void *p;
  /* Round up to the next chunk size */
  *length = ((*length) / 65536) * 65536 + 65536;
  *clean = _BLOCK_CLEAN;
   p = calloc(*length,1);
   return (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
  free( p );
  return;
}

int main(void)
{
  void    *initial_block;
  long   rc;
  Heap_t  myheap;
  char    *ptr;
```

```
int     initial_sz;
/* Get initial area to start heap */
initial_sz = 65536;
initial_block = malloc(initial_sz);
if(initial_block == NULL) return (1);
/* create a user heap */
myheap = _ucreate(initial_block, initial_sz, _BLOCK_CLEAN,
                  _HEAP_REGULAR, get_fn, release_fn);
if (myheap == NULL) return(2);

/* allocate from user heap and cause it to grow */
ptr = _umalloc(myheap, 100000);
_ufree(ptr);

/* destroy user heap */
if (_udestroy(myheap, _FORCE)) return(3);

/* return initial block used to create heap */

free(initial_block);
return 0;
```

When do we need to use the Heap?

When should you use the heap, and when should you use the stack? If you need to allocate a **large block** of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap. If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack, it's easier and faster. If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like malloc(), calloc(), realloc() and free() to manage that memory "by hand".

CONCLUSION:

We, in this paper, discussed about the facts and concepts beginning from the basics to the core of pointers and memory. Without pointers, we would have to allocate all the program data globally or in functions or the equivalent, and we would have no recourse if the amount of data grew beyond what we had originally allowed for. In most languages that use pointers, there are certain sorts of references that are pointers, and perhaps certain sorts of references that aren't, and there is no further notational difference. One of C's design goals was to write Unix in, and therefore it needed to handle memory locations in a detailed manner. Therefore, it is possible to manipulate C pointers directly, assigning memory addresses and calculating new ones. in most languages nowadays, you use pointers constantly without being reminded of the fact.

REFERENCES:

1. http://cslibrary.stanford.edu/102/PointersAndMemory.pdf
2. http://en.wikipedia.org/wiki/Pointer_(computer_programming)
3. http://www.codingunit.com/c-tutorial-call-by-value-or-call-by-reference
4. http://stackoverflow.com/questions/10200628/heap-memory-in-c-programming
5.http://en.wikipedia.org/wiki/C_dynamic_memory_allocation