International Journal of Scientific & Engineering Research, Volume 4, Issue 11, November-2013
ISSN 2229-5518


# Role of memory allocation in 'C' programming

Reshant Chandra, Saurabh Rawat

**Abstract**— In the following paper different type of memory allocation for different type of data in a C program has been explained. This paper deals with different segment where data is stores and its effect on the output of the program. This paper illustrates and explains the various problems with memory allocation such as stack overflow and fragmentation. We have also shown with the help of a code the cause of fragmentation in heap. This paper sees a C program from a memory point of view, two example codes has also been illustrated whose output is not as expected cause of memory allocation.

**Index Terms**— Fragmentation, Heap, Memory allocation, recursive function, Signed integer, Stack, Stack overflow,Unsingned integer

## 1 Introduction

Simple C programs need to go through a series of process before it is can be executed. First all the macros are included using pre-processor into the source code. The modified source code is then compiled using compiler, it checks for syntax error and converts the texted source code into equivalent machine language code. The end result of compiler is a reallocated object file with extension '.obj'. Then the linker combines all the reallocated object files, resolving all external reference in different file. It replaces function call with the address of calling function. The end product is executable file with extension '.exe'.

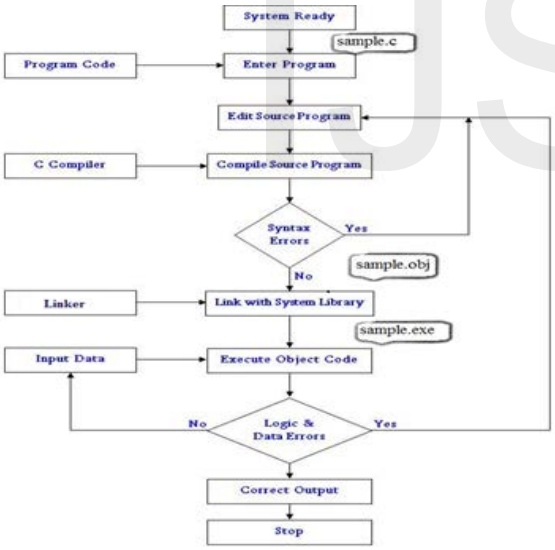


Fig. 1

- *Author Reshant Chandra is a faculty of computer science department in Graphic Era University, India. E-Mail: reshant_chandra@gmail.com*
- *Co-Author Saurabh Rawat, Graphic Era University, India.*

## 2 Memory Layout Of C Program

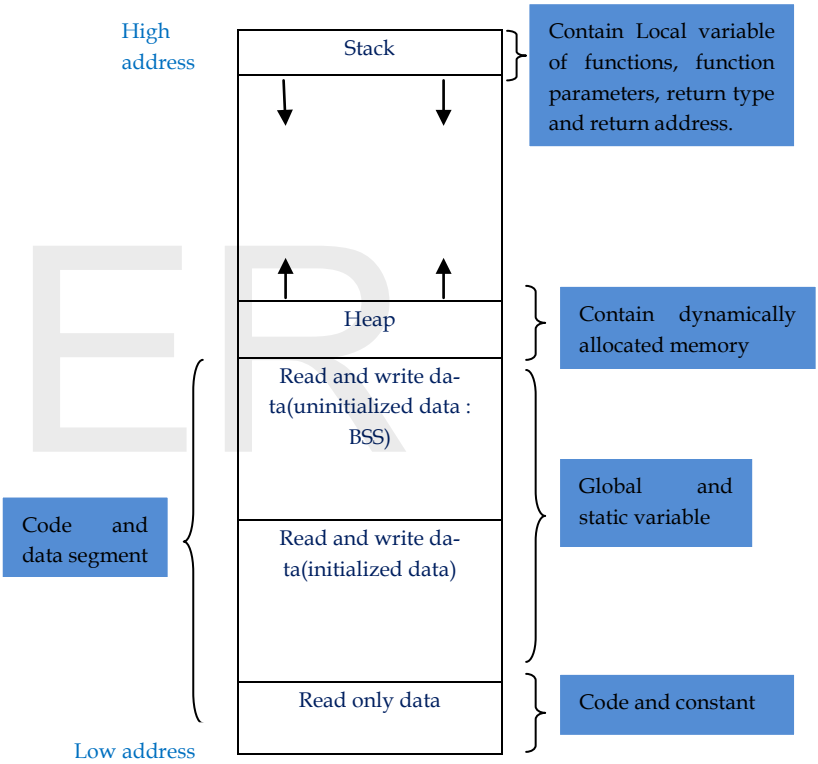


Fig 2 Memory layout of a running process

## 3 Stack

Where local variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, are saved on the stack. The newly called function then allocates room on the stack for its local variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function. Once the function is complete its entry is pop out from the stack.

International Journal of Scientific & Engineering Research Volume 4, Issue 11, November-2013
ISSN 2229-5518

Let us consider an example to illustrate the function call in stack:

```
main(){
        int n=5;
        int sum=0;
        sum= calculate(n);
        printf("%d",sum);
}
int calculate(int n){
        if(n<=1){
            return n;
        }
        else{
            return (n+ calculate(n-1)); }
}
```

It's a simple program to calculate sum of first n natural number using recursion. Here n is considered to be 5. Expected result is 5+4+3+2+1. Let's see how it is working at memory level in fig 3.1 and fig 3.2.

For each recursive call a separate memory is allocated of it with separate local variable in the stack. Once it reaches to the at most depth back tracking take place as shown in fig 3.1.

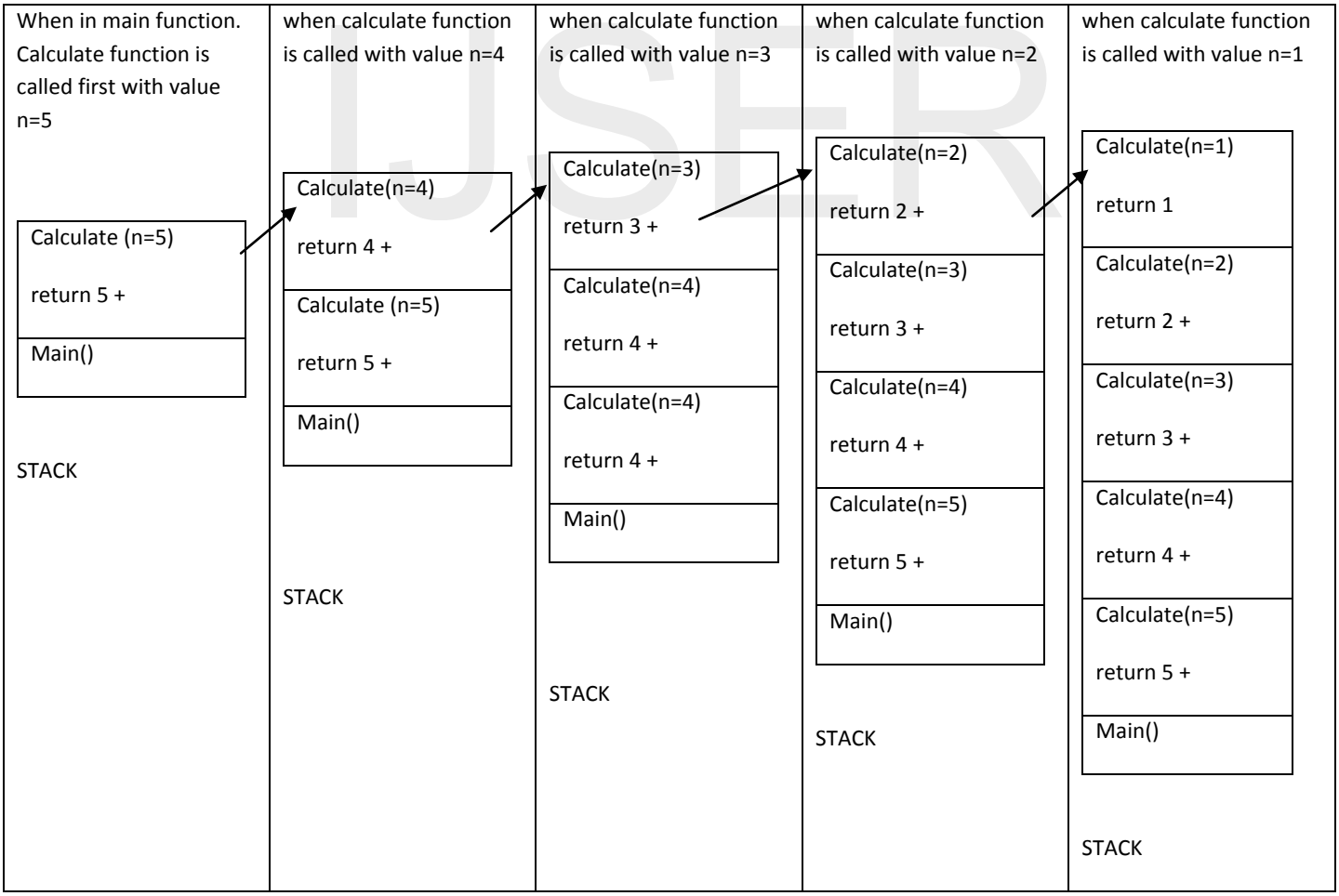


Fig 3.1 recursive calling of a function

International Journal of Scientific & Engineering Research Volume 4, Issue 11, November-2013
ISSN 2229-5518

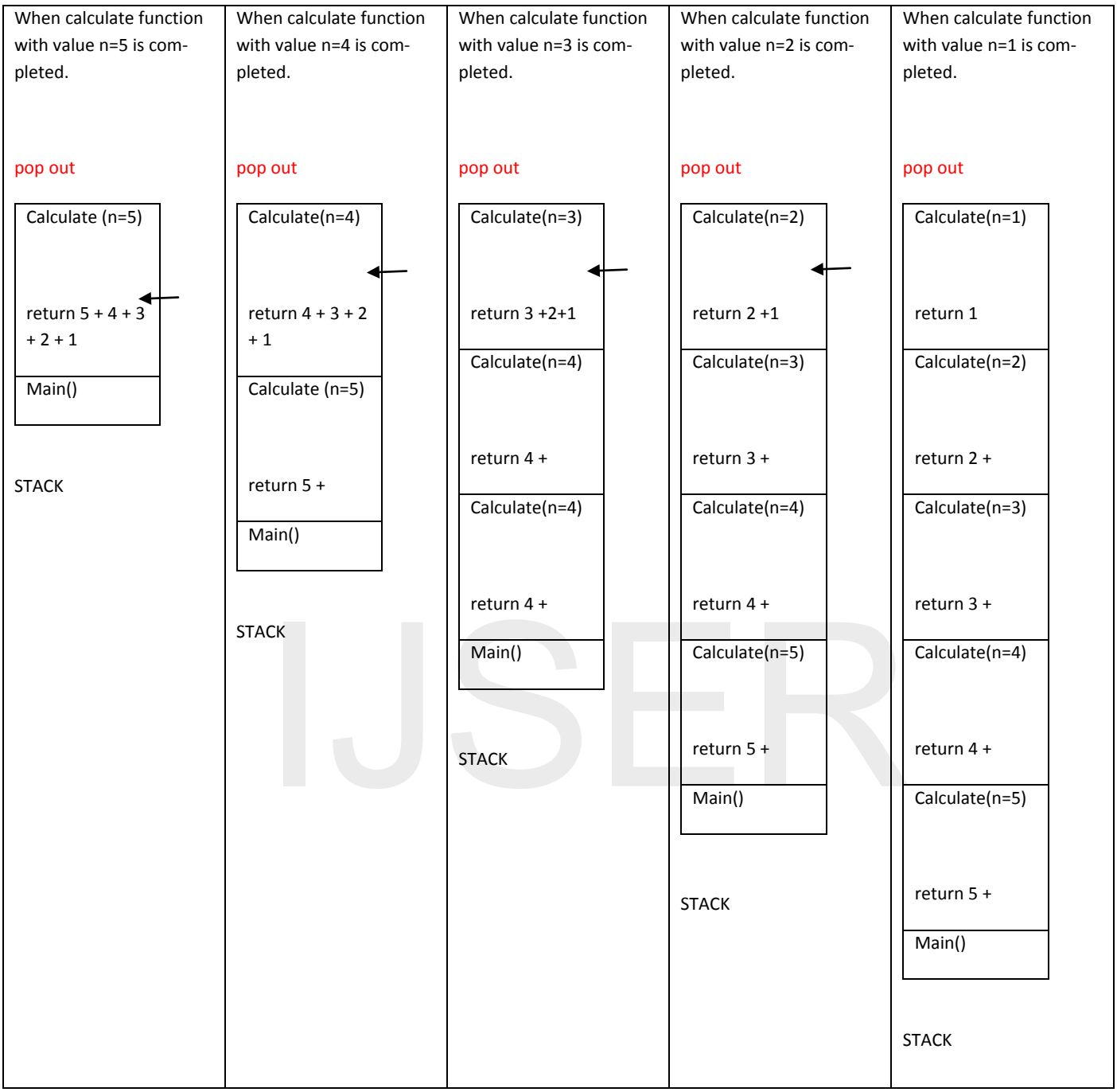


Fig 3.2 Backtracking of recursive call

Hence when it finally return to main
Sum= 5+4+3+2+1= 15 as expected.
Hence the main function will print the value 15

### 3.1 Stack overflow

IJSER stack has a limitation the memory allocated to it is fixed. What if we run out of memory in stack? A stack overflow occurs when the stack pointer exceeds the stack bound. It is caused by two factors.

a. *Too many nested function call*: let suppose in the last example the value of n is a very large number close to infinity. Then the number of function call will be close to infinity too. Since for each call a separate memory is allocated in the stack, the stack will run out of memory soon. This will result in stack over flow.

b. Very large local variable: The other major cause of a stack overflow results from an attempt to allocate more memory on the stack than will fit, for example by creating local array variables that are too large.

International Journal of Scientific & Engineering Research Volume 4, Issue 11, November-2013
ISSN 2229-5518

Let's see an example

```
int main(){
    double x[1000000]; }
```

The declared array consumes 8 megabytes of data (assuming each double is 8 bytes); if this is more memory than is available on the stack, a stack overflow will occur

### 3.2 Possible solution to stack overflow

a. If possible we can replace the recursion call into loop, then we can avoid stack overflow as in loop memory in allocated only once while in recursion or nested function call a separate memory is allocated every time a new function is called. For example converting the last shown code from recursion to loop for function 'calculate'

```
int calculate(int n){
        int sum = 0;
        while(n>=1){
                sum = sum + n;
                n=n-1;
        }
        return sum;
}
```

This function won't create stack over flow like previous one for any value of n.

b. By declaring array with large memory allocation dynamically or globally instead of locally.

## 4 Heap

The task of fulfilling an allocation request consists of locating a block of unused memory of sufficient size. Memory requests are satisfied by allocating portions from a large pool of memory called *the heap*. At any given time, some parts of the heap are in use, while some are "free" (unused) and thus available for future allocations. Dynamic memory allocation is performed in the C programming language via a group of functions in the C standard library, namely malloc, realloc, calloc and free.Pointers are use to point the dynamically allocated memory.

### 4.1 Fragmentation in Heap

The heap method suffers from fragmentation. Like any method of memory allocation, the heap will become fragmented; that is, there will be sections of used and unused memory in the allocated space on the heap. A good allocator will attempt to find an unused area of already allocated memory to use before resorting to expanding the heap. The major problem with this method is that the heap has only two significant attributes: base, or the beginning of the heap in virtual memory space; and length, or its size. The heap requires enough system memory to fill its entire length, and its base can never change. Thus, any large areas of unused memory are wasted. The heap can get "stuck" in this position if a small used segment exists at the end of the heap, which could waste any magnitude of address space, from a few megabytes to a few hundred.

### 4.2 Code to illustrate that memory is allocated in chunks which may cause fragmentation:

```
main()
{   char*p,*ptr;            //two pointers are declared. Line 1

    p=(char*)calloc(2,sizeof(char));    // memory space of 2
    character are allocated to p. Line 2

    ptr=(char*)calloc(16,sizeof(char));          // memory
    space of 16 character are allocated to ptr Line 3

    strcpy(p,"abcdefghijklmnopqrstuvwxyz"); // a string
    is copied to p. Line 4

    strcpy(ptr,"hello world"); // a string is copied to ptr.
    Line 5

    printf("\n%s",p);       // pointer p is printed. Line 6
}
```

Due to the rules governing memory allocation, more computer memory is sometimes allocated than is needed. For example, memory can only be provided to programs in chunks divisible by 4, 8 or 16, and as a result if a program requests perhaps 23 byte, it will actually get a chunk of 24. Here we are considering a Chunk of 16 byte.
Since there are no other allocation and de-allocation taking place we are considering that the memory allocated are in sequence as shown in fig 4.1 (which will be the case most of the time).

Memory allocated to p:
In line 2 Requested for 2 bytes. It will actually get a chunk of 16 bytes as shown in fig 4.1.
Memory allocated to ptr:
In line 2 Requested for 16 bytes, as it matches the size of the chunk it will get 16 bytes as shown in fig 4.1.

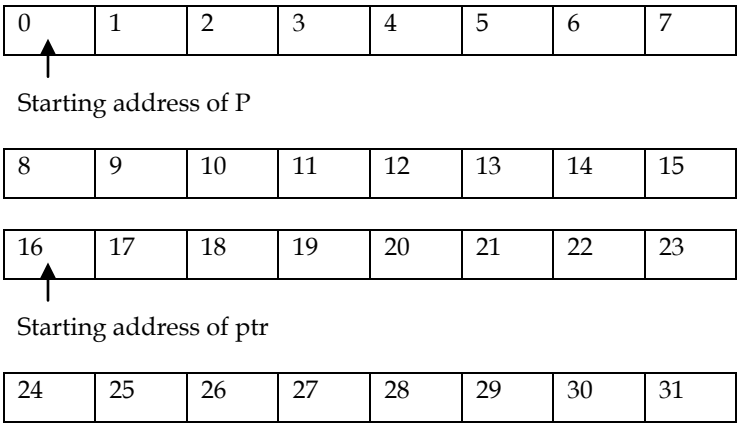


Fig 4.1 Memory allocation for P and Ptr pointer

International Journal of Scientific & Engineering Research Volume 4, Issue 11, November-2013
ISSN 2229-5518

After line 4 the memory allocation will be:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | H |
| i | j | k | l | m | n | o | P |
| q | r | s | t | u | v | w | X |
| y | z | \0 | | | | | |

Fig 4.2 Memory allocation for P and Ptr pointer

As strcpy function will copy the string starting from the memory location denoted by p, sequentially till the entire string is copies and then will insert '\0' denoting end of string. It will go out of bound if memory allocated us not sufficient.

After line 5. The string "hello world" will be printed starting from memory location denoted by pointer ptr. It will overwrite the element inserted by line 4.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | H |

Starting address of pointer p

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| i | J | k | l | m | n | o | P |
| h | e | l | l | o | | w | o |

Starting address of pointer ptr

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| r | l | d | \0 | | | | |

Fig 4.3 Memory allocation for P and Ptr pointer

Hence finally the output in line 6 will be "abcdefghijklmnophello world" since "%s" will print till it finds a '\0'.
Hence we can conclude that:
Dynamic memory is allocated in chunks. Although pointer p asked for only 2 byte of memory it was allocated 16 bytes of memory. If it requires only 2 bytes rest 14 bytes are wasted causing internal fragmentation.

## 5 Effect/Error of memory allocation in C programming

1. Consider the following code snippet:

```
main()
{
    int *ptr ;          // an integer pointer
    ptr = sample();     //returns address pointer
    printf("%d",ptr[2]);
}
```

```
int* sample()  //function which return address
{
    int array[5]={1,2,3,4,5};   //array defined
    return array;               //it will return the starting
}                               address of the array
```

Because
In the above code the expected answer is 3, but it will print garbage value. The inconsistence is cause of the stack. When function 'sample' was called in the main function a memory was allocated for it in the stack along with the memory for its local variable which in this case is array [5]. When the function completed it phase it is pops out of the stack and the memory is de-allocated hence the reference of its local variable is lot which result in garbage value as pointer ptr is pointed to a lost reference. Hence we can conclude that **reference of local variable declared in the function are lost once it completes it's course.**

2. Consider the following code snippet:

```
main(){

    int i = -1;     //Line 1 .Declaring signed integer i and assigning -1 to it.

    unsigned int u=1;  // Line 2. Declaring unsigned integer u and assigning 1 to it.

    If(i<u){                        // Line 3
        Prinf("u is greater");      // Line 4
    }  else{                        // Line 5
        Printf("i is greater");     // Line 6
    }}
```

Expected result "u is greater" actual result "i is greater"

To find the answer to the code we need to first understand how integer is stored in the memory. We have a 4 byte integer i.e. 32 bit integer. Hence the integer decimal value will be converted into binary and then saved as 32 bit. 31 bits representing the value and left most bit representing the sign. (1 for negative and 0 for positive.)

int i = -1 will be saved as shown in fig 5.1.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

It represents sign of integer

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Fig 5.1 Memory allocation of signed interger

International Journal of Scientific & Engineering Research Volume 4, Issue 11, November-2013
ISSN 2229-5518

Unsigned int u = 1 will be saved as shown in fig 5.2.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Fig 5.2 Memory allocation of unsigned interger

In fig 5.1 left most bit indicate the sign of the integer and rest indicate the value of the integer. In fig 5.2 all 32 bits indicate the value of the integer there is no sogn bit in the unsidned integer.

In line 3 of the given code type casting will take place since i and u are two different data type. Signed int will be converted to unsigned int since vice versa may result to loss of data. After type casting the int i = -1 will be represented as shown in fig 5.3.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Fig 5.3 Memory alter type casting of signed integer

The entire 32 bit represents value of the integer (fig 5.3) there is no signed bit. Hence it will be very large number. **Hence the if condition in line 3 will hold false and line 6 will be printed**

## 6 Conclusion

We can conclude that a C program can be divided broadly into three parts, stack memory for function call and local variable, heap for dynamically allocated memory and code segment containing the text code, global and static variable. We have illustrated the function of stack in case or recursive function along with stack overflow problem and its possible solution. We have also learned that memory allocated in heap are in small chunks which may lead to fragmentation. Viewing a C program from memory point of view has helped us in explaining the unexpected output of a program. In the end we can finally say that all the execution is done at the memory level hence understanding the memory allocation makes us develop better and more efficient programs.

## References

[1] *Kernighan; Dennis Ritchie (March 1988). The C Programming Language (2nd ed.).*
[2] *Donald E. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Programming,. Addison Wesley, Reading, MA, 2nd edition, 1973.*
[3] *Introduction to Data Structures in C ,By: Ashok N. Kamthane*
[4] *PROGRAMMING IN C, By: Ashok Kamthane*
[5] *Advanced Memory Management: Dynamic Allocation, Part 1, By Andrei Milea*
[6] Stroustrup, Bjarne (2008). *Programming: Principles and Practice Using C++*
[7] .Summit, Steve. "C Programming Notes - Chapter 11: Memory Allocation"
[8] Links: http://en.wikipedia.org/wiki/Memory_pool
http://en.wikipedia.org/wiki/C_dynamic_memory_allocation
http://en.wikipedia.org/wiki/Fragmentation_(computer)

IJSER