Recap: Did you Read on

- **Using C programming**
  - **Creating a new process using *fork() system call***
  - **CREATING multiple processes using fork () and pipe()**
  - **Demonstrate the concept of Orphan Process.**
  - **Using execv and execvp system calls**

What did you find out?

ICS 2305 : SYSTEMS PROGRAMMING

MEMORY MANAGEMENT

Karanja Mwangi

Lesson objective

- **At the end of this class you will**
  - Appreciate the concepts of Memory Management as used in Operating systems
  - Differentiate the roles of global, stack and heap memory in C
  - Describe and implement dynamic and static Memory allocation
  - Explore the concept of releasing memory

Memory Management

- In previous studies of Operating systems , You learnt that

Memory management is the process of controlling and coordinating the way a software application access **computer memory specifically the Random Access Memory**.

Hard disk drives have huge space and seemingly can be added “infinitely”, but **RAM is not infinite**. If a program keeps on consuming memory without freeing it, ultimately it will run out of memory and crash itself or even worse crash the operating system

Memory Management -2

- When a program runs on a target Operating system say Linux it needs access to the computers **RAM**(Random-access memory) to:
  - load its own **bytecode** that needs to be executed
  - store the **data values** and **data structures** used by the program that is executed
  - load any **run-time systems** that are required for the program to execute

Variable Types and Memory Allocation-2

- We have different types of variables : Global or local
- The global variables ~ used throughout the program by different functions and blocks.
  - Hence memory area allocated to them needs to exist throughout the program.
  - Hence they get memory allocated at the internal memories of the system, which are known as **permanent storage area** or the **Global Memory**.
  - Similarly the program and their statements also need to exist throughout when system is on. Hence they also need to occupy permanent storage area

Variable Types and Memory Allocation-3

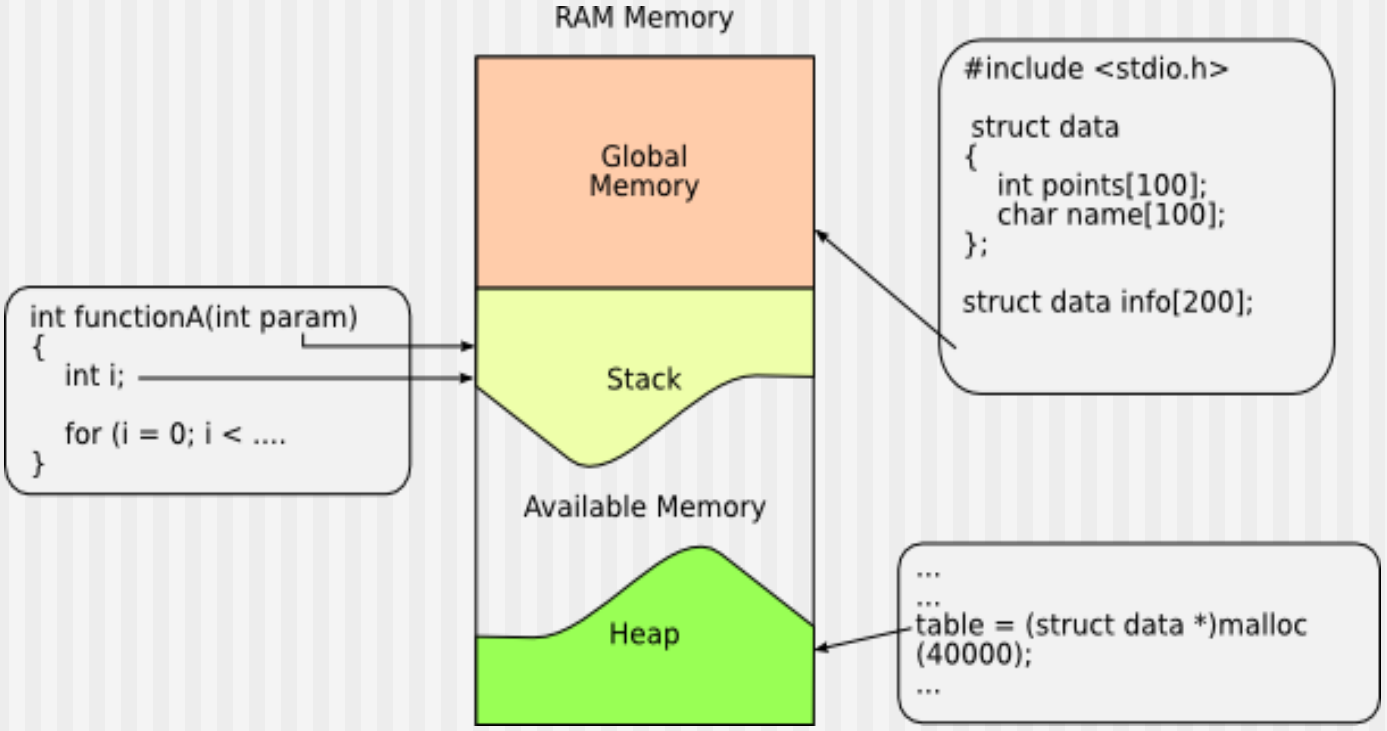
- **Local variables** ~ exists in the particular block or function where they are declared.
- If we store them in permanent storage area, it will be waste of memory as we keep the memory allocate which are not in use. Hence we use stack memory to store the local variables and remove them from the stack as the use of local variable is over.
- There is a free memory space between this stack memory and permanent storage area called heap memory.
- This memory is flexible memory area and keeps changing the size. Hence they are suitable for allocating the memory during the execution of the program. That **means dynamic memory allocations** use these heap memories.

Variable Types and Memory Allocation-4

- In Summary , C program stores all its memory data in three different areas.
  - **Global Memory or permanent storage** – stores data that are present from the very beginning of a program until the end of the execution.
  - **The stack** -It is an area in which the variables appear and disappear at a certain point during the execution of a program. It is used mainly to store the variables local to a function.
  - **The Heap**. This area contains memory available to be reserved and freed at any point during the execution of a program. **It is not reserved for local variable functions as in the stack,** but for memory known as "dynamic" for data structures that are not known to be needed or even their size until the program is being executed

Variable Types and Memory Allocation-5

- Note that only the **global memory** has a fixed size that is known when the program starts execution. Both the stack and the heap store data the size of which cannot be know until the program is executing

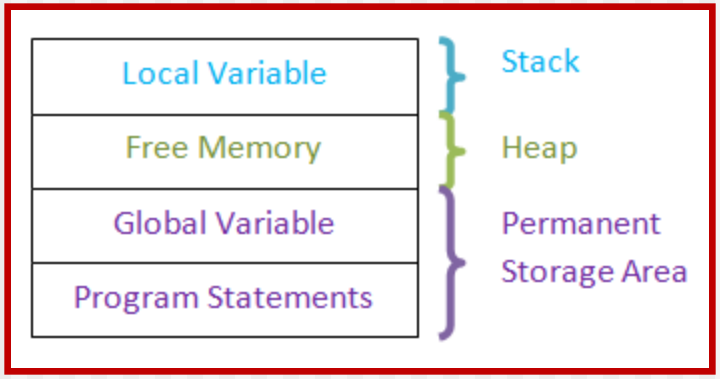


Variable Types and Memory Allocation-6

- Trivial (Global, Stack or Heap)
  - static variable is stored in …….
  - A parameter that is passed to a function is stored in……..
  - A local variable is stored in ……..

Variable Types and Memory Allocation-7

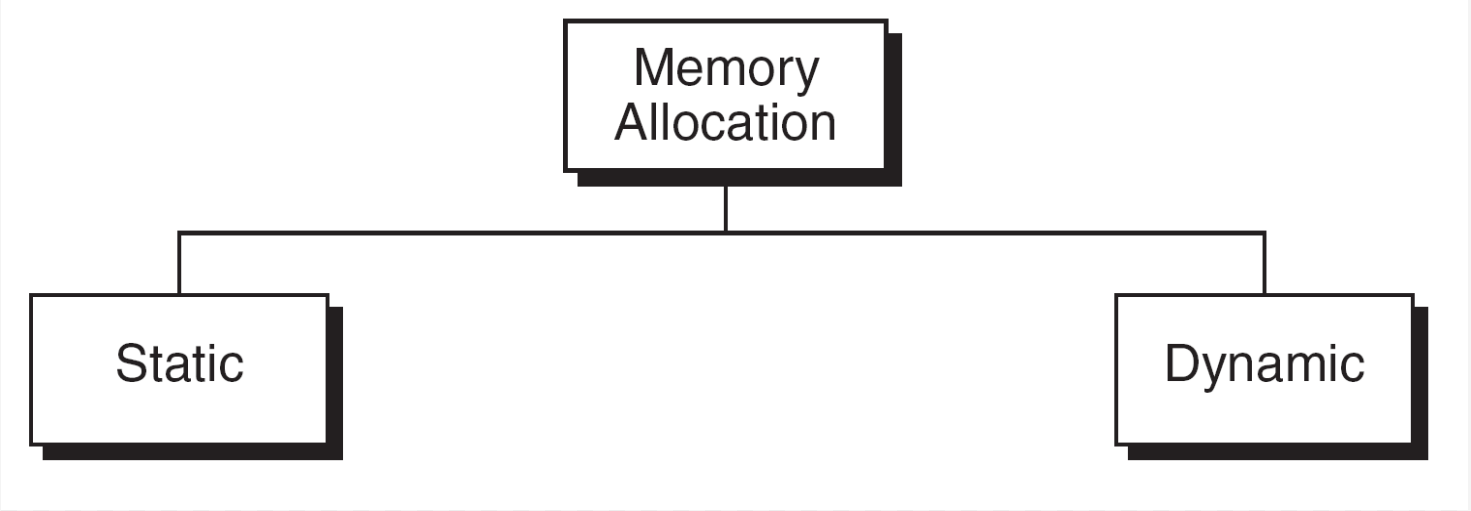
- Trivial (Global, Stack or Heap)
  - static variable is stored in ……. Global memory
  - A parameter that is passed to a function is stored in……..Stack Memory
  - A local variable is stored in …….. Stack Memory



Variable Types and Memory Allocation

- In C programming, variable type and the amount of memory allocated are strongly linked
  - int : integer number : 4-bytes
  - short : integer number : 2-bytes
  - long : integer number : 8-bytes
  - char : character : 1-byte
  - float : floating point number : 4-bytes
  - double : floating point number : 8-bytes
  - void * : pointers : 8-bytes on (64 bit machines)

Memory Allocation in C



Local Memory Allocation: Static Memory Allocation-1

- The **stack** is used for static memory allocation
- last in first out(**LIFO**) stack…
  - **push** : push an item on to the top of the stack
  - **pop** : pop an item off the top of the stack
- Why the **stack**?
  - stack is **very fast** as there is no lookup required, you just store and retrieve data from the topmost block on it.
  - stack has to be **finite and static**(The size of the data is known at compile-time).
  - The **stack frame** contains all the allocated memory from variable deliberations as well as a pointer to the execution point of the calling function is structured as a stack
  - **Multi-threaded applications** can have a **stack per thread**.
  - Memory management of the stack is **simple and straightforward** and is done by the OS.

Static Memory Allocation -3

- In the program below to add two integers

```
#include <stdio.h>
int main() {
int number1, number2, sum;
printf("Enter two integers: ");
scanf("%d %d", &number1, &number2);
// calculating sum
sum = number1 + number2;
printf("%d + %d = %d", number1, number2, sum);
return 0;
}
```

  - The declaration of the integer number1, number2, sum will allocate memory for the storage for an integer (4-bytes) to each of the variable. At compile time itself we know that there are 3 integer variables. Hence it is easy for the compiler to reserve the memory for these variables.
  - Such reserved variables will have same size and memory address till the end of the program. There will not be any change in size, type and memory location for those variables.

Static Memory Allocation -2

- In summary for Static Memory allocation
  - There is no need to explicitly allocate memory to the variables.
  - When a function is called, memory is allocated for all of its parameters and local variables.
  - Each active function call has memory on the stack (with the current function call on top)
  - When a function call terminates,

    the memory is deallocated (“freed up”)
  - These variables can be local or global variables. But we need to know in advance the size and type of the variable
  - variables cannot store more than predefined size of data

Dynamic Memory Allocation -Problem with Arrays 1

- Suppose we need to add any number of numbers that are entered by the user. Here we are not sure about how many numbers are entered by the user. We only know that it he is entering only integers.
- In this case we cannot pre-assign any memory to the variables. He can enter only 2 numbers or 100s of numbers. If user enters less numbers then the program should be flexible enough to assign to those less number of numbers and as the numbers increase memory allocation space also should increase. But this can be determined only at the run time – depends on the user who enters the value. Thus we need to allocate space at the run time which is done by **using dynamic memory allocation methods.**

Dynamic Memory Allocation -Problem with Arrays 2

- Example: Search for an element in an array of N elements
- One solution: find the maximum possible value of N and allocate an array of N elements
  - Wasteful of memory space, as N may be much smaller in some executions
  - Example: maximum value of N may be 10,000, but a particular run may need to search only among 100 elements
  - Using array of size 10,000 always wastes memory in most cases

Better Solution

- Dynamic memory allocation
  - Know how much memory is needed after the program is run
    - Example: ask the user to enter from keyboard
  - Dynamically allocate only the amount of memory needed
- The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.
  - malloc, calloc, realloc, free

Memory Allocation Functions

- malloc
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space
- calloc
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- free
  - Frees previously allocated space.
- realloc
  - Modifies the size of previously allocated space.

Malloc: Allocating a Block of Memory

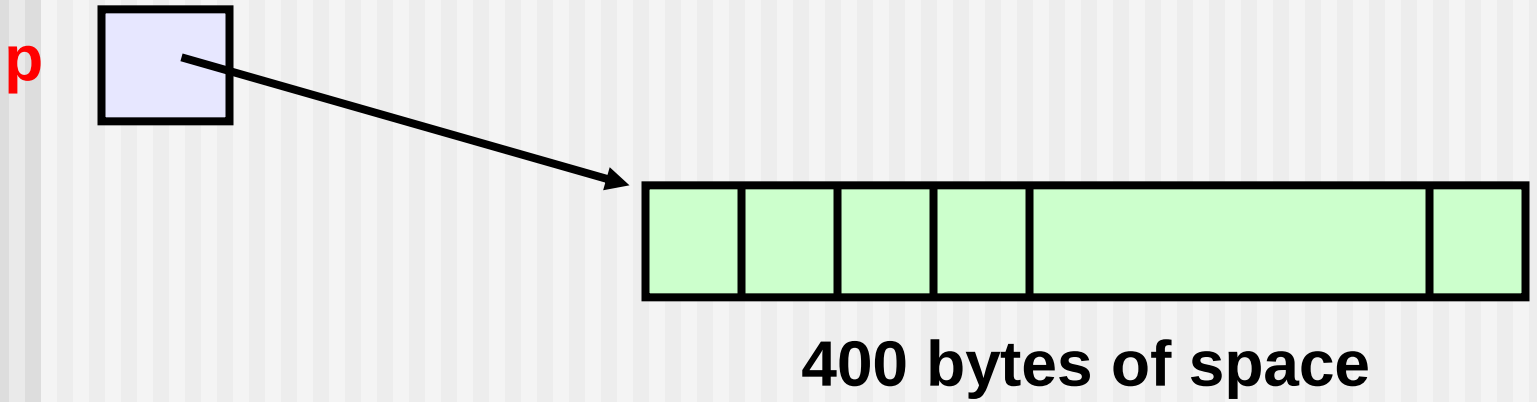
- A block of memory can be allocated using the function malloc
  - Reserves a block of memory of specified size and returns a pointer of type void
  - The return pointer can be type-casted to any pointer type
- General format:

```
type *p;
p =  (type *) malloc (byte_size);
```

Example

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

- A memory space equivalent to 100 times the size of an int bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer p of type int



Contd.

- cptr = (char *) malloc (20);

  Allocates 20 bytes of space for the pointer cptr of type char

- sptr = (struct stud *) malloc(10*sizeof(struct stud));

  Allocates space for a structure array of 10 elements. sptr points to a structure element of type struct stud

Notes on Malloc

- malloc always allocates a block of contiguous bytes
  - The allocation can fail if sufficient contiguous memory space is not available
  - If it fails, malloc returns NULL

```
if  ((p = (int *) malloc(100 * sizeof(int))) ==
   NULL)
  {
      printf ("\n Memory cannot be
   allocated");
      exit();
   }
```

Notes on Using the malloc on Array

- Once the memory is allocated, it can be used with pointers, or with array notation
- Example:

```
int *p, n, i;
scanf("%d", &n);
p = (int *) malloc (n * sizeof(int));
for (i=0; i<n; ++i)
    scanf("%d", &p[i]);
```

The n integers allocated can be accessed as *p, *(p+1), *(p+2),…, *(p+n-1) or just as p[0], p[1], p[2], …,p[n-1]

Example: The n integers allocated can be accessed as *p, *(p+1), *(p+2),..., *(p+n-1) or just as p[0], p[1], p[2], ...,p[n-1]

```
int main()
{
  int i,N;
  float *height;
  float sum=0,avg;

  printf("Input no. of students\n");
  scanf("%d", &N);

  height = (float *)
       malloc(N * sizeof(float));
```

```
printf("Input heights for %d
students \n",N);
  for (i=0; i<N; i++)
   scanf ("%f", &height[i]);

  for(i=0;i<N;i++)
    sum += height[i];

  avg = sum / (float) N;

  printf("Average height = %f \n",
                avg);
  free (height);
  return 0;
}
```

Free: Releasing the Allocated Space:

- An allocated block can be returned to the system for future use by using the free function
- General syntax:

  free (ptr);

  where ptr is a pointer to a memory block which has been previously created using malloc
- Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

Can we allocate only arrays?

- malloc can be used to allocate memory for single variables also
  - **p = (int *) malloc (sizeof(int));**
  - Allocates space for a single int, which can be accessed as *p
- Single variable allocations are just special case of array allocations
  - Array with only one element

Pointers to Pointers

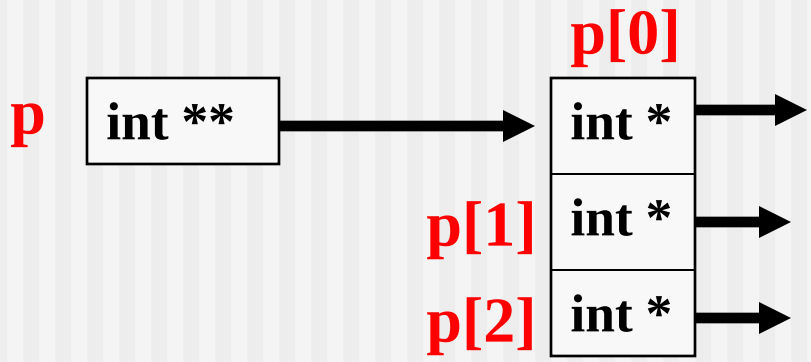
- Pointers are also variables (storing addresses), so they have a memory location, so they also have an address
- Pointer to pointer – stores the address of a pointer variable

```
int x = 10, *p, **q;
p = &x;
q = &p;
printf("%d  %d  %d", x, *p, *(*q));

will print 10  10  10 (since *q = p)
```

Allocating Pointer to Pointer

```
int **p;
p = (int **) malloc(3 * sizeof(int *));
```



Dynamic Allocation of 2-d Arrays

- Recall that address of [i][j]-th element is found by first finding the address of first element of i-th row, then adding j to it
- Now think of a 2-d array of dimension [M][N] as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k-th row can be found by adding 1 to the starting address of (k-1)-th row)
- This is done by allocating an array p of M pointers, the pointer p[k] to store the starting address of the k-th row

Contd.

- Now, allocate the M arrays, each of N elements, with p[k] holding the pointer for the k-th row array
- Now p can be subscripted and used as a 2-d array
- Address of p[i][j] = *(p+i) + j (note that *(p+i) is a pointer itself, and p is a pointer to a pointer)

Dynamic Memory Allocation Summary

- Heap-allocated memory
  - This is used for *persistent* data, that must survive beyond the lifetime of a function call
    - global variables
    - dynamically allocated memory – C statements can create new heap data
    - Heap memory is allocated in a more complex way than stack memory
  - Like stack-allocated memory, the OS determines where to get more memory – the programmer doesn't have to search for free memory space!

Dynamic Memory Vs. Static Allocation

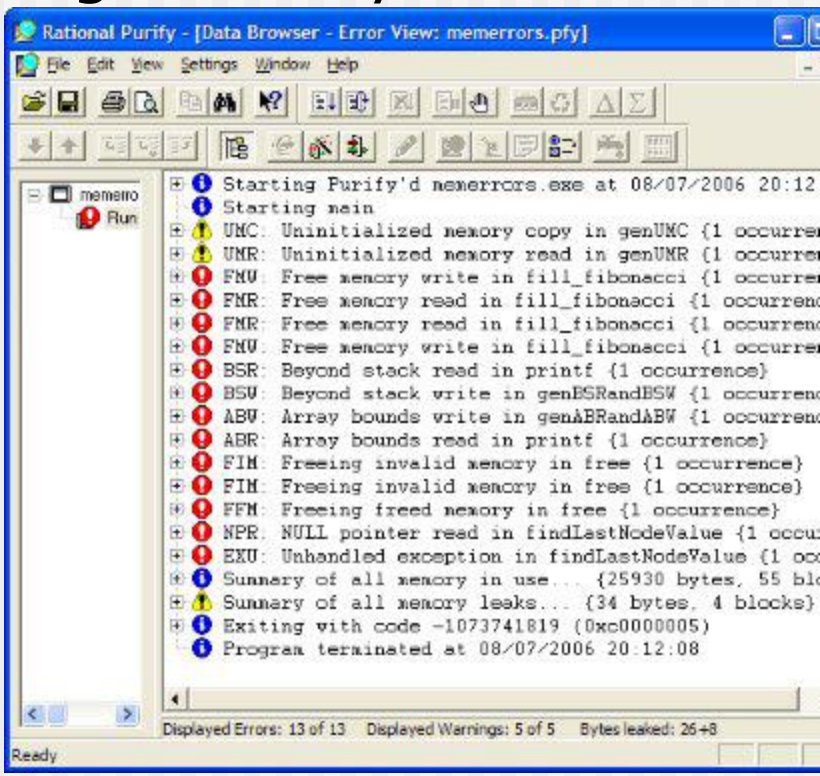
- Static memory assignment is at compilation time while the dynamic can be assigned only at the run time.
- We can refer to memory allocated in the heap only through a pointer.
  - Using a pointer after its memory has been released is a common programming error. Guard against it by clearing the pointer. (see examples http://www.cs.ecu.edu/karl/3300/spr16/Notes/C/Memory/dangling.html
  - The pointer used to free memory must be of the same type as the pointer used to allocate the memory.

Summary

Memory Allocation	Description
Static Memory Allocation	Allocated when variables are declared. Compiler knows in advance the size of the variables. Allocated memory is fixed and does not change in the program. Allocated memory is released at the end of the program. Usually local, static, and global variables are allocated statically.
Dynamic Memory	Memory is allocated at the run time. Compiler cannot determine the size and type of the variable before executing the program. Allocated memory can increase or decrease or remain same throughout the program. Allocated memory need not be at the same location throughout the program. We need to explicitly release the allocated memory in the program, if not used. Usually used for pointers, arrays and structures.
malloc()	This function allocates specified amount of memory in bytes and points to the beginning of the assigned memory location. It returns the void pointer to the variable. We need to typecast while allocating memory to get the datatype of the variable. It does not initialize the value and hence contains garbage value at the beginning. Needs to pass the total memory size that needs to be allocated.
calloc()	This function allocates specified amount of memory in bytes and points to the beginning of the assigned memory location. It returns the void pointer to the variable. We need to typecast while allocating memory to get the datatype of the variable. It initializes the value to zero when memory is allocated. Needs to pass the total number of data to be stored and the size of the individual data.
realloc ()	This will reassign the memory to the previously allocated variable. It can increase or decrease the memory allocated. This will allocate memory at totally new memory address than the previous.
free ()	It frees the memory held by the variable.

Tools for analyzing memory management

- Purify: runtime analysis for finding memory errors
  - dynamic analysis tool: collects information on memory management while program runs
  - contrast with static analysis tool like `lint`, which analyzes source code without compiling, executing



Further Reading and Reference

- Read about Functions and Pointers in C
  - Parameter Passing in C : What do we mean when we say : call by value and call by reference?
- S.C. Gupta and S. Sreenivasamurthy. "Navigating 'C' in a 'leaky' boat? Try Purify". http://www.ibm.com/developerworks/rational/library/06/0822_satish-giridhar/
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-s096-introduction-to-c-and-c-january-iap-2013/lectures-and-assignments/c-memory-management/

Exercises -1

Attempt the C programming Exercise

https://disalw3.epfl.ch/teaching/signals_instruments_systems/ay_2009-10/exercises/hwk01/SIS_09-10_hwk01_assignment.pdf

Exercises -2

- *Exercise 1 : Call by reference vs call by value*

```
#include <stdio.h>
void square(int num) {
        num = num * num;
}
int main() {
        int x = 4;
        square(x);
        printf("%d\n", x);
        return 0;
}
```

What does this program Print?

Exercises -2 – answer

- *Exercise 1 : Call by reference vs call by value*

```
#include <stdio.h>
void square(int num) {
        num = num * num;
}
int main() {
        int x = 4;
        square(x);
        printf("%d\n", x);
        return 0;
}
```

This program prints 4 : why function variables are passed by value, not reference