Embedded C notes

- ***Advantages of C language :***
  - ✓ Extends to newer systems architecture
  - ✓ High efficiency and performance
  - ✓ Low-level access
- ***Disadvantages of C languages***
  - ✓ No exceptions
  - ✓ No range checking
  - ✓ No automatic garbage collection
  - ✓ No support for OOP
- ***What`s Embedded C language***

  A proper subset from C language suitable for embedded systems
- ***Common aims of embedded C techniques (ex : AUTOSAR , MISRA , …)***
  - ✓ Readability
  - ✓ Reliability
  - ✓ Maintainability
  - ✓ Testability
  - ✓ Portability
  - ✓ Extensibility
- ***Flow of code :***

  All embedded code is called within an infinite loop in main except (initialization code and ISRs)
- ***Multiple incursion :***
  - ✓ If (f.c) includes (h1.h and h2.h) and (h1.h) includes h2.h , this produces a development error
  - ✓ This is solved by

    ```
    #ifndef h2_h_
    #define h2_h_

    #endif
    ```
- ***Interrupts Vs. polling***
  - ✓ In interrupt method

1) Whenever any device needs the ECU service , the device notifies it by sending an interrupt signal
2) Upon receiving the interrupt signal , the ECU stops whatever it`s doing and serves the device

- ✓ **In polling method :**
  1) The ECU continuously monitors the status of a given device
  2) When the status condition is met , it performs the service
  3) After finishing the service , the ECU moves on to monitor the next device until eachone is served
- ✓ **The advantage of interrupts is that :**
  1) The ECU can serve many devices where each device is served according to the priority assigned to it
  2) The ECU can ignore (mask) a device request for a service
- ✓ **The disadvantage of polling method is that**
  1) It can't assign priority as it checks all devices in a round robin fashion
  2) It wastes much of the ECU time by polling devices that don't need services

➢ *Sources of Interrupts*

- ✓ Internal interrupts generated by on-chip peripherals such as timers EEPROM , serial and parallel ports
- ✓ External interrupts generated by peripherals connected to the ECU
- ✓ Exceptions thrown by the processors
- ✓ Software interrupts

➢ *Interrupt service routine (ISR) or interrupt handler :*

A program run by the ECU when an interrupt is invoked

➢ *Interrupt vector table :*

The group of memory locations set aside to hold the addresses of ISRs

➢ *Maskable Vs. non-Maskble interrupts*

- ✓ Maskable interrupts can be ignored by disabling them are giving them lower priority
- ✓ Non-Maskable interrupts can't be ignored like hardware failer , system reset , watchdog and memory parity failer

➢ *Interrupt nesting :*

The ability to leave the current interrupt and serve another interrupt

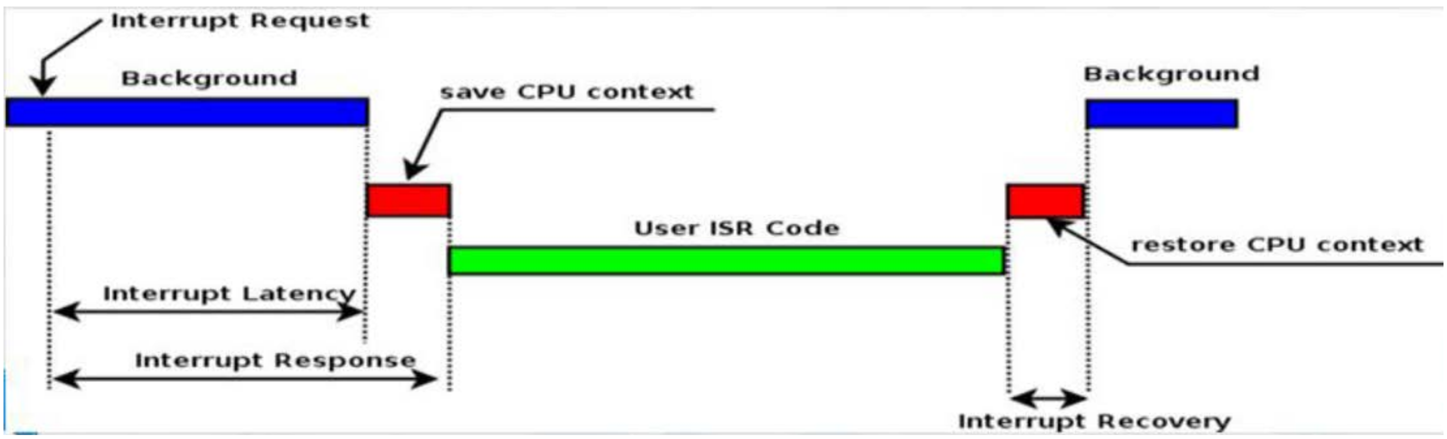
➢ *Interrupt servicing process :*

- ✓ The ECU finishes the instruction it's executing

- ✓ The ECU saves the address of the next instruction on the stack
- ✓ The ECU jumb to the address of the interrupt vector table in memory
- ✓ The ECU locates the ISR in the interrupt vector table and jumps to it
- ✓ The ECU start executing the the ISR until hitting a return
- ✓ The ECU restore the state information of the main program and resume executing from where it was interrupted

➢ *Interrupt timing :*

- ✓ Interrupts latency : the maximum amount of time interrupts are delayed before executing the 1st instruction in the ISR
- ✓ Interrupt response time : the interrupt latency + Time to save the CPU context before the interrupt
- ✓ Interrupt recovery time : time to restore the CPU context



➢ *Critical section of code (Critical region) :*

Code that once started executing , it mustn't be interrupted (interrupts are disabled before starting it)

➢ *Start-up code actions*

- ✓ Disable all interrupts
- ✓ Clock initialization and stabilization
- ✓ Allocate space for and initialize the stack
- ✓ Initialize the processor's stack pointer
- ✓ Initialize peripheral registers
- ✓ Call the main

➢ *Main 3 steps in C-build process*

- ✓ The preprocessor produces a (.i) file from a (.c) file
- ✓ The compiler produces a (.o) file from the (.i) file
- ✓ The linker produces executable file from different (.o) files

➢ *C preprocessor :*

- ✓ Reads and copies the (.c) file into a (.i) file
- ✓ Strips comments
- ✓ Makes (text-replacement) based on lines beginning with (#)
- ✓ In embedded systems , the preprocessor also process vendor-specific directives (non-ANSI) such as #pragma
- ✓ Keywords : #define , #include , #ifdef , #ifndef, #elif , #else , #endif , #pragma , …

➢ *#define :*

- ✓ Creates symbolic names (symbols) for expressions #define led 3
- ✓ Creates Macros

```
#define min(A,B) if( (A) < (B) ){ \
                                    return A; \
                            }\
else{ \
   return B; }
#define min(A,B) ( (A)<(B)? (A) : (B) )
```

- ✓ #define is a text replacement , so don't terminate it with a (;) to keep its symbols usability

➢ *#include*

- ✓ #include <header.h>
  Searches for (header.h) in the include paths
- ✓ #include "header.h"
  Searches for (header.h) in the include paths and the current directory of the project

➢ *Conditional preprocessors*

- ✓ They can control which lines are compiled (Conditional Compilation)
- ✓ Used in header files to ensure that declarations done (Header guards)
- ✓ Keywords (#if , #ifdef , #ifndef , #elif , #else , #endif)

➢ *Header Guards (inclusion guards)*

- ✓ Ensure that a headerfile is included only once in a c file
- ✓ Avoids multiple definitions and compilations deadlock

```
#ifndef  HEADER_FILE_H
#define HEADER_FILE _H
#include "header_file.h"
#endif
```

➢ *Preprocessor error Directive (#error)*

- ✓ Causes the preprocessor to generate an error message and the compilation to fail
- ✓ Example :
  #define BUFFER_SIZE 255
  #if BUFFER_SIZE < 256
  #error "Buffer size is too small"
  #endif

➢ *Bit-Manipulation :*

- ✓ Using bit-mask macros
  #define SET(port,mask) port|=mask
  #define SET(port,pin) port|=(1<<pin)
  #define CLEAR(port,mask) port&=~mask
  #define CLEAR(port, mask) port&=~(1<<pin)
- ✓ Using structs

```
/* Structure size is 2*sizeof(unsigned int) */
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;

/* Structure size is sizeof(unsigned int) */
struct
{
  unsigned int widthValidated  :2;
  unsigned int heightValidated :2;
} status;

/* Structure size is sizeof(unsigned int) */
struct
{
  widthValidated  :2;
  heightValidated :2;
} status;
```

```c
/* Structure size is sizeof(unsigned int) */
struct
{
  widthValidated  :2;
                  :2;
  heightValidated :2;
} status;

/* Structure size is 2*sizeof(unsigned int) */
struct
{
  widthValidated  :2;
                  :0;
  heightValidated :2;
} status;
```

- ***#pragma***
  - ✓ it tells the compiler to do something, set some option, take some action, override some default, etc. that may or may not apply to all machines and operating systems.
  - ✓ Pragma is implementation specific directive
  - ✓ There are many type of pragma directive and varies from one compiler to another compiler
  - ✓ If compiler does not recognize a particular pragma , it simply ignore that pragma statement without showing any error or warning message

- ***Compilation process :***
  - ✓ The compiler allocates memory for definitions and generates opcodes for executable statements
  - ✓ The compiler works with one translation unit (parsed C file) at a time
  - ✓ The compiler and assembler create relocatable object file

- ***compilation stages :***
  - ✓ Front end (source code parsing)
  - ✓ Middle end (optimiation)
  - ✓ Back end (code generation)

- ***Front end process (source code parsing)***
  - ✓ **Pre-processing**
    1) Evaluate pre-processing directives (#)
    2) Input : pre-processed translation unit (.c)

3) Output : post- processed translation unit (.i in code-blocks)

- ✓ **White-space removal**
  Stripping out all white spaces
- ✓ **Tokenizing**
  Identify tokens like keywords , operators ,identifiers , comments ,literals , ….
- ✓ **Syntax analysis**
  1) Ensures that tokens are organized according to C-rules
  2) Help to avoid compiler syntax errors
  3) Output : parse tree
- ✓ **Intermediate representation(optional)**
  1) Exists in a compiler supports multiple languages on different targets (gcc)
  2) Transforms the parse tree into abstract syntax tree (AST or pseudo code) which is a machine independent representation

➢ *Middle end process (optimization)*

- ✓ **Semantic analysis**
  1) Adding info to the AST
  2) Checks logical structure of the program
  3) Problems found here are warnings (not error)
  4) Program symbol table is constructed and debug information is inserted
- ✓ **Optimization**
  1) Transforms code into smaller or faster one but functionally-equivalent
  2) This multi-level process includes :
     Inline expansion of functions
     Dead code removal
     Loop unrolling
     Register allocation

➢ *Code generation :*

Converts the intermediate representation code structure into target opcodes

➢ *Memory allocation :*

- ✓ **The compiler allocates memory for code and data in sections**
- ✓ **These sections are defined by name or attributes of info stored in them as follows :**
  1) Code is stored in (.text) section
  2) Globally declared variables without initialization are stored in (.bss) section
  3) Globally declared variables with initialization are stored in (.data) section
  4) Constants are stored in (.rodata) section
  5) Automatic (local) variables are stored in (.stack) section or general registers (R#)

6) Dynamic data are stored in (.heap) section

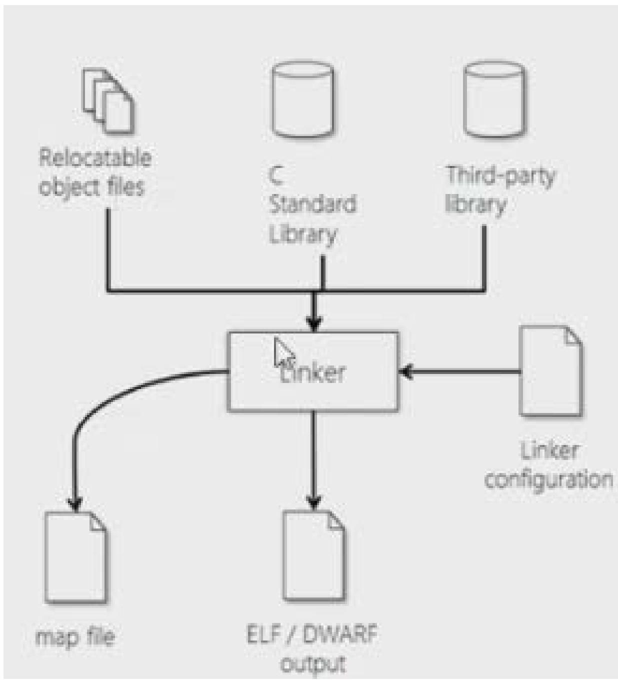
✓ **Attributes are used by linker for locating sections in memory**

➢ *Linking Process*

✓ Combining object files into a single executable file

✓ Stages :

1) Symbol resolution
2) Section Concatenation
3) Section location
4) Data initialization



➢ *Symbol Resolution :*

✓ Resolve References between object files

✓ Search for unresolved symbols in libraries to resolve them

✓ No resolution = unresolved symbol error

✓ If the linker finds the same symbol defined in two object files , it will report a "redefinition" error

➢ *Section concatenation :*

✓ Concatenating like-named sections from the input object files

✓ Program addresses are adjusted to take account of concatenation

➢ *Section location*

✓ Each section is given an absolute address in memory

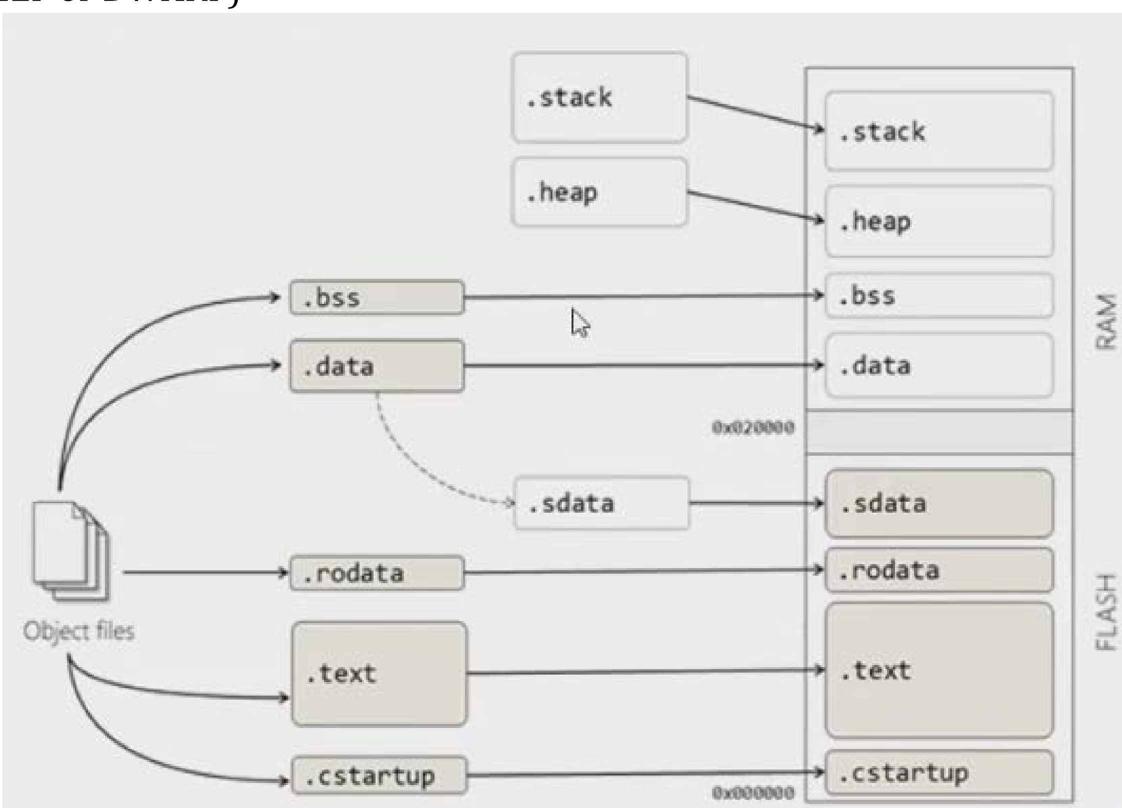
✓ There's a base address in non-volatile memory for persistent sections and there's another base address in volatile memory for non-persistent sections

➢ *Data initialization*

- ✓ Any initialized data is stored in the non-volatile memory
- ✓ Linker must create extra sections to enable copying data from ROM to RAM to speed up execution
- ✓ Each initialized section by copying is divided into a section in ROM (shadow section) and another in RAM
- ✓ (.bss) section has no shadow section . It's initialized by startup code
- ✓ If manual initialization isn't used , the linker arranges for the startup code to perform initialization

➢ *Linker control :*

- ✓ The detailed operation of the linker can be controlled by invocation options (command-line) or linker control file (linker script , linker configuration file or scatter-loading)
- ✓ LCF(linker control file) defines physical memory layout and placement of different memory regions
- ✓ LCF syntax is compiler-dependent
- ✓ When an IDE is used , linking options can be relatively friendly specified
- ✓ The output of linking stage is a loadable file in a platform-independent format (.ELF or DWARF)



➢ *Loading process :*

- ✓ ELF or DWARF are target-independent format
- ✓ The ELF must be converted into a native (Flash or PROM) format (.bin or .hex) to be loaded into the target

➢ *Data structure alignments and padding*

- ✓ Data alignment : means putting the data at a memory address equal to some multiple of the word size.
- ✓ Data structure padding: means inserting some meaningless bytes between the end of the last data structure and the start of the next to align the data
- ✓ For an array of structures , padding is only inserted when a structure member is followed by a member with a larger alignment requirement or at the end of the structure.
- ✓ Disadvantage :
  1) If the highest and lowest bytes in a datum are not within the same memory word the , computer must split the datum access into multiple memory accesses.
  2) This requires a lot of complex circuitry to generate the memory accesses and coordinate them.
  3) When defining a C-struct , it may take a larger size in memory due to padding

➢ *Memory alignment :*

- ✓ A memory address , is said to be n-byte aligned when it's a multiple of n bytes
- ✓ A memory access is said to be aligned when the datum being accessed is n bytes long and the datum address is n-byte aligned
- ✓ When a memory access is not aligned, it is said to be misaligned.
- ✓ A memory pointer that refers to primitive data that is n bytes long is said to be aligned if it is only allowed to contain addresses that are n-byte aligned, otherwise it is said to be unaligned.

➢ *Data types :*

- C has a small family of data types.
  - Numeric (int, float, double)
  - Character (char)
  - User defined (struct, union, arrays ...)
- Numeric data types

	signed	unsigned
short	short int x; short y;	unsigned short int x; unsigned short y;
default	int x;	unsigned int x;
long	long x;	unsigned long x;
float	float x;	NA
double	double x;	NA
char	char x; signed char x;	unsigned char x;

➢ *Variables sizes and endianness*

- Sizes are machine/compiler dependent.
  - sizeof(char) < sizeof(short) <= sizeof(int) <= sizeof(long)
  - sizeof(char) < sizeof(short) <= sizeof(float) <= sizeof(double)
- For datatypes spanning multiple bytes, the order of arrangement of the individual bytes is important.
  - Big endian vs. little endian

- Big-endian:
  - Store most significant byte in low address and least significant byte in high address (Freescale)
- Little-endian:
  - Store least significant byte in low address and most significant byte in high address (Intel)
- Bi-endian:
  - Can be configured to efficiently handle both big and little endian (PowerPC/ARM).

Variable scopes in C :

- Global
- File
- Function (local)
- Block

C keywords for variables attributes :

- ***Auto :***
  1) A local variable defined inside a function
  2) The are created when the variable is called and killed when the function finishes its execution
  3) Their initial values are unknown if not initialized
- ***Register :***
  1) A variable created directly in one of the processor's general purpose registers instead of memory
  2) This minimize the over head of loading / storing variables between the memory and the processor's register
  3) Register variables must be of a simple type and local or function arguments
  4) Excess/unallowed register declarations are ignored
  5) We can't make a pointer of a "Register" variable as it doesn't reside in addressed memory
- ***Static :***
  1) Limiting scope usage : Outside a function , static variables / functions are only visible with that file (not global variables / functions)
  2) Local variable persistence usage : Inside a function , the local static variable is initialized only during the program initialization and remain without needing initialization with each function call
- ***extern :***

  used to locally declare a variable (globally declared in another file)
- ***const :***
  1) Const variables are initialized once and can't be changed
  2) Const variables are stored in the flash memory
- ***volatile :***
  1) A variable that can change expectedly , so we prevent compiler optimization
  2) The optimizer must reload the variable every time it's used instead of holding a copy in a register

Examples of volatile variables :

- Hardware register in peripherals (Ex : status registers)
- Variable referenced with an ISR
- Shared variable in multi-tasking or multi-threaded applications

- ***Can a parameter be constant and volatile ? illustrate with an example.***
  - ✓ Yes
  - ✓ Example :
    1) Read-only status register
    2) It's volatile as it can be changed unexpectedly
    3) It's constant because the program shouldn't attempt to modify it

- ***Can a pointer be volatile ? explain.***
  - ✓ Yes although it's not common
  - ✓ Example :
    When an interrupt service routine modifies a pointer to a buffer

- ***Advantages of function concept***
  - ✓ Modularity
  - ✓ Reusability
  - ✓ Readability
  - ✓ Maintainability

- ***Function parameters Vs. function arguments***
  - ✓ Function parameters are the inputs defined while prototyping or defining a function
  - ✓ Function arguments are inputs passed to a function while calling (invoking) it

- ***Main Function :***
  - ✓ The entry point for a c program so it`s called once
  - ✓ It's called by an OS or the start-up code
  - ✓ In an embedded system we can enforce starting from another entry point rather than the Main function by configuring the start-up code
  - ✓ Starting from another function helps to avoid some optimizing codes added by the compiler when using the default "Main function"

- ***Macros Vs. Functions***
  - ✓ ***Macros***:
    1) Increase the code size as they are text replacement
    2) Less execution time as there's no jump and return
  - ✓ ***Functions :***
    1) Decrease the code size as its code is written once
    2) Increase execution time as it depends on jump and return technique

➢ *inline Function:*

- ✓ Defined by using the keyword “inline”
- ✓ Tells the compiler to substitute the body of the function code at the address of each function call
- ✓ Saves the overhead of invocation and return

➢ *Inline Function Vs. Macro :*

- ✓ ***Inline function :***
  1) Make suggestion for text replacement that can be ignore by the compiler for technical considerations
  2) allow type checking by the compiler
  3) Concerned with compile time
- ✓ ***Macro :***
  1) enforces text replacement
  2) doesn’t allow type checking by the compiler
  3) concerned with preprocessing time

➢ *Synchronous Vs Asynchronous functions :*

- ✓ **Synchronous function** : returns the control to the caller after finishing its task
- ✓ **Asynchronous function** : returns the controller to the caller after starting its task while continuing its task as a back ground process

➢ *Reentrant functions :*

- ✓ Allow different concurrent invocations from different context
- ✓ Examples : Functions shared between different tasks in a multi-tasking system

➢ *What doesn’t have addresses in C (we can’t make a pointer to it) ?*

- ✓ Register variables
- ✓ Expressions unless the result is a variable
- ✓ Constants , literals and preprocessors

➢ *Pointers arithmetic :*

- ✓ *pn++ : fetches what (pn) points to , then increments the pointer (pn)
- ✓ *++pn : increments the pointer (pn) , then fetches wht it’s pointing to
- ✓ *(arr+7) : means accessing the 7th element in an array using a pointer notion
- ✓ 7[arr] is the same as *(arr+7)

✓ *Pointers Casting*

- ✓ Casting explicitly is needed when moving data among pointers of different types
  int *p;

Float *pf =(float*) p;

- ✓ Casting implicitly is done while moving to and from a void pointer

➢ *Special cases of pointers :*

- ✓ Void pointer :
  1) A pointer that can point to anything of any type
  2) It can't be dereferenced until it's casted to a known type of pointer
  3) It's treated arithmetically the same as a char pointer
- ✓ Null pointer : a pointer (whether it's void or of a specific type) dereferences nothing (stores value of 0)

➢ *Pointer to pointer*

- ✓ A (pointer to pointer) addresses a location of an address in memory

int n=4;
int *pn =&n;
int **ppn=&pn;

- ✓ It's needed for
  1) Pointer array
     int *arr[20];
  2) Multi-dimensional array
     int world[20][30];
     *(world +5)[4] is the same as *(*(world +5)+4)
     int **p=world //this is illegal
  3) String array

➢ *Pointers and functions :*

- ✓ **Function returning a pointer**

char *getMessage();

- ✓ **A pointer to function**
  1) A function name (without "()") is a reference to its address
  2) a pointer to a function takes a pointer to char and returns an integer :
     char *line;
     int (*p) (char*) ;
     int L =p(line);
  3) array of pointers to functions
     int add(char*);
     int (*p)[4] (char*);
     p[0] = add;

➢ *Dynamic Memory Allocation :*

- ✓ To allocate memory , we use this standard function void *malloc(size_t size)

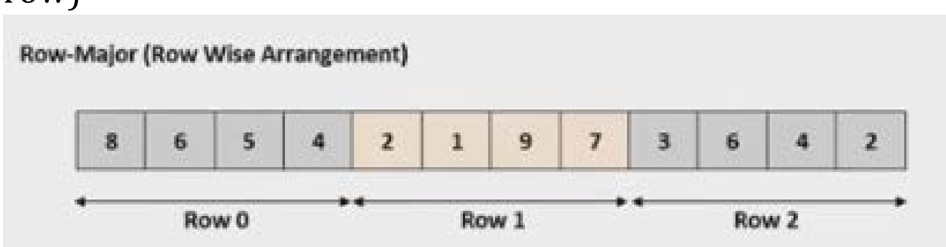
- ✓ To free memory locations , we use this standard function
  void free(void *ptr)
- ✓ Disadvantages :
  1) Memory leakage as a result of not releasing allocated memories
  2) Memory fragmentation

➢ *Typedef*

- ✓ A facility to allow creating a new name for an existing data type
  typedef char * string ; //string is a new name for char *
  string x ; // x is a variable of type char *
- ✓ Helps to improve the readability of the program
- ✓ "typedef"s are usually placed in header files

➢ *Arrays :*

- ✓ A constant pointer to a block of contiguous data in memory
- ✓ C has no range checking for arrays , so we can mistakenly access a something out of the array bounds without a compilation error
- ✓ A macro calculating the length of an array
  #define ArrSize(A) (sizeof(A)==0 ? 0: sizeof(A)/sizeof(A[0]))
- ✓ A 2-D array is represented in memory in a row-major way (arranged as row by row)



- ✓ A function that takes an argument of 2-D arrays
  int total(int arr[ ][cols], int rows);

➢ *Unions :*

- ✓ Holds objects of different types in the same memory location
- ✓ Union size is equal to the size of its largest element
- ✓ A non-homogenous array is an array of unions that can hold elements of different types

➢ *Enumeration (enum)*

- ✓ A set of integers referenced symbolically
- ✓ If an enum has a starting value , each symbol in turn represents the next int
  enum days{sun=2, mon , tue , wed , thu , fri , sat=1 };
  enum days today , yesterday , tomorrow;