Embedded Systems. Thursday 17/5/2018

C Programming

* Introduction :-

- Program ⇒ data & instructions.

→ Data provided through a data structure.

→ Instructions provided through algorithms ⇒ commands that perform a specific function.

- Levels of coding :-

→ Machine language ⇒ 0's & 1's.

→ Low level language ⇒ assembly.

→ High level language ⇒ C, C++, Java.

→ Very high level language ⇒ PLAN

- Interpreter vs Compiler :-

→ Interpreter ⇒ checks every line of code & executes. If an error is detected execution stops.

→ Compiler ⇒ translates the program into an exe file containing system calls. (OS dependent)

- Types of programming :-

→ Linear programming ⇒ sequential execution.

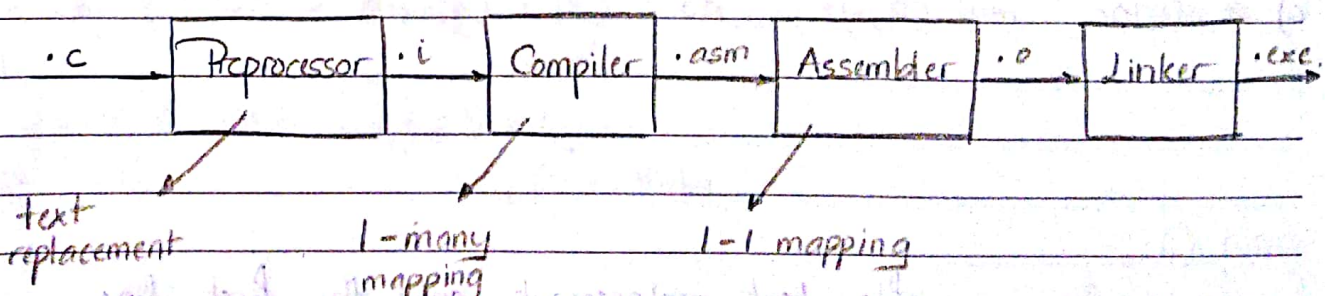
→ Structured programming ⇒ main program calls other sub programs.

→ Object oriented programming

→ Agent oriented programming.

* Build Process of a C Program :-

-



* Preprocessing :-

- Preprocessing ⇒ first part of the program executed before compilation starts.
- Preprocessor Directives :-

⟶ # include ⇒ include libraries.

- # include < > ⇒ compiler directory / standard libraries.
- # include " " ⇒ file directory / programmer made libraries.

⟶ # define ⇒ text replacement

- Used to define macros.
- Macros → object like macros ⇒ # define x 10
  Macros → function like macros.
- Function like macros :-
  - Advantages ⇒ saves execution time.
  - Disadvantages ⇒ increases code size.
  - Best usage ⇒ functions with few lines.
  - Adding a semicolon at the end of a function like macro ⇒ No compilation error but handled as an empty line of code.
  - Examples :-

a) # define sum(x,y) (x) + (y)

b) # define complex (x,y)
```
x= 2y; \
y= 2x; \
x= 6y; \
x+y = 3
```

c) # define mul (x,y)
```
do { (x) + (y) ; \
     (x)/2 ;
     (y) * 3 ; }
while (0) ;
```

```
main ()
{ int x= 3;
  if (x== 10)
    mul(x,y)
}
```

after text replacement only the first line will be in the if ⇒ use do { } while(0); to ensure at least 1 execution.

- Function like macros vs. functions:-

a) Functions perform datatype checking but function like macros don't.

b) Function like macros are handled in preprocessing while functions are handled in compilation

→ #if #elif #else #endif ⇒ precompilation configuration. Only the required code will be compiled & the other parts of code will be deleted.

→ #ifdef #ifndef

→ #error ⇒ stops compilation.

→ #warning

→ #pragma ⇒ adjust compiler configuration

- Compiler directive or preprocessor directive.

- Functions [optimization = ON].

a) May remove dead code (condition will never be satisfied)

b) If 2 loops repeat the same number of times & are independent on each other ⇒ place them in the same loop.

c) Variable caching ⇒ place frequently used variables in cache mem. near ALU

→ Example:-

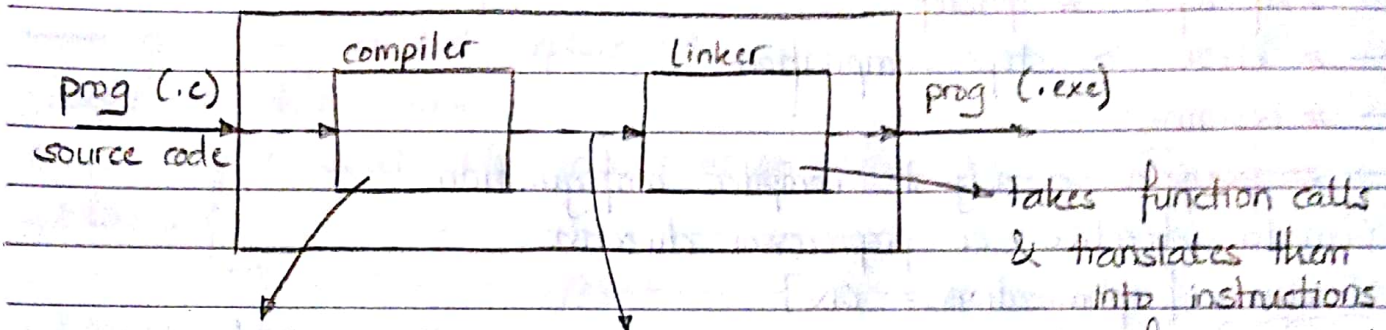
```
#define CAR BMW
#define BMW 0
#define Volvo 1
#if (CAR == BMW)
    printf ("CAR is BMW")
#elif (CAR == Volvo)
    printf ("CAR is Volvo")
#else
    printf ("CAR is N/A")
#endif
```

```
#ifndef CAR
#error "CAR not defined"
#endif
```

```
#warning "CAR not defined"
```

* Compiler

- Compilers check syntax & symantics of the program & generate an assembly code by mapping every program instruction to many assembly instructions.
- Cross compiler ⇒ the target of the program is different than the host on which the programming takes place.



- Linker ⇒ translates the virtual addresses created by the compiler into physical addresses.
- Static linking ⇒ all instructions end up in one executable file.

* Datatypes:-

- Primitive Datatypes:-
  - → char (unsigned 1 byte)
  - → int (signed 2 bytes)
  - → float (signed 4 bytes)
  - → double (signed 8 bytes)
  - → void.

- Derived Datatypes:-

→ Arrays:-

• Arrays represent a group of data of the same datatype that have a logical relation between them.

• Characteristics of an array:-

a) Fixed size.
b) Homogenous.
c) Ordered.
d) Indexed.
e) Contigous ⇒ placed in memory after each other
f) Random access / direct access.

• Maximum number of elements an array can have is 2^{16} (the maximum unsigned int that can be stored in memory).
The size of every element depends on the datatype.

• Example:-

```
int arr [10];
```

↳ constant pointer to int not allocated in memory.

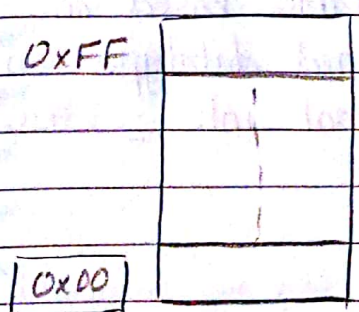
arr & &arr both return the address of the first element

*arr ⇒ arr[0]

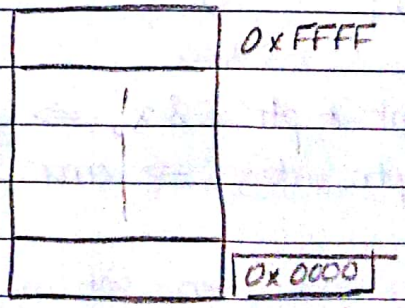
→ Pointers:-

• Pointers are variable that contain the address of other variables.

• The size of pointer is independent on the datatype to which it points; it depends on the size of the memory of the controller.



size of ptr = 1 byte.



size of ptr = 2 bytes

• The main function of pointers is to reference other variables.

• Examples :-

```
int y = &x          int * ptr = &x
*y = 3                *ptr = 3
⇒ Not allowed       ⇒ x = 3
```

```
char x;                          0000  0011
int * ptr = &x;      ⇒           0000  0000
*ptr = 3;
```

⇒ extra place in memory is overridden

```
int x = 256;
char * ptr = &x;     ⇒           0000  0001
*ptr = 1;                        0000  0001
```

⇒ ptr points only to the first byte

• Call by address vs. call by reference :-

func (int * ptr) ⇒ call by address.

func (int & ptr) ⇒ call by reference [Not available in C]

• Constant Hacking :-

```
const int x = 5;
int * ptr = &x;      } ⇒ x = 3
  *ptr = 3;
```

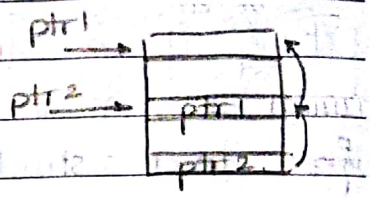
Pointers are dummy variables that deal with other variables based on the assigned datatypes.

but `const int * ptr = &x;` ⇒ pointer to constant int

`*ptr = 5;` ⇒ error.

- Constant pointer :-

```
int x;
int y;
int * const ptr = &x; ⇒ constant pointer to int.
*ptr = 5;
ptr = &y; ⟶ error
```



- Wild pointer ⇒ uninitialized pointer.
- Pointer to pointer ⇒ int * * ptr2 = & ptr (access to 2 memory locations)

⟶ Functions

- User Defined Datatypes :-

⟶ typedef ⇒ defines a new datatype.

- Example :-

typedef unsigned char uint8_t;

- typedef vs. # define :-

# define int_ptr int *	typedef int * int_ptr;
int_ptr x, y;	int_ptr x, y;
⇒ due to text replacement only x will be a pointer to int	⇒ both x & y will be pointers to int.

⟶ structures :-

- Blueprint :-

```
struct Name
{
    uint8 x;
    uint8 y;
};
```

```
typedef struct
{
    uint8 x;
    uint8 y;
} Name;
```

struct name.

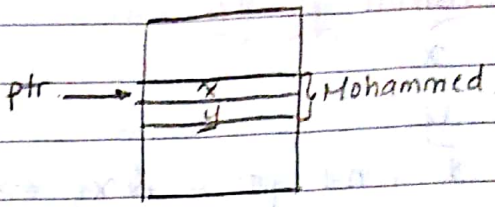
```
struct ⟶ Nameless struct
{
    uint8 x;
    uint8 y;
} Name;
```

only ⟶ object from struct.

The blueprint is not saved neither in the code memory (flash) nor in the data memory (RAM). It is created for the compiler like int, float;

• Pointer to structure :-

```
struct student Mohammed;
struct student * ptr = & Mohammed;
```



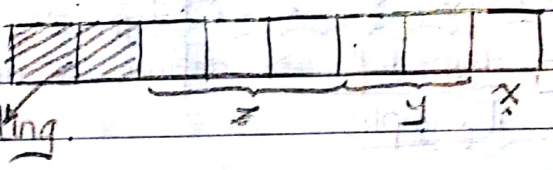
• Bit Field Struct :-

Use bit field structures to save memory and assign variables to certain bits in memory.

⇒

```
struct structure
{
    uint8 x : 1 ⇒ 1 bit
    uint8 y : 2 ⇒ 2 bits
    uint8 z : 3 ⇒ 3 bits
}
```

total size = 8 bits



⇒ Data Alignment :-

Every datatype in C/C++ will have alignment requirements based on processor architecture to ensure that a variable is read in minimum number of cycles.

⇒ Structure Padding & Packing :-

Padding ⇒ adding extra bits to ensure data alignment

Example :-

```
struct Base
{
    char a;
    char b;
    int i;
}
```

assume 32 bit processor with padding

⇒ size = 8 bytes

|a|b|-|-|i|i|i|i|

padding | 1 word

```
struct Base
{
    char a;
    int i;
    char b;
}
```

⇒ size = 12 bytes

|a|-|-|-|i|i|i|i|b|-|-|-|

Note :-

8 bit ⇒ 1 word = 1 byte

16 bit ⇒ 1 word = 2 bytes

32 bit ⇒ 1 word = 4 bytes

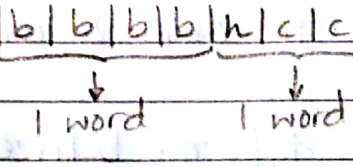
64 bit ⇒ 1 word = 8 bytes

↓ processor

For a 32 bit processor to be able to read an integer in the least number of memory cycles it needs to be stored in multiples of 4 & therefore padding is required.

Packing ⇒ compiler configuration that disable padding to save memory but increases the number of cycles required to read a variable.

```
# pragma pack()
struct Base
{
    int b;
    char h;
    int c;
    char d;
};
```

⇒ 

⇒ size = 10 bytes

In this case, the processor will read the word & extract the variable ⇒ more processing.

→ union :-

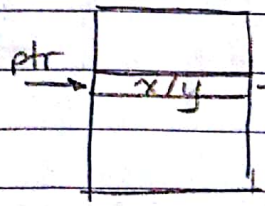
```
union Name
{
  uint8 x;
  uint8 y;
};
```

```
typedef union
{
  uint8 x;
  uint8 y;
} Name;
```

The only difference between struct & union is that union saves place in memory the size of the larger element

```
⇒ union Name Moh;
  union Name * ptr = & Moh;
  ptr -> x = 3;
  printf ( "%d", ptr -> y );
```

⇒ y = 3.



→ enum :-

- Use enum to define the range of values allowed

⇒ enum week.

{

sun, → 0

mon, → 1

tue, → 2

wed, → 3

thu, → 4

fri, → 5

sat → 6

};

Main advantage :-

Readability.

⇒ the objects from type week can only take values from 0 → 6 but no error if the don't

⇒ enum week Myweek;

size = size of int.

⇒ sun → sat are not elements in the enum & can be used on their own

⇒ difference between enum & # define is that enum is not preprocessor

⇒ No text replacement

⇒ Boolean datatype with enum

typedef enum

{

False,

True

} bool;

⇒ bool x;

x = True;

x = 5; ✓ → No compilation error.

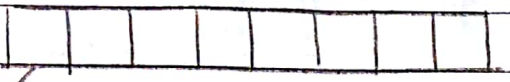
⇒ int y = False ✓

* Modifiers :-

- Modifiers / qualifiers are used to change the primitive datatypes.
- Sign Modifiers

→ signed, unsigned.

→



sign bit

0 ⇒ +ve

1 ⇒ -ve

⇒ signed

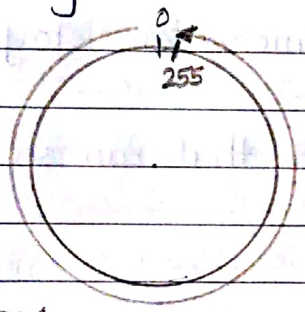
$-2^{n-1} \rightarrow 2^{n-1}-1$

⇒ unsigned

$0 \rightarrow 2^{n}-1$

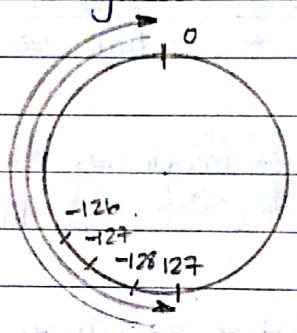
→ Cyclic property :-

⇒ unsigned.



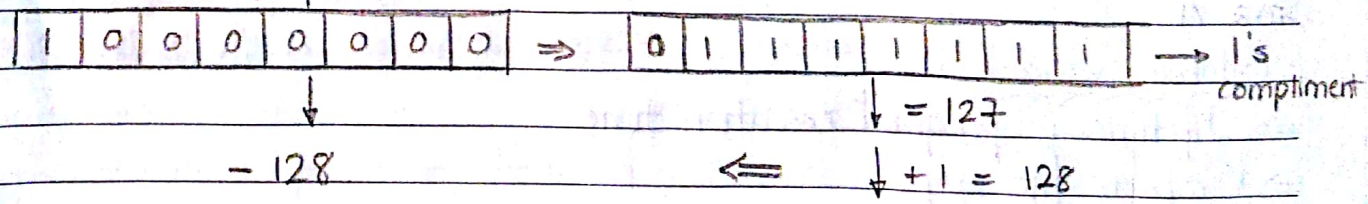
⇒ 256 = zeros

⇒ signed.



→ 2' compliment :-

- Negative numbers are represented using 2's compliment.



- Size Modifiers:-

→ short , long

→ short char ;
  long char ; } ⇒ error.

- Constant Modifier:-

→ const

→ Remember constant hacking.

- Storage Modifiers:-

→ Local vs. Global Variables:-

properties of any variable { lifetime ⇒ the amount of time a var stays in memory.
scope ⇒ the part of the code that can see the var.

• Local variables:-

⇒ Lifetime:- function execution time.

⇒ Scope:- function scope.

⇒ Stored in stack but static local vars. stored in heap.

⇒ Initial values = garbage but static local vars. initialized by zero.

• Global variables:-

⇒ Lifetime:- program execution time.

⇒ Scope:- file scope.

⇒ Stored in heap.

⇒ Initial value = zero.

→ static, extern, register, volatile, inline

→

	Static		Extern	
	Scope	Lifetime	Scope	Lifetime
Local variables		increase lifetime		
Global variables	decreases scope		increase scope	
Functions	static func. cannot be used externally		func. are by default extern	

- Static modifier changes the lifetime of the local variable from function execution time to program execution time.
- Static global variables cannot be externed.
- You cannot use extern with local variables.
- All functions are by default extern (in source)
- Extern increases global variables scope (in destination)

→ register & inline are not usually used in embedded systems because they're unpredictable.

- Use register to store a declared variable in the CPU register instead of in RAM.
- Use inline for functions to eliminate the call/return sequence for faster execution.

→ Use volatile modifier to prevent variables from being cached when it is frequently used by h/w ex. interrupts.

* Operators :-

- Arithmetic operators :-

→ Binary :- needs 2 operands ⇒ +, -, *, /, %

→ Unary :- needs 1 operand ⇒ ++, --, !, ( ) → type casting

Not

- Prefix vs. Postfix :-

a) pre-increment has the highest priority.

b) post-increment has the lowest priority.

Examples :-

1- y = x++ + ++x + x++ ; x = 5;
     6   6   6

⇒ y = 18 , x = 8

2- y = x++ + ++x + ++x ; x = 5;
     7   6   7

⇒ y = 20 , x = 8

- Bitwise operators :- &, |, ~, <<, >>, ^ → toggle.

~ : 1's comp

<< , >> : linear shift

- Assignment operators :- +=, -=, *=, /=, %=, &=, |=, ^=, ~=, <<=, >>=, = (assignment operators have the lowest priority before postfix)

- Relational operators :- <, >, <=, >=, ==, !=

- Logical operators :- (short circuit operators) &&, ||

- Terniary operator :- ?:

- Operators on Pointers :-

→ ptr ++; ⇒ increase by 1 unit depending on datatype.

→ * ptr ++; ⇒ increment has higher priority i.e increment pointer then access new address. (⇔ * (ptr ++)).

→ (* ptr) ++; ⇒ increment value in current address.