➡️ C programming

- 👉 Introduction
- 👉 Program structure and first program
- 👉 Escape Sequence
- 👉 Tokens
- 👉 Naming Rules
- 👉 Data types
- 👉 Data Specifiers or Formats
- 👉 Const Qualifier
- 👉 Error Types
- 👉 Literals (integer and floating point literals)
- 👉 Operators and bit manipulation
- 👉 Integer promotion
- 👉 logical and arithmetic shift
- 👉 Conditional operator
- 👉 Different statements (if-switch-for-while)
- 👉 Drawing Some different patterns
- 👉 Functions
- 👉 Casting and its types
- 👉 Memory layout introduction
- 👉 Storage classes
- 👉 Build process overview
- 👉 Macros
- 👉 Preprocessor Conditional Compilation Directives
- 👉 Preprocessor Directive Operators
- 👉 Arrays and Arrays of characters
- 👉 Two Dimensional Array
- 👉 Passing array as a character
- 👉 Pointers and its different types
- 👉 Pointer Arithmetic
- 👉 Pointer with const keyword
- 👉 Pointers with strings
- 👉 Double pointer concept
- 👉 Return address / array from function
- 👉 Pointer with Arrays and Array of characters
- 👉 2d arrays of characters
- 👉 Recursion
- 👉 implementation of most function of string.h library
- 👉 Function pointer concept and using it as an argument
- 👉 Binding concept
- 👉 Modules software design approach and call back function
- 👉 Structure in depth
- 👉 Typedef, Union, Enum
- 👉 Dynamic Memory Allocation
- 👉 Dealing with Files
- 👉 Inline function
- 👉 Data Structure
- 👉 Stack and dynamic stack
- 👉 Queue and Circular Queue
- 👉 Linked list and implementation of its. functions

C- Programming

*Introduction to C-Programming :-

- C is a general-purpose, high-level language
- structured language (structured modular programming language)
  - because while solving large and complex program, C-programming divides the problem into smaller modules called function, each of these functions has specific job, and entire problem is solved by collecting such functions or smaller modules.
- it produces efficient programs (Assembly language ~ تكتبها قريبة)
- it can handle low-level activities (hardware → memory يمكنها التعامل مع)

*steps to write C-program :-

1- create a program 2- compile program 3- run program

* First program in C- language :

```
#include <stdio.h>
int main ( ) {
    printf ("hello world");
    return 0;
}
```

1 lines begining with # are processed by the pre-processor before compilation and the first line tells the pre-processor to include the standard input/output header file in the program this file contains standard functions as printF to print something

2 function main begins program execution
- every program in C begins executing at the function main

3 any instruction in C must end by (;) → semicolon

* Comment :-

1 one line comment

```
// Comment
```

2 multi lines comment

```
/* Comment
   Comment.. */
```

* comments are like helping text in C-program and they are ignored by the compiler.

* program structure

1- Documentation section → (معلومات حول الملف والكاتب)
2- Link section → (header & user ملفات إضافية)
3- Definition section → (#define PI 3.14)
4- Global declaration section → (int var ;) & function declaration
5- main program section → int main ( ) { }
6- subprogram section → function definition

* Escape sequences :-

- Is two or more characters that often begin with an escape character that tell the computer to perform a function or command

→ (\n) : newline position the cursor at the begining of the next line
→ (\t) : Horizontal tab
→ (\a) : produces a sound (Alert)
→ (\\) : Insert a backslash character in a string
→ (\") : In sert a double-quote // // // //
→ (\v) : vertical tab
→ (\b) : backspace
→ (\ooo) : octal number
→ (\xhh) : hexadecimal number
→ (\0) : Null

ex:

printf (" 1 2 3 4 "); → output : 1 2 3 4

printf (" 1 2 \n 3 4"); → out put : 1 2 / 3 4

printf (" 1 \t 2 \\\ " 3 4 "); → // : 1 2 \ " 3 4

printf (" \\\ " ahmed \ " \\ "); → // : \ "ahmed " \

* Tokens :

- Key word (int - static - const ····)
- Constant (constant Values 1, 2, 3··)
- identifiers (variables names)
- ~~String~~ (symbols) ((), [], "", { }, ;)
- operators (== , $, | , <, > ·····)
- string (array of character)

تقوم عملية Tokenizing عن طريق أداة Tokenizer بتقسيم الأوامر Combiler لمجموعة من Tokens والتأكد من أنها تنتمي لإحدى Tokens وإلا سينتج error (Compilation error)

* قواعد تسمية المتغيرات Identifiers

١- يجب ألا يبدأ برقم
٢- إتاحة استخدام (_)
٣- استخدام A-Z a-z
٤- لا يحتوى على أى فراغات
٥- لا يحتوى على علامات خاصة مثل ($, @) ماعدا (_)
٦- ألا يكون كلمة من الكلمات المحجوزة للغة Keywords (long - else - auto...)

* Data_Types :-

[1] Basic Types
- a) integer types (char - int - short - long) → signed or unsigned
- b) floating-point types (Float - double - long double)

[2] Enumerated types :- they are used to define variables that can only assign certain discrete integer values throughout the program

[3] Derived types :-
- Pointer type
- Structure
- Function
- Array
- union

[4] the type Void : Void indicates that no value is available.

Type	storage size	value range
char	1byte = 8 bit	$2^8 = 256 \rightarrow -128 : 127$
unsigned char	1byte = 8 bit	$2^8 = 256 \rightarrow 0 : 255$

* To get the exact size of a type or a variable on a particular platform "compiler" we can use the sizeof operator and we should include <limits.h> header file

ex:-

```
printf ("%d", sizeof (char));
```
⟹ output : 1 → 1 byte

* formats or data specifiers :-

%s : format of string
%i : // // integer
%d : // // decimal
%F : // // float
%c : // // char
%o : // // octal
%x : // // hexadecimal

%u : format of unsigned integer = decimal
%p : // // (point of address)
%d : // // int & signed int
%hd : // // short int & signed short int
%ld : // // long // & // long //
%lld : // // LL // & // LL //
%hu %lu %llu حل ما سبق

%lf → double
%LF → Long //

```
int Num_1 = 5;
int Num_2 = 9;
Printf (" Num 1 = %i \n", Num-1 );
Printf ( " Num1 + Num2 = %i \n", (Num-1 + Num_2));
```

Output:-

Num 1 = 5
Num 1 + Num2 = 14

```
Float test = 3.1456789 ;
Printf ("test = %f \n ", test);
Printf ("test = %0.2f \n", test);
```

Output: ← تحدد عدد الأرقام بعد العلامة العشرية

test = 3.1456789
test = 3.14

Test = 'a'
%i ⟶ 97
97: is the ascii code of char 'a'
Test = 98
%c ⟶ b
→ the ascii code of 'b' is 98

---

* Difference between declaring a variable and defining a variable :-

[1] Declaration: hints the compiler about the type, name and size of the variable in compile time, and no space is reserved in memory for any variable in case of declaration

[2] Definition : means declaring a variable and also Allocating space to hold it

Definition = Declaration + space reservation

```
int var_1 ;  ⟶ declaration
int var_2 = 10; ⟶ definition
```

* any global uninitialized variable will has initial value equal to zero
* any local uninitialized variable will has initial value equal to = unknown (garbage value).

---

* Const Qualifier in C: the qualifier const can be applied to the declaration of any variable to specify that its value will not be changed, we can change the value of of const variable by using pointer if the variable is local variable, if it was a global variable we cannot change it because it was be Read only variable

*if we tried to change directly the global const variable, we will hava a compilation error

→ syntax: const data-type var-name;

---

*Error types:

*programming errors are also known as the bugs or faults, and the process of removing these bugs is known as debugging

1 syntax errors oR compilation errors:-

- they occurred at the compilation time (thrown by the compiler)
- these errors are mainly occurred due to the mistakes while typing or don't follow the syntax of the Language
- these errors can be easily corrected

2 Run-time errors:-

- these errors occurred during the execution time even after the successful compilation
- the main cause of these errors is not able to perform the operation when the program is running as division by zero
- these error are very difficult to find, as the compiler does not point to these errors

3 Linker errors:-

- these errors are mainly generated when the executable file (.exe) of the program is not created
- this can be happend due to the wrong function prototyping or usage of the wrong header file or using a variable declared without any definition
- words: undefined reference oR ld

4 logical errors:.

- these errors leads to an undesired output
- these errors are error-free but produce the incorrect output

```
int num=0;
for(num=0;num<5;num++);
  {
    printf("number=%i\n", num);
  }
```

→ {the cause of an undesired output:

output: → number = 6

5 semantic errors:

- these errors occurred when the statements are not understandable by the combiler

→ as using of an uninitialized variable

```
unsigned number;
number = number+2;
```

→ Errors in expressions

```
unsigned int num1, num2, num3;
num1 + num2 = num3;
```

→ Array index out of bound

```
unsigned int arr[10];
arr[50] = 55;
```

* لايوجد في لغة الـ boundary check
* في لغة الـ لا تعتبر مشكلة وكنها تعتبر مشكلة طالبنا

→ Type compatibility

```
unsigned int num = "mohammed";
```

---

* Literals :- 1 Integer Literals:-

- prefix : specifies the base or radix → تحديد النظام العددي

0X	:	for hexadecimal	→ 0X 4b
0	:	for octal	→ 075
nothing	:	for decimal	→ 85
0b	:	for binary	→ 0b 1001 1010

- suffix : • Combination of U and L For unsigned and Long respectively
  • the suffix can be uppercase or lowercase and can be in any order

30	→ int
30U	→ unsigned int
30L	→ Long
30UL	→ unsigned long
210.0F	→ Float

---

ex:

- 212 → legal
- 212u → Legal
- 0Xfeel → legal
- 212ul → legal
- 078 → illegal : 8 is not an octal digit
- 032uu → illegal : Can not repeat a suffix
- 212LL → legal : Long long

2 Floating-Point Literals.

- A floating point literal has an integar part, decimal part, fractional part and an exponent part
- the signed exponent is introduced by e or E

ex:

3.14159 → legal 3.14159E-5L → Legal

5l0E → illegal : incomplete exponent

2l0F → illegal : no decimal or exponent

.e55 → illegal : missing integer or fraction

---

* operators :-

1 arithmatic operators :-

precedence	Description	Associativity
++ --	Postfix increment / Decrem	Left to Right
++ --	prefix increment/ Decrement	Right to Left
* / %	Multiplication & Division and Modulo	Left to Right
+ -	Addition & subtraction	left to Right

ex: num = 1 ;

num = num++ ; ← لاحظ أن تغيرت قيمة المتغير أ

num = 1 أ

Var = num++ ; ← Var = 1 & num = 2

2 Relational operators :-

< <= → relational less than & less than equal to

> >= → // Greater // & greater // // //

== != → Equality & inequality

} left to right

3 Logical operator :-

&& → Logical AND

|| → Logical OR

! → Logical Not (// negation)

} Left to Right → R to L

→ (A&&B)

→ (A||B)

→ !(A&&B)

4 Bitwise oprators :-

~	→ bitwise complement	R to L
>>	→ bitwise right shift	Left to right
<<	→ " Left "	
&	→ bitwise AND	
^	→ bitwise XOR	
|	→ bitwise OR	

يتم تحويل الأرقام إلى النظام الثنائي ثم يتم التعامل معها على حسب operator

*whence apply logical shift left the Value will be double and vice versa with logical shift right

A = 60 B = 13

A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

S << 2 S = 0101 → L.S → 0001 0100 = 20

S >> 2 S = 0101 → R.S → 0000 0001

* Bitwise XOR : لو عدد (1) زوجي ← 0
لو عدد (1) فردي ← 1

* Bitwise logical operators are useful for bit manipulation

* setting a bit → if n^{th} bit is 0 then set it to 1
→ if it is 1 then leave it unchanged

→ num1 |= (1 << (num2-1));

num1 → the real number
num2 → n^{th} bit of num1

* clearing a bit → if n^{th} bit is 1 then clear it to 0
→ if it is 0 then leave it unchanged

→ num1 &= ~(1 << (num2-1));

* toggling a bit → if n^{th} bit is 1 then change it to 0
→ " " " " 0 " " " " 1

→ num1 ^= (1 << (num2-1));

* getting a bit → check if the n^{th} bit is set (=1) or Not (=0)

→ 1 & (num1 >> (num2-1));

* integer promotions :-

```
Char a = 30; → default sign
Char b = 40;
Char d = (a*b)/10;
printf("%d",d);
```

output : 120

* expression (a*b) cause arithmatic overflow because signed characters can have value only from -128 to 127 and the value of (a*b) equal to 1200 which is greater than 127. But integer promotion happens here and arithmatic done on char types and we get the result without any overflow

another ex:

```
char num1 = 0XFB;
unsigned char num2 = 0XFB
printf("%c\n", num1);
printf("%c\n", num2);
printf("%i\n", num1);
printf("%i\n", num2);
if (num1 == num2) {
   printf("yes");
}
else {
   printf("No");
}
```

output :-

√
√
-5
251
No

* when we print num 1, 2 the same character is printed But when we compare them we get the output as "No". Num 1,2 have the same biliary representation as character. But when comparison operation is performed on num 1,2 they are first converted to int. num1 is a sign char when it is converted to int it's value become -5 (signed value of 0XFB). num2 is unsigned char, when it is converted to int, it's valu became 251 (-5 and 251) have different replesentation as int

* some data types like char, short int and enum take less number of bytes than int, these data types are automatically promoted to int or unsigned int
* if int can represent all values of the operations, the value is converted to int. otherwise, it is converted to unsigned int

---

* arithmatic shift: dealing with bytes or word which represent signed integers. the logical shift right won't work for negative numbers.

ex:

```
1111 1100  ⟶ -4   ⟶ signed number (negative)
0111 1110  ⟶ 126
```

* the problem is that the sign bit 7th is getting set to 0
* this gives an incorrect result. instead of a small negative value, you get a large positive value.
* with an arithmatic shift, the sign bit stays the same as the data shifts

ex:

```
1111 1100  ⟶ -4
1111 1110  ⟶ -2
```

* if (3 ⟶ positive) is right shifted once, it gives 1
  ⟶ because (1.5 rounded down is 1)
* if (-3 ⟶ negative) is right shifted once, it gives -2
  ⟶ because (-1.5 is rounded down is -2)
* then the (-ve) & (+ve) numbers are rounded down
* you don't relly need an arithmatic shift left, because a logical shift left does exactly the same thing

---

```
printf ("%d ", sizeof (printf("mohammed")));
```

ما ترجع بعدد الحروف = 8 أو أنت سم النوع int

```
printf("%d", sizeof(int)); ⟶ Compiler dependant = 2 OR 4
```

```
printf("%s - %i - %s - %s", __FILE__, __LINE__, __DATE__,
                            __TIME__);
```

لطباعة اسم الملف ورقم السطر والتاريخ والوقت. كلهم string ماعدا رقم السطر Integer فهو

*the conditional operator (? :)

Syntax:-

Variable = Expression1 ? Expression2 : Expression3;

ex:

```
int var = (X<25) ? 0 : (X<50) ? 1 : (X<75) ? 2 : 3;
```

* if (X < 25) then var = 0
* if else (X < 50) then var = 1
* if else (X < 75) then var = 2
* else then var = 3

---

ex:
```
int K = 15;
printf("%d ", sizeof(K /= 5+10));
printf("%d ", K);
```
output : 4 15

4 15 ← output

*4: sizeof int → compiler dependent

*the value of K will not be changed because of sizeof

*أن عليّة تحت بداخل sizeof ← لا تغير من قيمة المتغيرات

---

int i = 1, 2, 3; ⟶ syntax error ⟶ compile time error

```
int i;
i = 1,2,3;
```
⟶ i = 1 ⟶ *the assignment operator has higher precedence than comma

↳ (i=1),2,3

---

int i = (1,2,3); ⟶ i = 3 ⟶

* the bracket operator has higher precedence than assignment operator
* the expression within bracket // evaluated from left to right
* But it is always the result of the last expression which get assigned

---

```
int y = 0
int X = (~y == 1);
printf("%d", X);
```
output = 0

y = 0 ⟶ ~y = 1111 1111 = 256 ⟶ ∴ (~y ≠ 1) → ∴ X = 0

* لا يتم التعامل مع هذه العملية على حسب Bitwise Not لأن precedence عملية unary operator وهي كذلك هي أعلى في الأولوية من (==)

```
x = 22;
if (x > 20){
  printf ("11");
}
else{
   printf ("22");
}
```

output : 11

```
x = 22;
if (x > 20)
  printf ("11");
else
  printf ("22");
```

→ with no brackets

output : 11

```
x = 22;
if (x > 20)
  printf ("10");
  printf ("11");
else
  printf ("22");
```

syntax error

* عند استخدام switch case لا يسمح بأن يكون بداخل الأقواس أي قيمة غير integer

switch (value) ⟵ value must be integer.

* عند استخدام cases without break في switch إذا تحقق شرط أحد cases التي تكون بدون break فإن thenextcase سوف تكون true ويتم تحقيقها بدا فلها

* في حالة switch يمكن استخدام default في أي مكان بداخل switch ولكن المتعارف عليه أن تكون في الآخر

ex:

```
switch(var){
    case1:
        printf("1");
    break;
    case 2:
        printf("2");
    case 3:
        printf("3");
    break;
    default:
        printf("4");
}
```

اختياري ⟵ break;

if → var = 1 ⟵ output = 1

var = 2 ⟵ output = 23

```
switch (var) {
   case 1:
     printf ("1");
   break;
   default:
     printf("2");
   case 3:
     printf("3");
}
```

if var = 1 ⟵ output = 1

var = 3 ⟵ output = 3

var = 5 ⟵ output = 23

the output of default

because of there is no break statement with it, so it is good to follow default statement by break statement

وذلك لتجنب الدخول في case التي تليها وتنفيذها

* break statement

```
while (EXPression) {
    // Codes
    if (condition to break){
        break;
    }
    // Codes
}
```

```
do {
    // Codes
    if (condition to break) {
        break;
    }
    // Codes
} while ( Expression );
```

```
for ( init; Exp; inc) {
    // Codes
    if (condition) {
        break;
    }
    // Codes
}
```

* the break statement terminate the execution of the nearest enclosing loop and switch body
* after termination, control passes to the statement that follows the terminated statement
* the break statement shall appear only in a switch body or loop body
* the break statement is almost always used with if statement inside the loop

* continue statement

```
while ( Expression) {
    // Codes
    if (condition) {
        continue;
    }
    // Codes
}
```

```
do {
    // Codes
    if (condition) {
        continue;
    }
    // Codes
} while (Expression);
```

```
for (init; EXP; inC) {
    // Codes
    if ( condition ) {
        continue;
    }
    // Codes
}
```

* we can terminate an iteration without exiting the loop body using the continue keyword
* when continue (jump statement) excute within the body of the loop, all the statements after the continue will be skipped and anew iteration will start
* the continue statement is used only within the loop body

* Patterns in C :

* How to print the given patterns :-

1

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

→ the solution :-

```
int i , k;
for (i=1 ; i<=5 ; i++)
{
  for( k=1 ; k <=5 ; k++)
  {
     printf( "%d ", k);
  }
  printf( "\n");
}
```

2

```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

→ the solution :-

```
for(i=1 ; i<=5 ; i++)
{
  for( k=1 ; k <=5 ; k++)
  {
     printf ("%d ", i);
  }
  printf ("\n");
}
```

3

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

→ the solution :-

```
for(i=1 ; i<=5 ; i++)
{
  for( k=1 ; k<=5; k++)
  {
    printf ("* ");
  }
  printf ("\n");
}
```

4

```
A B C D E
A B C D E
A B C D E
A B C D E
A B C D E
```

* the ASCII code of 'A' = 65

→ the solution

```
for( i = 1 ; i<=5 ; i++)
{
  for( k=1 ; k<=5 ; k++)
  {
     printf ("%c ",(k +64));
  }
  printf ("\n");
}
```

5

```
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
```

→ the solution

```
for (i=1 ; i<=5 ; i++)
{
  for(k=5 ; k>=1; k--)
  {
     printf ("%d ", k);
  }
  printf("\n");
}
```

6

```
5 5 5 5 5
4 4 4 4 4
3 3 3 3 3
2 2 2 2 2
1 1 1 1 1
```

→ the solution :-

```
for(i=5 ; i>=1 ; i--)
{
  for(k=5 ; k>=1 ; k--)
  {
    printf("%d ", i);
  }
  printf("\n");
}
```

7

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

→ the solution :-

~~for(i=1 ; i<=5 ; i++)~~

```
for(i=1 ; i<=5 ; i++)
{
  for(k=1 ; k<=i ; k++)
  {
    printf("%d ", k);
  }
  printf("\n");
}
```

8

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

→ the Solution :-

the same as the previus code but we will print (i)

```
printf("%d ", i);
```

9

```
*
* *
* * *
* * * *
* * * * *
```

→ the solution :-

the same as the previous code but we will print (*)

```
printf("*");
```

10

```
A
A B
A B C
A B C D
A B C D E
```

→ the solution

```
printf("%c ", k+64);
```

11

```
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1
```

→ the solution

```
for(i=5 ; i>=1 ; i--)
{
  for(k=5 ; k>=i ; k--)
  {
    printf("%d ", k);
  }
  printf("\n");
}
```

12

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

the solution :-

```
for(i=1 ; i<=5 ; i++)
{
  for(k=i ; k>=1 ; k--)
  {
    printf("%d ", k);
  }
  printf("\n");
}
```

في معظم وليس كل الحالات

قواعد كتابة (K-Loop)

١- نفحص أول عمود إذا كان
- رقم ثابت ← رقم = Initial K
- أرقام مختلفة ← Initial K=(i)

٢- إذا كان كل منهن يحدث له
- زيادة ← فإن (K++)
  ↳ K <=
- نقصان ← فإن (K--)
  ↳ (K>=)

٣- إذا كان أخر رقم في كل سطر
- متشابه ← (K <= رقم) OR (K >= رقم)
- مختلف ← (K<=i) OR (K>=i)

13

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

→ the solution :-

```
For(i=5 ; i>=1 ; i--)
 {
  For(k=1 ; k<=i ; k++)
   {
    printf("%d ", K);
   }
   printf("\n");
 }
```

14

```
1 2 3 4 5
2 3 4 5
3 4 5
4 5
5
```

16

→ the solution :-

```
for(i=1 ; i<=5 ; i++)
 {
  for(k=i ; k<=5 ; k++)
   {
    printf("%d ", K);
   }
   printf("\n");
 }
```

15

```
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1
```

→ the solution :-

```
for(i=5 ; i>=1 ; i--)
 {
  for(k=5 ; k>=i ; k--)
   {
    printf("%d ", K);
   }
   printf("\n");
 }
```

16

```
5 4 3 2 1
5 4 3 2
5 4 3
5 4
5
```

→ the solution :-

```
for(i=1 ; i<=5 ; i++)
 {
  for(k=5 ; k>=i ; k--)
   {
    printf("%d ", K);
   }
   printf("\n");
 }
```

17

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

the solution :-

```
for (i = 5 ; i >= 1 ; i--)
for (k = i ; k >= 1 ; k--)
```

18

```
A
B A
C B A
D C B A
E D C B A
```

⟹

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

12

→ the solution :-

```
printf ("%c", K+64);
```

19

```
A B C D E
A B C D
A B C
A B
A
```

⟹

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

13

→ the solution :-

```
printf ("%c", k+64);
```

20

```
* * * * *
* * * *
* * *
* *
*
```

```
for (i=5 ; i >= 1 ; i--)
for (k= 1 ; k <= i ; k++)
 printf("*");
```


21

```
* * * * *
*       *
*       *
*       *
* * * * *
```

```
for (i = 1 ; i <= 5 ; i++) {
for (k=1 ; k <= 5 ; k++)
 {
   if ( i==1 || i==5 || k==1 || k==5)
    printf ("*");
    else
     printf (" ");
  }
  }
```

22

```
*
* *
*   *
*     *
* * * * *
```

```
if ( k==1 || i==5 || k==i)
   printf ("*");
else
   printf (" ");
```

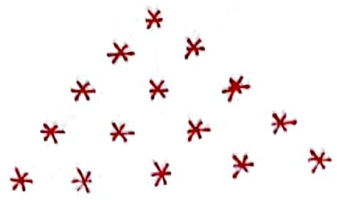
23

```
    *
   * * *
  * * * * *
 * * * * * * *
```

→ the solution :-

```
int i,j,k;
for(i=1 ; i<=5 ; i++) {
  for(j=i ; j<=5 ; j++){
     printf(" ");
  }
  for (k = 1 ; k < (i*2) ; k++) {
     printf ("*");
  }
   printf( "\n");
}
```

24



→ the solution:-

```
int i, c, k;
for(i=1 ; i<=5 ; i++){
    for(c=i ; c<=5 ; c++){
        printf(" ");
    }
    for(k=1 ; k<(i*2) ; k+=2){
        printf("*" " ");
    }
    printf("\n");
}
```

25

```
1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
```

```
for(i=1 ; i<=5 ; i++){
  for(j=1 ; j<=i ; j++){
      printf("%d", j);
  }
  for(k=(i-1) ; k>=1 ; k--){
      printf("%d", k);
  }
  printf("\n");
}
```

26

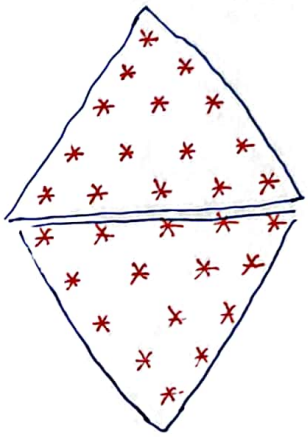
```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

18

→ the solution

```
int i, k;
for(i=1 ; i<=5 ; i++){
  for(k=i ; k<5 ; k++){
      printf(" ");
  }
  for(k=i ; k<(i*2) ; k++){
      printf("%d", k);
  }
  for(k=((i*2)-2) ; k>=i ; k--){
      printf("%d", k);
  }
  printf("\n");
}
```

27



→ the solution:-

```
for(i=1 ; i<=5 ; i++){
  for(k=i ; k<=5 ; k++){
      printf(" ");
  }
  for(k=i ; k>=1 ; k--){
      printf("*" " ");
  }
  printf("\n");
}
for(i=1 ; i<=5 ; i++){
  for(k=1 ; k<i ; k++){
      printf(" ");
  }
  for(k=i ; k<=5 ; k++){
      printf(" " "*");
  }
  printf("\n");
}
```

28

→ the solution :-

```
for(i=1; i<=5; i++){
    for(k=i; k<=5; k++){
        printf(" ");
    }
    for(k=1; k<=(2*i-1); k++){
        if(k==1 || k==(2*i-1))
            printf("*");
        else
            printf(" ");
    }
    printf("\n");
}
for(i=5; i>=1; i--){
    for(k=5; k>=i; k--){
        printf(" ");
    }
    for(k=1; k<=(2*i-1); k++){
        if(k==1 || k==(2*i-1))
            printf("*");
        else
            printf(" ");
    }
    printf("\n");
}
```

→ solution :-

```
int i, j, k=1;
for(i=1; i<=6; i++){
    for(j=1; j<=i; j++){
        printf("%i ", k);
        k++;
    }
    printf("\n");
}
```

29

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
```

* Functions In C

- A function is a set of statements that together perform a specific task

• Advantages of functions:-

- the function increases the modularity of the program

* Modularity ١- تعني تقسيم المشروع إلى عدة أجزاء منفصلة تسمى functions بحيث تكون كل function لا تعتمد على أي function أخرى •

* مسموح لك أن تستخدم function أخرى ولكن التعديل في إحداهن لا يؤثر على الأخرى

- the function increases the reusability because function are reusable
  - once you have created a function you can call it anywhere without copying and pasting entire logic
- Because function increases the modularity of the program. So the program becomes more maintainable (سهولة الصيانة)
  - if you want to modify the program sometimes later, you only need to update your function without changing the base code

* Types of functions:

1. library functions
   - these function are built-in C-programming Library functions to perform various operations as input/output operations
   - before using any library function you must include the corresponding header file
   - ex: printf(), scanf(), strcpy(), strcmp(), etc.
2. user-defined functions
   - these function created by the user according to the requirements
   - ex: motor_move_Right(), motor_move_left(), etc....

* aspects of functions:

1) function definition
2) function call
3) function declaration

1 Function definition :-

- contains single or groups of statements that perform specific tasks.
- syntax:-

```
Return_Type  function_name ( Type1  argument1 , ..... )
{
    local_variables ;
    statement 1;
    statement 2; .....
    return (expression)
}
```

* Return_Type:
  - if your function returns any value then you have to mention the Type of the return value (int, char, double .....)
  - if the function returns no value, then we must use the Void
* parameters : (arguments)
  - the function can have any number of parameters which is used to receive the values from the outer world.
  - if the function does not have any parameter then the parentheses () has void oR we left it empty
  - sometimes we used void in parentheses to avoid passing any value

Function declaration ⟶
```
void print_name (void);
int main () {
```
Function call ⟶
```
    print_name ();
    return 0;
}
```
Function definition ⟶
```
void print_name (void) {
    printf ("Mohammed");
}
```

* if we called a function without it has no definition, this will cause "linking issue"

* Return error state :-

* حالة بحيث أن رقمين أقل من (5)

```
int get_summing (int num1, int num2){
    int Error_state =0; /* No Error */
    if((num1 >5) || (num2>5)){
      Error_state=1;
    }
    else{
      printf("summing =%i\n", (num1+num2));
    }
    return Error_state;
}
```

* قاعدة عامة :
- أي function ليها inputs لازم أعمل validation على inputs
- أي function يجب أن يكون لها Error_state

```
int main(){
    int Error_state=0;
    Error_state = get_summing(2,3);
    Error_state |= get_summing(2,7);
    Error_state |= get_summing(1,5);
    Error_state |= get_summing(1,1);
  return 0; printf("ES =%i\n",Error_state);
}
```

* نقوم بعمل (OR) على عليح Error_state لا overwrite حتى يتم التأكد من أن جميع functions تمت معالجتها بصورة صحيحة

---

*casting OR Type conversion

- Converting one data-type into another
- Type conversion is done at compile time and it is also called widening conversion because the destination data-type can't be smaller than the source data-type

*Types of casting:

1- implicit type conversion : also known as "automatic type conversion".
- it is done by the compiler on its own, without any external trigger from the user
- generally takes place when in an expression more than one data type is present. in such conditions type conversion (Type promotion) takes place to avoid loss of data
- All datatypes of the variables are upgraded to the data type of the variable with the largest data type.

→ char → short int → int → unsigned int → long → unsigned long →

→ long long → unsigned long long → float → double → long double.

* all char and short are converted to int or unsigned int

* if any of the operand in the expression is X then others will be converte to X and we will get the result in X

* X → Long double
  - → double
  - → float
  - → unsigned long int
  - → unsigned long long
  - → long int
  - → long long
  - → unsigned int

* if unsigned int can be converted to long int then it will be converted into Long int and the result will be long int

* Else, both will be converted into unsigned long int and the result will be in unsigned long int

* it is possible for implicit conversions to lose information. signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when Long long is implicitly converted to Float)

ex:

```
int X = 9 /2 ;
printf( " X = %d ", X);
```

output:

X = 4

2- Explicit Type conversion :-

- this process is user-defined
- the user can typecast the result to make it of a particular data type.

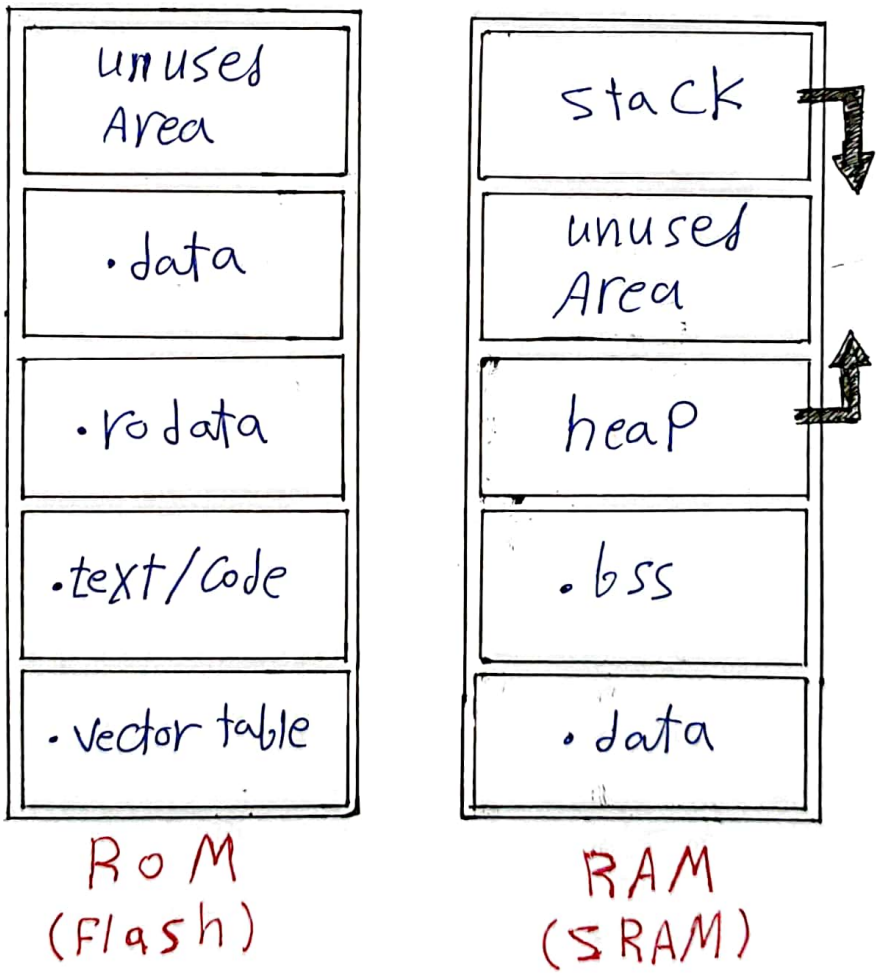
* Syntax:

(type-name) expression

- the following rules have to be followed while converting the expression from one type to another to avoid the loss of information.

① All integer types to be converted to float
② All float types to be converted to double
③ All character types to be converted to integer

{ memory layout introduction }



* (.Text)
  → contains code text
* (.rodata)
  → const global variables
* (.data)
  → 1- initialized (non zero) global variable
  2- initialized (non zero) static global variable
  3- initialized (non zero) static local variable

  → the size of this segment is determined by the size of the values in the program's source code and does not change at run Time
  → It has read/write permission so the value of the variable of this segment can be changed at run time by changing the value of the variable which located in (.data segment in ram) not the original version which located in (.data segment in rom)
  → start up code: runs after electrical power connect to micro control
    → copy contents of (.data→ROM) to (.data→RAM)

* (.bss) → (uninitialized data segment)
  → 1- unintialized (or zero) global variable
  2- unintialized (or zero) static global variable
  3- unintialized (or zero) static local variable
  → start up code: makes the all contents of (.bss) equal to zero

* (.heap)
  → for dynamic memory allocation
    → allocate the memory at runtime
  → managed by the memory managment functions like (malloc. calloc ... etc)
  → it grows and shrinks in the opposite direction of the stack

* stack segment :-

→ 1: initialized OR NOT OR zero local variable
2: Constant local variable
3: local variable from functions and parameters of functions

→ Stack Frame: will create in the stack when a function is called and each function has one stack frame

→ Stack Frame contains:
- the function's local variable, arguments and return value

→ SP (stack pointer):
- register trackes the top of the stack

→ the stack contains a LIFO structure (last input firsoutPu

→ function executed in stack frame, after execution of the function the stack frame will removed from stack segment

---

* Storage classes :- decides the extent (life Time) and scope of the variable

→ global life time → static
→ local lifetiine → automatic

1 auto : the variables defined by auto storage class are called as local variables

* كل local هو auto , والعكس ليس كل global هو auto والعكس

2 register:
→ store local variable not global variable in cpu register instead of ram to have quick access to these variable

→ register variables's address cannot be token

← تصبح auto في أنها تستخدم مع local variable نفقة

← تستخدم بكثرة مع looping

→ these register variable has no default value

3 extern :-

- used when we have functions or variables which are shared between two or more files
- cannot be initialized because it has already been defined in the original file
- any static global variable or static function cannot be extern
- the variables declared as extern are not allocated any memory. it is only declaration are intended to specify that the variable is define elsewhere in the program
- the extern keyword in fact does nothing when applied to function declaration. This because all function declarations have an implicit extern applied
- by using extern you are telling the compiler that whatever follows it will be found at link-time. and Linker throws an error when it finds no such variable exists.
- when an extern variable is initialized, then memory for this is allocate and it will be considered defined. and then the extern does nothing.

```c
#include <stdio.h>
#include "motor.h"
int main() {
 printf("%d", num1);
 return 0;
}
```

output: 5

main.c

```c
#include "motor.h"
unsigned int num1=5;
```

motor.c

```c
extern unsigned int num1;
```

motor.h

4 static :

- static variables are allocated memory in (.data) segment not stack segment
- static global variable & static functions are only visible in its object file instead of the whole program
- static keyword undoes the work of the implicit extern

- static variables are initialized to zero if not initialized
- the auto variables are destroyed when a function call ended where the static variable is over.
  - in other words, static local variable is different from local variable. it is initialized only once no matter how many times that function it resides (تتواجد) is called. it may be used as a counter variable.

ex:-

```
void count (void) {
    static int var1=0;
     int var2 =0;
     var1++;
     var2++;
     printf(" value of var1 = %d \n value of var2 = %d ", var1, var2);
}

int main ( ) {
    count ( );
    count ( );
    count ( );
    return 0
}
```

output:-

value of var1 = 1
// // var2 = 1
// // var1 = 2
// // var2 = 1
// // var1 = 3
// // var2 = 1

Build process or compilation process

* compiler :- يقوم بتحويل High level language الى Low level language
* compiler tool-chain :- مجموعة executable files تستخدم لتحويل compile

HLL ← LLL

* the differences between Host and target machine :-

- Host (PC) : - a computer system on which all the programming tools run
  - where the embedded software is developed, compiled, tested, debugged, optimized and prior to its translation into target device

- target : (MC) - After writing the program, compiled, assembled and linked, it is moved to target

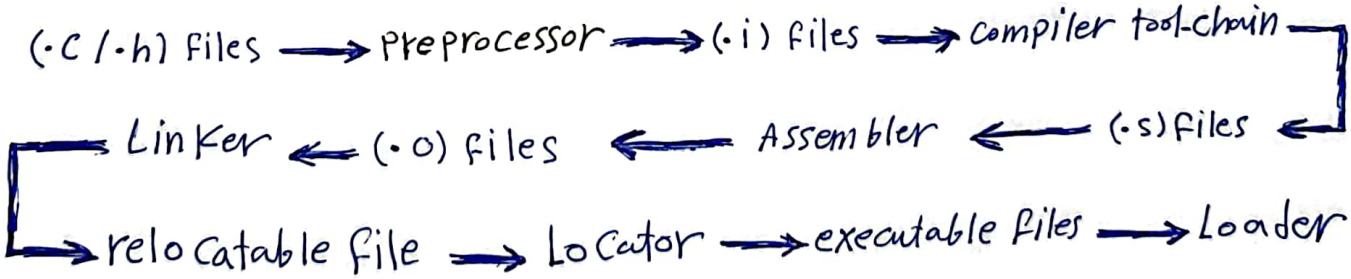
* Differences between Native and cross compiler :-

- Native :- Compiler that generates code for the same plat form on which it runs . it converts high language into computer's native Language (machine language) .e.g , GCC-Compiler
- Cross : - compiler that generates executable code for aplatform other than one onwhich the compiler is running

Native Compiler	Cross Compiler
- translates program for same hardware / plat form / machine on it is running	- translates program for different hardware /platform / machine other than the platform which it is running
- it is used to build programs for same system / machine / os it is installed	- it is used to build programs for other system / machine like avr / arm / pic
- it can generate executable file like (.exe)	- it can generate raw code (.hex)
- TurboC OR gcc	- Keil

* Build process is a Linear process → the output of a tool is input to another tool

ex: the output of the preprocessor is input to the compiler



1: (.c /.h) files :- → pre processed files

2: preprocessor tool :- → converts (.c/.h) → (.i) files

↳ tool chain → cpp.exe

3- (.i) files:- intermidiate file or postProcessed file

→ to open (.i) file from cmd screen we write this command line

>CPP file.c > file.i

→ to specify the output filename, use -o option:-

>gcc -o hello.exe hello.c fun.c .... (.c file) قد يكون ملف واحد أو أكثر

// compile and link source files hello.c and fun.c.... into executable
// hello.exe
>hello
// execute hello.exe

* preprocessor executes any instruction starts with # → (#include, #define.....) and removes any comments.

4 - compiler toolchain :- converts (.i) file ⟶ (.s) file

5 - (.s) file:- assembly file ⟶ assembly code

→ to open .s file from cmd we write:-

>gcc -S file.i

* the compiler compiles the intermidiate file into assembly code

6 - assembler :- converts the assembly code into machine code in the object file "file.o"

7- (.o) Files :- object file (.o) or (.obj)

→ to open (.o) file in cmd we write:

>as -o hello.o hello.s

8 - Linker: the linker (Ld.exe) Links the object file with the library file to produce an executable file (file.exe)

→ to generate (.exe) file we write:-

>ld -o hello.exe hello.o .... Libraries...

*we can see the detailed compilation process by enabling -V (verbose) option

>gcc -V -o hello.exe hello.c

* To enable optimization (4 Level 0,1,2,3)

>gcc -S -o 0 1 2 3 file.i * if 0 → there is no optimization

*Macros :- text replacement by preprocessor

① one line macro

```
#define PI_VALUE (3.14)
```

② Multi line macro

```
#\
define\
TEST\
(77)
```

→ #define TEST (77)

continuation operator (preprocessor directive operator)

* يجب أن تكون هناك مسافة بين \ و define وأيضاً بين \ و TEST

③ Empty macro

```
#define PI
```

* Expression macro

```
#define EXP (2*3)
```

* string macro

```
#define NAME "Mohammed"
```

* Paramatized macro (Function like macro) OR (macro function)

```
#define SUMMING(NUM1, NUM2) (NUM1+NUM2)
```

→ Swap function using macro function

```
#define SWAP(X,Y)\
{\
    *(X) = *(X) ^ *(Y);\
    *(Y) = *(X) ^ *(Y);\
    *(X) = *(X) ^ *(Y);\
}
int main() {
    int X=2, Y=3;
    printf("X =%i\t y=%i\n", X,Y);
    SWAP(&X, &Y);
    printf("X=%i\t y=%i\n", X,Y);
    SWAP(&X,&Y);
    printf("X=%i\t y=%i\n", X,Y);
```

XOR method

→ X= 2 Y=3

→ X= 3 Y=2

→ X= 2 Y=3

* macros VS function:

1: macros

① Macros are preprocessed → processed before your program compiles
② No type checking is done
③ Macros donot check for compilation error
④ using macro increases the code length
⑤ speed of execution using macro is faster
⑥ macro does not check any compile-Time errors.
⑦ macros are useful when small code is repeated many times
⑧ Difficult to debug as they cause simple replacement.

2: function

① functions are not preprocessed but "compiled"
② there is a type checking → warning if issue found
③ check for compilation error
④ using function keeps the code length unaffected
⑤ speed of execution is slower than macros
⑥ function check for any compile-Time error
⑦ functions are useful when large code is to be written
⑧ easy to debug and the stack frame located in the stack memory

*Comments with macro.

① with one Line macro :-

```
#define PI (3.14) // define the ΠΠ Value
#define PI (3.14) /* define the ΠΠ Value */
```

→ with one line macro, any type of comments cause no error (problem)

② with multi Line macro :-

①
```
#\
define \ // "comment"
TEST \ /* "comment" */
77
```

*أى شكل من هذه الأشكال فى كتابة Comment يقوم بالغاء مابعده بحيث يتم اعتبار ما بعد comments هو أيضاً من Comment

②
```
#\
define /* comment */ \
TEST /*Comment*/ \
77
```

الطريقة ② من الطريقة الصحيحة لكتابة multi line macro مع comments

---

* macro function to set, Get, toggle & clear a bit

```
#define SET_BIT (VAR, BIT_POS) ((VAR |= (1<<(BIT_POS)))
#define GET_BIT (VAR, BIT_POS) ( 1 &(VAR) >> (BIT_POS))
#define TOGGLE_BIT (VAR, BIT_POS) ( VAR ^ ( 1<< (BIT_POS))
#define CLEAR_BIT (VAR, BIT_POS) ((VAR) &= ~( 1<<( BIT_POS))
```

---

preprocessor conditional compilation directives

1 #if ..... #endif OR #if .... #elif .... #endif

```
#if LCD_MODE == 4
   print_data_LCD_4bit(data);
#elif LCD_MODE ==8
   print_data_LCD_8bit(data);
#endif
```

```
#if 0 → zero
#endif
```

مابينهم يعتبر كود فلن أى أن preprocessor يعتبر ما بينهم comment فيقوم بالغاءه

2 #ifdef ..... #endif

```
#ifdef anymacro
   statements
#endif
```

* يتم تنفيذ statements اذا تم تعريف anymacro قبل حتى اذا كانت تعريف empty macro

3 #if (cond) #elif (cond) #elif (cond) .... #else .... #endif

---

4- ⊛ Header guard OR File guard.

→ this ensure that when you include headerfile in multiple places, you don't get duplicate declarations of functions, variables .... etc

by using (#ifndef OR #pargma)

→ by using (#ifndef)

اسم الملف ← LCD.h

```
#ifndef _LCD_H_
#define _LCD_H_
   ⋮ file contents
#endif
```

→ define Null if not define

```
#ifndef NULL
#define ((void*)NULL)
#endif
```

⟹ Null Value = 0
↳ ((void*) 0)

5- #error message : Leads to stop preprocessing and show error message at the location where the directive is

يفضل استخدام #error بداخل شرط ما

```
#ifdef TEST
#error "test 1 is defined"
#else
#error "test 2 is defined"
#endif
```

6- #Line new-line-number "new-file-name"

```
65 printf("%i\n", __LINE__);   ⟶ 65
66 #line 33
67 printf("%i\n", __LINE__);   ⟶ 33
68 printf("%i\n", __LINE__);   ⟶ 34
69
```

7 #pragma ⟹ compiler dependant

1 #pragma once

* تقوم بنفس عمل file guard ولكن لا يفضل استخدامها حيث أنها غير مدعومة في معظم standard iso الخاصة بـ C/C++

2 #pragma GCC poison printf ⟶ GCC خاصة بـ

* تقوم بمنع المستخدم من استخدام بعض identifier مثل printf أو أسماء أي function حيث يقوم بإظهار error عند استخدامها

3 #pragma GCC warning "APP.h is used"

* تقوم بإعطاء warning عند استخدام file الذي تتواجد به #pragma

4 #pragma GCC error "APP.h is used"

* تقوم بإظهار error عند استخدام file أو عند عمل include لـ file الذي تتواجد به #pragma

* هناك استخدامات كثيرة لـ #pragma ولكنها تعتمد على Compiler ولذلك يفضل الرجوع إلى compiler documentation قبل استخدامها

---

preprocessor directive operators

1 defined operator :- • tests whether a macro name is defined or not
• used in #if and #elif expressions
• #if defined (name)

ex:

```
#if defined (TEST)
    void fun1(void);
#else
    void fun2 (void);
#end if
```

تشبه

```
#ifdef preprocessor
 compilation directive
```

⟶ the difference between the #ifdef and #if defined is that #ifdef can only use a single condition, while #if defined can do compound conditionals

ex:

```
#if !defined (TEST1) && defined (TEST2)
#error " TEST2 needs TEST1"
#end if
```

(!defined ⟶ if not defined)

[2] stringize operator # :- converts a macro arguments into a string constant

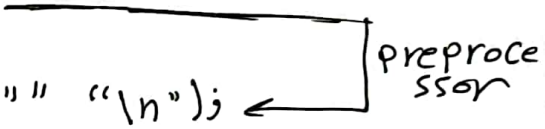
```
#define PRINT_NAME (F_N, S_N)  printf(#F_N " "#S_N "\n");

int main(){
    PRINT_NAME (Mohammed, Ahmed);
    printf("Mohammed" " " "Ahmed" "\n");
```

output:

Mohammed Ahmed

* نلاحظ أنه يمكن عمل أكثر من " " بداخل printf وسوف يؤدي نفس الغرض

```
PRINT_NAME (Mohammed, "Ahmed");
printf("Mohammed" " " "\"Ahmed\"" "\n");
```



← لطباعة " " نضع قبلها (\)

[3] continuation operator :

* يتم استخدامه في macros لعمل macro أكثر من سطر

Arrays

* An array is a collection of elements
* the data types for all elements must be the same and store at the contiguous (متجاور) memory Locations
* each element referenced by using an index
* the index of the array always starts with 0
* syntax of one dimensional array :-

```
Data_type Array_name[Array_size];
```

Array_size ← no. of elements

* initializing the element of an array

```
int array[5] = {11, 22, 33, 44, 55};
```

0 1 2 3 4 ← index → 0 : n-1

* size of array = size of an element (int) * no. of. elements
= 4 * 5 = 20

* another way to initialize the elements of an array

```
int array [0] = 11;
int array [1] = 22;
```

printf ("first element = %i \n", array [0]); → first element = 11

→ C-programming don't provide any specification which deal with problem of accessing invalid index

ex:

```
int arr[5] = { 11, 22, 33, 44, 55 }
int number = 88;
printf ("number = %i \n", number);
arr [20] = 50;
arr [5] = 66;
printf ("the value of index 20 = %i \n", arr[20]);
printf ("the value of index 5 = %i \n", arr[5]);
printf ("number = %i \n", number);
```

the output :-

number = 88
the value of index 20 = 50
the value of index 5 = 66
number = 66

* the valid index 0 : 4
* index 5, 20 → invalid
* number stored in index 5
it's value changed from 88 → 66
because there is no bounds checkin
in C- language

* the begining address of an array

printf ("Add = 0x%X \n", & arr[0]); → 0x403010
printf ("Add = 0x%X \n", arr); → 0x403010

* arr اسم الـ array هو عنوان أول عنصر

* the address of the first element = the begining address of an array
and (&arr[0] = arr)

* to get the element X in an array

→ arr[X] = عنوان أول عنصر + (X * حجم العنصر الواحد)

array اسم

* زمن الوصول لأى عنصر فى array ثابت لا يتغير لأنه يتم تطبيقه نفس المعادلة فى كل مرة يتم فيها عمل access لأى element

→ print an array elements using loop

```
int arr_index = 0
for(arr_index=0 ; arr_index < 5 ; arr_index++) {
    printf("arr elements : %i \t ", arr[arr_index]);
}
```

→ scan an array elements

```
printf("enter array elements : ");
for(arr_index =0 ; arr_index <5 ; arr_index++) {
    scanf("%i", &arr[arr_index]);
}
```

→ number of array elements

```
int arr[5];
int arr_size = sizeof(arr) / sizeof(arr[0])
               20 / 4 = 5
```

---

* array of character :-

```
Char array[] = "Mohammed Ahmed";
printf("%s \n", array); → Mohammed Ahmed
scanf("%s", array);
```

* scanf with array not with array elements → without & operator
* scanf don't accept spaces between characters

* ولذلك من الممكن استخدام gets function

→ gets(array);

* لو أردنا اللى هاحدد عدد indecies فى array of char لازم أزود (1) على عدد الحروف

* Char arr[9] = "Mohammed";

no. of characters of Mohammed = 8

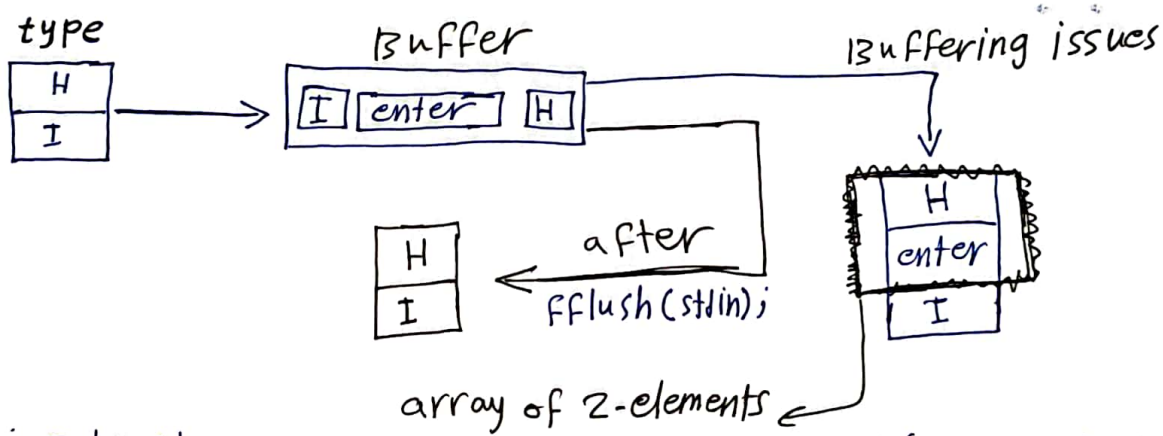
But no. of elements = 9 → no. of characters + Null termination (\0)

* Null termination ← يستخدم فى إنهاء النص فى memory

```
* for (arr_index = 0 ; arr_index < 5 ; arr_index++){
      scanf ("%c", &arr[arr_index]);
      FFLush (stdin);   /*clear buffer*/
  }
```

* عند استقبال char بـ scanf تحدث مشكلة تسمى buffering . لو أنا مثلاً هاستقبل (5) عناصر فإنه بعد إدخال العنصر الأول نقوم بالضغط على enter فيتم تخزين العنصر الأول فى المصفوفة ولكن أيضاً يتم تخزين ASCII code of enter ويتم حفظها أيضاً فى المصفوفة وعندها تستقبل المصفوفة ٣ عناصر فقط بدلاً من ٥ عناصر ولحل هذه المشكلة يتم استخدام أمر fflush(stdin); حيث بعد إدخال العنصر الأول يقوم الأمر بعمل clear للـ Buffer الذى يقوم بالتخزين المؤقت قبل حفظ العنصر فى المصفوفة فيقوم بحفظ العنصر الأول ثم نضغط على enter ثم يقوم الأمر بعد حفظ العنصر الأول بعمل clear للأمر enter من Buffer وعندها تستقبل المصفوفة ٥ عناصر بدلاً من ٣ عناصر.

* we need to store H & I characters



* علشان كده لو أنا بخزن فى مصفوفة onedimension وعدد عناصرها ٢ فوف تستقبل حرف واحد فقط مع enter press بدون عمل clear لـ Buffer.

---

* Two Dimensional Array :-

* syntax:

```
Data_type Array_name [rows][columns];
```

* initialization

```
int arr [2][2] = { 0, 1, 2, 3 };
int arr [2][2] = { {0,1}, {2,3} };
arr [0][1] = 1 ;
```

→ print an array elements (2D array)

```
for(index1=0; index1<3; index1++){
    for(index2=0; index2<3; index2++){
        printf("%i \t", arr[index1][index2]);
    }
    printf("\n");
}
```

→ Scan an array element:

```
for(index1=0; index1<3; index1++){
    for(index2=0; index2<3; index2++){
        scanf("%i", &arr[index1][index2]);
    }
}
```

* int arr [ ][3]; ⟶ has no issues
* int arr [3][ ]; ⟶ compiler error

→ size of an array :-

int arr [2][3]; ⟶ size of an element (int) * no. of element (2*3) = 4*2*3 = 24 byte

char arr [2][3]; ⟶ (1 ⟶ size of char) * 2*3 = 6 byte

ex:

```
Char std_Names [5][30] = { "Mohammed Ahmed",
                           " sameh hussein",
                           " ramadan mohammed",
                           "ramy salah",
                           " galal Ahmed"};
```

→ to print the first element:-

printf("%s \n", std_Names [0]); ⟶ Mohammed Ahmed

→ to print all element

```
for(index=0; index<5; index++){
    printf("%s \n", std_Names [index]);
}
```

* (3D Array)

int arr [index1] [index2] [index3];

int arr [3][3][3];

* How to pass an array as a parameter

→ first way : as an unsized array

```
Void fun (int arr[]){
    ----
}
```

→ second way : as a sized array :-

```
Void fun (int arr[5]){
    ----
}
```

* أى function حاتعمل أى عمليات على أى Array لازم يكون معرف بالنسبة ليها Array length & Array

ex:

```
Void fun (int arr[], int arr_len){
    int index=0
    for(index=0 ; index < arr_len ; index++){
        printf("arr[%i] = %i\n", index, arr[index]);
    }
}

int main(){
    int arr[5] = {11, 22, 33, 44, 55};
    fun (arr, 5);
    return 0;
}
```

* لو مش عايز user يغير قيمتها const اكتبها

output :-

arr[0] = 11
arr[1] = 22
arr[2] = 33
arr[3] = 44
arr[4] = 55

---

* How to pass an 2D array as a parameter.

* First array dimension does not have to be specified
* the second (and any subsequent) dimensions must be given

ex :

```
Void print_Names (const char arr_Names[][30], const int Arr_length){
    int index=0;
    for(index=0 ; index < arr_length ; index++){
        printf("%s\n", arr_Names[index]);
    }
}

print_Names (stu_Names, 5);
```

output :-
Mohamed Ahmed
Samch hussein
ramadan mohammed
ramy salah
galal Ahmed

عرفنا للصنوفة قبل كدة

Pointers

- Pointer is similar to a variable but the difference is that:-
  - Pointer are storing the address of a location in memory
  - Variable Stores the Value.
- syntax:

  Datatype * pointer_name ;

  int *ptr; ⟶ ptr is a pointer point to int

* Data type of a pointer must be the same type that the pointer will point to
* (*) indirection operator or dereference operator: used to access a value indirectly through a pointer and used in the declaration of the pointer
* (&) address operator: gives the address of the operand

(ptr = (&(*ptr))

ex:

```
int num = 55;
int *ptr;
ptr = &num ; ⟶ // assiging addres of num to pointer
printf ("num value = %i\n", num1);
Printf ("num Value = %i\n", *(&num));
printf ("num value = %i\n", *(ptr));
printf ("num address = 0X%X\n", &num);
printf ("num address = 0X%X\n", ptr);
```

output:

```
num Value = 55
num value = 99
num Value 99
num address = 0X4030l0
num address = 0X4030l0
```

```
*ptr = 99; ⟵ indirect access to num value
printf ("num Value = %i\n", nuin);
```

output

```
num value = 99
```

* Data type of a pointer must be the same type that the pointer will point to

ex:

```
unsigned int num = 0X11223344;      ——> 4 byte
unsigned char *ptr1 = &num;         ——> 1 byte
unsigned short *ptr2 = &num;        ——> 2 byte
unsigned int *ptr3 = &num;          ——> 4 byte
```

دا الحجم اللى Pointer
هيشاور عليه قيمة المتغير
وليس حجم Pointer
لأن حجم pointer ثابت
size of pointer = 8 byte

```
printf("Value = 0X%X\n", *ptr1);
printf("Value = 0X%X \n", *ptr2);
printf("Value = 0X%X \n", *ptr3);
```

output:-

```
Value = 0X44
Value = 0X3344
Value = 0X11223344
```

* size of Pointer depend upon operating system and compiler (platform)
  - for 32 bit system ——> size of pointer = 4 byte
  - for 64 bit system ——> size of pointer = 8 byte

* (4 byte & 8 byte) are the same size of address bus (width of address bus)

---

* Wild Pointer:-

- uninitialized Pointer
- global wild Pointer point to address 0X0 in memory, access of this location cause a program to crash or behave badly
- local wild pointer points to a garbage value which is changing continously
- wild Pointer Points to some arbitrary (تعسفى) memory location
- compilers warn about the wild pointer

syntax:

```
int *ptr;
```

- generally: the behavior of wild Pointer is undefined behavior

the solution:

```
int *ptr = &num;   ——> Valid address
int *ptr = Null;   ——> Null pointer
```

* Null Pointer

- A Null pointer is a pointer that points to nothing
- in standard C Null equal to :-

```
#define Null 0
#define Null 0ll
#define Null ((void *)0)
```

- Syntax :

```
int *ptr=NULL;
int *ptr=0;
```

- Usage of the Null Pointer:-
  - A Null pointer prevent the surprising behavior of the program
  - A pointer that is not pointing the address of a valid object or valid memory should be initialized to Null so it prevents the pointer to become a wild pointer and ensure that the pointer is not pointing anywhere.
  - the Null pointer helps in error handling
  - you must validate the pointer before its use.
  - the usage of Null is preferred than usage of zero, because it makes it explicit in code
  - if you try to dereference the Null pointer you will get a segmentation error and the program crash

---

* introduction to dynamic memory allocation:-

- dynamic memory allocation :-
  - enables the programmer to allocate memory at run time
  - enables the ability of program growth and shrinkth as the program run
- there are 4 library functions to facilitate the dynamic memory allocation.

1 - malloc() 2 - calloc() 3 - free() 4 - realloc()

* 1 - malloc function:

- allocates space "in bytes"
- it returns the address of the first space in the allocate space in heap segment in memory
- if there is no space available, the malloc function return Null

*Syntax :-

```
ptr = malloc (byte-size);
ptr = (cast-type *) malloc (byte-size);
```

ex:

```
ptr = malloc (26);        → by user
ptr = malloc (variable);
ptr = (int *) malloc ( 100 * sizeof (int));   → 4 byte
                       100*4
```

- the function will allocate 400 bytes of memory
- if i don't need the reserved space any more. I'll use the free() functi

syntax :

```
free (ptr);
```

* بعد تحرر هذه المساحة ك أن ptr يقوم بتخزين عنوان لايحل بداخله قيمة ن هذه الحالة ptr يسمى Dangling pointer

*Dangling pointer

- When we try to access dangling pointer. it crashes the program
- we can solve this problem by using the Null pointer

```
free (ptr);
ptr = Null; ⟶ ptr is no longer dangling pointer
```

*another example :-

```
int *ptr = Null;
{
    int num = 55;
    ptr = &num;
}
*ptr = 99;
```

⟶ * here ptr is dangling pointer, because the variable num goes out of scope
* we must assign ptr to Null after finishing the operations on variable.

*you must validate the pointer before its use to avoide the segmentation error

1
```
if (ptr != Null)
  {
    /* valid address */
  }
else {
  /* error */
}
```

2
```
if (ptr == Null) {
  /*do nothing */
}
else {
  /*valid address */
}
```

3
```
if (ptr) {
  /* valid */
}
else {
  /*do nothing */
}
```

*Core Dump (segmentation fault)

- is a specific kind of error caused by accessing memory that does not belong to the user
- when we try to read/write operations in a read only location in memory or freed block of memory, it is known as core dump
- it is an error indicating memory corruption (تلف الذاكرة)

*Common segmentation fault scenarios :-

- accessing an address that is freed
- stack overflow : because of recursive function gets called repeatedly
- Dereferencing unintialized pointer

---

*Void Pointer :-

- is a gereric type of pointer that can store the address of any type
- Point to any data type
- it can be type_casted to any data_type

- syntax:

```
Void *ptr = Null;
```

- size of void pointe = 8 bytes → like any pointersize ⟶ address bus size
- in gcc compiler the size of void = 1 byte depend upon the compiler

*we can not use the dereference operator (*) directly with Void pointer to get back the value which is pointed by the pointer

- this is because a void pointer has no data type that creates a problem for the compiler to predict the size of the pointing object.
- So before dereferencing the void point we have to type cast it.

ex:

```
int num = 0x55;
Void *ptr = &num;
printf("num value = 0x%X \n", *ptr);
```

output:-

error: invalid use of void expression

→ this is because I tried to dereference the void pointer without type casting

*the solution:-

```
printf("num value = 0x%X \n", *(int*)ptr);
```

→ we type-cast the void pointer

output:-

num value = 0x55

→ When we need to change the value of num we don't write

```
*ptr= 99;
```

XXX wrong way

→ but we write

```
*((int*)ptr)=99;
```

---

ex: Swap function

[1] call by Value

```
Void swap_num(int num1, int num2){
    int temp=0;
    temp = num1;
    num1 = num2;
    num2 = temp;
}printf("num1=%i \t num2=%i \n",
        num1, num2);
}
```

[2] Call by reference

```
Void swap_num(int *ptr1, int *ptr2){
    int temp=0;
    temp = *ptr1;
   *ptr1 = *ptr2;
   *ptr2 = temp;
printf("num 1 =%i \t num 2 = %i \n",
      (*ptr1), (*ptr2));
}
```

```
int main () {
    int num1 = 55;
    int num2 = 66;
    printf ("num1 =%i \t num2 = %i \n",
            num1, num2);
    swap_num (num1, num2);
    printf ("num1 =%i \t num2 =%i \n",
            num1, num2);
```

```
int main () {
    ...
    swap_num (&num1, &num2);
    ...
```

output :-

```
num1 = 55      num2 = 66
num1 = 66      num2 = 55
num1 = 55      num2 = 66
```

output :-

```
num1 = 55      num2 = 66
num1 = 66      num2 = 55
num1 = 66      num2 = 55
```

* In the call by value :-
  - Values of actual parameters are copied to function's parameters
  - any change made inside functions are not refelected in actual parameters

* In the call by reference :-
  - addresses of the actual arguments are copied and then assigned to the corresponding arguments
  - So, in the call by reference both actual and function's parameters are pointing to the same memory location.
  - So, any change made inside function is reflected in actual parameters

ex:-

```
int square (int n) {
    return (n*=n);
}
int main () {
int y= square (x);
    return 0;
}
```

after function call :

y = 4 & X = 2

(X) was not changed by function

```
void square (int *n) {
    (*n) *= (*n);
}
int main () {
  int x=0;
  square (&x);
   return 0;
}
```

after function call

X = 4

(X) was changed by function

* function return it's value through the parameters
* call by ref function needn't return expression because the function's changes refelect in actual parameters.

*if there is a function that return the square of 3 numbers, the normal return can't do that, so we must use pointer because we don't have to use return

```
Void Get_square (int *ptr1, int *ptr2, int *ptr3) {
    (*ptr1) *= (*ptr1);
    (*ptr2) *= (*ptr2);
    (*ptr3) *= (*ptr3);
}
```

→ if the user enter the value of *ptr = Null, the compiler 'll give us an error so we should first check the validation of the pointer

```
if ((Null == ptr1) || (Null == ptr2) || (Null == ptr3)) {
    printf ("Null pointer passed");
}
else {
    ═══════
    ═══════
}
```

```
*scanf ("%d\n", &num);
*scanf ("%d\n", ptr);   → ptr = &num
```

ex:-

```
Char square_Num (int *ptr){
    Char Error_state=0;/*No error*/
    if (Null == ptr) {
        Error-state= 1; /* error*/
    }
    else {
        (*ptr) *= (*ptr);
    }
    return Error_state;
}

Char Getsquare (int *ptr1, int *ptr2, int *ptr3) {
    Char Error_state=0   /* No error */
    Error_state = square_Num (ptr1);
    Error_state |= square_Num (ptr2);
    Error_state |= square_Num (ptr3);
    return Error_state;
}
```

```
int main(){
int num1 =2, num2=3, num3=4;
Char Error_state =0;
Error_state = Getsquare (&num1,
                         &num2,
                         &num3);
if (1 == Error_state) {
    printf ("Error..");
}
else {
    printf ("%i\t%i\t%i\t\n",
             num1, num2, num3);
}
return 0;
}
```

═══════
═══════
═══════

Pointer Arithmetic

* a pointer is an address, which is a numeric value
* when we increment or decrement the pointer then pointer point to the next or previous memory location that depend upon the pointed data type.

Data_type var1 = 55;
Data_type *ptr = &Var1;

- if Data_type ⟶ int ⟶ take 4 byte
- if the address ⟶ 0X403010 then
  ⟶ the next address after increment will be 0X403014
- if Data_type ⟶ short ⟶ take 2Byte ⟶ 0X403012 and so on.....

ex:

```
int nums[5] = { 11, 22, 33, 44, 55 };
int *ptr = Null;
ptr = & nums [0];
Printf (" ptr= 0x%X - Value =%i \n ", ptr, *ptr);
Ptr += 1;
printf(" ptr = 0X%X - Value = %i \n ", ptr, *ptr);
ptr += 1;
Printf ("ptr = 0X%X - Value = %i \n ", ptr, *ptr);
Ptr -= 1;
Printf("ptr = 0X%X - Value = %i \n", ptr, *ptr);
```

Output:

ptr = 0X403010 - Value = 11
ptr = 0X 403014 - Value = 22
ptr = 0X 403018 - Value = 33
ptr = 0X 403014 - Value = 22

---

unsigned short Nums[5] = {0X0011, 0X0022, 0X0033, 0X0044, 0X0055};
unsigned int *ptr = & Nums [0]; Array ptr سنرى مختلف عند خلى بالك ان

⟶ ptr = 0X403010 Value = 00220011
⟶ ptr += 1;
⟶ ptr = 0X403014 Value = 00440033
⟶ ptr += 1;
⟶ ptr = 0X403018 Value = 00550044

* we can not use each arithmetic operators with pointer, because there are no meaning to use them like (* , /), and also there is no meaning to addition of two pointers.

* Pointer subtraction :-
  - the subtraction of one pointer from another pointer result to calculate the number of bytes between the two pointer or give us the number of elements in array if the two pointer pointing to the start and the end of the array.

* pointer comparison :-
  - we can also compare the pointer with another but they should be pointed to the same array or memory block.
  - pointers comparison is invalid when pointers point to the 2 different memory blocks

---

* Arithmetic operations on Void pointers

```
int Numbers[2] = { 0XAA BB CC DD , 0X 11 22 33 44 }
Void **ptr = &Numbers[0];
Printf("0X%X\n", *((int*)ptr));  ⟶ 0XAA BB CC DD
ptr += 1;
prilitf("0X%X\n", *((int*)ptr));  ⟶ 0X 44 AA BB CC
ptr+=1;
Printf("0X%X\n", *((int*)ptr));  ⟶ 0X 3344 AA BB
```

(1 byte) ← Void فقط لأن حجم (1 byte) * نقوم بزيادة
(4 byte) * وللقيام بزيادة

```
ptr = &Numbers[0];  ⟶ 0XAA BB CC DD
ptr = (int*) ptr+1;  ⟶ 0X 11 22 33 44
```

explicit casting * ويكون حسب تكون قيمة الزيادة

* if we do this expression

```
(int*) ptr += 1;  ⟶ In Valid expression
ptr = &Numbers[0];  ⟶ 0XAA BB CC DD
ptr = (short*) ptr+1;  ⟶ 0X 3344 AA BB
printf("0X%X\n", *((int*)ptr));  ⟶ 0X3344AABB
```

هنا بنقوله شوف تد ايه في مكانك
هنا بنقوله اتحرك تد ايه في memory

p++
*p++
*(p++)
} post increment Pointer (*p)++ → post increment data pointed by the pointer

* If ptr is a void pointer:.

* ptr += 1 ⟶ increment 1 byte as size of void is 1byte

* ptr = (int*) ptr+1 ⟶ increment 4byte as we caste the pointer to int before we increment it.

* ptr = (int*) ++ptr ⟶ increment 1byte as we increment the pointer before we cast it.

ex:

```
int Numbers[5] = {0XAABBCCDD, 0X11223344, 0X55, 0X66, 0X77};
Void *ptr = &Number[0]; ⟶ 0XAABBCCDD
ptr = (int*) ptr++; ⟶ A=A++ تظل القيمة كما هى قبل
printf("0X%X\n", *((int*) ptr)); ⟶ 0XAABBCCDD
ptr = (int*) ++ptr; ⟶ 1byt حايعمل زيادة بمقدار
printf("0X%X\n", *((int*) ptr)); ⟶ ثم ينظر له جديد ال 4byte
                                  ⟶ 0X44AABBCC
ptr = (int*) ++ptr;
printf("0X%X\n", *((int*) ptr)); ⟶ 0X33 44 AA BB
```

```
ptr = &Numbers[0];
printf("0X%X\n", *((int*) ptr)); ⟶ 0XAABBCCDD
printf("0X%X\n", *((int*) ptr+1)); ⟶ 0X11223344 } هذه العمليات لاتغير
printf("0X%X\n", *((int*) ptr+2)); ⟶ 0X00000055 } فى قيمة Pointer نفسه
printf("0X%X\n", *((int*) ptr)); ⟶ 0XAABBCCDD ← بنفس القيمة } لذلك يظل محتفظ
```

int num[5] = {0XAABBCCDD, 0X11223344, 0X55, 0X66, 0X77};

int *ptr_num = &num[0]; ⟶ 0XAABBCCDD

int *ptr = &num[0]; ⟶ 0XAABBCCDD

ptr = ptr_num++; ⟶ ptr = 0XAABBCCDD
↳ ptr_num = 0X11223344

---

ptr = &num[0]; ⟶ 0XAABBCCDD

ptr_num = &num[0]; ⟶ 0XAABBCCDD ⟶ address = 0X403010

ptr = *ptr_num++; ⟶ ptr = 0XAABBCCDD ⟶ not real address
OR = *(ptr_num++); ⟶ ptr_num = 0X403014 ⟶ real address

في هذه العملية ← (ptr = *ptr_num++;) نقوم بعمل عملية assignment أولاً للقيمة 0XAABBCCDD وهي (ptr_num) القيمة التي بداخل العنوان المخزن في (address) وليس (Value) ثم نقوم بعدها بعملية increment للمؤشر ptr_num عملية زيادة على pointer نفسه وليس القيمة التي بداخله

* وبعد انتهاء العملية يكون بداخل كل pointer
ptr ⟶ Value
ptr_num ⟶ address

* عند عملية dereferencing إلى ptr يؤدي إلى crash لأن ما بداخل ptr ليس عنوان حقيقي ولكنه قيمة

ex : In details

printf ⟶ *ptr ⟶ 0XAABBCCDD (Value)
printf ⟶ *ptr_num ⟶ 0XAABBCCDD (Value)
printf ⟶ ptr ⟶ 0X403010 ⟶ (address)
printf ⟶ ptr_num ⟶ 0X403010 ⟶ (address)
ptr = *ptr_num++; OR ptr = *(ptr_num++)
printf ⟶ ptr ⟶ 0XAABBCCDD ⟶ (Value)
printf ⟶ ptr_num ⟶ 0X403014 ⟶ (Address)
printf ⟶ *ptr_num ⟶ 0X11223344 ⟶ (Value)
printf ⟶ *ptr ⟶ program crash because we tried to access the value as an address

remember

- *(p++) is the same as (*p++)
- *(++p) // // // // (*++p)

ex: Decrement :-

```
int num[5] = {0XAABBCCDD, 0X11223344, 0x55, 0x66, 0x77};
int *ptr = num;   ⟶  *ptr = &num[0]
int *ptr_num = num;
```

printf ⟶ *ptr ⟶ 0XAABBCCDD
printf ⟶ *ptr_num ⟶ 0XAABBCCDD
printf ⟶ ptr ⟶ 0X403010
printf ⟶ ptr_num ⟶ 0X403010

[1] ptr = ++ptr_num;

printf ⟶ *ptr ⟶ 0X11223344
printf ⟶ *ptr_num ⟶ 0X11223344

[2] ptr = *++ptr_num; oR ptr = *(++ptr_num);

printf ⟶ *ptr ⟶ program crash
printf ⟶ *ptr_num ⟶ 0X11223344
printf ⟶ ptr ⟶ 0X11223344 ⟶ not real address
printf ⟶ ptr_num ⟶ 0X403014 ⟶ real address

[3] ptr = ++(*ptr_num);

printf ⟶ ptr ⟶ 0XAABBCCDE ⟶ increment data by 1
printf ⟶ ptr_num ⟶ 0X403010 ⟶ address of first element
printf ⟶ *ptr ⟶ program crash
printf ⟶ *ptr_num ⟶ 0XAABBCCDE

---

*Modify the pointer itself

pre_inc: *(++p) oR (*++p) oR (++p)
post_inc: *(p++) oR (*p++) oR (p++)

*Modify the data (value) pointed to by the pointer

++(*p) and (*p)++

* Pointers with const keyword

[1] Constant Pointer to non constant data

- in this case, the pointer points to a fixed memory location, and the value at that location can be changed becaus it is a variable, but the pointer will always point to the same location because it is made constant here

*syntax:

```
Data_type *const pointer_name = address;
```

ex:

```
int a=1 , b=2;
int *const ptr = &a;
```

← يجب أن يتم تعيين قيمة pointer من هذا السطر حتى لا يعطى error التالى أن عند تعريف pointer

```
ptr = &b;
```

← error: assignment of read only variable "ptr"

[2] non constant pointer to constant data.

in this case, the data pointed by the pointer is constant and cannot be changed. Although the pointer itself can change and points some where else (as the pointer itself is a variable)

*syntax:

```
const Data_type * pointer_name;
Data_type const *pointer_names;
```

ex:

```
int a = 10 , b = 5;
const int *ptr= &a;
a = 4;
```

→ acceptable assignment

```
*ptr = 6;
```

→ error: assignment of read only location

```
ptr =& b;
```

→ acceptable

[3] constant pointer to a constant data

in this case, the data pointed to by the pointer is constant and cannot be changed, the pointer itself is constant and cannot change and point somewhere else

* syntax:

```
const data_type *const pointer_name;
```

* Pointers with strings :-

→ creating a string using character array :-

```
char arr[6] = "Hello";
```

→ string is a sequence of characters which we save in an array. And in C Language the '\0' Null character marks the end of a string.

→ the array variable-name of the string arr holds the fixed address of the first element of the array. i.e, it points at the starting memory address. So, we can create a character pointer ptr and store the address of the string arr variable in it, this way ptr will point at the string arr

```
Char *ptr = "Hello";   ——→ directly
Char *ptr = str;       ——→ str is a character array
```

→ the pointer will be given the address of the first element (character)

So → *ptr = H

→ when strings are declared as character array, they are stored like other types of arrays in C. For example;

- if array is auto (local) variable then string is stored in stack segment
- if it's a global or static variable then stored in data_segment, etc...

→ when strings are declared as character pointer, they are stored in a read only block (generally in data segment). in the above "Hello" is stored in a read only location, but pointer is stored in a read/write memory. you can change ptr to point something else but cannot change value at present ptr so this kind of string should only be used when we don't want to modify string at a later stage in the program

→ if we tried to change the value of ptr this lead to program crash

→ we can avoid this problem by using const keyword with data i.e,

```
const char *ptr="Hello";
```

→ we can change the pointer to point something else but there we lose the address of the previous data

→ we can avoid this by using const keyword with pointer i.e,

```
const char *const ptr="Hello";
```

ex:

```
char *Msg = "Embedded diploma";
```

Msg ↑ (Msg+10) ↑ → offset

* you can increment or add an offset to the pointer to access subsequent characters

printf ⟶ %s ⟶ Msg ⟶ Embedded diploma

printf ⟶ %s ⟶ (Msg+10) ⟶ diploma

printf ⟶ %c ⟶ *(Msg) ⟶ E

printf ⟶ %c ⟶ *(Msg+10) ⟶ d

printf ⟶ %s ⟶ *(Msg+10) ⟶ program crash

← لأن التعبير *(Msg+10) عبارة عن حرف وانا بحاول أطبع string

← إذا استخدمت عملية dereferencing يجب أن استخدم %c formate

---

* Double pointer concept:

⟶ when we define a pointer to pointer, the first pointer is used to store the address of the variable and the second pointer is used to store the address of the first pointer

ex:

```
int num = 0x11;
int *ptr = &num;
int **ptr2 = &ptr;
```

	address	Value
num	0x403010	0x11
ptr	0x403018	0x403010
ptr2	0x403026	0x403018

printf ⟶ 0x%x ⟶ &num ⟶ 0x403010

printf ⟶ 0x%x ⟶ ptr ⟶ 0x403010

printf ⟶ 0x%x ⟶ *ptr ⟶ 0x11

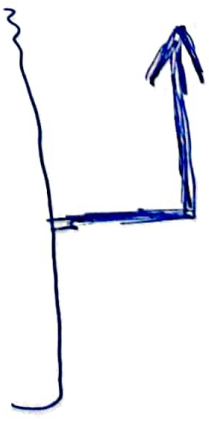
printf ⟶ 0x%x ⟶ &ptr ⟶ 0x403018

printf ⟶ 0x%x ⟶ *(&ptr) ⟶ 0x403010

printf ⟶ 0x%x ⟶ ptr2 ⟶ 0x403018

printf ⟶ 0x%x ⟶ *ptr2 ⟶ 0x403010

printf ⟶ 0x%x ⟶ **ptr2 ⟶ 0x11



*return address from function in C:-

→ If we need to return address of a variable from a function, this address must be a valid address after the function execution.

→ this means that, the variable needs to be "static variable"

→ static variable: has a property of preserving it's value even after being out of it's scope

ex:

```
int * return_add (void){
    static int num = 0x44;
    return (&num);
}
int main(){
    int * ptr = return_add ();
    printf("value = 0x%x", *ptr); ——→ value = 0x44
    return 0;
}
```

---

* return array from function:

→ C-programming does not allow to return an entire array as an argument to a function. However, we can return a pointer to an array

ex:

```
int * ret_arr(void){
    static int arr[5] = {11,22,33,44,55};
    return arr;
}
int main(){
    int *ptr = ret_arr();
    int count = 0;
for(count = 0; count<5; count++){
    printf("value = %i \t", *ptr);
    ptr++;
}
return 0;
}
```

output → 11, 22, 33, 44, 55

*after finishing loop the ptr pointing to the memory location next to the last element in the array

* if we need to make ptr after finishing the loop to still point to the first element of an array we do this.

```
For (count=0 ; count<5 ; count++){
   printf("array = %i\t", *(ptr+count));
}
```

output:.

array = 11 22 33 44 55

and ptr still point to the first element of an array.

---

* Relationship between pointer and array in C :

→ the name of an array is a fixed address, it always points to the first element of the array

ex:

```
unsigned int arr[5] = {11,22,33,44,55};
unsigned int *ptr = arr;
```

printf ⟶ &arr[0] ⟶ 0x404020

printf ⟶ arr ⟶ 0x404020

printf ⟶ ptr ⟶ 0x404020

arr++; ⟶ error: arr is a fixed address

ptr++;

printf ⟶ ptr ⟶ 0x404024

* another way to access array elements :-

arr[index] = *(arr+index)

printf ⟶ *arr OR *(arr+0) ⟶ 11

printf ⟶ *(arr+1) ⟶ 22

printf ⟶ *(arr+2) ⟶ 33

---

* access 2D-array elements:

arr[a][b] = *(arr[a] + b)

OR

arr[a][b] = *(*(arr+a)+b)

ex:

```
unsigned int arr[3][3] = {{11,22,33},
                          {44,55,66},
                          {77,88,99}};

unsigned int ptr = arr;
```

printf ⟶ arr[1][1] ⟶ 55
printf ⟶ arr[2][2] ⟶ 99
printf ⟶ *(arr[2]+2) ⟶ 99
printf ⟶ *(ptr+4) ⟶ 55
printf ⟶ *(*(arr+0)) ⟶ 11
printf ⟶ *(*(arr+1)) ⟶ 44
printf ⟶ *(*(arr+2)) ⟶ 77
printf ⟶ *(*(arr+2)+0) ⟶ 77
printf ⟶ *(*(arr+2)+1) ⟶ 88
printf ⟶ *(*(arr+2)+2) ⟶ 99

```
for(index1=0 ; index1<3 ; index1++){
    for(index2=0 ; index2<3 ; index2++){
        printf("%i\t",(*(*(arr+index1)+index2)));
    }
    printf("\n");
}
```

output:

```
11  22  33
44  55  66
77  88  99
```

*(arr+0)	11	22	33
*(arr+1)	44	55	66
*(arr+2)	77	88	99

((arr+0)+0) ←	11
((arr+0)+1) ←	22
((arr+0)+2) ←	33
((arr+1)+0) ←	44
((arr+1)+1) ←	55
((arr+1)+2) ←	66
((arr+2)+0) ←	77
((arr+2)+1) ←	88
((arr+2)+2) ←	99

```
char arr[9] = "Mohammed";
```
- `[9]` → +1 for Null termination
- `char` → 1D imensional array

```
Printf("%C\n", arr[0]); ⟶ M
Printf("%C\n", *(arr+1)); ⟶ O
```

```
char arr[2][9] = {"Mohammed", "Ahmed"};
```
- `[2]` → 2D-array

```
printf("%C\n", *(arr[0]+0)); ⟶ M
printf("%C\n", *(arr[1]+3)); ⟶ e
printf("%s\n", *(arr[0])); ⟶ Mohammed
printf("%s\n", *(arr[1])); ⟶ Ahmed
```

---

* string array using the array of pointer to char:

```
char *Names[2] = {"Mohammed", "Ahmed"};
```
- → single dimensional array of 2 element
- → each element is a pointer (Address)
- → it is called array of pointer

unsigned int *ptr[2];

→ ptr is an array of 2 element, each element is a pointer pointing to unsigned int

ex:

```
unsigned int arr1[5] = {11,22,33};
unsigned int arr2[3] = {44,55,66};
unsigned int * ptrArr[2] = {(&arr1[0]) , (&arr2[0])};
```

printf ⟶ *ptrArr[0] ⟶ 11
printf ⟶ *(++ptrArr[0] ⟶ 22
printf ⟶ *ptrArr[0] ⟶ 22
printf ⟶ *(--ptrArr[0]) ⟶ 11

* تم تغيير مكان المؤشر ولا يعود إلى مكانه إلا إذا قمنا بعمل العملية العكسية وهي decrement واذا اردنا تجنب هذه المشكلة نستخدم const keyword

```
unsigned int *const ptrArr[2] = { &arr1[0] , &arr2[0] };
```

printf ⟶ *(++ptrArr[0]) ⟶ error : ptrArr → array of const pointer

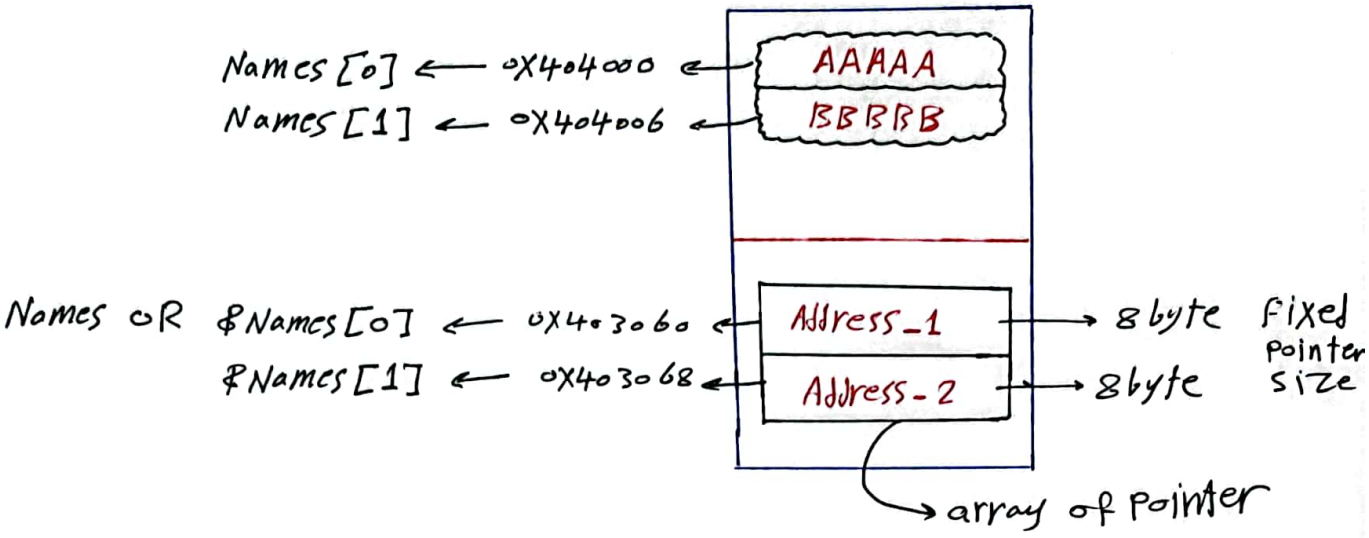
printf ⟶ *(ptrArr[0]+1) ⟶ 22
printf ⟶ *ptr[1] ⟶ 44
printf ⟶ *(ptr[1]+2) ⟶ 66

---

* Access array of pointer to string :-

1 By using pointer to the 1-D array

```
Char *Names [6] = {"AAAAA", "BBBBBB"}; /* creat 1D array of pointer that points to string */

Char *(*ptr_name)[6] = Names; OR (&Names) /* Pointer to an array of 6 Char elements */

int main(){
    printf("%s\n", (*ptr_name)[0]); ⟶ AAAAA
    printf("%s\n", (*ptr_name)[1]); ⟶ BBBBBB
    printf("0x%X\n", Names); ⟶ 0X403060
    printf("0x%X\n", Names[0]); ⟶ 0X404000
    printf("0x%X\n", Names[1]); ⟶ 0X404006
```



* the array located in (.data segment) in a different place from two strings but it has their addresses
* As strings when we declare it as a character pointer, they are stored in a read_only block, if we tried to change the value of them the program crash

---

[2] By using pointer to Pointer

```
Char *Names [6] = { "AAAAA", "BBBBBB" };
Char **ptr_name = Names;    OR (&Names)
printf ("%s \n", ptr_name [0]); ——→ AAAAA
printf ("%s \n", ptr_name [1]); ——→ BBBBBB
```

---

*ACCESS 2D-array of characters using the pointer to the array:

[1] Pointer to an array of 6 character elements :-

```
Char (*ptr) [6] = Null;
Char Names [2][6] = { "AAAAA", "BBBBBB" };
ptr = Names;
printf ("%s\n", ptr); ——→ AAAAA
                  ↳ OR ptr [0]
                  ptr [1] ——→ BBBBBB
ptr++; ——→ increment by 6 bytes ——→ array size
printf ("%s \n", ptr); ——→ BBBBBB
```

sizeof (ptr) → 8 byte
sizeof (*ptr) → 6 byte

[2] Pointer to 2D array

```
Char Names [2][6] = {"AAAAA", "BBBBBB"};
Char (*ptr) [2][6] = Names;
```

```
Printf ("%s\n", (*ptr) [0]); ——→ AAAAA
printf ("%s\n", ptr); ——→ AAAAA
printf ("%s\n", (*ptr) [1]); ——→ BBBBBB
```

---

ex:

```
unsigned int numbers [2][3] = { {11, 22, 33}, {44, 55, 66} };
unsigned int (*ptr) [3] = numbers;
```

printf ——→ %X ——→ *ptr ——→ 0X403010
printf ——→ %i ——→ *(*ptr) ——→ 11
printf ——→ %i ——→ *(ptr[0]) ——→ 11
ptr++; ——→ pointer points to 44 ——→ move 3 element ——→ array size
printf ——→ %i ——→ *(*ptr) ——→ 44
printf ——→ %i ——→ *(ptr[0]) ——→ 44
printf ——→ %X ——→ *(ptr+1) ——→ 0X403028 ——→ move 3 element
printf ——→ %i ——→ *(ptr[0]+1) ——→ 55 ——→ move 1 element
printf ——→ %i ——→ *(ptr[0]+2) ——→ 66
printf ——→ %i ——→ (*ptr[0]+1) ——→ 45
(ptr[0] → 44)

---

* Recursion :-

Recursion: is a process in which function call itself and this function called recursive function

* recursive function should have the termination condition to break the recursion, otherwise, the stack-overflow will occur and your program will crash.

* recursive function are useful to solve many mathematical problems, like calculating the factorial.

*Types of recursion:

[1] Direct recursion:-

```
Void function ( parameter)
{
  // code
  if (condition) {
     function (parameter);
  }
  // code
}
```

[2] indirect recursion

```
void function_A (parameter) {
    function_B (parameter);
}

Void function_B ( parameter) {
    function_A (parameter);
}
```

Ex: Calculate the factorial of a number using recursive function:-

```
unsigned int factorial ( unsigned int number) {
    if ( 0 == number) {
        return 1;
    }
    else {
        return ( number * factorial (number-1));
    }
}
```

---

*Advantages of recursion

→ you can make your code simpler and you can solve problems in an easy way

→ useful to solve many mathematical problems

*disadvantages:

→ recursion makes your code slow because each time needs to create a stack frame for the function call

→ recursion takes alot of stack memory

→ recursive function needs a base condition to break the recursive calling of the function either it will cause of stack-overflow, and if recursion is too deep then there is a danger of running out of space on the stack and it can cause of the program crashes.

*recursion vs iteration

→ recursion is slower as compared to the iteration.

→ recursion uses more memory as compared to the iteration

→ sometimes it is much simpler to write the algorithm using recursion as compared to the iteration.

* Passing an 1-D array as a parameter using pointer.

```
int numbers [5] = {11, 22, 33, 44, 55};
Char print_numbers (const int *My_numbers);
int main (){
    char ret_Val = 0;
    ret_Val = print_numbers ( Null );   ---> ret_Val = 1 = NOK
    ret_Val = print_numbers ( numbers); ---> ret_Val = 0 = OK
    return 0;
}
char print_numbers (const int *My_numbers) {
      char ret_Val = 0 ;  /* OK */
      if (Null == My_numbers) {
          ret_Val = 1 ;  /* NOK */
       }
       else {
           for (int index = 0 ; index < 5 ; index ++) {
               printf ("%i \t", My_numbers (index) ;
           }
       }
```

---

ex: write a function to get an element index in an array:-

```
unsigned char Get_Elem_index ( const unsigned int *MyNumbers,
                               const unsigned int MyNumbersLen,
                               const unsigned int elem,
                               signed int * const Elem_index )
{
 unsigned char RetVal = 0; /* OK */
 unsigned int Number_index = 0;
 if ((Null == MyNumbers) || (Null == Elem_index))
 {
    RetVal = 1; /* NOK */
    *Elem_index = -1;
 }
```

```
else{
  for(Number_index=0;Number_index<MyNumberLen;Number_index++){
     if(elem==MyNumbers[Number_index]){
        *Elem_index=Number_index;
         break;
     }
     else{
        *Elem_index=-1;
     }
  }
}
return RetVal;
}
```

*passing 2D-array

→ slides page·166

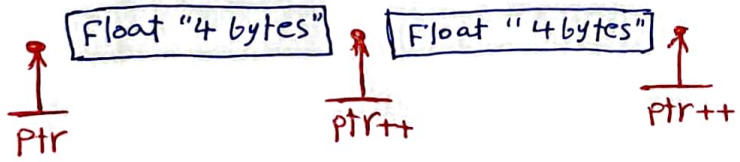
---

*How to find size of array in C without using sizeof:-

→ when we increment or decrement the pointer then pointer point the next or previous memory location

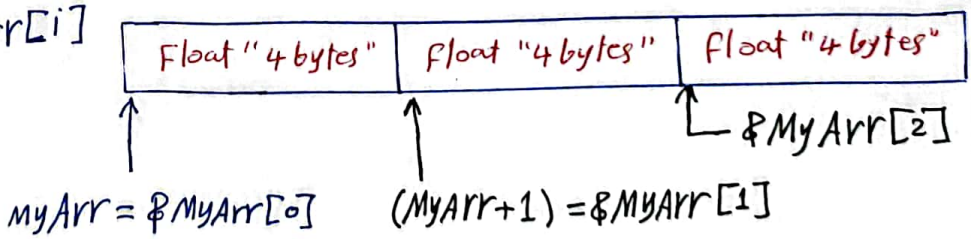
→ the next or previous location depends on the pointer type

→ if the ptr is a pointer to the float and the float size is 4 bytes then the next location will be 4 bytes ahead of the current location



→ MyArr[i] = *(MyArr+i)

→ (MyArr+i) = &MyArr[i]



→ unsigned int Arr[6] = {0x11,0x22,0x33,0x44,0x55,0x66};
  ↳ array of 6 element

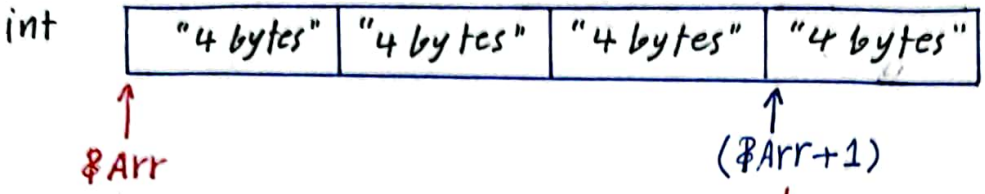
→ Arr → pointer to the first element of the array

→ &Arr → pointer to an array of 6 element

→ &Arr+1 → Address of the next memory block (Address ahead of 6 integers)

→ *(&Arr+1) → it gives the address of first element of second memory block

```
int Arr[] = {0x11, 0x22, 0x33};
*(&Arr+1)-Arr;
```



↳ Pointer to an array of 6 element

↳ address of next memory block (ahead of 3 integers)

*(&Arr+1) -Arr;

↳ same address as (&arr+1) but type of pointer is int

↳ Points to the address of 3 integers ahead of array

→ So the difference between (*(&Arr+1) -Arr) is 3 which is the value of difference of two pointer from integer data-type so the difference calculate the number of blocks between two pointers.

---

* memset function:-

- Description:
  the C library function that copies the character c to the first n character of string pointed to, by the argument str.
- Declaration:
  void *memset (void *str, int c, int length)
- parameters:
  - str ⟶ this is a pointer to the block of memory to fill
  - c ⟶ this is the value to be set. the value is passed as an int, but the function fills the blocks of memory using the unsigned char conversion of this value
- Return value:
  the memset function return the value of str, which is a void pointer so that we can pass any type of pointer.

- example :-

```
char name1[30] = "Mohammed Ahmed";
char name2[30] = "Mohammed Ahmed";
printf("%s\n", name1); ⟶ Mohammed Ahmed
memset(name1, '*', 30);
printf("%s\n", name1); ⟶ ***.....*** ⟶ 30
memset(name2+3, '*', 6);
printf("%s\n", name2); ⟶ Moh******Ahmed
                          3   6
```

- implementation of memset function:

```
void *my_memset(void *dest, int value, unsigned int length){
    unsigned char *temp_ptr = dest;  // create a temp char pointer to hold
                                     // the passed address.
    if(NULL == dest){   // validate the parameters against Null pointer
        printf("Faild due to a null pointer !!\n");
    }
    else{
        while(length--){
            *temp_ptr++ = (unsigned char) value;
        }
    }
    return dest;
}
```

---

* memcpy function :-

- Description:-
  ⟶ copies 'n' characters from source object to the destination object.
- Declaration:-

```
void *My_memcpy(void *dest, const void *src, unsigned int char_count);
```

- Parameters:-
  dest: pointer to the destination object
  src: pointer to the source object
  char_count: Number of bytes to be copied

- return value :-

→ this function returns the pointer to the destination

- example :-

```
Char name1 [30] = "Mohammed Ahmed";
Char name2 [30];

printf("%s\n", name1); ——→ Mohammed Ahmed
printf("%s\n", name2); ——→ space فراغ
memcpy (name2, name1, 5);
printf("%s\n", name2); ——→ Moham ——→ 5 char
```

---

* implementation of memcpy function :-

```
Void *My_memcpy (void *dest, const void *src, unsigned int char_count) {
  /* create a temp char_ pointer to hold the passed address*/
  Char *temp_dest = (char *) dest;        → اختياري
  const char * temp_src = (const char *) src;
  /* validate the parameters against Null pointer */
  if ((Null == dest) || (Null == src)) {
      printf("faild due to a null pointer !!\n");
  }
  else {
      while (char_count) {
          /* copy byte by byte */
          *(temp_dest++) = *(temp_src++);
          --char_count;
      }
  }
  return dest;
}
```

---

* memcpy function :-

- Description :-

→ compares the first n bytes of memory area str1 and memory area str2

- Declaration :-

```
int memcpy (const void * str1, const void *str2, unsigned int n);
```

- parameters :-

  str1 :- this is the pointer to a block of memory
  str2 :- " " " " " " " " " "
  n :- This is the number of bytes to be compared (length)

- return value :-

  → if str1 < str2 then return value < 0 (=-1)
  → if str2 < str1 then return value > 0 (=1)
  → if str2 = str1 then return value = 0

- example

```
int str1[5] = {1,2,3,4,5};
    int str2[5] = {1,2,6,4,5};
      int ret = 0;
      ret = memcmp(str1, str2, 5);
      printf("%i\n", ret); ⟶ ret = -1 ⟶ (str2 > str1)
```

---

implementation of memcmp function :-

```
int my_memcmp(const void *str1, const void *str2, unsigned int Length){
    int return_state = 0; /* str1 = str2 */
    unsigned char *temp1 = str1;
    unsigned char *temp2 = str2;
    if((Null == str1) || (Null == str2)){
        printf("function faild due to a Null pointer!!\n");
    }
    else{
        if(temp1 == temp2){
            return_state = 0; /*indicates that 1st address is equal to 2nd address */
        }
        else{
            while(length){
                if(*temp1 != *temp2){
                    return_state = (*temp1 < *temp2) ? -1 : 1;
                }
                else{
                    length--;
                    temp1++;
                } temp2++;
    } } }
    return return_state; }
```

* memchr() function :-

- Description

→ searches for the first occurrence of the character c in the first n bytes of the string pointed to, by the argument str.

- Declaration:.

```
Void *memchr(const void *str, int c, unsigned int length);
```

- Parameters:

str ⟶ this is the pointer to a block of memory where the search is performed

c ⟶ this is the value to be passed as an int but the function performs a byte per byte search using the unsigned char conversion of this value

n ⟶ this is the number of bytes to be analyzed

- return value:

this function returns a pointer to the matching byte or Null if the character does not occur in the given memory area

* implementation of memchr function :-

```
Void *my_memchr(const void *str, int elem, unsigned int length){
    unsigned char *temp = str;
    if(Null == str){
        printf("function faild due to null pointer\n");
        temp = Null;
        return temp;
    }
    else{
        while(length){
            if(*temp == (unsigned char) elem){
                return temp;
            }
            else{
                length--;
                temp++;
            }
        }
    }
    return Null;
}
```

* memmove function:

→ Copies n characters from str2 to str1 (copies a block of memory from a location to another) using intermediate buffer.

→ memmove is a safer than memcpy because of overlapping memory blocks.

- Declaration :-

```
Void *memmove(Void *str1, const Void *str2, unsigned int length);
```

- return value:-

→ this function returns a pointer to the destination, which is str1.

---

* implemetation of memmove function.

```
Void * my_memmove(Void* str1, const Void *str2, unsigned int length){
    unsigned char * temp_dest = str1;
    unsigned char *temp_src = str2;
    unsigned char * temp = (unsigned char *) malloc (length);
    if((Null == str1) || (Null == str2)){
      printf("function failed due to null pointer\n");
       return Null;
    }
    else{
        while(length--){
            *(temp + length) = *(temp_src + length);
            *(temp_dest + length) = *(temp + length);
        }
        free(temp);
    }
    return str1;
}
```

* عند استخدام memmove() في main function يجب أن يكون str1 من نوع

Character array ←

وليس من نوع character pointer ولكن str2 يمكن أن يكون أحدهما

لأن

Character pointer → read only

Character array → read and write

*strcat function:-

→ appends (الحق) the string pointed to by src to the end of the string pointed to by dest.

- Declaration:-

```
char* strcat (char *dest, const char *src);
```

- Return value:-

→ this function returns a pointer to the resulting string dest

---

*implementation of strcat function

```
char *my_strcat (char * dest, const char *src) {
    if ((Null==dest) || (NULL== src)) {
        printf(" Null pointer passed ");
        return Null;
    }
    else {
        char *ptr = dest + strLen(dest);
        while (*src != '\0'){
            *ptr++ = *src++;
        }
    }
    return dest;
}
```

*using strcpy function to implement strcat function:-

```
char* my_strCat(char *dest, const char *src){
    strCpy((dest + strlen(dest)), src);
    return dest;
}
```

---

*another way

```
char *My_strcat(char *dest, const char *src){
 int i=0, j=0;
 while(dest[i]!='\0'){
    i++;
 }
 while( src[j] != '\0' ){
     dest[i] = src[j]
     i++;
     j++;
 }
 dest[i] = '\0'; }  return dest;
```

* Function Pointers concepts (pointer to function):-

→ a function pointer: stores the address of a function
→ in the program whenever required we can invoke the pointed function using the function pointer
→ using the function pointer, we can provide the run time binding in C Programming which resolves the many problems.
→ we can provide a SW module with a capability to call a function from upper layer (call back)

* Declaration:

```
function_return_type (*fun_ptr_name)(fun_argument list);
```

ex:
→ it can point to a function which takes an int as an argument and return nothing.

→ `Void (*ptr)(int);`

→ `Void (*ptr)(int Var);`

* Braces have a lot of importance when you declare a pointer to functio

ex:

```
Void (*ptr)(const char *);
Void *ptr(const char *);
```

→ when we remove the braces, then the meaning will be change, it becomes the declaration of a function that takes the const char pointer as argument and return a void pointer.

→ A function pointer must have the same signature to the function that it is pointing to, in a simple word the function pointer and its pointed functior should be the same in the parameters list and return type.

→ A function pointer is initialized to the address of a function
→ you can use the address operator (&) with function name or you can use directly function name (function name also represents the begining address of the function)

ex :

```
Void print_Mohammed (void);
Void (*ptrfun)(void) = Null;

int main(){
 print_Mohammed();
 ptrfun = print_Mohammed;
 ptrfun();
 return 0;
}
Void print_Mohammed (void){
    printf("Mohammed");
}
```

اسم fun هو عنوانها

---

*initialization of array of function pointer:-

```
Void print_Ahmed (void);
Void print_Mohammed (void);
Void (*ptr[2])(void) = {print_Ahmed, print_Mohammed};
```

OR

```
ptr[0] = print_Ahmed;
ptr[1] = print_Mohammed;

int main(){
    ptr[0]();  ---------> Ahmed
    ptr[1]();  ---------> Mohammed
    return 0;
}
Void print_Ahmed(void){
    printf("Ahmed");
}
Void print_Mohammed(void){
    printf("Mohammed");
}
```

*function pointer as an argument:-

→ we cannot pass the function as an argument to another function, But we can pass the reference of a function as a parameter by using a function pointer

ex:

```
Void print_summing_1 ( int num1, int num2){
     printf("summing_1 =%i \n", (num1+num2));
}
Void print_summing_2 ( int num1, int num2) {
     printf(" summing_2 = %i \n", (num1+num2));
}
Void print_summing ( int num1, int num2,
                     Void (*ptr_sum)(int, int)){
     ptr_sum (2, 3);
}
int main ( ) {
     int choise =0;
     printf(" please enter choise method : ");
     scanf("%i", &choise);
     if(choise ==1){
          print_summing ( 2, 3, print_summing_1) ;
     }
     else if(choise ==2){
          printf_summing (2, 3, print_summing_2);
     }
     else{
        printf(" invalid choise");
     }
```

if choise = 1	choise = 2
summing_1 = 5	summing_2 = 5

* Binding concept :-

→ Binding is done for each variable and functions

→ for function:- it means that matching the call with the right function definition by the compiler, and it takes place either at compile time or at run time

→ there are 2 kinds of binding : "static binding" & "Dynamic binding"

① static binding / compile time binding / Early binding

- All the information necessary in order to perform that association is available at compile time
- association happening at compile time
- Happens by default → Normal function call
- makes our program run faster

② Dynamic binding / Runtime binding / Late binding

- association happening during run-time
- All the information necessary in order to perform that association is not available at compile time, it is available at run-time
- makes our program a little bit slower
- Very flexible → Decide which function definition to invoke at run time.

---

* Modules Software Design approach:-

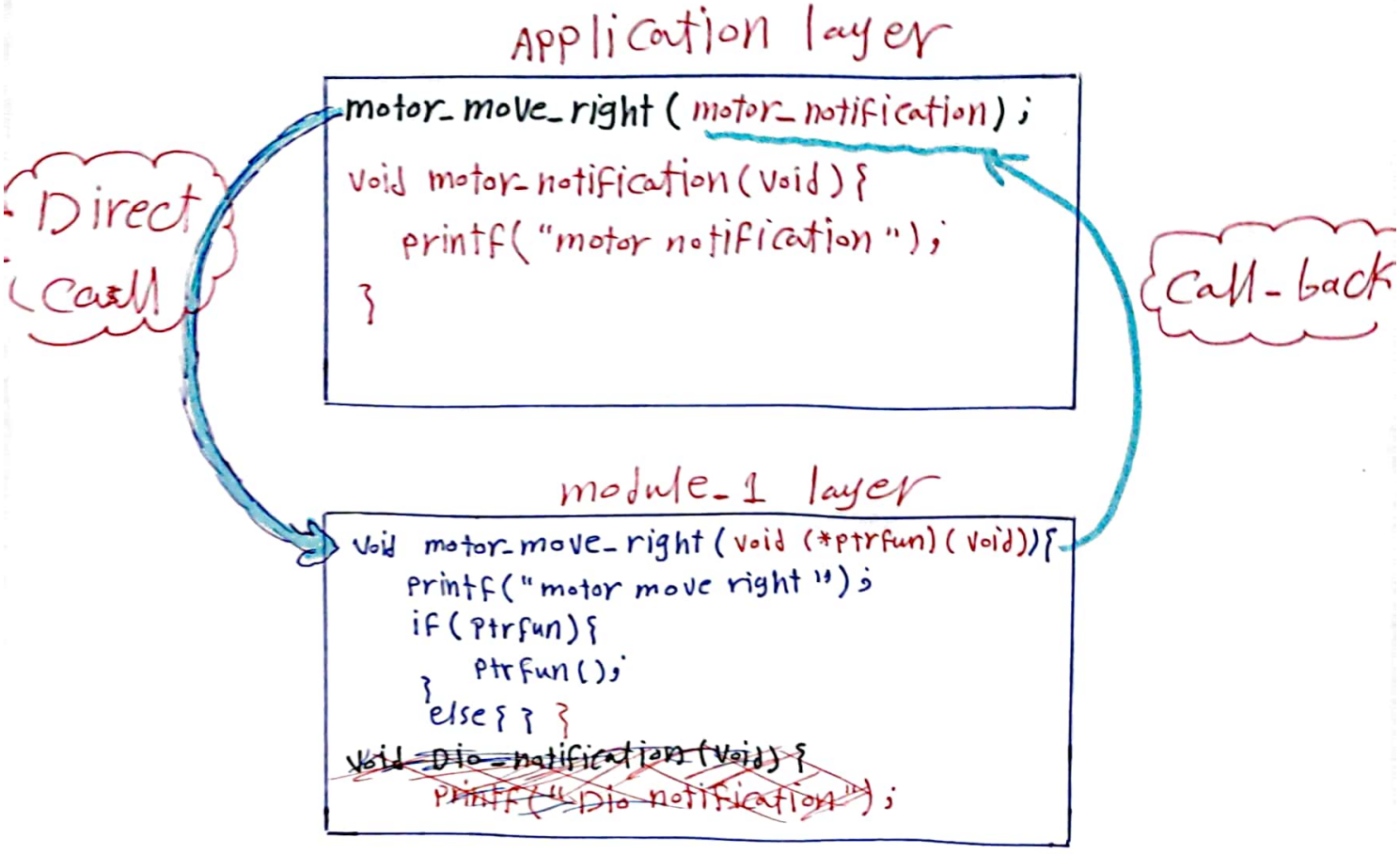
* هو طريقة لتصميم الـ software حيث أنه يقوم بتقسيم الـ functionality الخاصة بالمشروع إلى عدة modules حيث أن كل module يكون independent على Module أخر

* قد يستخدم أحد modules الـ function الـ module أخر ولكن إذا تم تعديل أو إضافة أو تغيير module فإنه لا يؤثر على أي module آخر موجود في المشروع أي أنه interchangeable "قابل للتعديل"

* يتم تقسيم Modules إلى عدة طبقات بحيث أن أي Module في طبقة أعلى من الـ module أخر فيكون مسموح له أن يعمل direct call لأي function في الطبقة الأقل وليس العكس

* ولكن يتم السماح للطبقة الأقل بعمل ما يسمى بـ Call-back functions لأي function في الطبقة الأعلى عن طريق استخدام function pointer

* كل طبقة تحتوى على عدد من modules وتسمى الطبقة بـ layer
* كل layer تفصل بين ال layer اللى قبلها واللى بعدها
* تكون طبقة application أعلى طبقة ← main.c



* Call-back function

→ we can implement the call-back function using the function pointer

→ A callback function is any executable code that is passed as an argument to other code that is expected to call back (execute) the argument at a given time

→ this execution may be immediate as in a synchronous call back, or it might happen at a later times as in an asynchronous call back.

- A synchronous call back: is invoked before a function returns, that is, while the API receiving the call back remains on the stack, this means that your application will not execute any thing untill a response is returned by the API, on the contrary there is an Asynchronous call back is not blocking or wait for the API call to return.

structure

- structure : is a user defined data type which used to create and store a package of data , related to each other in one place

- For example : an entity (student) may have its name (string) , id number (int) an marks (float)

*syntax :

```
struct struct-name {
    Data-type member1-name;
    Data-type member2-name;
    .....
};
```

→ structure Body

*عند تعريف struct لايتم حجز مساحة فى الذاكرة حتى يتم تعريف object من هذا struct

ex :

```
struct student_info {
    char std-name [30];
    float std-degree;
    unsigned short std-id;
} Ali , ramy , ..... ;

struct student_info sayed;
```

sayed → struct object

* هناك طريقتان لعمل object من struct وذلك عند وجود اسم لل struct عند تعريفه

* فى لغة السى من الاختيارى أن تضع اسم أو لا لل struct

*إذا لم يتم تعريف اسم لل struct فهناك طريقة واحدة لتعريف object لل struct

ex :

```
struct {
    ____
    ____
} Hany , ..... ;
```

→ struct object

* Hany is an object from struct and it is following the scope rule

* A struct does not contain an instance of it self (incomplete type) but contain a pointer to itself

* عند عمل struct لـ array من النوع member

```
struct name{
    char arr[30]; ⟶ acceptable fixed array
    char _arr[]; ⟶ not acceptable flexible array
    int id;
}
```

```
Struct name{
    char arr[30]; ⟶ acceptable
    int id
    char _arr[]; ⟶ valid in cgg and c11
}                  ↳ must be the last member
```

* يسمح بعمل struct بداخل fixed array من النوع just one member
ولكن لا يسمح بعمل ذلك مع flexible array

* We cannot initialize the member of the struc at the time of the struct declaration, because there is no memory is allocated to the members at the time of declaration.

* initialization of structure:

* method 1:-

```
struct student_info Ali = {"Ali", 3.4, 10}; ⟶
printf("%i\n", sizeof(Ali)); ⟶ 40
```

* يجب الحفاظ على الترتيب طبقاً لـ struct declaration

* دائماً حجم struct يكون أكبر من أو يساوي مجموع data-types بداخل struct وذلك يعود لـ struct padding

* method2 :-

```
struct student_info Ahmed;
Ahmed.std_name[0] = 'A';
Ahmed.std_name[1] = 'h'; .....
```

OR

```
strcpy(Ahmed.std_name, "Ahmed");
Ahmed.std_degree = 3.4;
Ahmed.std_id = 10;
```

* method 3 :- designated initializer:

→ we can initialize the members of the structure in any order using (.) dot and member name (.member) and this method valid in (c99, c11

ex:

```
struct student Ali = {
    .std_name = "Ali",   → comma
    .std_id = 10,
    .std_degree = 3.4 };
```

---

ex:

```
struct student {
    char std_name[20];
    float std_age;
    int  std_id;
};

struct student ahmed = { "Ahmed ", 15.5 , 15 };
struct student ali;
memset ( ali.std_name , '\0' , 20);   → to avoid having a garbage value
memcpy ( ali.std_name , " Ali sayed ", 9);   → 9 : Ali sayed length
```

OR → strcpy (ali.std_name, "Ali sayed");

```
ali.std_id = 20;
ali.std_age = 11.5;
printf ("Ahmed name :%s\n", ahmed.std_name);   → Ahmed
printf ("Ahmed age =%0.3f\n", ahmed.std_age);   → 15.500
printf ("Ali id =%i\n", ali.std_id);   → 20
```

---

typedef introduction

→ typedef : defines a new name for existing types and not introduce a new type

→ it is the (partial storage-class specifier) compiler directive mainly use with user defined data types (struct-union-enum) to

reduce their complexity and increase code readability and portability.

→ we can use typedef to create shorter or more meaningful names for types already defined by C

* syntax:

typedef type New_type_name;

*Note: A typedef creats synonyms (مرادفات) or a new name for existing types it doesnot create a new types

example:-

typedef unsigned int uint32;
uint32 var = 20;

uint32 ← new name for (unsigned int)

* use of typedef with a struct:.

→ when we use typedef with structure then it creates the alias of the structure

→ there is no need to write struct keyword every time with a variable declaration that means typedef save extra keystroke and make the code cleaner and readable

ex:

typedef struct student { ← اختياري

} student_t; ← typedef للدلالة على أنها
اختياري مع typedef

* student_t : it is not an object from struct but it is called an alias of structure and no memory reservation is taken.

student_t Ahmed = { "Ahmed", 3.14, 10 };
studen_t Ali;
Ali.std_id = 20;

* accessing structure members using pointers.

→ when we have an object from a struct type we use a dot (.) to access struct members.

→ when we have a pointer points to a struct object we use an arrow (->) operator to access members.

→ the pointer object and the struct object must be from the same struct type.

ex:

```
typedef struct {
        char std_name[20];
        float std_degree;
        int std_id;
} student_t;

student_t Ahmed = {"Ahmed", 3.4, 10};
student_t *ptr_Ahmed = Null;
ptr_Ahmed = &Ahmed;
printf("%s\n", (ptr_Ahmed -> std_name)); ——→ Ahmed
```

* Dynamic memory allocation for a struct object:-

```
student_t *ptr_ali = Null
ptr_ali = (student_t *) malloc (sizeof(student_t));
if (ptr_ali != Null){
        strcpy (ptr_ali -> std_name, "Ali");
        ptr_ali -> std_degree = 3.4;
        ptr_ali -> std_id = 10;
}
else {
        ═══
}
printf("%s\n", ptr_ali -> std_name); ——→ Ali
Free (ptr_ali);
```

* passing structure object to a function as a parameter.

[1] pass by reference → recommended to avoid multiple copies of large data, and if the struct object is too large, so you will save your stack memory if you use passing by reference method

ex:-

```
typedef struct {
        char std_name[30];
        float std_degree;
        unsigned short std_id;
} std_t;

void get_std_data (std_t *student) {
    printf("enter std name : ");
     gets(&(student -> std_name));
     printf("enter std degree");
     scanf("%F", &(student -> std_degree)); ..... etc
}
void print_std_data (std_t *ptr) {
    printf("std name : %s \n", ptr-> std_name);
    printf("std degree : %0.2F \n", ptr -> std_degree); ....
}

int main() {
    std_t Ahmed;
    get_std_data (&Ahmed);
    print_std_data (&Ahmed);
    return 0;
}
```

output:

```
Enter std name : Ahmed Ali
enter std degree : 3.4
enter std id : 10
std name : Ahmed Ali
std degree : 3.40
std id : 10
```

[2] pass by Value "not recommended"

slides 220

*using structure to return multiple values from function :-

```
typedef struct {
        int sum;
        int product;
} my_struct_t;

my_struct_t  get_sum_and_product (int a, int b){
        my_struct_t temp;
        temp.sum = a+b;
        temp.product = a*b;
        return temp;
}

int main(){
        int num1, num2;
        my_struct_t result;
        printf("enter two integers\n");
        scanf("%d%d", &num1, &num2);
        result = get-sum-and_product (num1, num2);
        printf("sum =%d\n product =%d", result.sum, result.product);
        return 0;
}
```

out put :-

enter two integer

3 7

sum = 10

product = 21

* structure padding with alignment :-

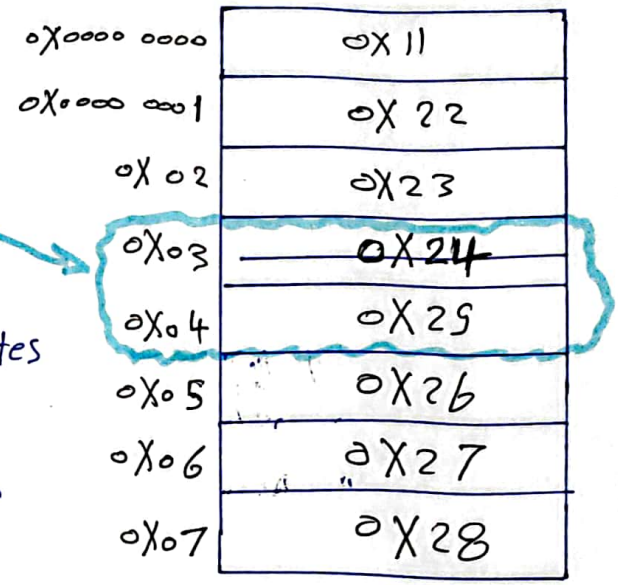
→ in real-world processor, does not read or write the memory byte by byte but actually, for performance reason, it accesses the memory in the formats like 2, 4, 8, 16 and 32 byte of chunks (قطع) at a time

ex: processor may be able to access the memory only at four bytes boundaries

* if the processor needs to read address 0x03 and 0x04

→ the processor has to access the memory location with address 0 and read four consecutive (متتالية) bytes (0x00: 0x03)

→ Next, it has to use shift operator to separate the content of address 0x03 from the other three bytes

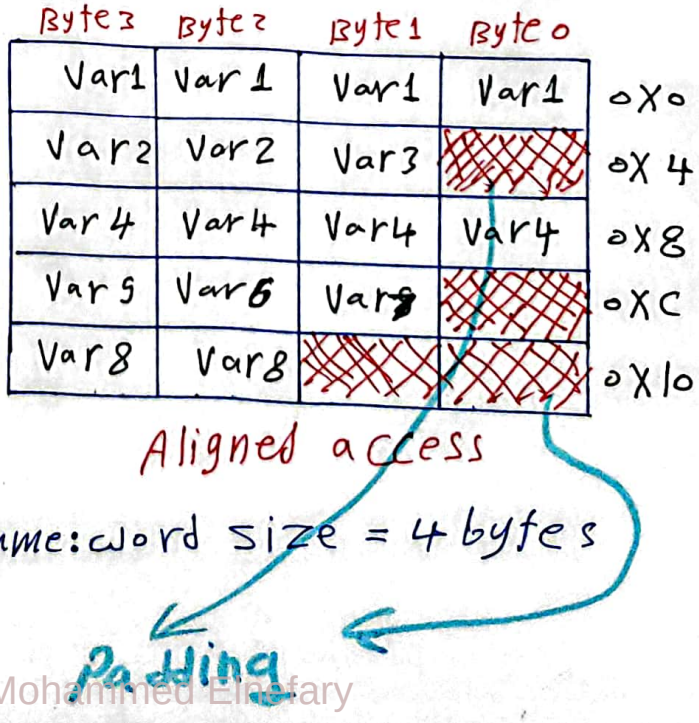


→ similarly, the processor can access address 4 and read another four bytes then use shift operator to separate the desired byte from the other 3 bytes

---

* memory Alignment and padding

```
int  Var1 ; // 4 bytes → word
short Var2; // 2 bytes
char  Var3; // 1 byte
int  Var4; // 4 bytes → word
char Var5; // 1 byte
char Var6; // 1 byte
char Var7; // 1 byte
short Var8; // 2 bytes
```



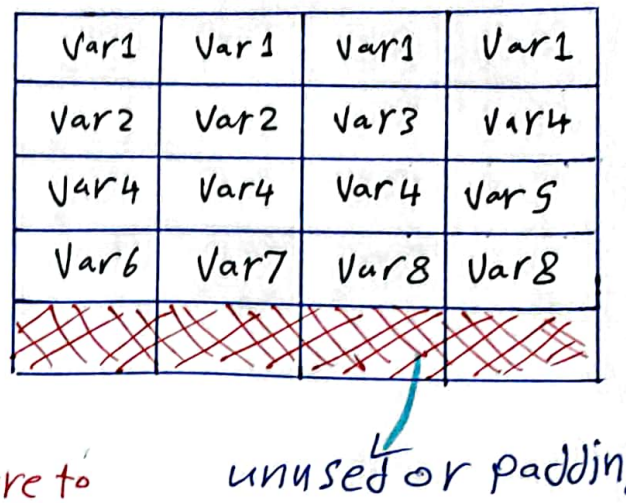
* when memory is aligned then the processor easily fetches the data from the memory

→ the processor takes one cycle to access the aligned data

→ when memory is not aligned then the processor takes some extra ticks to access the unaligned memory

→ the processor takes one cycle or more to access the unaligned data

unaligned access



---

* structure Padding :.

- when you create the structure or union then compiler inserts some extra bytes between the members of structure for the alignment
- these extr unused bytes are called padding bytes and this technique is called structure padding.
- Padding increase the performance of the processor at the penalty of memory على حساب الذاكرة
- in struct or union data members aligned as persize of the highest bytes member to the lowest one to prevent the penalty of memory (in order of largest size to smallest)

→ in order to rearrang the structure member in the order of largest size to smallest size to avoid structure Padding there is another way to do this.

*Avoid structure Padding (structure packing)

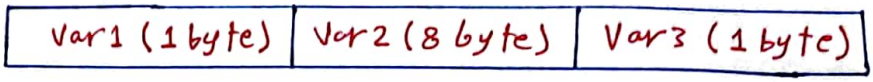
→ we can avoid structure padding by using the pragma pack (#Pragma pack(1)) or attribute (__attribute__((__Packed__)))

ex:

```
#Pragma pack(1)        // compiler dependent
    typedef struct{
        char var1;
        double var2;
        char var3;
    }test_t;
#Pragma pack(1)
test_t test;
printf("size of double = %i\n", sizeof(double));
printf("size of test = %i\n", sizeof(test));
```

output:-

size of double = 8
size of test = 10



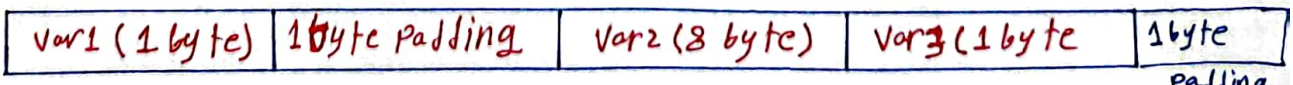
---

ex:

```
#Pragma pack(2)
    typedef struct{
        char var1;
        double var2;
        char var3;
    }test_t;
#Pragma pack(2)
test_t test;
printf("size of double = %i\n", sizeof(double));
printf("size of test = %i\n", sizeof(test));
```

output:

sizeof double = 8
size of test = 12



*Conclusion:-

- memory alignment increase the performance of processor
- CPU performs better with aligned data as compared to the unaligned data because some processor takes an extra cycle to access the unaligned data.
- When we create the structure, union then we have to rearrange the members (largest to smallest) in a careful way for the better performance of the program.
- the size of structure must be divided by the size of the largest member in the structure, to force the alignment when we create array of this structure
- we can avoid structure padding by structure packing using the preprocessor directive provided by the compiler #pragma pack (compiler dependent) or using the ⟶ Attribute packed

ex:

```
typedef struct __attribute__((__packed__)){
    unsigned char var1;
    unsigned int var2;
    unsigned char var3;
} test_t;

test_t test;
printf("size of test = %i\n", sizeof(test));
```

output:-

size of test = 6

Structure and Bit field

* the bit field allows the packing of data in a structure to prevent the wastage of memory
* A bit_field must be a member of a structure or union. it defines how long in bits the field is to be

* syntax of bit filed:

```
struct register {
    data_type  bit_field_name : size_in_bits;
    .....
};
```

ex:

```
typedef struct {
    unsigned char var1 : 4;
    unsigned char var2 : 4;
} test_t;
  test_t test;
printf("%i\n", sizeof(test));
```

size in bits

→ 1 Byte

if (unsigned char var2 : 5 ;) then
the size of test will be 2 byte

* the (size in bits) specifies the width of the field in bits and must be a non negative integer value.

---

union with structure and bit field

union : is used to store different data types in the same memory location in union, we can define many members as per the requirement, but her we need to remember that stored value is shared by all members

• Note:

A union type declaration is a template only

→ there is no memory reserved for the union untill a variable is declared

→ the size of union equal to the size of the largest member (+required Padding)

→ only one member can be accessed at a time with union

→ ~~this~~ with union, we can initilize only the first member of union and the members that follow it increase by 1

→ union provides an efficient way of using the same memory location but carefully

ex:

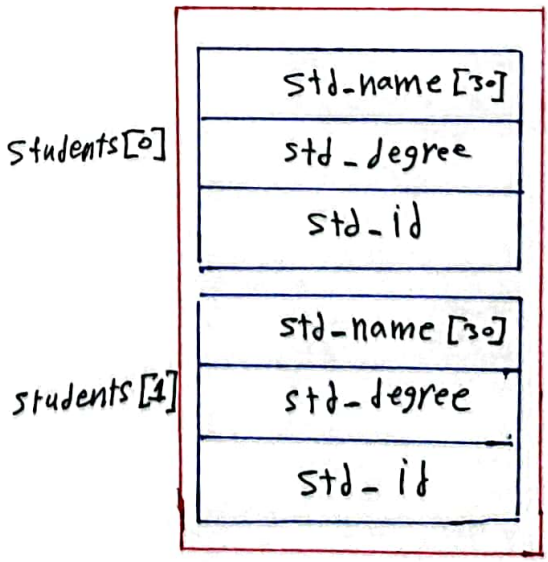
```
typedef union {
        struct{
           uint8 pin0 : 1;
           uint8 pin1 : 1;
           uint8 pin2 : 1;
           uint8 pin3 : 1;
           uint8 pin4 : 1;
           uint8 pin5 : 1;
           uint8 pin6 : 1;
           uint8 pin7 : 1;
        };
        uint8 AllPin;
    } Port_t;
```

```
Port_t Port;
Port.pin0 = 1 ;   → Port=1   /* 0000 0001 */
Port.pin1 = 1 ;   → Port=3   /* 0000 0011 */
Port.pin2 = 1 ;   → Port=7   /* 0000 0111 */
Port.AllPin = 0 ; → Port=0   /* 0000 0000 */
```

All Pin variable can write on all port pin because the shared memory location between all member.

Array of structure

```
typedef struct{
    char std_name[30];
    float std_degree;
    short std_id;
} student_t;
student_t students[2];
```

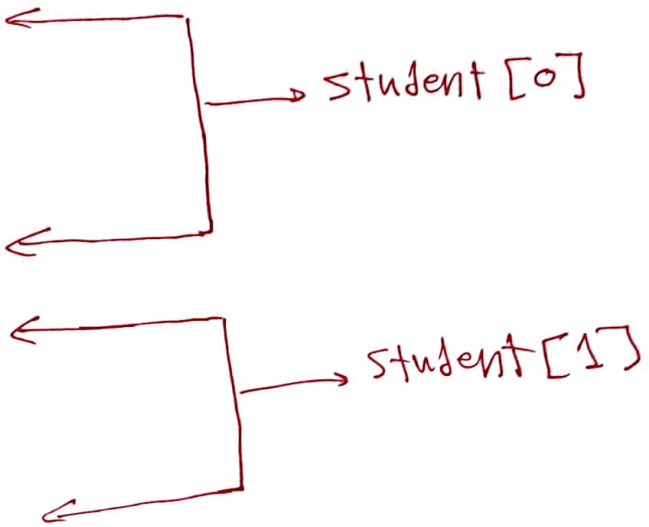


* initialization:

```
strcpy(student[0].std_name, "Mohamed Ahmed");
student[0].std_degree = 3.5;
....
```

OR:

```
student_t students[2] = {
    {
        "Ahmed ALi",
        3.5,
        10
    },
    {
        "samy Adel",
        3.5,
        4
    }
};
```



---

* Passing the 1-D array of struct to a function :-

```
Void get_std_info (studen_t std_info[], unsigned int arr_size){
    unsigned char std_info_C = 0;
    for( std_info_C = 0 ; std_info_C < arr_size; std_info_C++) {
        printf("Please enter student [%i] name \n", std_info_C);
        gets( std_info [std_info_C]. std_name);
        printf("Please enter student [%i] degree \n", std_info_C);
        scanf("%f", &(std_info [std_info_C]. std_degree));
        ..... .....
        FFlush(stdin);
    }
}
```

---

*Array of pointers to structure:-

```
typedef struct {
        char std_name [30];
        float std_degree;
        int   std_id;
} std_t;
std_t  student [2];          ⟶ array of struct
std_t *std_ptr[2];           ⟶ array of pointer to struct
strcpy ( student [0]. std_name, "Mohammed ");
student [0]. std_id = 10;  ..... and so on
std_ptr [0] = & student [0];
std_ptr [1] = & student [1];
printf ( "%s \n ", std_ptr -> std_name);
printf ( "%f \n ", std_ptr -> std_degree);  .....
```

output

```
Mohammed
10  .....
```

---

*struct to struct (struct object inside another struct)

```
typedef struct {
        char Father_name [30];
        char mother_name [30];
        char home_address [50];
} std_details_t;

typedef struct {
        std_details_t std_details;
        char std_name [30];
        float std_degree;
        short std_id;
} std_t;

std_t Ahmed;
strcpy (Ahmed. std_name , " Ahmed Ali ");

strcpy (Ahmed. std_details. father_name, "Ali mohammed ");
Ahmed. std_degree = 3.4;

printf("%s \n", Ahmed. std_details. father_name); ⟶ Ali mohammed
```

* Pointer to struct type in the same struct type

```
typedef struct student {
    struct student *std_friend;
    char std_name[30];
    std_details_t std_details;
    ......
} studen_t;
```

* struct name must be found there

* using typedef optional

studen_t → alias of struct

```
typedef struct {
    char father_name[30];
    .....
} std_details_t
```

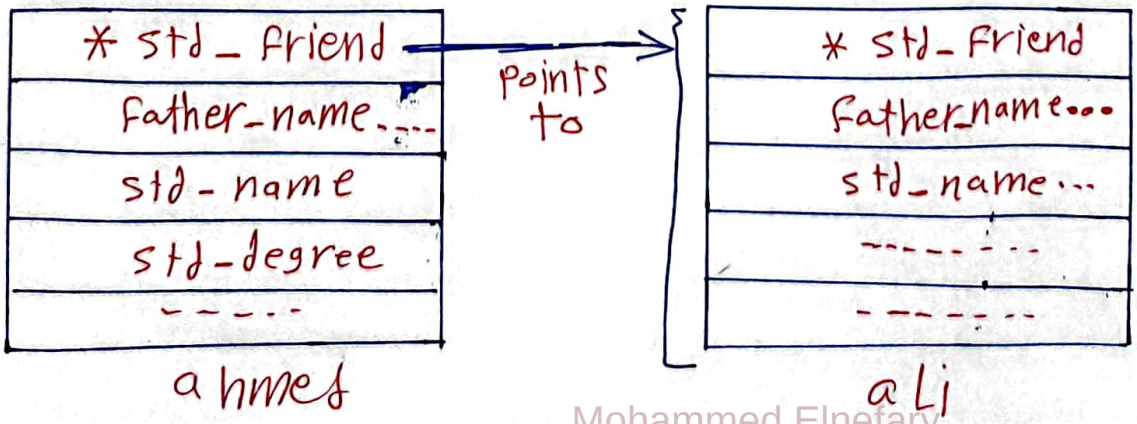
```
student_t ahmed , ali;

ahmed.std_friend = &ali;
printf("address of ali = 0X%X \n", &ali);  ⟶ 0X 40 7A 20
printf("0X%X \n", ahmed.std_friend);  ⟶ 0X 40 7A 20

strcpy(ali.std_details.father_name , "samy");

printf("%s \n", ahmed.std_friend -> std_details.father_name);
```

output:

address of ali 0X 40 7A 20

0X 40 7A 20 ← access address of ali through ahmed

samy ← access ali details through ahmed



* Struct and usage of function pointer in C :-

→ We cannot create a member function in the structure but with the help of the pointer to a function, we can store the address of the function

→ A user can use this structure to store the address of a function using the function pointer as per the requirements and called this function whenever required in the program.

main.C

```
#include "gpio.h"
gpio_t  gpio_1;
Void gpio_notification (Void) {
    printf("gpio performed request \n");
}
int main( ) {
   gpio_1.pin_number = 11;
   gpio.ptr = gpio_notification;
   gpio_write_5Volt (&gpio_1);
   gpio_write_0Volt (&gpio_1);
   return 0;
}
```

GPIO.C

```
#include "gpio.h"
Void gpio_write_5Volt(gpio_t *gpio_obj) {
    printf("gpio pin number %i is (ON) \n", gpio_obj->pin_number);
     gpio_obj -> ptr ( );
}
```

gpio.h

```
typedef struct {
        int pin_number;
        Void (*ptr)(void);
} gpio_t;
Void gpio_write_5Volt(gpio_t *gpio_obj);
Void gpio_write_0Volt(gpio_t *gpio_obj);
```

ex:

```
typedef struct {
        Char std_name[30];
        int  std_id;
        Void (*ptr_std_info)(void);
} std_t;

std_t ahmed;
Void print_std_info(void);
int main(){
    strcpy(ahmed.std_name, "samy Ali");
    ahmed.std_id = 11;
    ahmed.ptr_std_info = print_std_info;
    ahmed.ptr_std_info();
    return 0;
}
void print_std_info(void){
    printf("student id :%i\n", ahmed.std_id);
}
```

output:

```
student id = 11
```

typedef

*A typedef defines a new name for existing types and doesnot introduce a new type
→ it is the compiler directive mainly use with user defined data types (struct - union - enum) to reduce their complexity and increase code readability and portability

* syntax :-

```
typedef   data-type   new-data-type-name;
```

ex:-

```
typedef   unsigned int   uint32;
```

* the problem of code readability :-

```
uint32 Ret_fun1 (void);
uint32 Ret_fun2 (void);
uint32 Ret_fun3 (void);
uint32 Ret_fun4 (void);

uint32 (*ptr[4]) (void) = { Ret_fun1, Ret_fun2,
                            Ret_fun3, Ret_fun4 };
/* ptr is an array of 4 element, each element is a pointer to
   function that takes void and return uint32 */
uint32 (*(* my_fun (void)) [4] (void);
/* my_fun is a function that returns a pointer to an array whose
   size is 4 and contains pointers to function returning an int */

int main ( ) {
   uint32 (*(* mainptr) [4] ) (void) = my_fun ( );
   printf ("%i\n", (*mainptr)[0]());
   printf ("%i \n", (*mainptr)[1]());
   printf ("%i\n", ((*ptr[0])[0])());
   printf ("%i\n", ((*ptr[1])[1])());
   return 0;
}
uint32 Ret_fun1 (void){
   return 11;
}
```

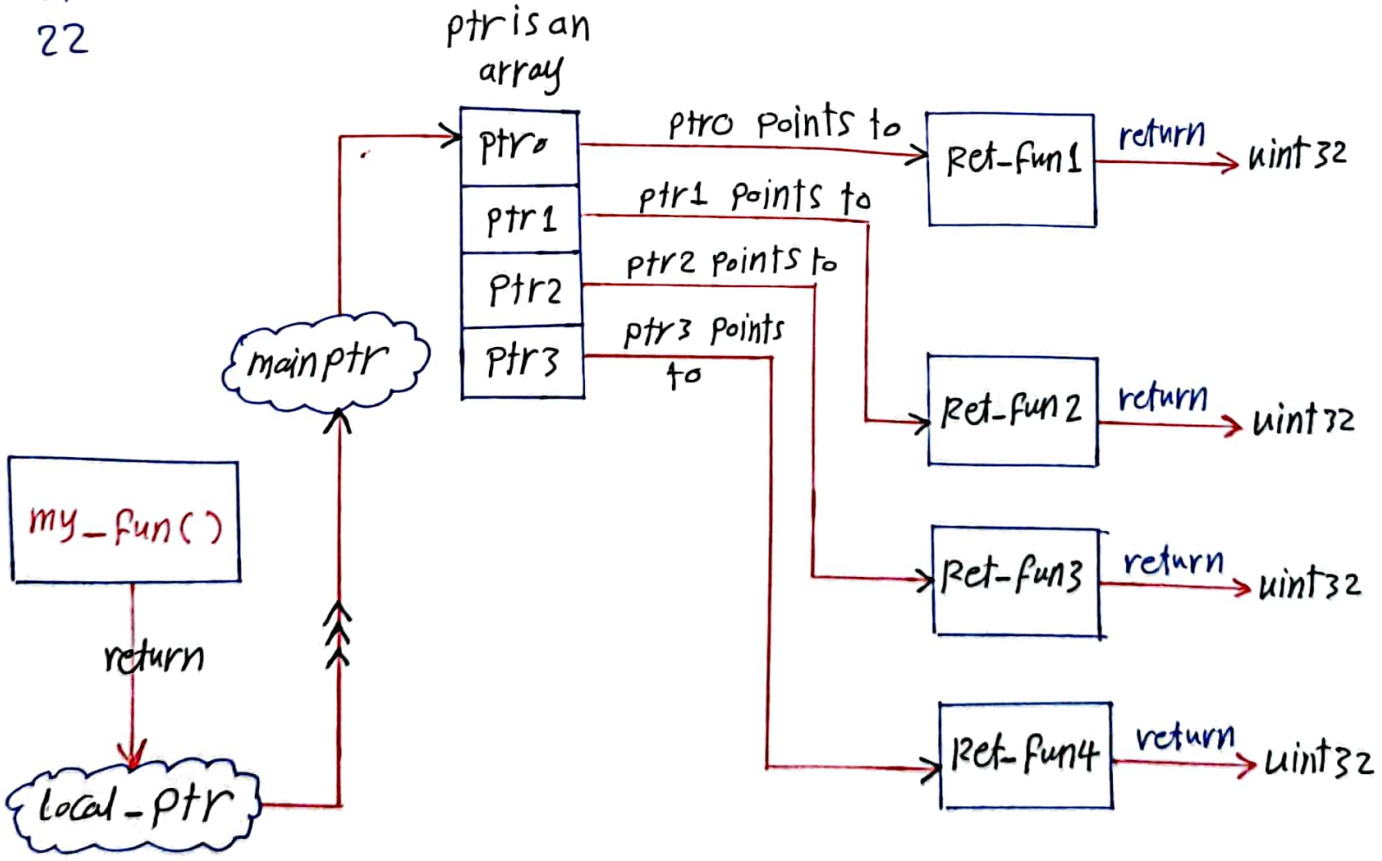
```
uint32 Ret_fun2 (void){
        return 22;
}
uint32 Ret_fun3 (void){
        return 33;
}
uint32 Ret_fun4 (void){
        return 44;
}
uint32 (*(*my_fun(void))[4](void){
        uint32 (*(*local_ptr)(void)) = &ptr;
        return local_ptr;
}
```

[4]

uint32 (*(*local-ptr)[4](void) = &ptr

output:-

11
22
11
22



* using typedef to increase code readability :-

```
uint32 Ret_fun1 (void);
uint32 Ret_fun2 (void);
uint32 Ret_fun3 (void);
uint32 Ret_fun4 (void);

typedef uint32 (* fun_ptr) (void);
/* fun_ptr is the type of pointer to function returning uint32 */

typedef fun_ptr (*ptr_arr_4_PF) [4];
/* ptr_arr_4_PF is the type of pointer to array whose size is 4 and
   which contains (fun_ptr) which is pointers to function returning
   uint32 */

fun_ptr my_fun (void);
/* my function is a function returning a pointer to an array whose
   size is 4 and contains pointers to function returning uint32 */

fun_ptr my_fun (void) {
    ptr_arr_4_PF local_ptr = &ptr;
    return local_ptr;
}

fun_ptr ptr_Ret_fun [4] = { Ret_fun1 , Ret_fun2
                            Ret_fun3, Ret_fun4 };
/* ptr_Ret_fun : is an array of 4 element each element is a pointer
   to function that takes void and returns uint32 */

int main() {
    ptr_arr_4_PF [redacted] local_main_ptr = my_fun();
    printf("%i\n", (*local_main_ptr)[0]() );  ---> 11
    printf("%i\n", (*local_main_ptr)[1]() );  ---> 22
    printf("%i\n", (*ptr_Ret_fun[0])() );     ---> 11
    printf("%i\n", (*ptr_Ret_fun[1])() );     ---> 22
```

union

- A union is used to store different data types in the same memory location
- union is user define data type or non-primitive data type
- in union we can define many members as per the requirement but here we need to remember that stored value is shared by all members.
- it means a union provides an efficient way of using the same memory location but carefully.

* syntax :-

```
union union_name {
    /* members */
};
```

* Note : A union type declaration is a template only.
  → there is no memory reserved for the union untill a variable is declared.

ex :

```
union std_info {
    int std_id;
    float std_deg;
};
```

/* std_info is a union user defined data type and has two member variables */

→ Create a variable from union type :-

1

```
union std_info ahmed;
```

2

```
union std_info {
    ____
    ____
} ahmed, ali;
```

* using typedef with union as the same as using it with structure and we use the same way to accessing union member.

→ size of union is sufficient to contain the largest of its members (plus required padding)

→ it means union follows the alignment as per the largest member.

→ the main purpose of a union is to save memory because union members share the same memory location.

→ a union has at most one "active" member at each moment.

ex:
```
typedef union
{
    uint32 std_id;
    float32 std_degree;
} student_t;
```

```
student_t ahmed;
student_t *ptr = &ahmed;
ahmed.std_id = 44;
printf("ahmed id = %i\n", ahmed.std_id);   ⟵ ahmed id = 44
ptr -> std_id = 99;
printf("ahmed id = %i\n", ptr->std_id);    ⟵ ahmed id = 99
(*ptr).std_id);
ahmed.std_id);
```

---

* a union with no tag is called an anonymous union:

→ anonymous union is introduced in C11 not supported by the C99 or older compiler

ex:

`union { int num; };`	`union { };`	`union name { };`
anonymous union	anonymous union	empty union

Enum

* Enum is a user-defined data type and it consists of set of named constant integer
* we use enum to increase the code readability and it is easy to debug the code as compared to macro
* it follows the scope rule
* the compiler automatically assigns the value to its member constant

→ A variable of enumeration type stores one of the values of the enumeration list defined by that type

* Syntax:-

```
enum enum_tag {
    enum list,
    ...
};
```

* enum members may contain a duplicate value
* By default, all member of the enum would be the integer constant
* the size of enum equal to the size of integer
* By default the compiler always assign 0 to the first constant member of enum list

ex:

```
enum Days {
    sat,    → default value = 0
    sun,    → default value = 1
    mon     → default value = 2
};

enum Days day = sat;
printf("%i \n", day); → 0
day = sun;
printf("%i \n", day); → 1 .....
```

* if we assigned any value to any member of the enum List the next element will be Increase by one

ex:

```
enum Days {
    sat,          ⟶ 0
    sun,          ⟶ 1
    mon = 11,     ⟶ 11
    tue,          ⟶ 12
    wen = 20,     ⟶ 20
    thu           ⟶ 21
    fri           ⟶ 22
};

enum Days day;          // OR int day but not recommended
for (day = sat; day < mon; day++) {
    printf ("%i ", day);
}
```

output :-

0 12 3 4 5 6 7 8 9 10

---

* increasing readability :-

- using enum with switch case & function parameter :

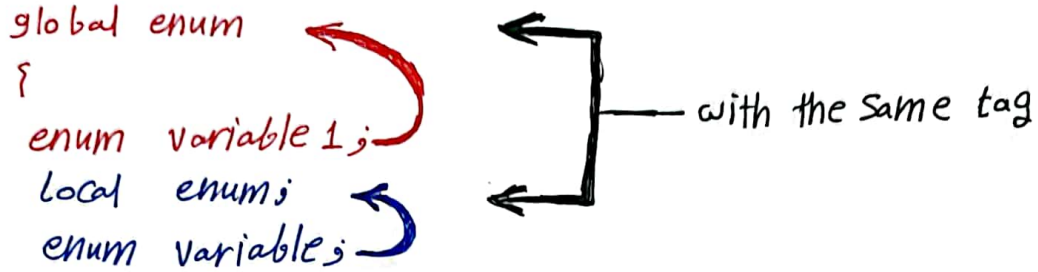
ex

```
enum Days { sat, sun, mon ... };
Void print_day_name (enum days day) {
    switch (day) {
        case sat : printf ("saturday "); break;
        case sun : printf ("sunday "); break;
        .....
    }
}
```

print_day_name (sat); ⟶ saturday

* يمكن تعريف 2-enums ويكونوا بنفس الاسم ولكن with different scope بحيث مثلاً يكون أحدهم global والآخر local وعند تعريف متغيرات منهم يكون المتغير تابع لأقرب enum

ex:



* إذا تم تعريف two enum with same tag ولهم نفس scope then the compiler through "re-declaration" error.

* إذا كان لهم نفس scope إذا غير مسموح بعمل Variables بنفس الاسم بداخل two-enums

---

Enum VS macro (#define)

- An Enum increase the code readability and easy to debug in comparison to the micro
- the macro is replaced in the pre-processing time (before starting the compiling)
- All elements of enum grouped together which is not possible with macro
- enum in C define new type but macro does not define a new type
- enum follows scope rules and compiler automatic assigns the value to its member constant
- enum in C type is an integer but the macro type can be any type
- we can use typedef with enum then it increase code readability

---

Dynamic memory allocation

* we need DMA when we don't know the worst case requirements for memory then, it is impossible to statically allocate the necessary memory, because we donot know how much we will need
* at run time, we can create, resize and deallocate the memory based on our requirement using DMA with Pointers

DMA :→ memory management functions

→ there are alot of library functions (malloc, calloc .....) which are used to allocate memory dynamically

→ one of the problems with dynamically allocated memory is that: it is not destroyed by the compiler itself that means it is the responsibility of the user to deallocate the allocated memory

→ when we allocate the memory using the memory management functions they return a pointer to the allocated memory block that points to the start address of the memory block

→ if there is no space available, the function will return a null pointer.

---

1 malloc for dynamic memory allocation

syntax:

```
void *malloc (size_t size);
```

* the malloc function allocates space for an object whose size is specified by (size_t size) "in bytes" in memory at run time
* if there is no space available, the malloc function return Null pointer
* after using malloc function we should first check the validation of the pointer, to make sure that memory space has been reserved.
* Standard Library function, are defined in <stdlib.h>

ex:

```
Char *ptr = Null;
ptr = (char*) malloc (30); OR malloc ((6 * sizeof(int)));
```

* يجب رابطة arithmatic pointer حتى يتم الاحتفاظ بعنوان first reserved space

if → ptr = 0xcg1460

∴ (ptr+1) = 0xcg1464 → and ptr = 0xcg1460

∴ (ptr++) = 0xcg1464 → and ptr = 0xcg1464

* to save the first address we can use (ptr + offset) or we can make ptr as a const pointer

→ `Char * const Ptr = (char *) malloc (sizeof(char));`

→ if we try to use (ptr++) the compiler will give an error so we must use (ptr+offset).

[2] calloc for dynamic memory allocation

→ the calloc function allocates space for an array of nmemb objects, each of whose size is object_size

→ all space is initialized to zero

→ the calloc function returns either a null pointer or a pointer to the allocated space.

syntax:-

```
Void *calloc (size_t nmemb , size_t object_size);
```

Note:- if you don't want to initilize the allocated memory with zero, it would be better to use malloc over calloc.

ex:-

```
char *ptr = Null;
ptr = (char*) calloc ( 30, sizeof(char));
```

---

[3] realloc for dynamic memory allocation

→ the realloc function is different from the malloc and calloc, it deallocates the old object and allocates again with the newly specified size.

→ if the new size is less than what is currently allocated, then the extra memory is returned to the heap, there is no ensure that the excess (extra) memory will be cleared

→ if the size is greater than what is currently allocated, then if possible, the memory will be allocated from the region following the current allocation immediately, otherwise, memory is allocated from a different region of the heap and the old memory is copied to the new region.

→ If ptr is the null pointer, the realloc behaves like the malloc function

→ if ptr not pointing a dynamically allocated memory, the behavior of realloc is undefined.

→ the return value of the realloc function is the pointer to a new object, or a null pointer if the new object could not be allocated

* syntax :

```
Void * realloc (Void *ptr, size_t size);
```

* free to deallocate the allocated memory :-

→ if the job of the dynamically allocated memory is finished then it is our responsibility to release that memory so that it can be reused.

→ the free() is used to release the memory space allocated dynamically.

* if ptr ⟶ argument of free().

→ is a null pointer, the free function does not perform any thing.

→ don't point to the memory that is allocated by the memory management function, the behavior of free function would be undefined

→ is pointing to a memory that has been deallocated, the behavior of free function would be undefined.

* Syntax:

Void free (void * ptr); ⟶ slides → Page-190.

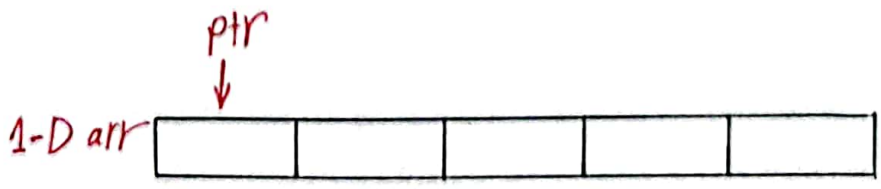
Dynamically allocate a 1D & 2D array

1 1D array:

- we can do this by creating a pointer to an integer and assign it heap memory
- when memory is successfully assigned to the pointer then we can use this pointer as a 1D array and using the square braces "[ ]" we can access the pointer as Like the statically allocated array

ex:

```
int *ptr = (int *) malloc (5 * sizeof(int));
ptr[0] = 1;
ptr[1] = 2;
printf("ptr[0] = %i\n", ptr[0]); ⟶ ptr[0] = 1
free(ptr);
```



[2] Create 2D array using the dynamic memory allocation.

1- create a pointer to pointer and allocate the memory for the no of rows using malloc or calloc

```
unsigned int **ptr=Null;
ptr=(unsigned int **) malloc (no.of.rows * sizeof(unsigned int*));
```

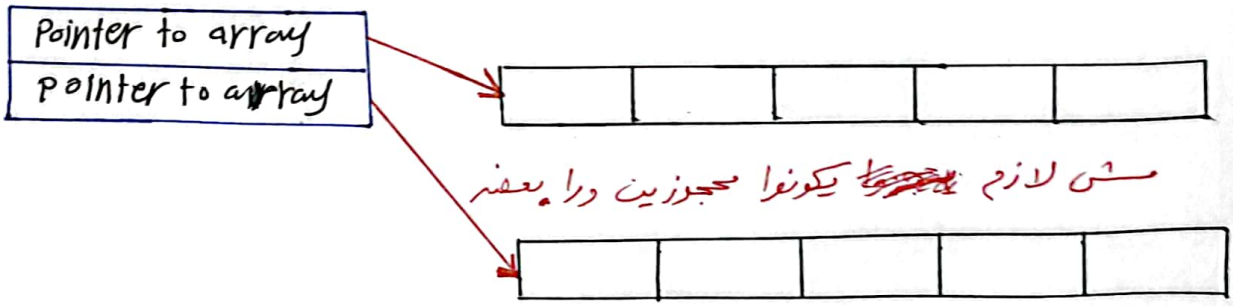
if → no.of rows = 2 then we have something like this

ptr to array
ptr to array

2- allocate memory for each row-column using the malloc.

```
for (counter=0 ; counter < no.of.rows ; counter++){
    ptr[counter]= malloc (no.of columns * sizeof(unsigned int));
```

→ now we have something like this. if no.of col = 5



* if each row does not have the same number of columns then allocate memory for each row individually.

```
ptr[0] = malloc (columns_1 * sizeof(unsigned int));
ptr[1] = malloc (columns_2 * sizeof(unsigned int));
```

slides page 192 : 193

Files

* the output of C program is generally deleted when the program is closed, sometimes, we need to store that output for purposes like data analysis, comparison for different condition ...etc, the use of file handling is exactly what the situation calls for.
* Basically, file represents a sequence of bytes that store in the disk permanently.

→ file handling enables us to create, update, read, write and delete the files using the C program

---

1 opening or creating a file

→ we used the Fopen function which is used to create a new file or open an existing file

→ it is declared in <stdio.h>

→ it has two arguments:

1 filename : this is the C string containing the name of the file to be opened

2 mode : this is the C string containing a file access mode, these modes are predefined. if you are not using these modes then the behavior would be undefined.

* syntax:

```
FILE *fopen (constchar * filename , const char * mode);
```

* return value:

this function returns a file pointer. otherwise, Null is returned. So we must validate the returned pointer and check if it equal Null or not

→ after finishing the operation on the file we must close it using fclose function

syntax:

```
int fclose(FILE *ptr);
```

→ returned 0 if the file successfully closed

→ returned EOF (end of file) if the error is happend

ptr: is a pointer to a file object

*Modes:

→ "r" : opens a file for reading, the file must exist
→ "w" : creates an empty file for writing. if a file with the same name already exists, its content is erased and the file is considered as a new empty file
→ "a" : Appends to a file. writing operations, append data at the end of the file and the file is created if it does not exist
→ "r+" : opens a file to update both reading and writing, the file must exist
→ "w+" : creats an empty file for both reading and writing
→ "a+" : opens a file for reading and appending

---

writing to a file

1 Formatted output functions for file: (Fprintf function)

→ the fprintf function: is used to write the formatted data to the file
→ the arguments of fprintf function are similar to printf function except that fprintf has an extra argument which is a file pointer (first argument)
→ on success, fprintf function returns the total number of characters written (transmitted) and on error return a negative number.

---

* fgets function:

→ is used to read a line from the specified file and store it into string pointed to by (*str), it stops when either (n-1) characters are read. and it terminates when the new line character is read or the end of the file is reached, whichever comes first

*Declaration:-

```
Char *fgets(char *str, int n, FILE *ptr);
```

* str : this is the pointer to an array of chars where the string read is stored.
* n : this is the max number of characters to be read + '\0' usually the length of the array passed as str is used
* ptr : this is the pointer to a file object where characters are read from

* atoi function :-

→ it converts the string argument str to an integer type

* Declaration:

```
int atoi (const char *str);
```

→ this function returns the converted integeral number as an int value if no valid conversion could be performed, it returns zero.

ex:

```
int val=0;
char str[ ]="12345";
val = atoi (str);
printf ("%i \n", val ); ⟶ 12345
```

---

* atoF function :-

→ it converts the string argument str to a floating_point-number (double)

* Declaration:-

```
double atof (const char *str);
```

→ this function returns the converted floating point number as a double value, if no valid conversion could be performed, it returns zero (0.0)

ex:

```
float val=0.0;
char str[]="12345";

val = atof (str);
printf ("%f \n", val ); ⟶ 12345.00000
```

---

* atoL : converts string to long integer

* atoLL : converts string to Long long integer

Inline function

→ inline function are parsed by the compiler

→ Debugging is easy for an inline function as error checking is done during compilation time

→ inline function looks like a regular function but, is preceded (سبقت) by the keyword inline and it is substituted at the place where its function call is happened

→ function substitution is totally compiler choice.

ex:-

```
int inline get_sum (int num1, int num2);

int main() {
    int ret = get_sum(2,3);
    printf("%i\n", ret);
    return 0;
}
```

output:
undefined reference to 'get_sum'

→ this is one of the side effect of GCC compiler the way it handle inline function when compiled, GCC performs inline substitution as the part of optimization. So there is no function call present "get_sum" inside main function.

→ we can solve this problem by using :-

1. with function declaration not definition

```
int inline get_sum (int num2, int num2) __attribute__ ((always_inline));
```

OR

```
static int inline get_sum( int num2, int num2) ;
```

this force the compiler to consider this inline function in the linker and hence the program compiles and run successfully

	normal function	function like macro	inline function

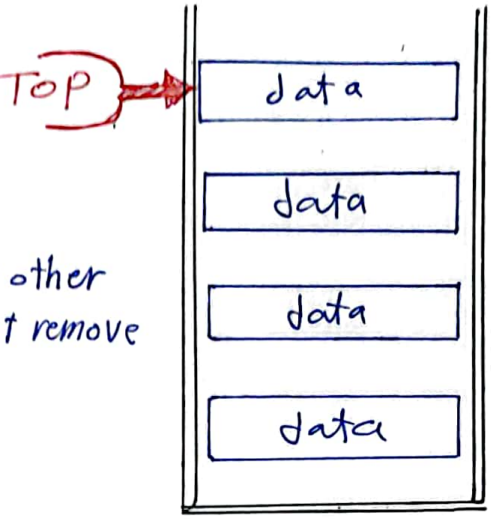
Example	int sum(int x, int y) {
	return (x+y);
	}
		#define ADD(X,Y)\
		(X+Y)
			inline int add(int x, int y){
			return (x+y);
			}

validation	exist	Not exist	exist
memory size	if the function size = 1 Byte then 1 Call → 1 byte 10 Call → 10 bytes	if the function size =1 byte then 1 Call → 1 byte 10 Call → 10 bytes	if the function size =1 byte then 1 Call → 1 byte 10 call → 10 bytes
execution time	speed is slower than macro and inline because of context switching	speed is faster than normal function because of there is no context switching	speed is faster than normal function because of there is no context switching
Tool chain	Compile	Preprocessor	Compiler (dependant)

Data structure (stack)

* A stack is a linear list in which all additions and deletions are restricted to one end, called the top
* Data input as {1, 2, 3, 4} is removed as {4, 3, 2, 1}
* this reversing attribute is why stacks are known as the last input first output (LIFO) data structure.

* in a stack, you can only add or remove an object at the top of the stack



computer stack

* if you want to remove any object other than the one at the top you must first remove

* Basic stack operations :-

- there are three basic stack operations :-

1 push data to stack :-

→ is used to insert data into the stack

→ we must ensure that the stack is not full, Here we can create a helper function to check if the stack is not full.

→ if there is not enough room, the stack is in an overflow state and the item cannot be added.

2 Pop data from the stack :-

→ pop removes data from a stack and returns the data to the calling module

→ we must ensure that the stack is not empty. Here we can create a helper function to check if the stack is empty or not.

→ if pop is called when the stack is empty, it is in an underflow sta

3 stack top :-

→ stack top returns the data at the top of the stack without deleting the data from the stack.

→ stack top can also result in under flow state if the stack is empty, so we need to verify first of the stack is empty or not

---

* there are three non-mandatory stack operations :-

① stack initilize :.
- initilize the stack data_type and pointer

② get the size of the stack :.
((number of elements) / (number of elements * size of one element))

③ Display stack elements :.
- Display the content of the stack.

---

* algorithm of basic stack operations :.

[1] push data :.

* Parameters :-
→ my_stack : is an object from struct passed by reference.
→ Value : contain data to be pushed into the stack

* Procedure :
→ validate if the stack is not full, and return error if the stack is full to prevent stack overflow.
→ increment the stack pointer
→ update the new location by the data provided by the user

[2] Pop data

* parameters :-
→ my_stack : is an object from struct passed by reference
→ Value : is reference variable to receive data

* procedure :.
→ validate if the stack is not empty, and return error if the stack is empty to prevent stack under flow
→ set value to data in top node
→ Decrement stack pointer.

3 stack top :-

* parameters :-

→ my_stack : is an object from struct passed by reference

→ value : is reference variable to receive data

* procedure :-

→ validate if the stack is not empty and return error if stack is empty (prevent stack underflow)

→ set value to data in top node

---

* Helper function

1 stack empty : Determines if stack is empty and returns a boolean

* parameters :-

→ my_stack : is an object from struct to a valid stack passed by reference.

→ returns stack status :-

true → if stack is empty false : if stack contains data

2 stack full : Determines if stack is full and returns a boolean.

* parameters :-

→ my_stack : is an object from struct to a valid stack passed by reference

→ return stack status :.

true → stack full false → if memory available

3 stack count : return the number of elements currently in stack

* parameters :-

→ my_stack : is an object from struct to a valid stack passed by reference

→ return integer count of number of elements in stack

* Functions declaration :-

```
return_status_t   stack_init (stack_t *my_stack);
return_status_t   stack_push (stack_t * my_stack, uint32 value);
return_status_t   stack_pop (stack_t * my_stack, uint32 * value);
return_status_t   stack_top (stack_t * my_stack, uint32 * value);
return_status_t   stack_size (stack_t * my_stack, uint32 * size);
return_status_t   stack_display (stack_t * my_stack);
```

Dynamic stack
(Array based)

```
typedef struct{
    Void **stack_array;   // points to the array that allocated in the heap
    sint32 element_count; // Has the actual number of elements in the stack
    sint32 max_size;      // Has the max number of elements in the stack
    sint32 stack_top;     // Has the index of the top element in the stack
} stack_t;
```

* A dynamic stack : is a stack data structure whose capacity (maximum elements that can be stored) increases or decreases in runtime, based on the operations (push or pop) performed on it.

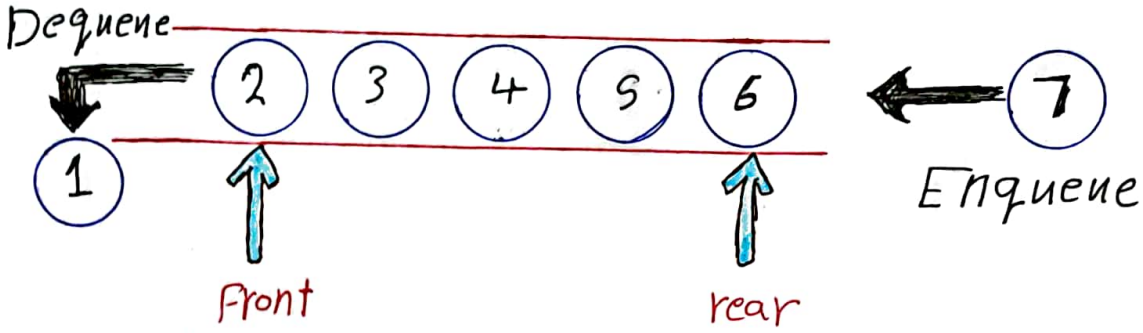
→ dynamic stack is implemented using dynamic memory allocation functions.
→ the stack_top : moves up and down as data pushed or popped
→ to push data into the stack, we add 1 to stack_top and use it as the array index for the new data.
→ to pop an element from the stack, we copy the data at index (stack_top) and then subtract 1 from top.

* function declaration :-

```
stack_t create_stack (uint32 max_size , stack_status_t *ret_status);
stack_t destroy_stack( stack_t *stack_obj , stack_status_t *ret_status);
stack_status_t stack_push (stack_t*stack_obj , void * itemptr);
Void * stack_pop (stack_t*stack_obj , stack_status_t *ret_status);
Void * stack_top (stack_t*stack_obj , stack_status_t *ret_status);
stack_status_t stack_count (stack_t *stack_obj , uint32 *stack_count);
```

Queue Data structure

- A queue is a Linear List in which data can only be inserted at one end called the (rear - back - tail) and deleted from the other end called the (front - head)
- A queue is a FIFO (First input - First output) data structure



* Queue operations :-

→ data can be inserted at the rear → Enqueue
→ data can be deleted from the front → Dequeue
→ return element from the front without removing it → Queue front
→ return element from the rear without removing it → Queue rear

[1] Enqueue operations :-

- after the data have been inserted into the queue, the new element becomes the rear.
- if there is not enough room for another element in the queue, then the queue is in an overflow state

→ Parameters :-

[1] queue_obj passed by reference
[2] item_ptr contains data to be inserted into the queue from the rear side

→ Procedure :-

- Validate if the queue is not full
- return error if queue is full (prevent queue overflow)
- increment the queue rear pointer
- update the new location by the data provided by the user
- increment the number of elements in the queue

2 Dequeue operations:-

- the data at the front of the queue are returned to the user and removed from the queue
- if there are no data in the queue when a dequeue is attempted, the queue is in an underflow state
- Dequeue deletes an element at the front of the queue.

→ Parameters :-

- queue_obj passed by reference.

→ Procedure :

- validate if the queue is not empty
- return error if queue is empty (prevent queue underflow)
- copy the front element in to local variable
- increment the queue front pointer
- Decrement the number of elements in the queue.

3 return element from the front (Queue_front):

- it returns the data at the front of the queue without removing it
- if there are no data in the queue when a queue front is attempted, then the queue is in an underflow state.

4 return element from the rear (Queue_rear)

- it returns the data at the rear of the queue without removing it.
- if there are no data in the queue when a queue rear is attempted, then the queue is in an underflow state.

* Declarations :-

```
typedef struct{
    Void **Queue_array;
    uint32 Queue_max_size;
    sint32 element_count;
    sint32 Queue_front;
    sint32 Queue_rear;
} Queue_t;
```

```
Queue_t *createQueue (uint32 max_size , QueueState_t *ret_status);
QueueState_t EnqueueElement (Queue_t *Queue_obj , Void *item_ptr);
Void *DequeueElement (Queue_t *Queue_obj , QueueState_t *ret_status);
Void *QueueFront (Queue_t *Queue_obj , QueueState_t *ret_status);
Void *QueueRear (Queue_t *Queue_obj , QueueState_t *ret_status);
QueueState_t QueueCount (Queue_t *Queue_obj , uint32 *QueueCount);
Queue state_t Queue_Destroy (Queue_t *Queue_obj);
```

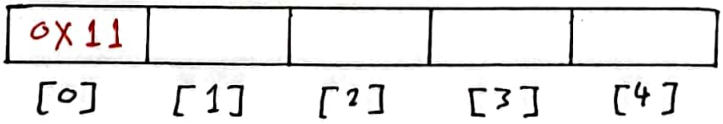
implementation hints :.

- like the stack when we implement a queue in an array, we use indexes rather than pointers
- we need 2 pointers, → Queue Front : to store the index of the front element
  → Queue rear : to store the index of the rear element
- if the queue is empty and no elements added yet, both the (Queue front) and (Queue rear) = -1

Queue Front = Queue_rear = -1

* when we insert an element to an empty queue, it is placed in the first index of the array, which becomes both the front and the rear

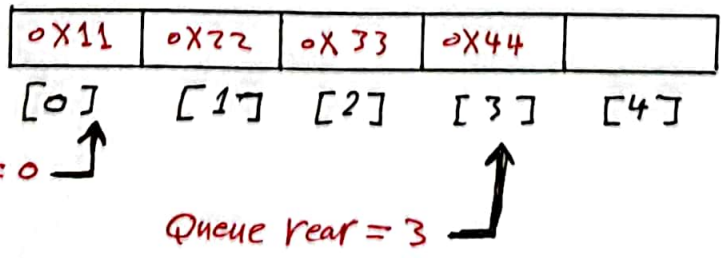
→ EnqueueElement (0X11)



QueueFront = Queue_rear = 0

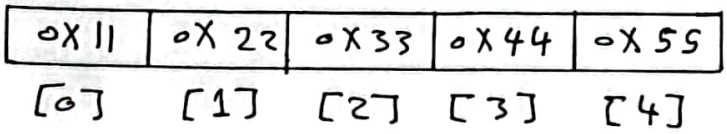
* subsequent enqueues are placed at the array location following the rear.

→ EnqueueElement (0X22), (0X33), (0X44)



Queue count = 4

→ EnqueueElement (0X55)



Queue front = 0

Queue rear = 4 (Queue_max_size - 1)

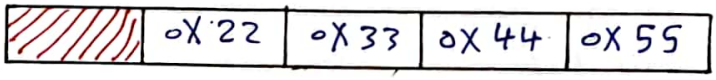
Queue count = 5

* because arrays have a finite number of elements, we can determine that the queue is full by testing the queue count against the maximum number of elements.

---

* Dequeue takes place at the front of the queue.
* as an element is deleted from the queue, the queue front is advanced to the next location, so queue front becomes queue front plus one.

→ DequeueElement (my_queue)

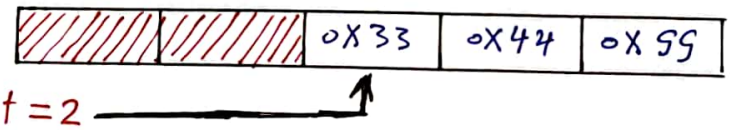


→ Dequeue operation return → 0X11

Queue front = 1

Queue rear = 4 Queue Count = 4

→ Dequeue Element (my_queue)



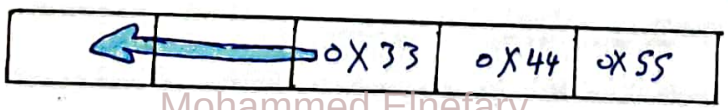
Queue front = 2

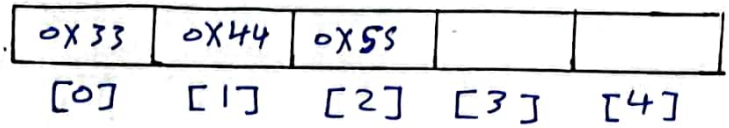
Queue rear = 4 Queue count = 3

* we know that we can insert new data just at the rear of the queue
* so if we need to insert anew element where the last element in the array
* if the last element of the array is occupied, but the queue is not full because there are empty element at the begining of the array

→ when the data are grouped at the end of the array, we need to find a place for the new element when we enqueue data

→ the first solution is to shift all of the elements from the end to the begining of the array.





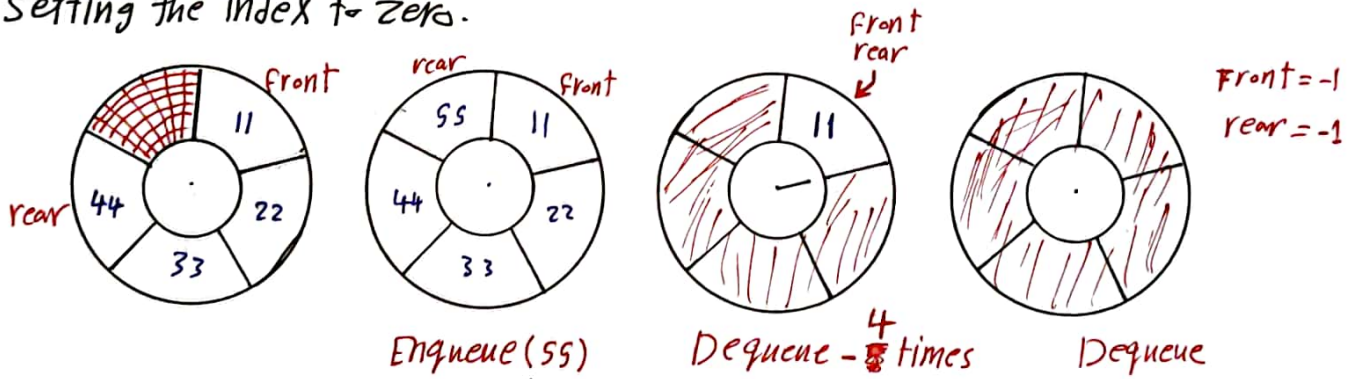
Queue front = 0 Queue rear = 2

Queue Count = 3 maxsize = 5

* another efficient solution is to use a circular array

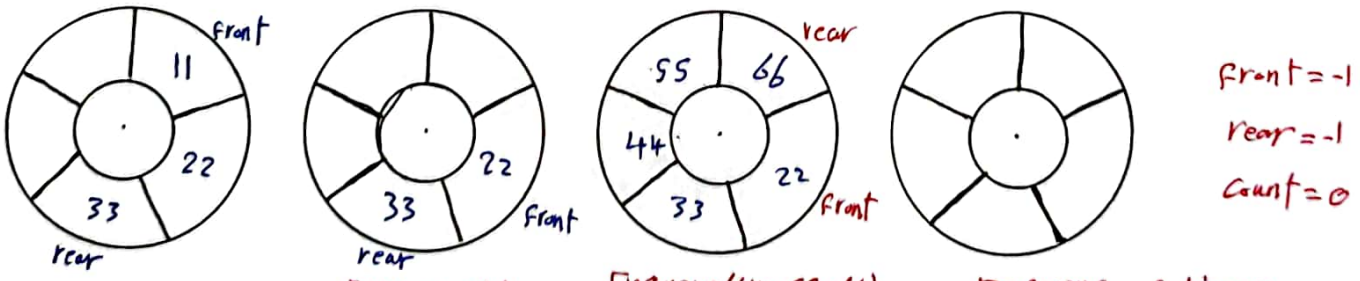
→ in a circular array, the last element is logically followed by the first element.

→ this is done by testing for the last element and, rather than adding one, setting the index to zero.



* we can't Enqueue overflow.

* we can't Dequeue underflow.



Linked List data structure

* a linked list is a linear data structure. unlike arrays, linked list elements are not stored at a contiguous location, as the elements are linked using pointers. they include a series of connected nodes, each node stores the data and the address of the next node

* why linked list:.
  * Arrays can be used to store linear data of similar types but have the following limitations:.
    - the size of the array is fixed, so we must know the upper limit on the number of elements.
    - Also, generally, the allocated memory is equal to the upper limit irrespective (= regardless) of the usage.
    - insertion of a new element / Deletion of a existing element in an array of elements is expensive, as the room has to be created for the new elements and to create room existing elements have to be shifted. but in Linked List if we have the head node then we can traverse to any node through it and insert new node at the required position

* Advantages of linked list
  - Dynamic array
  - ease (easily) of insertion / deletion.
  - no memory wastage: as the size of LL increase or decrease at runtime
  - flexible: this because the elements in linked list are not stored in contiguous memory location unlike the array
  - Efficient for large data.
  - Scalability (قابلية التوسع): contains the ability to add or remove elements at any position.

* Drawbacks (disadvantages) of linked list:.
  - memory usage:. more memory is required is the linked list as compared to an array. because a pointer is also required to store the address of the next element and it requires extra memory for it self
  - traversing (التنقل):. Random access is not allowed, we have to access elements sequentially starting from the first node (head). So we cannot do a binary search (will be discussed in details later) with linked list efficiently with its default implementation.

- Not Cache-friendly. since array elements are contiguous location, ther is locality of reference which is not there in case of linked list.
- it takes alot of time in traversing and changing the pointers.

* Basic operations on linked list:.

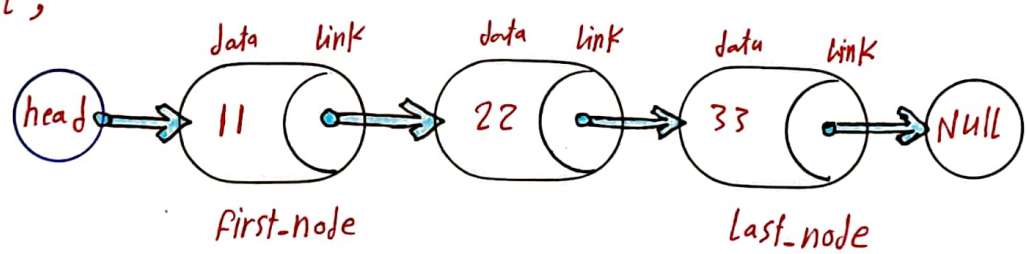
- insertion ( ): Adding element to the list. at the (begining - between - end)
- Deletion ( ): removing element from the list, from (begining - between - end)
- Display ( ): retrieve (return) data from the list or determine the data position.

---

* the basic building block for the linked list implementaion is the node.
* each node object must hold at least two pieces of information.
  1. First, the nude must contain the list item itself → Data
  2. second, each node must hold a reference to the next node → Link

ex:

```
typedef struct node {
    uint32 *Data;
    struct node *link;
} node_t;
```



1 insertion ( ):.

1.1 insert a node at the begining:.

→ A linked list is represented by a pointer to the first node of the linked list (Head)

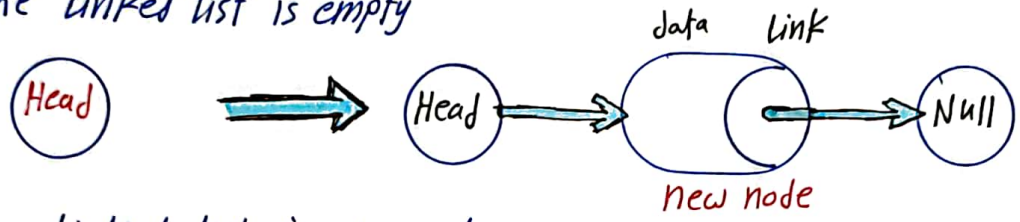
→ if the linked list is empty, then the value of the head is Null

```
struct Node {
    uint32 Node_Data;
    struct Node *Node_Link;
};
```

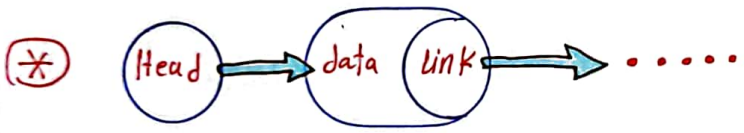
```
struct Node *Head = Null;
```

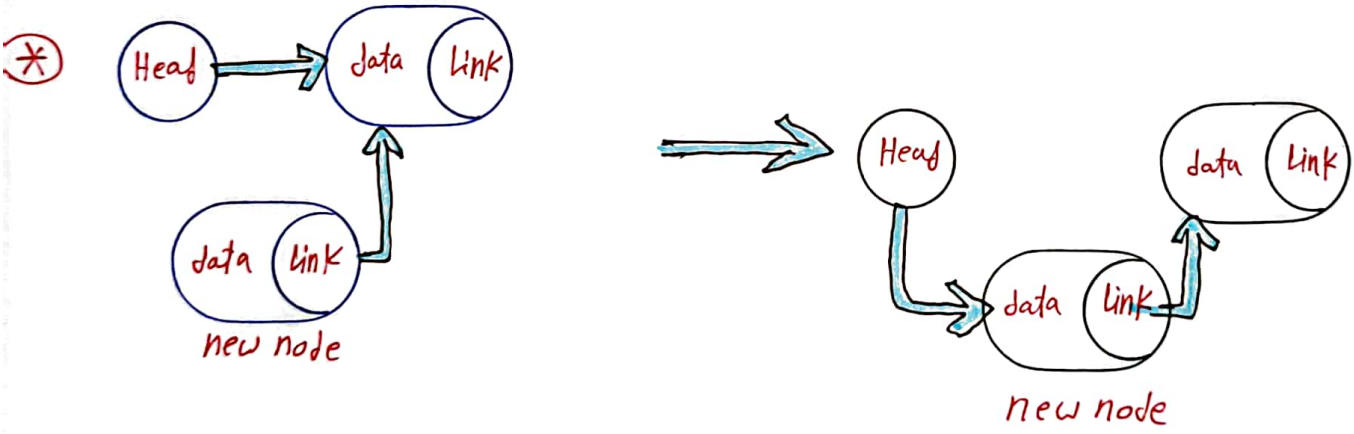
* To pass a linked list to a function and this function will change the Head.
You need to pass it by reference → (struct Node **Head)
→ if you passed it by value. So when it modifies head, it modifies the local copy of the pointer. So the insert function does not change the head variable you defined
→ Void insert_node_at_the_begining (struct Node **List);
→ insert_node_at_the_begining (& Head);

1 if the Linked list is empty

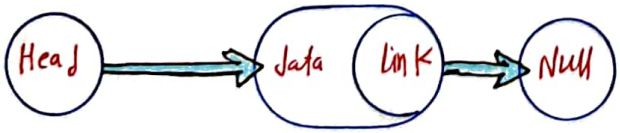


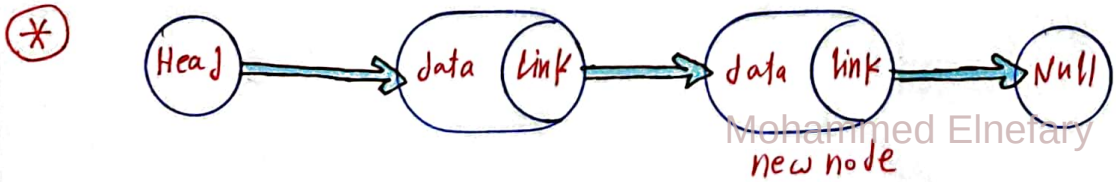
2 if the Linked list is not empty





1.2 insert a node at the end
1 if the Linked list is empty:.
نفس الكلام اللى فوق
2 if the Linked list is not empty:.





1.3 insert a node after another node

→ algorithm :-

① ask the user for the node position

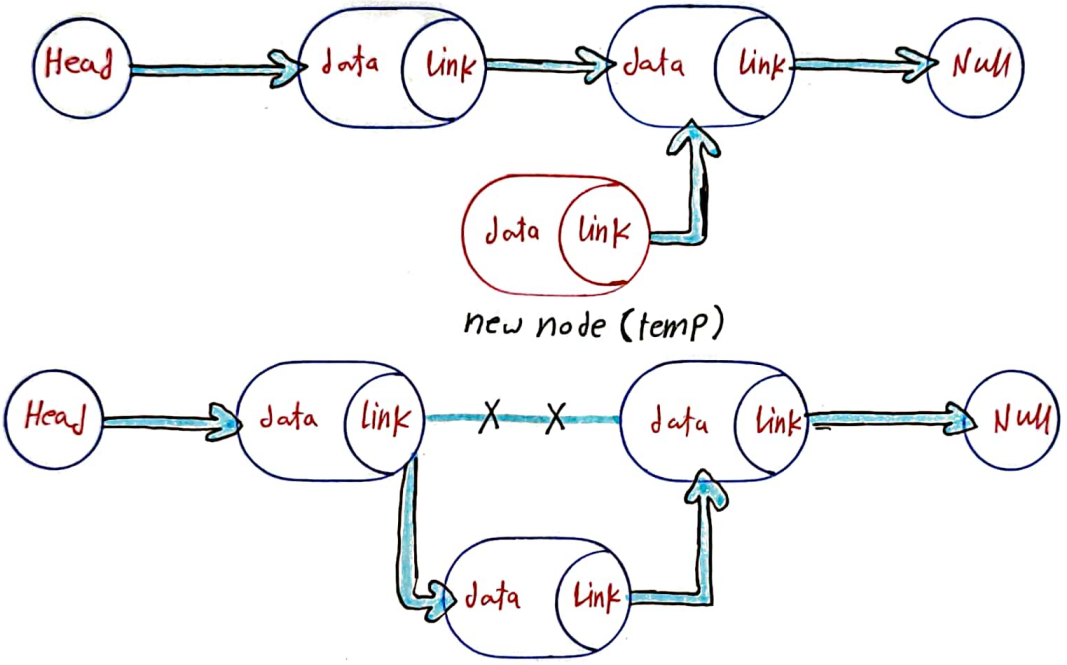
② check the validation of this position, by getting the length of the list and validate if the node position is greater than the length or not.

③ assign the value inside Head (address of the first node) to the temp pointer and use it to traverse between nodes, until we reach to the specified position

④ allocate a new node (temp) and assign the data to temp

⑤ make the next of temp as the next of the previous node.

⑥ finally, move the next of the previous node to point to temp.



1.4 Delete node from the begining :-

insert a node at the begining ← من العلية العكسية لـ

← نفس الخطوات ولكن بالعكس وذلك أيضاً مع

* Delete node at specific position ⇌ insert node after another node

← تكون أيضاً نفس الخطوات ولكن معكوسة ..

* if we need to display all nodes, we just traverse (iterate) between nodes then we display the data of each node, we can also define a local variable and increment it with each iteration to get the size of list

* reverse a single linked list :.

```
struct Node {
    uint32 Node_data;
    struct Node *Node_Link;
}
```

```
Void reverse_List ( struct Node ** List) {
    struct Node *Temp_Node = *List;
    struct Node *Temp_ptr_prev = Null;
    struct Node *Temp_ptr_next = Null;
    if (Null == Temp_Node -> Node_link){
        Temp_Node = Null;
    }
    else {
        while ( Null != Temp_Node) {
            Temp_ptr_next = Temp_Node -> Node_link;
            Temp_Node -> Node_link = Temp_ptr_prev;
            Temp_ptr_prev = Temp_Node;
            Temp_Node = Temp_ptr_next;
        }
        * List = Temp_ptr_prev;
    }
}
```

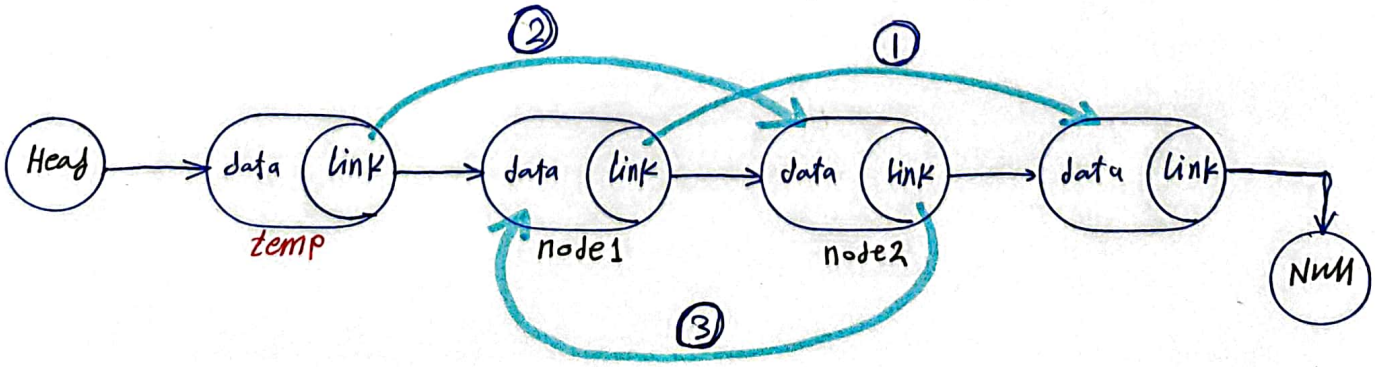
---

* Swap 2 nodes in the list :

→ the problem has the following cases to be handle :

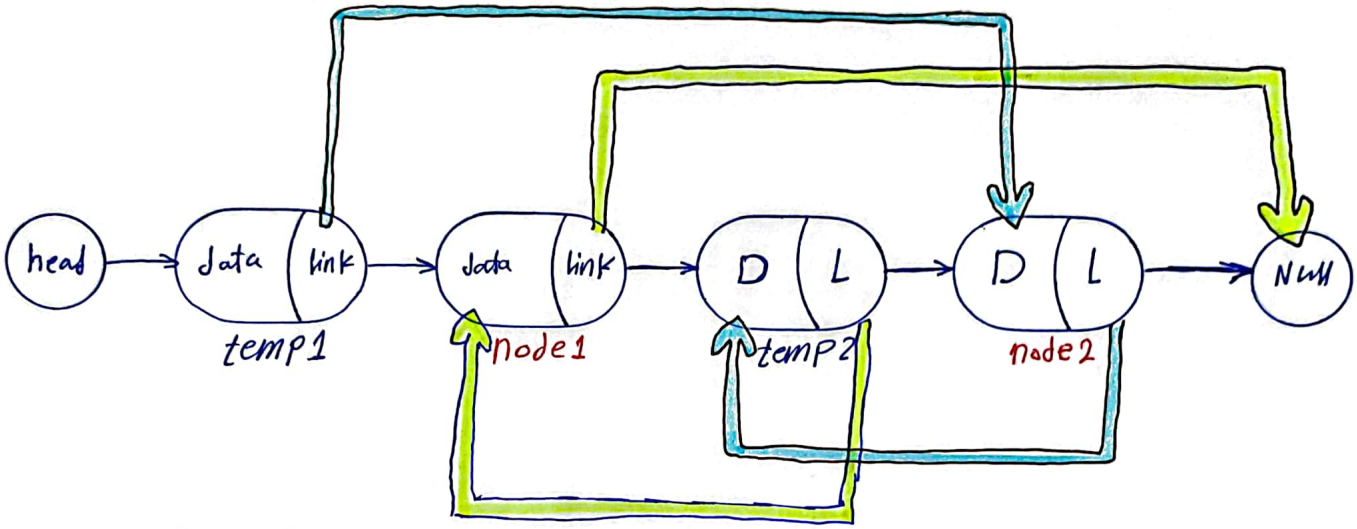
- 2 nodes may be adjacent or not
- one of 2 nodes may be a head node
- one of 2 nodes may be the last node
- one or two nodes may not be present in the list

[1] the 2 nodes are adjacent → swap (node1 ⇐⇒ node2)



① node1 -> link = node2 -> link ;
② temp -> link = node2 ;
③ node2 -> link = node1 ;

[2] if the 2 nodes are not adjacent :



(*) required variables :.

→ node 1 & node 2
→ temp 1 ⟶ the previouse of node1 (prev1)
→ temp 2 ⟶ // // // node2 (prev2)
→ temp ⟶ the next of node1

```
Void  Swap_nodes ( struct Node ** head_ref , int pos1 , int pos2 ) {
    if (pos1 == pos2) {
        return ;
    }
    struct Node * prev1 = Null ;
    struct Node * node1 = *head_ref ;
    struct Node * prev2 = Null ;
    struct Node * node2 = * head_ref;
    int Count = 1 ;
    while ((node1) && (Count != pos1)) {        /* find the nodes to be */
        prev1 = node1 ;                          // swapped
        node1 = node1 -> NodeLink
        Count ++;
    }
```

```
Count = 1 ;
while ((node2) && (Count != Pos2)) {
        prev2 = node2 ;
        node2 = node2 -> NodeLink;
        Count++ ;
}
if ((!node1) || (!node2)) {          /* if either of the nodes is not */
        return ;                     /* found, there's nothing to swap */
}

if (!prev1) {                        /* if the first node is the head of the list */
    *head_ref = node2 ;
}
else {
    prev1 -> NodeLink = node2 ;
}
if (!Prev2) {
    *head_ref = node1 ;              /* if the 2nd node is the head of the list */
}
else {
   prev2 -> NodeLink = node1 ;
}

struct Node * temp = node1 -> NodeLink;
 node1 -> Nodelink = node2 -> Nodelink;
 node2 -> Nodelink = temp ;
}
```

---

* function to sort the nodes in the list

```
// this function takes a double pointer to the head of the linked list as
// input

Void sort_nodes (struct Node ** head_ref ) {
```

```
unsigned int counter_1 = 1 ;        // initilize variables for counting current
unsigned int counter_2 = 2;         // node and next node

struct Node *node1 = *head_ref;     // initilize a pointer to the first node
                                    // in the list
 while (node1 -> node_link != Null ) {
    if ((node1-> node_data) > (node1 -> node_link -> node_data)) {
 // if the data of the current node is greater than the next node data

  swap_nodes ( head_ref , counter_1 , counter_2);
 // Call the swap_nodes function which we have already defined
  node1 = * head_ref;
   counter_1 = 1;
   counter_2 = 2;
}

else {
      node1 = node1 -> node_link;
      counter_1 ++;
      counter_2++;
     }
  }
}
```

Function to Print the middle of a given linked list

```
Void print_middle_node ( struct Node *head_ref ) {
  if (head_ref == Null || head_ref -> node_link == Null ) {
      printf (" the middle node is %i \n ", head_ref -> Node_data);
       return ;
  }
  struct Node * ptr_one_step = head_ref ;     → traverse in one step
  Struct Node * ptr_dbl_step = head_ref;      → traverse in two step
```

```
if (head_ref != Null)
 {
   while ((ptr_dbl_step != Null) && (ptr_dbl_step -> Node_Link != Null)){
       ptr_dbl_step = ptr_dbl_step -> Node_Link -> Node_Link;
       ptr_one_step = ptr_one_step -> Node_Link;
   }
  printf(" the middle node data is %i \n", ptr_one_step-> Node_data);
 }
 else{
    printf("empty list \n");
 }
}
```

---

*Function to delete the middle node of a single linked list

```
Void delet_middle_node (struct Node *head_ref){
    struct Node * ptr_dbl_step = head_ref ;
    struct Node * ptr_one_step = head_ref;
    struct Node * prev = Null;
     while ( ptr_dbl_step != Null && ptr_dbl_step -> Node_Link != Null) {
          ptr_dbl_step = ptr_dbl_step -> Node_Link -> Node_Link;
           prev = ptr_one_step ;
           ptr_one_step= ptr_one_step -> Node_Link;
     }
     if (prev != Null) {
         prev -> Node_Link = ptr_one_step -> Node_Link;
         free (ptr_one_step);
     }
}
```

---

* function to delete the duplicate elements from sorted linked list.

```
Void remove_duplicate ( struct Node *head ) {
    struct Node * node1 = head ;
    struct Node *node2 = node1 -> Node_Link;
    while ( node1 != Null && node2 != Null ) {
        if ( node1 -> Node_data == node2 -> Node_data ) {
            node1 -> Node_Link = node2 -> Node:Link;
            free ( node2 );
        }
        else {
            node1 = node1 -> Node_Link;
            node2 = node2 -> Node_link;
        }
    }
}
```

ex:

List : 1 2 2 3 4 4 5 6 ⟹ 1 2 3 4 5 6

---

* function to merge a linked list into another linked list

```
Void merge_two_list ( struct Node ** head_ref_1 , struct.Node * *head_ref_2){
    if ( *head_ref_1 == Null ) {
        *head_ref_1 = *head_ref_2 ;
        *head_ref_2 = Null ;
        return ;
    }
    if ( *head_ref_2 == Null ) {
        return ;
    }
    struct Node *node1 = *head_ref_1 ;
    struct Node *node2 = *head_ref_2 ;
    while ( node1 != Null && node2 != Null && node2 -> Node_Link != Null ) {
        node2 = node2 -> Node_Link ;
    }
```

If we need :
head_ref_1 → list2 → list1

```
node2->Node_link = *head_ref_1;
*head_ref_1 = *head_ref_2;
*head_ref_2 = Null;
```

If we need :
head_ref_1 → list1 → list2

```
while (node1->Node_link != Null){
    node1 = node1->Node_link;
}
node1->Node_link = *head_ref_2;
*head_ref_2 = Null;
```

* Function to delete a linked list.

```
Void delete_linked_list (struct Node **head_ref){
    struct Node *node1 = *head_ref;
    struct Node *next_node = Null;
    while (node1 != Null){
        next_node = node1->Node_link;
        Free (node1);
        node1 = next_node;
    }
    *head_ref = Null;
}
```