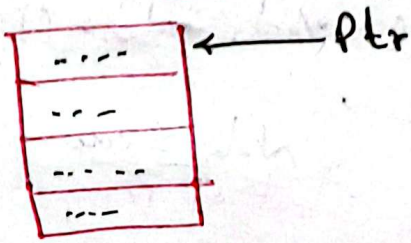
# Pointers : C implementation

↳ store the base address for a Variable

Ex: integer
4 bytes




## Declaration :

data_type *pointer_name

int *ptr; → Pointer to an integer

## Initializing :

① By assigning the adress of other variable

```
int x = 5;
int *ptr;
ptr = &x;
```

reference operator

address of

```
int x = 5, *ptr = &x;
```

Note:
int * x, *y;
→ a Pointer and a Variable
Not 2 Pointers

Common mistake

## Access :

```
int x=5, *ptr= &x;
Printf("%d", *ptr);
```

value of → dereference operator

*ptr = 10;

XXX int *ptr; ← uninitialized Pointer
Printf("%d", *ptr); *ptr = 1; ← Very dangerous

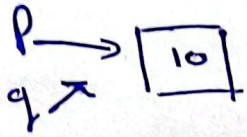
using dereference pointer → Segmentation fault : trying to access illegal memory

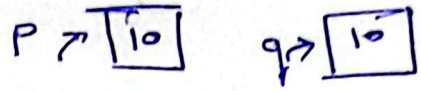
② Assigning the Value of another ~~Variable~~ Pointer

```
int i = 10, *ptr = &i;
int *P; = ptr;   or  P=ptr;
Printf(" %d, %d ", *P, *q); → 10, 10
```

Note: P = q  Not Same  *P = *q

↳ Waste Content of P




Returning Pointers: Never return a Pointer of a local Variable

why return by itself destroys the function

So also its local variables got destroyed

```
int main( ) {
  int a[ ] = {1,2,3,4,5};
  int n = sizeof(a)/sizeof a[0];
  int *mid = find mid(a,n);
}

int * findmid(int a[ ], int n)
{ return &a[n/2]; }
```

Not a Local Variable For the Function

```
int * fun( ) {
  int i = 10;
  return &i;
}
```

→ Local Variable

Error

XXX

* Pointers Are used in Pass by reference: swap fun, max, min

* Pointers are powerful But they can't change memory that is Read-only → Constants

```
Const int x = 5;   *ptr = &x;
*ptr = 10; → ERror : Read only memory
```

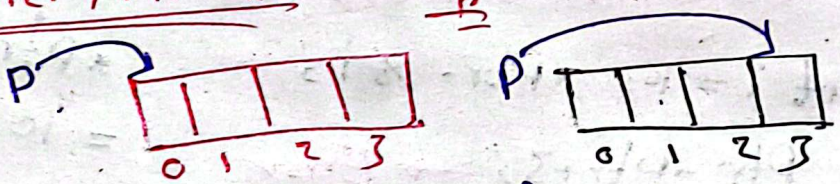
%p : format specifier for any Location

```
int i = 10;
Dprintf("%p", &i);
```

Think of it
* &i = i
* *p = p

## Pointer Arithmetic

① ptr + i




P = P+3

if P = &a[i]
P = P+j ⇒ &a[i+j]

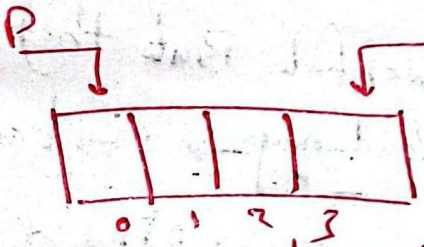
P = P+3 → move the pointer forward by 3 Positions

determined by type
- 4 bytes int
- 1 byte char

For int if P = 1000 ∴ P+1 = 1004
Position = 4

② ptr - i  P = P-3;
↳ move the pointer 3 Positions backward

③ ptr1 - ptr2




4

q-p = 3 → distance between 2 Pointers

Not 3*4 ← this is implicit

$$q - p = 2 \equiv \boxed{\frac{1012 - 1000}{4} = 3}$$

→ you don't Subtract addresses

* Performing arithmetic on Pointers of element not arrays get undefined behaviors

```
int i = 10; *Ptr = &i;
Ptr = Ptr + 3; → Error
```

*ptr + 1 Note
= 10 + 1 = 11

* you can't Perform Subtraction of 2 pointers if they are Pointing to different arrays

Remember: you don't subtract addresses

```
int a[] = {1,7,5,4}
int b[] = {5, 6,7,8}
int *p = &a[0]
int *q = &b[3]
Printf ("%d", q-p); → undefined behavior
```

you subtract index

# Pointer Arithmetic



(4) increment_decrement

*(P++) → move the Pointer forward by one Position

(*P)++ → increase the Value of Pointer by 1

The Value that Pointer is Pointing at

So, the Pointer still Pointing to the same memory Location

Ex

```
int a[ ] = { 5,16, 7, 8, 9}
int *ptr = &a[2];
printf("%d", *(--p));   16
printf("%d", *(p--));   16
printf("%d", (*p)++);   6
```

Note

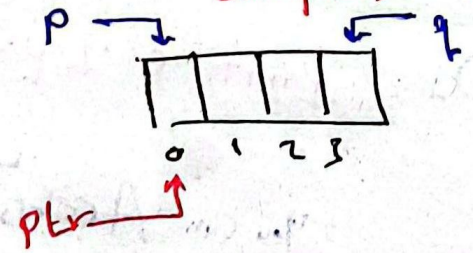
*P++ is same as *(P++)

*++ptr = *(++P)

5. Comparing Pointers

only possible for two Pointers in the same array

Comparison is done between index




q <= P → false

P <= q → true

P == ptr → true

Ex:

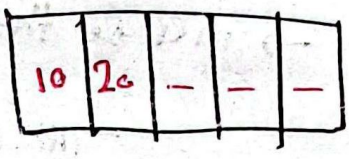
```
int a[ ] = {11, 12, 36, 5, 2};
int Sum = 0, *p;
for(p = &a[0]; P <= &a[4]; P++)  Sum += *P;

printf("%d", Sum);   66
```

Name of array can be used as a Pointer to the first element in the array



Ex: int a[5];
*a = 10;
*(a+1) = 20;
*(a+2) = 30;




a

∴ *(a+i) = a[i]

Rewriting

```
int a[ ] = {11, 12, 36, 5, 2};
int Sum = 0, *P;
for(P = a; P-a<4; P++) Sum += *P;
```

out: Sum = 66

Note: you can use the array name as Pointer
But you are not allowed to change or Assign it

Ex int a[ ] = {11, 22, 36, 512};

printf("%p", a++); ← Error

printf("%p", a+1); ← you can Access Not Assign

you can also use another ptr

```
int *Ptr = a;
printf("%p", P++);
```

## Passing an array to a function:

The name of the array is always passed as a Pointer

```
int add( int *b  /* int b[]; */ int len] {
    int sum = 0, i;
    for(i=0; i< len; i++) sum += b[i];
    return sum; }

int main() {
    int a[] = {1,2,3,4,5}
    int len = sizeof(a) / sizeof(a[0]);
    Printf("%d", add(a,len));
```

in both cases you receive a Pointer

Note: sizeof() is an operator Not function

you are passing a Pointer

you are not the whole array passing

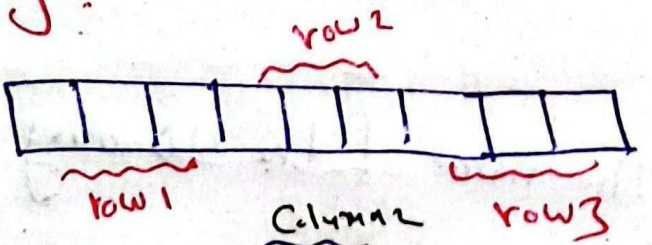
→ you can access a Pointer like an array

$*(P+i) = P[i]$ $*P = P[0]$

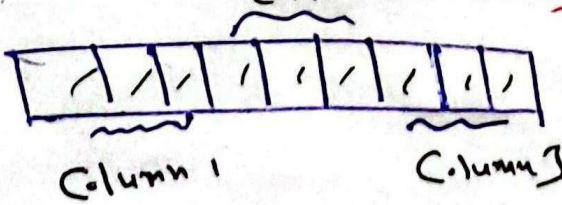
---

## Pointers with 2D array:

Row major order:




Column major order:




in C: Multidimentional Arrays are in Row major order [Row First]

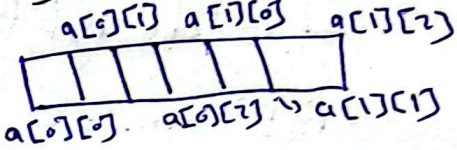
# Accessing 2D array:

* Traditionally: two For Loops

row: No. of rows
cols: No. of cols

For rows → `for(i=0; i<row; i++)`

For cols → `for(j=0; j<col; j++) Print("%d", a[i][j]);`

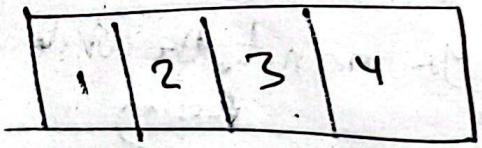



* using Pointers: one For Loop

```
for(int *p = &a[0][0]; p<=&a[row-1][col-1]; p++){
    Printf("%d", *p);
```

---

## Address Arithmetic of multi-dimensional Arrays:

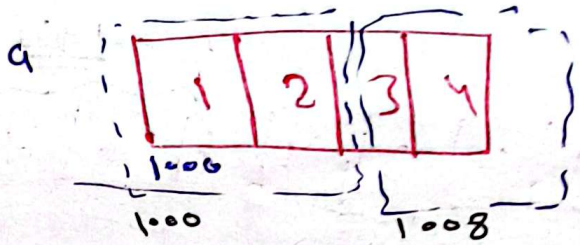



← 2D arrays contains 2 1D arrays

↳ Contains 2 elements

The name of the 2D array is a Pointer to the first 1D (Not the first element)

# 2D Array




a ⇒ 1000

(a+1) ⇒ [redacted] ⇒ 1000 + 8 = 1008 → address of second 1D array

↓

Pointer to the First 1D array

So adding 1 is moving the Pointer one Position

one 1D array

Next one element

*a ⇒ Pointer of first element of first 1D array

↳ *(a+0) = a[0] = a[0][0]

*a : will Not give the element but its address

**a ⇒ The first element of the first 1D array

↳ * (*a) ⇒ * (*(a+0)) ⇒ * (a[0][0]) = a[0][0]

∴ **a = a[0][0]

a+1 : Pointer of 2nd 2D array

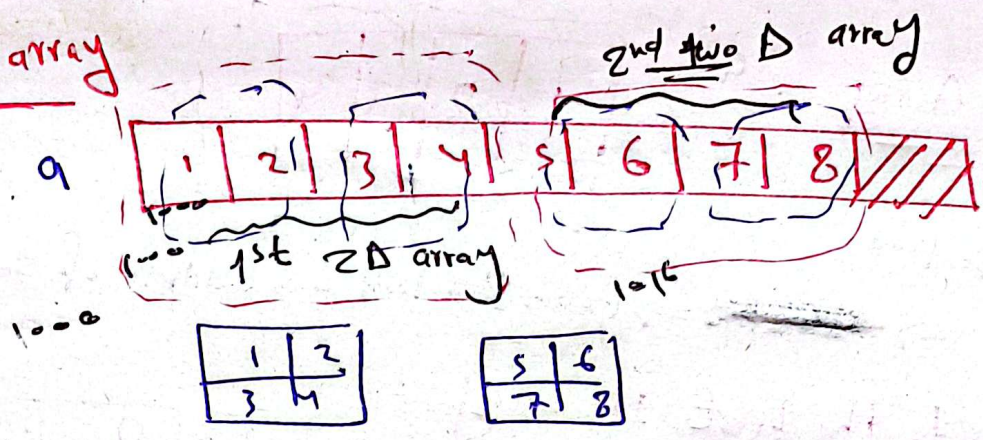
a[1] *(a+1) : Pointer of 1st element in 2nd 1D array

a[1][0] **(a+1) : first element in 2nd 1D array

*(a+1) + 1 ⇒ Pointer of second element of second 1D array

Pointer

# 3D array




| 1 | 2 |
|---|---|
| 3 | 4 |

| 5 | 6 |
|---|---|
| 7 | 8 |

The name of 3D array is the Pointer to the 1st 2D array

```
int a [2] [2] [2];
```

- [2] → Pointer to 1st 2D Array
- [2] → 2 2D arrays / 2 1D Array in Each 2D array
- [2] → 2 Elements in 1D Array

a ⟶ Pointer of first 2D array 1000

a+1 ⟶ Pointer of 2nd 2D array 1016

*a ⟶ Pointer of first 1D in 1st 2D array

**a ⟶ Pointer of first element in first 1D in 1st 2D

***a ⟶ first element a[-][-][-]

*** (a+1) ⟶ first element in 2nd 2D array 5

Ex: To access 2nd element:

*(**a + 1) 2

To access 2nd last: *(*(*(a+1) + 1)) 7

Question: For char a[100][100], &a[0][0] = 0

Gate ∴ &a[40][50] is 4050
40*100+50

There is a formula you can watch on the video, Nesc Pointers Playlist: Video No. 18

---

for int X[4][3] &X[0][0] is 2000

unsigned int X[4][3] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} };

Printf("%u, %u, %u", X+3, *(X+3), *(X+2)+3);

2000 + 3*4*3 2036, 2036, 2036

---

→ Pointers of an Entire array

Remember: int a[5] = {1,2,3,4,5};

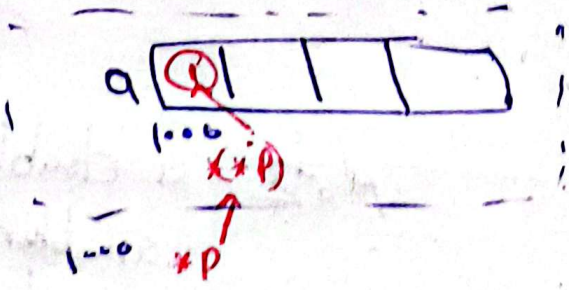
int *p = a; ← Pointer to the first element of array

But int (*p)[5] = &a; ← Come out of the box

Pointer to array of 5 elements [integers]

Ex int (*p)[5] = &a;

Printf("%d", **p); ⇒ 1




*p → Pointer of first element

Note int *p[10]; → array of Pointers But int (*p)[10]; ← Pointer of Entire array

# Pointers of an Entire array



Ex
```
int a[][3] = {1,2,3,4,5,6};
int (*ptr)[3] = a;        → first 1D array
Printf("%d %d", (*ptr)[1], (*ptr)[2]);
++ptr;  ⟶
Printf("%d %d", (*ptr)[1], (*ptr)[2]);
```
↑ 2nd 1D array

Ans: 2 3 5 6

---

Neso Pointers playlist Question: Video 22 ⟶ 26

---

```
int c, *b, **a;
c = 4; b = &c; a = &b;
Printf("%d", **a);  →4
```

Pointer to a Pointer
" Double Pointer "

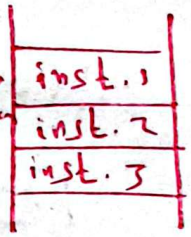
Note
```
int (*p)[5] = &a;
Printf("%d", **p);
```
→ it is a Pointer of Entire array
Not a double Pointer

```
int a[5][5];
Printf("%d", **a)
```
→ First element of first 1D array
Also it is not a double Pointer

Note: a double Pointer should have address of another Pointer
don't try to assign another variable Location

# Function Pointers

↳ a Pointer to the first instruction in a function




a Function can be stored as set of instructions

declaration:

```
int (*ptr) (int, int)
```

Pointer of a function

has 2 integer Parameters

returns an int.

Assigning the address of a function to a function Pointer

```
int add(int a, int b) { return a+b; }
```

```
int (*ptr)(int,int) = &add;
```

name of function only

use

Ex: `int result = *ptr(10, 20);` → 30

or

```
int (*ptr)(int,int) = add;
```

The name of the function is also its address

```
int result = ptr(10,20);
```
→ 30

Application: Calculator [dynamically based]

Remember int *Ptr[4]; → array of Pointers

```
float Sum(float a, float b) {return a+b;}
float Sub(float a, float b) {return a-b;}
float mult ⟶
float div ⟶
```

```
int main() {
    float (*Ptr[4])(float, float) = {Sum, Sub, mult, div};
```

Array of function Pointers

```
    float a,b; int choice;
    Printf("0: Sum, 1: diff, 2: mult, 3: div \n")
    Scanf("%d", &choice);
    Scanf("%f %f", &a, &b);
```

Very Easy → instead of Switch Lif

```
    Printf("%f", Ptr[choice](a,b));
}
```

Reduce Redundancy