```
K=K-3;
x= & arr[1];
y= & arr[5];
Printf ("%d", y-x);
j= & arr[4];
K= (arr+4);
if (j==K)
    Printf  2 pointers point to the
            same loc.
else
    The 2 pointers point to different
            locations.
```

Arrays & pointers:-

Arrays - Stored in contiguous memory locations.

Pointer when incremented always point to the next loc. of its type.

```
main()
{
   int num[]= {24,34,12,44,56,17}
      int i, * ptr;
      ptr= & num[0];  | ptr = num;
   for (i=0; i<=5; i++)
   {  Printf ("%u %d", ptr, * ptr);  * (ptr+i)
                                     , *(num+i)
        ptr++;                       ,* (i+num),
                                      num[i],
   }  }                               i[num]);
```

O/P:

65512	24	24	...	24
65516	34	34	...	34
⋮				
65532	17	17	...	17

Note:

Accessing array elements by pointers is faster than subscripts. (if elements are accessed in fixed order).

No logic → access using subscripts.

1st take this.

(ex).

```
main()
{
    int a[] = {10,20,3,4,5};
              1000
    int i;
    for (i=0; i<5; i++)
        Printf("\n%d", a[i]);
                       *(a+i)
}
```

Internally,

a[i] = base addr + i × size of datatype; →1st

*(a+i).

Other ways, a[i], i[a], *(a+i), *(i+a).

Passing array as pointers to a function

```c
#include <stdio.h>
void display (int *);
void main()
{   int num[] = {24,34,12,44,56};
    display (num);
}
void display ( int *ptr)
{   int i;
    for( i=0; i<5 ; i++)
    { Printf ("ele = %d", *ptr);
      ptr++;
    }
```

Returning array:-

```c
#include <stdio.h>
int *fun (int *num);
main()
{
        int max, *p, i;
        p= fun (&max);
        for(i=0; i<max; i++)
              Printf ("%d", *(p+i));
                                  *p;
}             p++;
int *fun (int *num)
```

```
{
    static int arr[]= {1,2,3,4,5};
       * num = sizeof(arr) / sizeof(arr[0]);

    return arr;
}.
```

2nd

```
main()
{
    int a[] = {1,2,3,4,5}
    int i, *p;
    p= a
    for ( i=0; i<5; i++)
          Printf("%d", *(p+i));
```

~~Pointers & 2D array.~~

Functions returning pointers.

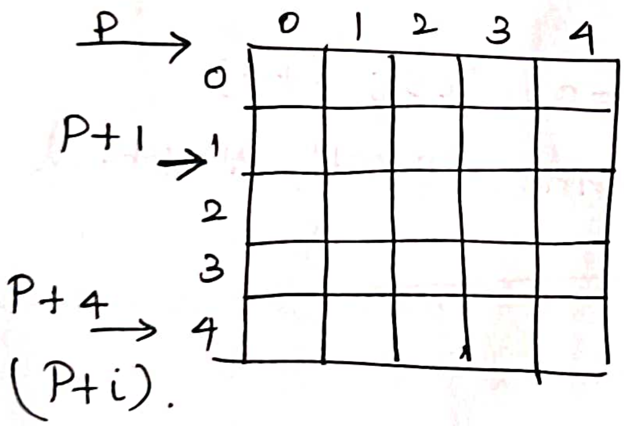
```
int * larger (int *, int *);
main()
{   int a=10, b=20;
    int *p;
```

```c
P = larger (&a, &b);
   Printf ("%d", *P);

int *larger (int *x, int *y)
{
   if (*x > *y)
        return (x);       → address of a
   else
        return (y);       → address of b
}
```

Pointers & 2D array.

(5 × 5) matrix



int a[5][5];

P → pointer to 1st row

2nd

P+i → " " ith row.

*(P+i) → pointer to 1st element in the ith row

*(p+i)+j → pointer to jth element in the ith row

((p+i)+j) → value stored at cell (i, j).

ex:

```
main()
{
    int s[4][2] = {{1234, 56},
                   {1212, 33},
                   {1434, 80},
                   {1312, 78}};

  int i,j;
  for (i=0; i<=3; i++)
  {   for (j=0; j<=1; j++)
          Printf ("\t%d", *(*(s+i)+j));
      Printf ("\n");
  }

}.
```

→ explain the concept of 2D nested loops

then follow

o/p:-

1234	56
1212	33
1434	80
1312	78.

1st teach this,

```
main()
{
    int s[4][2];
```

```
    int i;
    for (i= 0; i<4 ; i++)
        Printf ("%u\n", s[i]);
}
```

O/P:

65508
65516
65524
65532

s[2] → 65524 → address of 2nd 1-D array

s[2][1]

↳ (s[2]+1) → 65524 +1

→ 65528

* (s[2]+1).

$\boxed{*(*(s+2)+1).}$

∵ num[i] = *(num+i)

Pointer to an array (Array pointer)

A pointer to an array is a pointer that points to the whole array instead of the 1st element of the array.

example.

It considers the whole array as a single unit instead of it being a collection of elements.

syntax: type (* Ptr)[size];

ex) int (* ptr)[10];