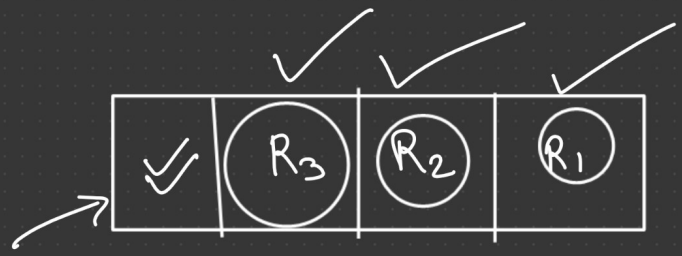
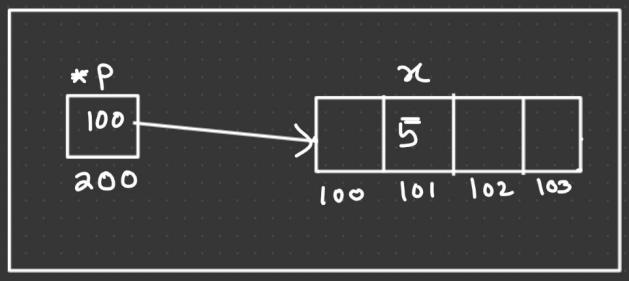
Pointers

```
main()
{
    int x;
    char y;
    float z;

    int *p;
    char *p;
    float *p;

}
```




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

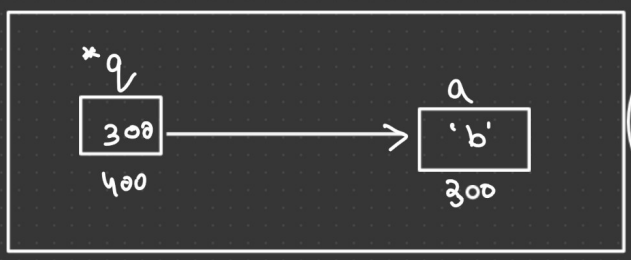
RAM



value at address of x

* &x = p

```
char a = 'b';
char *q = &a;
```



value at address of a

* &a = q

	Address	Value
int x = 5;	&x	x
char a = 'b';	&a	a
int *p = &x;	p	*(&x) = *p
char *q = &a;	q	*(&a) = *q

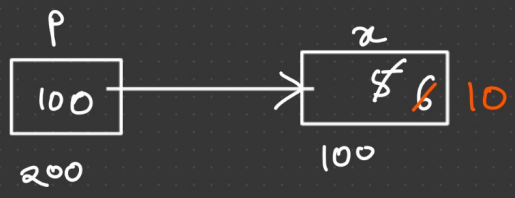
*p = x

p = &x

⇒ x = *&x (v.v.Imp)

*& → cancel each other

```
int main()
{
    int x = 5;
    int *p = &x;
    x = 6;
    printf("%d %d", x, *p);
    *p = 10;
    printf("%d %d", x, *p);
}
```



6 6 (↑ ↑)

10 10 (↓ ↓)

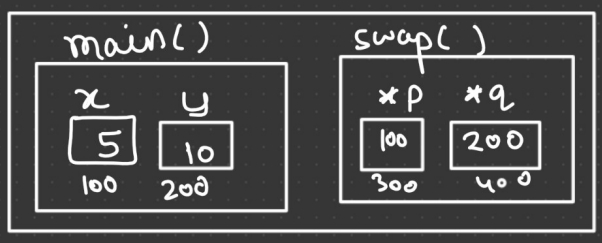
*p = *(100) = value at 100

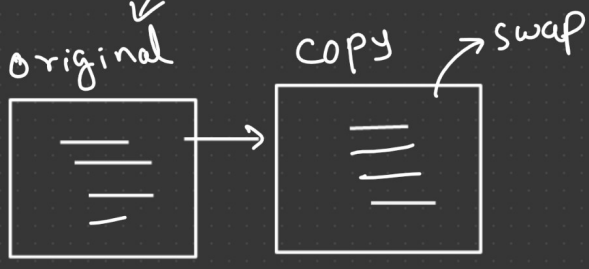
*(100) = 10;

```
int main()
{
    int x=5, y=10;
    printf("Before swapping x=%d, y=%d", x, y);
    swap(&x, &y);
    printf("After swapping x=%d, y=%d", x, y);
    return 0;
}
```

① swap(x, y); — copy

② swap(&x, &y); — permission





swap(&x, &y)

```
void swap(int *p, int *q)
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

// int *p=&x; int *q=&y;

```
void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}
```

*p = x

int int

int *t = 5

R to L

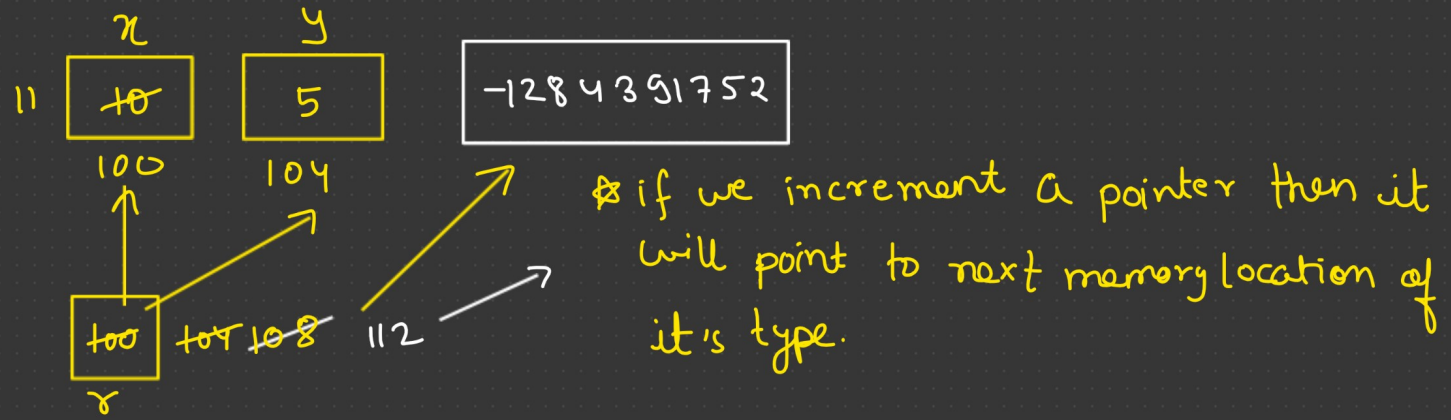
int *r = &x;

++*r = 11 ⟶ ++(*r) = ++x = 11

*++r = 5 ⟶ *(++r) = 5 (Garbage)

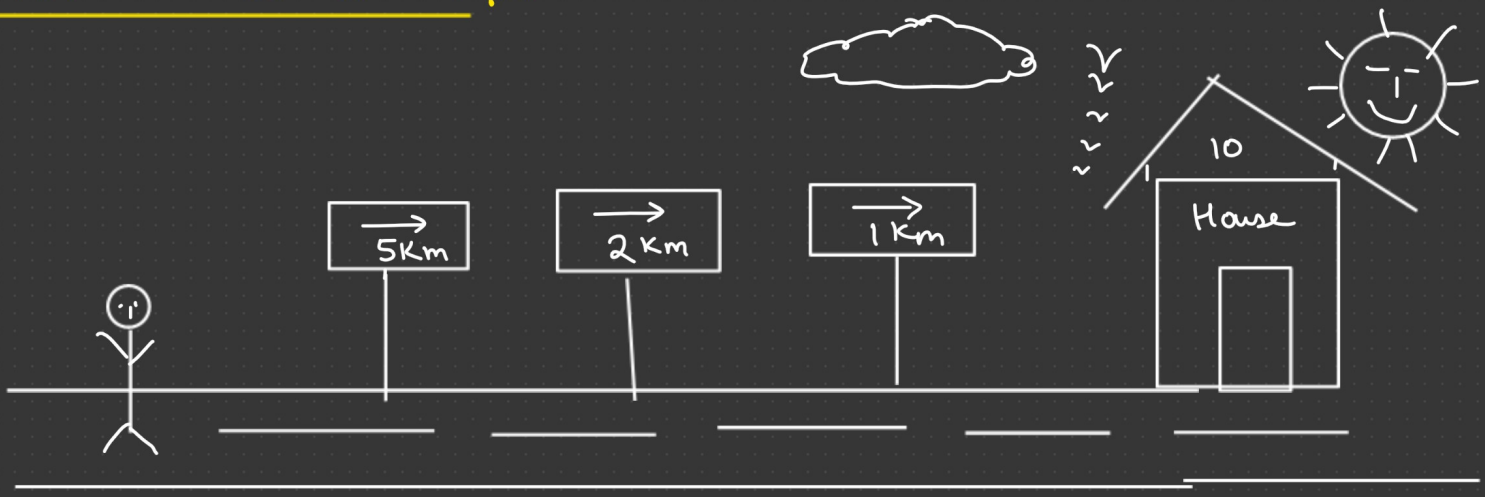
*r++ = 5 ⟶ (*r) = 5 (Garbage) =

(*r)++ = -1284391752



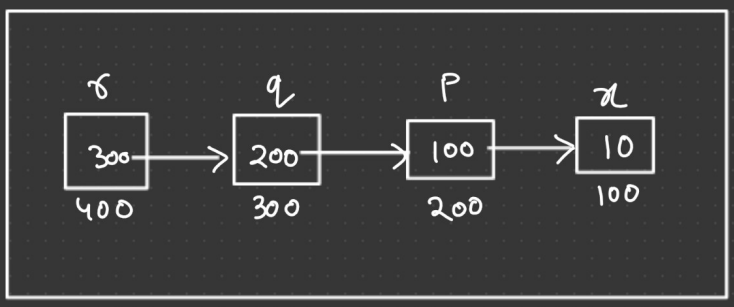
* if we increment a pointer then it will point to next memory location of it's type.

Pointer to Pointer :-



int *** r → int ** q → int* p → int x;

```
int x = 10;
int *p= &x;
int **q = &p;
int ***r = &q;
```



P = 100 , *P = 10 ; &P = 200

	400	300	200	100	10
→ r	&r	r	*r	**r	***r
→ q	-	&q	q	*q	**q
→ p	-	-	&p	p	*p
x	-	-	-	&x	x

* value at ①
& address ②
~~(*&)~~ ③

P = &x
q = &p
r = &q

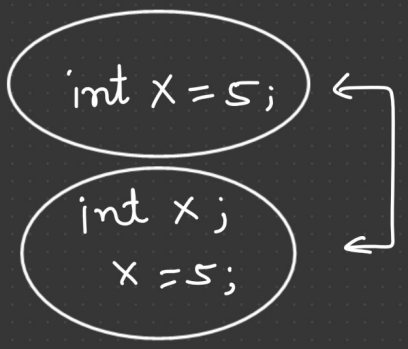
***r = **q = *p = x = 10

**q = **&p = *p = *&x = x = 10

***r = **&q = **q = **&p = *p = *&x = x = 10

**r = **&q = *q = *&p = p = &x = 100

1) int x = 5;
int *p = &x; ✓



2) int x = 5;
int *p; ✓
p = &x;

3) int *p; ✗
*p = &x; //wrong

1) ++***q = ++***&p = ++*p = ++*&x = ++x = 11 ✓

2) ***r + **q = 10+10 = 20

Returning pointer from a function:-

```
int main()
{
   int x = 10;
   int *p = fun(&x);
}
```

```
int * fun(int *q)
{
   return q;
}
```

```
int *p;
p = fun(&x);
```

```
int main()
{
   int *p;
   p = fun();
}
```

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

logically incorrect

Dangling Pointer

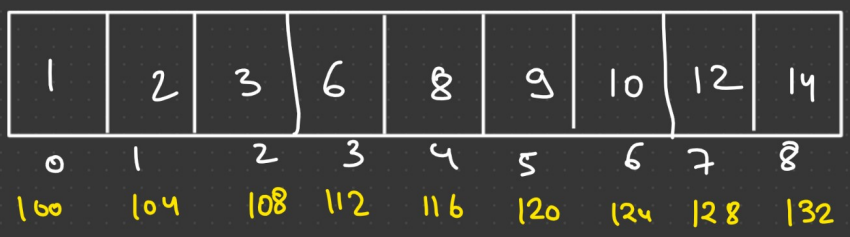
Array & Pointers :-

int a[9] = {1,2,3,6,8,9,10,12,14};

1) int *p = a ; ✓

2) int *p = &a[0]; ✓

3) int *p = &a; ✗



P = &a[0]

*P = *&a[0] = a[0] = 1

*(P+1) = a[1] = 2

*(P+2) = 3

(P+6) = 124 = &a[6]

a[2] = *(P+2)

a[i] = *(P+i)

*(P+i) = *(i+P) = *(a+i) = *(i+a) = a[i] = i[a] = P[i] = i[P]

2D Array & Pointer :—

```
int a[4][4];

int *p = a;
```

	100	104	108	112	
	0	1	2	3	
100 → 0	1	2	3	4	→ a+0
116 → 1	5	6	7	8	→ a+1
132 → 2	9	10	11	12	→ a+2
148 → 3	13	14	15	16	→ a+3

p = 0th row = 100

p+1 = 1st row = 104

p+2 = 2nd row = 108

p+3 = 3rd row = 112

*(p+0) = 1

*(p+1) = 2

*(p+2) = 3

*(p+3) = 10;

program → 1) Create 2D array

int (*p)[4] = a; → 2) pointer that points to 2D array.

3) Print [p, *p], [(p+1), *(p+1)]

same same

Q:- WAP to reverse elements of array by passing array to function using pointers ?

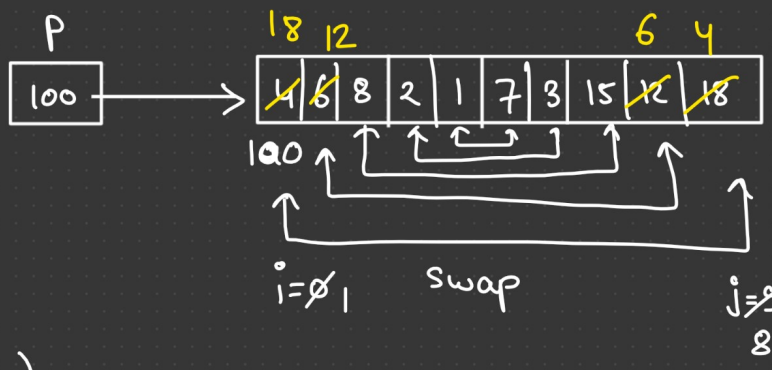
```
int main()
{
   int a[ ] = {4, 6, 8, 2, 1, 7, 3, 15, 12, 18};
   reverse(a, 10);
   for(i=0; i<10 ; i++)
     printf(" %d", a[i]);
   return 0;
}

void reverse( int *p, int n)
{
   int i, j, temp;
   for(i=0, j= n-1; i<=j ; i++, j--)
   {
        temp = p[i];    // *(p+i)
        p[i] = p[j];
        p[j] = temp;
   }
}
```

int * p = &a ;

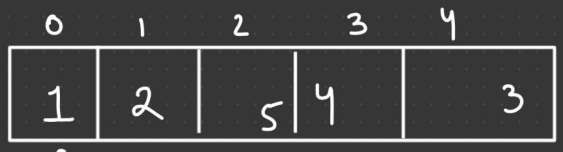
↓

wrong

∵ name of array itself points to base address.

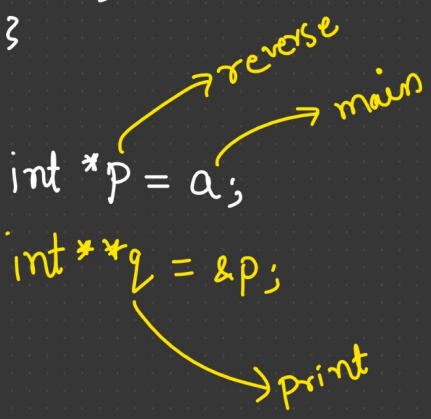


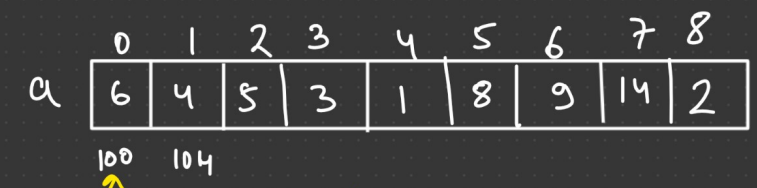
Sorting:-

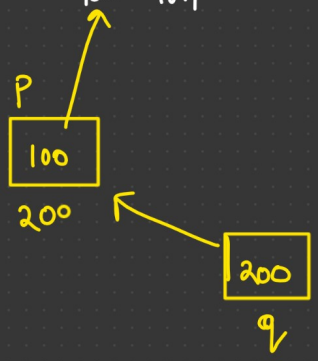


i = 1 j = 2 & 4

```
for(i=0; i<n-1; i++)
{
  for(j=i+1; j<n; j++)
  {  if(a[i]>a[j])
     {
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
     }
  }
}
```







**q = ** &p
= *p = *a = a[0]

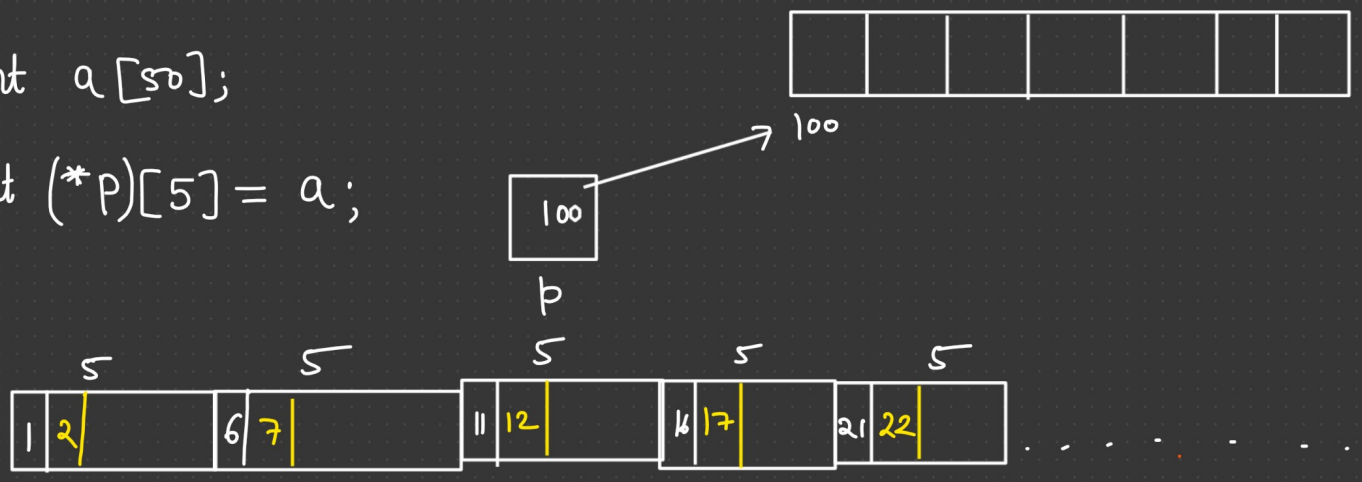
*(*q + 1) = 4

*(*q + 5) = 8

*(*q + i) = a[i]

Pointer to a Array :-

```
int a[50];
int (*p)[5] = a;
```



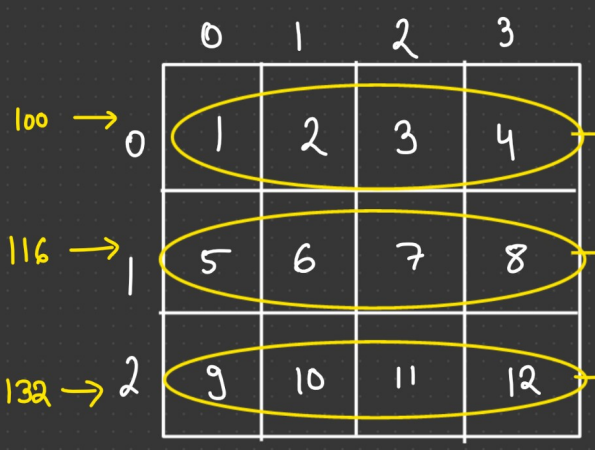
Pointer to 2D array using Pointer to array.

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

```
int (*p)[4] = &a;
int (*p)[4] = a;
```

`&a = &a[0]`

`a = &a[0][0]`



* p = Address of 0th element of 0th row = &a[0][0]

p = points to 0th row having 4 int block. = &a[0]

p+1 = points to 1st row having 4 int block = &a[1]

*(p+1) = Address of 0th element of 1st row = &a[1][0]

p+2 = points to 2nd row having 4 int block = &a[2]

*(p+2) = Address of 0th element of 2nd row = &a[2][0]

(*p+1) = (*(p+0)+1) Address of 1st element of 0th row = &a[0][1]

(*(p+1)+1) = Address of 1st element of 1st row = &a[1][1].

(*(p+2)+1) = Address of 1st element of 2nd row = &a[2][1]

((p+0)+1) = *&a[0][1] = a[0][1] ⇒ value of 1st element at 0th row = 2

((p+1)+1) = *&a[1][1] = a[1][1] ⇒ value of 1st element at 1st row = 6

((p+2)+1) = *&a[2][1] = a[2][1] ⇒ value of 1th element at 2nd row = 10

Final Conclusion :- *(*(p+2)+1) = a[2][1] ✓

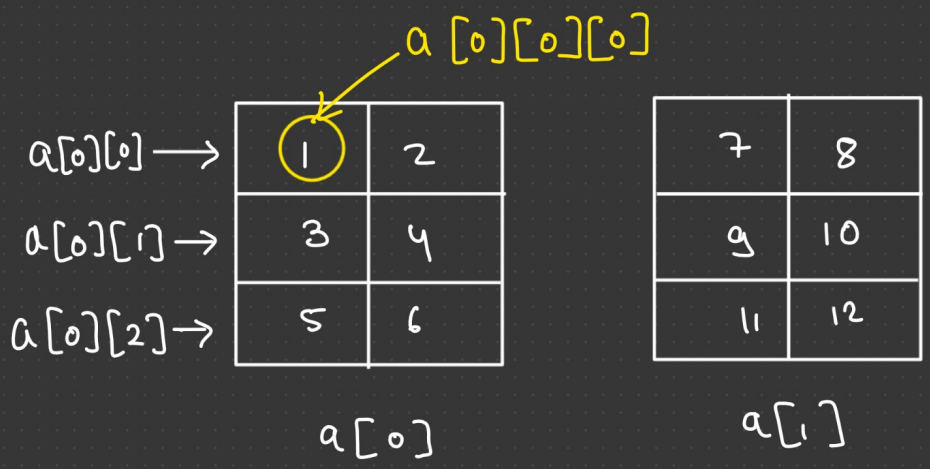
→ *(*(p+i)+j) = a[i][j] ✓

Same as *(p+i) = a[i]

∴ we can say that *(*(*(p+i)+j)+k) = a[i][j][k]

Pointer to 3D array :-

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

int (*p)[3][2] = &a;
```



$p = \&a[0]$

$p+1 = \&a[1]$

$*p = *\&a[0] = a[0]$
$= 0^{th}$ 2D array

$*(p+1) = *\&a[1] = a[1]$
$= 1^{st}$ 2D array

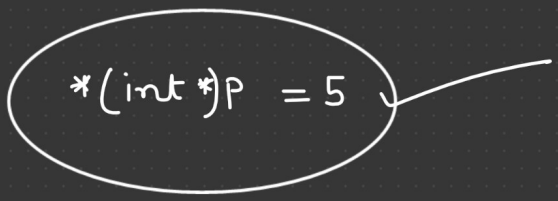
$*(*(p+0)+0) = a[0][0] \Rightarrow *(*(*(p+0)+0)+0) = a[0][0][0]$

$*(*(p+1)+0) = a[1][0] \Rightarrow *(*(*(p+1)+0)+1) = a[1][0][1]$

Types of Pointers :-

1) **Void Pointer :-** (Generic pointer) = It can point to any datatype.

```
int main()
{
    int x = 10;
    char y = 'a';
    void *p = &x;
    printf("%d", *(int*)p);
    p = &y;
    printf("%c", *(char*)p);
    return 0;
}
```



NULL Pointer

```
int *p = NULL;
```

Wild Pointer

```
int *p;
```
→ invalid location

Dangling Pointer

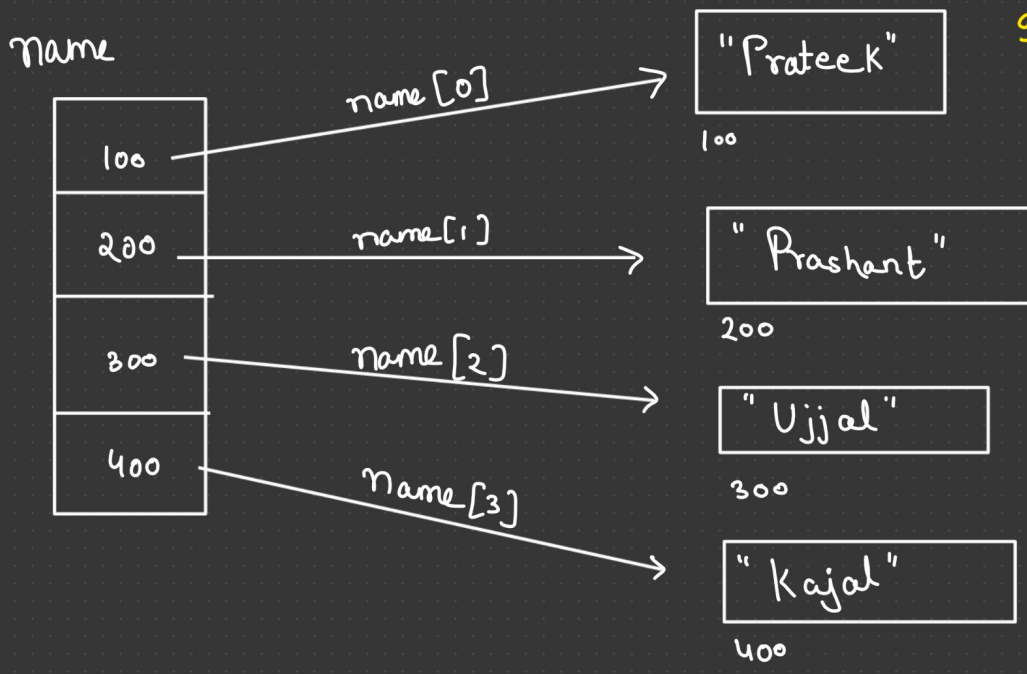
```
int main()
{
    int *p = fun();
    return 0;
}
```

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

→ Pointer that points to non-existing memory

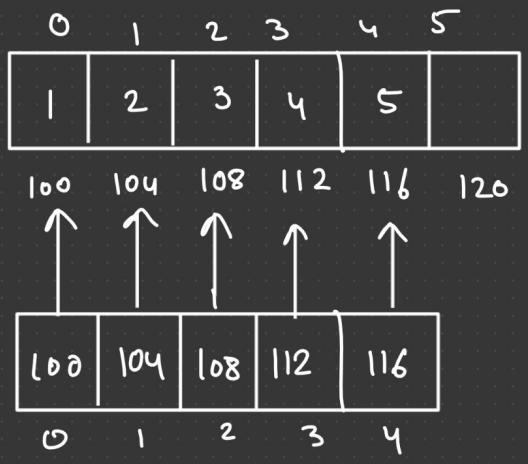
Array of Pointers

```
Char *name[20] = { "prateek", "Prashant", "Ujjal", "kajal" };
```



9555031137

```
int a[5] = { 1, 2, 3, 4, 5 };
int *p[5];
```




```
p[0] = &a[0];        *p[0] = a[0]
p[1] = &a[1];        *p[1] = a[1]
p[2] = &a[2];        *p[2] = a[2]
p[3] = &a[3];           :
p[4] = &a[4];           :
```

Const Pointer :-

```
int x = 10, y = 5;
int * p;
```

p = &x; ✓	p++; ✓
p = &y; ✓	

```
int *const q;
```

q = &x; ✓

q = &y; ✗

q++; ✗

→ Const pointer

DMA (Dynamic Memory Allocation)

If we want to allocate memory at runtime (when program is in execution). Then we need dynamic memory allocation.

We have 4 function for it :-

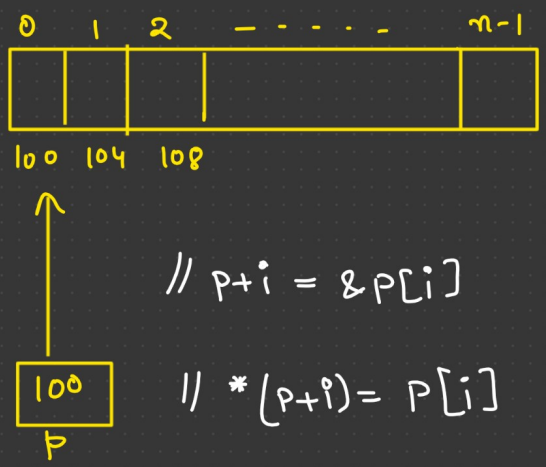
1) malloc() → It allocate the "n bytes" of memory with garbage value.
2) calloc() → It allocate sequence of blocks in bytes (like array) with zero as a default value.
3) realloc() → It helps to change the size the memory at runtime.
4) free() → It helps to free our allocated memory.

⚠ We must need to include `#include <stdlib.h>` header file to use above functions.

1) Malloc() :-

```
int main()
{
    int n, i;
    int *p;
    printf("Enter value of n");
    scanf("%d", &n);
        p = (int *)malloc(n * sizeof(int));
    for(i=0; i<n; i++)
    {
      scanf("%d", p+i);
    }
    for(i=0; i<n; i++)
    {
      printf("%d", *(p+i));
    }
    return 0;
}
```

It will return void pointer
∴ we need type casting.

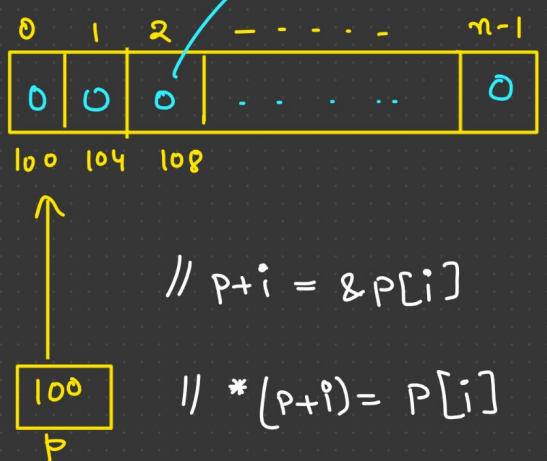


2) calloc() :-

```
int main()
{
    int n, i;
    int *p;
    printf("Enter value of n");
    scanf("%d", &n);
        p = (int *) calloc (n, sizeof(int));
    for(i=0; i<n; i++)
    {
      scanf("%d", p+i);
    }
    for(i=0; i<n; i++)
    {
      printf("%d", *(p+i));
    }
   return 0;
}
```

It will return void pointer
∴ we need type casting.

By default zero value.



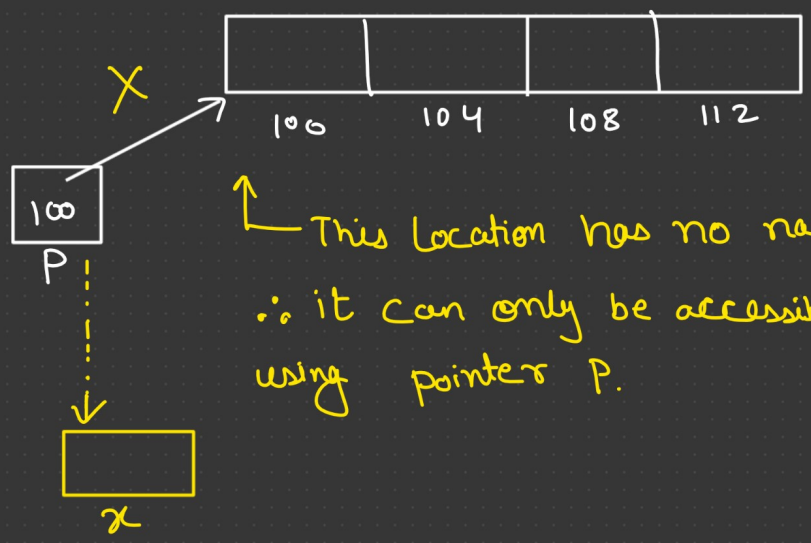
3) free() :-

```
int *p = (int *) calloc (4, sizeof(int));
```

p = &x;

↓

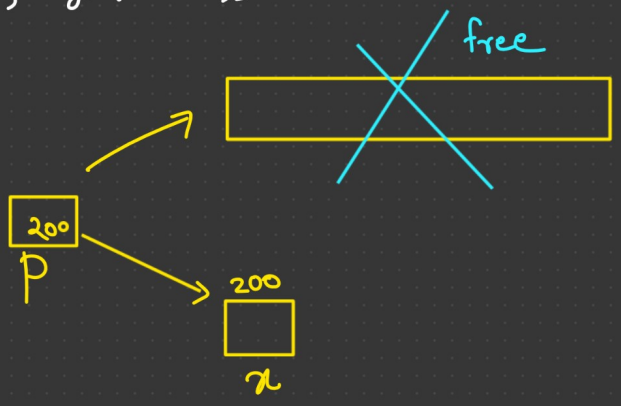
if our pointer points to other location then memory allocated by calloc is not accessible.



↳ This location has no name ∴ it can only be accessible using pointer P.

⁂ So, before changing the address of any pointer. first we need to free that memory location that is going to be unused / inaccessible in future.

```
int *p = (int *) calloc (4, sizeof(int));
free(p);
p = &x;
```

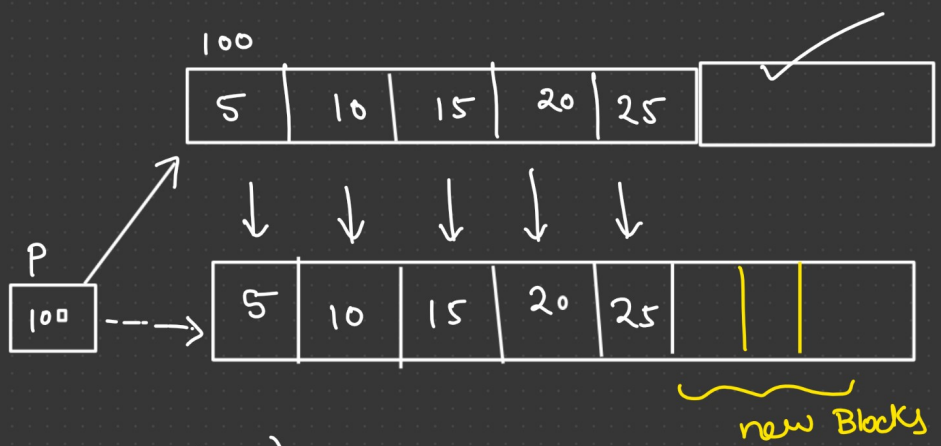


Note:- It is your responsibility to free all memory that is allocated by you using malloc, calloc or realloc.
This unused memory is also called as "Memory Leak"

4) realloc():—

```
int *p = (int*) calloc( 4, sizeof(int));
```

```
p[0] = 5;
p[1] = 10;
p[2] = 15;
p[3] = 20;
p[4] = 25;
```




```
p = realloc( p, 8* sizeof(int));
```

* By default values in new blocks will have garbage values.