* Pointers :-

1) Pointer -

Pointer is a variable which stores address.

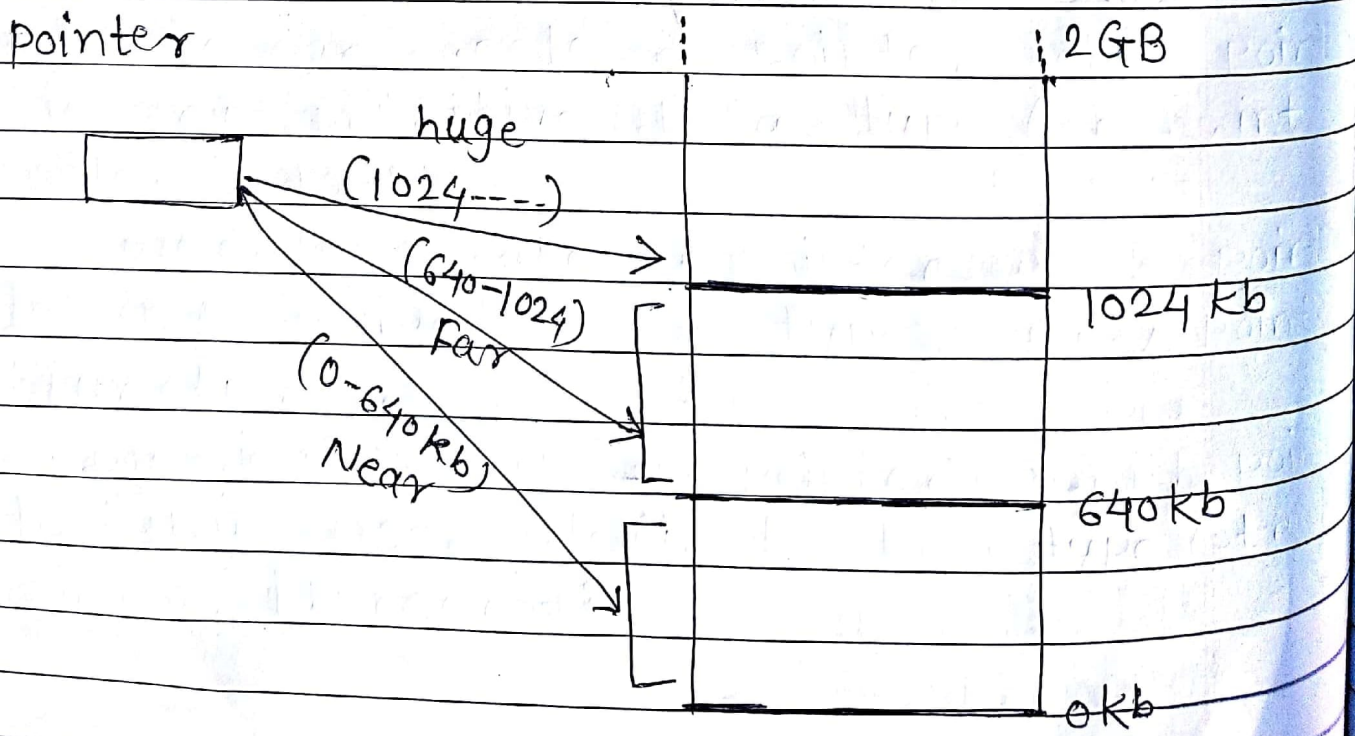
As pointer stores the address an address is always integer means pointer size is always of 8 bytes.

2) Types of pointers :-

In old operating system (Real mode) there are three types of pointers are

1) Near pointer
2) Far pointer
3) Huge pointer

This types of pointers are not applicable in today's operating system. because the latest OS works in protected mode.

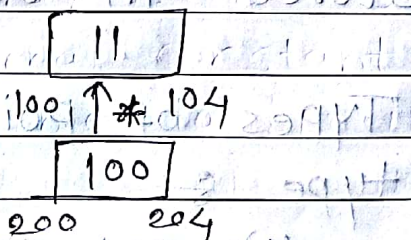


If pointer points in between 0 to 640 kb then it is near pointer.
If pointer points in between 640 - 1024 kb then it is far pointer.
If pointer points 1024 kb and above it is huge pointer.
All this types are not applicable in latest operating system.

3) pointer creation :-

To create the pointer we have to use special operator (*).
* is considered at the reference operator.
To store the address inside pointer we have to use address of operator. (&).

```
int no=10;
int *p=&no;
```



no is a variable of type integer currently initialise with value 11.
p is a pointer which holds the address of integer, currently it holds address of no. where no is the variable of type integer intialise with value 11.

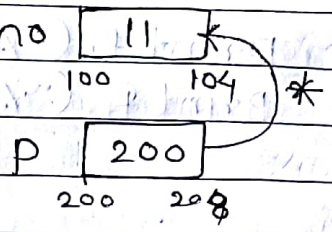
```
printf("%d", no); // 11
printf("%d", &no); // 100
```

```
printf ("%d", p); // 100
printf ("%d", &p); // 200
printf ("%d", *p); // 11
```

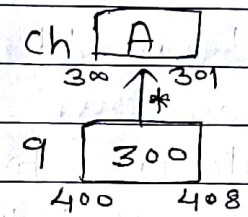
4) The reference operator is (*)
To create the pointer we have to use this operator.
If we want to access the contents of the pointed data then we use * operator to fetch its value.
In above example when we specify *p we get the result as 11 because 11 is the value which is stored in variable no and address of no is stored in our pointer P.

5) Types of pointer according to data type :-
we can create a pointer which points to any primitive data type (char, int, float, double)
we can create a pointer which points to any derived data type (array, pointer function).
we can create a pointer which points to any user defined data type (structure union, Enumeration)

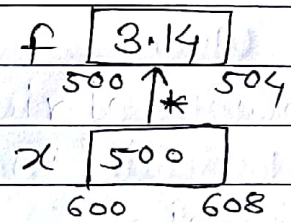
1) int no = 11;
int *p = &no;



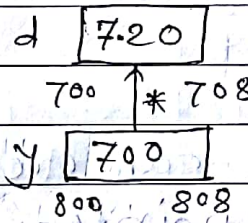
2) char ch = 'A';
char *q = &ch;



3) Float f = 3.14;
float *x = &f;



4) double d = 7.20;
double *y = &d;

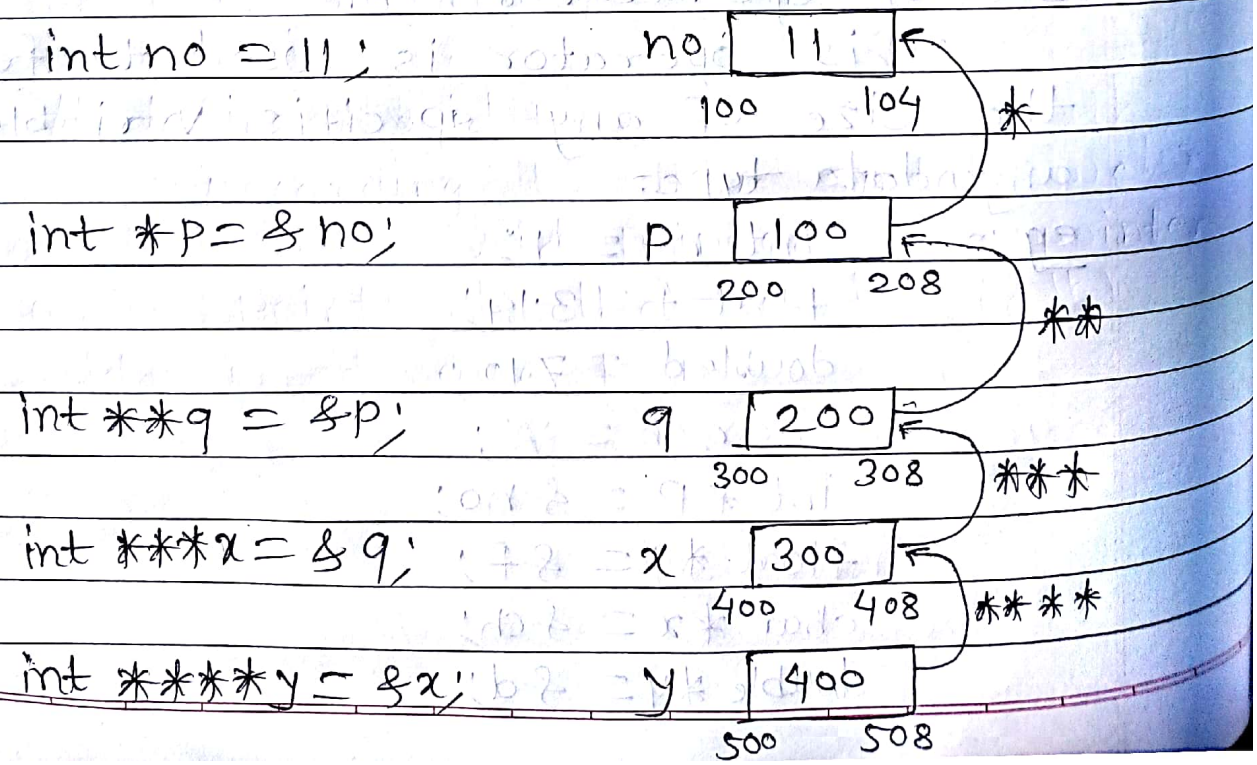


6) Size of operator :-
This operator is used to fetch the size of any specific variable of any data type.

eg :- int no = 11;
float f = 3.14;
double d = 7.10;
char ch = 'A';
int *p = &no;
float *q = &f;
char *x = &ch;
double *y = &d;

```
printf("%d", sizeof(no)); // 4
printf("%d", sizeof(d)); // 8
sizeof(ch); // 1
sizeof(p); // 8
sizeof(int); // 4
sizeof(float); // 4
sizeof(double); // 8
sizeof(x); // 8
sizeof(y); // 8
sizeof(*p); // 4
sizeof(*x); // 1
sizeof(*y); // 8
sizeof(3.10); // 8
sizeof(11); // 4
sizeof(3.14f); // 4
```

7) We can create pointer to pointer, pointer to pointer means our pointer holds address of another pointer.



```
printf ("%d", no); //11
pfint f ("%d" &no); //100
printf ("%d" &p); // 200
printf ("%d", *p); // 11
Printf ("%d", &x); // 400
printf ("%d" **y); // 200
printf ("%d" &q); // 300
printf ("%d" **q); // 11
printf ("%d" ***x); // 11
printf ("%d" ****y); // 11
Printf ("%d" **y); // 200
printf ("%d" &(**y)); // 300
printf ("%d" &(***x)); // 100
printf ("%d" &(*q)); // 200
```

8) Real mode vs protected mode :-

- This are the two types of modes in which the operating system executes.
- In case of Real mode only single process executes at a time whereas in case of protected mode more than one process executes at a time.
- In case of Real mode first mb of the memory is accessible whereas in case of protected mode, the whole RAM is accessible.
- In case of Real mode if first process is running then all the other processor are in waiting state. But in case of protected

mode multiple process can execute simultaneously.
- In todays operating system we start in a real mode and shift into the protected mode. at time of shutdown again our os goes into the real mode.

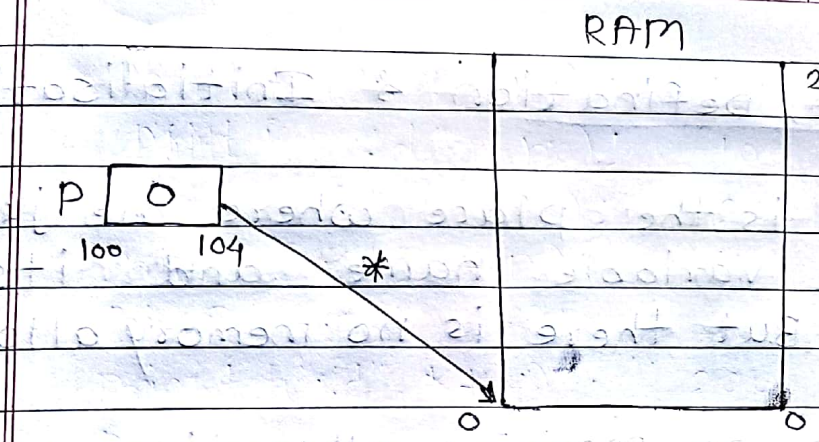
9) NULL Pointer :-

- When we create any variable it gets initialized with its default value.
- But in case of pointer we have to initialise it with an inbuilt keyword. i.e NULL.

It is considered as good programming practice to initialise pointer to NULL because it avoids runtime accidents.
- In almost every header file of c Programming NULL is an inbuilt macro which is internally defined as.

define (void *) 0

According to above syntax we conclude that void pointer is a pointer which points to the zeroth (0^{th}) address of our RAM.



eg. If we want to initialise the pointer to a specific value then these is no need to initialise it with NULL.

```
int no = 11;
int *p = &no;
```

Consider the below syntax where we create the pointer and then initialise it with the NULL.

```
eg: int *p = NULL; //Defination
    int no = 11;   //Defination
    // code
    p = &no;       // initialisation
```

NULL is used to initialise uninitialised pointer.

It is good programming practice to initialise every pointer NULL.

10) Declaration, Defination & Initialisation -

- Declaration is the place where we just declare the variable name and its data type. But there is no memory allocation.

- As there is no memory allocation we cant initialise at the point of declaration

- Defination :-
  It is the place where we allocate the memory.
  As we allocate the memory we can also initialise with some value

- Initialisation :-
  Initialisation mean storing some value into the allocated memory.

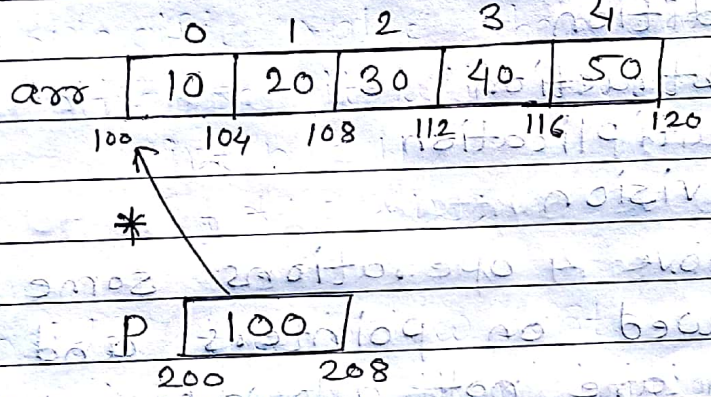
eg.

① Extern int i; // Declaration
  int j; // Defination
  int k=21; //Defination with initialisation
  extern int a=25; // not allowed

11) We can create a pointer which points to an array.

example :-

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = NULL;
p = arr;
```



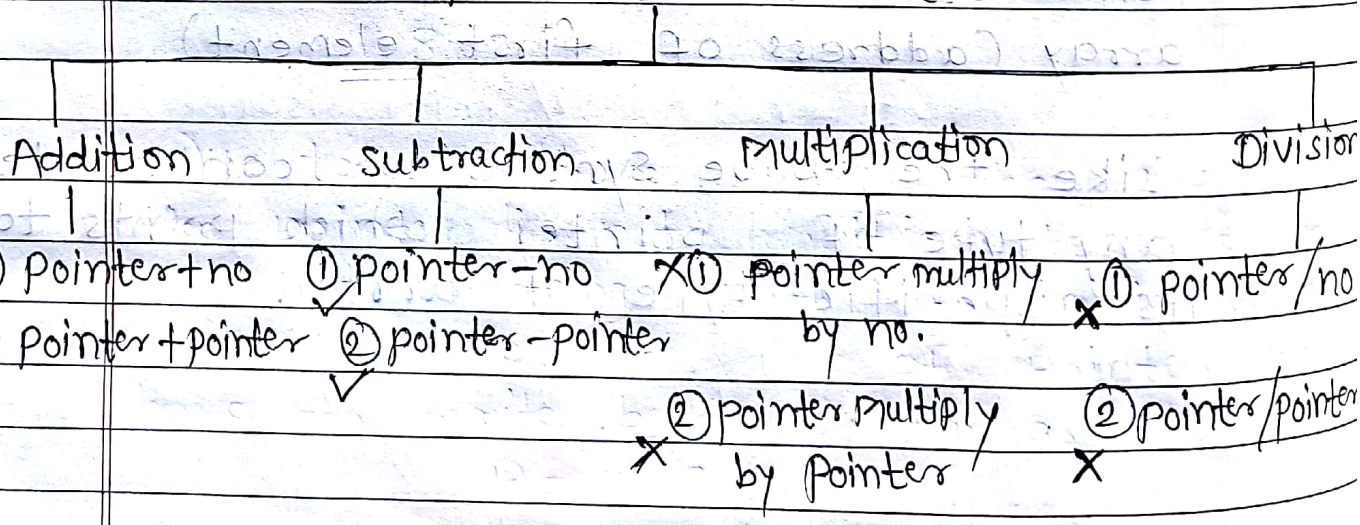
- According to above dia. our pointer p points to the base address of the array (address of first element)

- Like the above syntax we can create any type of pointer which points to its specific type of array.

12) Pointer Arithmatic :-

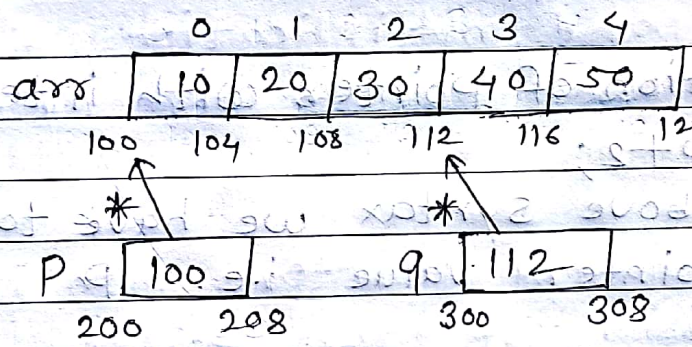
- Pointer arithmatic means the arithmatic operations that we can perform on the pointer.
- There are four types of Arithmatic operations are
  1. Addition
  2. subtraction
  3. Multiplication
  4. Division

From Above 4 operations some operations are allowed on pointers and some operation are not allowed due to memory restrictions.



example :-

```
int arr[] = {10, 20, 30, 40, 50};
int *p = NULL;
int *q = NULL;
p = arr; // p = &(arr[0]);
q = &(arr[3])
```




```
printf("%d", arr);          // 100
printf("%d", &arr);         // 100
printf("%d", arr+1);        // 104
printf("%d", arr+3);        // 112
printf("%d", (&arr)+1);     // 120
printf("%d", &(arr[0]));    // 100
printf("%d", &(arr[4]));    // 116
printf("%d", p);            // 100
printf("%d", q);            // 112
printf("%d", *p);           // 10
printf("%d", *q);           // 40
printf("%d", *(p+2));       // 30
printf("%d", *(q+1));       // 50
printf("%d", *(q-2));       // 20
printf("%d", q-p);          // 3
```

```
printf("%d", p*2);   // error
p+q;                 // error
p*q;                 // error
p/3;                 // error
p/q;                 // error
```

1) Addition of pointer :-

① Addition of pointer with integral constant

eg. p+2;

in above syntax we have to add two in our pointer value i.e. p

```
p+2 * sizeof(int);
p+2 * 4;
p+8;
100+8;
108;
```

Due to the Above syntax the value of the pointer is not modified because we are not updating with its initial value.

To update the value of pointer we have to write below syntax as

```
p = p+2;
p = p+2 * (sizeof(int));
p = p+2*4;
p = p+8;
p = 100+8;
p = 108;
```

After this syntax our p pointer points to the address 108.

② Addition of two pointers :

eg. p+q; // not allowed

Due to memory restriction it is not allowed.

2) Subtraction of pointer

① subtracting integral const. from pointer.

eg.

```
q = q-2;
q = q-2 * sizeof(int);
q = q-2 * 4
q = q-8
q = 112-8
q = 104
```

In above syntax sizeof(int) indicates the size of pointer type.

② subtracting two pointers from each other:-

eg.

```
q-p
(q-p)/ sizeof(int);
(q-p)/4
(112-100)/4
12/4
3
```

In above syntax we divide by size of pointer.

- Subtraction of two pointer is allowed if both the pointers are pointing to some memory region (some array).
- Both the pointers should be of some type

3) Multiplication -

multiplication of pointer with integral constant and multiplication of two pointer is not allowed due to memory restrictions.

4) Division -

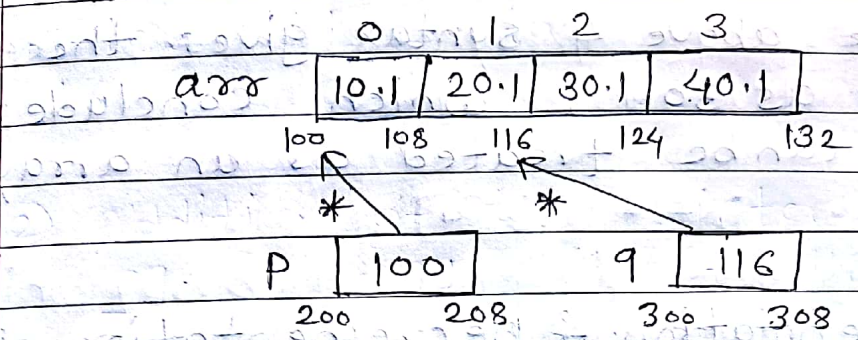
division of pointer with integral const. and division of two pointer is not allowed because of the memory restriction.

```
q*2;
p*2;
p/q;
p/3;
```
} not allowed.

13) Array is internally considered as pointer

when we consider the array its address gets considered automatically by the operating system.

eg.
```
double arr[] = {10.1, 20.1, 30.1, 40.1};
double *p = NULL;
p = arr;
```



double *q = &(arr[2]);

consider the below syntax of array as

arr[2];

↓

*(arr+2);

↓

*(2+arr);

↓

2[arr];

All the above syntax gives the same output. i.e: 30.1 which concludes that array is internally considered as pointer.

14) An array is considered as pointer we can also treat pointer as array.

consider the same Example as above.

*(p+2);

*(2+p);

p[2];

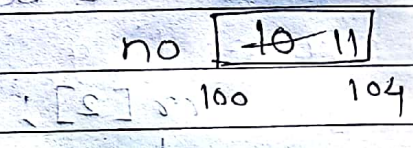
2[p];

All the above 4 syntax gives the same output as 30.1; which concludes that pointer can be treated as an array.

15) Incrementation & Decrementation of pointer.

1) ++, -- are considered as the increment and decrement operators.

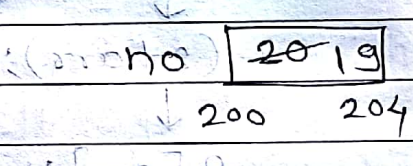
eg: int no = 10;



no++;

The above syntax is internally treated as,

int no = 20;



no--;

above syntax is internally treated as,

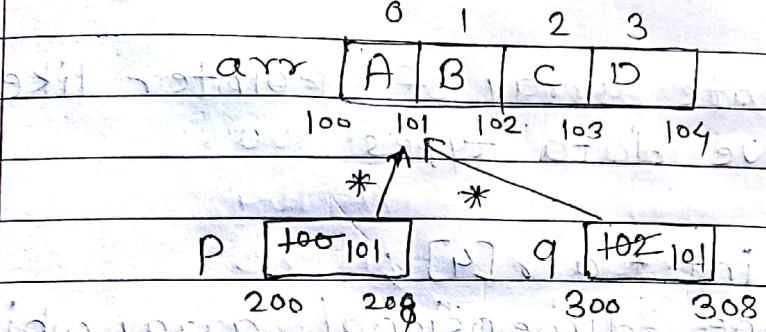
no = no - 1;

Like above operation an integer we can perform the same operation on pointer.

syntax: char arr[] = {'A', 'B', 'C', 'D', };

char *p = arr;

char *q = &(arr[2]);




```
printf("%c", *p);      // A
        p++;
        p = p+1;
        p = p+1*(sizeof(char));
        p = p+1*1;
        p = 100+1;
        p = 101
printf("%c", *p);      // B
printf("%c", *q);
        q--;
        q = q-1;
        q = q-1*(sizeof(char));
        q = q-1*1
        q = 102-1
        q = 101;

printf("%c", *q);      // B
```

16) We can create array of pointer like array of primitive data types

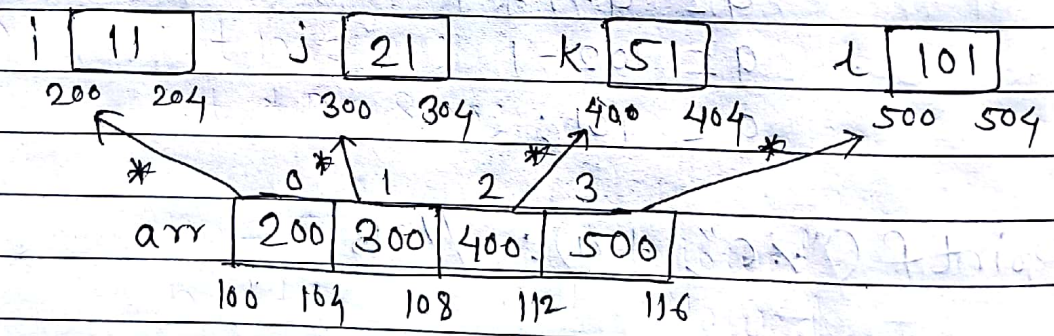
<u>syntax</u> : int *arr[4];

arr is one dimensional array which contains four elements, each element is of type int*.

In simple terms arr is array of pointers

```
int i = 11;
int j = 21;
int k = 51;
int l = 101;

arr[0] = &i;
arr[1] = &j;
arr[2] = &k;
arr[3] = &l;
```

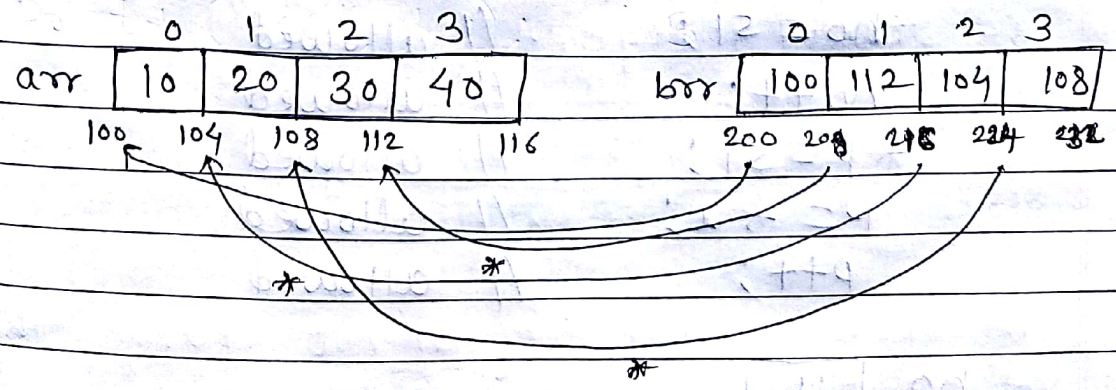



```
print("%d", arr);      // 100
      &arr;            // 100
      arr+1;           // 104
      &arr+1;          // 116
      arr[0]           // 200
```

```
arr[3];          // 500
&arr[2];         // 108
*(arr+1);        // 300
*(arr[2]);       // 21
&j;              // 300
k;               // 51
*(*(arr+2));     // 51
*(2+arr);        // 400
*(3[arr]);       // 101
*(*(2+arr);      // 51
```

17) We can create array which contains pointers in it. where that pointer points to the another member of array.

```
int arr[4] = {10, 20, 30, 40};
int *brr[4];
brr[0] = arr;
brr[1] = &(arr[3]);
brr[2] = arr+1;
brr[3] = arr+2;
```



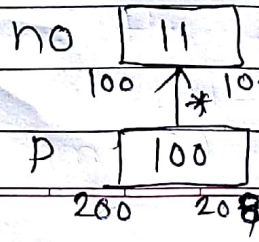
```
printf("%d", brr);   // 200
arr;                 // 100
arr+1;               // 104
arr[1];              // 20
brr[2];              // 104
brr+2;               // 216
*(brr[2]);           // 20
*(*(2+brr));         // 20
*(brr[3]);           // 30
```

18) Constant with pointer :-

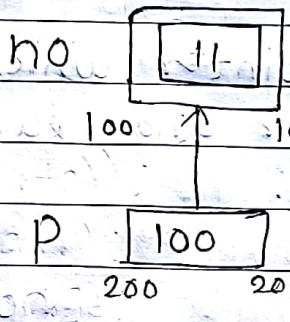
We can use the concept of constant with the concept of pointer. constant is considered as data type Qualifier due to which we can not change the value of any variable.
There are four different variations. as

1)
```
int no = 11;
int *p = &no;

no = 51;     // allowed
no++;        // allowed
*p = 51;     // allowed
p = &x;      // allowed
p++;         // allowed
```



2) const int no = 11;
const int *p = &no;



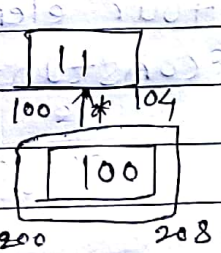
⇒ no is variable of type integer constant currently initialised with 11.

p is a pointer which holds the address of integer constant, currently it holds address of no.

```
no = 51;    // not allowed
no++;       // not allowed
*p = 51;    // not allowed
p = &x;     // allowed
p++;        // allowed
```

3) int no = 11;
int *const p = &no;

⇒ p is a constant pointer which holds address of integer

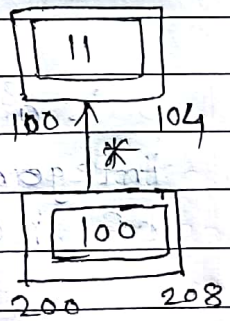


Same
```
no = 51;    // allowed
no++;       // allowed
*p = 51;    // allowed
p = &x;     // not allowed
p++;        // not allowed
```

4) const int no = 11;
const int *const p = &no;
// int const *const p = &no;

⇒ p is constant pointer which holds address of constant integer and currently holds the address of no.

```
no = 51;
no++;
*p = 51;
p = &x;
p++;
```
} not allowed



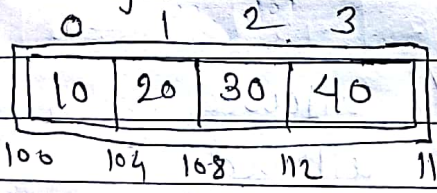
above syntax demonstrate the concept of pointer with the concept of constant.

19) We can create the array of constant.

<u>syntax</u> :-

```
const int arr[4] = {10,20,30,40};
```

⇒ arr is one dimensional array which contains four elements each element is of type constant integer.



in above syntax we can not change the value of the elements of array.

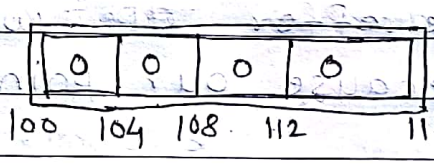
```
arr[0] = 11;    // not allowed
(arr[0])++;     // not allowed
```

20) If array is constant array then it should be initialise immediatly by using the concept of member initiallisation list.

To the constant array member by member initialisation is not allowed.

```
const int arr[4];
```

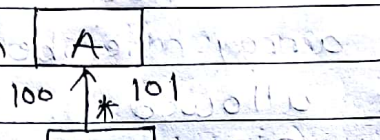
If the above array is global then its default value. i.e. (0) is considered as constant value.



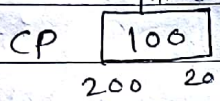
21) Generic pointer (void *) :-

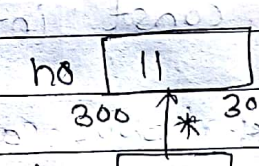
- There are two types of pointer as specific pointer and generic pointer.
- In case of specific pointer the type of pointed data type is already known.
- but in case of generic pointer type of pointed data type is not known.
- If we create character pointer, int pointer, float pointer, double pointer, then the type of pointed data type is already known
- In case of generic pointer we just create the pointer section of the pointed data type.

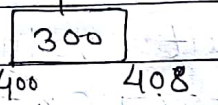
char ch = 'A';
char *cp = &ch;



int no = 11;
int *ip = &no;



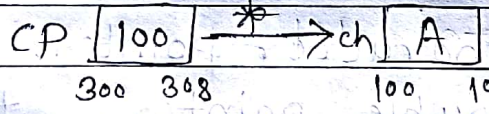


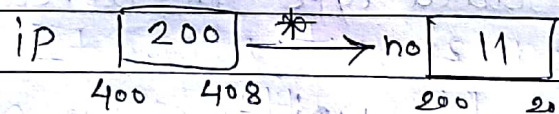


when we dereffer cp we can fetch only one byte because our pointer is character pointer.

similarly, when we dereffer ip we can fetch four byte because our pointer is integer pointer.

char ch = 'A';
int no = 11;
void *cp = &ch;
void *ip = &no;





printf("%c", *cp) // error

in above syntax there is an error because void

pointer Unable to decide how many bytes to fetch. to avoid this problem we can use the concept of type casting.

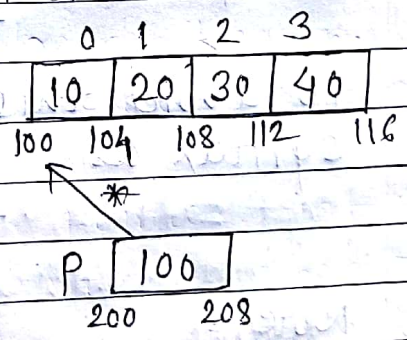
Typecasting is the concept where one data type gets converted into the another data type in a temporary format.

```
printf ("%c", *(char*) cp); // A
printf ("%d", *ip); //error
printf ("%d", *(int*) ip); //11
```

22) Pointer Arithmatic on void pointer.

We have to consider the addition and substraction operation on the pointer.

```
int arr[4] = {10, 20, 30, 40};
void *p = arr;
```




```
printf ("%d", *p); //error
*(int*)p; // 10
p=p+2; //error
```

above syntax generate error because we perform pointer arithmatic of void pointer.

- But in case of void pointer the size of pointer type is not known.
- To avoid this we have to perform typecasting as.

```
p=(int*)p+2;
```

Due to the above typecasting our pointer incremented by 8 byte.

For every pointer arithmatic we have to use concept of type casting.

If we perform incrementation as

```
p++; //error
```

it generates error because our pointer is void pointer.

on some compiler increment operator on void pointer increments its value by one

To avoid this we have to update our syntax as

```
++(int*)p;
```

Note:-

If data type is not known then we can create the void pointer otherwise we can create specific pointer.

23) Function pointer :-

① It is a type of pointer which points to the base address of the function.

② When we compile the program all compile instructions gets stored into the text section. means our function pointer points to the text section.

```
int add (int, int);
    ↓   ↓   ↓↓  ↓↓
int (*fp) (int, int);
```

add is a function which accepts two parameters first is integer and 2nd is integer, that function returns integer.

fp is a pointer which points to a function which accepts two parameters, first is integer, second is integer & that function returns integer.

In c programming there are two technical concepts where the name is indirectly considered as address.

Name of the array and name of the function is internally considered as the address.

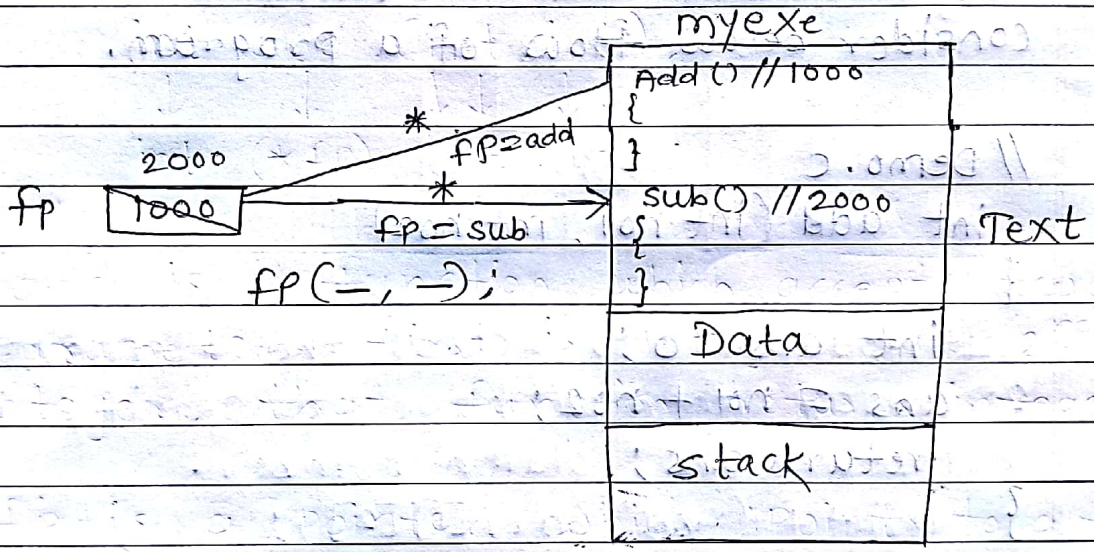
```
//Demo.c

#include<stdio.h>
int add (int, int);
int sub (int, int);
int main()
{
    int x=10;
    int y=5;
    int ret=0;
    int (*fp) (int, int);
    fp=add;  // fp=1000
    ret=fp(x,y)  //ret=add(10,5)
    printf("%d",ret);  //15
    fp=sub;  //2000
    ret=fp(x,y);  //ret=sub(10,5)
    printf("%d",ret);  //05
    return 0;
}
    int add (int no1, int no2)  //1000
{
    int ans=0;
    ans=no1+no2;
    return ans;
}
```

```
int sub (int no1, int no2)   //2000
{
    int ans=0;
    ans = no1 - no2;
    return ans;
}
```

gcc demo.c -o myexe↵

./c. myexe↵



The concept of function pointer is used in case of dynamic link library.

To create the function pointer first we have to consider the prototype of the function.

Like the other pointers function pointer also requires four bytes of memory.

Internal mechanism of function pointer :-

- When we write the function it gets converted into assembly language and assembly lang. function gets converted into the binary language.
- When we create function pointer our pointer points to the function instruction of a function in text section.

consider below flow of a program.

```
// Demo.c
int add (int no1, int no2)
{
  int ans = 0;
  ans = no1 + no2;
  return ans;
}
```

```
// Demo.asm
ADD:
    MOV EAX, no1
    MOV EBX, no2
    ADD EAX, EBX
    RETURN EAX
```

//Demo.obj	//Demo.exe	
57 05 08	0110 1011 0110	← //1000
57 06 09	1101 1011 0111	
61 05 06	1000 0100 1111	
65 05	1011 1101	

In above program the base address our function is thousand.
If you want to store the address of that function then we have to creates its function pointer.

```
int Add (int, int)
 ↓   ↓   ↓ ↓ ↓ ↓ ↓
int (*fp) (int, int)
```

now we have to store address of add function into the fp pointer.

```
fp = Add;    // fp = 1000;
```

now we can call the function by using function pointer fp as

```
ret = fp(10,5); //Add(10,5)
```

In case of function pointer there is no need to use * operator to dereffer the data like the other pointers.

By using the function pointer we can call the targeted fun^n whose address is stored in our pointer.

Date
Page

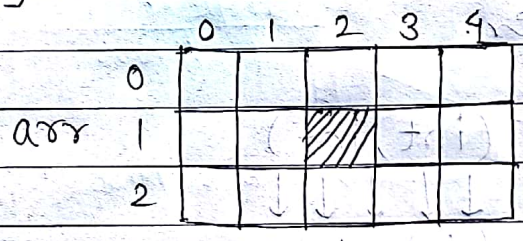
* Multidimensional Array

It is considered as array of array.

eg. int arr[3][5];

⇒ arr is two dimensional array, which contains three one dimensional array, each one dimensional array contains five elements, and each element is of type integer.

Digrammatic Representation:



If you want to access the element from 2D array then we have to specify its row number and column number.

```
printf("%d", arr[1][2]);
```

As one dimensional array is internally considered as pointer we can also represent 2D array as a pointer.

① Arr[3][5];

⇒ *(*(Arr+3)+5);

② Arr[3][2];

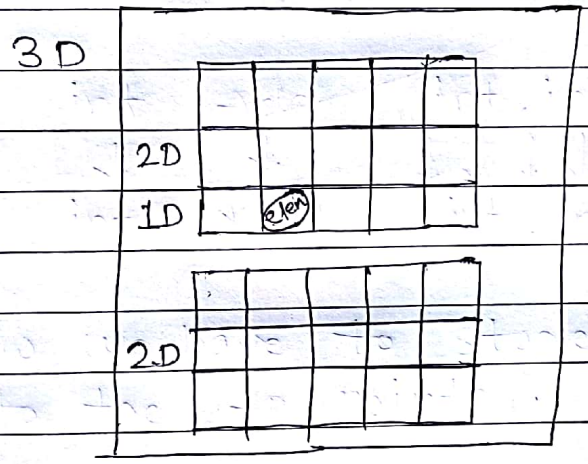
⇒ *(*(Arr+3)+2);

Like the 2D array we can also create 3D array as

float arr [3], [5] [6];

⇒ arr is 3D array, which contains 3 two dimensional array, each 2D array contains 5 one dimensional array, each 1D array contains six elements and each elements is of type float.

double arr [2] [3] [5];



arr is 3D array which contains two 2D array, each 2D array contains 3 one dimensional array each 1D array contains 5 elements and each elements is of type double.

Arr [2] [1] [3]

pointer representation as

((*(Arr+2)+1)+3);