C notes

Data type :-

1] auto (By default)
2] static
3] extern

Stack area	Static area	Heap area
auto variable	Static value	

=> **auto** int a=10; (Stack is used wherever funct fini, Stack Pop) By default; (every time call the value is assigned) (Garbage value)

=> static => only the size is asinged once (at runtime) (the default value is '0')

=> extern => declaring outside the main fun(); (intial value '0')

Example :-

```
{ int a=3;
  if (a!=0)
    print (int a)
  }
  print (int a)
    {
      a--;
    }
```

```
Main ()
{ int i=3;
  if(i--)
  { pf(i);
    main(i)
  }
}
```

without static
infinite Loop

Static mem allocate at runtime (once)

```
i [ 3 ]
i--  [ 2 ] -> [ 1 ]
               [ 0 ]
```

int a=10;   2 bytes data reserved

```
     2 bytes
a = [ 10 ]
    1000
```

int *b; //

```
b = [      ]  4 byte
     2000
```

int c = &a // error as address in 4 byte and we storing in 2 byte int (if not error the int 4 byte)

*b ⇒ value at that address
int *b ⇒ pointer declaration (holds 4 byte address)

⇒ * we use array instead of stack, as we can access any index element, in stack we have to do operation (push & pop)

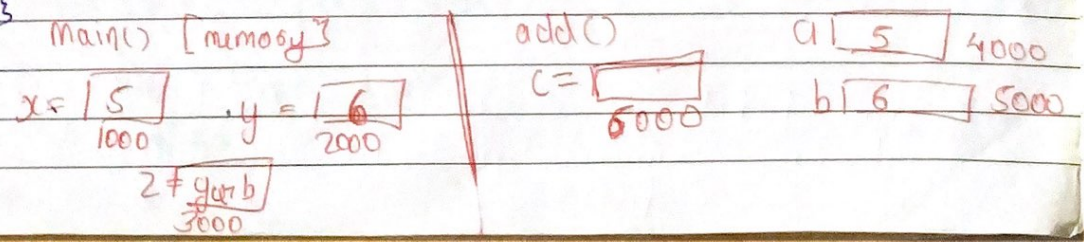
⇒ a[10 50]
- to find from the random index
  - 1000 + (60 - 10) × 2 byte (Data byte size)
    Starting memory loc ↓to access / start address

⇒ So to avoid extra calculation the index start from [0].

* Function :-
- Alway return a value :

```
void main ()
{
  int x = 5, y = 6 ;
  int z
  add (x, y);
  printf ("%d", z);
}
```

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



* call by reference :-

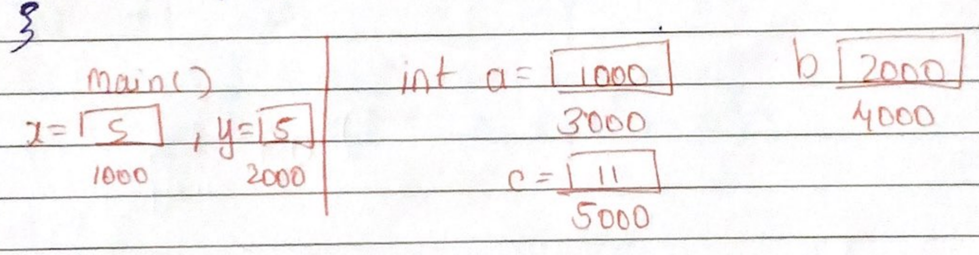
c) when we pass address as a parameter to function then it call by reference.

⇒ To Hold ~~pointer~~ Address we need pointer

Example :-

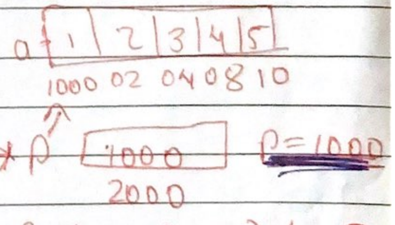
```
{
 int x=5, y=5;
  int *z;
   z=add (&x, &y);
  Pf ("%d", z)
 }
```

```
int* add (int *a, int *b)
{
   int c= *a + *b;
   return &c;
 }
```



* Passing Array to function :-

```
{
 int a[] ={1,2,3,4,5}
 print (a, 5)
 }
                       call by referene, by value
void print (int *p, int x)
 { for (int i=0; i<x ; i++)
       {
        pf(" ", *p);
         p++;
       }
```



=p++   (p+2) 1002
= p+ datatype size
=

☆ to pass address we should have same datatype as if we increment the pointer then it will increment by data size,

Ex. int x=10;
add (&x);
add(float *a);   *a++;
Here the index will increment by 4 byte instead of 2 byte (int datatyp

☆ Pointer questions (Double pointer)

```
int a[] = {0,1,2;3,4,5}
int *p = {a, a+1, a+2, a+3, a+4};
pf ("%u %u %d", p, *p, **p);
```

To print address

a:

0	1	2	3	4
1000	02	04	06	08

p

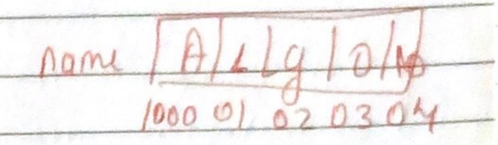
1000	1002	1004	06	08
2000	2004	2008	2012	2016

address will be of 4 byts

P = 2000
*P ⇒ (value at P) = 1000
**P ⇒ 0

Strings :-

char name[] = "Algo"



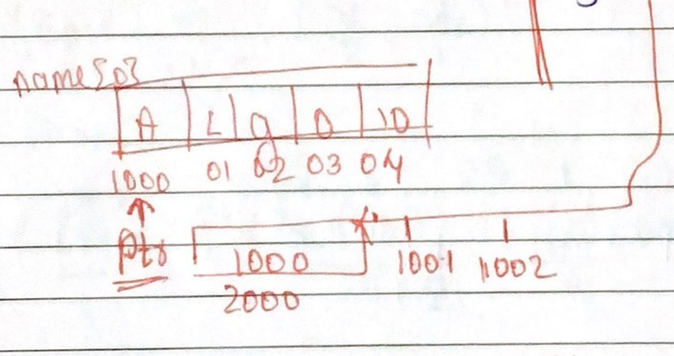
→ At the end of String the compiler will ~~the~~ Add '\0',

Example:- name[i]

```
while (T != '\0')
{ pf ("%c", name[i]);
    i++;
}
```

Pointer

```
char *ptr;
 ptr = name;
while(*ptr != '\0')
{ pf ("%c", *ptr);
    ptr ++;
}
```



To Accept multiverse String
gets(name);
scanf("%s", &name) will not accept 2 words.

IMP If we declare the pointer of type int then the address will be incremented by '2' or '4' byte So pointer and variable array name must be same.

* Read complex pointer :-

operator	Precedence	Associativity
(),[ ]	1	left to right
*, identifier	2	right to left
Datatype	3	

Example :-

```
char (* ptr) [3]
 4    2   1   3
```

```
1] (*ptr) => ( *  ptr)
               2   1
2] [3]    => 3
3] Char   => 4
```

```
            2 1       3  1  2         3   1     2
=>  void   (*ptr)  (int (*)[2] , int (*) (void))
     9       1       5   3  4  2  8   6     7
```

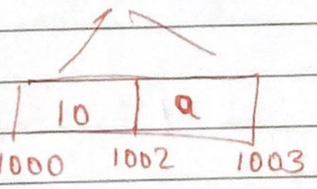
Structure :-

=> To accept the values for different data types.

Example :-

```
Struct node
  {
    int a; //2 byte
    Char b; //1 byte
  };
```

Struct node p = {10, 'a'}



* To Access the element use "$\underset{10}{p.a}$" & "$\underset{a}{p.b}$"

* We can also take Array for Structure

Example :-

```
Struct node p[5];
```

* Variation for declaring Structure.

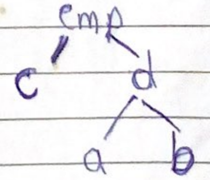
So to access
Emp.c //20
Emp.d.a //10

```
Struct node
 { int a; char b; }
```

```
Struct node1 <— 3 byte —>
{ int c;   Struct node d; }
```

this will total have 5 byte

Stuct nodel Emp



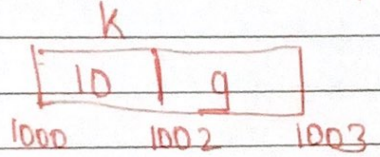
Structure with pointer:-

```
Struct node
  {
    int a;
    char b;
  };
```

```
Struct node *l = &k

p.f("%d", (*l).a) // 10
p.f("%c", (*l).b) // g
```

Struct node k = {10, 'g'};



★ imp
in traditional method we use
l → a (which internally convert to)
(*l).a

// ★ struct node * S[10];

⇒ S[10] is a array which holds the address of structure whose type is node.

★ union :-

⇒ The memory of higest datatype is only assinge (i.e only 2bytes) & in Structure whole 3 bytes will be assigned.

a[0]	a[1]	a[2]	a[3]
0	G		

```
A union test
  {
    int x;
    char a[4];
     int y;
  };
```

```
{
union test t;
  t.x =0
  t.a[1] = 'G';
  pf ("%s", t.a);
return 0;
}
```

print still '\0' is found.

Here 4 bytes will be only take Higest datatype created will be assigned in the memory.