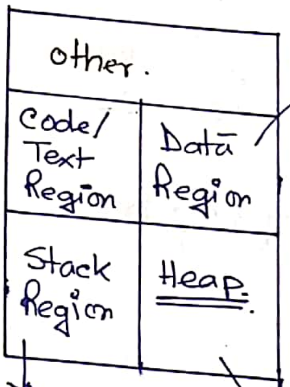
Programming Language → C [gatē wt. : 8-12 marks]

SHORT NOTES

- Memory Regions:



- Static Var
- Global Var.

C program.

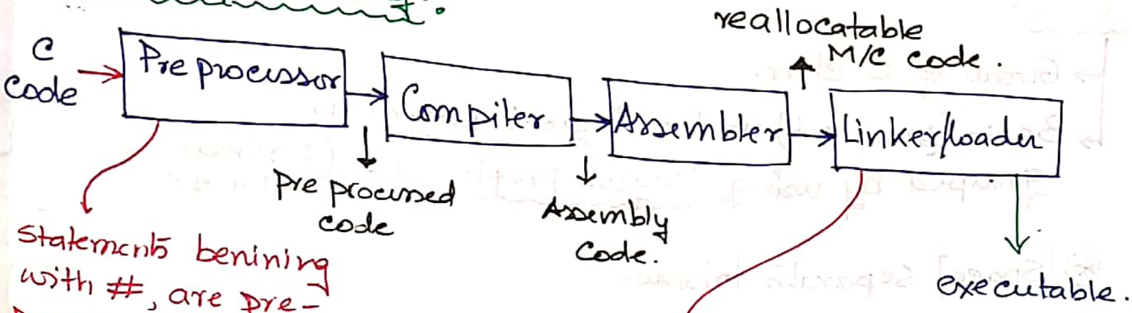
- local variable
- stack frame
- parameters

- Explicit memory mgmt.
- malloc, calloc, . . .

- Static Mem : → Code Region (System) → Data Region (prog.)

- Dynamic Mem : → Stack Region (System) → Heap. (prog.)

- C language processing :



Statements benining with #, are pre-processor directive

→ file inclusion (# include)

→ Macros (# define)

- Linker → Resolves external files and variables.
- Loader → [Performs reallocation]

→ It does the allocation / deallocation.

[Alaways there in main Mem]

- Memory formats (Ex → 2)

1.> Little endian (Reverse order) → [Fist LSB them MSB] → 0000 0010 0000 0000.

2.> Big Endiam [Forward order] → [First MSB, LSB last] → 0000 0000 0000 0010

- C Language/Prog:

CODE

- → Instruction.
- → Statements — Group of Instruction.
- → Programs. — sequence of statements
- → Language — Set of Programs.

DATA

↓

[Final representation of something, that cannot be converted to any other form.]

㊀ Characters → Tokens → Syntax → Semantic ⇒ Logic

Compile.

[maybe correct or wrong.]

- Token:
  - → Group of C char.
  - → Basic (smallest) unit of Logical program, grouped by using longest prefix rule. (Maximal Munch)

㊀ [Space] separates tokens.

⇒ Types of Token:

1. keywords : [int, float, char, ... short, long, signed ..., extern, if, do, while, for, break ... ... etc.]
   ↓
   [Reserved words]
2. operators : [arith, logical, bitwise, Relational ...]
3. Identifiers: → Should not, begin with digit.
4. Constants / literals.
5. Special Symbols.

NOTE : $x = y$;

y → (rvalue) : [Can be anything]

x → (l value) :
- Can't be constant.
- Array Name.

• **Precedence :**

Note : [if equal precedence ⇒ then determined via associativity.]

Precedence Table :

1.> ( ), [ ], . , →, ++ (post), --(post)

● 2.> ++(pre), --(pre), +(unary), -(unary), *, &, !, ~, sizeof( ), (type)

3.> *, /, %

4.> +, -

5> <<, >> [bitwise shift]

6.> <, <=, >, >=

7.> ==, !=

8.> &

9.> ^

10> |

11.> &&

12.> ||

● 13.> ? :

● 14.> =, +=, -=, <<=, /=, *=, >>= . . . [Assignment].

15.> ,

Note:

● → Right associative [R → L]

others : Left associative [L → R].

NOTE :

→ printf( ); returns the no. of characters printed.

→ scanf ( ); returns, the no. of arguments passed.

Ex: scanf("%d %d", &x, &y)

↓ returns (2) → [2 arguments]

• Storage Classifiers:

[might be given or may not be given.]

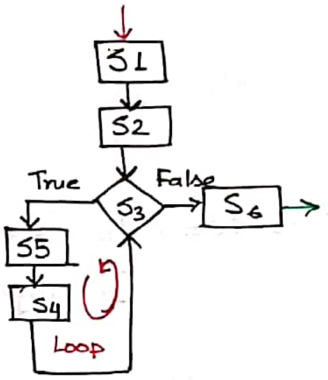
Classes →	Auto	Register	Static	Extern.
Memory Region	Stack	Stack / Reg.	Data	Data
Lifetime	Block.	Block	Program Execution.	~~Pro~~ All program.
Scope	Block.	Block	Block / Program.	All program
Initial value.	Garbage value	Garbage Value	0	0
# Declaration	1	1	≥ 1	≥ 1
# defination	1	1	1	1
Local / Global.	Local.	Local.	Local / Global.	Global.

• Control Structures :

→ for-Loop:

```
S1;
for (S2; S3; S4)
    S5;
S6;
```

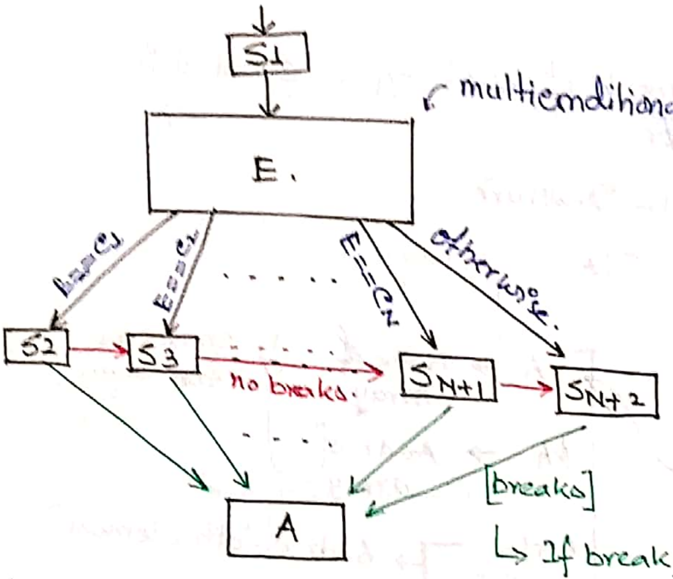
? condition



→ Switch :

```
S1;
switch (E)
 {
   case C1 :      S2 ;
                 <break ; >
   case C2 :      S3 ;
                 <break ; >
    :
    :
   case CN :      SN+1 ;
                 <break;>
   default :      SN+2 ;
                 <break;>
 }
    A;
```

< > → optional.



multiconditional statement.

If there are no breaks, then continue serially until break/end.

↳ If break; → come out of the block.

⇒ if, if-else, do-while, while, etc. are in class notes.

NOTE → Very Important:

- if ( ); while ( ); → Compilation Error. [condition is MUST]

  for (; ;); ← (no Error) → [If condition not given, it is assumed to be TRUE.]

- if (① || ____); → This statement is not executed.

  if (⓪ && ____);

  for (;⓪; ☐); → this is also not executed.

- if (0, 1, 2, 3, ④); → only this condition matters.

  But these are executed.

  same for while (___); and for (;___;);

  if ((x) = ⑤); → First assign, then check variable.

- S1, S2, S3; ⇔ S1; S2; S3;

- Arrays:

↳ Collection of elements of same type, created in contigous manner.

↳ Is a derived data Structure.

⇒ C only supports STATIC Array.

⇒ [ROW MAJOR]

Ex 1: (1-D)

int A[100];

- A → Name of Array.
- &A → Addr of array.

} should be the same.

- A+0, &A[0] ↳ Addr of '0th element.
- A+i = &A[i] = Addr of 'i' th element
- A[i] = *(A+i) = value of 'i'.

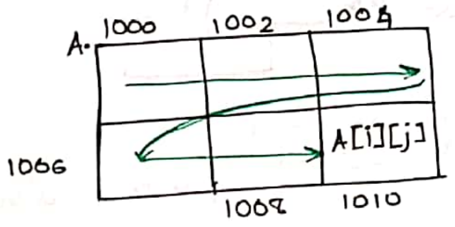
⇒ [&A[i] = A + i × sizeof(element)]

2-D: int A[2][3];

A → arr. name
&A → arr. addr } (same)

A+0 = &A[0] ↳ Addr of Row '0'
A+1 = &A[1] ↳ Addr of Row '1'

A[0] = Row '0' name
A[1] = Row '1' name }



&A[i] = A+i

&A[i][j] ⇒ A[i]+j

⇒ Resolving Addr of A[i][j] using Base Addr (A):

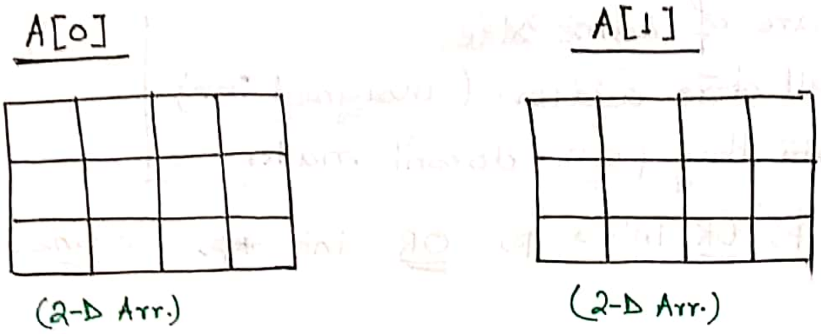
int A[m][n];

&A[i][j] = [No of elements before Row-'i' + No of elements in Row 'i' before col 'j'] × size + A.

∴ $\boxed{\&A[i][j] = [i \times n + j] \times \text{sizeof(int)} + A}$

3-D : int A[2][3][4] ;

$$\text{int } A[\underset{\boxed{\text{Table}}}{2}][\underset{\text{Row}}{3}][\underset{\text{Col.}}{4}];$$



Storage: First store table by table, then Row by Row.

⇒ Resolving Addr of A[i][j][k] : [int A[X][Y][Z];]

&A[i][j][k] = [No of elements in prev 'i' Tables + No of elem. in prev 'j' rows + No of elem. in row 'j' before col 'k'] x size + A.

$$\boxed{\&A[i][j][k] = [i\times(Y\times Z) + (j\times Z) + k] \times \text{size of(int)} + A.}$$

NOTE : int A[5]; → Local (all Garbage) } ✓
→ Global (all 0)

int A[5] = {10, 20, 30, 40, 50} ✓

int A[5] = {10, 20} → [Rem. are taken as 0] ✓

int A[ ] = {10, 20, 30, 40, 50} ✓.
[will be calculated as total no of elements initialised.]

⊛ For 2D array → [col] should be mentioned explicitly.

int [2][3] = {1, 2, 3, 4, 5, 6}; } Possible.
int [2][3] = {1, 2, 3, 4, 5, 6}.

① ②

int A[2][3] = {{1, 2, 3}, {4, 5, 6}} → this list has explicitly defined the no. of cols, Hence → [VALID]

int [2][ ] = {1, 2, 3, 4, 5, 6}; → [Compilation Error]

• Pointers :

↳ It is a variable, that holds the addr of another var.

(*) All pointers are of same size
↳ ∵ they all store address (unsigned int)
So the data they point doesn't matter

Syntax: int* p; OR int * p; OR int *p; → Same.

Note: int *p; ⇒ uninitialised ~~poin~~ pointers / wild pointers.

↓

*p; → *(garbage)
↓
[segmentation fault]

⇒ Arithmetic operation on pointers: (let p,q → pointers)

- (p+i) → p+ i x (size of the data pointed to.)

p+ constant; } Allowed.
p- constant; }

p* constant; } Not allowed.
p/ constant; }

p+q → Not possible.

(p-q) → Possible → ~~(p-q)~~ → [let p = 1000, q = 2000 and int* both]

↓

(q-p)

p*q } → X
p/q } [Not allowed]   ~~=> -200~~

This gives the no. of element in b/w the pointers. ←

$$\Rightarrow \frac{2000-1000}{\text{sizeof(int)}}$$

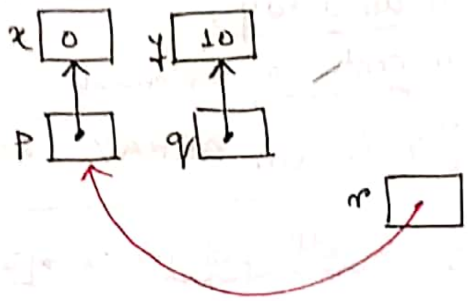
$$\Rightarrow \frac{1000}{2} = 500$$

→ Double pointer :

↳ {Pointer to another pointer}

```
main() {
int x=0; y=10;
int *p, *q, **r;
    p = &x;
    q = &y;
    r = &p;
}
```

Layout :

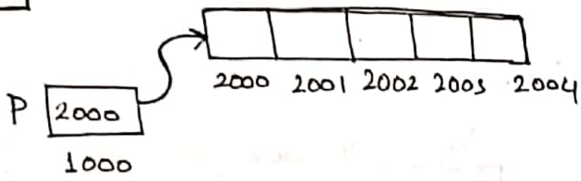


→ Pointer to an array :

<data Type> (*P) [5]

↳ P points to an array of '5' elements. ∴ P has the starting addr (addr of 1st element).

Layout :



Example / Explanation :

```
int A[] = {1,2,3,4,5};
int (*P)[5] = A;
```

⊛ sizeof (P) = 2B
(Addr size.)

⊛ Let $\boxed{A = 2000}$

A+1 = 2000 + 1×sizeof(in

∴ $\boxed{A+1 = 2002}$

But,

P+1 = 2000 + 5*2

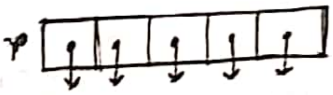
$\boxed{P+1 = 2010}$

↳ This jumps 5 elements
[Since its a ptr to an array of 5 elements]

→ Array of pointers

Layout :

`int *r[5];`



r → is an array, which contains 5 elements.

[r is an ARRAY not a pointer]

- Problems with pointer : →[IMPORTANT]

① Wild pointer / uninitialized pointer :

```
int *p;
 *p=20;
```

P [garbage]   ∴ *(garbage) Invalid.

② NULL Pointer :

```
int *p = NULL ;
*p = 20;
```

P [NULL]   *(NULL) → X. ~~Ers~~ Error.

③ Dangling pointer :

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

void main ()
 {
  int *p;
  p=f();
  :
 }
```

→[There is this problem only when x is auto / Register variable]

← After the function ends, the variable 'x' gets deleted. But, p still pointing to 'x'. └→{Invalid}

p=f(); →(Dangling pointer)

}→[x deleted] ∴ invalid Addr.

④ Lost Memory :

```
main()
{
    int *p;
    p = malloc(4);   → [this memory is lost when the
      :                 new block is assigned]
    P = malloc (6);  ←
      :                        => [LOST MEMORY]
    free (P);
}
```

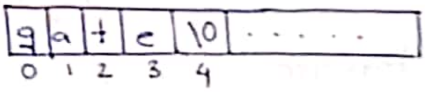
- C-Strings :

↳ Sequence/array of char , ending with '\0'.
↳ NULL termination

=> Can be stored in 2 ways →

1> [Array of char] :

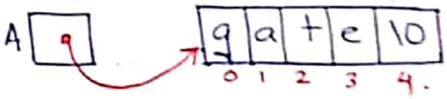
char A[] = "gate"; =>



↳ sizeof (A] => 10B.
↳ strlen (A) => 4.

2. [Char pointer]:

char* A = "gate"; =>



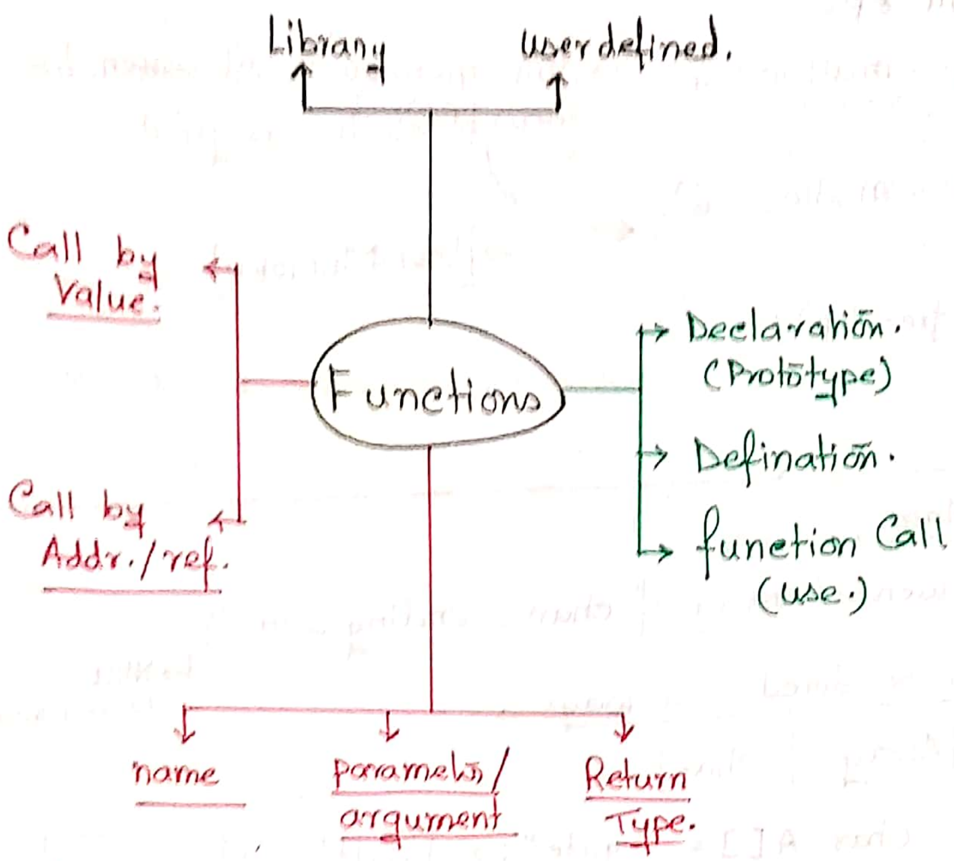
NOTE:

- %c → print only as char.
  ↳ Has to be provided with the char itself.

  %s → print until it hits an '\0'.
  ↳ Has to be provided with start Addr.

- "gate"; [This string literal returns only the address at which it is stored in memory.]

⇒ FUNCTIONS : → [Very Imp.]



- Parameter Passing Techniques :

→ Call by Value :

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

>> c = sum (x, y);
            actuals
```

- Actuals and formals have diff. mem.
- formals CANNOT Change actuals

→ Call by Ref. / Addr :

```
int sum(int* a, int* b)
{
    return (*a + *b)
}

>> c= sum (&x, &y);
```

- Actuals and formals have diff memory.
- formals CAN change actuals

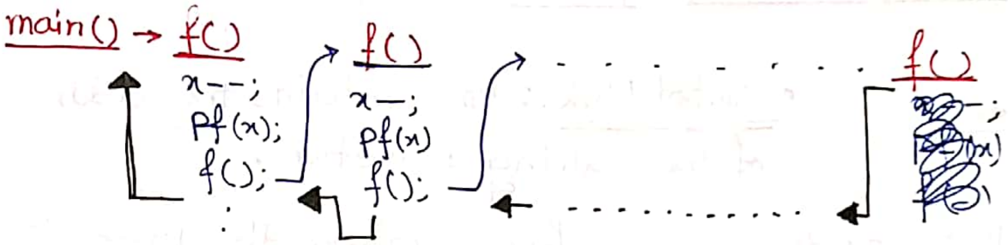
- Recursion :
  - → Defn : A func. that calls itself.
  - → Stack can support recursion.
  - → Same func. can have more activation Record.

Example :

```
int x=10;
void f()
{
    if (x>0) {
        x--;
        printf("%d", x);
        f();
    }
};
```

x = ~~10~~ ~~9~~ 8

OP : [9 8 7 6 5 4 3 2 1 0]



NOTE :

① while (True){
    :
}

→ INFINITE LOOP:
↳ the program will run indefinately. and will NOT CRASH.

② function ( )
{
    function ( );
    :
}

→ INFINITE RECURSION :
↳ Stack will overflow. and program WILL CRASH eventually.

⇒ Be careful of the environment of functions (A.R.) when solving problems.

- Activation Record.
  - → data structure of related data.
  - → Information (Active) of function is stored. (Present Status)

function_id

Local var: auto / Reg.

Resources: PC, Reg, file, I/O.

Return Value.

Access Link. → Array of pointers.

Control Link

Prev. func. in stack.

- Access Link: To access all the Global var. and Local Static var. (it's a list of pointers.)

- Control Link.: will contain, the addr of the calling function.

⇒ [it is a Link to the prev. func. where the present function was called.]

NOTE:

- In C, if there are many variables having the same name, then → the variable with nearest scope is visible.
  - ↳ ∴ Some access Links are not made, even though, there are global variables..

- function(function());
  - ↓ func. taking func. as parametn.
  - Strict Evaluation → the parameters are first evaluated before function call.

- Structures and Union :

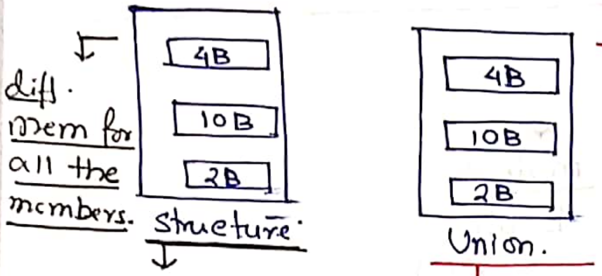
→ Structure : Collection of elements of heterogenious type.

declaration :

```
struct tagname
  {
      mem 1  decl;
      mem 2  decl;
        :
        :
  };
```

size = total size of all members.

→ Union : (similar to structure) →



→ same memory for all the members.

→ [Any one variable can exist at a time.]

size = (2+4+10)B
= 16B.

size = Max(2, 4, 10) B.
= 10B.

NOTE : ~~Stu.~~ [Structures and unions can be nested at any Level.]

NOTE :

Const. int * p ; → Value (*p) = Constant.

int Const. * p ; → Value = Constant.

int * Const p ; → Addr (p) = constant.

Const int * Const p ;

↳ Value (*p) and ~~[illegible]~~ Addr (p), both are constant.

* Constant : → Must be initialised at declaration

→ Cannot be modified throughout execution.

→ Cannot be used as L value.

- Scoping → Static and dynamic :

- Static ~~So~~ Scope :
  ↳ the scopes of variables and functions depend on structure / syntax.

  ⇒ [Access Link is known before execution.]

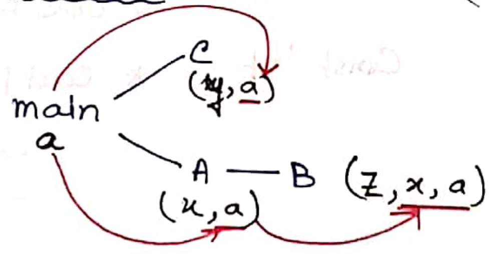
- Dynamic Scope :
  ↳ the scope of variables and functions depend on call sequence / call tree.

→ Access Link : known during execution.

Example / Explanation :

```
int a ;
→ Procedure A
      begin
          int x
          B( );
      end
→ Procedure C
      begin
        int y
          Procedure B
              begin
                  int z
              end.
      end.
```

```
main :
  begin
      C( );
      A ( );
  end.
```

Statically Scoped.	Dynamically Scoped.
Scope of A : a, x Scope of B : z, y, a Scope of C : y, a Scope of main : a .	• Call tree :  C (y, a) main a A (x, a) B (z, x, a)