C programming Languages :

SHORT NOTES

- **Basic Data Types :**

char	1 byte (8 bits) with range -128 to 127
int	16-bit OS : 2 bytes with range -32768 to 32767 32-bit OS : 4 bytes with range -2,147,483,648 to 2,147,483,647
float	4 bytes with range 10^{-38} to 10^{38} with 7 digits of precision
double	8 bytes with range 10^{-308} to 10^{308} with 15 digits of precision.
void	generic pointer, used to indicate no functions parameter etc.

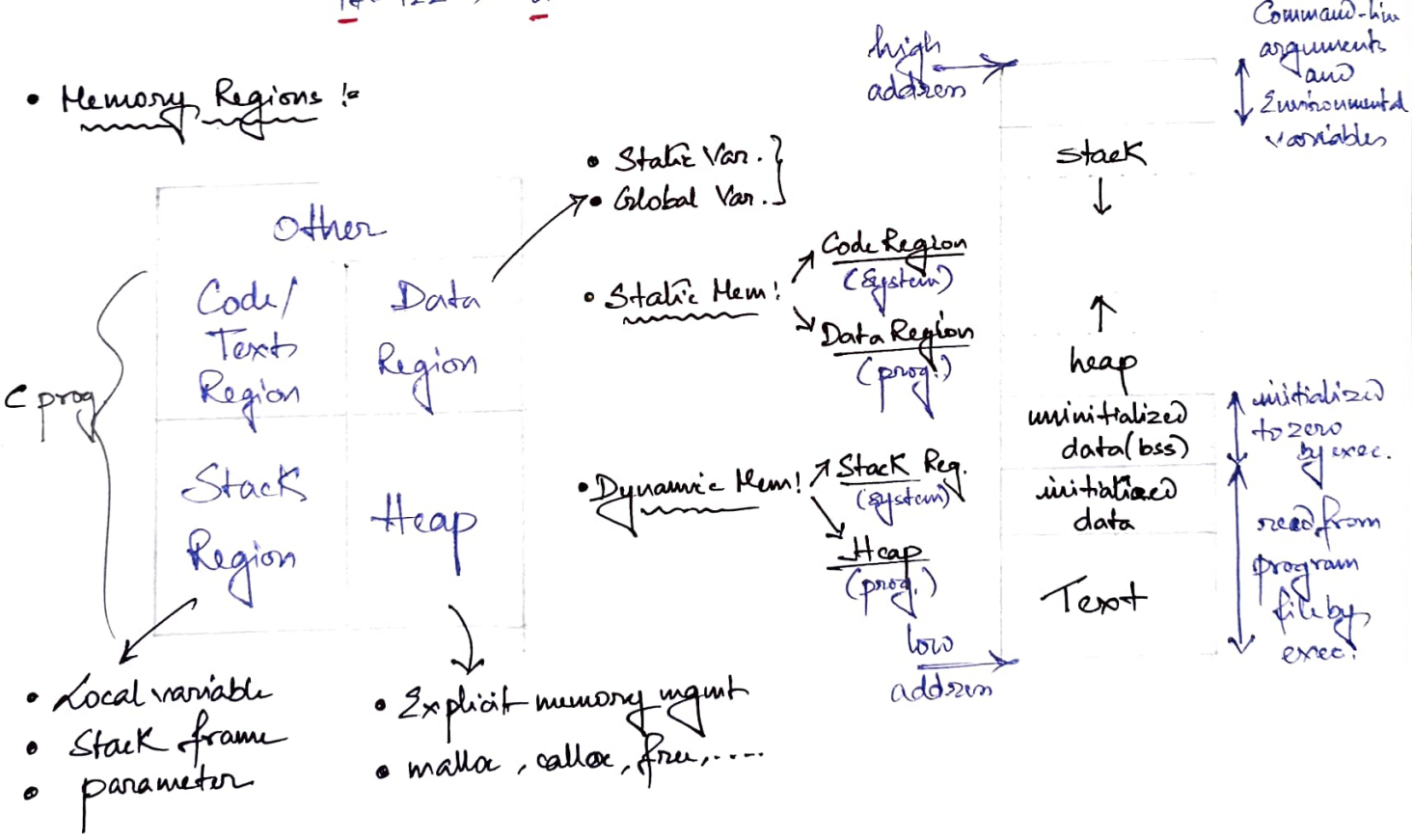
* In C prog., 0.7 is taken as double when we compare the float variable 0.7 with double 0.7 it won't be equal and also less than 0.7 due to inaccuracy while storing floating-point numbers in memory.

- **ASCII Value :**

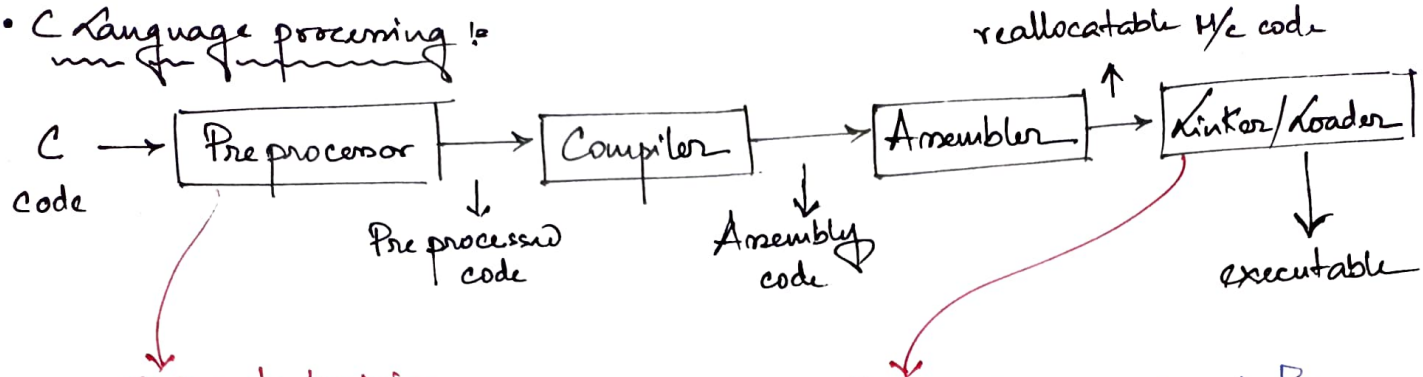
new line (10), '\n'

48 - 59 ⇒ 0 to 9
65 - 90 ⇒ A to Z
97 - 122 ⇒ a to z

- **Memory Regions :**



- C Language processing :-



Statements beginning with #, are preprocessor directive
- file inclusion (# inclusion)
- Macros (# define)

- Linker :- Resolves external files and variables
- Loader :- [Performs Reallocation.]
  → It does the allocation/deallocation.

[Always there in main mem.]

- Memory formats :- (Ex → 2)

1> Little endian :- [first LSB then MSB] → 0000 0010 0000 0000
(Reverse Order)

2> Big endian :- [First MSB, then LSB] → 0000 0000 0000 0010
(Forward Order)

C Language / Prog.
- CODE.
  - Instructions ← Group of Instructions
  - Statements ← Sequence of statements
  - Programs ← set of programs
  - Language
- DATA.
  [final representation of something, that cannot be converted to any other form.]

(*) Characters → Tokens → Syntax → Semantic ⇒ Logic (compile)
[may be correct or wrong]

Errors

- Programming error
  - Compile time
    - Compiler
      - Lexical
      - Syntax (e.g. missing parenthesis)
      - Semantics
        - type mismatch
        - L value (a+b = c)
        - use of var w/o declaration
    - Linker
  - Runtime error
    - Logical error
      - pointer prob
      - Div. by 0
      - Out of bound
      - infinite loop.
      - Stack overflow
      - Segmentation fault.
- System error.
  - Lack of resources
  - Fatal error
  - During runtime

(*) Logical error will be runtime error, but runtime error need not be logical error.

TOKENS

Token:
- Group of C char.
- Basic (smallest) unit of logical program, grouped by using longest prefix rule (maximal munch)

(*) [Space] separates tokens.

Types of Token:

① Keywords! (32)

[Reserved words]

User defined	Modifiers	Types	constant	Storageclass	control stat.
enum	short	int	const	static	if
union	long	char	volatile	register	else
struct	signed	void		extern	return
typedef	unsigned	float		auto	switch
- sizeof		double			default
					case
					break
					while
					do
					for
					continue
					goto

② Operators: [arith, logical, bitwise, Relational, ...]

(a ? b : c) — T: b, F: c

Ternary operator:

- "?:" operator the no. of ? and : are same
- every : will match with just before ques. mark (?).
- every ? will followed by : not immediately but following

③ **Identifiers:** – name given to var/func.

– **Rules** – ① no Keywords
② start with – or letter
③ Letter, Digit – allowed.

– **Should not** begin with digit.

④ **Constants/Literals.** `int x = 123;` (123 ↑ literals) `const int x = 10` [x] (x ↑ const)

⑤ **Special Symbols.**

⑥ **printf, scanf statements:** [NOTE]

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

→ scanf(): returns the no. of arguments passed.

Ex: `scanf("%d %d %d", &x, &y, &z);` returns (3) [3 arguments]

```
int n=5;
printf(n+"GATE APPLIED");
```

Soln
In the printf
n+ skip the n char

Answer: APPLIED.

NOTE: `x = y;`

(rvalue): [can be anything]

(lvalue): can't be constant
Array Name.

Associativity → AUTO rides in the right side

- **Precedence:**

NOTE: [If equal precedence ⇒ then determined via associativity] REBL

Precedence Table: PUHA S PERL TAC

Primary postfix	1>	(), [], ., →, ++(post), --(post)
Unary	2>	++(pre), --(pre), +(unary), -(unary), *, &, !, ~, sizeof(), (type)
Multi	3>	*, /, %
Additive	4>	+, -
Shift	5>	<<, >> [bitwise shift]
Relational	6>	<, <=, >, >=
Equality	7>	==, !=
Bitwise	8>	&
	9>	^ → bitwise XOR
	10>	| → bitwise OR
Logical	11>	&&
	12>	||
Ternary	13>	?:
Assignment	14>	=, +=, -=, <<=, /=, *=, >>= ... [Assignment]
Comma	15>	,

Note:
• → Right Associative [R→L]
others → Left Associative [L→R]

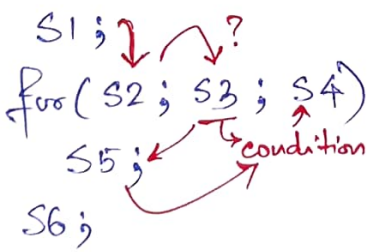
- **Storage Classifiers :=**

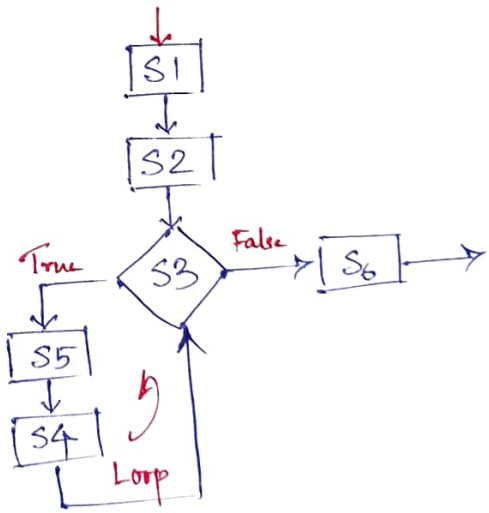
[might be given or may not be given
if register not available then stack]

Classes →	Auto	Register	Static	Extern.
Memory Region	stack	stack / Reg.	Data	Data
Lifetime	Block	Block	Program Execution (mem created just before prog. begin)	All program.
Scope	Block	Block	Depends on declaration ① local ② Global (within the prog.)	All program.
Initial value	Garbage value	Garbage value	0	0
# Declaration	1	1	≥ 1	≥ 1
# definition	1	1	1	1
Local/Global	Always Local, never be global, automatically created and deleted with the block	Always Local	Can use as both Local/Global	Always Global

- **Control Structures :=**

→ **for-loop :**





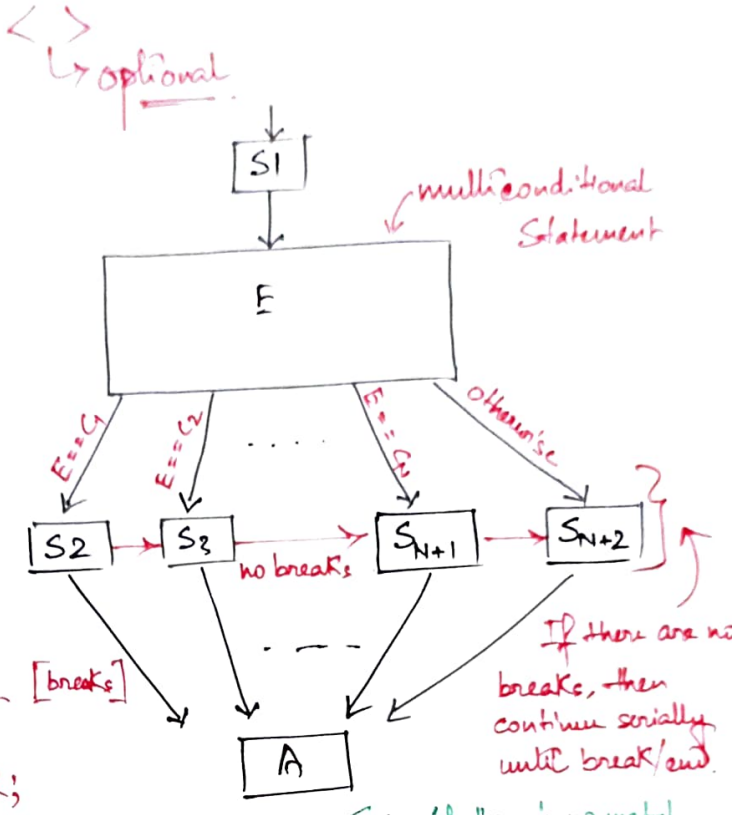
(*) Break - transfer the controls to outside of the current loop (inner).

(*) Continue says skip the remaining statement of the loop and continue the next iteration of the loop (back to the beginning of the loop)

(*) Keyword Break is not part of if-else statement.
Hence it will show compiler error : Misplaced Break.

Switch :

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



if break;
↓ come out of the block.

* If there is no match then execution starts from default.

NOTE → Very Important

- if ( ); / while ( ); → Compilation Error [condition is must]
- for (; ;); ← (no error) [logical error (runtime error) but not compile time error] } infinite loop.
  ↳ [If condition not given, it is assumed to be TRUE]
- S1, S2, S3; ⟺ S1; S2; S3;
- if (1 || ____);
  ↳ This Statement is not executed
- if (0 && ____);
- if (a = 5); → First assign, then check variable
  (a: 2, 1)
- for (; 0; [ ]);
  ↳ This is also not executed
- if (0, 1, 2, 3, 4); Only this conditions matters
  But these are executed.

Same for while(_); and for(;_;);

Return	Exit
- It is a keyword	- It is a function
- Return the ctrl to the called function	- Return the ctrl to the OS. Whole prog. terminates

(*) `j = (x, y);` → x executes then y will be assign to j

(*) `printf("%d %d %d", i, i++, ++i);` o/p - Compiler dependent.
The compiler may evaluate from left to right or from right to left.

- Don't compare float double value by ==
- Case duplicate in switch is an error

Arrays

→ Collection of elements of same type, created in contiguous manner.
→ It is a derived data structure
→ C only support STATIC Array.
→ [ROW MAJOR]

Ex-1: (1-D)
`int A[100];`

A → Name of Array. } Should be the same.
&A → Addr. of Array }

A+0] → Addr. of '0'th element
&A[0]]

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

⇒ [&A[i] = A + i × size of (element)]

Compile error case

(1) `int A[0];`
(2) `int B[-1];`
(3) `int A[3] = {1,2,3,4,5};`

'97' = not char
'0' = char
'1' = char
'9' = char

`char A[3] = {97,98,99};`

A [a | b | c]

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

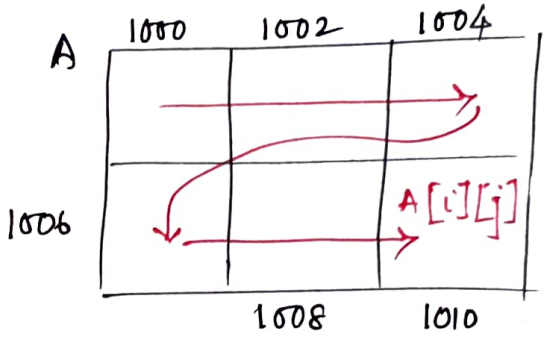
A → arr. name
&A → arr. addr.
} (same)

&A[0]+1 → &(*(A+0))+1 = A+1
&A[1][2] → add of an element
A[1][2] → name of an element.

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):

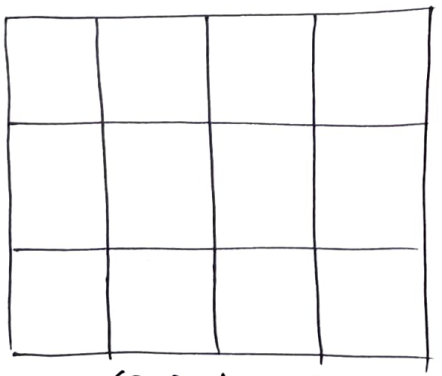
$$\&A[i][j] = \left[\begin{array}{l}\text{No of elements}\\ \text{before Row 'i'}\end{array} + \begin{array}{l}\text{No of elements}\\ \text{in Row 'i'}\\ \text{before col 'j'}\end{array}\right] \times \text{size} + A.$$

$$\&A[i][j] = [\,i \times n + j\,] \times \text{size of (int)} + A$$

3-D: int A[2][3][4];
[Table] [Row Col]

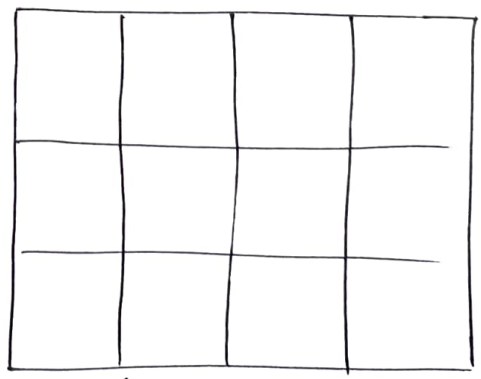
Storage: First store table by table, then Row by Row.

A[0]



(2-D Array)

A[1]



(2-D Array)

Resolving Addr. of A[i][j][K]: [int A[x][y][z];]

$$\&A[i][j][K] = \left[\begin{array}{l}\text{No of elements in prev. 'i' tables} + \text{No. of elements in prev 'j' rows.} + \text{No of elements in row 'j' before col 'K'}\end{array}\right] \times \text{size} + A$$

$$\boxed{\&A[i][j][K] = [\,i \times (y \times z) + (j \times z) + K\,] \times \text{size of (int)} + A}$$

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[5] = {10, 20, 30, 40, 50};

[will be calculated as total no of elements initialised]

$A + i = A + i * \text{size of row}$

$\&A + i = A + i * \text{size of array}$

int A[] = {10, 20, 30}; → A | 10 | 20 | 30 |

int A[2<3] = {10}; → A | 10 |

int A[1, 2] = {10, 20}; → A | 10 | 20 |

(*) For 2D Array → [Col] should be mentioned explicitly

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

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

} Possible

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

$\&A[0][0] + 3 = \&A[0][0] + 3 \times \text{size of element}$

- Pointer &

→ 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

Sytax: int* p; OR int * p; OR int *p; → Same

NOTE:
① int *p; → uninitialised pointer / wild pointer
↳ *p → *(garbage) → [Segmentation fault]

② int *p = &x ; ⟺ int *p ; p = &x ; (both are same)

- Arithmetic operations on pointer; (Let p,q → pointers)

* (p+i) → p + i × (size of the data pointed to)

p + constant ;
p - constant ; } allowed

p * constant ;
p / constant ; } Not allowed

p+q → Not possible
p-q → possible

[Let p = 1000, q = 2000 and int * both]

p * q
p / q } → Not allowed

(q - p) ⇒ (2000 - 1000) / sizeof(int) ← imp
⇒ 1000 / 2 = 500

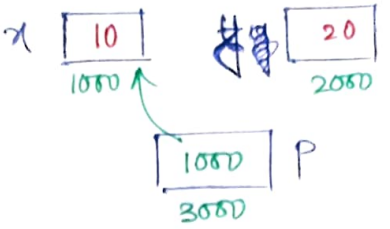
This gives the no of elements in b/w the pointers

p++ ; ⇒ ((p++)) → ① *p ② p = p+1 ; ① p = p+1 ; ② *p ; → pointer inc

++*p ; ⇒ (++(*p))
(*p)++ ; ⇒ ++(*p)
++(*p) ; ⇒ ++(*p)
} → value increment

Example

```
int x = 10;
int y = 20;
int *p = &x;
```



		x	y	p	*p

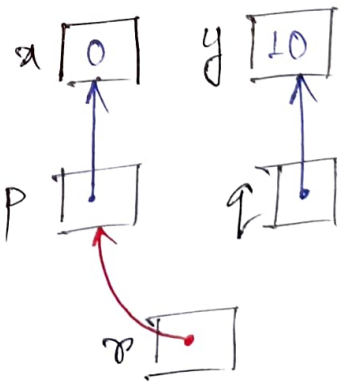
①	`y = *p--;`
	 `= (*(p--)) = *p;`
	 `p = p-1;`
		10	~~10~~ 10	998	garbage

②	`y = --*p;`	9	~~20~~ 9	1000	9
③	`y = (*p)--;`	9	10	1000	9
④	`y = --(*p);`	9	9	1000	9
⑤	`*p--;`	10	20	998	garbage
⑥	`--*p;`	9	20	1000	9
⑦	`(*p)--;`	9	20	1000	9
⑧	`--(*p);`	9	20	1000	9

Double pointer :- [pointer to another pointer]

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

layout!

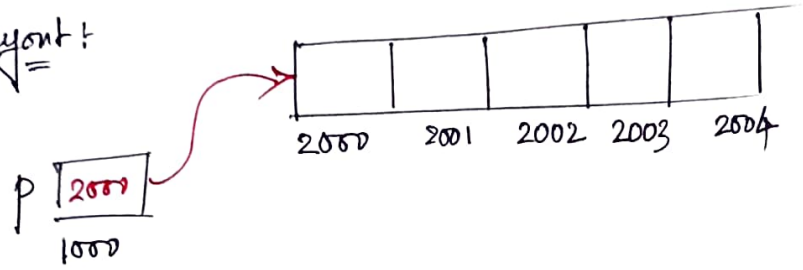


Pointer to an Array

<datatype> (*p)[5]

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

Layout:



Example/Explanation:

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

(*) Let A = 2000
∴ A+1 = A + 1 * sizeof(int)
→ A+1 = 2000 + 1 * 2 = 2002

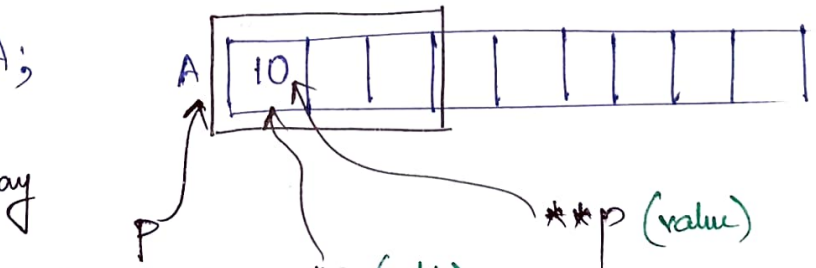
(*) sizeof(p) = 2B
(Addr. size)

But,
p+1 = 2000 + 5 × 2
p+1 = 2010

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

(2) int *p[3] = A;

↑ It is like 2-D Array

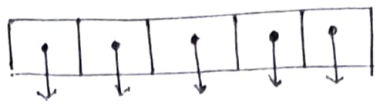


(3) 1 Dimension in Array means 1 pointer.

Array of pointers

```
int *r[5];
```

Layout:

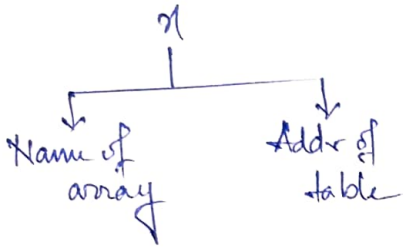


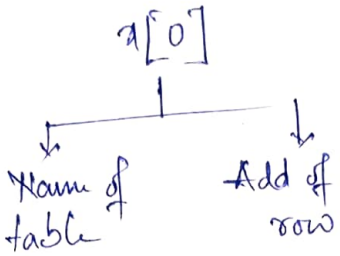
r → is an array, which contains 5 elements [r is an array not a pointer]

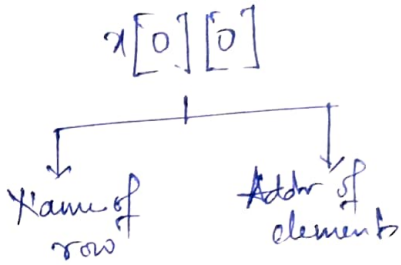
equivalent

```
(int *) r[5];
```

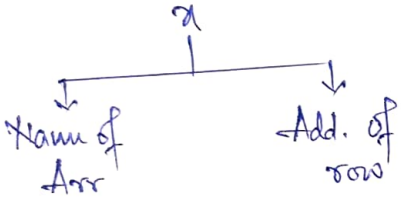
In 3D

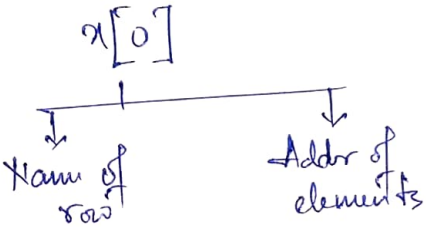






In 2D





(*) The no of ~~elements~~ columns in the array must be known to compute the mem location of any other array element.

- **Problems with Pointer:** [Important]

WARDEN

(1) **Wild pointer / uninitialized pointer:**

```
int *p;
*p = 200;
```

p [garbage] → *(garbage) Invalid

Soln: `int *p = (int *) 200;` → now 200 in address

(2) **Null pointer:** → zero addr defined in stdio.h

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

p [NULL]

*(NULL) → ✗ Error

*(0) →

Soln

```
if (*p == NULL)
    don't update;
else
    *p = 20;
```

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}

→ Dangling pointer

→ [x deleted]

∴ Invalid addr.

Solution

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

④ Lost memory :

```
main()
{
    int *p;
    p = malloc(4);
    p = malloc(6);
    free(p);
}
```

[This memory is Lost when the new block is assigned]

⇒ [Lost Memory]

⑤ Can access the void pointer only can't change void pointer

```
void *p;
p = (char *)&x;
```

} Invalid.

Sol^n

```
p = &x;
printf("%d", *((int *)p));
```

valid

C strings

↳ Sequence / array of char, ending with "\0" ↳ Null terminator

Can be stored in 2 ways →

(1) Array of char

```
char A[] = "gate";
```

→ A [g | a | t | e | \0 | ...]
0 1 2 3 4

→ sizeof(A) = 10B

→ strlen(A) = 4

(2) Char pointer :

```
char * A = "gate";
```

→ A → [g | a | t | e | \0]
0 1 2 3 4

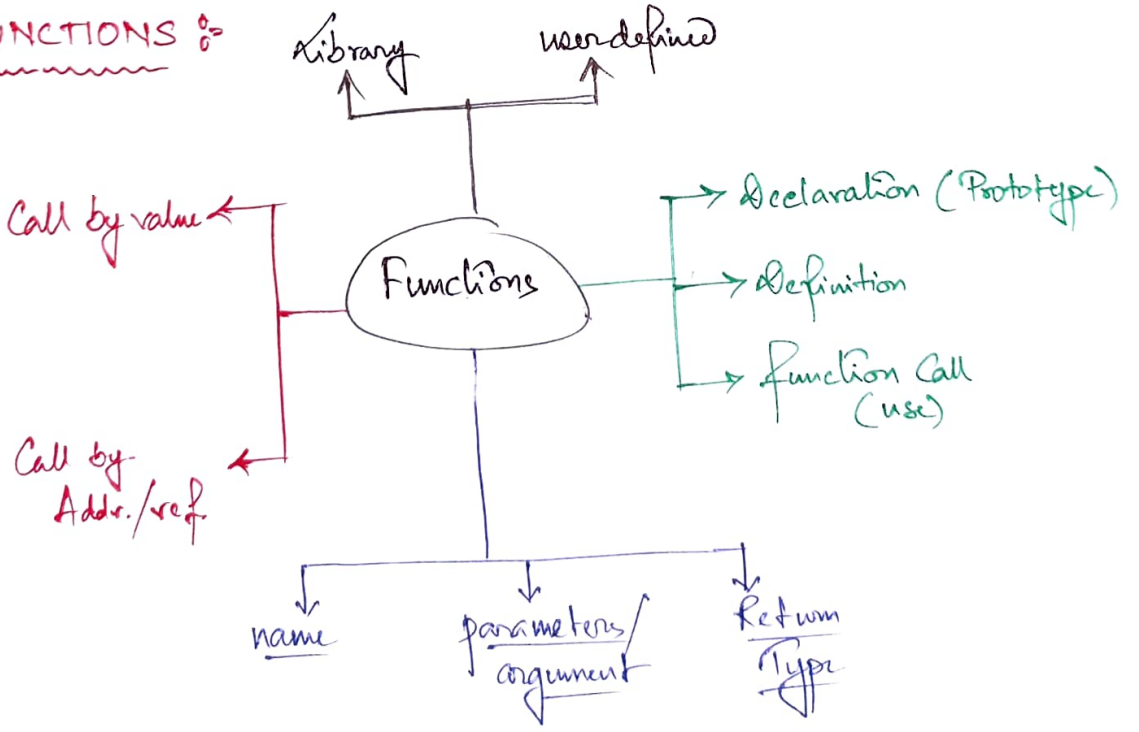
NOTE :-

%c ⇒ prints only a char
↳ Has to be provided with the char itself

%s ⇒ prints until it hits on '\0'
↳ Has to be provided with start addr.

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

FUNCTIONS :-



- Parameter Passing Techniques:

Call by value:

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

(a, b: formal)

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

(x, y: actuals)

- Actuals and formals have diff memory.
- formals CANNOT change actuals

Call by Ref./Addr.:

```
int sum (int *a, int *b)
{
    return (*a + *b);
}
>> c = sum (&x, &y);
```

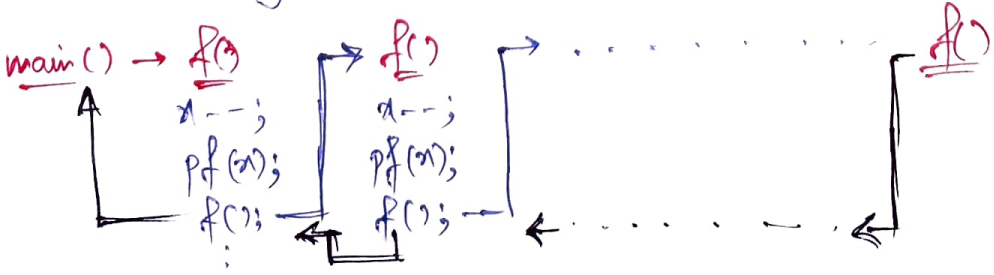
- Actuals and formals have diff memory.
- formals CAN change actuals

- Recursion: Def": A func. that calls itself
  Stack can support Recursion.
  Same function can have more activation Record.

Example:

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

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



NOTE!

① `while (true){ ... }` → INFINITE LOOP
the program will run indefinitely.
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.

Access Link : To access all the Global var. and Local 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.

L> :. some access links are not made, even though, there are global variables.

- function(function());
  - func. taking func. as parameter
  - Strict Evaluation : the parameters are first evaluated before function call.

Structures and Union:

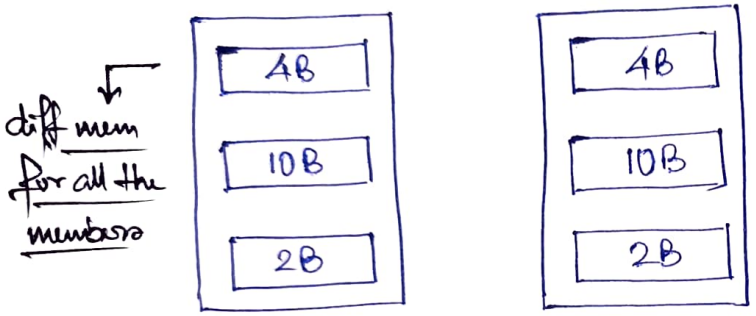
* **Structure:** Collection of elements of heterogenious type.

declaration:

```
struct tagname
{
    mem1 dec1;
    mem2 dec2;
    :
};
```

Size: Total size of all members

* **Union:** (Similar to similar)



→ Same memory for all the members.

→ [Any one variable can exist at a time]

Structure → Size = (2+4+10)B = 16B

Union → Size = Max (2,4,10)B = 10B

NOTE: [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
                            Addr (P) both are constants
```

* **Constant:** → Must be initialized at declaration
→ Cannot be modified throughout execution
→ Cannot be used as L value.

* Scoping: Static & Dynamic

* Static 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 is 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

Scope of A: a, x

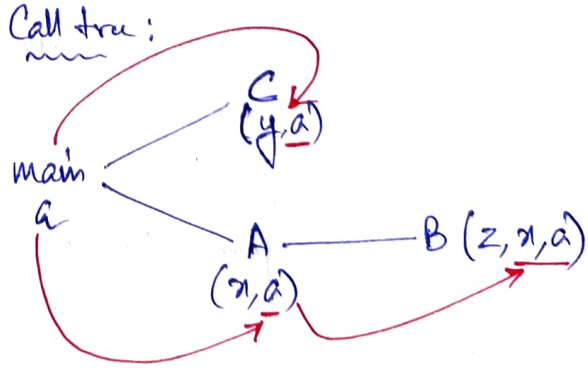
Scope of B: z, y, a

Scope of C: y, a

Scope of main: a

Dynamically scoped

• Call tree:



Declarations and Notations

1) `int *p;` p can be a pointer to an integer.
2) `int *p[10];` p is a 10 element array of pointers to integer.
3) `int (*p)[10];` p is a pointer to a 10-element int array.
4) `int *p();` p is a function that returns a pointer to an int
5) `int*p(char*a);` p is a function that takes an argument as pointer to a character and returns a pointer to an integer.
6) `int (*p)(char*a);` p is a pointer to a function that takes an argument as pointer to a character and returns an integer.
7) `int (*p(char*a))[10];` p is a function that takes an argument as pointer to a char and returns a pointer to a 10 element int array.
8) `intp(char(*a)[]);` p is a fun" that takes a arg. as pointer to a char array and returns an int.
9) `int p(char *a[]);` p is a fun" that takes a arg as array of pointers to char and returns an int.
10) `int *p(char a[]);` p is fun" arg - char array; return - int pointer
11) `int *p(char (*a)[]);` p is fun" arg - pointer to char array; return - int pointer
12) `int *p(char *a[]);` p is fun" arg - array of pointers to char; return - int pointer
13) `int (*p)(char(*a)[]);` p is a pointer to a fun; arg - pointer to a char array; return - an int
14) `int *(*p)(char(*a)[]);` p is a pointer to a fun that takes an arg as pointer to a char array. and returns a pointer to an int.
15) `int *(*p)(char*a[]);` same as previous except the argument → array of pointers to char.
16) `int (*p[10])();` p is a 10 element array of pointers to fun and each fun returns an int
17) `int*(*p[10])(char*a);` p is a 10 element array of pointer to fun and each fun. takes an int pointer to a char and returns a pointer to an integer.

* Enumeration (or enum) in C :-

→ Enumeration (enum) is a user defined datatype in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Declaration : `enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };`

- `enum` → Keyword
- `days_of_week` → enum variable
- `Sun` → State = 0
- `Mon` → State = 1
- `Sat` → State = 6
- `Sun, Mon, Tue, Wed, Thu, Fri, Sat` → Enumerators (list of constants separated by commas)

Instantiation : `enum days_of_week day;`

→ object of enum days_of_week.

Operation : `day = wed;` ⟶ day [2] ← As state of wed = 2.

eg :-

```
# include <stdio.h>
enum numbers
{
    zero, one, two, three, four = 3, five, six, seven = 0, eight
};          0    1    2     3     3     4     5     0      1
void main (){ printf ("%d", five); }
```

→ o/p : 4

*** NOTE :-**

(*) `int i = 10;`
`static int x = i;` → Compiler Error

Static variables are load time entity while auto variables are runtime entity. We can't initialize any load time variable by the run time variable.

here, i → runtime
x → load time.

(*)
```
# include <stdio.h>
# define JOIN (s1, s2) printf ("%s = %s  %s = %s", #s1, s1, #s2, s2);
int main (){
    char *str1 = "Applied";
    char *str2 = "Course";
    JOIN (str1, str2);
    return 0;
}
```

print the name of the variable

o/p → str1 = Applied str2 = Course.