7/March/2021

C language (Part-1)

- C lang. is developed to create UNIX OS.
- C lang. is developed by Dennis Ritchie in 1972
- Procedural, High level language

IDE - Integrated Development Environment

Identifiers.

- Variable
  is a name of memory location, ~~for~~ to store any value. (~~apha~~ only allows ~~of~~ alphabet, digit, undersocre) name not start with digit.

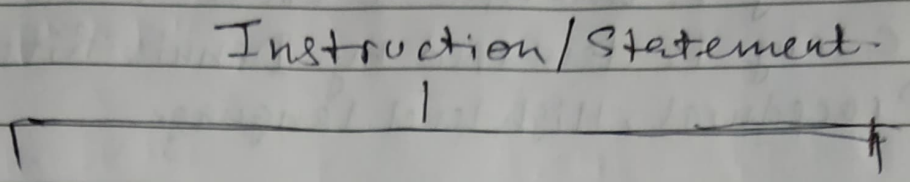
- Keyword.
  Reserved words.
  Predefined words.

Datatype

- Primitive Datatype – Built in D.T
  int, char, float, double

- Non Primitive Datatype – User Defined.
  (Using. struct, union and enum Keywords)

Data Type Declaration Instruction / Statement



- Declaration Statement
  - ↳ Data type Declaration instruction
- Action Statement
  - Input / op.
  - Arithmetic
  - Control statement

- In C language we have total 8 Datatypes.

char const.	1) Char	1 bytes.	
int const	2) int.	4 bytes (64 bit)	2 bytes (32 bits)
real const	3) float	4 bytes.	
real const	4) double	8 bytes.	8) Long double 16 bytes.
	5) Short	2 byte.	
	6) Long	4 byte.	
	~~7) long long~~	~~8 byte~~	

* Unasigned variable contains garbage value

```
int a, b=5;
    b assign 5
```

* Declaration Stmt. don't run, they ~~for~~ are for compiler, only Action Stmt run.

Block Structured Programming

Output Instruction and Escape sequence

Printf (" ");

- Printf Prints at the cursor Position.

Printf("\n"); — new line

Printf(" Hello\n world ");

Escape sequences.

\n — newline
\t — tab space
\r — carriage return (opp. of \n) // start of same lin
\b — backspace
\\ — Print Blackslash.
\" — Print double quotes.
\' — Print Single quote.

Printing value of a variable or expression

```
int a = 50;
Printf("%d", a);
Printf(" a = %d", a);
```

a = 50

format Specifier.
%d — int
%c — char
%f — float.
%lf — double
%u — unsigned

Printf("\\n"); → \n

Input Instruction.

* on entering value it moves control to the next line. / newline

```
int a;
Scanf("%d", &a);
```

```
int a, b;
Scanf("%d %d", &a, &b);
                  ↳ address of a.
```

* Enter, Space & Tab Keys are delimiters

* To change delimiter.

```
Scanf("%d : %d", &a, &b);
```

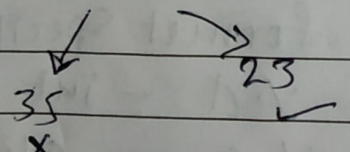
":" → Now this is act as a delimiter.

Operators in C language.

Airthmetic Instruction

3 + 4 → Airthmetic Instruction

3 + 4 * 5



BODMAS — C lang me Nahi

// → comment.

/* */ → multi line comment

On the Basis of No. of operands.

1) +c, -c → Unary operator.
2) a+b, a-c → binary operator
3) ? : → Ternary operator.

Operators. (Arrange in Precedence order)

1) Unary.
2) ~~Binary~~ Airthmetic
3) Bitwise.
4) Relational
5) Logical.
6) Conditional
7) Assignment

Unary operators.
requires only one operand to Perform its work.

+ , - , ++ , --

+5
-5 > tells the signs of No.

- Increament operator.

x++ , ++x.

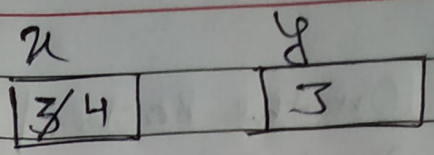
x=x+1 x=x+1.

Post Inc. Pre Inc.

- Phela a me assign fir inc.
- Phele a increment fir a me assign.

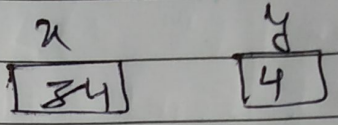
++x > x++
Pre > Post
↓
High Precedence in context with other operators. like (=) equal
Post has low Preced. than any other OP.
y=x++

```
int x=3, y;
y = x++;
Printf("%d %d", x, y);
```



→ 4 , 3

```
y = ++x;
Printf("%d %d", x, y);
```



→ 4 4

* always dry run using blocks

- Decreament Operator.

Post Decreament op.

```
x--;
```

Pre Decreament op.

```
--x;
```

- sizeof()
- & → add of value
- * → value at that add
- ! → Not
- ~ → Bitwise Not

} all are unary

```
a = 1
cout<< ++a << a++ << a;
```

→ 1 1 1

right to left execution & left to right Print

```
cout<< ++a << a*a << a++;
```

→ 3 3 1

Arithmetic operators (Left to Right)

*, /, %. → High Precedence

+, -

a * b / c

① ②

→ L to R

Numbers.

integer: 1, 5, 7 etc.

Real. 2.4, 5.4 etc

3 + 4 → 7

3 - 4 → -1

3 * 4 → 12

3 / 4 → 0 (int)

3.0 / 4 → 0.75

3 / 4.0 → 0.75

3.0 / 4.0 → 0.75

Real / int → Real.

int / Real → Real.

Real / Real → Real

5 % 2 → 1

2 | 5 | 2

4

1 ←

17 % 6 → 5

6 | 17 | 2

12

5

22 % 5 → 2

5 | 22 | 4

20

2 → ←

20 % 5 → 0

x % y → 0

y divides x completely

3 % 4 → 3 ←

Bitwise operators

works on Bit

&	|	^	~	>>	<<
↓	↓	↓	↓	↓	↓
AND	OR	XOR	NOT	Right Shift	Left Shift

```
int x;
x = 25 & 46;
```

2^4 ~~2^3 2^2 2^1 2^0~~

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
25 →	0	0	0	1	1	0	0	1
46 →	0	0	1	0	1	1	1	0

$0 \& 0 = 0$	$0 \wedge 0 = 0$
$0 \& 1 = 0$	$0 \wedge 1 = 1$ (opposite ton 1)
$1 \& 0 = 0$	$1 \wedge 0 = 1$
$1 \& 1 = 1$	$1 \wedge 1 = 0$

$0 \mid 0 = 0$	$\sim 0 = 1$
$0 \mid 1 = 1$	$\sim 1 = 0$
$1 \mid 0 = 1$	
$1 \mid 1 = 1$	

25 = 00000000 00000000 00000000 00011001

46 = 00000000 00000000 00000000 00101110

25 & 46 = 00000000 00000000 00000000 00001000 ⇒ 8

25 | 46 = 00000000 00000000 00000000 00111111 ⇒ 63

25 ^ 46 = 00000000 00000000 00000000 00110111 ⇒ 55

- Not (Bitwise) ~

```
u = ~5;
```

0 +ve
1 -ve
Most significant bit.

5 = 00000000 00000000 00000000 00000101

~5 = 11111111 11111111 11111111 11111010

-K

* Negtive no. handles by 2's complement.

K = b1) 2's
-K = b2)

2's = 1's + 1

$-K \Rightarrow \sim 5 =$ 1111 1111 1111 1111 1111 1111 1111 1010

1's ⇒ 0000 0000 0000 0000 0000 0000 0000 0101

+1

2's ⇒ " " " 0000 0110 ⇒ 6. = K

K =	b1 ⟲ 2's.
−K =	b2 ⟳

- Bitwise operator.

(>>) RightShift. n = 44 >> 2;

44 ⇒ 0000 0000 0000 0000 0000 0000 0010 1100

→ towards Right shift.

⇒ 0000 0000 0000 0000 0000 0000 0000 1011 ⇒ 11

(<<) left Shift.

n = 12 << 3;

12 = 0000 0000 0000 0000 0000 0000 0000 1100

← Shift ~~toward Right~~ left.

0000 0000 0000 0000 0000 0000 0110 0000 ⇒ 96

Relational Operator. (T/F) (1/0)

>, <, >=, <=

==, !=

1 → x = 5>4;

0 → x = 4<4;

1 → x = 3!=4;

0 → x ⇒ 5 > 4 > 3 // L to R

0 → x = 1 > 3

True

False (0)

Logical operator.

! NOT (Unary)

&& AND

~~OR~~ || OR

- ! Not

!T ⇒ F | Nonzero values are T, 1

!F ⇒ T | Zero F, 0

x = !5 > -2;

x = 0 > -2;

T

5 → T

!5 → F → 0

- && (AND)

x > 0 && y > 0

T && T → T
T && F → F
F && T → F
F && F → F

- OR (||)

x > 0 || y > 0

F || F → F
F || T → T
T || F → T
T || T → T

Assignment operator.

= x = 4;
4 = x; X Error

x = x + 1;
↓ ↘
Container content

- Compound Assignment operator.

+=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=

x += 4; ← x = x + 4

x -= 2; ← x = x - 2 ①

~~x =*~~ x *= 7; ← x = x * 7 ②

x * 7; ①

int x = 3;

x = x * 4 + 3;

op op op

3 * 4 +3

12 + 3

⇒ 15

int x = 3;

x *= 4 + 3;

op op

4+3 ⇒ 7

x *= 7; x = 3 * 7;

x = 21

Basic layout/ syntax of C Program.

```
#include <stdio.h>
 int main()
{
   .....
   .....
   .....
   return 0;
}
```

* main func. is a type of integer (int) therefore it should return an integer value
return 0 is a success status.

4 bit → 1 Nibble
8 bit → 1 byte

~~Sing~~

- Signed int →
  - [-2147483648 to +2147483647]
  - can hold -ve value
  - represented in twos complement.
- Unsigned int →
  - [0 to 4294967295]
  - can hold only non-negative value.
  - +ve values.

* always dry run using blocks.

- int → 2 byte (32 bit os) -32,768 to 32767
  4 byte (64 bit os) -21,474_ to 21474_
- short → 2 byte -32,768 to 32,767
- long → 8 byte. (64 bit os)
  4 byte (32 bit os)

#define is used to define "Preprocessor" variable.

Decision control

if statement.

```
if ( condition )
{
    ----------
    ----------
    ----------
}
```

```
if (     )
    Printf(" ");
else
    Printf(     );
```

* if if Block contain only one stmt. then it will run without any error

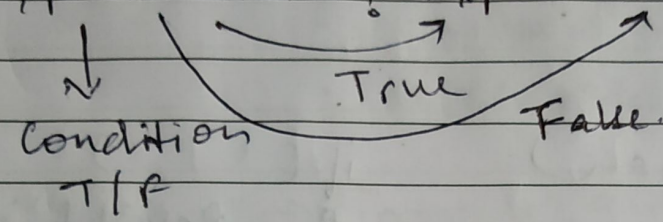
if - else statement

```
if (     )
{

}
else
{


}
```

Conditional operator.

also called Ternary operator.

```
expression1 ? exp2 : exp3 ;
```



Conditional op. ~~both~~ can behave like an exp.

```
x = y>0 ? 5:4 ;
```

Nesting

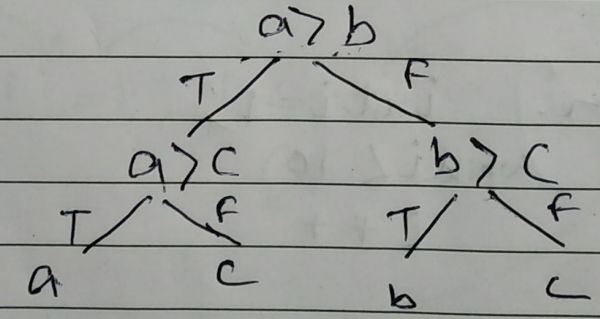
```
if ( . )
{
    if(    )
    {

    }
    else
    {
    }
}
else
{
    if (    )
    {    }
    else{ }
}
```

```
if(    )
{
    if ( )
    else
}
else
{
}
```

if else if ladder.

```
if ( )
{
}
else if ( )
{
}
else if ( )
{
}
else
```

Q find greater b/w three No.



LOOP

While Loop (Entry control loop)

```
While (condition)
{
...}
```

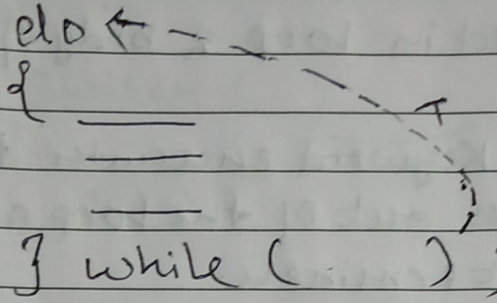
* Every non zero value is True.

eg:
```
int i=1;
while (i<=5)
{
    Printf(" Hi ");
    i++;
}
```

i
| ~~123~~ 456 |

- initialisation. int i=1;
- Condition. while(i<10)
- Updation. i++

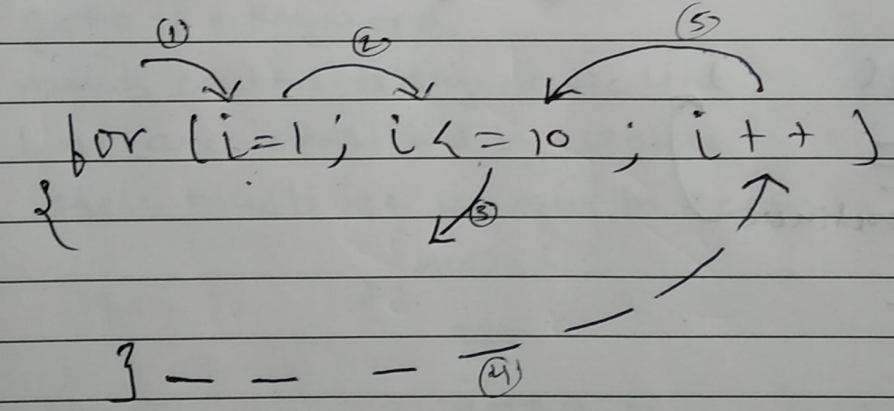
do while. [~~Ex~~ Exit control loop]



* Give atleast one output

For loop [Entry control loop]

```
int i;
for(i=1; i<=10; i++)
{

}
```



eg
```
#include<stdio.h>
int main()
{ int n=5;
  for( ; n; n--)      // 0 is treated as false condition.
    Printf("%d", n);
}
```

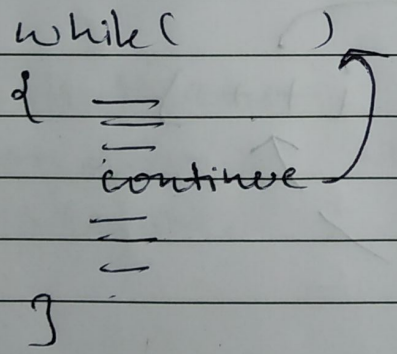
Keyword break and continue.

break Keyword used in loop & body of switch.

- break, when break Keyword encounter then control moves out of the loop and loop will terminate.

```
break;
```

- continue, only use in body of loop. used to skip any iteration. forces the next iteration of the loop to take place skipping any code in between.



Switch and goto control

Switch case control Instruction

```
switch ( )
{
   case 1 :
      ==
      break;
   case 2 :
      ==
      break;
   default :
      ==
}
```

Integer value

goto control instruction

- goto is a keyword.
- moves control to any specific line
- Lines are identified by labels
- Labels must be unique in scope.

```
main ( )
{
   ==
   goto a;
   ==
a: ==
b: ==
   goto c;
   ==
c: ==
   ==
} goto a;
```

25/March/21

C language (Part-2)

Functions

The basic Building Block of C Program

- No Keyword is a function.
- Function are of two types.
  - User defined.
  - Pre defined.
- inner blocks are not function
- reuseble code.
- outer most block
- Block of code. to solve Particular Problem

How functions work

```
main()
{   a();
    b();
    a();
}
 a()
{   Printf("A");  }
 b()
{  a();
   Printf("B");
}
```

Benefits

1) Easy to read.
2) Easy to modify.
3) Easy to debug
4) It avoids re-writing of same code over and over.
5) Better memory utilisation.

- # Main() function is called by O.S (operating sys).
The function which is called is first Load on RAM
A fun. cannot access variables of another function.

Ways to define a function

Define
a()
{ =
}

Define : main, a, b.
Call : a, b, Printf.

```
int sum(a,b)
```

↓ return type (int, void, float, char, double)

↓ function Name

→ arguments.

- function definition.

eg:
```
void sum(int a, int b)
{ int c;
  c = a+b;
  Printf("%d", c);
}
→ void Main()
{
  sum(2,3);
}
```

* Complier only knows Keywords, & Not functions

- function declaration.

```
#include <stdio.h>
Void Sum(int a, int b); // function declaration
                        or
                        prototype
Void Main()
{
   Sum(3,2);
}
 Void Sum(int a, int b)
{ int c;
   c = a+b;
   Printf("%d", c);
}
```

* We can also declare a func. inside a main() but then only main() func. is able to call that declared func.

* If any function is make before the main() func. then their is no need of declaration.

* If any function is create after the main() func. then that func. must be declare just after header file

Their are four ways to define a function

1) Takes Nothing Returns Nothing.
2) Takes Something Returns Nothing.
3) Takes Nothing Return Something.
4) Takes Something Retun Something.

1) Takes Nothing Return Nothing.

```
void Sum( );  // func. declaration.
main( )
{
    Sum( );   <- Nothing
}
void Sum( )   <- Takes Nothing.
{ int a,b,c;
  ---
  ---
  ---
  Print( );
}
```

Return Nothing ->

* If we don't write return type of any func. then by default int will considered.

* And If we don't declare any function. the compiler implicitly declare the func with int return type.

* If return type of func. at defination & declaration doesn't match then it will throw an error.

Header files contains declaration of Predefined functions such as Printf(), Scanf(), getch() etc

∴ Code of Predefined function is written in Library file.

is a Preprocessor directive and include is Preprocessor command.

PreProcessor runs before compilation and it will target only those statements whose starting with # ~~sybol~~. symbol.

→ # include <Stdio.h>
include file or code ~~in~~ written in Stdio.h file

2. Take Something, Returns Nothing

```
#include<stdio.h>  ---> void sum(int, int);
main()
{
   int x, y;
   Printf(" Enter two numbers ");
   Scanf(" %d%d ", &a, &b);
   Sum(x, y); // actual arguments
}
```

return Nothing → `void sum(int a, int b) // formal arg.` ← Takes Something

```
{  int c;
   c = a + b;
   Printf(" Sum of %d and %d is %d", a, b, c);
}
```

3. Take Nothing, Returns Something

```
#include<stdio.h>
int sum();
Void main()
{  int S;
   S = sum();  <----------,
   Printf(" Sum is %d ", S);   |
}                              |
int sum()  T.N                 |
{  int a, b, c;                |
   Printf(" Enter the values "); |
   Scanf("%d%d", &a, &b);      |
   c = a + b;                  |
   return(c);  ----------------'
}
```

* return is a Keyword.
* return c , return(c) → both are correct
* return keywords terminates the function
* we cannot return more than one value

4) Takes Something, Return Something

```
#include<stdio.h>
int sum(int a, int b);
 int main()
{  int x, y, s;
   Printf(" Enter two no. ");
   Scanf(" %d %d", &x, &y);
   S = Sum(x, y);
   Printf(" Sum of %d and %d is %d", x, y, s);
}
 int Sum(int a, int b)
{  int c
   c = a + b;
   return c;
}
```

global variable
variable written before main(), autometically initialize by 0.

Recursion

Function calling itself called recursion

```
void fun()
{
   =
   =
   fun();
   =
}
```

Trick to solve recursion Problem.

1) Sum(n) = 1+2+3....(n-1)+n
2) Recursive case (Simpler call)
   n + Sum(n-1)
3) Base case.
   if(n==1)
   return(1);

```
int Sum(int N)
{
   if(N==1)
      return 1;
   else
   return (N+Sum(N-1));
}
```

Arrays

One Dimensional array.

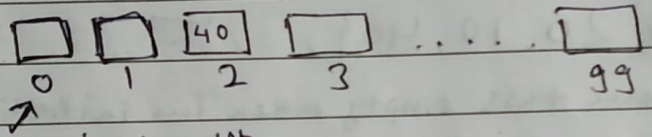
Array is a collection of homogeneous elements/var. also known as subscript variable

Declaration of a Array.

```
int a[100];
```

[ ] is a subscript operator

100 continous variables



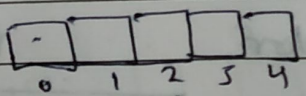
Index/subscript value.

```
a[2]=40;
```

```
for(i=0; i<=99; i++)
    scanf("%d", &a[i]);
```

Array Rules

1) `int a[5];`



2) `int a[];` error.

3) `int a[ ];` ← Natural Number

4) `int a[5];` ← Total no. of varible

`a[2]=40;` ← Index

5) `int a[5]={20, 10, 14, 3, 7};`

6) `int a[5]={20,10,4,3,7,15,25};` - ✗ error

7)
```
int a[5];
for(int i=0 ; i<=9; i++)
    scanf("%d", &a[i]);
```
} No error but wrong.

8) `int a[5]={20,24};`

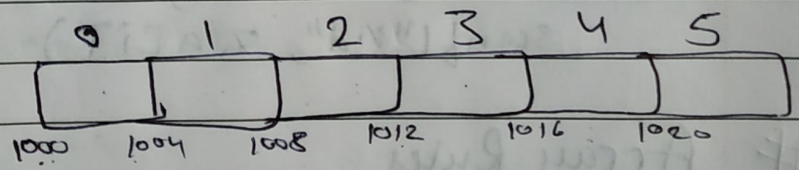
20	24	0	0	0
0	1	2	3	4

9) `int a[ ]={20,10,40};`

we can leave this empty when we initialise an array at the time of declaration.

How array variables can be accessed in constant time

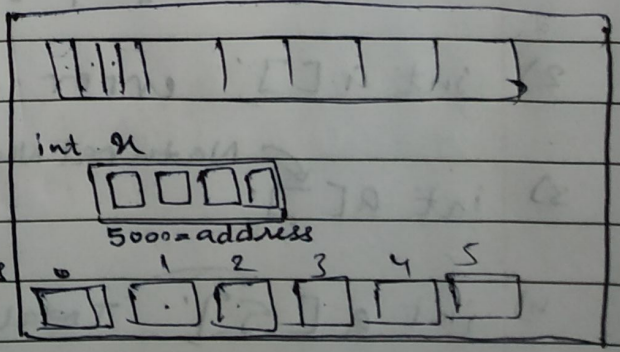
`int a[6];`



`a[4]=7;`

`[ ]` ⇒ subscript operator

RAM



$a[4] \rightarrow {}^{*}(a+4)$

$a[i] \Rightarrow {}^{*}(a+i)$

*pointer → value at that address

here $a \simeq$ add(1000) (Base add.) $a = 1000$

$a[4] = a+4$

$1000 + 4*4$

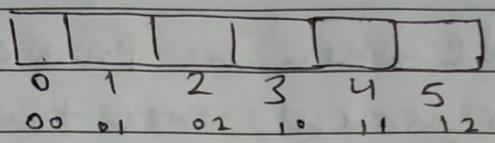
$\Rightarrow 1016$

$a[i] = {}^{*}(\text{addr} + \text{sizeof}(\text{type}) * i)$

it takes const time to access.

Two dimensional Arrays

int a[2][3];

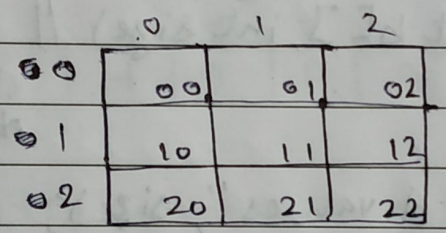


no. of variable in 2D array 2 X 3 ⇒ 6

2-D array store variable linearly

2-D array = multiple one dimensional array

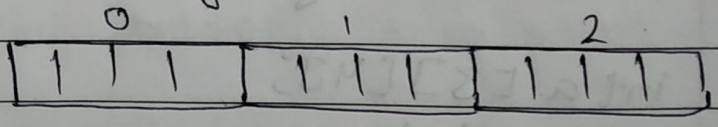
int a[2][3];
Row col



conceptual diagram

we can also declare three dimensional array
int a[4][5][10];

Multidimensional array declaration rules.



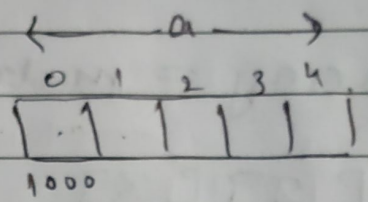
- int a[3][4];
- int a[][]; } error
- int a[3][]; }
- int a[][4];

- int a[][] = {10,20,30,40,50,60,70,80,90,100,110,120} } error
- int a[3][] = {10,20,30,40,50,60,70,80,90,100,110,120}
- int a[][4] = {10,20,30,40,50,60,70,80,90,100,110,120} NO error
- int a[][2] = {{2,13},{9,13}};

Function call by passing an array.

In C when we pass an array as a function argument, then the Base address of the array will be passed

```
main()
{
  int a[5];
  input(a, 5);
}
void input( ~~int~~ int b[], int size) // b ≃ 1000
{                                      b[0]
  int i;                               *(b+0)
  Printf(" Enter %d values", size);
  for(int i=0; i<size; i++)
        Scanf("%d", &b[i]);
}
```



- for 2-D array

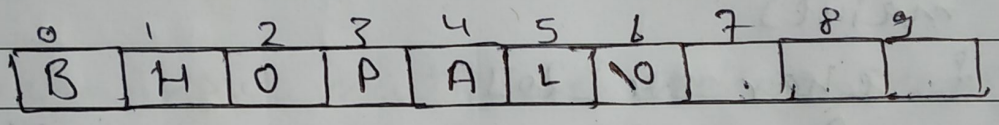
```
int a[3][4];
 input(a);

void input(int b[][4])
```

Strings

Null Terminated string

- Sequence of character terminated at null is called string. (null terminated string)
- String can be store in char array

```
main()
{
 char str[10]={'B','H','O','P','A','L'};
  ....
}
```



- automatically stores zero in unoccupied Blocks.
- char stores in their equ. ASCII codes.
  ASCII code of null character is 0.
- It is not visible on screen when Print in character format. (only space (empty) is visible).

* Printing String using for loop.

```
for(int i=0; str[i]; i++)
        Printf("%c",str[i]);
```

* Printing string using while Loop

```
int i=0;
while(str[i]!='\0')
{ Printf("%c", str[i]);
  i++;
}
```

* Another way to Print String (without loop)

```
Printf("%s", &str[0]);
        or
Printf("%s", str);
```

→ address of first element of string or array.

str ≃ &str[0]

Taking input from user

* Using scanf() fun:

```
main()
{ char str[20];
   Printf("Enter your name");
   Scanf("%s", &str[0]);
   Printf("Hello %s", str);
}
```

Amit Kumar ↵

→ Amit

• Drawback of Scanf()

1) Scanf is not capable to input multiword string.

2) By default scanf consider three delimiters
1) Spaces
2) Tab
3) Enter

* Taking input using gets( ) function

```
main()
{
  char Str[20];
  Printf(" Enter your name");
  Gets(Str);
  Printf("Hello, %s", Str);
}
```

- gets()

1) gets can input multiword strings, Scanf cannot.
2) gets can input only one string at a time.
3) gets can input only string.
4) No need of format specifier in gets().

* Print string using Puts( ) function

```
Puts(" Enter string");
```

- Puts by deafult move cursoar to new line.
- gets(), Puts() both are defined in <Stdio.h> (or declare)

String functions

1) Strlen( ) → ~~convert~~ return length of String
2) Strrev( ) → reverse the string
3) Strlwr( ) → convert string in lowercase
4) Strupr( ) → convert string in uppercase
5) Strcomp( ) → • return 0 if str are same
   • return 1 if str are not same
   • return -1 if str is in Dictionary order
6) Strcpy( ) → Strcpy(S2, S1) copy str1 in Str2
7) Strcat( ) → concatinate two strings

```
char str[20];
gets(str);
```

str ≃ &str[0] ≃ address of element of Str

```
len = strlen("Noida");
```

"Noida" ≃ address.

Function call by Passing String

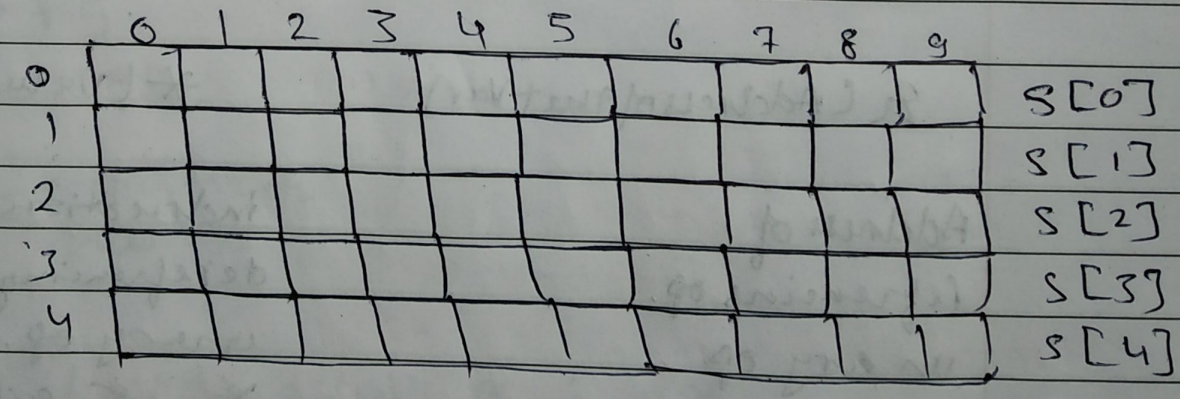
```
Main ( )
{
    l = length ( str);
}

int length (char s[ ])
{

}
```

str → &str[0] → 1000 → s

str ≃ &str[0] ≃ 1000 ≃ s

Handling Multiple strings

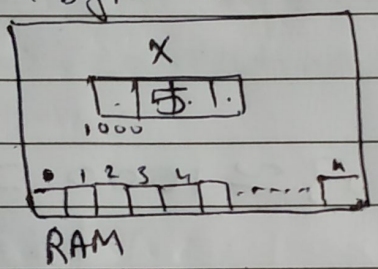
```
char s[5][10];
for(int i=0; i<=4; i++)
    gets (s[i]);
```



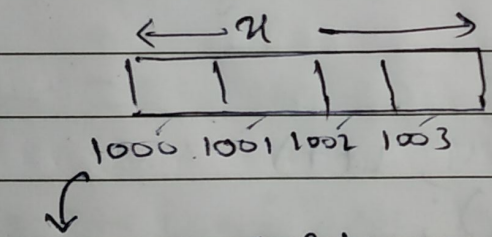
Pointers

Pointer Preface

```
int x = 5;
Printf("%d", x);  → 5
Printf("%d", &x);  → 1000
Printf("%d", *&x);  → 5
```



Whenever code is executed it acquires some memory in RAM



address ≃ Refrence.

&(Address of that Var) *(value at that add.)

Address of
refrencing op.
un ary op
&__ ← variable name.

indirection op.
derefrencing op.
unary op.
*__ ← address

$$*\&x \simeq x$$

Pointer ke pheche agr star(*) laga de toh voh vo var ban jata jise ko point karre hai.

What is a Pointer

Pointer is a variable which contains address of another variable.

```
int x = 5;
&x = 7;
```

x	j
5	1000
1000	2000

↖ error

```
&x
↓
1000 = 7
```

↖ constant

Here, x is not a variable but it is a constant.

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

// j is a pointer variable as j contains add. of x, therefore we say j is pointing to x.

x → 5
j → 1000

data ← ordinary
address

Printf("%d %d %d %d %d", x, j, &x); → 5, 1000, 1000

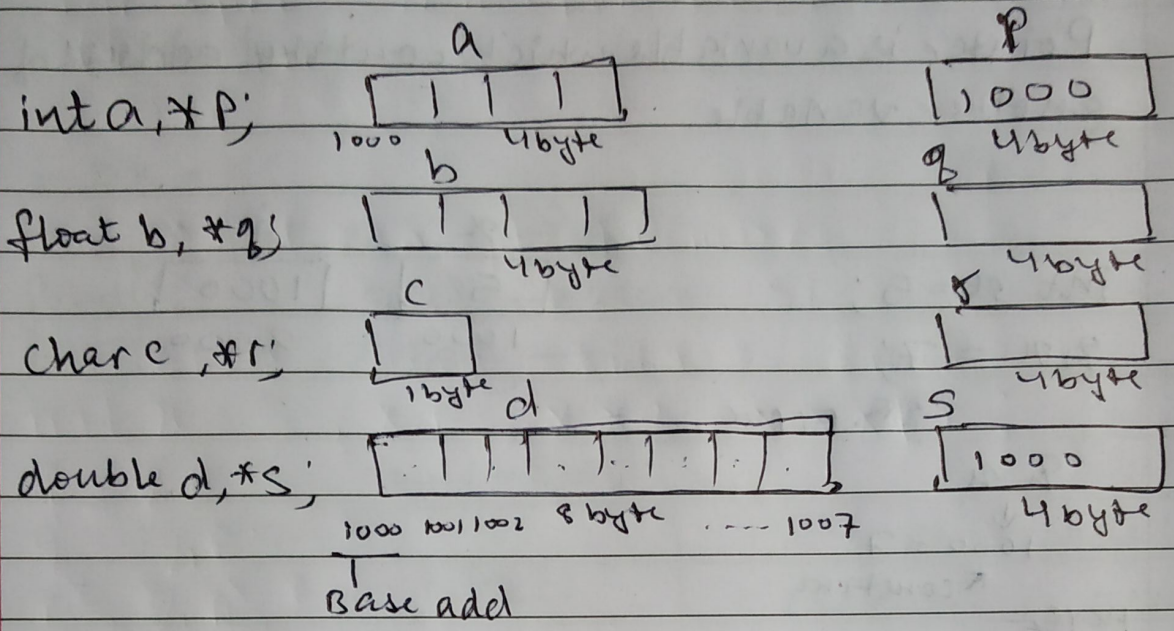
Printf("%d %d %d %d %d", *j, *&x, &j); → 5, 5, 2000

*j ≃ x

*1000 ≃ x
add

*j ≃ x *j is equal to the var, it is pointing. (x)

Concept of Base address



a → ordinary var.
*a → Pointer var.

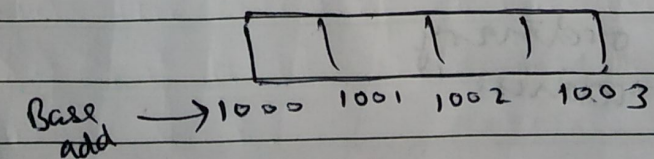
S = &d;

1) Pointer var always takes 4 bytes

2) Base add; First byte of a variable

eg:

int a



3) Pointer always contains Base address.

p = &a;
*p ≃ a
*p = 5;

4) Pointer var ~~contains~~ only contains address of oridinary var having same data type as type of Pointer var.

eg: int a, *p;

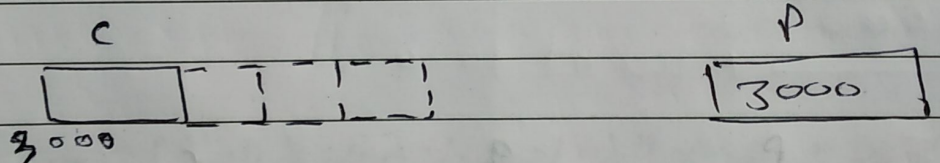
*p only contains add of a.

double d, *s

*s only contains add of d

- p = &c; → gives warning in C lang.
  → gives error in C++.

int type char type



p = &c; ~~// p points to~~

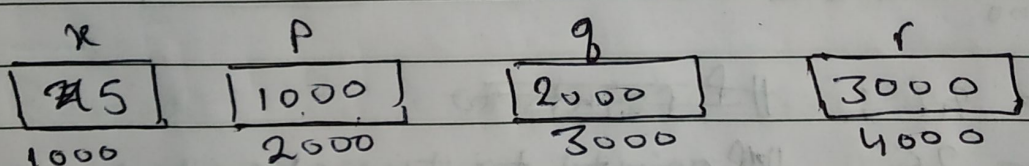
*p = 25; // *p points to the char type var. So p ~~the~~ pointer checks for other 3 uninitialised memory. or ureserved memory and this will crash our Program, illeglememory. access.

- ~~jis type ka add Pointer hold kar raha vo yahi manke chalega~~
- pointer jis type ka hai, vo ye manke chalega ki usi type ke variable ka add. vo hold kar raha hai
- Pointer same ~~same~~ type ake var. ko he point karega (Except void pointer)

```
int x;
int *ptr;  // int pointer, points to int variable
ptr = &x
```

Extended concept of Pointers

```
        0 level   1 level   2 level      3 level.
int x=5 , *p , **q , ***r ;
p = &x;
q = &p;
r = &q;
```



*p, p holds the add. of ordinary var.
but **q, q holds the add. of another Pointer var.

- Pointer khud jis * level ka hota hai, vo usse ek kam level ke variable ke he add hold. karta hai
  eg: r is of level 3, then it only hold add. of q

1) Printf("%d %d %d %d", *r, q, &P, x);

→ 2000, 2000, 2000, 5

*r ≃ q

value at 3000

2) Printf("%d %d %d %d", **q, &r, &x, P);

→ 5, 4000, 1000, 1000

*(*q) value at 2000 → 1000
value at 1000 → 5

~~**q~~ P *q ≃ P

**q ≃ *P ≃ x

3) Printf("%d %d %d %d", **r, *P, *&q, &q);

→ 1000, 5, 2000, 3000

*(*r) value at 3000 → 2000
value at 2000 → 1000 answer.

*r ≃ q

**r ≃ *q

*q ≃ 1000, P

→ **r ≃ P

4) Printf("%d %d %d %d", ***r, **qq, *qr, *qn);

→ 5, 1000, 5, 3000, 5.

r = 3000

- ***r value at 3000 → 2000
  value at 2000 → 1000
  Value at 1000 → 5

r ⇒ 3000

** (*r 3000
value at 3000

* (*2000
value at 2000

(*1000
value at 1000

→ 5.

- ** (*qq. nullify → 2000

*(*2000 → 1000

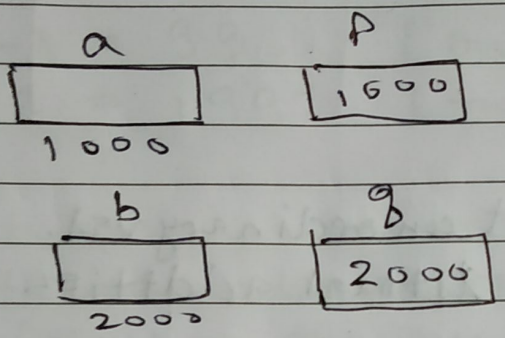
* (*2000
value at 2000 → 1000

*1000 → 5

When pointers points to another pointer called extended concept of pointer

Pointers Airithmetic

```
int a,b, *p, *q;
p= &a;
q= &b;
```



- Arithmetic operation not Performed on Pointers

1) p+q ×
   &a + &b ×

2) p * q ×
   &a * &b ×

3) p / q ×
   &a / &b ×

4) p * 5 × multiply by ordinary var ~~on~~ with ptr.
   &a * 5 ×

5) p / 5 ×
   &a / 5 ×

1. Arithmetic operation performed on Pointer
(Not literal operation)

i) p + 1
q + 3

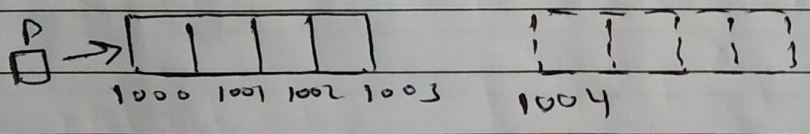
ii) p - 1
q - 2

p + 1 → p is not an ordinary val.
Here, literal addition can't performed

p + 1 ≠ 1001 // literal addition not performed here

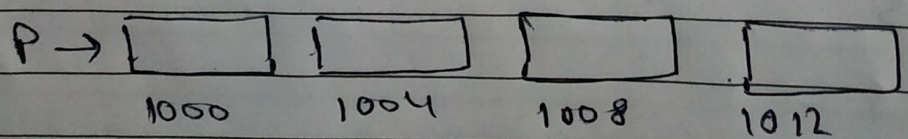
- p + 1 = 1004

p is of int type



* When we add one to the pointer, it points to the next ~~variab~~ Block of same type

- p + 3 = ~~100~~ 1012



P+3 ⇒ 1000 + 3*4 ⇒ 1012

→ (address + n * size of (type of Ptr))

double Ptr;
Ptr + 5;
3000 + 5 * 8 → 3040

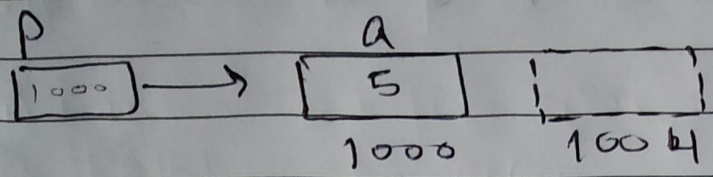
- p - 1 = 996 (one Block of int type subtract)
  1000
  q - 2 = 1992 (two Block of int type subtract)
  2000

* When we subtract one from the Pointer then it Points to the previous Block of same type.

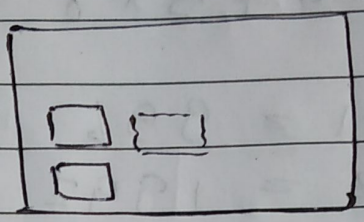
3) q - p ✓ We can subtract two pointers (address)
   q & b - q a ✓ (Not on a literal subtraction)
   subtraction only possible when both
   2000 - 1000 are of same type
   ⇒ 1000 this is not a ans.
   q - p = gives how many # variable block of same type were created between them.

2000 - 1000 = 1000 / (Size of (type of Ptr))
⇒ 1000/4 → 250 variable blocks.
(int blocks)

p - q = -250




```
int *P, a;
P = &a;
P+1
```



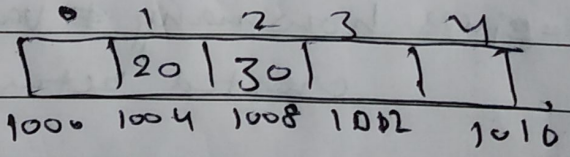
Memory assign to Program on execution is of two types 1) consumed memory area 2) Free memory area.

P+1 = 1004, which is unreserved.
illegle memory access.

~~P+1~~
~~location taken~~

P+1 is valid when we create array

```
int a[5];
```




```
int *p;
p = &a[0];
*(p+1) = 20;
*(p+2) = 30;
```

p ++ → p = p + 1

p pointer points to next block

- *p ++ ⇒ *(p++)

x = *p ++ ;

5 then → points to next Block.

*p ⇒ *1000 ⇒ 5

- ++ *p ≃ ++ a

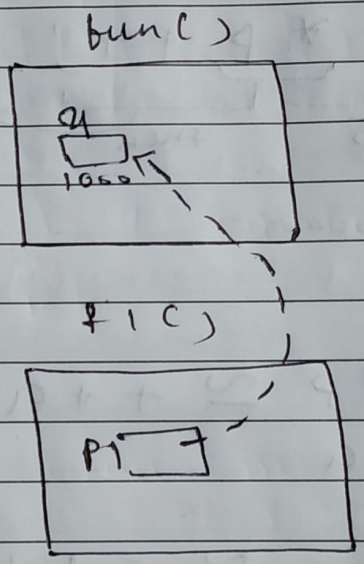
*p ≃ a

- (*p) ++ ≃ a ++

Applications of Pointer.

1) Pointers and function
2) Pointers and Array
3) Pointers and String
4) Structure pointer

```
fun()
{
    int x;
    f1();
}
f1()
{
    x
}  int *p
```



*p = x

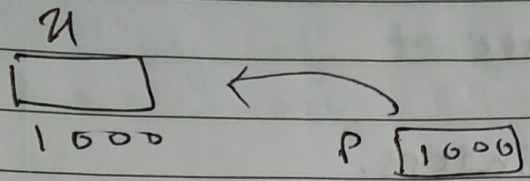
There are two type of accessing variable.

1) By its Name

By name, it is only accessible by the those func. it which it is declare

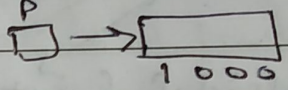
2) By its adderess.

By adrel. can access throughout the Program.



*p = 5;

2) two types of variable declaration

Static Mem. Allocation	Dynamic Mem. alloc.
int x;	p = malloc ( ) 
variable with Name	Varibble with address

Call by reference

function and Pointers.

1) call by value.

```
Void f1(int, int);
 main()
{ int a=5, b=6;
   f1(a, b); // call by val., actual arg.
}
 Void f1(int x, int y) // formal arg.
{
   x = 4;
}
```

a b
5 6

copy of a, b

x y
5 6

In call by value, we cannot modify values of actual arg.

2) Call by refrence

```
Void f1(int*, int*);
 main()
{
   int a = 5, b=6;
   f1(&a, &b);
}
 Void f1(int *p, int* q)
{   *p = 4;
}
```

a b
5 6
1000 2000

1000 2000

p q
1000 2000

Call by ref takes values takes refrence (memory address) from actual to formal arg.

Function Pointers.

```
int *P;    int pointer
float *P;  float pointer.

int (*P)(int);   // function pointer
                 P is a func pointer.
```

① One arg of type int.

② return type int

```
int (*P)(int, int);
```

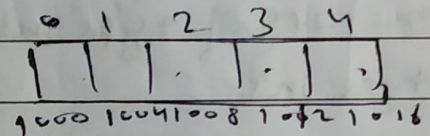
```
int square(int n)
{
   return (n*n);
}

int cube(int n)
{
   return n*n*n;
}

P = square;   // address of function
r=P(5);
printf("%d", r);    25
P = cube;
r = P(5);
printf("%d", r);    125
```

P points to function square and then points to cube

Pointers and Arrays

```
int a[5];
int *p;
p = &a[0];
```

printf()		scanf()	
a[0]	*(p+0)	&a[0]	p+0
a[1]	*(p+1)	&a[1]	p+1
a[2]	*(p+2)		
		&a[i]	p+i
a[i]	*(p+i)		



a ≃ &a[0]

```
int a[10];
input(a);
```

a ≃ &a[0] ≃ 1000

a[2] ⟺ *(a+2)

*(p+2) ⟺ p[2]

a is const, we can't assign anything to a

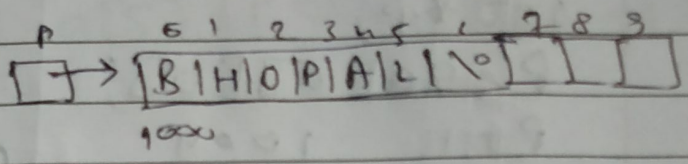
p is pointer var, assignment can be done.

2[a] → *(2+a)] also correct

2[p] *(2+p)]

Pointers and Strings

```
char str[10] = "BHOPAL";
char *P;
P = &str[0];
```



str[3] → *(str + 3)

P[3] → *(P+3)

str ≃ &str[0] ≃ 1000 ≃ P

str = const = Error when assigning

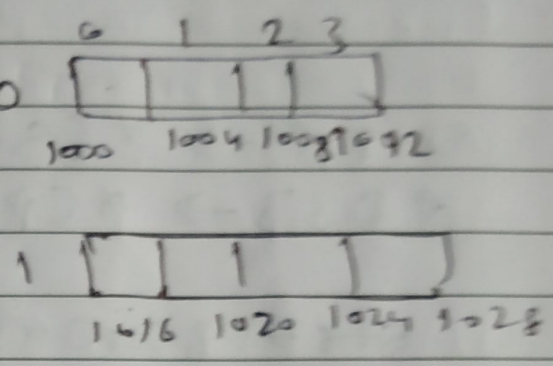
P = const = No Error.

Pointers and 2-d Array

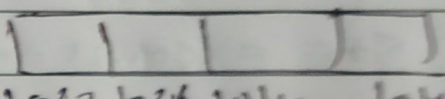
1) int a[3][4];

int *p;
p = &a[0][0];

p+1	1004
p+2	1008
p+3	1012
p+4	1016



a[1][0] *(p+4) ⇒



complicate when use in code

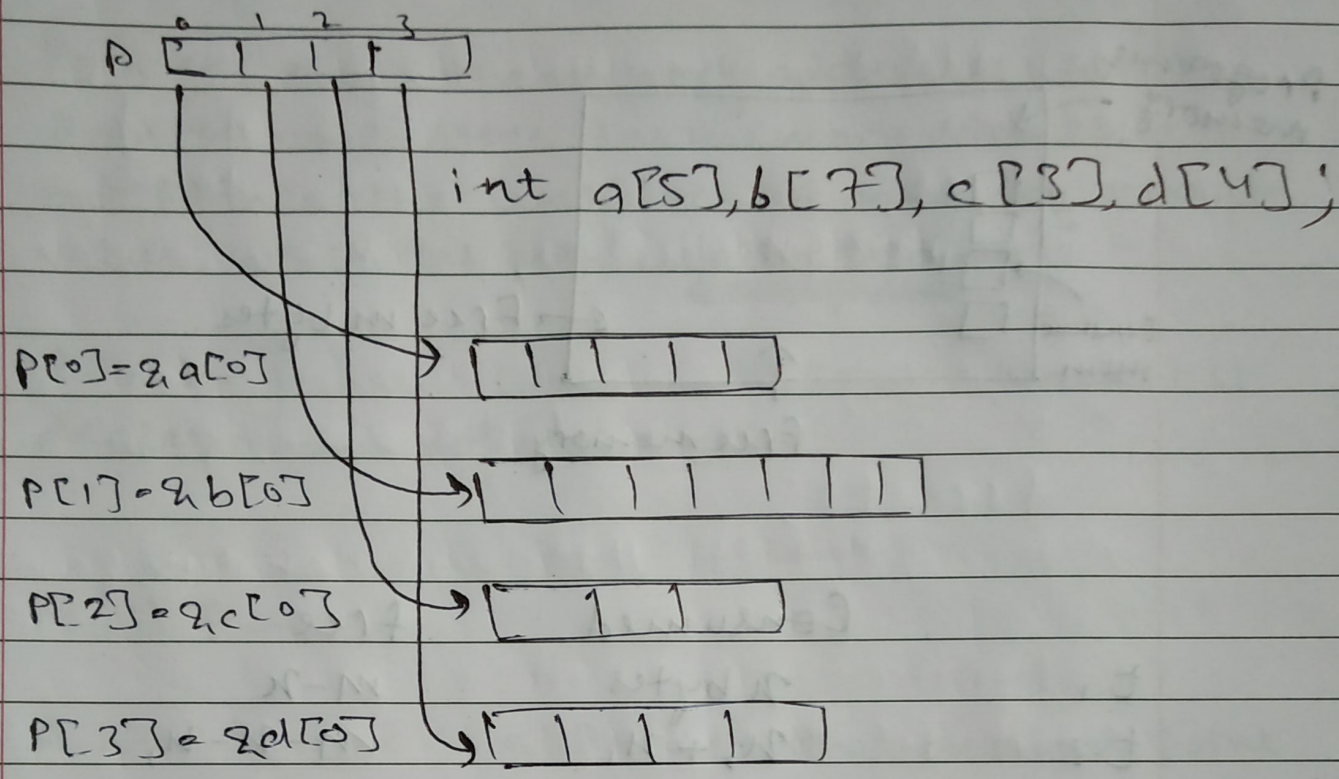
2) int (*p)[4]; // Pointer to an array
p = &a[0][0]; Arrys each of size 4.

p+1	1016
p+2	1032

a[i][j] ≈ p[i][j]

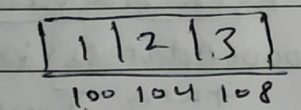
≈ *(*(p+i)+j)

3) int *P[4]; // Array of Pointers.



~~int~~

int a[ ] = {1, 2, 3}



P// &a[0] = a = Baseadd = 100

*P = a;

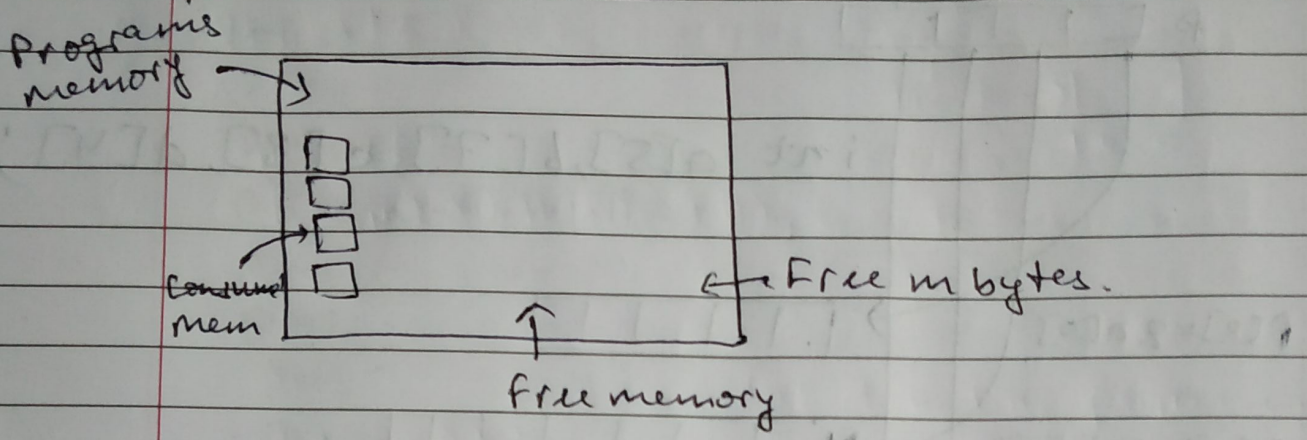
*(P+1) → 100 + 4×1 ⇒ 100+4 = 104

* P = &a; // &a → complete array address.

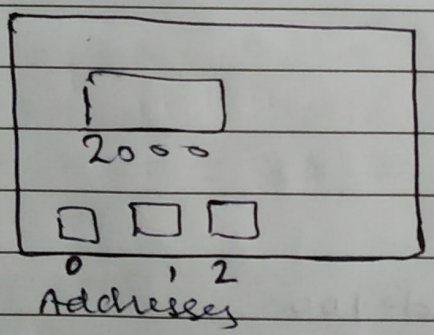
*(P+1) → ~~100~~ + Jumps to next array at add = 112

↳ usually use in 2-D array.

Wild and Null Pointers

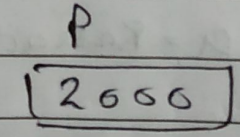


	Consumed	free
t1	n bytes	m-n
t2	$n_0 + n_1$	$m-n-n_1$
	n	m-n

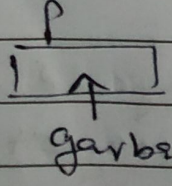


If 2000(Address) is consumed area Address then it is called valid area address.

If 2000(Address) is free area add. then Invalid address



int *p; // uninitialized Pointer, WildPointer.



Wild Pointer: (uninitialized memory)

Pointers store the memory addresses. Wild Pointer also store the memory addresses. but points the unallocated memory or data value. which has been deallocated.
~~(Dangling Pointer)~~

A Pointer behaves like a wild pointer when it is declared but not initialized.

Null Pointer.

A Null Pointer is a Pointer that does not point to any memory location
A Null pointer basically ~~store~~ the NULL value.

eg: int *p = NULL;

```
#include <stdio.h>
#define NULL 0
```

Similarity b/w Wild & Dangling Ptr

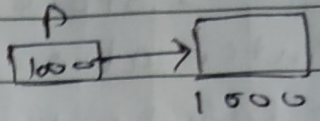
- Both points to an invalid address.

Dissimilarity b/w Wild & Dangling Ptr

- Pointer whose pointing object has been deleted called dangling Pointer.
- Pointer which has not been initialized. called wild Pointer.

Dangling Pointer.

```
Void f1()
{
  int *P;
  P = (int *) malloc(4);
  .....
  .....
  .....
  free(P);
}
```



P is dangling pointer.

P = NULL; // to avoid dangling pointer.

```
Void f1()
{
  int *P;
  =
  =
  {
    int x;        // x is only has life time
    P = &x;       //   within the block
  }

  P               // Dangling Pointer
  P = NULL; // To avoid Dangling Pointer.
}
```

Dangling Pointer:
arise during object destruction, when an object that has an incoming refrence is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of deallocated memory.

When a Pointer Pointes to a valid add. but when that memory location is deleted or deallocated, without modifying the value of pointer, and ~~pointer~~ that pointer points to invalid add.

Void Pointer

Void *p; //P is void Pointer. having size of 4 byte

Void Pointer, which is not associated with any data type. Void pointer contains address of any type of block

Problem with void pointer.

Void pointer points to Base add. of any block so, it cannot determine how many bytes are Present after the Block of Base add.

To handle this we have to type cast the void pointer

eg: {

```
Void *p;
int x=5; double y=3.14;
p=&x;
Printf("%d", *p) Error.
Printf(" x=%d", *(int*)p);
p=&y;
Printf("%lf", *p) Error
Printf(" y=%lf", *(double*)p);
}
```

Structure

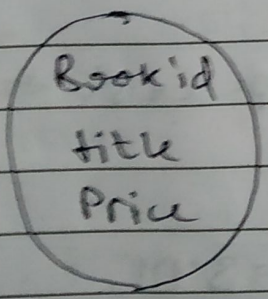
Introduction to structure

1. Structure is used to group variables
2. Structure is a collection of heterogeneous elem.
3. By using structure we can create ~~a~~ custom data type.

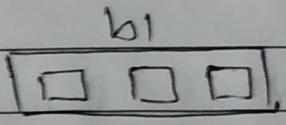
data type

1) Primitive
int
char
float
double

2) Non Primitive
Book



Book b1;



Defining a Structure

```
Struct Book          // Here Book is a data type (Custom)
{
    int bookid;      // No variable is created here, zero bytes are consumed here
    char title[20];
    float Price;     // Var inside Structer called member variables.
};
```

1) local defination of Structure → when written within a function.
2) Global defination of Structure → when written in outside (globally)

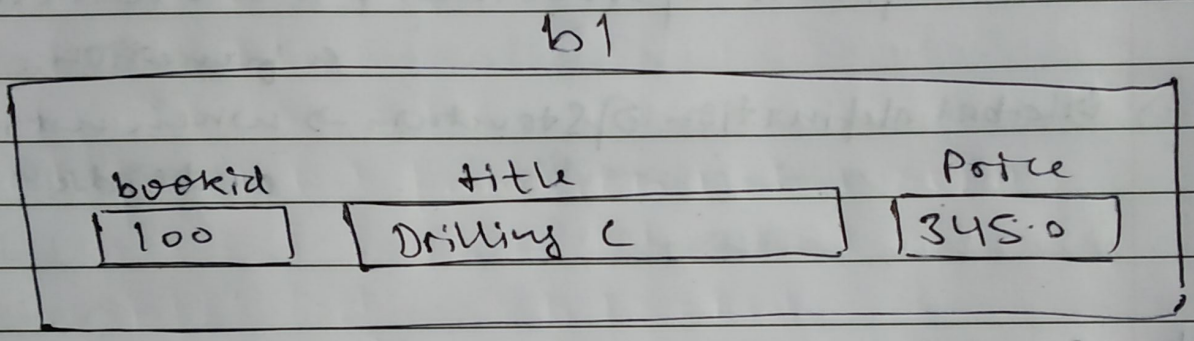
Creating Structure variable

```
Struct Book
{
    int Bookid;
    char title[20];
    bloat Price;
} b3, b4;            // global Structure variable
main()
{
    Struct Book b1, b2;   // two variable of 28 bytes each.
}
```

Data type of Var.

Initializing Structure var during declaration

```
main()
{
  Struct Book b1={100,"Drilling c",345.0};
}
```



Initializing Structure var. after declaration

```
main()
{
   Struct Book b1;
   b1.bookid =100;
x  b1.title ="Drilling c";   //Error bcoz
   b1.strcpy(b1.title,"Drilling c");  left hand side
   b1.Price =345.0;                   constant.
                                      (Base add. of Array)
}
```

Initializing structure var through user input.

```
main ( ).
{
    Struct Boob b1;
    Printf (" Entr bookid, title and Price ");
    Scanf (" %.d", &b1. bookid );
     gets ( b1. title );  → fflush(stdin)
     Scanf ( " %.f ", &b1. price );
}
```

Function returning Structure

TNRS
TSRS

```
Struct Book
   int f1( )
   {
      int x;  Struct Book b;
      =
      =
      return x;
   }
```

function call by passing ~~structure~~ structure

TSRN
TSRMS

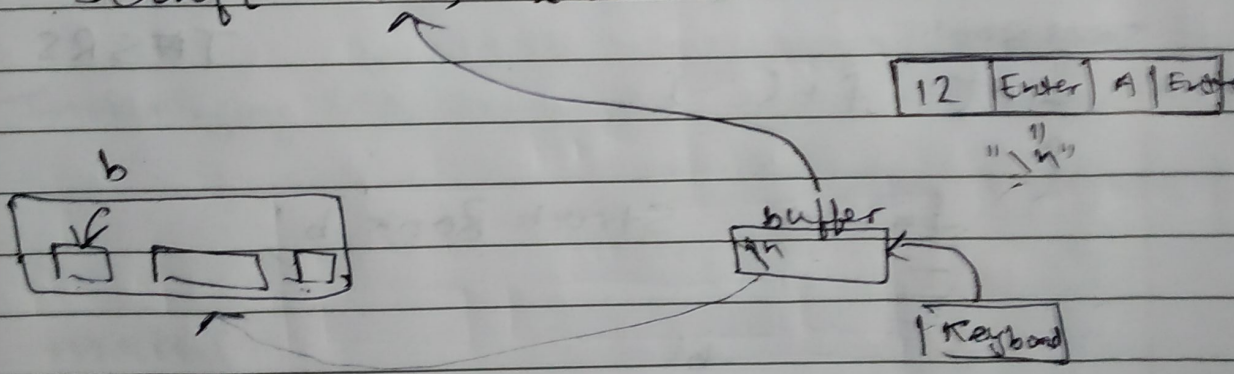
struct Book b

void f1 (~~int x~~)
{

}

- why gets( ) function not working

Scanf("%d", &b.bookid)



gets() fun. cannot invite fresh data from keyboard, and take or consider remaining data in buffer.

Scanf(), gets() both seeks to pick data from buffer, if buffer is empty. then take fresh input from keyboard.

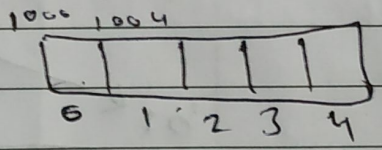
Copy Structure data.

```
main()
{
    Struct Book b1, b2;
    b1 = input();

    b2.bookid = b1.bookid;
    strcpy(b2.title, b1.title);
    b2.Price = b1.Price;
}
```

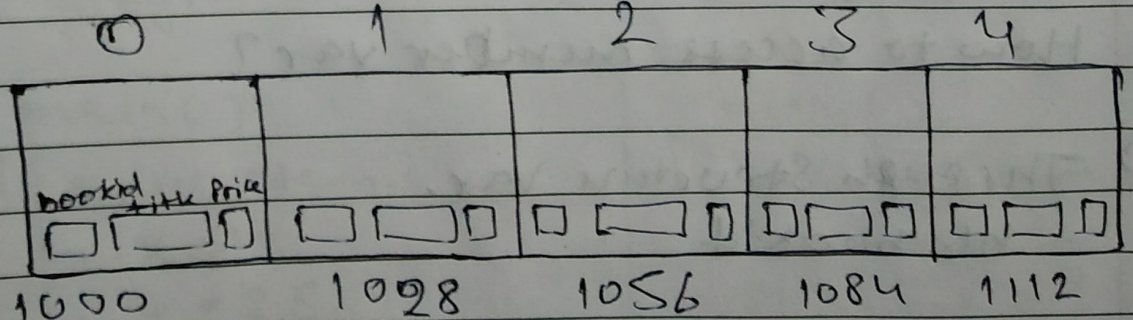
} or `b2 = b1;`

Structer Array.

```
int a[5];
float b[10];
char t[20];
```

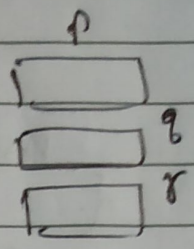


Struct Book b[5];
Datatype



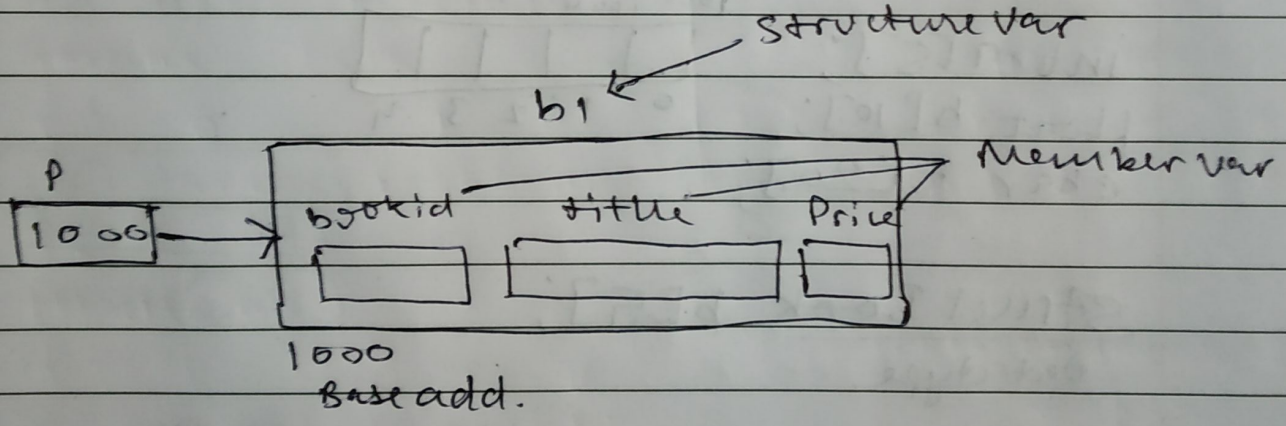
Structure Pointer

```
int *p;      // int pointer
float *q;    // float pointer
char *r;     // char pointer
```




```
Struct Book b1;   // ordinary var
Struct Book *p;   // Pointer var
p = &b1;
p -> bookid = 100;
```

$*p \simeq b1$



- How to access member var?

1) Through Structure var.
   b1. bookid.

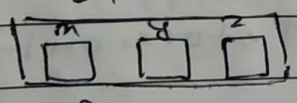
2) Through structure pointer.
   p -> bookid.

Other types and Storage classes

Union

- Union is a Keyword.
- union is user defined data type
- defination, member accessing, etc everything is same as structure but the diffrence is in their memory concept.
- In Union all members shares the same memory location.

```
struct Item
{
   int x;      4
   float y;    4
   char z;     1
};             9
```

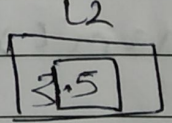


9 byte

```
main()
{ struct Item i1;
  i1.x = 4;
  i1.y = 2.5;
  i1.z = 'A';
}
```

```
Union Item
{
   int x;      → 4 byte
   float y;    → 4 byte
   char z;     → 1 byte
};
```

maximum byte from this



4 bytes.

```
main()
{ Union Item i2;
  i2.x = 5
  printf("%d", i2.y); → 5
  i2.y = 3.5
  Printf("%d", i2.x); → 3.5
```

⇔ access only one var. at a time.

Enumerator

- enum is a Keyword
- enum is a user defined datatype
- It is used to assign names to integral constants.
- multiple names can be assigned to the same integer value.
- Default starting is from 0.
- All subsequent names have integer value greater than one from the previous value by default.
- We can assign any value to any name
- values must be integers only.

```
enum State
{    0      1       2
   start , Processing , dead , end=2 };
};
   main()
{                          4 bytes stores on integer value
   enum State S1;
   S1 = start;
   ---
   ---
   ---
   S1 = Processing;
   ---
   ---
   ---
   S1 = dead;
}
```

```
enum boolean
{
    0 , 1
    True , false
};  False , True
```

typedef

- typedef is used to give data type a new name

Syntax: typedef int sides.

Here int has a new name sides.

- ```
  typedef int* PTR;
  main()
  { PTR P1;   // we can store int type add. in
    int x;        P1 variable.
    P1 = &x;
    *P1 = 5;
    printf("%d", x); → 5
  }
  ```

- ```
  typedef struct {
      int item;
      struct node *next;
  } node;
  ```
  → Data type (user defined)

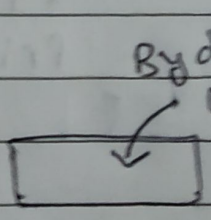
```
main()
{   node *start;   // Alternative name.
}
```

Dynamic Memory Allocation (DMA)

SMA	DMA
Static Memory alloc.	Dynamic memory all
int a; float b; char title[20]; Struct Book bl;	malloc( ); calloc( );
• for compiler • Decide on compile time: Liftime - within block • Scope - within Block • size decide on compile time	• no name • Pointer • Life → HH program ends → free( ) • Scope anywhere in Program.

malloc( )

```
float *p;
p = malloc(4)
      ↳ Size of the variable
      ↳ No data type
```



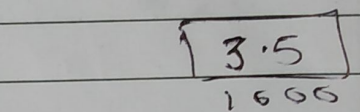
malloc( ) fun reserved 4 bytes from free memory space and return its address.

Void * malloc (unsigned int);

↑ address ↑ Size of Variable, the ingter type is not known.

this gives warning in C, error in C++, to handle this, typecast is done

```
float *p;
p = (float*) malloc(4);
```




```
*p = 3.5;
```

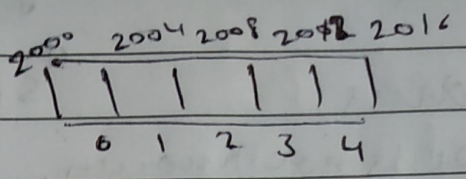
```
if( Somecondition)
    p= (float*) malloc(4);
```

- The variable created by malloc fun. by default creats garbage value
- malloc() and calloc() fun. are declared in Stdlib.h

calloc()

SMA

```
int a[5];
```

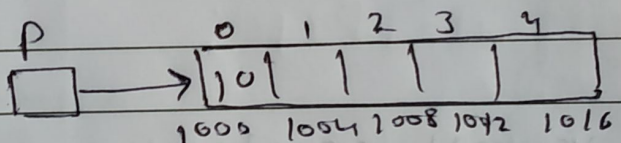


calloc() func. generates sequence of variable dynamically

```
int *p;
p = calloc(5, 4)
    calloc(n, 4)
```

Size of Array

Size of each block



Calloc() func. returns Base address of Array

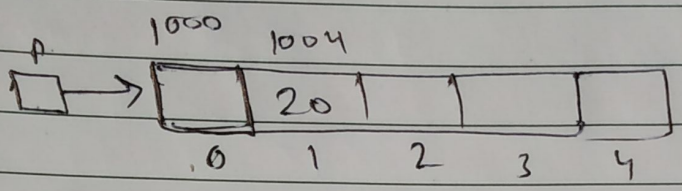
```
p = (int*) calloc(5,4);
```

```
*(p+0)=10;
p[0]=10;
```
} same

malloc()	calloc()
• generates single var.	• generates sequence of var.
• By default gen base value.	• By default Zero

- Malloc() fun also create sequence of var.

```
int *p;
p = (int*) malloc(20);
```



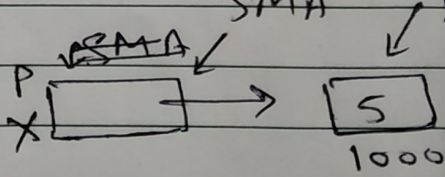
20 bytes.

5 no blocks of each of size 4 total memory consum → 20

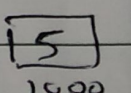
```
p[0] = 10;
*(p+1) ≃ p[1] = 20;
```

free()

free fun used to release memory of Dynamica -ly created variable.




```
Void f1()
{ int *p;
p = (int*) malloc(4);
*p = 5;
≡
free(p); → free(1000);
}
```



Memory Leak

Storage classes.

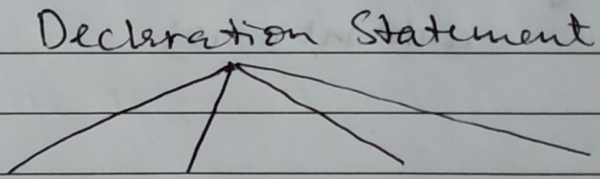
int x = 5; // Declaration Statement.

Properties we determine from D.S

- Name → x
- Size → 4 byte
- Datatype → int

Other Properties of variable.

- Default Value
- Storage
- Scope
- Life



Storage classes ⇒ Auto Registor Static External

Storage classes

- Auto
- Registor
- Static
- External.

- Automatic

Keyword : auto

DefaultVal : Garbage

Storage : Ram

Scope : limited to the block in which it is declared.

Life : Till the execution of block in which it is declared.

eg: int x = 10; // By deafault every D.S
auto int x = 10; is of type auto

- Registor

- Keyword : register
Defaut val : Garbage
Storage : register
scope : Same as auto
life : Same as auto.

When a variable accessed repetedly then registor class is used.
registor class only assign to int & char var.
for all other data type compiler implicitly changes registor to auto.

- Static

Keyword: Static
Default Val: 0 (zero)
Storage : RAM
Scope : Same as auto
Life : Till the Program ends

eg: Static int i ; By default i=0 ;

```
void f1( );
main( )
{
  f1( );   → 1
  f1( );   → 2
}
Void f1( )
{
  Static int i ;
  i++;
  Printf( " i=%d\n", i);
}
```

- External

Keyword: extern
Default value: 0 (zero)
Storage : RAM
Scope: ~~Same~~ Global
Life: Same as Static (throughout the Program)

eg:

```
int x;
main()
{
  -
}
-
```

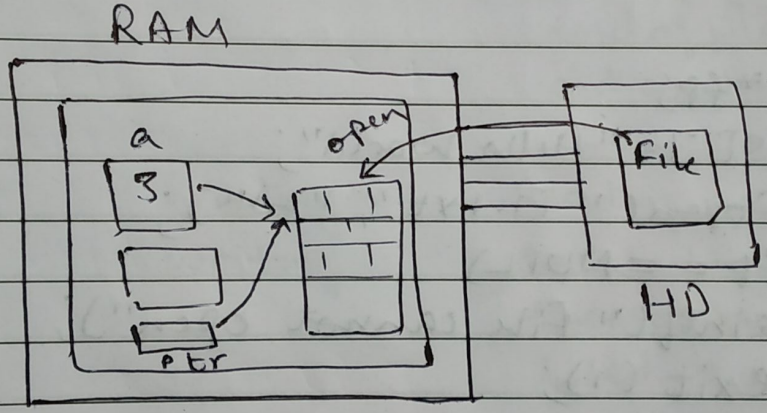
```
main()
{
  extern int x;
}
int x;
```

extern int x; → Informative stmnt. ↓ inform compiler about global var.

Data file Handling

File handling in C

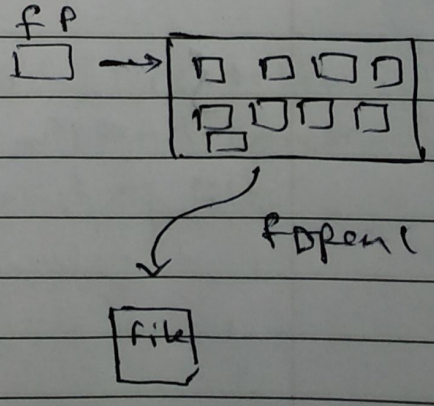
If same data is to be Processed again at some later stage, ~~again~~ again we have to enter it. to Prevent this we have to store data in secondary storage.



FILE (non Primitive data ~~Dt~~ type)

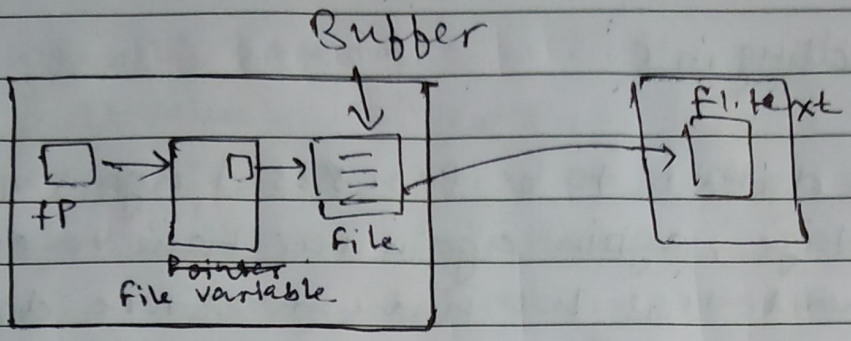
```
typedef struct{
short level;
unigned flags;
char fd;
unsigned char hold;
short bsize;
unsigned char *buffer;
unsigned char *curp;
unsigned istemp;
short token;
} FILE;
```

FILE *fp



curp ⇒ current pointer

Writing in a file



fp contain add. of ~~Pointer~~ File var.

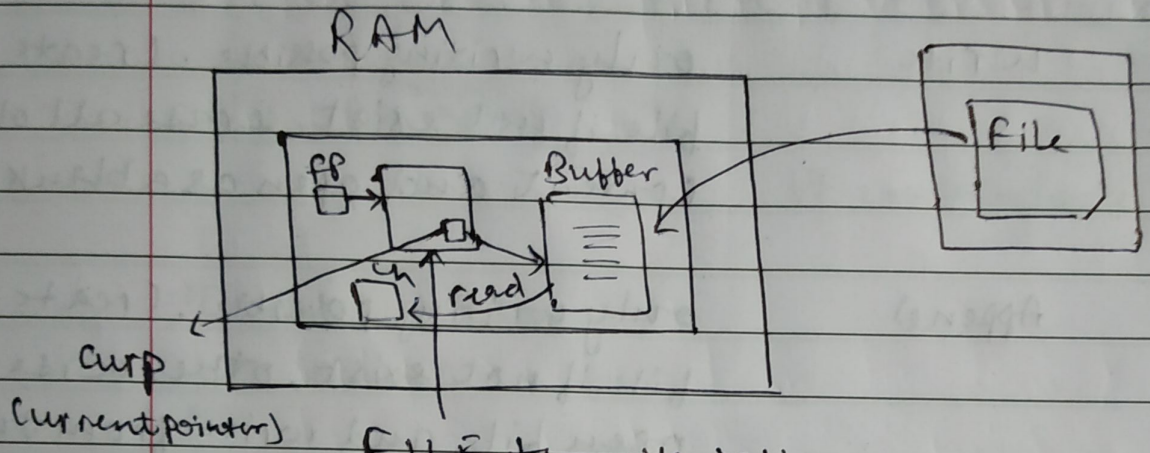
```
#include <stdio.h>
int main()
{
    int i;
    FILE *fp;
    char S[] = "Hello World";
    fp = fopen("f1.txt", "w");
    if( fp == NULL)
    {   Printf("File cannot open");
        exit(1);
    }
    for(i=0; i< Strlen(S); i++)
        fputc(S[i], fp);
    fclose(fp);  // Data saved to file.

}
```

File opening Modes.

• r	reading	only reading possible. Not create file if not exist.
• w	Write	only writing possible. Create file if not exist, erase all old content and open as a blank file.
• a	Append	only writing possible. Create file if not exist, otherwise open file and write from the end of file. (does not earse old content)
• r+	Reading + writing	Reading and writing possible create file if not exist. Over writing existing data Used to modifying content.
• w+	Reading + writing	R & W possible. create file if not exist. Erase old content.
• a+	Reading + writing	R & W possible. Create file if not exist. Append content at the end of file.

~~Writing in a file.~~

Reading from a file.




```
#include<stdio.h>
main()
{
  char ch;
  FILE *fp;
  fp= fopen("f1.text","r");
  if(fp==NULL)
  { Printf("File Not found"); exit(1);}
  ch = fgetc(fp);
  while(!feof(fp))
  { Printf("%c", ch);
    ch = fgetc(ch);
  }
    fclose(fp);
}
```

Reading from file using fgets()

fgets(str, n, fp);

fgets return a NULL value when it reads EOF

```
#include<stdio.h>
int main()
{
   char str[10];
   FILE *fp;
   fp = fopen("f1.text", "r");
   if (fp == NULL)
   { Printf("File Not found");
      exit(1);
   }
   while (fgets (str, 9, fp) != NULL) {
      Printf ("%s", str);
   }
   fclose (fp)
}
```

Writing in a file using fputs()

fputs(Str, fp);

fopen create, file type variable, bubber

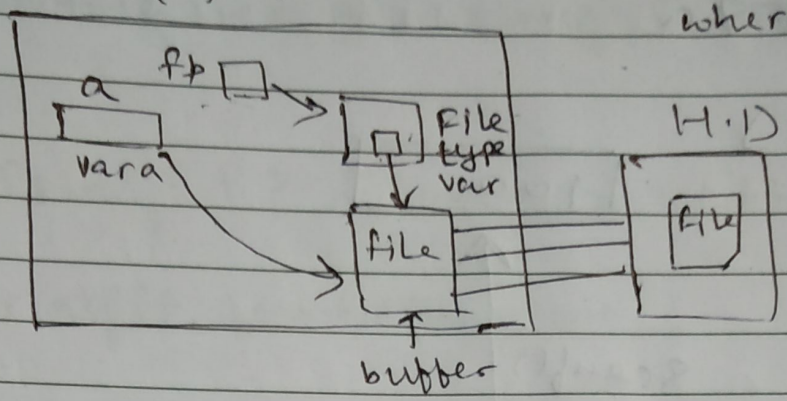
```
#include<stdio.h>
int main()
{
    char Str[20];
    FILE *fp;
    fp = open("f1.text", "w");
    Printf("Enter your name:");
    gets(Str);
    fputs(Str, fp);
    fclose(fp);
}
```

Writing in a file using fwrite()

Binary mode

- fwrite() used, to write content in Binary mode.

Text	Binary
• .txt textfile	.dat audio, vidio image image.
• w, r, a, r+, a+ wt	rb, wb, ab, rbt, wbt, abt

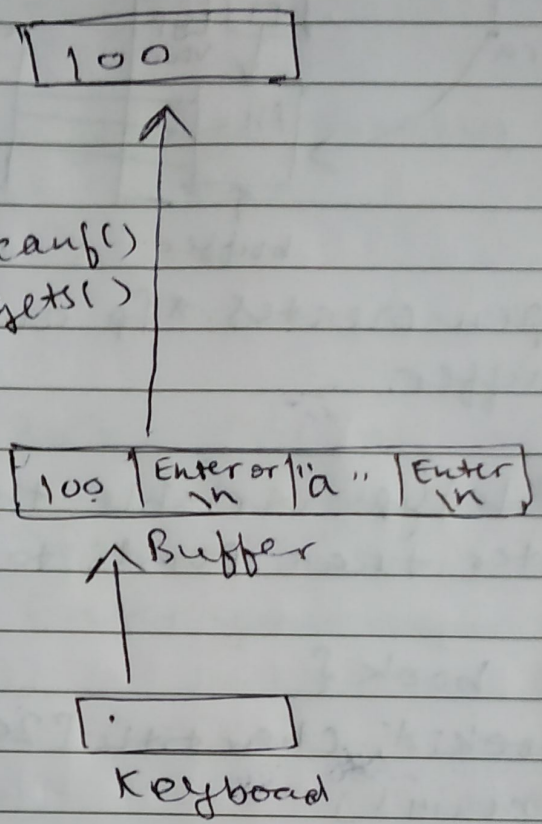


- In fopen creates *fp and FILE type variable, buffer.
- In file type variable, there is a current pointer that points to the file in buffer.

```
Struct book {
int bookid; char title[20]; float Price; };
void main()
{
    Struct book b1;
    FILE *fb;
    fp = fopen("myfile.dat", "wb");
    Printf("Enter bookid, Price, title");
    Scanf("%d", &b1.bookid);
    fflush(stdin);
    gets(b1.title);
    scanf("%f", &b1.Price);

    fwrite(&b1, sizeof(b1), 1, fp);
    fclose(fb);

}
```

~~fwrite(&a~~

- fflush( )



Reading from a file using fread( )

binary mode.

fread(&a, sizeof(var a), 1, fb)

↳ no. of records.

- fread(&b1, sizeof(b1), 1, fp)
- while(fread(&b1, sizeof(b1), 1, fp) > 0 {
  printf("%d.%s.%f\n" .....);
  }

fread( ) returns 0 → when NO record is there

1 → when record is present in file.

Writing in a file using fprintf()

fprintf(fp, "Sum of %d & %d is %d", a, b, a+b);

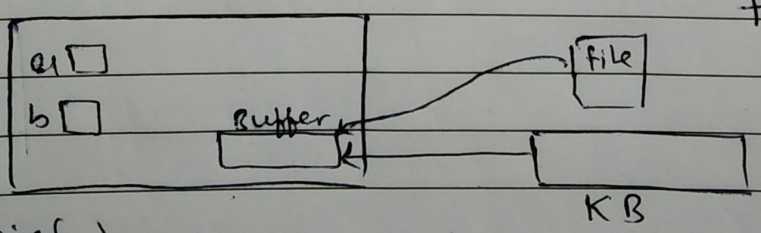
- fprintf(fp, formated string, var);

```
main()
{ FILE *fp;
  int a, b;
  fp = fopen("f1.txt", "w");
  Printf("Enter two no.");
  Scanf("%d%d", &a, &b);
 fPrintf(fp, "Sum of %d and %d is %d", a, b, a+b);
  fclose(fp);
}
```

Reading from of a file using fscanf()

Scanf("%d %d", &a, &b);

fscanf(fp, "%d%d", a, b);




```
main()
{FILE *fp;
 int a,b,c;
 fp=fopen("f1.txt", "r");
 fscanf(fp,"%d %d %d", &a, &b, &c);
 Printf("a=%d b=%d c=%d", a, b, c);
 fclose(fp);
}
```