Embedded Engineer

Embeddem System :- It is a combination of computer hardware and software designed for specific function.

* To become embedded engineer :
  1) Programming language
  2) Microcontroller architecture
  3) RTOS
  4) Electronics
  5) Networking protocols
  6) Debugging & testing
  7) Software development methodology
  8) Safety and security
  9) System integration.

* History of C :-
- C is a procedural programming language.
- It was initially developed by Dennis Ritichie between 1969 and 1973.
- It was mainly developed as a system programming language to write operating system.
- Low level access to memory.
- Simply set of keyboards.
- Clean style.

* Advantages of C programming :-
- C is a middle level language.
- Helps to understand and fundamentals of computer theories.
- Fewer libraries.
- C is very fast language.
- Embedded programming.

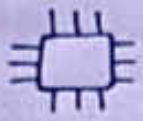
Microcontrollers and embedded programming is widely used in auto-motive, Robotics etc.

* Advantages of linux in Embedded System :-

- Easy customization.
- Used in device specific purpose - built applications.
- Power consumption is lower
- Easily portable
- low cost.

* Booting :- Booting can be done either through hardware (pressing the start button) or by giving software commands Therefore, a boot device is a device that loads the operating system. Moreover, it contains the instructions and files which start the computer. Examples are the hard drive, floopy disk drive, CD drive, etc.

* To run a program :-

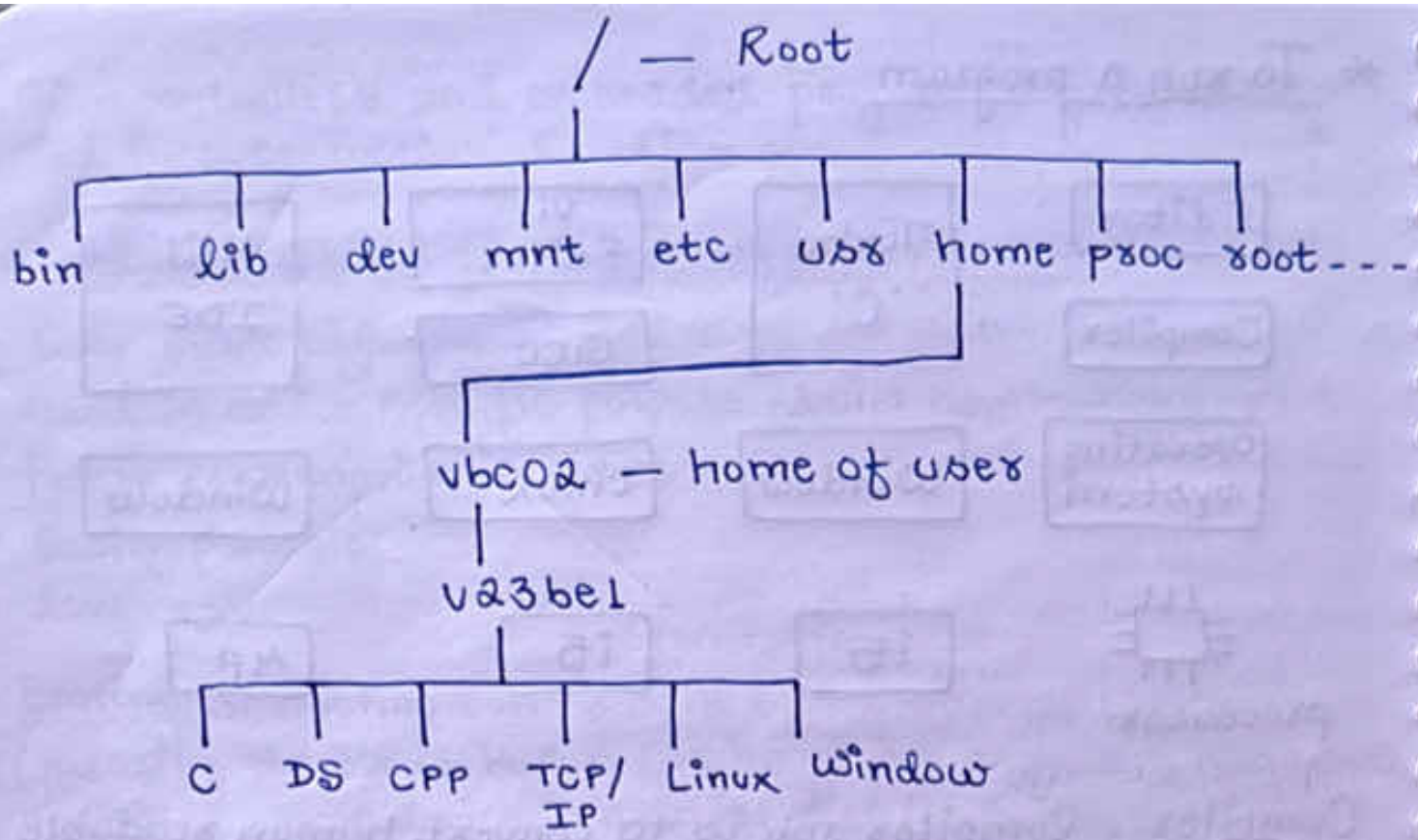
Editor	Turbo 'C'	vi Editor	Keil IDE
Compiler		GCC	
Operating system	Window	Linux	Window
 Processor	i5	i5	MP

Compiler : Compiler job is to convert human readable code into machine understand code.

- Native compiler :- working on one environment & executing result in same environment.
- Cross compiler :- working on one environment & executing result in other environment.
  or
  target board is different.

* To open command prompt → Alt + ctrl + t
  To increase font size → ctrl + shift ++++
  To decrease font size → ctrl + shift ----
  To clear the screen → ctrl + l

* $ = command prompt is ready to access the command.
  ls = It displays list of files in present working directory.
  mkdir = used to create a directory.
  cd = change directory.
  pwd = present working directory.



bin :- This directory contains all the commands which are in executable file formats.
Ex :- cd, ls, pwd, mkdir etc.

lib :- This directory contains library files also called as supporting files.
Ex :- printf, scanf etc.

dev :- This directory contains device drivers files.

mnt :- This directory is used to mount external hardware.

etc :- This directory contains network configuration files.

usr :- This directory contains users related information.

home :- This directory contains local users.

proc :- This directory contains information about processes.

* Operating System is used for multitasking.

* In linux command prompt we can supply the path in two different ways i.e,

1) Absolute path :- starts with (/)
2) Reference path / Relative path :- starts with (. or ..)

Here (.) = present working directory .
(..) = parent directory .

Absolute method ↓

mkdir / home / ubco2 / v23be1 / ds / one] path creating method
mkdir../ds / one

↑ Reference method

* Files :- Collections of data presented in hardware .

Compiler are extension dependent .

Ex :- .c, .java, .py etc

vi editor works in two modes

* By default vi editor is in command mode (cmd) .

wq → save and quit

escape + shift : wq
escape + shift + zz] save & quit

Compilation Keys :- 1) cc Hello . c
2) cc Hello . c — O Pi

New executable file name

* Compilers are extension dependent whereas Operating System may be or may not be extension dependent.

Ex :- windows

Ex :- Linux

* Machine understandable Code

RAW

without OS it can run on hardware directly

ex :- hex files

generated by cross compilers.

Executable code

on the top of OS it will execute

ex :- icons, browsers, a.out, .exe

generated by native compilers.

rmdir :- remove the directory (if directory is empty).

rm -r :- remove the directory (if directory is full).

* escape dd :- It will delete the line at where the cursor is present.

esc 3dd :- delete the 3 line.

u :- undo.

YY :- It is used for copying the line.

P :- paste.

esc yw :- copy the particular word.

2yw :- for copying two continuous word.

dw :- deleting particular word.

:q! :- exit.

w :- save.

:3 :- To move the cursor to 3rd line.

/Hello :- To move the cursor to a particular word.

esc + shift :%S/search/replace/g

↓

To replace the particular text in a code.

```
# include <stdio.h>
void main ()
{
    printf ("Hello World using vi .... \n");
}
```

Basic structure of C-Program :-

```
#include <stdio.h>      — header file
// global variables
// user defined function prototypes (declaration)
void main ()
{
    // local variables
    // logic
}
// user defined function definition .
```

cat Hello.c :- To see the content (code) in the screen presented inside Hello.c file.

* How to enable line numbers :-

step 1 - cd ↵

" 2 - vi .vimrc

```
set nu
set si
set ai
```

* The building blocks of our C-program is function

Functions are of two types :

1) Predefined function or compiler supported function.

Ex :- printf , scanf .

2) Users defined function .

Whenever their is a term called functions, programmers need to think 3 points

1) Function declaration
2) Function call
3) Function definition .

- All the predefined function declaration are present in header files.
- All the predefined function definition are present in library (lib).
- Header files contains the predefined function declaration.
- As per programming standard before calling any function it should be declared above.

1) Why main functions ?

cc Hello.c

↓ ↑

waarning :- executable file is created

Errors :- executable file is not created

↓ if still want to create

cc -nostartfiles Hello.c

exit (0); :- This function is presented in

#include <stdlib.h>

* Program should starts from a particular function that why main is used.

2) Can we write a C-program without any function ?

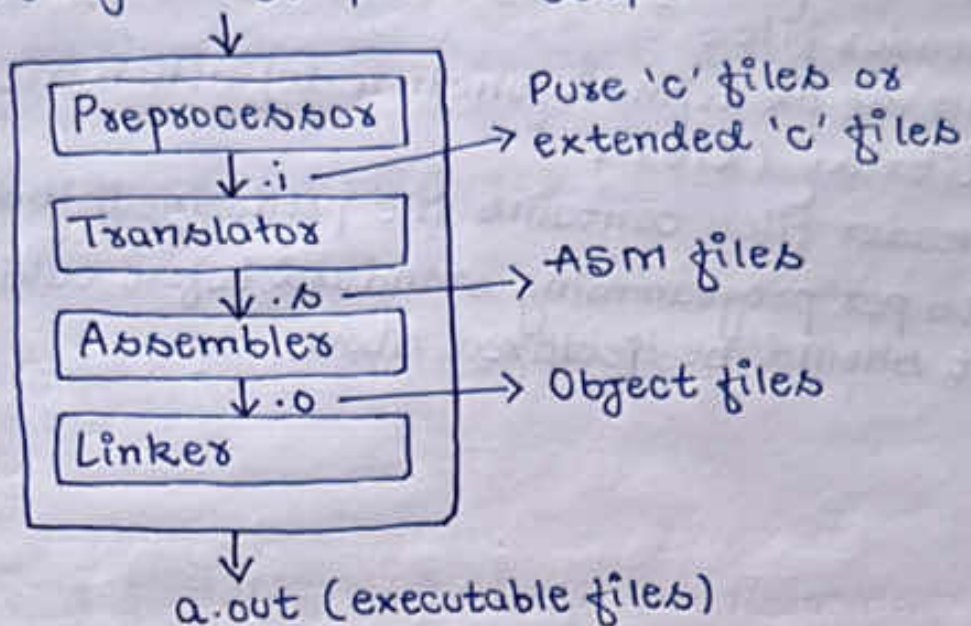
⇒ No, minimum one function is required.

3) Can we write a C-program without main function ?

⇒ Yes, we need to compile in different ways

i.e, cc -nostartfiles Hello.c .

* Compilation stages / Compilation steps :-



1) Preprocessor :- input : .c
output : .i

a) includes header files
b) Remove comments
c) Replace macros
d) Conditional compilation

cc -E p1.c -o p1.i

2) Translator :- input : .i
output : .s

a) check syntax errors
b) converts .c into ASM

cc -S p1.i -o p1.s

3) Assembler :- input : .s
output : .o

a) converts the ASM codes into op-codes

cc -c p1.s -o p1.o

4) Linker :- input : .o
output : a.out

a) It will link with libraries
b) Adds operating system informations
c) Search functions definitions

cc p1.o

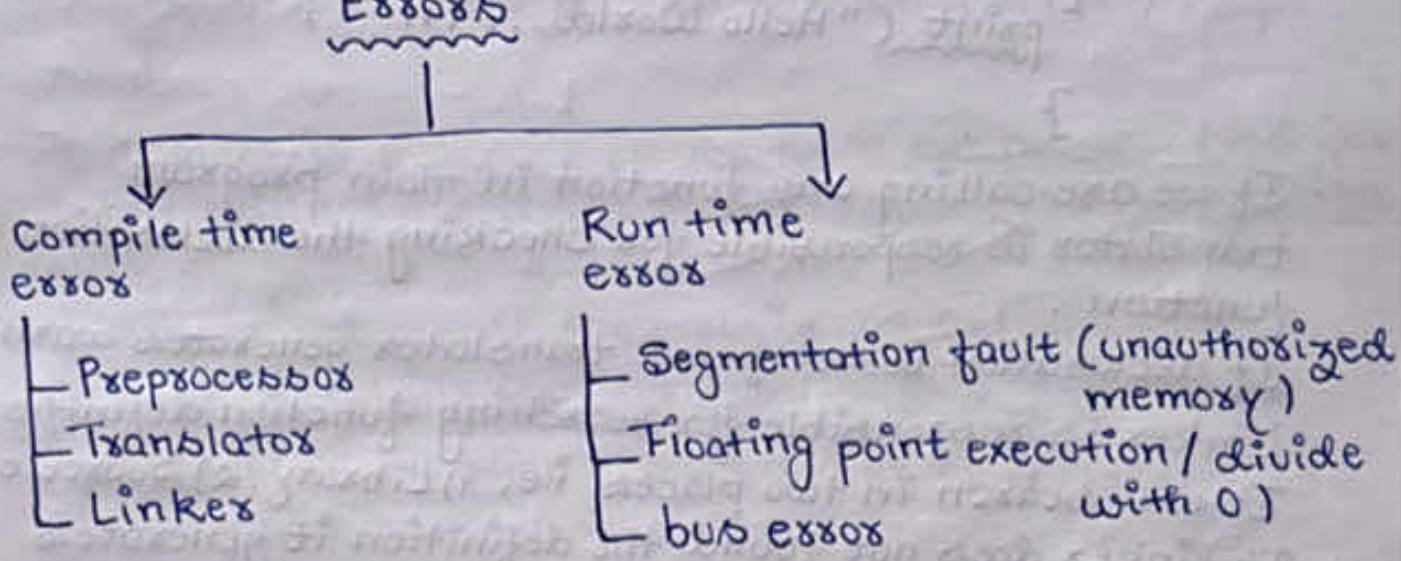
* Lib :- contains predefined function definition & contains multiple lines.

* Header files :- contains predefined function declarations.

cc -S p1.c :- with this command we are informing to the compiler to stop compilation after translator.

cc -c p1.c :- with this command we are informing to compiler to stop compilation after assembler.

*



- Compile time errors are generated by compiler.
- Run time error are generated by operating system.
- Header files errors are generated by preprocessor.
- Declaration or syntax error are generated by translator.
- Undefined error are generated by linker.
- Warning are generated by translator.
- Error are generated by Linker.
- Predefined are compiler supported function.
- Declaration is a type of prototype.

* There is a chance of getting assembler errors also. If we modify .s file & give it to assembler if assembler is unable to convert that code into op-code, then we will get assembler error.

```
# include <stdio.h>
void main ()
{
  print ("Hello World... \n");
}
```

- If we are calling any function in main program translator is responsible for checking the declaration of function.
- If declaration is not found translator generates warning
- Linker is responsible for searching function definition. It will search in two places i.e, 1) Library 2) Source code
- If linker does not found the definition it generates errors.

rm *.* :- delete all the files in a single command.

```
Ex :- ls ↵
data  p1.c  p2.c  Hello.c
      rm *.*
      ./a.out
      ls ↵
data.
```

```
#include <stdio.h>
void main()
{
    int i;
    printf("Enter i\n");
    scanf("%d", &i);
    printf("i = %d %o %x \n", i, i, i);
}
```

Variable name

Data type

Address of variables

- If memory is available in the RAM but not reserved for us but still we are trying to access we get segmentation fault.
- Memory is not at all available in our address space while we are trying that memory the bus error occurs.
- Floating point exception error / divide by 0 :-

```
#include <stdio.h>
void main()
{
    int i, j, k;
    printf("Enter the i and j \n");
    scanf("%d %d", &i, &j);
    k = i/j;
    printf("k = %d \n", k);
}
```

Ex :- i = 10, j = 2;

k = i/j;

k = 10/2;

k = 5.

↓

No error

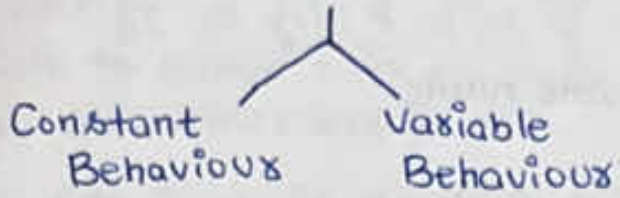
i = 10, j = 0

k = 10/0

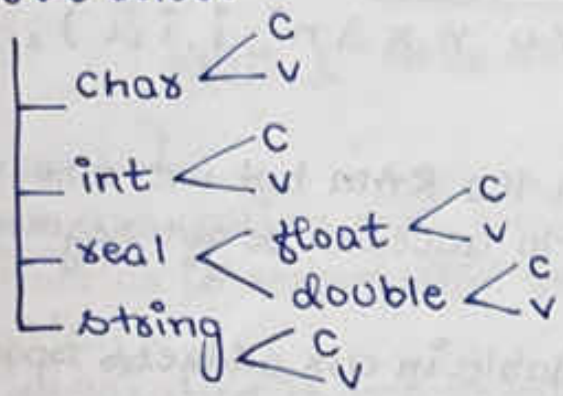
↓

floating point exception error.

* **Data Behaviours**



Data types



'1' - char constant
1 - integer constant
1.a - double constant
1.0f - float constant
"1" - string constant

- If we want to deal with variable nature of data we need to declare a variable.

We can declare a variable in two variable places :-

1) above the main which is called global variable.
2) within a function called as local variable.

* How to declare data types :-

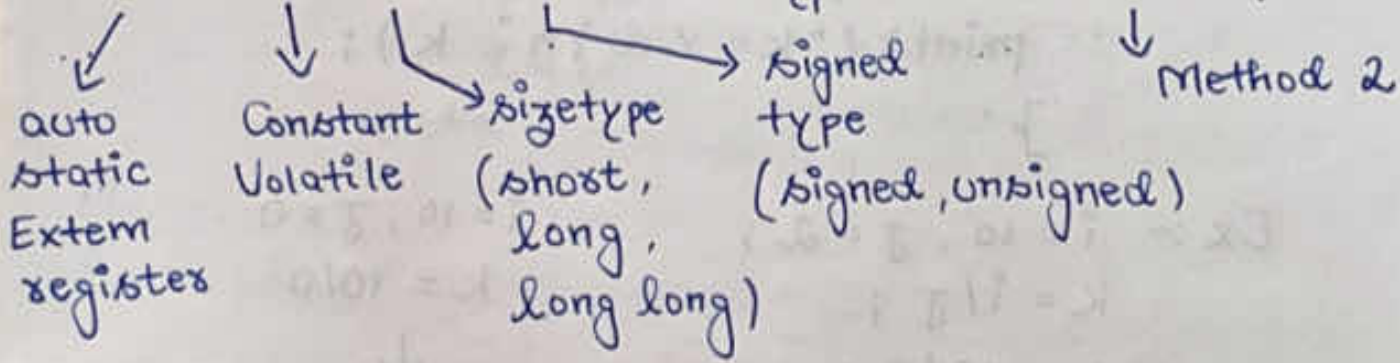
Method 1

Storage class type, type qualifier

datatype variable name ;

datatype variable name ;

Method 2



* Number System :-

Binary :- 0, 1 (2)

Octal (8) :- 0, 1, 2, 3, 4, 5, 6, 7

Decimal (10) :- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Hexadecimal (16) :- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Total number of conversion = 12.

* Binary Conversion :-

10101010_B → $[252]_8$

10101010_B → $128+32+8+2 = 170$ → $[170]_{10}$

10101010_B → $[AA]_{16}$

* Octal Conversion :-

$[1234]_8$ → $0001\ 0010\ 0011\ 0100$ → $[1010011100]_2$

$[1234]_8$ → $512+128+24+4 = 668$ → $[668]_{10}$

$[1234]_8$ → 3 binary digits → 1010011100 → $[29C]_{16}$

* Decimal Conversion :-

$[96]_{10} \downarrow_{B}^{D}$

2 | 96	
2 | 48	- 0
2 | 24	- 0
2 | 12	- 0
2 | 6	- 0
2 | 3	- D
1	- 1

$\Downarrow$

$[01100000]_2$

$[96]_{10} \downarrow_{O}^{D}$

8 | 96	
8 | 12	- 0
1	- 4

$\Downarrow$

$[140]_8$

$[96]_{10} \downarrow_{H}^{D}$

16 | 96	
6	- 0

$\Downarrow$

$[60]_{16}$

* Hexadecimal conversion :-

$[BCE]_{16} \downarrow_{B}^{H}$

$[1011\ 1100\ 1110]_2$

$[BCE]_{16} \downarrow_{O}^{H}$

3 binary digit

$[\underbrace{101}_{5}\ \underbrace{111}_{7}\ \underbrace{001}_{1}\ \underbrace{110}_{6}]$

$\Downarrow$

$[5716]_8$

$[BCE]_{16} \downarrow_{D}^{H}$

11 12 14

$\times 16^2$ $\times 16^1$ $\times 16^0$

$= (11 \times 16^2 + 12 \times 16^1 + 16^0 \times 14)$

$= 2816 + 192 + 14$

$= [3022]_{10}$

Binary Numbers

0 - 000	8 - 1000	16 - 10000
1 - 001	9 - 1001	17 - 10001
2 - 010	10 - 1010 - A	18 - 10010
3 - 011	11 - 1011 - B	19 - 10011
4 - 100	12 - 1100 - C	20 - 10100
5 - 101	13 - 1101 - D	
6 - 110	14 - 1110 - E	
7 - 111	15 - 1111 - F	

Q. Why we need to declare variable ?
⇒ To process variable nature of data .

```
#include <stdio.h>
void main ()
{
   char ch ;
}
```

Datatype ← char ch ; → Variable name

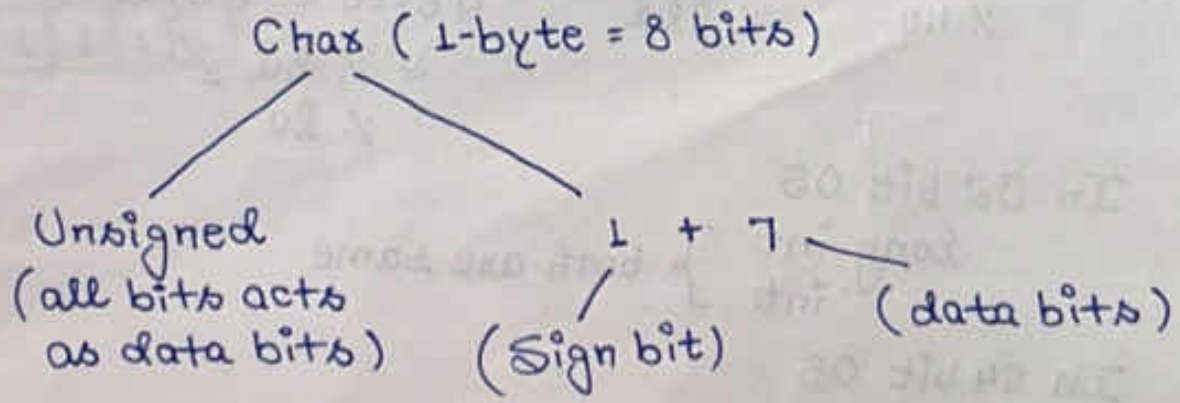
- When we declare a character variable compiler will reserve one byte of memory .
- When we declare a local variable bydefault we need to consider the data as junk data .

char ch ; :- declaration
ch = 'a' ; :- assigning
char ch = 'a' :- initialization i.e, declaring and assigning in a one line .

Characters are stored in its ASCII value :-

'a' = 97
'A' = 65
'0' = 48

- Characters are treated as 1 byte integer .



- If we are not mentioned any signed or unsigned then bydefault compiler considers as signed .

Q. Why we need to declare variable ?
⇒ To process variable nature of data .

```
#include <stdio.h>
void main ()
{
   char ch ;
}
```

Datatype ← char ch ; → Variable name

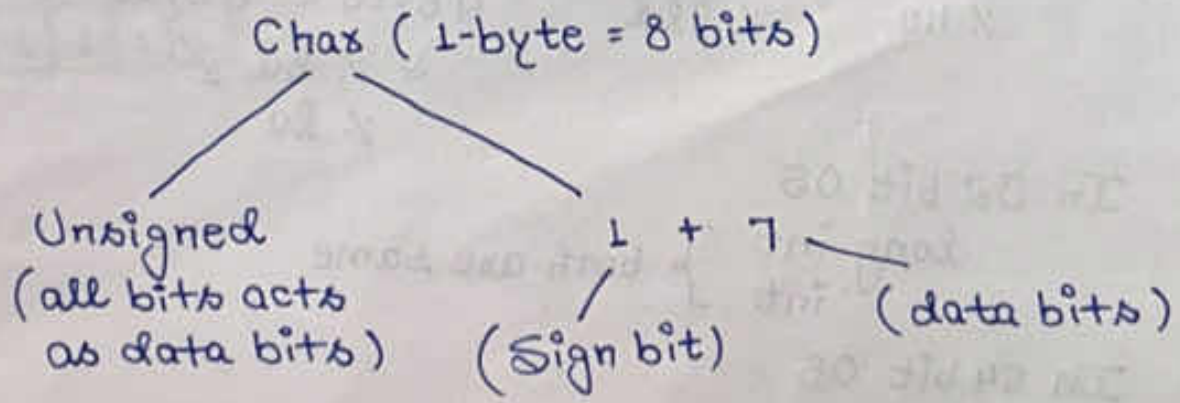
- When we declare a character variable compiler will reserve one byte of memory .
- When we declare a local variable bydefault we need to consider the data as junk data .

char ch ; :- declaration
ch = 'a' ; :- assigning
char ch = 'a' :- initialization i.e, declaring and assigning in a one line .

Characters are stored in its ASCII value :-

'a' = 97
'A' = 65
'0' = 48

- Characters are treated as 1 byte integer .



- If we are not mentioned any signed or unsigned then bydefault compiler considers as signed .

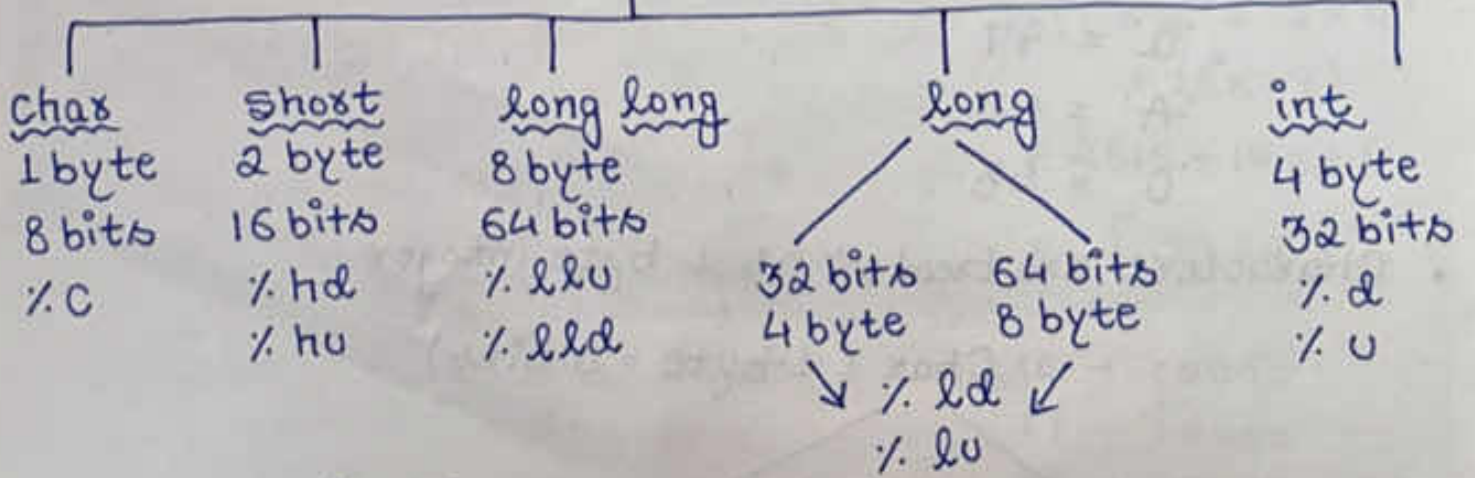
```
# include <stdio.h>
void main()
{
    unsigned char ch = 97;
    printf("ch = %d\n", ch);
    ch = ch + 3;
    printf("ch + 3 = %d\n", ch);
    ch = ch * 2;
    printf("ch * 2 = %d\n", ch);
    ch = ch + 60;
    printf("ch + 60 = %d\n", ch);
}
```

output

97
100
200
4

. If signed bit is '0', then no need to find 2's compliment whereas if signed bit is '1', then we need to find out the 2's compliment.

* Integers

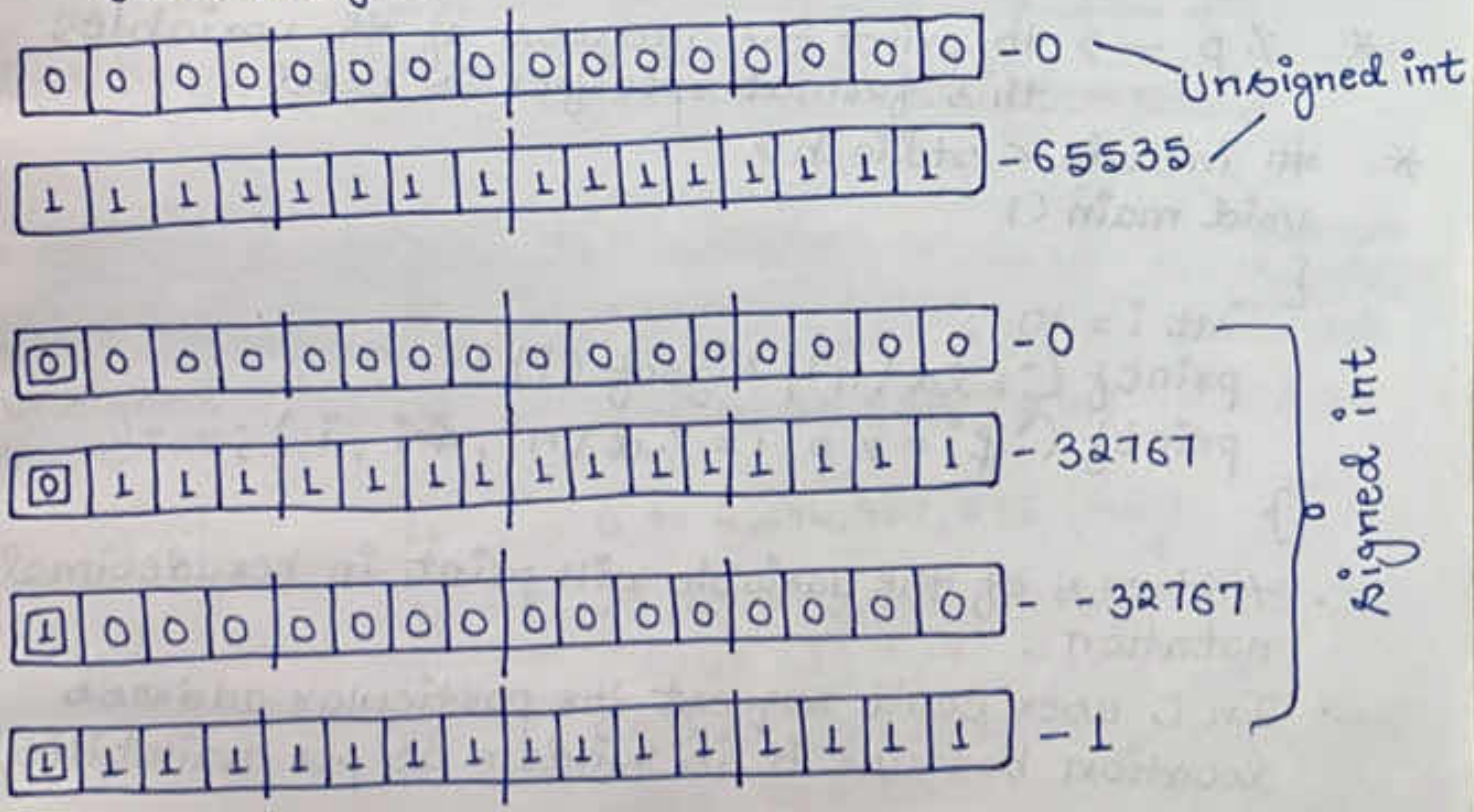


In 32 bit OS
long int, int } both are same

In 64 bit OS
long int, long long int } both are same

```
# include < stdio.h >
void main ()
{
  printf ("%ld\n", sizeof (char));
  printf ("%ld\n", sizeof (short int));
  printf ("%ld\n", sizeof (long int));
  printf ("%ld\n", sizeof (long long int));
  printf ("%ld\n", sizeof (int));
}
```

* Short integer :-



vi /usr/include/limits.h

* How to scan a short int and how to print on the screen.

```
#include <stdio.h>
void main()
{
    short int num;
    printf("Enter the number....\n");
    scanf("%hd", &num);
    printf("num = %hd\n", num);
}
```

* %p ⟶ to print the address of the variable, this format specifier is used.

*
```
#include <stdio.h>
void main()
{
  int i = 10;
  printf("%ld\n", sizeof(i));
  printf("&i = %p  i = %d\n", &i, i);
}
```

- Address of the variable will print in hexadecimal notation.
- In C user can't request for particular address location because it is random as per availability.

Q. In a given memory location how a given data is stored ?

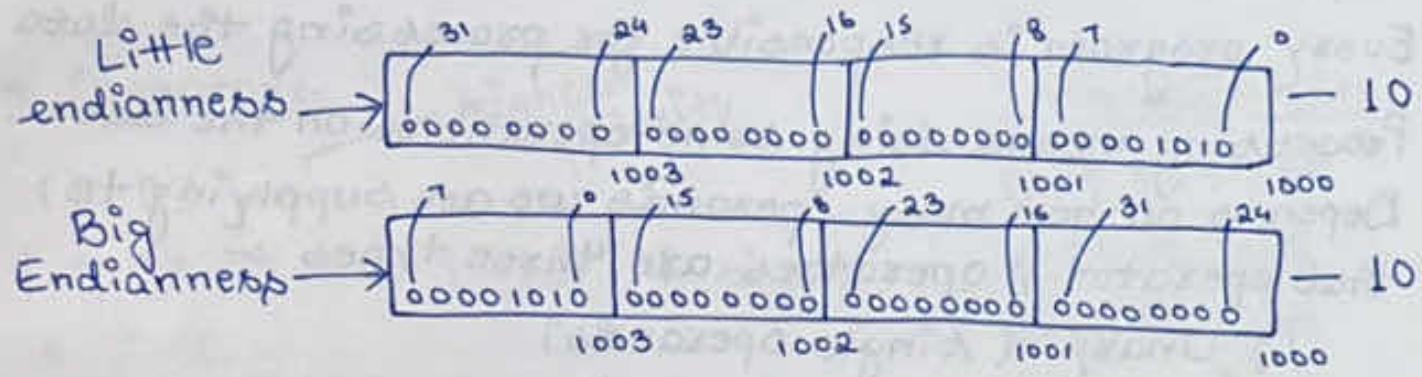
⇒ Endianness (byte ordering, byte arrangement, byte storage)

- Little endianness
- Big endianness

- Endianness is hardware dependent.

- In little endianness, LSB is stored in given lower address.
- In big endianness, LSB is stored in given higher address.
- Intel processor follow little endianness environment
- Motorola processor follow big endianness environment.

Ex :- int i = 10 ;



* Data type	Size (bytes)	Range	Format Specifies
short int	2	-32,768 to 32767	%hd
unsigned short int	2	0 to 65,535 (2G)	%hu
unsigned int	4	0 to 4,294,967,295 (4G)	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
Unsigned long int	4	0 to 4,294,967,295 (4G)	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c

float	4	* 1.2E-308 to 1.7E+308	%lf
double	4	* 1.2E-308 to 3.4E+38	%f
long double	16	3.4E-4932 to 1.1E+4932	%Lf

* Operators :-

- Every program is responsible for processing the data.
- Processing means doing some operations on the data.
- Depends on how many operands we are supplying to that operator, operators are three types :-
  1) Unary (single operands)
  2) Binary (two operands)
  3) Texanary.
- Depends on what operations that operator is doing again operators are divided into below types :-
  1) Arithmetic – (+, -, *, /, %)
  2) Relational – (>, <, >=, <=, ==, !=)
  3) Logical – (&&, ||, !)
  4) Bitwise – (&, |, ^ x-OR, ~ compliment)
  5) Shift – (<<, >>)
  6) Assignment – (=)
  7) inc and dec – (++, --)
  8) sizeof ()
  9) Reference – (&)
     defrence – (*)
     (Address)
  10) Conditional – (? :)
  11) Index operator – ([ ])

12) Grouping — (())
13) dot — (.)
14) comma — (,)
15) Arrow — (->)
16) compound assignment (+=, -=, *=, /=)

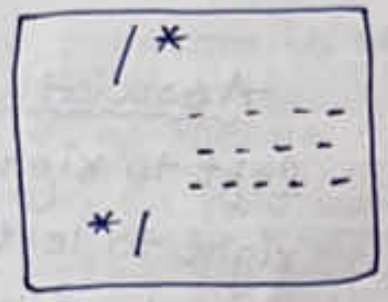
man operator — To see what were the operator compiler will support.

* Operators	Associativity
highest priority ↓	
() [] -> .	left to right
! - ++ -- + ~ (type)* & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= <<= >>= &= ^= |=	right to left
, lowest priority	left to right

- In an expression if multiple operators are present we need to think priority.
- In an expression if multiple operators with same priority comes we need to think associativity.

* Assignment operator (=) :-

- op1 = op2
- It is a binary operator requires two operands to do operands.
- Assignment operator will assign side side operand to left side and the result is whatever it assigned is the result.



→ To comment out particular paragraph in code.

```
#include <stdio.h>
void main()
{
 /* int i ;                          ↳ i = 10
    i = 10 ;
    printf ("i = %d \n", i) ;
 */
 /* int i = 10 , j ;
    j = i ;                          ↳ i = 10
    printf ("i = %d \n", i, j) ;       j = 10
 */
To comment single line — // i = 10 ;
    int i ;
    10 = i ;                         ↳ error
    printf ("i = %d \n", i);
}
```

```
i = 10 + 20 ; — no error
10 + 20 = i ; — error
i + j = k ; — error
```

(i + j together it will be a constant)

```
i = j = 10 ;  - no error
j = 10 = i ;  - error
```

error because of this assignment as per right to left associativity.

* W.A.P for swapping of two numbers using 3rd variable.

=>
```
#include <stdio.h>
void main ()
{
  int n1, n2, t ;
  printf ("Enter n1 and n2 \n");
  scanf ("%d %d", &n1, &n2);
  printf ("Before swap n1 = %d n2 = %d \n", n1, n2);
    t = n1 ;
    n1 = n2 ;
    n2 = t ;
  printf ("After swap n1 = %d n2 = %d \n", n1, n2);
}
```

Iput - [15] [20]

output - [20] [15]

* Arithmetic Operators :-

- ☐ + ☐
- ☐ - ☐
- ☐ * ☐
- ☐ / ☐
- ☐ % ☐

- All the above arithmetic operators are binary
- All the arithmetic operator will not modify the operands so we can supply both operands either constant or variable.

Ex :- 10 + 20
i + j
i + 20
20 + j

- We cannot use modulus (%) operator on real numbers (float or double).

```
int i = 13 , j = 2 , k
k = i / j ;   —  L> 6 - Quotient
k = i % j ;   —  L> 1 - Remainder
```

r = num % 10 - To get the last digit from a number

Ex :- r = num % 10
num = 123
r = 3 .

r = num % 100 - To get the last two digit from a number

Ex :- 123
r = num % 100
= 23 .

r = num % 2 - To find whether the number is real or not .

Ex :- 12 % 2
= 0 - even number

13 % 2
= 1 - odd number

num = num / 10 :- To delete the last digit from a given number

Ex :- num = 123
num = num / 10
= 12 .

* W.A.P to delete a last digit from a given i/p number .

=>
```
#include <stdio.h>
void main ()
{
  int num , r ;
  printf ("Enter the number \n") ;
  scanf ("%d", & num) ;
  r = num / 10 ;
  printf ("%d \n", r) ;
}
```

Ex :- 154
L> 15 .

* W.A.P to extract last digit from the number.

⇒
```
#include <stdio.h>
void main()
{
  int num, x;
  printf("Enter the num \n");
  scanf("%d", &num);
  x = num % 10;
  printf("%d \n", x);
}
```

Ex :- 154
↳ 4

* W.A.P for swapping of two number using arithmetic operators and without using temporary variable.

⇒
```
#include <stdio.h>
void main()
{
  int n1, n2;
  printf("Enter n1 and n2 \n");
  scanf("%d %d", &n1, &n2);
  printf("Before swap n1 = %d n2 = %d \n", n1, n2);
  n1 = n1 + n2;
  n2 = n1 - n2;
  n1 = n1 - n2;
  printf("After swap n1 = %d n2 = %d \n", n1, n2);
}
```

Input - [10] [20]

Output - [20] [10]

*
```
#include <stdio.h>
void main()
{
    int n1 and n2 ;
    printf ("Enter n1 and n2 \n", );
    scanf ("%d %d ", &n1, &n2 );
    printf ("Before swap n1 = %d n2 = %d \n", n1, n2);
    n2 = n1 + n2 - (n1 = n2);
```

n2 = n1 + n2 - (Exp)
= 10 + 20 - (Exp)
= 30 - 20
= 10

10 20 — I/P
20 10 — O/P

- Grouping is the highest priority but it will not solve the operand result inside that group.

* Different expressions for swapping :-

1)
```
t = n1 ;
n1 = n2 ;
n2 = t ;
```

2)
```
n1 = n1 + n2 ;
n2 = n1 - n2 ;
n1 = n1 - n2 ;
```

3)
```
n1 = n1 * n2
n2 = n1 / n2
n1 = n1 / n2
```

4) `n2 = n1 + n2 - (n1 = n2);` 5) `n2 = n1 * n2 / (n1 = n2);`

* When constant and variable of different types are mixed in an expression they are all converted to the same type.
* The compiler converts all operands up to the type of the largest operand, which is called type promotion.
* Once this step has been completed all the other conversion are done operation by operation as described in the following type conversion algorithm :-

→ IF an operand is a long double
THEN the second is converted to long double.

→ ELSE IF operand is a double
THEN the second is converted to double.

→ ELSE IF an operand is float
THEN the second is converted to float.

→ ELSE IF an operand is unsigned long
THEN the second is converted to unsigned long.

* When a compiler is solving any expression, it will allocate temporary memory to store the result temporary.
* That buffer size depends on operands.

* Relational Operators :- (< <= > >= == !=)
  ↓ (output always in 0's & 1's) ↓ (binary operators)

→
```
#include <stdio.h>
void main()
{
  int a=1, b=-2;
  printf("%d\n", a>=b);
}
```
↳ 1

→
```
{
  int a=1;
  unsigned int b=-2;
  printf("%d\n", a>=b);
}
```
↳ 0

// -2 is converted to (signed to unsigned) then comparison will take place.

→
```
{
  int a=-1;
  unsigned int b=2;
  printf("%d\n", a>=b);
}
```
↳ 1

→
```
{
  printf("%d\n", 1>=2<=3==4<5!=2);
}
```
1>=2 → 0<=3 → 0==4<5 → 0==4<5 → 0==1 → 0!=2 → 1

↳ 1

→
```
{
  int a=1, b=2, c=3, d=4, e=5, f=6, x;
  printf("%d\n", x);
  x=a+1<3+b-c%d==e-f;
}
```
↳ 0

* Logical Operators :- (&& || !)

binary unary

- All these operators will not modify the operands so we can supply the operands as constant or variable.
- All these logical operator gives the result either 0 or 1.

A	B	A&&B
0	0	0
0	1	0
Non-zero 1	0	0
1	1	1

A	B	A||B
0	0	0
0	1	1
Non-zero 1	0	1
1	1	1

A	!A
0	1
Non-zero 1	0

*
```
#include <stdio>
void main()
{
  int = 10, j = 20, k;
  k = i && j;
  printf("k= %d \n", k);      — 1
  k = i || j;
  printf("k= %d \n", k);      — 1
  k = !i
  printf("k= %d \n", k);      — 10
}
```

*
```
#include <stdio.h>
void main()
{ int i = 10;
  printf("i= %d \n", i);      — 10
  !i;
  printf("i= %d \n", i);      — 10
}
{
  ..  "  "  "
  "   "  "  "                 — 10
  i = !i
  printf("i= %d \n", i);      — 0
}
```

- In logical AND if the first operand is 0, compiler will not check the second operand, it decides the result as 0.
- In logical OR if the first operand is 1, compiler will not solve the second operand, it decides the result as 1.

*
```
{
int i=0, j=20, k;
printf("i=%d j=%d\n", i, j);
k = i && (j=200);
printf("i=%d j=%d\n", i, j);
printf("k=%d\n", k);
}
```
↳ 0 20 0

```
{
int i=10, j=20, k;
" " " " "
k = i && (j=200);
printf("i=%d j=%d k=%d\n", i, j, k);
}
```
↳ 10 200 1

Q. If we gives bigger expression to the compiler how the compiler is going to be solved that expression.

⇒
```
#include <stdio.h>
void main()
{ int i=10, j=20, k=30, l=40, m=50, x;
x= i && (j=200) || (k=300) && (l=400) || (m=500);
printf("i=%d j=%d k=%d l=%d m=%d x=%d\n",
        i,j,k,l,m,x); }
```

x = i && exp || exp1 && exp2 || exp3
(i: 1, exp: 1)

x = exp4 || exp5 && exp6
(exp4: 1, exp5: x)

x = exp6 || exp6
(exp6: 1, exp6: x)

x = exp7
(exp7: 1)

x = 1

10 200 30 40 50 1

```
{ int i=10, j=20, k=30, l=40, m=50, x;
  x = i || (j=200) && (k=300) || (l=400) && (m=500);
  printf("i=%d j=%d k=%d l=%d m=%d x=%d\n",
          i, j, k, l, m, x);}
```

x = i || exp && exp1 || exp2 && exp3

= i || exp4 || exp5

= exp6 || exp5

= exp7

= 1

10
200
300
40
50
1

* Bitwise Operators :- (&, |, ^, ~, <<, >>)

- All these bitwise operators are binary operators except compliment (~) bitwise operator.
- All these bitwise operators will not modify the operands, so we can supply the operands as constants as well as variable.
- We cannot use bitwise operators on real numbers (float & double).

A	B	A&&B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

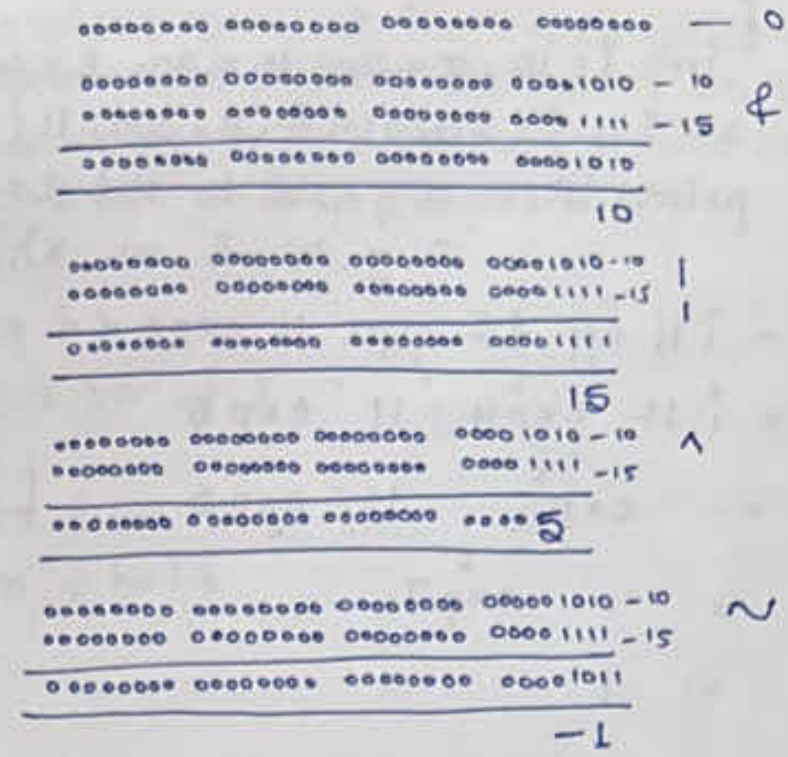
X-OR

A	~A
0	1
1	0

Compliment

```
#include<stdio.h>
void main()
{ int i=10, j=15, k;
  k= i&j;
  printf("k=%d\n",k);
  k= i|j;
  printf("k=%d\n",k);
  k= i^j;
  printf("k=%d\n",k);
  k= ~i;
  printf("k=%d\n",k);
}
```

↳ 10
15
5
-11




```
{ int i= -1
  printf(" i = %d  i= %u \n", i, i);
}
```

↳ i = -1
i = 4G1

* Comparison :-

! ☐	~ ☐
1) Unary operator.	1) Unary operator.
2) Logical operator.	2) Witwise operator.
3) Operands will not modify.	3) Operand will not modify.
4) o/p always in 0 & 1.	4) Any number depends on i/p.
5) Operands can be any type	5) Operand can be only integer family (i.e float & double not allowed)
6) No need to convert the number into binary.	6) Binary conversion required.
int i=10, j; j= !i; = 0.	int i=10, j; j= ~i; = -11.

* Shift operators :- (<< >>)

□ << □ - left shift

□ >> □ - right shift

- These are binary operators.
- Modification of operands is not done.
- 1st operand is shifted by second operand times.
- 1st operand only need to convert into binary.

results = num x 2^{shifts}

x = 10 << 2

= 10 x 2^2

= 40.

```
{ int i=10, j=2, k;
  k = i<<j;
  printf("k=%d\n", k);
}
```

$1 \times 2^0 = 1$
$1 \times 2^1 = 2$
$1 \times 2^2 = 4$
$1 \times 2^3 = 8$
$1 \times 2^4 = 16$
$1 \times 2^5 = 32$
$1 \times 2^6 = 64$
$1 \times 2^7 = 128$

- If the 1st operand is a character, 2nd operand should be in between 0 to 7. [char] << [0 to 7]
- If the 1st operand is a short int, 2nd operand should be in between 0 to 15. [short int] << [0 to 15]
- If the 1st operand is a int, 2nd operand should be in between 0 to 31. [int] << [0 to 31]
- If the 1st operand is a long int, 2nd operand should be in between 0 to 63. [long int] << [0 to 63]

Same for both << as well as >>

- If the 2nd operand is greater than 63 than output will be unpredictable.

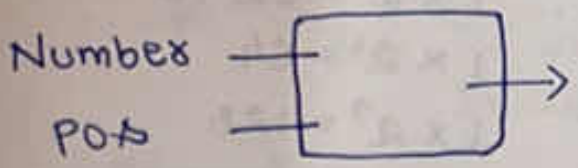
- If every left shift number becomes doubles whereas in every right shift number becomes half.

```
{
  int i=0, j=1, k;
  int i = i-1;  —  -1
  k = i>>j;
  P.f("k=%d\n", k);
}                  └> -1
```

```
{
  unsigned int i=0, j=1, k
  i = i-1;  —  4G
  k = i>>j;
  P.f("k=%d\n", k);
}                  └> 2G
```

* Sign bit copy mechanism :-
if signed bit is 1 than right shift will be in the form of 1

Sign :- (0-1) = -1
Unsigned :- (0-1) = 4G



Num	pos	Set
10	0	11
10	1	10
10	2	14
10	3	10
10	4	26

Num	pos	clear
10	0	10
10	1	8
10	2	10
10	3	2
10	4	10

Num	pos	Compliment
10	0	11
10	1	8
10	2	14
10	3	2
10	4	26

num = num | 1<<pos

num = num & ~(1<<pos)

num = num ^ 1<<pos

Step 1 :- 1 << pos
Step 2 :- ~ for clearing only
Step 3 :- num | step 1 o/p — Set
num & step 2 o/p — Clear
num ^ step 1 o/p — Compliment

num	pos	r = num & 1<<pos	r = num >> pos & 1
15	0	1	1
15	1	2	1
15	2	4	1
15	3	8	1
15	4	0	0
15	5	0	0

`r = num >> pos & 1` ⟶ gives o/p always in 0's & 1's
0 - if the bit is clear.
1 - if the bit is set.

* Conditional Operator (Ternary operator) :- (__ ? __ : __)

- op1 ? op2 : op3
- The result of this ternary operator is either op2 or op3 depends on op1.
- If op1 is non-zero result is op2 else if op1 is zero result is op3.

```
k = i<j ? 100 : 200 ;
k = 100 .

{
  int i=10, j=20, k ;
  k = i<j ? 100 : 200 ;      L> k = 100
  p.f ("k = %d \n", k);
}
i<j ? k=100 : k=200 ;        L> lvalue required, error
```

op1 ? op2 : op3

- If there is any expression in the op3 then we need to do grouping of that expression in order to make that whole expression after colon (:) as op3.

```
i<j ? k=100 : k=200   ↳ error
op1   op2    op3

i<j ? k=100 : (k=200)   ↳ no error
op1   op2      op3
```

* Nested Conditional operation :-

```
{ int i=100, j=200, k=300, x ;
  x = i>j ? (i>k ? i : k) : (j>k ? j : k)
      op1      op2            op3
  printf ("x = %d \n", x);
}                    ↳ x = 300
```

- In ternary operation if op3 contains assignment operation we need to group it otherwise it is a error.
- In ternary operator op2 & op3 can be a function call

* W.A.P to scan the numbers from the user & display is it odd or even.

```
{ int num ;
  P.f ("Enter the number ... \n");
  S.f ("%d", &num );                    ↳ 8
  x = x = num % 2 ;                       Even
  x ? printf ("odd \n") : printf ("Even \n");
}

num % 2 ? printf ("odd \n") : printf ("Even \n");
```

* W.A.P to scan a number and bit position from the user and display in that number that bit is set or clear.

=>
```
{ int num, pos ;
  p.f ("Enter the number .. \n") ;
  s.f ("%d", &num) ;
  p.f ("Enter the bit pos .. \n") ;
  s.f ("%d", &pos) ;
  num >> pos & 1 ? printf ("set \n") : printf ("clear \n");
}
```
↳ 15
03
= set .

```
{
  ..  ..  ..
 num & 1 << pos ? p.f ("set \n") : p.f ("clear \n") ;
}
```
↳ 15
5
= clear .

* Functing data passing :-

```
{ int k = 35 ;
  p.f ("%d %d %d \n", k == 35, k = 50, k > 40) ;
}
```
↳ 0, 50, 0

Step 1 :- Function data passing happens from right to left
Step 2 :- Printf happens from left to right.

* Whatever present in printf statement is called as arguments. (Ex :- p.f ("..... ", ...))

Arguments

* Printf function return type :-

printf returns a number of printable charactable count

```
{
 int i ;
 i = printf ("Hello\n");
 printf ("i= %d \n", i);
}
```

↳ Hello
i = 6

```
{
 printf ("%d \n", printf (" Hai ... \n"));
}
```

↳ Hai ...
7 ⟶ include \n as a char to count.

* Increment and decrement operator :-(++ --)

- These are unary operator
- These operator will modify the operands so only variable are allowed as operands .
- These operators are again divided into two

a) Pre increment or Pre decrement

++□ ↑↓
--□

b) Post increment or Post decrement

□++ ↑↓
□--

- Increment operator increment the operand by 1 value .
- Decrement operator decrement the operand by 1 value .

```
{ int i = 10 ;
  p.f ("i= %d \n", i);
   ++i ;
  p.f ("i= %d \n", i);
}
```

↳ i = 10
i = 11

```
{ int i= 10 , j ;
  j = i++ ;
  P.f ("i= %d j = %d", i, j);
}
```
↳ i = 11, j = 10

← Post increment or decrement

```
{
  j = i-- ;
}
```
↳ i = 9, j = 10

```
{
  ++i ;
}
```
↳ i = 11, j = 11

← Pre increment or decrement

```
{
  j = --i ;
}
```
↳ i=9, j=9

Pre increment :- first it will increment the value and then do assigning.

Post increment :- first it will assign the value and then do increment.

- If we want to increment the operand by 1 value use increment operator. (i.e, ++i)
- If we want to increment the operand by more than 1 value use arithmetic + an assignment operator. (i.e, i = i+ 5)

* The behaviour of increment & decrement operations in printf function.

```
{ int i = 10;
  printf("%d %d %d \n", i, i, i);
}          L> 10 10 10

{ int i = 10;
  printf("%d %d %d \n", i++, i++, i++);
}          L> 12 11 10

{ int i = 10;
  printf("%d %d %d \n", ++i, ++i, ++i);
}          L> 13 13 13

{ int i = 10;
  printf("%d %d %d %d %d \n", i++, ++i, i, i++, i);
}          L> 12 13 13 10 13
```

- In pre increment & no increment the updated value gets printed in printf function.

```
{ volatile int i = 10;
  printf("%d %d %d %d %d", i++, ++i, i, i++, i);
}          L> 12 12 11 10 10
```

* Code Optimization :- When we supply a program to the compiler, compiler observes the code and do necessary changes if required. Those changes technically called as code optimization.

Compilers will do optimization for two reasons :-

1) To save the memory.
2) To make our code to execute faster.

- Compiler to compiler code optimization techniques varies or changes.

Ex :- 1 int i, j, k, l, m, n

In the above example programmer declared 6 variables out of them only 4 is used, remaining two is unused. Smart compiler will not allocate a memory for remaining two variable, so memory gets save.

Ex :- 2
```
x = num % 10 ;
sum = sum + x ;
```

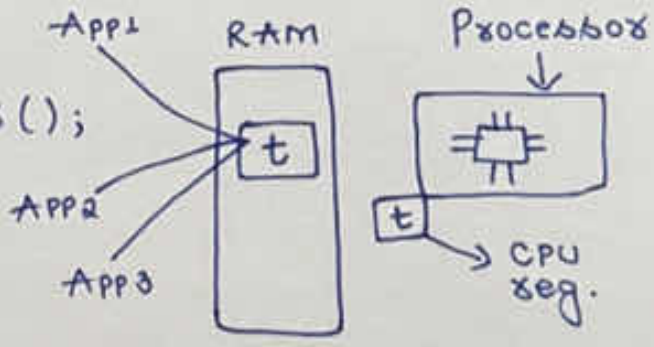
```
sum = sum + num % 10 ;
```

In the above example first two example are merged and written as a single expression so that no change in the result but variable is saved and one operation is reduced.

Ex :- 3
```
void abc (void)
{
- - - -
- - - - -
return 0 ;
- - - -
- - - - -  --> Dead code
- - - - -
```

In the above example function after return, programmer written some code which will never ever execute. Compiler treats that code as dead code and removes from executable files so that memory is saved.

Ex :- 4
```
while (1)
{
 t = get_data_sensor ();
 - - - - -
 - - - - -
 - - - - -
 - - - - -
}
```



In the above example we are collecting data from sensor and storing into the variable called t continuously.
So, some smart compiler observes this code and create a duplicate memory for t variable in CPU register to make the program run faster.

Advantages :-

1) Program runs faster - fetching the data from external RAM and fetching the data from internal register, in these two internal CPU registers accessing is faster.

Disadvantages :-

1) The sensors data is updating in CPU register whereas some other application are accessing the data from RAM so that the applications will not get real time data.

To overcome this disadvantage while declaring the variable use volatile Keyword (when we create volatile, compiler will not create duplicate copy).

Q. What is a volatile ?

⇒ • Volatile is a type-qualifier.
• When we declare a variable with volatile keyword we are informing to the compiler don't optimize that variable (don't create a duplicate copy).

* While we are writing an expression we need to avoid increment or decrement operation more than one time on single operand in a single line.