C Programming

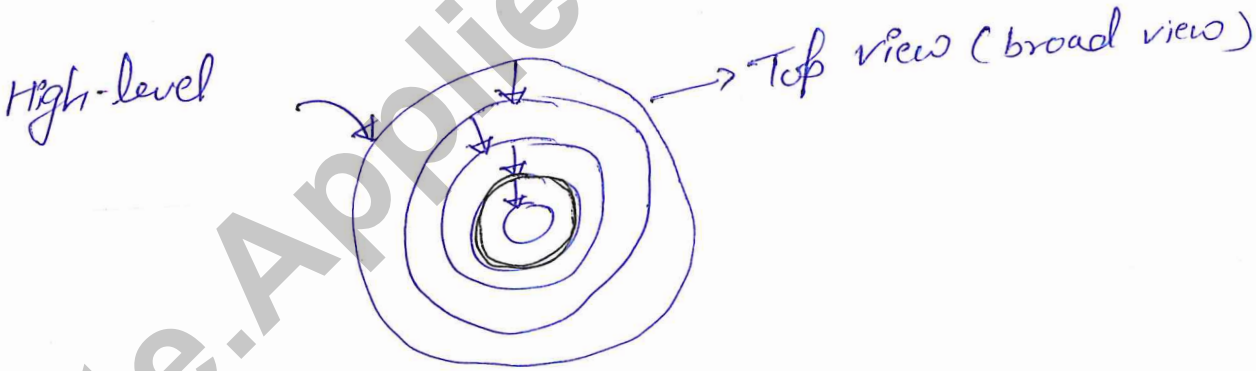
Introduction to C Programming

1.1 C: What, Why and How

(1) Basic Assumptions: don't know programming
some programming
get better

Basic

(2) Peeling an onion: Layer-by-layer



Start with each topic and dive completely into it.

(3) Coding: online tool → Linux, Mac, Windows

Share code ← Googledoc, email, execute

What is programming ?

It's the sequence of step by step or precise instructions that give to a computer to perform a specific task.

It is very similar to english

Like english is a way to communicate with people & people have intelligence. Similarly languages are to communicate between people & computer.

Why C ?

For understanding C++, you need to learn 'C'.
Every programming language has its role.

JavaScrip (JS) is very popular for building websites or run code in browsers.

For ML & AI course, python is popular.
For writing android app, java is used

All the operating systems like android, win, Linux, mac, ios are built using C/C++

Kernal: heard of operating system.

Lot of scientific computing, C/C++ is used.

AI/ML [Tensorflow] → built using C/C++

Weather forecasting

In general, C/C++ code is very faster than a code written in java, python, JS and PHP.

⇒ And also gaming softwares are also written using C/C++. (has to be faster)

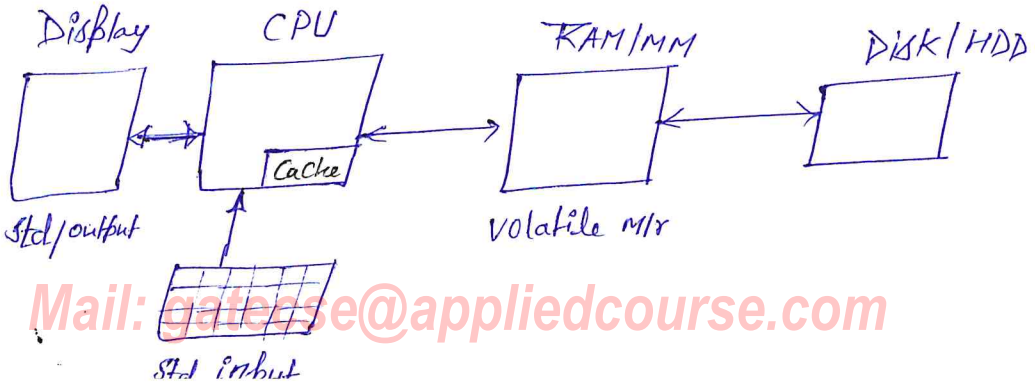
⇒ Drivers are also written in C/C++.

⇒ hardware/IOT is also designed using C/C++.

⇒ If you want to learn foundation of computer science and want a very fast run code.

⇒ If speed of execution matters, then C/C++ is probably the best choice.

[1.2] Simplified model of a computer



bask.com (for image)

① Chip [ ] → CPU (Central Processing Unit)
↓
It does all the computation

② RAM slot → (Thin horizontal strip)
(Main-Memory)

③ Disk (Hard Drive) : use to store data

④ FAN for cooling or for dissipating heat

⑤ PSU → connected to electrical socket

⑥ CD-ROM →

⑦ Mother Board → Everything is connected to this.

⑧ Video cards → are useful to run game faster.
↳ GPU

1.3 Tools for coding

⇒ extension for c program ⇒ .c

main.c (file) → [compiler] → executable file

executable file → loaded into RAM & accessed by processor → Display, Input

GCC → GNU C compiler (most popular c compiler)

GCC comes inbuilt in Linux, Mac.

⇒ use online compiler, easier/faster to

① onlinegdb.com ② ideone.com

Basics with examples

[2.1] Input and output.

Full Chapter

Objective: high level overview of key concept

Not rigorously

Std input → keyboard
Std. output → Monitor
} functional

```
main() {
Printf("Hello World");
}
```

→ Does not follow many good practi
→ Show on Screen

Goto the main fuch O/P: hello world

⇒ all execution of c programs is starts from main.

⇒ Printf means print formatted.

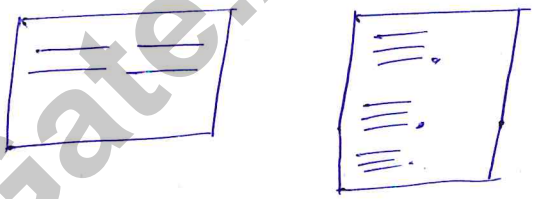
⇒ Whatever I want to print in printf, just but it in double quotes (" ").

⇒ If you miss ; after printf.
error : expected ; before } .

⇒ If you forget) in printf statement
error : expected ')' before } token.
error : expected ';' before } token.

⇒ Syntax → Rules of C

⇒ Print formatted means printing in some format



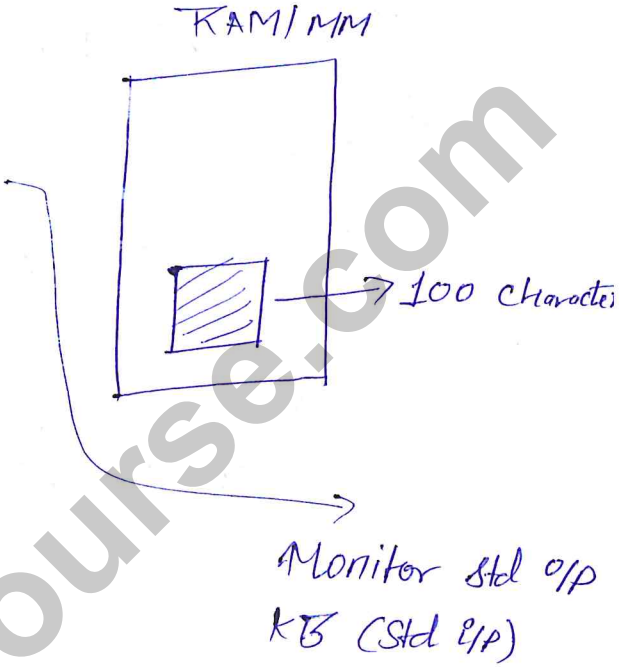
```
main ()
{
Printf ("hello World \t I am here");
}
```

O/P : hello World I am here

\n ⇒ new line

```
main ()
{
 char n[100];
Printf("What's your name");
Scanf("%s", n);
Printf("Hello, %s", n);
}
```

String



O/P: What's your name: veerna
hello, veerna

[2.2] Choosing one option versus another → C/C++/Java

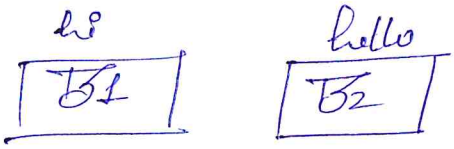
eg. Imagine MCQ

Question
a.
b.
c.
d.

choice

If correct ans = a
and your ans = a
then marks
else dont mark

eg



If B1 is pressed print hi
and B2 is pressed print hello

```
main(){
int a;
Printf("enter a number");
Scanf("%d", &a);
if(a%2==0){
Printf("even");
}
else{
Printf("odd");
}
}
```

&a (am)

(&a)
|
address of a

O/P: enter a number 12 — 12%2 remainder = 0
even — so even

```
main(){
int a;
Printf("enter a number:");
Scanf("%d", &a);
if(a%3==0){
Printf("remainder is 0");
}
else if(a%3==1){
Printf("remainder is 1");
}
else
Printf("remainder is 2");
}}
```

O/P: enter a number: 7
remainder is 1

2.3 Repeating over and over

Machines can repeat over & over unless humans, they tired

Task: if a is prime no. or not.

Passwords, banking → encrypted

⇒ Compilers are brilliant to repeat over & over.

⇒ Prime no. are used for encryption.

⇒ **Cryptography:** Prime no. are core of cryptography.

a = 12

logic: 2, 3, 4 11
↑ ↑
2 a-1

If 12 is divisible by one number then it's not prime.

exactly divide a % i = 0
↑
2, 3, 4 . .

repeatedly checking for 2 . . 11

12 % 2 = 0 12 is not prime

Ex:

a = 13

i = 2, 3, 4 12

13 % 2 = 1
13 % 3 = 1
13 % 4 = 1
13 % 5 = 3
:
13 % 12 = 1

13 % i ≠ 0

∴ 13 is prime no.

repeating → Loops

for choices → If - else

for repeatation → for loops

```
main()
{
 int a;
 int i;
Printf("enter a number:");
Scanf("%d", &a);
for(i=2; i<=a-1; i++);{
    if(a%i==0){
     Printf("NOT PRIME, divisible by %d", i);
       exit(0);
          }
          }
Printf("PRIME");
}
```

Output:

for (i=2; i<a-1; i++)

13%2 = 1

a	i
13	2

for good programming : logic → Write down the logics in english

↓

Convert it C (Syntax)

[2.4] Reuse & Simplify

Task : input two num ⇒ input 1st num → Prime or not
input 2nd num → prime or not

Logic : input 1st num → 3 lines
{ for loop to test if num is prime or not → 6 lines
input 2nd num → 3 lines
{ for loop to test if num is prime or not. → 6 lines

isPrime(a)
1 : Prime
0 : not Prime

```
a_P = isPrime(a);
if (a_P == 1) {
 Printf("%d is prime", a);
 }
 else {
Printf("%d is NOT a prime", a);
 }
}
```

```
main()
{
  int a;
  int a_P;
Printf("enter a number");
Scanf("%d", &a);
  a_P = isPrime(a);
  if (a_P == 1)
    Printf("%d is prime", a);
  else
    Printf("%d is not prime", a);
Printf("\n enter another num:");
  Scanf("%d", &a);
```

O/P: Enter another number: 13
13 is prime

13 / 15

```
#include <stdio.h>

int isPrime(int a){
   int i;
for(i=2; i<a-1; i++){
     if(a%i==0){
          return(0);
        }
      }
      return 1;
   }
   main()
   {
   int a;
   int a_P;
   Printf("enter a number:");
   Scanf("%d", &a);
   a_P = isPrime(a);
   if(a_P == 1){
   Printf("%d is Prime", a);
   }
   else{
   Printf("%d is NOT a prime", a);
   }
}
```

1 if a is prime
0 NOT
isPrime(15)

logic to determine prime or not

Core basics:
input and output
Choice of if-else/
repeat (for)
reuse (function)
file - disk → pointers

[3] Lets dive in

We will start from basics & dive into it to the broad overview.

[3.1] Character set in C

Set of characters, that are used in C.

A, B, C ---- Z } alphabets
a, b, c, ---- z }

0, 1, 2 ---- 9 } digits

Space, newline } white spaces

for ex:

⇒ If after printing, abcd you want to go to the new line that is white space.

⇒ new line is a character, which take you to the new line.

⇒ a b : a space b
a b : a tab b (multiple spaces)

+, *, <, >, $, ! → Special char

⇒ just google as special characters in C.

ASCII - Older → 0 to 255 [256 numbers]

Unicode - Modern → 1,114,112

(large character set)

'A' → Number → binary → Memory - RAM/disk

Character : Numeric representation → binary form → Memory

⇒ C uses ASCII codes. There are some modern C compilers which use unicode.

⇒ Just google ascii characters in C.

'A' → 65 '0' → 48

'a' → 97

} Standered mapping

⇒ TAB is also printable character

⇒ characters from 128 to 255 is not printable char

[3.2] Keywords & Identifiers

⇒ Words are basically combination of several character.

Words → Keywords → 32 predefined words in 'C' which we can not use for any other purpose.

Ex: if, else, int, char, while

→ Identifiers

⇒ google search for keywords

Identifiers:

variables, function

⇒ These are words that we use for variable name, func^n name

⇒ Identifiers are valid words in C.

① Alphabets, _ (underscore), digits are allowed.

② First character in identifier needs to be an alphabet (U/L) and an underscore. It can not be a digit

③ You can not use keywords as identifier or variable name.

④ These are case - sensitive.

```
abc = 10
aBc = 20
```

⑤ length of Identifier is compiler/implementation dependent.

→ older (8)
→ ANSI C → 31 char

gcc - visual studio (tool of Microsoft)

Ex: Invalid Identifiers:

① abc 47 ✗ (↳ space)

② ab!cd ✗

③ 2ab ✗

④ if ✗

3.3 Data Types

⇒ Fundamentally there are four types of data types

1. char to store characters.
2. int to store integers.
3. float → single precision
4. Double → Double precision } store real values.

void Data type

⇒ It means nothing

⇒ useful when we implement function

ex:

char a; → It means in RAM you are creating some space which you are accessing using a. And you can store one char in it

⇒ There are two types of qualifiers

1. short long
2. signed, unsigned

⇒ secondary types of data types

Array, String, pointers, structures, enumerators

⇒ No. of bytes used to represent a character depends on the actual system and compiler.

Byte → 8-bits → _ _ _ _ _ _ _ _ (each _ : 0/1, bit)

in each space, you can store either 0 or 1.

Data types	Range	Byte	format
Signed char	-128 to 127	1	%c
Unsigned char	0 to 255	1	%c
Short signed int	-32768 to 32767	2	%d
Short unsigned int	0 to 65535	2	%u
signed int	-32768 to 32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to ...47	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to 3.4e38	4	%f
double	-1.7e308 to 1.7e308	8	%lf
long double	-1.7e4932 to 1.7e4932	10	%Lf

⇒

0 0
0 1
1 0
1 1

$2^2 = 4$ for 8 slots ⇒ 2^8 = 256

⇒ Character is also represented as number internally.

ASCII → A - 65, a → 97, 0 →

sign char

use

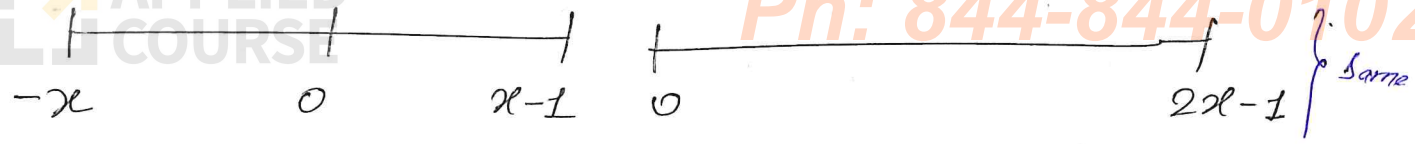
use first bit to represent sign

represent no.

+0
-0
are same

⇒ signed basically means you have both signed & negative & positive values.

⇒ unsigned - only positive values



⇒ Short means you are going to use lesser amount of mlr.

⇒ Long means larger memory.

⇒ float can stor

⇒ long double – for storing extremely large number use for scientific purpose.

```
main()
{
Printf("sizeof(char) = %u byte \n", sizeof(char));
Printf( sizeof(short) = %d byte \n", sizeof(short));
Printf( sizeof(int) = %d byte \n", sizeof(int));
Printf( sizeof(long) = %d byte \n", sizeof(long));
Printf(sizeof(double) = %d byte \n", sizeof(double));
Printf( sizeof(float) = %d byte \n", sizeof(float));
Printf( sizeof(long double) = %d byte \n", sizeof(long double));
 return 0;
}
```

sizeof ( ) : this function is used for getting size of what you give

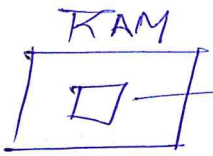
find factorial of large no. 1000 and store it.

1000! → linked-list

↳ Integer can not store such a large no.

[3.4] variables

int a;



creating memory for storing some integer here a is called variable.

all variable names → Identifiers

↳ Data types

```
main (){
int a;
a = 10;
a = 5;
}
```

here value is changing from 10 to 5 value is varied so it's called variable

2 Main concept of variable

→ Declaration
→ initialization
→ usage

⇒ you have to always declare a variable before using it.

⇒ Typically variable name starts with lower case or (–) underscore.

fName

⇒ We give variable names, which are easy to read.

int id ; int x ;

float f = 0.23 ;

double d = 1.2e7 Scientific representation

1.2×10^7

⇒ int a, b, c, d ; declaring multiple variable in a single line

⇒ int a = 5, b = 2 ; declaring & initializing variables

⇒ int a, b, c = 0 ; three variables with value 0

declaration initialization

```
int a ;
int b ;
int c ;
a = 0 ;
b = 0 ;
c = 0 ;
```

instead of writing 6 line code, we are writing it in one line.

3.5 Constants → do not change

Types: char, string, integer, real-valued

define constant : const int a=5;

Char: '#', 'g', '$'

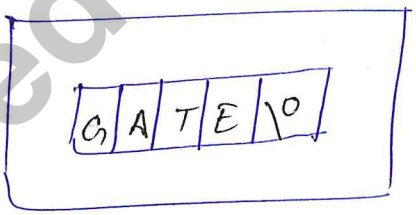
'abc' X Not a char

' ' X empty can not be char

g X Not valid char w/o single quotes.

String: sequence of characters.

"GATE"



\0 - ASCII = 0

Null terminator

⇒ If we don't store back-slash (\0) then it will everything it gets because memory is contiguous.

⇒ ('\0') for this, we get to know our string is ended here.

Real-value : 1.2f, 1.2f, 1.2L, 1.2l

integer → decimal system

Const int a = 123;

1	2	3
x	x	x
10^2	10^1	10^0

100 + 20 + 3 = 123

123456L
↳ long int const

1234U
↳ unsigned integer const

1234UL
↳ unsigned log integer constant

Octal System → 0, 1, 2, 3, 4, 5, 6, 7 (only 8-digits)
(Base 8)

int a = 012; { read this as octal as we are placing 0
octal number

~~0~~	1	2
	x	x
	8^1	8^0

8 + 2 = 10 decimal

Hexa-decimal (Base 16)

0, 1, 2 - - 9, A, B, C, D, E, F
10 15

0X or 0x

0 followed by capital or small x. int a = 0x12

$(12)_{16}$

```
  1 2
  X X
 16 16
-----
 16+2 = 18
```

int a = 8,126 ✗ comma

2.5 ✗ decimal

12 5 ✗ space

mai

⇒ other way of defining constants #define

```
#define PI 3.14159
main()
{
  const a=5;
  Printf("PI: %f", PI);
  int x=012;
  Printf("\n x=%d",x);
  int y=0x12;
  Printf("\n y=%d",y);
}
```

⇒ once you have a constant, you can't change it during the program.

O/P:

PI = 3.14159

x = 10

y = 18

$012 = (12)_8 = (10)_{10}$

$0x12 = (12)_{16} = (18)_{10}$

K & R → C | Concise & crisp

C in depth (Shrivastava) → examples cover most cases

[3.6] Input and output

Printf, Scanf are two popular function c.

↓ ↓ → formatted

O/P I/P

```
#include<stdio.h>  → Printf, Scanf are defined in stdio.h file (Standard input output)
main()
{
Initializing
  char ch = 'a';        → a
  char str[10] = "abcdef";   → a|b|c|d|e|f|\0
  float real = 12.34;
  int number = 100;
  double dbl = 12.345678;

Printf ("Character: %c\n", ch);
Printf ("Character as integer: %d\n", ch);
Printf ("String: %s\n", str);
Printf ("float: %f\n", real);
Printf ("Double: %lf\n", dbl);
Printf ("Scintific\(e\): %e\n", dbl);
Printf ("Scientific\(E\): %E\n", dbl);
Printf ("Integer: %d\n", number);
Printf ("Octal: %o\n", number);
Printf ("Hexadecimal: %x\n", number)
Printf ("Hexadecimal: %x\t String: %s\t Float: %f\n",
        number, str, real);
```

%u → unsigned integer
%g → floating number

gcc compiler include stdio.h automatically.

\n → new line character
%d → as integer so ascii (a)

%lf → long float

\t → Multiple spaces

O/P:

```
Character: a
Character as int: 97
String: abcdef
Float: 12.340000
Double: 12.345678
Scientific (e): 1.234568 e+01
Scientific (E): 1.234568 E+01
Integer: 100
Octal: 144
Hexadecimal: 64
Hexadecimal: 64  String: abcdef
Float: 12.340000
```

} doesn't add 0 0x Prefixes

[3.7] Formatting input and output

Integers:

```
scanf("%2d %3d", &a, &b);
```

① 6, 39 → a = 6, b = 39

② 12 123 → a → 12, b → 123

③ 123 1234 → a → 12, b → 3

④ 12 1234 → a → 12, b → 123

When it get space it think, I am done

%nd → where n represents no of integers, it will read

```
printf("a = %4d, b = %3d", a, b);
```

↑ Min no. → formatted printing

① a = 12, b = 14

a = − − 12, b = − 14

floating point numbers

```
scanf("%3f %4f", &x, &y);   ← input
```

① 5 5.9 → x = 5.0, y = 5.9

② 5.1 1.23 → x = 5.1, y = 1.23

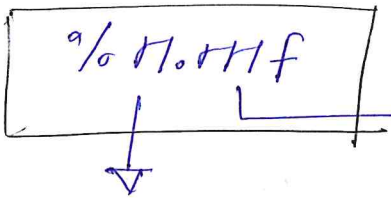
③ 1.25 4.5678 → x = 1.2, y = 3.0

Printf("x = %4.1f, y = %7.2f", x, y);

Print only 1 digit after decimal Prints

① x = 8.0 , y = 5.9 ⇒ x = _8.0 y = ___5.90

② 15.231 y = 65.12345 ⇒ x = 15.2 y = __65.12



→ no. of digits after the decimal

Min. no. of chars

String

```
char str[10];
scanf("%s", str);
scanf("%3s", str);
printf("%.3s", "abcdef");
```

① i/p - abcdef
abcdef\0

i/p : abcdef
str : abc\0 ignore def

→ abcdef

* `printf("%5s", "ab");` → ___ab

`printf("%.3s", "abcdef");` → abc

⇒ In String also, you can follow %m.nnf format

Suppression char

`scanf("%d%*d%d", &a, &b, &c)`

don't take second integer

25 30 35 a = 25 b = 35 (random/garbage

Printf scanf

⇒ getchar() and Putchar() are available for character i/o

```
char c;
Printf ("enter a char");
    c=getchar();        behaves same as scanf("%c",&c);
Printf("The value you entered");
    Putchar(c);
```

[4] Operators and Expressions

⇒ operations on integers, floats and logical

[4.1] Arithmetic operators: Integers

/* ---------- */ comment line

⇒ Whatever is written in between this is totally ignored.

⇒ Multiline comment

⇒ Comments are english sentences, which we write for our clarity

// ---------- ⇒ Single line comment

Unary operators: Unp → 1 −3 here − is unary operator

Binary operators: Bi → 2

```
int main()
{
int x=-3, y=+4;          // unary integers
Printf("%d \t %d \n", x, y);
int a=17, b=4;
Printf("Sum = %d \n", a+b);
Printf("Difference = %d \n", a-b);
Printf("Product = %d \n", a*b);
Printf("Quotient = %d \n", a/b);
Printf("Remainder = %d \n", a%b);
return 0;
}
```

-3 4

Sum = 21
Difference = 13
Product = 68
Quotient = 4
Remainder = 1

$\frac{17}{4} = 4.25$ take 4 only

[4.2] Arithmetic operations : floats

⇒ Modulo is not meaningful when you are dividing real numbers. It is meaningful when you are dividing an integer with an integer.

```
int main(){
float a=13.4, b=3.2;
Printf("sum is %f \n", a+b);
Printf("Difference = %.2f \n", a-b);
Printf("Product = %.2f \n", a*b);
Printf("a/b = %.2f \n", a/b);
```

16.60
10.2
42.88
a/b = 4.19

```
// Mixed mode arithmetic
    int x=7, y=2;
    float z=2.0;
Printf("\n 7/2 : %f", x/y); // int/int is an int
Printf ("\n 7/2 : %d", x/y);
Printf ("\n 7/2.0 : %f", x/z); // (int/float) or (float/int)
                                  is a float
return 0;
}
```

7/2 : 4.18

7/2 : 3

7/2 : 3.500,000

[4.3] Assignment operators

```
#include <stdio.h>
int main()
{
  int x,y,z,t;
  x=5;
  y=6;
  z=x+y*z;
  t=x;   t=5
  Printf("\n x: %d, y=%d, z=%d, t=%d", x,y,z,t);
    x=y=z=10; // Multiple variables with same value.
    x=x+5;   x+=5;   y*=5;   z/=2;   x%=5;
Printf("\n x: %d, y: %d, z=%d, t=%d", x, y, z, t);
  return 0;
}
```

O/P:

x=5 y=6 z=17 t=5

x=0 y=50 z=5 t=5

[4.4] Increment & Decrement operators

⇒ There are two types of operators
- → Prefix operators.
- → Postfix operators.

```
int main()
{
  int x=8;
① Printf("x=%d\t",x);
② Printf("x=%d\t",++x); // prefix increment
③ Printf("x=%d\t",x);
④ Printf("x=%d\t",--x); // prefix decrement
⑤ Printf("x=%d\t",x);
  return 0;
}
```

O/P: x=8 x=9 x=9 x=8 x=8

for postfix operators:
- ① use the variable x
- ② x = x+1

② and ④ statement as postfix then

O/P: x=8 x=8 x=9 x=9 x=8

[4.5] Solved Problem [GATE 2017]


```
#include <stdio.h>
int main ()
{
  int m=10;
  int n, n1;
  n = ++m;
  n1 = m++;
  n--;
  --n1;
  n- =n1;
  Printf ("%d", n);
  return 0;
}
```

m = ~~10~~ 11

n = ++m = ++10 = 11

n1 = m++ = 11++ = 11

m = ~~11~~ 12

n-- = 11-- = 10

--n1 = --11 = 10

n = n - n1

= 10 - 10

= 0

The output of the code is 0

[4.6] Relational operators

<, >, <=, >=, are relational operators.

```
#include <Stdio.h>
int main ()
{ int a, b;
  Scanf ("%d %d", &a, &b);        → 12, 4
  Printf("enter values for a & b:");
  if(a<b)
  Printf ("%d is less than %d\n", a, b);
```

```
if(a<=b)
Printf("%d is less than or equal to %d\n",a,b);
if(a==b)
Printf("%d is equal to %d\n",a,b);
if(a>b)
Printf("%d is greater than %d\n",a,b);
if(a>=b)
Printf("%d is greater than or equal to %d\n",a,b);
if(a!=b)
Printf("%d is not equal to %d\n",a,b);
if(a=b) // value of expression is same as the value
        of b and not 1 or 0.
Printf("Assignment operator");
Printf("\n%d",a=b);
return 0;
}
```

a=b
a=4

O/P: enter values for a & b : 12 4

12 is not equal to 4
12 is greater than 4
12 is greater than or equal to 4
4

[4.7] Logical or Boolean operators

⇒ logical or boolean operators && AND
|| OR

```
int main()
{
 int a=40, b=20;
 int c=20, d=30;
 if(a>b && a!=0)        40>20
  {
  printf("&& operators: Both conditions are true\n");
  }
 if(a>d || d!=20)
  {
 Printf("|| operator: only one condition is true\n");
 }
 if(!(a>b && a!=0)){
 Printf("! operator: Both conditions are true\n");
  }
else {
Printf("! operator: Both conditions are true, But
        status is inverted as false");
   }
 if(a&&b){ Printf("\n a&&b); }
if(a &&0){ Printf("\n a &&0); }
if(a||0){ Printf("\n a ||0); }
```

[4.8] Comma operator


```
#include <stdio.h>
int main()
{
  int a, b, c, sum;
  sum = (a=8, b=7, c=9, a+b+c);
  printf("sum = %d\n", sum);
  return 0;
}
```

O/P ⇒ 24

⇒ Works left to right

sum = (a=8, b=7, c=9, a+b+c)

a → 8
b → 7
c → 9

8 + 9 = 24

⇒ return value of right most sub-expression.

```
#include <stdio.h>
int main()
{
  int a=8, b=7, temp;
  printf("a=%d, b=%d\n", a, b);
  temp = a, a=b; b=temp;
  printf("a=%d, b=%d\n", a, b);
  return 0;
}
```

temp = 8
a = 7
b = 8

[4.9] Condition (ternary) operator

⇒ ternary means something that contains three parts.

```
#include <stdio.h>
int main()
{
  int a, b, max;
  Printf ("enter values for a and b:");
  Scanf ("%d %d", &a, &b);
  max = a > b ? a : b;
  Printf ("Larger of %d and %d is %d\n", a, b, max);
  return 0;
}
```

O/P: a → 4, b → 2

a > b ? a : b // If a is greater than b then returns a else b

4 > 2 (4)

⇒ This ternary operator can also be written using the following code.

```
if (a > b)
{
  Printf ("a is greater");
}
else {
  Printf ("b is greater");
}
```

[4.10] Solved Problem Gate 2008

Q: Which combination of the integer variables x, y and z makes the variable a get the value 4 in the following expression

```
a = (x > y) ? ((x > z) ? x : z) : ((x > z) ? y : z)
```

(A) x = 3, y = 4, z = 2
(B) x = 6, y = 5, z = 3
(C) x = 6, y = 3, z = 5
(D) x = 5, y = 4, z = 5

Solution:

a = 4

Nested ternary

```
(x > y) ? ((x > z) ? x : z) : ((x > z) ? y : z)
```

(A) 3 > 4 goto false

3 > 2 ↳ y = 4

so option (A) is correct.

[4.11] Sizeof operator

⇒ Sizeof is an unary operator. It only takes one parameter.

⇒ operates on single parameter. That's why it is called unary operator

1) Printf ("\n%d", sizeof (int)); → No. of bytes are are required to store an integer
↑ datatype

int a;

2) Printf ("\n%d", sizeof(a));
↑ variable

3) Printf ("\n%d", sizeof(2));
↑ constant

⇒ sizeof() is not a function. It is an operator which is used in C.

⇒ It is used to get no. of bytes it takes to make your code portable

[4.12] Precedence & Associativity

int a = 2*3+5 possibilities:

(2*3)+5 = 11

2*(3+5) = 16

⇒ C says multiplication must be performed before addition.

associativity: Have two operators of same precedence. (L to R)

100/10*10 → 10*10 = 100

100/100 = 1

	Operators	Associativity	Precedence
	( ) Parenthesis to function call	Left to right	Highest 14
	[ ] Array subscript		
	. Dot (Member of structure)		
	→ Arrow (Member of structure)		
	! Logical NOT	Right to left	13
	~ one's complement		
	− Unary Minus (Negation)		
	++ Increment		
	−− Decrement		
	& Address of		
	* Indirection		
	(type) Cast		
	sizeof sizeof		
M	* Multiplication	Left to right	12
	/ Division		
	% Modulo (Remainder)		
A	+ Addition	Left to right	11
	− Substraction		
S	<< Left shift	Left to right	10
	>> Right shift		
R	< Less than	Left to right	8
	<= Less than or equal to		
	> Greater than		
	>= Greater than or equal to		
E	== Equal to	Left to right	8
	!= NOT equal to		

	Operators	Associativity	Precedence
B	& Bitwise AND	Left to Right	7
	^ Bitwise XOR	"	6
	| Bitwise OR	"	5
L	&& Logical AND	"	4
	|| Logical OR	"	3
T	?: Conditional	Right to left	2
R	=, +=, *=, etc	Assignment operators	Right to left	1
C	,	comma	Left to right	Lowest 0

PUMA'S REBL TAC

⇒ Unary, conditional (ternary), and assignment operators are right to left.

* Auto drivers are always right

AUT [Assignment, Unary, ternary operators]

```
int main()
{
int a, b, c, d, e, f, g, h, k;
a=8, b=4, c=2, d=1, e=5, f=20;
Printf("%d \t", a+b-(c+d)*3%e+f/g);
```

8+4 − (2+1) * 3 % 5 + 20/g

8+4 − 3*3 % 5 + 20/g

go L to R 8+4 − 9 % 5 + 20/g

8+4 − 4 + 2 = 10

a=17, b=5, c=6, d=3, e=5

Printf("%d\t", a%6 - b/2 + (c*d-5)/e);

17%6 - 5/2 + (6*3-5)/5

17%6 - 5/2 + 13/5

= 5 - 2 + 2

= 5

a=4, b=5, c=6, d=3, e=5, f=10;

Printf("%d\t", a * b - c/d < e+f);

4*5 - 6/3 < 5+10

20 - 2 < 15

18 < 15 false 0

a=8, b=5, c=8, d=3, e=65, f=10, g=2,

h=5, k=2;

Printf("%d\t", a-b+c/d == e/f-g+h%k);

8-5+8/3 == 65/10 - 2 + 5%2

8-5+2 == 6 - 2 + 1

5 == 5 true = 1

a=8, b=3, c=2, d=3, e=2, f=11

Printf("%d\n", a-b || (a-b*c)+d && e-f%3);

return 0;

&& has higher precedence than ||

8-3 || (8-3*2)+3 && 2 - 11%3

8-3 || 2+3 && 2 - 2

5 || 5 && 0 = 5 || 0 = 1 = true

① $y = (++x) + (--x)$

C does not clearly specify which operand should be executed first.

do not write code like this

ternary

② $a > b \; ? \; a : b$

③ ( ) && ( ) && ( ) } False

① ② ③ evaluated

If this is false → Whole expression will not be evaluated at all.

④ (10 > 2) || ( ) } true

If this is true → Whole expression will not be evaluated at all

If this is false → this will be evaluated, If this is true then it will also be true

[4.13] Type Casting / conversion

float f = 2.0 + 3 ← int

float constant

① 3 → 3.0

② 2.0 + 3.0 = 5.0

③ f = 5.0

long double → double → float → long int → int → char → short int
(2) (1) (1)

⇒ Suppose long int is added to int.

= 2L + 3 → 3L
= 2L + 3L
= 5L

⇒ What if we have unsigned int

= unsigned int + long int
= (try to convert into)

Case 1:
⇒ If unsigned int fits into long int

Case 2:
⇒ If unsigned int is a large number. both of them will be converted into unsigned long int.

how type conversion happen during assignment operation

Ex:1 int $i_1 = 80.56$; We are trying to store float into integer.

Atomatic demotion

Ex:2 char $c_1 = i_1$; → ASCII value to 80 = P

Ex:3 float $f_1 = 12.6$
int $i_1 = f_1$; $i_1 \rightarrow 12$
float $f_2 = i_1$ $f_2 \rightarrow 12.0$

⇒ If you want to write reliable code, better to use explicit type cast.

⇒ It's unary operator

Ex:

```
int x=20, y=3;
float z=x/y;   // z = 20/3 → 6.0
```

} automatic type cast

⇓

```
float z = (float) x/y;
```

20.0/3 = 20.0/3.0
= 6.66

· `float z = (float)(x/y)`

division will happen first as parenthesis has higher precedence than typecast.

[4.14] Bitwise operations

⇒ Binary → 0, 1 (base 2) Decimal → 0 ... 9 (base 10)

⇒ At the end every computer uses 0, 1.

⇒ 8-bits = 1 byte

$2^7\ 2^6\ 2^5\ 2^4\ .\ 2^3\ 2^2\ 2^1\ 2^0$

bits → 0/1

$(20)_{10} = (\quad)$

20 → 16 + 4

$2^4 + 2^2$

$(000\ 10100)_2$ → (binary representation of 20)

$(10)_{10}$ = _ _ _ _ 2^3 _ 2^1 _

0 0 0 0 1 0 1 0

↓

8+2

↓

2^3+2^1

Bitwise AND (&)

a = 20 $\quad$ 20 = $(0001,0100)_2$

b = 10 $\quad$ 10 = $(0000\,1010)_2$

a&b = 0001, 0100

0000 1010

(0) = $(0000\,0000)_2$

0 → false

1 → true

Bitwise OR (|)

a | b = 0001 0100

0000 1010

0001 1110 → $(30)_{10}$

$2^4\,2^3\,2^2\,2^1\,2^0$

Bitwise XOR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

```
  0 0 0 1 0 1 0 0
  0 0 0 0 1 0 1 0
---------------------
a^b: 0 0 0 1 1 1 1 0
---------------------
```

Left shift (<<)

a << 2 shift all the bits left by 2

(00010100) << 2

01010000 = 64+16 = 80

⇒ left shift by 2-bits mean. add two 0's at the end and discard first two digits.

2 Bytes

(1100,0000 , 0001 0100) << 2

0000,0000 , 01010000

Right shift (>>)

a >> 4 shift all bits right by 4

0001 0100 >> 4

0000 0001

Control Statement

⇒ to change the order of execution.

if -- else, for loops,

Basic control statement

[6.1] If else

```
#include <stdio.h>
int main()
{
  int a, b;
  Printf("enter two numbers:");
  Scanf("%d %d", &a, &b);
   if(a>b)
    Printf("Bigger number = %d\n", a);
   else
    Printf("Bigger number = %d\n", b);
   return 0;
  }
```

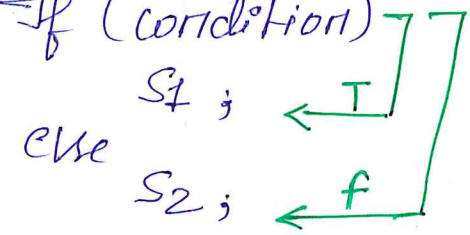
O/P → a → 12 , b → 4

12 > 4

Bigger number = 12

```
if (condition)
    S1;
```

```
if (condition)
    S1;
else
    S2;
```



⇒ We can have if without else.

Program to check entered number is negative.

```
#include <stdio.h>
{
  int num;
 Printf ("enter a number:");
  Scanf ("%d", &num);
  if (num < 0)
  {
    Printf ("Number entered is negative \n");
     num = -num;
  }
    Printf ("value of num is : %d \n", num);
    return 0;
  }
```

block

O/P:

```
entered a number: -56
Number entered is negative
value of num is : 56
```

Program to find whether a year is leap or not

Nested if else

```
#include <stdio.h>
int main()
{
  int year;
  Printf("enter year:");
  Scanf("%d", &year);
  if(year%100!=0)
  {
    if(year%4==0)
      Printf("Leap year\n");
    else
      Printf("NOT leap\n");
  }
else
  {
    if(year%400==0)
      Printf("Leap year\n");
    else
      Printf("Not leap\n");
  }
  return 0;
}
```

1900

1900 → else

1900 % 400

↓

Not leap year

2000 → else → 2000 % 400

↓

leap year

WAP to find out grade of a student when the marks of 4-subjects are given.

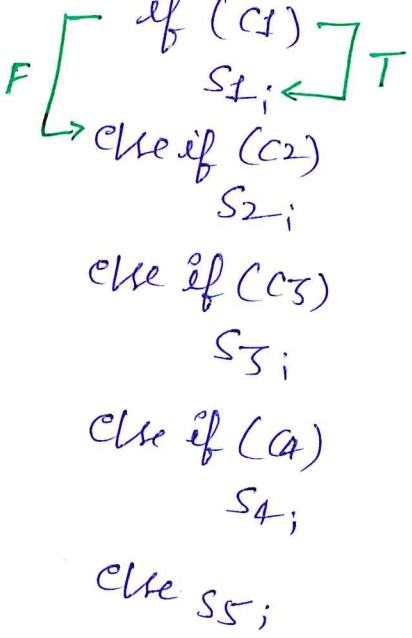
P >= 85 g = A
P < 85 and P >= 70 g = B
P < 70 and P >= 55 g = C
P < 55 and P >= 40 g = D
P < 40 g = E

```
#include <stdio.h>
int main()
{
float m1, m2, m3, m4, total, Per;
char g;
Printf("enter marks of 4 subjects:");
Scanf("%f %f %f %f", &m1, &m2, &m3, &m4);
total = m1+m2+m3+m4;
Per = total/4;
if (Per >= 85)
    g = 'A';
else if (Per >= 70)
    g = 'B';
else if (Per >= 55)
    g = 'C';
else if (Per >= 40).
    g = 'D';
else
    g = 'E';
Printf("Percentage is %f, Grade is %c\n", Per, g);
return 0;
}
```

O/P:

enter marks of 4 subjects: 45 55 65
Percentage is 60.000000, Grade is C

→ If else ladder:



[6.2] While loop

⇒ Works same as while loop & for loop.

WAP to print numbers from 1 to 10 using while loop

```
#include <stdio.h>
int main ()
{   int i=1;
  while (i<=10)
   {
    Printf ("%d\t", i);
      i=i+1;
   }
  Printf ("\n");
   return 0;
}
```

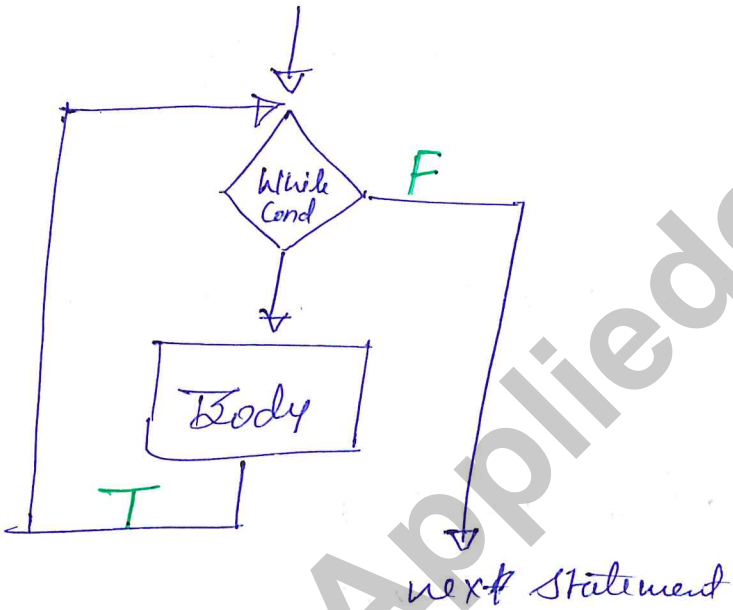
o/p: 1 2 3 4 5 6 7 8 9 10

⇒ When there is repeatation then we can use loops.

```
While (condition)
{
   S1 ;
   S2 ;
   S3 ;
   S4 ;
}
```

```
While (condition)
   S1 ;
```

Pre-statement (outside)



WAP to print the sum of digits of any number.

```
int main ()
{ int n, sum = 0, rem ;
  Printf ("enter a number:");
  Scanf ("%d", &n);
  While (n>0){
              rem = n % 10 ;
              Sum+ = rem ;
              n /= 10 ;
            }
  Printf ("Sum of digits = %d\n", Sum);
  return 0;
}
```

O/P :

enter a number : 1234
Sum of digits = 10

n = 1234

① while (1234 > 0) ✓

rem = 1234 % 10 = 4

Sum = Sum + rem = 0 + 4 = 4

n = n/10 = 1234/10 = 123

② while (123 > 0)

rem = 123 % 10 = 3

Sum = 4 + 3 = 7

n = 123/10 = 12

③ while (12 > 0)

rem = 12 % 10 = 2

Sum = 7 + 2 = 9

n = 12/10 = 1

④ while (1 > 0)

rem = 1 % 10 = 1

Sum = 9 + 1 = 10

n = 1/10 = 0

while (0 > 0) ✗

∴ Sum of digits = 10

WAP to convert a binary number into decimal number.

```
int main()
{
int n, binary, rem, d, j=1, dec=0;
Printf("enter the number in binary:");
Scanf("%d", &n);
binary = n;
while(n>0)
{
  rem = n % 10;
   d = rem * j;
  dec += d;
  j *= 2;
  n /= 10;
}
Printf("Binary number = %d, Decimal number = %d\n",
   binary, dec);
return 0; }
```

Enter the number in binary: 10101

① (10101 > 0) ✓

rem = 10101 % 10 = 1

d = 1 * 1 = 1

dec = dec + d = 0 + 1 = 1

f = f * 2 = 2

n = 10101 / 10 = 1010

② while (1010 > 0)

rem = 1010 % 10 = 0

d = 0 × 1 = 0

dec = 1 + 0 = 1

f = 2 × 2 = 4

n = 1010 / 10 = 101

③ while (101 > 0)

rem = 101 % 10 = 1

d = 1 * f = 4

dec = dec + d = 1 + 4 = 5

f = 2 × 4 = 8

n = 101 / 10 = 10

④ while (10 > 0)

rem = 10 % 10 = 0

d = 0 × 8 = 0

dec = 0 + 5 = 5

f = 2 × 8 = 16

n = 10 / 10 = 1

⑤ while (1 > 0)

rem = 1 % 101

d = 1 × 16

dec = 5 + 16 = 21

f = 2 × 16 = 32

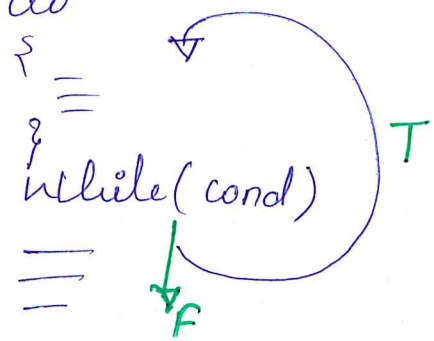
n = 1 / 10 = 0

Binary number = 10101 , Decimal number = 21

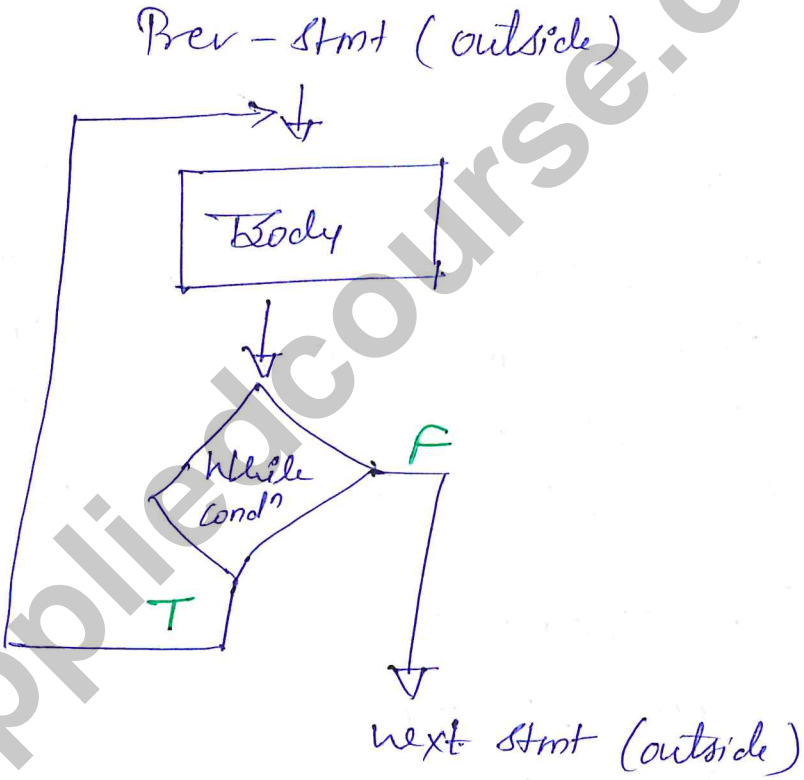
[6.3] do-while loop


```
while (cond)
{
  ==
}
```

```
do
{
  ==
}
while(cond)
==
```




```
do
{
  S1;
}
while (cond)
```



⇒ Body of do-while loop will be executed atleast once.

```
int main()
{
int i=11;
do
{
printf("%d\t",i);
   i=i+1;
}while(i<=10)
printf("\n");
return 0;
}
```

O/P: 11

[6.4] for - loop

⇒ for (exp1 ; exp2 ; exp3) { ≡ }
- exp1 → init
- exp2 → condition
- exp3 → update

⇒ In body part, either we can have one statement or multiple statements.

```
#include <stdio.h>
{
  int k;
  for(k=10; k>=2; k-=2)
      Printf("%d", k);
      Printf("\n");
      return 0;
  }
```

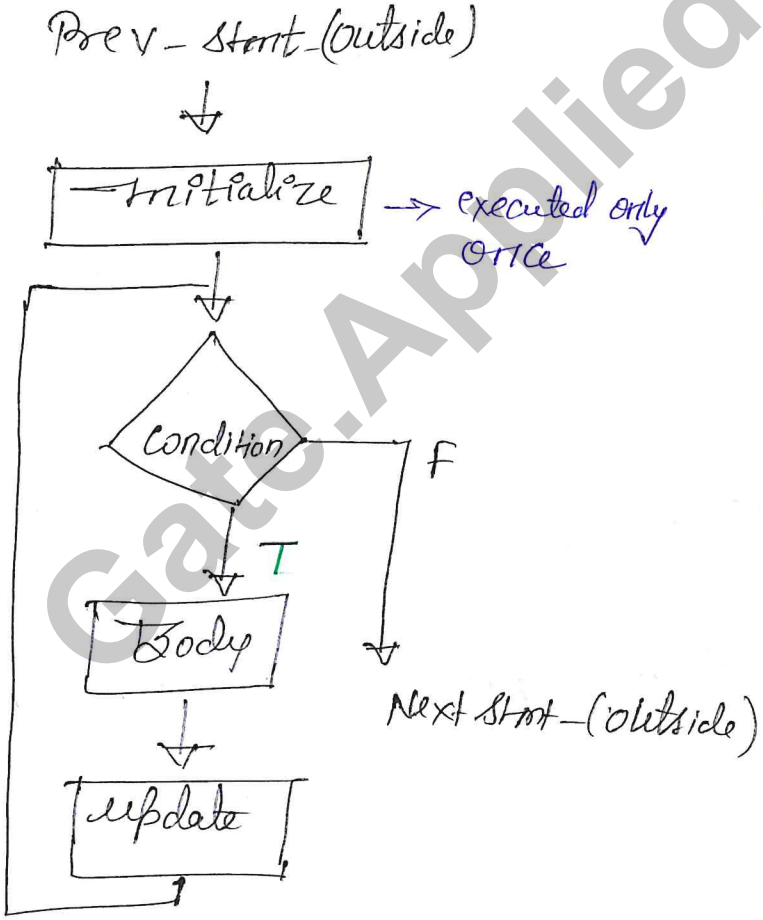
k = 10 exp1

10 >= 2 ✓ exp2

k = 10

k = k-2 = 10-2 = 8 exp3

8 >= 2	4 >= 2
k = 8	k = 4
k = 6	k = 2
6 >= 2	2 >= 2
k = 6	k = 2
k = 4	k = 0
	0 >= 0 ✗



for (init; cond; update)
{ body }
⇔
init
while (cond) {
Body
Update
}

run a loop n-times

```
int i;
for(i=0; i<n; i++){
        ≡   } n times
}
```

i=0, i=1, i=2, ... i=n-1

find the sum of this series upto n terms.

1+2+4+7+11+16+ --- ---

```
#include<stdio.h>
int main()
{
 int n, i, sum=0, term=1;
 Printf("enter number of terms: ");
 Scanf("%d", &n);
 for(i=1; i<=n; i++)
  {
   sum += term;
   term = term+i;
 Printf("\n%d", term);
  }
Printf("The sum of series upto %d terms is %d\n",
          n, sum);
 return 0;
}
```

O/P: enter number of terms: 5

Raji	1st	2nd	3rd	4th	5th
term	1	2	4	7	11

i=1 1<=5

Sum = Sum + term = 0+1 = 1

term = term + i = 1+1 = 2

Pf(term) = 2

Update i ⇒ i++ = 1++ = 2

2<=5

Sum = 1+2 = 3

term = 2+2 = 4

Pf(term) = 4

i=3

3<=5

Sum = 3+4 = 7

term = 4+3 = 7

Pf(term) = 7

i=4

4<=5

Sum = 7+7 = 14

term = 7+4 = 11

Pf(term) = 11

i=5

5<=5

Sum = 14+11 = 25

term = 11+5 = 16

Pf(term) = 16

i=6

6<=5 X

The sum of series upto 5 terms is 25.

Fiboracci number

⇒ Fiboracci sequence appears in indian mathematics in connection with Sanskrit prosody as pointed out by Parmanand Singh in 1985.

⇒ F_n denoted form of sequence.

$F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$

WAP to print fiborracci series

1, 1, 2, 3, 5, 8, 13, 34, 55, 89 .....

(In this series each number is a sum of the previous two numbers)

```
#include <stdio.h>
int main()
{
  long x, y, z;
  int i, n;
   x = 0; y = 1;
Printf("enter the number of terms:");
Scanf("%d", &n);
Printf("%ld", y);
for(i=1; i<n; i++)
{
  z = x+y
Printf("%ld", z);
   x = y;
   y = z;
}
Printf("\n");
return 0;
}
```

x = 0, y = 1

Enter the number of terms: 5

n = 5

y = 1

i = 1, 1 < 5

z = x + y = 0 + 1 = 1 ✓

x = 1, y = 1

i = 2, 2 < 5

z = 1 + 1 = 2 ✓

x = 1, y = 2

i = 3, 3 < 5

z = 1 + 2 = 3 ✓

x = 2, y = 3

i = 4, 4 < 5

z = 2 + 3 = 5 ✓

x = 3, y = 5

i = 5, 5 < 5 X

⇒ We can use comma operator with in for loop.

$\frac{i=0, j=10}{i++, j--}$ valid

```
#include <stdio.h>
int main()
{
  int i, j;
  for (i=0, j=10 ; i<=j ; i++, j--)
  Printf ("i=%d  j=%d\n", i, j);
  return 0;
}
```

i=0 , j=10	0<=10	0, 10
i=1 , j=9	1<=9	1, 9
i=2 , j=8	2<=8	2, 8
i=3 , j=7	3<=7	3, 7
i=4 , j=6	4<=6	4, 6
i=5 , j=5	5<=5	5 5
i=6, j=4	6<=4 X	Condition false

[6.5] Nested and infinite loops

⇒ If with in if-else, we have another if-else then it is called nested if-else

⇒ loops with in loop.

```
for (i=1 ; i<=5 ; i++){
    for (j=1 ; j<=3 ; j++) {
        Printf ("%d %d", i, j);
    }
}
```

Infinite loop : Common Mistakes

(1) `for(;;){`
`}`

→ I have nothing to initialize, nothing for condition and update.

keep running for infinite time

(2) `while(1){`
`}`

// always true so keep running to infinite time.

(3) `do{`
`}`
`while(1)`

Ex:1

```
int i=1;
while(i<=5);{
}
```

body of the loop is empty
Semicolon is the mistake here

Ex:2

```
for(i=0; i<=3; i--){
}
```

i = 0, -1, -2 ----
So I will be keep decreasing so loop will keep executing

Ex:3

```
while(i=2)
{
  =
  =
}
```

// assignment not comparison
so this is always true
keep running

Ex:4

```
float f=2.0;
while(f!=3.0)
{
  f+=0.2;
}
```

// code is looking perfect but in computers, float values are stored as 2.999999 or 3.000001 form
sometimes it is not stored exactly 3. instead of this you can write it f<=3.0

[6.6] break and continue

⇒ Prime no.: for a given no. 'n' if it is not divisible by 2,3,4,5 ... (n-1) then it is prime no.

n%2; n%3 - - - - - n%(n-1) // (n-2) modulo operations.

(n) ⟶ 2,3,4,5 ... sqrt(n)

n = 19

sqrt(n) = 4....

floor (sqrt(n))

⌊sqrt(n)⌋

⌊4.99⌋ = ⌊4.12⌋ = ⌊4.36⌋ = 4

WAP to find whether a number is prime or not.

```
#include <stdio.h>
#include <math.h>          // for sqrt()
int main()
{
  int i, n;
  Printf("enter a number:");
  Scanf("%d", &n);
  for (i=2; i<=sqrt(n); i++)
     if (n%i == 0)
        break;
  if (i > sqrt(n))
     Printf("%d is prime\n", n);
  else
     printf("%d is not prime\n", n);
  return 0;
}
```

// break means come out of the loop

n=19

i=2	i=3	i=4
19%2		

O/P:

enter a number: 19
19 is prime

```
for ( ) {
    for ( ){
        break;
    }
}
```

⇒ breaks only breaks out of the inner loop

⇒ Use of continue

WAP to find sum and average of 5 positive no.

```
#include<stdio.h>
int main()
{
 int i=1, n, sum=0;
 float avg;
Printf("enter 5 positive numbers:\n");
while(i<=5)
 {
 Printf("enter number %d:",i);
 Scanf("%d",&n);
  if(n<0)
  {
 Printf("enter only positive numbers\n");
   continue;
  }
 sum+=n;
   i++;
}
```

```
while(cond){
   ==
   ==
continue;
   ==
}
```

```
do{
   ==
continue;
   ==
}while(cond)
```

```
for(init; cond; update)
{
   ==
continue;
   ==
}
```

Goto [6.7]

⇒ label → Identifier

⇒ goto jump only within the function.

goto

goto label;
=
label:

forward jump

```
#include <stdio.h>
int main()
{
   int n;
Printf("enter a number:");
Scanf("%d", &n);
if(n%2==0)
    goto even;
else
    goto odd;
even:
    Printf("Number is even\n");
     goto end;
odd:
    Printf("Number is odd\n");
     goto end;
end:
   Printf("\n");
   return 0;
}
```

Please avoid goto unless it is mand

o/p when even goto end is commented.

o/p: enter a number: 1212
Number is even

enter a number: 1212
Number is even
Number is odd

(3) If you have label: & dont want to write anything then just place ; → null statement

```
for(    ){
  ===
  ===
  for(    ){
    ___
    goto endloops;
    ___
    }
    ===
  }
  ===
Endloops:
  ===
```

[6.8] Switch-Case

⇒

```
switch(expr)
{
case c1:
  ===
case c2:
  ===
case c3:
  ===
case c4:
  ===
default:
  ===
}
```

int / char / long / short

int / char / const

WAP to perform arithmetic calculations on integers.

```
#include <stdio.h>
int main()
{
  char op;
  int a, b;
  Printf("enter operator");
  Scanf("%s", &op);
Printf("enter two numbers:");
Scanf("%d %d", &a, &b);
Switch(op)
{
Case '+':
        Printf("Result = %d\n", a+b);
          break;
Case '-': Printf("Result = %d\n", a-b);
          break;
Case '*':
        Printf("Result = %d\n", a*b);
          break;
Case '/':
        Printf("Result = %d\n", a/b);
          break;
Case '%': Printf("Result = %d\n", a%b);
```

Valid

```
int i1, i2, i3;
Char c1, c2;
float f;

Switch (i2)
Switch (c1)
Switch (i1 > i2)
Switch (c1+c2-3)

Case 4:
Case 'a':
Case 2+4:
Case 'a' > 'b':
```

Invalid

```
Switch (f)  X
Switch (i1+4.5)  X
```
Should be integer only

```
Case "abc": X
```
String is not allowed

```
Case i1: X
```
Not allowed
This is integer variable not constant

```
Case i1 > i2 :  X
```
If & i2 are integer variables so this is not allowed.

[6.9] Sample Problem [Gate 2016]

The following function computes x^y for positive integers x and y.

```
int exp (int x, int y)
{ int res=1, a=x, b=y;
  While (b!=0)
  { if (b%2 == 0)
    {  a=a*a;
       b=b/2;
    }
    else {
       res = res * a;
       b = b-1;
    }
  }
  return res;
}
```

① res = 1 if * = 4 , 4 = 5

a = 4 , b = 5

(A) $x^y = a^b \Rightarrow 4^5 = 4^5$ could be correct

(B) $(1 * 4)^5 = (1 * 4)^5$ "

(C) $x^y = res * a^b \Rightarrow 4^5 = 1 * 4^5$

② IInd iteration res = 4 , a = 4 , b = 4

(A) $x^y = a^b \Rightarrow 4^5 = 4^4$ X

(B) $(4 * 4)^5 = (4 * 4)^4$ X

(C) $x^y = res * a^b \Rightarrow 4^5 = 4 * 4^5$ ✓ could be correct

(D) $x^y = (res * a)^b \Rightarrow 4^5 = (4 * 4)^4 = 4^8$ X

Ans: so option (c) is correct.

③ Iteration - 3

$a = (4^2)^2$

b = 1

res = 4

④ Iteration - 4

$res = 4 * (4^2)^2 = 4^5$

b = 0

$a = (4^2)^2$

x^y

Only 3 - Multiplicatio

$= 4 \times 4^2 \times 4^2$

[6.10] Solved Problem [GATE 2015]

Consider the following pseudo code, where x and y are positive integers

```
begin
    q := 0
    r := x
    while r >= y do
    begin
        r = r - 1
        q = q + 1
    end
end
```

$q = 0 \quad r = x$

$x - y - y - y \ldots$ (q times y is subtracted from x)

$r = x - qy$

(2) $r < y$

The post condition that needs to be satisfied after the program terminates is

(A) $\{r = qx + y \wedge r < y\}$

(C) $\{y = qx + r \wedge 0 < r < y\}$

(B) $\{x = qy + r \wedge r < y\}$

(D) $\{q + 10\}$

$r = x - qy \quad \because r < y$

$r + qy = x \quad r < y$

$x = r + qy$

$x = qy + r \quad r < y$ — option (B)

[6.11] Solved Problem [Gate 2015]

Consider the following c program:

```
#include <stdio.h>
int main()
{
  int i, j, k=0;
  j = 2 * 3/4 + 2.0/5 + 8/5;
  k = --j;
  for (i=0; i<5; i++)
  {
    switch (i+k)
    {
    Case 1:
    Case 2: Printf ("\n%d", i+k);
    Case 3: Printf ("\n%d", i+k);
    default: Printf ("\n%d", i+k);
    }
  }
return 0;
}
```

The no of times printf statement is executed is ____

(A) 8
(B) 9
(C) 10
(D) 11

k = 0

L to R

$j = 1 + 0.4 + 1$

$j = 6/4 + 2.0/5 + 8/5$

$= 1 + 0.4 + 1 = 2$ (typecasted to int)

$j = k - 2 = 1$

$k = k - 1 = 1$

for loop

i = 0

$i+k = 0 - 1 = -1$

default $i+k = 0 - 1 = -1$ — (1)

i = 1

$i+k = 1 - 1 = 0$

default: $i+k = 1 - 1 = 0$ — (1)

i = 2

$i+k = 2 - 1 = 1$

Case 1 — (3)

i = 3 $i+k = 3 - 1 = 2$ — (3)

Case 2

i = 4, $i+k = 4 - 1 = 3$ — (2)

i = 5

X

[6.12] Solved Problem [Gate 2014]

Consider the following Pseudo code. What is the total number of multiplication to be performed

```
D=2
for i=1 to n do
    for j=i to n do
        for k=j+1 to n do
            D=D+3
```

(A) Half of the product of 3-consecutive integers.
(B) One-third of the product of 3 consecutive integers.
(C) One-sixth of the product of 3-consecutive integers.
(D) None of the above.

Solution: at $i=1$

$j=1 \longrightarrow k=2,3\ldots n \longrightarrow (n-1)$ times
$j=2 \longrightarrow k=3,4\ldots n \longrightarrow (n-2)$ times
$\vdots$
$j=n \longrightarrow 0$

Sum of first n-integers $\Rightarrow n(n+1)/2$

$i=2$

$j=2 \longrightarrow k=3\ldots n$
$j=3 \longrightarrow k=4,5\ldots n$

$\Big\} (n-2)(n+1)/2$

$$S = \frac{n(n-1)}{2} + \frac{(n+1)(n-2)}{2} + \frac{(n-2)(n-3)}{2} + \cdots$$

$$2S = \{n^2 - n\} + \{(n-1)^2 - (n-1)\} + \{(n-2)^2 - (n-2)\}$$
$$+ \text{ - - - - - - -}$$

$$= \{n^2 + (n-1)^2 + (n-2)^2 + \cdots 1\} - \{n + (n-1) + (n-2) + \cdots\}$$

$$= \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$

$$2S = \frac{n(n+1)(2n+1) - 3n(n+1)}{6}$$

$$S = \frac{n(n-1)(n+1)}{6}$$

S is the $\frac{1}{6}$(product of 3 consecutive integers)

Solved Problem [Gate 2012]

What will be o/p of the following C program segment.

```
Char inchar = 'A'
Switch (inchar)
{
Case 'A' :
      Printf ("choice A \n");
Case 'B':
      Printf ("choice B \n");
Case 'C':
Case 'D':
Case 'E':
default:
Printf ("No choice");
}
```

(A) No choice
(B) Choice A
(C) Choice A
Choice B No Choice
(D) Program gives no output as it is erroneous.

Solution:

```
Switch (inchar)
        ⇓
Switch ('A')
        ⇓
Case 'A'
Case B
Case C:
Case D:
Case E:
default:
```

Choice A
Choice B — As there is no break
No Choice

[6.14] Solved Problem [GATE2004]

Consider the following C program (Euclid Algo for GCD)

```
main()
{
  int x, y, m, n;
  scanf("%d %d", &x, &y);
  m = x;
  n = y;
  while (m != n)
  {
    if (m > n)
      m = m - n;
    else
      n = n - m;
  }
  printf("%d", n);
}
```

The Program computes

(A) x+y using repeated subsn
(B) x mod y "
(C) the gcd of x & y
(D) LCM of x & y

Solution: Pick x & y

Input $x=6, y=3$

(A) $x \div y = 2$

(B) $x \bmod y = 0$

(C) $GCD(x,y) = 3$

(D) $LCM(x,y) = 6$

If $x=18, y=12$

(A) $18 \div 12 = 1$

(B) $18 \bmod 12 = 6$

(C) $GCD(18,12) = 6$

(D) $LCM(18,12) = 36$

$m=6, n=3$

$6! = 3$ ✓

if $(6>3)$ ✓

$m = 6-3 = 3$

$n = 3$

$3! = 3$ ✗

printf(n)

3

So ans is 3

[6.15] Solved Problem [Gate 2014]

Consider the function func shown below:

```
int func(int num)
{ int count = 0;
  while (num)
  {
    count++;
    num >>= 1;
  }
  return (count);
}
```

(A) 8
(B) 9
(C) 10
(D) 11

The value returned by func(435) is ___

func(435)

⇓

count=0

while(435)

⇓

count++;

435>>=1;

$2^9\ 2^8\ 2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$(1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1)_2$

Shifting 1-digit right

1 1 0 1 1 0 0 1 1

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ 9

Ans is 9

No. of times we need to shift so number will become zero.

---

7. Functions

⇒ functions in C

[7.1] Introduction to Functions

isPrime(int num)

Why do we need function: functions basically allow us to break up & organise code to logical & modular format.

Printf, Scanf, Sqrt → libraries & header file

Adv: to reuse

```
main()
{
    ≡
}
```

There are broadly two types of functions:

→ library functions → user-defined function

↓

Printf, Scanf

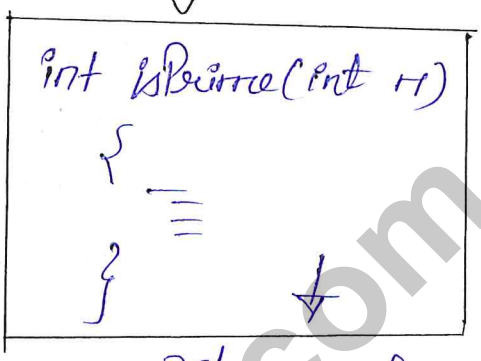
Sqrt

↓

#include <math.h>

#include <stdio.h>

Preprocessor directive



returns 1 if n is prime and 0 if n is not prime.

header file & library file

declare definition

WAP to find the sqrt of a number.

```
#include <stdio.h>
#include <math.h>          // declaration
int main ()
{
  double n, s;
  Printf ("enter a num:");
  Scanf ("%lf", &n);
  s = sqrt(n);
  Printf ("sqrt of %.2lf is %.2lf \n", n, s);
  return 0;
}
```

double Sqrt (double)

O/P: enter a num: 17

sqrt of 17.00 is 4.12

#WAP to find the sum of two numbers.

```
#include <stdio.h>
int main() → int sum(int x, int y); // fn declaration
{
  int a,b,S;
  Printf("enter values for a and b:");   Calling fn
  Scanf("%d %d", &a &b);
  S = Sum(a,b);      // Calling function sum
  Printf("Sum of %d and %d is %d\n",a,b,S);
  return 0;
}
int Sum(int x, int y) // function definition
{
   int s;
   s = x+y;
   return s;
}
```

Called fn

main → a, b, S
↓ ↓
45 34

Sum(45, 34)
x y

S = 45+34 = 79

Syntax:

identifier

return-type fn name (type1 arg1, type2 arg2)
{
  local variables declaration;
  ═══
  ═══
  return (expr);   // return is also optional
}   can be returned w/o parantheris

body

void - not going to return anything
int - default (optional)

⇒ If I define calling func before called fn then I need to declare called function.

⇒ If I define called fn above the calling fn then I need not to declare called function.

```
#include <stdio.h>
int sum (int x, int y)
{ =
}
int main(){
  ==
  ==
} sum();
```

called funcn

calling funcn

⇒ function declaration is optional not mandatory.

⇒ As soon as the func returns to the calling function, all the variables (arguments, local variable) will be removed from the memory and you can not access it.

① We can not define one function with in another function.

```
int main()
{
  int sum(){
          }
}
```

X Not allowed

(2) You can have different functions defined & declared in various files.

(3) only one variable can be returned.

return a, b; ✗ → Invalid

WAP to understand the use of return statement.

```
#include <stdio.h>
void selection (int age, float ht); //fn d
int main()
{
    int age;
    float ht;
    Printf("enter age and height:");
    scanf("%d %f", &age, &ht);
    Selection (age, ht);
    return 0;
}
void selection (int age, float ht)
{
    if (age > 20)
    {
        Printf("Age should be less than 20 \n");
        return;
    }
    if (ht < 5)
    {
        Printf("Height should be more than 5 \n");
        return;
    }
    Printf("selected \n");
}
```

[7.2] Let's dive into function

⇒ actual arguments vs formal arguments

```
int x = 10;
f(x);
```

↑ actual arguments

→ variable, const, expr, fn-call

```
int f(int a)
{
}
```

≡ ↑ formal/dummy argument

f(x); f(2); f(2+3+x);

⇒ There are four types of functions:

1. no args, no return
2. args, no return } void
3. no args, returns
4. args, returns

Rule

1. Can declare a function with in another function
2. Declaration → if a func^n returns int/char we need not declare it. Compiler assumes return type of int

③ Always declare function (good habit)

④ no-args → int f() ← declare
int f(void) ← explicit way of telling

no-return → void f(void);

⑤ Actual arguments & formal arguments:

ⓐ num of actual args > num of formal args

int f(int a); → f(2,3)
↓ ignored

ⓑ # formal args > # actual args

int f(int a, int b); | f(2);
↓ garbage

ⓒ What if there is data type mismatch

double sqrt(double d); | sqrt(3);
↑ ↑ int

C compiler will try to force type cast
(It will try and do implicit type-casting)
int → double ✓

If type-casting doesn't work then it will give some garbage value.

int f(int a);

f(2.125)

↓ double / float

try to type-cast

int f(float x)

f("abc")

no way to type-cast a string into float

Sqrt((double) 3)

6 int m=3, k;

k = add(m, m++)

3, 3 → 6

4, 3 → 7

{ order of evaluation of argument

Not defined in C

Compiler dependent

(Please avoid code like this)

7 main()

return → int

↳ 0 ⇒ terminated successfully

Non-zero ⇒ errors

⇒ OS calls the main funcn & returns goes back to

declaration → C-compiler

definition → programmer / user

return 0;

exit(0);

args (command line args)

⑧ Library function: Printf, Scanf, sqrt

Call — Programmer

declared → header-files

Definition → Libraries → obj code

→ Source code

Mathlib.c → I will write

↓

C code

↓

compile

↓

obj code

⑨ older versions → supported by modern compiler

Declaration:

ret type fn name ( ) ;

└ Argument list empty

⇒ Compiler needs return types & function name mostly.

```
ret-type fn-name (a,b,f)
{ int a,b;
  float f;
  {
  }
}
```

} gcc

7.3] Local, Global and Static variables

Global variables are accessible everywhere

```
#include <stdio.h>
void func1(void)
void func2(void)
int a,b=6;          // outside, global-var   b=6
int main()          fn                       a=0
{
Printf("Inside main(): a=%d, b=%d\n",a,b);
 func1(); // control goes to func1() body      0  6
 func2(); // after ending func1() when it come back then
 return 0;      func2() is called.
}
void func1(void)
{
print("Inside func1(): a=%d, b=%d\n",a,b);
}                                    0  6
void func2(void)
{
  int a=8; // local variable declared & initialized
 Printf("Inside func2(): a=%d, b=%d\n",a,b);
 }                                   8  6
```

O/P:

```
Inside main(): a=0, b=6
Inside func1(): a=0, b=6
Inside func2(): a=8, b=6
```

WAP to understand the use static variable

```
#include <stdio.h>
void func();
int main()
{
  func();
  func();
  func();
  return 0;
}
void func(void)
{
  int a=10;  // local variable
  static int b=10; // static local variable
Printf("a=%d, b=%d\n", a, b);
  a++;
  b++;
}
```

initialize with 0

main()
↓
func()
↓
a=10
Static b=10

Pf = [a=10, b=10]
a++; — 11
b++; — 11

func()
↓
a=10
Static b=11

Pf [a=10, b=11]
a++; 11
b++; 12

func()
↓
a=10
Static b=12

Pf [a=10, b=12]
a++=11
b++=13

[7.4] Recursion: the basics

Real World

$n! = n \times (n-1) \times (n-2) \times \cdots 1$ (repeatation)

$5! = 5 \times 4 \times 3 \times 2 \times 1$

↓ loops

↓ Iterative solution

```
int factorial (int n){
    int i, ret = 1;
for (i=1; i<=n; i++)
  {
    ret = ret + i;
  }
  return ret;
}
```

(Iterative)

→ if (n==0) return 1;

Recursive : simple way of solving large problem.

$n! = n * (n-1)!$

↓

$(n-1) * (n-2) * (n-3) \cdots 1$

factorial (n) = n * factorial (n-1) if $n \neq 0$ → recursive case

1 if $n = 0$ → base case or termination case

factorial (3) = 3 * factorial (2)

↓

2 * factorial (1)

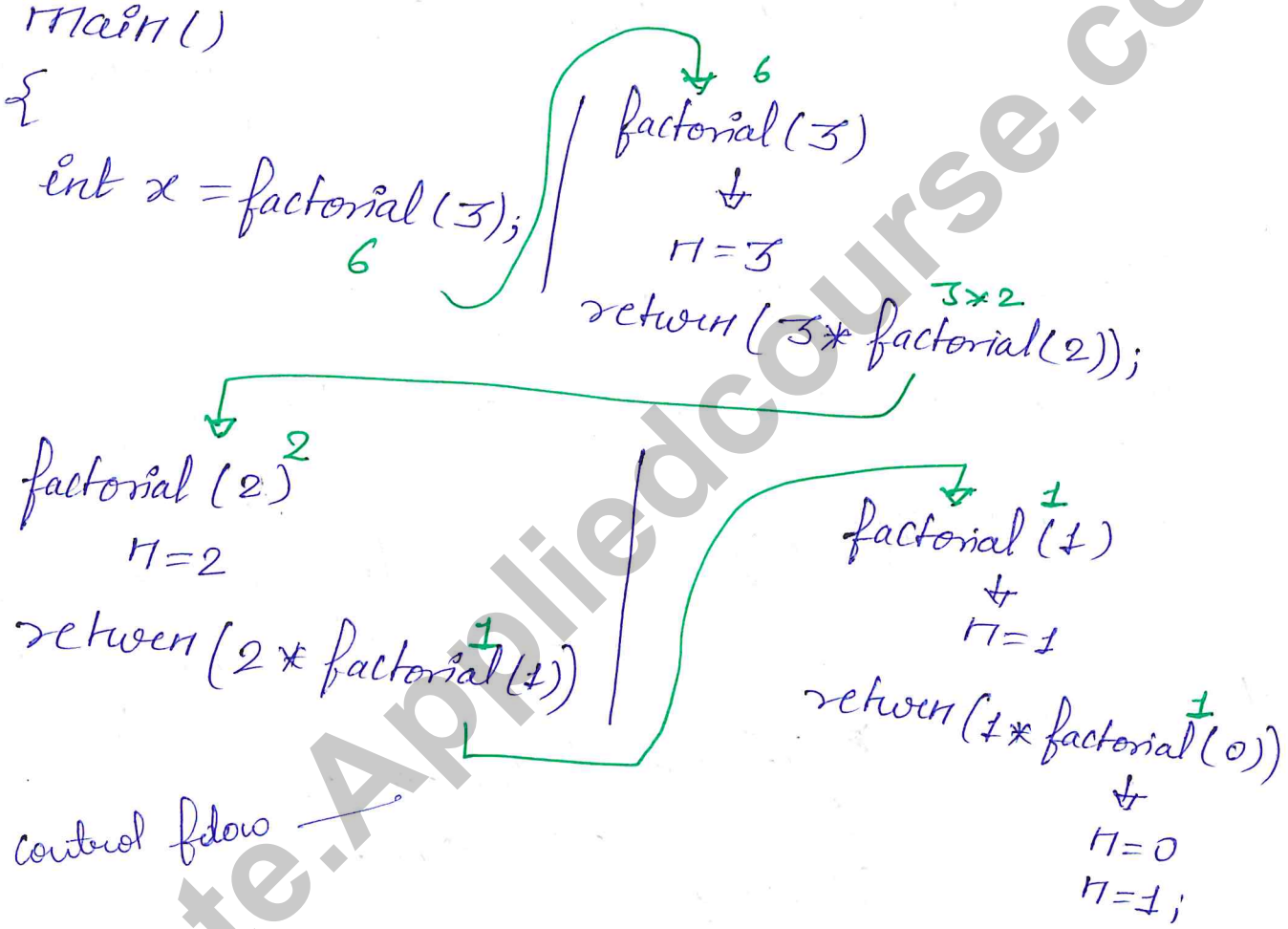
↓

1 * factorial (0) → 1

```
int factorial (int n)
{
    if (n==0)
        return 0;
    else
        return (n * factorial(n-1));
}
```

{ winding & unwinding }

```
main()
{
    int x = factorial(3);
}
```



Fibonacci numbers :

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

$fib(n) = fib(n-1) + fib(n-2)$ — recursive case if $n \geq 2$

0 if $n == 0$
1 if $n == 1$
} base / termination case

```
int fib(int n)
{
    if (n==0)
        return 0;
else if (n==1)
        return 1;
    else
        return (fib(n-1)+fib(n-2));
}
```

→ Code is trivial if we have the recursive definition

- Simplifying the code

(1) recursion ⟶ Iteration

(2) recursion → More function calls
↳ less-efficiency

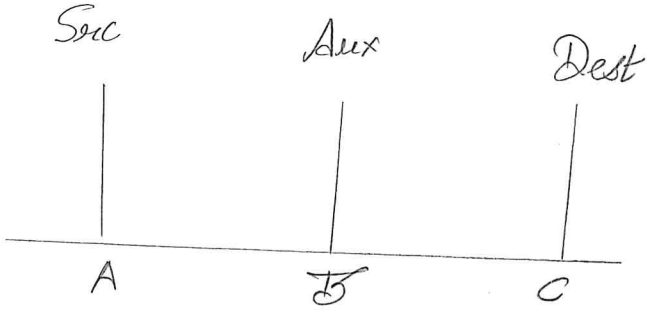
{ DS & algo }
↓
how recursion works internally :⟶ stack
↓
Call Stack

(3) tail recursion → objective of tail recursion is to make recursion more efficient

[7.5] Recursion: Tower of Hanoi

⇒ Tower of Hanoi is the mathematical game or Puzzle

⇒ Good example of recursion.



Smaller disk on larger disk

⇒ Indian temple Kashi Vishwanath, Surrounded by 64 golden disks. $2^{64} - 1 = 585$

⇒ Monastery in Hanoi, Vietnam.

for 1 disk: for moving from one tower to other.
$n = 1$ one Move.

for 2-disks: for moving from one src tower to dest tower
$n = 2$
Move smaller disk to Aux
Move larger disk to Dest.
Move smaller disk from Aux to dest

Breaking a larger problem into smaller size

Recursion ← (Pattern) → recursion equation → code

3-disk : 7 Moves

$(n, \underset{Src}{A}, \underset{dest}{C}, \underset{aux}{B})$

$\downarrow$

$(n-1), A, B, C \longrightarrow 1$

$+ n$th disk : $A \to C \longrightarrow 2$

$+ (n-1)$ disk : $B, C, A \longrightarrow 3$

↳ Multiple

$$toh(n, Src, dest, aux) = \begin{cases} \overset{n\text{-th disk}}{Src \to desc}, & n = 1 \\ toh(n-1, Src, aux, dest) \\ + n\text{th disk} : Src \to dest \\ + toh(n-1, aux, dest, Src) \end{cases}$$

↓ recursive case

```
#include <stdio.h>
void toh (int n, char src, char dest, char aux)
{
  if (n==1){
  Printf ("\n move disk 1 from %c to %c", src, dest);
   return ;
   }                                          // base case
  toh (n-1, src, aux, dest);
  Printf ("\n move disk %d from %c to %c", n, src, dest);
  toh (n-1, aux, dest, src);
  }
  int main ()
  { int n=4;
  toh (n, 'A', 'C', 'B');
   return 0;
  }
```

recursive

[7.6] Storage classes : auto, extern, static, register

```
#include <stdio.h>
void func ()
{ int x=2, y=5; // auto int x=2, y=5;
Printf ("x=%d, y=%d\n", x, y);
        x++;  y++;
}
int main ()
{
  func();
  func();
  func();
```

keyword

automatically mlr is created as i enter into the program

When I came out of func", memory is deleted

o/p: x = 2, y = 5
x = 2, y = 5
x = 2, y = 5

⇒ Any variable which is declared inside { } auto by default.

⇒ Lifetime of a variable is time between creation and destruction of a variable.

⇒ Scope of a variable is where can a variable be accessed.

Lifetime: block in which they are present and after they are declared

Scope: Can only access variables after they are declared & can be used with in this block only.

Init: initialize to garbage if you don't explicitly initialized.

default (auto) int x;

```
#include<stdio.h>
int main(void){
  int i=10;
  { int i=5;
    Printf("\n inside block: %d", i);
  }
  Printf("\n outside block: %d", i);
  return 0;
}
```

block

o/p:

inside block 5
outside block: 10

Extern : external storage classes
keyword

⇒ If you have variables that are used by many functions or across multiple files

```
int x=8; // definition
main(){
  ==
}
f1(){
  ==
}
f2(){
  ==
}
```

```
main(){
  extern int x; // then x will b
                // accessible in mai
}
f1(){
  extern int x;
}
int x=8; // defi
f2()
{
}
```

Not accessibl here

Global var

accessible here

⇒ extern int x; says this variable is present and it is created elsewhere.

file1.c

```
int x=8; // definition
main(){
}
f1(){
}
```

file2.c → a var of type int is created elsewhere

```
extern int x; // declaration
f2(){
}
f3(){
}
```

(accessible in f2 & f3)

① extern keyword is helping to increase the scope or region of the code where I can access this variable.

② extern → init to zero (by default).

Static : initialized to zero (by default)

```
void main()
{
 static int x=2, y=5; // local static variables
 Printf("x=%d, y=%d\n", x, y);
     x++;  y++;
}
int main()
{
 func();
 func();
 func();
 return 0;
}
```

Can not use x here. (out of scope)

RAM / MM

x → 2	3	4	5
y = 5	6	7	8
①		②	③

func()
x=2, y=5
Pf (2, 5)
2++; 3
5++; 6 } ①

func()
x=3 y=6
Pf (3, 6)
3++ 4
6++ 7 } ②

func()
x=4, y=7
Pf (4, 7)
4++; =5
7++ = 8 ③

As at the end of function x & y would have been removed from memory but here we are using static so they will not be removed from memory.

Lifetime of a static variable : from the time function first called till the time Program ends.

Scope of static variables : Region of code in which I can access the

⇒ I can initialize static variables as constants or as constant expressions.

```
static int x = 2;
             (2+3);
```

```
static int x = 9;   ✗
```

Can not initialize static variable using another variable

⇒ And also initialization of static variable is happend only when first time they called. And second time if we call they are already present in m/r they are not initialized and no memory is created.

Global static variables: which are called out of all functions

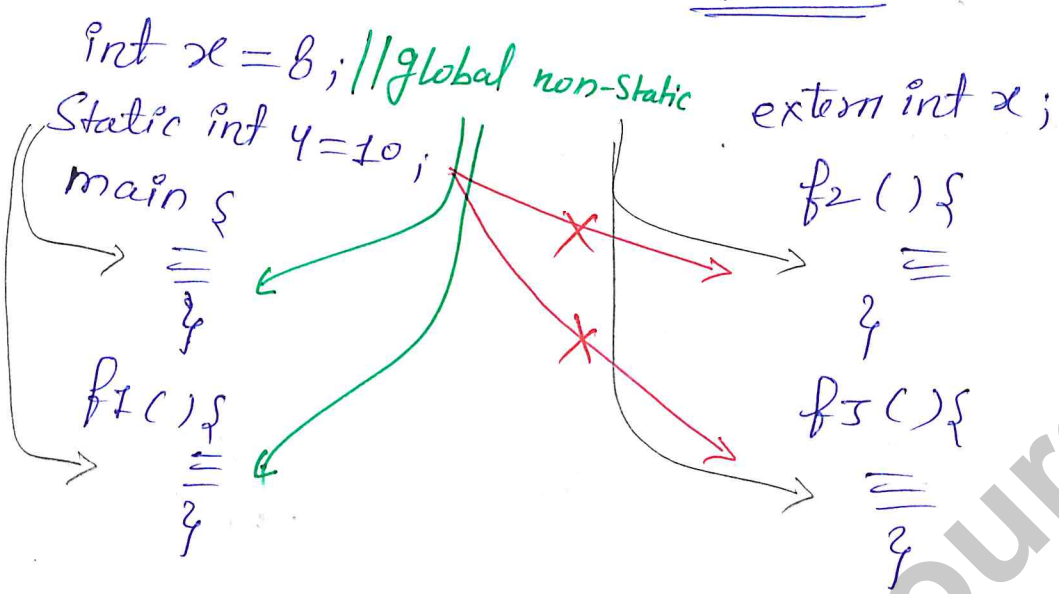
```
#include <stdio.h>
static int y = 5;
void func()
{
  int x = 2;
  Printf("x = %d, y = %d \n", x, y);
  x++;  y++;
}
int main()
{
  func();
  func();
  func();
  return 0;
}
```

O/P: x = 2, y = 5
x = 2, y = 6
x = 2, y = 7

use of global static variable:

file1.c

file2.c



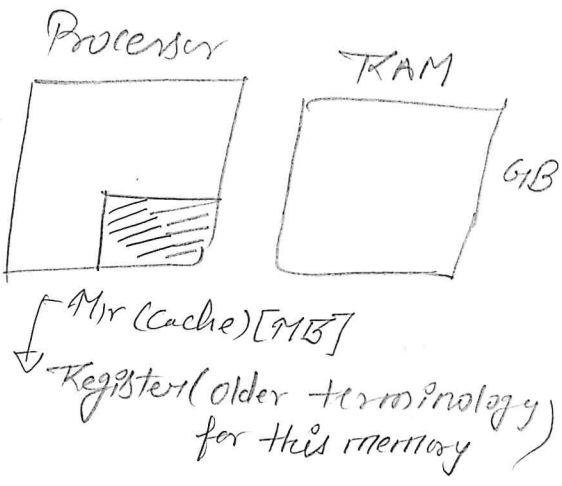
Not allowed y in f2() & f3()

⇒ scope of static variables is limited within the file.

⇒ Lifetime: Throughout the program.

Register : Keyword in 'c'. Cache / register processor

```
#include <stdio.h>
int main()
{
register int i;
for (i=0; i<100; i++)
printf("%d\n", i);
return 0;
}
```



Storage classes functions ← extern [default storage class for functions]
static

file1.c

```
main(){ }

f1(){ }
```

file2.c

```
extern void f1(); // Not mandate good program habit
f2(){ }

static f3(){ }
```

static → static global variable

[7.5] Solved Problems: 1 to 5 [Gate 2019, 2019, 2018, 2018, 2018]

Consider the following C program:

```
#include <stdio.h>
int jumble(int x, int y){
    x = 2*x + y;
    return x;
}
int main(){
    int x = 2, y = 5;
    y = jumble(y, x);   -12
    x = jumble(y, x);   -26
    printf("%d\n", x);
    return 0;
}
```

Start with main()

x = 2, y = 5

y = jumble(5, 2)
x y

x = 2*5 + 2
= 10 + 2 = 12

x = jumble(12, 2)
x y

x = 2*12 + 2 = 26

The value printed by the program is 26

17

Consider the following C function. recursive

```
void convert (int n) {
  if (n < 0)
    Printf ("%d", n);
  else {
    convert (n/2);
    Printf ("%d", n %2);
  }
}
```

C(10)
10<0 ×
C(5)
5<0 ×
C(2)
2<0
C(1)
1<0 ×
C(0)
C(0)
C(0)
C(0)
:
keep calling

Which one of the following will happen when the function convert is called with any positive integer n as argument.

(A) It will print the binary representation of n & terminate

(B) It will print the binary representation of n in the reverse order and terminate.

(C) It will print the binary representation of n but will not terminate.

(D) It will not print anything and will not terminate

DS-Algo → Call-Stack → Memory is full

Consider the following C program:

```
#include <stdio.h>
int Counter = 0;
int calc (int a, int b){
    int c;
  Counter++;
  if (b == 3)
    return (a*a*a);
  else {
    c = calc (a, b/3);
    return (c*c*c);
  }
}
int main (){
  Calc (4, 81);
Printf ("%d", Counter);
}
```

The output of this program is 4

Starting from main()

↓

calc (4, 81);

↓

a = 4, b = 81

Counter++; 1

if (81 == 3) ×

C = calc (a, 27)

↓

a = 4, b = 27

counter++; 2

if (27 == 3) ×

C = calc (3, 9)

↓

a = 3, b = 9

Counter++; 3

if (9 == 3) ×

C = calc (4, 3)

Counter++; 4

if (3 == 3) ✓

return (4*4*4) = 64

Consider the following C code. Assume that unsigned long int type length is 64 bits

```
unsigned long int fun (unsigned long int n){
unsigned long int i, p = 0, sum = 0;
for (i = n; i > 1; i = i/2) p++;
for (; p > 1; p = p/2) sum++;
  return sum;
}
```

The value returned when we call fun with i/p 2^{40} is

fun(2^{40})

i , $j=0$, Sum = 0

for ($i = 2^{40}$; $i > 1$;) $j = j++ = 1$

for (i $j > 1$; $j = j/2$) Sum++ → $j = 40$

$j = 40 \rightarrow$ Sum = 1

$j = 20 \rightarrow$ Sum = 2

$j = 10 \rightarrow$ Sum = 3

$j = 5 \rightarrow$ Sum = 4

$j = 2 \rightarrow$ Sum = 5

$j = 1 \rightarrow$ Sum = 6

$i = 2^{40} \rightarrow j = 1$

$i = 2^{39} \rightarrow j = 2$

$i = 2^{38} \rightarrow j = 3$

!

$i = 1 \rightarrow j = 40$

$i = 2^0 = 1$

$1 > 1$ loop termin

Consider the following program written in Pseudo code. Assume that x & y are integeres.

```
Count (x, y) {
if (y != 1) {
  if (x != 1) {
    Printf ("*");
    Count (x/2, y);
  }
  else {y = y - 1;
    Count (1024, y);
  }
}
```

C (1024, 1024)

(1024 != 1) ✗

(1024 != 1) ✗

Print (*) — ①

C (512, 1024)

* — ②

C (256, 1024)

* — ③

C (128, 1024)

* — ④

!

!

——— 10 times

C (1, 1024)

Now else part

The no. of times that the printf Statement is executed by the call Count (1024, 1024) is ___

```
if(y!=1) ——— C(1,1024)
  if(x!=1) x
  else
    y = y-1 = 1023
      ↓
    C(1024, 1023)
      ↓ 10
    C(1, 1023)
```

```
C(1,1023)
  ↓ y = 1022
C(1024, 1022)
  ↓ 10+
C(1, 1022)
  ↓
```

$\Rightarrow$ 10 → 1024
10 → 1023
10 → 1022
⋮
10 → 2

} $1023 \times 10 = 10230$

Chapter $\Rightarrow$ 9 Pointers

$\Rightarrow$ one of the most powerful concept in C

[9.1] Introduction to pointer

Pointers make programmer super powerful

how the MM or RAM looks

$4GB \rightarrow 4 \times 2^{32}$ Bytes

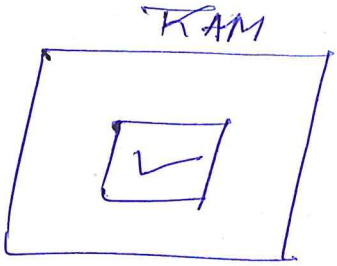
address - 0 to $4 * 2^{30} - 1$

$1\ KB = 2^{10}\ B = 1000$
$1\ MB = 2^{20}\ B = 1$ Million
$1\ GB = 2^{30}\ B = 1$ Billion
$1\ TB = 2^{40}\ B = 1$ Trillion

01

When you give me address, I can directly access it in RAM

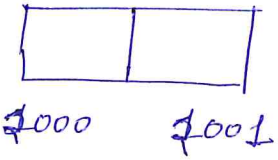
⇒ Sequential access



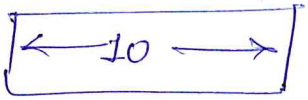
int i; // variable declaration

2B

Compiler allocated some memory in RAM



i = 10;



Stored in binary

Address operators : There are two operators :

Ampersand &

*

Ampersand (&) : Unary operator

Printf ("In %u", &i);

↓
% p (Pointer) ⟶ address of variable i

⇒ addresses are whole numbers.

⇒ Pointer stores address of variable.

&i; address of variable ✓

&123; ✗

Const

Pointer variable:

```
int *Pint;
```
→ pointer variable of integer typ

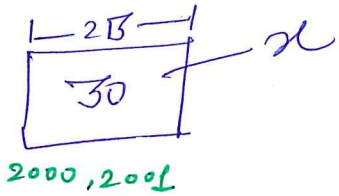
```
float *Pf;
```
pointer variable called Pf which can point to variables float.

⇒ Pint, Pf are also variables, so there is some m/r is allocated to store these variable also. Generally it is 2B, but it is computer specific.

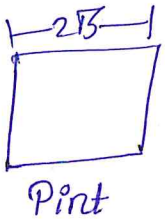
⇒
```
int i, j, *Pint;
```

Assignment of pointer variables

```
int *Pint, x=30;
Pint = &x; // assignment
```
2000

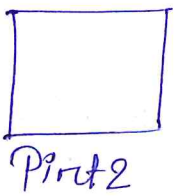


(assume)




```
float f=1.0;
Pint = &f;
```
→ Not compile error
→ Wrong output

② `int *Pint2 = Pint;`



→ I can assign pointer to another pointer.

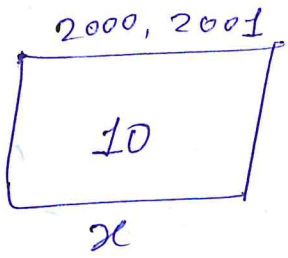
③ `Pint = NULL` → Zero = 0
↑ Const present in stdio.h

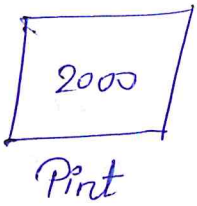
0 is boundary case which says not point to any thing.

[9.27] Dereferencing of pointers

*: Indirection operator (value stored at) : Unary

```
int x = 10;
int *Pint = &x;
```






```
*Pint = 9;
```

Value stored at this address location } value stored at 2000 = 9

x = 9

② (*Pint)++; ⇒ x++;

③ Printf("\n%d", *Pint); → 10

④ Scanf("%d", Pint);

↓

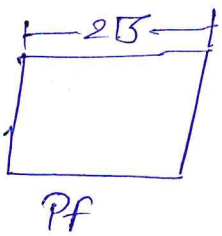
&x

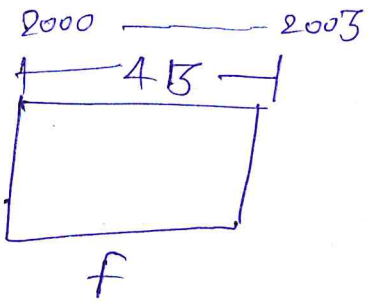
⑤ *(&x) ≡ x

*(2000) value that is stored here

Syntax : data type *Pname

```
float *Pfnt;
float f = 1.0;
Pf = &f;
Printf("\n%f", *Pf);
```





→ Indirection operation

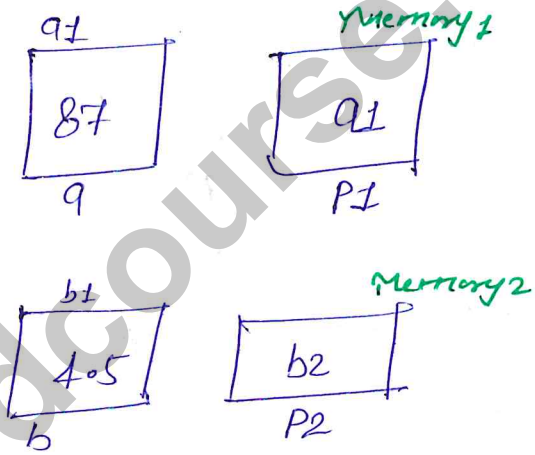
```
int *Pint, i=10;        // Pint itself contains 2000
Pint = &i;

sizeof(*Ptr);   // sizeof (int)
sizeof (Ptr);   // sizeof
```

↓

4B

```
#include <stdio.h>
 int main ()
{
  int a=87;
 float b=4.5;
  int *P1 =&a;
 float *P2 = &b;
Printf ("value of P1 = Address of a = %P\n", P1);
Printf ("value of P2 = Address of b = %P\n", P2);
Printf ("value of P1 = %P\n", &P1);
Printf ("address of P2 = %P\n", &P2);
Printf ("value of a = %d %d %d\n", a, *P1, *(&a));
Printf ("value of b = %f %f %f\n", b, *P2, *(&b));
 return 0;
}
```



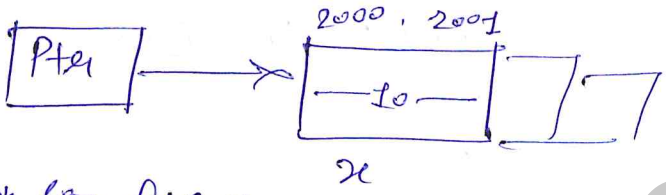
%P → using this, we get hexadecimal value

[9.3] Let's dive into pointers

Pointer Arithmetic

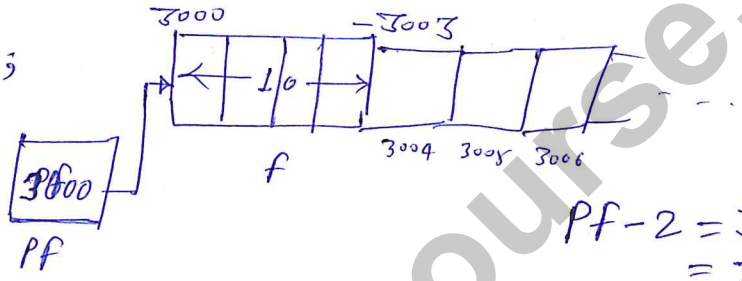
(Memory is contiguous)

```
int x = 10;
int *Ptr = &x;
```



Ptr + 1 ⇒ 2000 + 1 * sizeof(int) = 2002

```
float f = 1.0;
float *Pf = &f;
```



Pf - 2 = 3000 - 2(4)
= 3000 - 8
= 2992

Pf + 1 ⇒ 3000 + 1 * (sizeof(float)); = 3000 + 4
= 3004

valid

P + 2; ✓

P - 1; ✓

P++; ++P; ✓

P--; --P; ✓

P1 - P2; ✓

Ptr - 2;
= 2000 - 2 * (sizeof int)
= 2000 - 2 * (2)
= 2000 - 4
= 1996

Not valid

P1 + P2; ✗

P1 * P2; ✗

P1 / P2; ✗

P * 2; ✗

P / 2;

P + float or double

P * "

P / "

} ✗

Pointer Arithmetic

```
#include <stdio.h>
int main()
{
  int a = 5, *Pi = &a;
  Char b = 'x', *Pc = &b;
  float c = 5.5, *Pf = &c;
  Printf("value of Pi = Address of a = %P\n", Pi);
  Printf("value of Pc = Address of b = %P\n", Pc);
  Printf("value of Pf = Address of c = %P\n", Pf);
    Pi++;
    Pc++;
    Pf++;
  Printf("Now value of Pi = %P\n", Pi);
  Printf("Now value of Pc = %P\n", Pc);
  Printf("Now value of Pf = %P\n", Pf);
```

Precedence

++, --, * (indirection), & (address)

Unary → same precedence

(R to L) associativity

```
int x, y;
x = 10;
y = 12;
int *Ptr = &x;
```

2000, 2001, 2002, 2003

10	12
x	y

2000
Ptr

$x = *Ptr++$ Right to Left

↓ same

$*(Ptr++)$

$x = *Ptr = 10$

$Ptr = Ptr + 1 = 2002$

② $x = *++Ptr;$ ⇒ $*(++Ptr)$

$Ptr = 2002$

$x = *(2002) = 12$

③ $x = (*Ptr)++$ // value at this address will get increment

$x = 10$

$Ptr = 2000$

$\therefore x = 11$

④ $x = ++*Ptr;$

$= ++(*Ptr);$

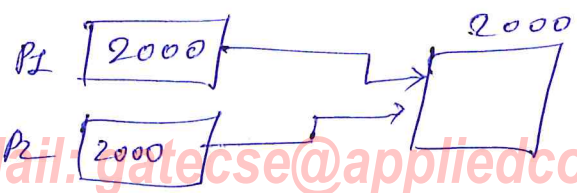
$x = 11$ and Ptr will remain 2000

Comparison between pointers

== , != , < , <= , > , >=

NULL or same address or same type

it will return true



void pointer is basically a pointer of void type - can Compare void pointer with any type of pointer

⇒ <, <=, >, >= operators are valid only if both the pointers are of same type or Null type.

True $P_1 == P_2$

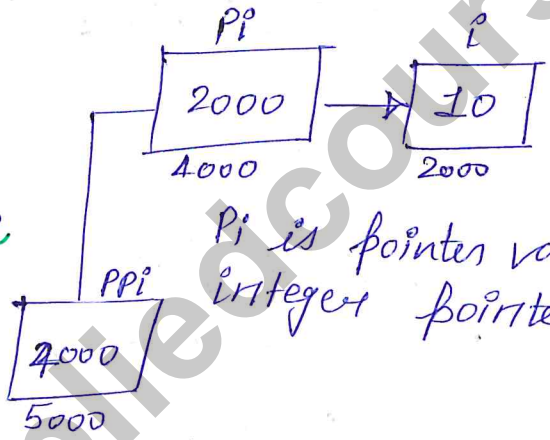
```
int *P1 = NULL;
float *f1 = NULL;
```

P1 == f1 true because both have value NULL.

Pointers to pointers

```
int i = 10;
int *Pi = &i;
int **PPi = &Pi;
```

Pointer to a pointer



Pi is pointer variable of type integer pointer.

Syntax : datatype ** Var-name ;

① $i \Leftrightarrow *Pi \Leftrightarrow **PPi$ → 2000

② $\&i \Leftrightarrow Pi \Leftrightarrow *PPi$ → 2000

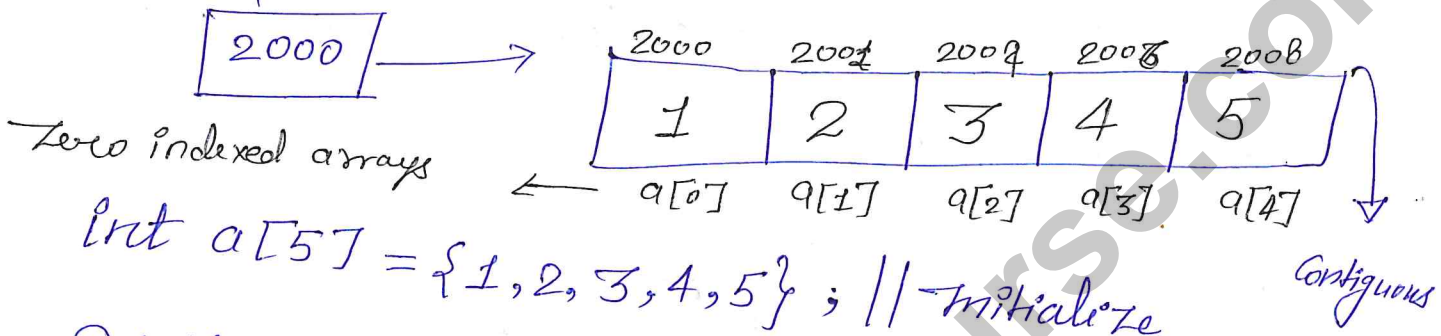
[9.4] Pointers and Array

Char str[10]; basic examples → Arrays

int a[5]; // Array declaration → constant pointer of type int

data type | name | size

Create space for storing 5-integers



int a[5] = {1,2,3,4,5}; // Initialize

Printf("\n %d", a[0]);

⇒ a is the constant pointer of type int, which is pointing to very first element. by storing the address of a[0] itself.

i+th → a[i-1] ✓

& a[0] = 2000

& a[1] = 2002

WAP to print the value and address of the elements of an array.

```
#include <stdio.h>
int main()
{
int arr[5] = {5,10,15,20,25};
int i;
for(i=0; i<5; i++)
{ Printf("value of arr[%d] = %d\t", i, arr[i]);
Printf("address of arr[%d] = %p\t", i, &arr[i]);
```

O/P:

value of a[0] = 5
value of a[1] = 10
value of a[2] = 15
value of a[3] = 20
value of a[4] = 25

Address of a[0] = 0x ---- 0
address of a[1] = 0x ---- 4
address of a[2] = 0x ---- 8
address of a[3] = 0x ---- C
address of a[4] = 0x ----

a → const pointer to 0th element. & a[0] → 2000
a+1 → const pointer to 1st element. & a[1] → 2002
⋮ ⋮
a+i → " to ith element. & a[i] →

*a → 1 = a[0]
*(a+1) → 2 = a[1]
*(a+i) → a[i] // This is core relationship between arrays & pointers in C

Whatever is written in the form of array, has an equivalent representation in terms of pointers.

a[i] ⟺ *(a+i) ⟺ *(i+a) ⟺ i[a]

*(a+i) = *(i+a)

```
int main()
{
 int arr[5]={5,10,15,20,25};
  int i=0;
 for(i=0; i<5; i++)
 {
 Printf("value of arr[%d] = ", i);
 Printf("%d\t", arr[i]);
 Printf("%d\t",*(arr+i));
 Printf("%d\t", *(i+arr));
 Printf("%d\ti", i[arr]);
 Printf("Address of arr[%d] = %P\n", i, &arr[i]);
 }
 return 0;
}
```

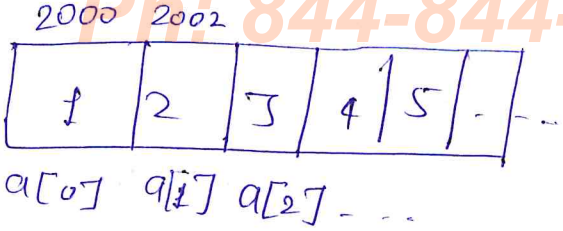
O/P:

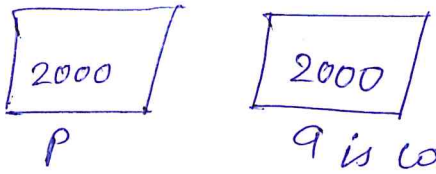
value of arr[0] = 5 5 5 5

Address of a[0] = 0x 0

valu

int *P = a;





*(P) = 1
*(P+1) = 2

a → array name → Const
P → Ptr →

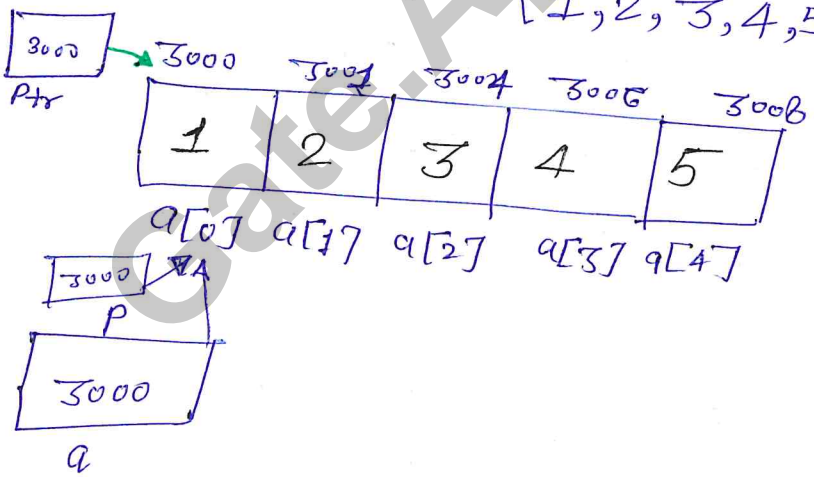
P = &Num ; ✓
P++ ; ✓
P = P-1 ; ✓

2000 ← P = P+1
2 ← *P

a = &Num; ✗
a++; ✗
a = a-1 ; ✗

Pointers to an Array

int *p; // Integer pointer
int (*Ptr)[5]; // pointer to an int array of size 5.
int a[5] = {1,2,3,4,5};



P = a; 0th element
Ptr = a; whole array
P++; → P = 3002
Ptr++; → Ptr = Ptr+1
⇒ Ptr + 1 * sizeof(5*2)
⇒ 3000 + 10
= 3010

Datatype (*varname)[size];

[9.5] Pointers and 2D Array

Declaration of 2D array:

```
int a[3][4]; // 2D array
```

1D array as a sequence of numbers.

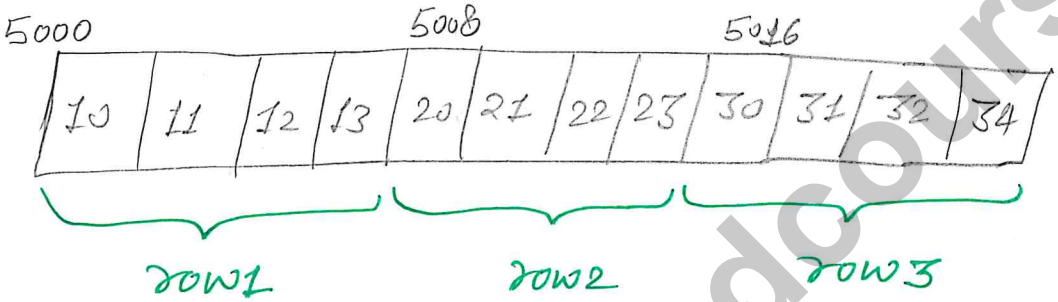
Initialization:

```
int a[3][4] = { { 10, 11, 12, 13 },
                { 20, 21, 22, 23 },
                { 30, 31, 32, 34 },
              };
```

3 - 1D arrays & each array contains 4 - elements.

	col1	col2	col3	col4
row1	10	11	12	13
row2	20	21	22	23
row3	30	31	32	34

How it is stored in Memory: Logical representation



row major order

```
a[0][3]; // goto 0th row & 3rd column
```

a[0][3] = 12 (in array) a[0][3] = 12 want to change as 12

⇒ C uses row-major order.

```
int a[3][4];
```

	col0	col1	col2	col3
row1	10 5000	11 5002	12 5004	13 5006
row2	20 5008	21 5010	22 5012	23 5014
row3	30 5016	31 5018	32 5020	34 5022

5000
a is the pointer to the 0th row - 1D array

5008
(a+1) : pointer to 1st row - 1D array

5016
(a+2) : pointer to 2nd row - 1D array

} size of each row = 4

a[0] → *(a+0) — 5000
a[1] → *(a+1) — 5008
a[2] → *(a+2) — 5016

Type : int pointer

(a+2) — 3rd row
*(a+2) - 1st element of 3rd row

a[i][j]; a[i] → *(a+i) → 1D array

*(a+i)+j

a[2][3];

1D pointer to the ith row : (a+i)

*(a+i) : integer pointer to the 0th column of ith row

*(a+2) = 5016 : integer pointer not array pointer

*(a+2)+3 = 5016 + 3 * sizeof(int)
= 5016 + 3*2 = 5022

((a+2)+3) find value of this.

↓
34 *(5022) = 34

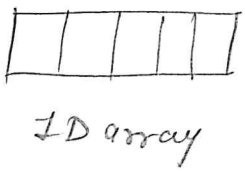
int (*Ptr)[4]; // Ptr is a pointer to an integer array of size 4.

Ptr = a; // creating a 1D pointer, (pointing to 1D array)

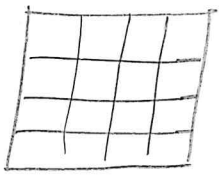
(Ptr+i); // 1D integer array pointer

Similarly, we can use 3D arrays, 4D arrays

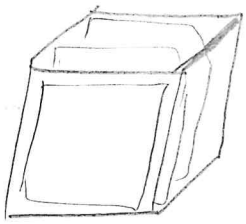
Ex: a int a[3][4][5];



1D array



2D array



3-D array (Multidimensional)

int a[3][4][5]

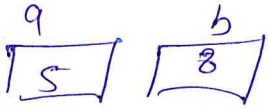
a represents pointer to 2D array

[9.6] Pointers and Functions

```
#include<stdio.h>
void value(int x, int y);
int main(void)
{
  int a=5, b=8;
  Printf("a=%d, b=%d", a, b);
   value(a, b);
  Printf("a=%d, b=%d\n", a, b);
    return 0;
}
void value(int x, int y)
{
    x++;
    y++;
Printf("x=%d, y=%d\n", x, y);
}
```

O/P = a=5 b=8
a=6 b=9
a=5 b=8

main()



value(5, 8)

x++ 6
y++ 9
x=6, y=9
after end of value

5,8

⇒ In call by value, value of actual arg copied to the formal arg, and if formal arg are modified.

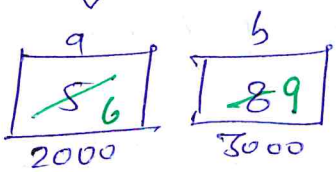
⇒ Whatever modifications are made in formal args that doesn't reflect back to calling function.

Pass by reference :- In pass by reference, we use pointe

By using address, we are modifying value.

```
#include <stdio.h>
void ref (int *p, int *q);
int main (void)
{
  int a=5, b=8;
  Printf("a = %d, b=%d\n", a,b);
    ref (&a, &b);
  Printf("a=%d, b=%d\n", a,b);
    return 0;
}
void ref (int *p, int *q)
{
    (*p)++;
    (*q)++;
  Printf("*p=%d, *q = %d\n", *p, *q);
}
```

main ()



ref (2000, 3000)

= (*p)++ =
= (2000)++ = 6
(*q)++ = 9

① return (expr); → return one value

We want to return multiple values → f (sq, ft, loc)
↳ price, tax

call by reference

C → address → Pointers

java does not have concept of pointers

```
main()
{
  f();
}
f(SQ.ft, loc, float *price, float *tax)
{
  ==
}
```

Pointer is also a data type.

② function return pointer

```
int *f(int x, int y)
{
  ==
}
```

Syntax:

```
type *f( )
{
  ==
}
```

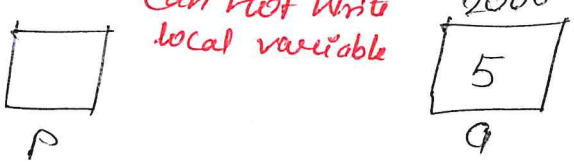
```
main()
{
int *P;
  P=f();
}
```

```
int *f()
{
int a=5;
return &a;
}
```

local variable for func f

Do not return address of a local variable

Can not write local variable



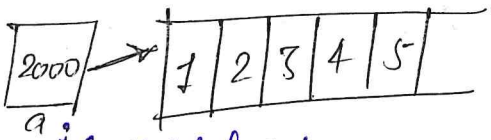
address of integer is integer pointer

So P=2000

func will return value as 2000

⑤ Passing Arrays to a function:

```
main()
{ int a[5] = {1,2,3,4,5};
   f(a);
}
void f(int a[])
{
  a[0]++;
}
```



a : const int pointer

formal arg

or int a[5]

or int *a

④ Passing 2D arrays to a function:

```
void f(int a[3][4]){---}   ✓

void f(int a[][4]){---}    ✓

void f(int a[][]){ }       ✗
```

Can not leave empty

// left most braces can leave empty but not second most

```
void f(int (*a)[4]){---}

void f(int a[][][7]){---}
```

Can not leave empty

// only left most can be left empty

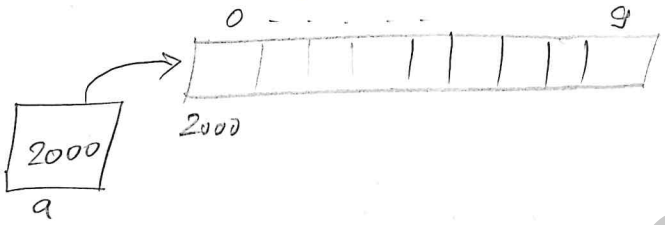
a[i][j] or *(*(a+i)+j)

```
void f(int a[][4][5]){---}   ✓
```

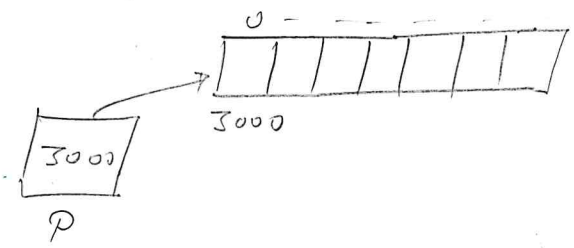
[9.7] Array of pointers and void pointers

int a[10]; // array of 10 integers

pointer → data type



int *P[10];

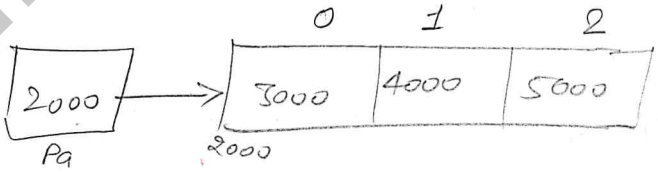


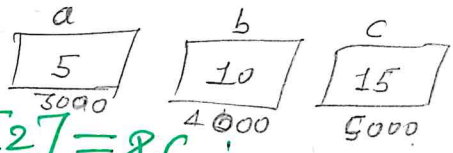
Creating integer pointer array of size 10

int (*P)[10]; // pointer to an integer array (1D) of size 10

WAP for Array of pointers.

```
#include <stdio.h>
int main(void)
{
  int *Pa[3]
  int i, a=5, b=10, c=15;
  Pa[0]=&a; Pa[1]=&b; Pa[2]=&c;
  for(i=0; i<3; i++)
  {
    Printf("Pa[%d]=%P\t", i, Pa[i]);
    Printf("*Pa[%d]=%d\n", i, *Pa[i]);
  }
  return 0;
}
```





O/P:

Pa[0] = 0x ---- C0 *Pa[0]=5

Pa[1] = 0x ---- C4 *Pa[1]=10

Pa[2] = 0x ---- C8 *Pa[2]=15

2D-Array

```
int a[3][4] = {{10,11,12,13},{20,21,22,23},{30,31,32,33}};
```

P[0]	5000 10	11	12	13
P[1]	5008 20	21	22	23
P[2]	5016 30	31	32	33

a[i][j] ⟺ *(P[i]+j)

a[i][j]
ith row jth col
a[1][2] = 22
*(P[1]+2)
↓
5008 + 2 × sizeof(int)
5008 + 4 = 5012

void pointer:

⇒ void pointer means I don't know how many bytes to access
*Vptr

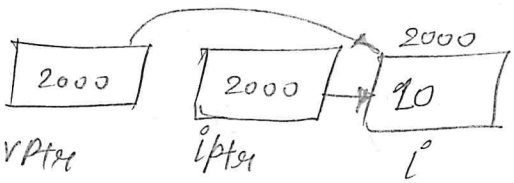
```
void *vptr;
int *iptr;
float *fptr;
int i=10;
float f=1.0;
iptr = &i
fptr = &f
```

iptr = fptr; ✗
Copying floating type pointer in integer pointer.

vptr = iptr; ✓
vptr = fptr; ✓
*vptr Dereferencing

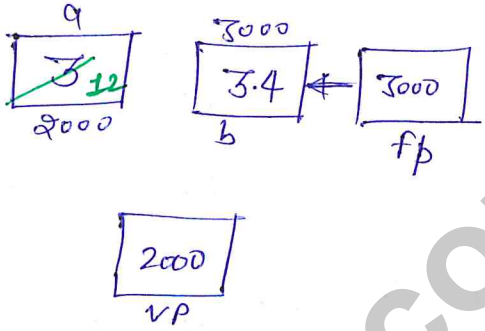
Type cast:

*((int *) Vptr) → 2 Bytes
↑ explicit type cast
Starting from address 2000



WAP for dereferencing a void pointer.

```
#include <stdio.h>
int main()
{
  int a = 3;
  float b = 3.4, *fp = &b;
  void *vp;
  vp = &a;
  Printf("value of a = %d\n", *(int*)vp);
    *(int*)vp = 12;
  Printf("value of a = %d\n", *(int*)vp);
      vp = fp;
  Printf("value of b = %f\n", *(float*)vp);
    return 0;
}
```

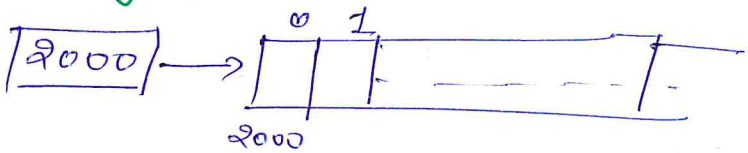


O/P: value of a = 3
value of a = 12
value of b = 3.400000

[9.8] Dynamic Memory Allocation (Pointers)

Static Memory allocation

⇒ int a[100];



⇒ Instead of storing 100 integer, I am just inserting 30 integers & this decision is taken at run time.

⇒ We can not do this as memory for 100 variables is already created and we can not change it.

⇒ We can not change this allocation dynamically.

⇒ Dynamically or run time : Size of block of memory
- ↳ Create
- ↳ Change

↓

going to be stored in memory area : heap

There are some functions which are used for creating dynamically :

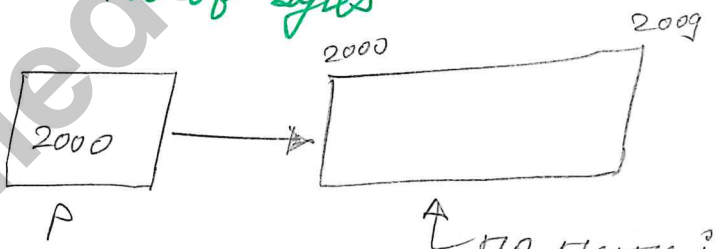
malloc, calloc, realloc, free } stdlib

```
int *p = (int *) malloc (10);
```

↑ No. of bytes

```
void * malloc (size_t size)
```

user defined



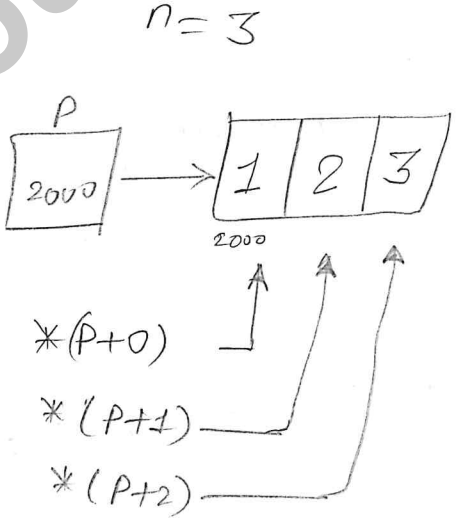
↳ no name is associated with dynamically created memory.

```
int *p = (int *) malloc (5 * sizeof(int));
if (p == NULL)
{
Printf ("\n Not sufficient memory");
}
```

NULL is a constant which is 0, and defined in stdlib.

⇒ If no memory is left then it will return NULL.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 int *p, n, i;
Printf("enter the no. of integers to be entered:");
Scanf("%d",&n);
p=(int*) malloc(n*sizeof(int));
if(p==NULL)
{
Printf("Memory not available\n");
  exit(1);
}
for(i=0; i<n; i++)
{
 Printf("enter an integer:");
  Scanf("%d", p+i);
}
for(i=0; i<n; i++)
   Printf("%d\t", *(p+i));
  return 0;
}
```



O/p: enter the no. of integers to be entered: 3
enter an integer: 1
enter an integer: 2
enter an integer: 3

1 2 3

② calloc

```
int *P = (int *)calloc (5, sizeof(int))
```

↑ No. of blocks ↑ size of a block

Same as

```
(int *) malloc (5, sizeof(int))
```

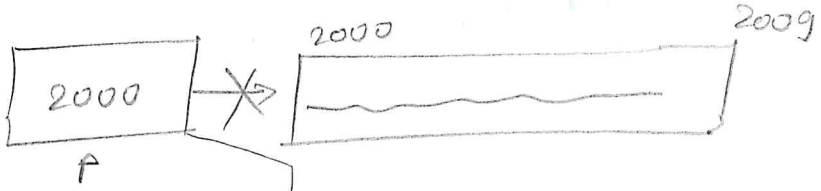
⇒ calloc also returns NULL if there is insufficient m/p

⇒ When you use malloc → It is initialized with some garbage or random value.

⇒ When you use calloc → M/r locations are initialized with zero.

③ Realloc

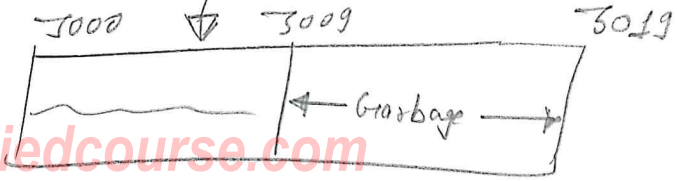
```
int *P = (int *) malloc (10);
```




```
int * P = (int *) realloc (p, 20)
```

⇒ New size must be either greater or less than older created memory.

NewSize > oldSize
NewSize < old Size



⇒ If **new size > old size**. There will be no loss as everything will be copied to newly created memory.

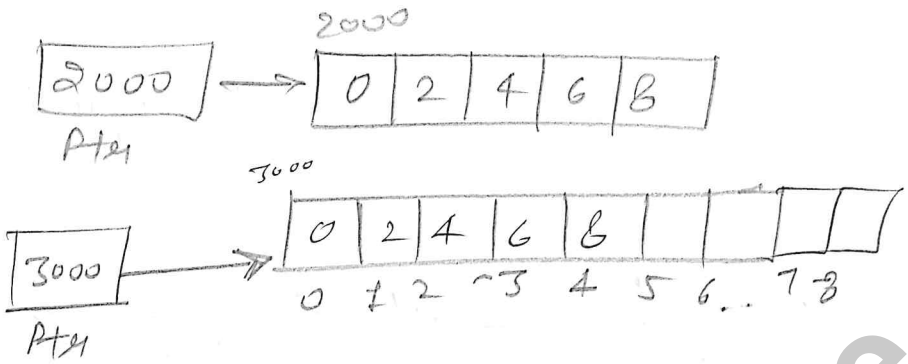
⇒ If **new size < old size**. There will be loss of data in newly created data as it can only store lesser data.

WAP for realloc function

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i, *Ptr;
  Ptr = (int*) malloc (5 * sizeof(int));
  if (Ptr == NULL)
  {
    Printf ("Memory not available\n");
    exit(1);
  }
  for (i=0; i<5; i++)
      *(Ptr+i) = i*2,
  Ptr = (int*) realloc (Ptr, 9 *sizeof(int));
  if (Ptr == NULL)
  {
    Printf("Memory not available\n");
      Exit(1);
  }
  for(i=5; i<9; i++)
    *(Ptr+i) = i*10;
```

```
for(i=0; i<9; i++)
    Printf("%d", *(Ptr+i));
    return 0;
}
```

O/P:



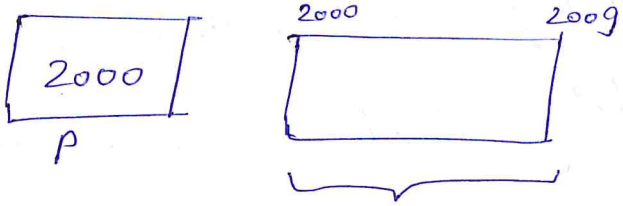
0 2 4 6 8 50 60 70 80

Ptr[5] → *(Ptr+5)

free()

```
void free(void *P);
int *P = (int*)malloc(10)
    free(P);
```

⇒ When program terminate



all mlr will be freed by default.

⇒ It's important to free memory so that other variables can use it

DAFM → heap — heap will become full.

```
f()
{
  int *P = (int *) malloc (10);
}
main()
{
}
```

→ local var

→ I can not use P again b'coz scope of P is with in this fn only.

→ If P is not needed in main function then we should always free this.

Memory leak : → I have a chunk of memory, but I dont have any way to access this memory.

Chapter – 10 Strings

[10.1] Introduction to Strings in C

⇒ Strings are nothing but character arrays

String const → String literals "abc" "" " "

"" → empty

" " → Nothing

Zero or more chars

"A" ≠ 'A'

"A" → String

'A' → Char

| a | b | c | \0 |

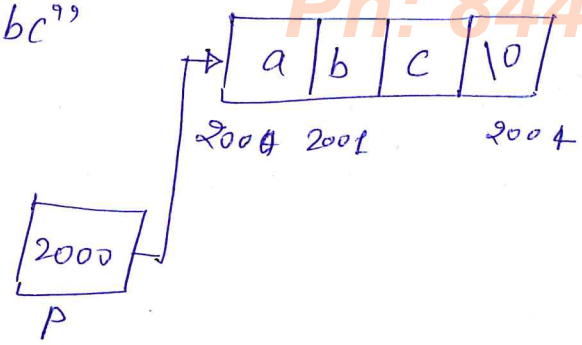
→ String terminator or Null character

→ to represent end of any string we store \0 (backslash zero) at the end of string

Ascii value of \0 ⇒ 0

| A | \0 |

Char *P = "abc"



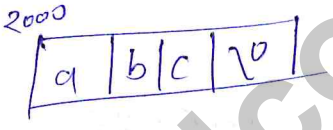
⇒ Internally a pointer is used to access the string constant.

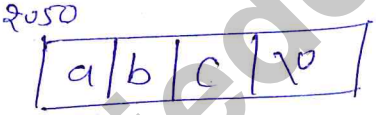
Printf ("Hi There:\n");

Printf (Char *)

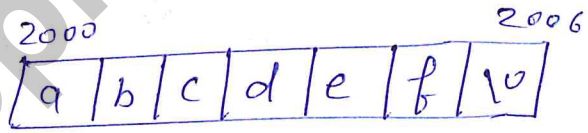
"abc"

"abc"





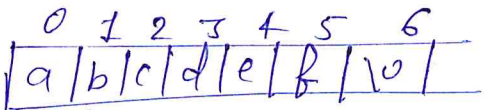
"abcdef"[2]



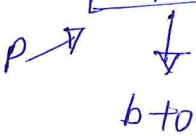
2000 + 2 sizeof (Char)

c

Char *P = "abcdef"

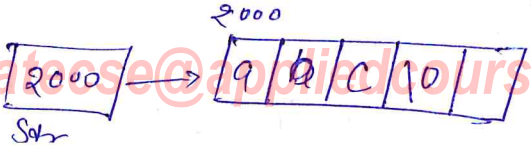


P[1] = a



undefined in C

Char Str[5] = "abc"; // String initialization



Exception: When you are using a string constant as part of a string initialization

→ In this case, char str[5] = "abc" → Here is no explicit memory is created

firstly character array of size 5 is initialized.
String constant not creating new memory.

Escape Character \

"abc\\def" → abc\def

abc"def → "abc\"def"

↓
telling compiler it is not closing of double quote.

"abc" "def" → "abcdef"

| a | b | c | d | e | f | \0 |

∴ String constant is just like character constant.

String Variables :

Same

char s[] = {'a', 'b', 'c', 'd', 'e', 'f', '\0'};

s → | a | b | c | d | e | f | \0 |

char s[] = "abcdef"; char s[10] = "abcdef";

WAP to print characters of a string and address of each character.

```
#include <stdio.h>
int main()
{
char str[] = "GATE",
int i;
for(i=0 ; str[i] != '\0' ; i++)
{
Printf("character = %c\t", str[i]);
Printf("Address = %p\n", &str[i]);
}
return 0;
}
```

Str → | G | A | T | E | \0 |

%c for char, %d for int
%f for float, %s for string

O/P:

Character = G Address = 0x---50
Character = A Address = 0x---51
Character = T Address = 0x---52
Character = E Address = 0x---53

} exactly 1B
Char is taking 1B only.

WAP for i/p & o/p strings using scanf() and printf().

```
#include <stdio.h>
int main()
{
char name[40];
Printf("enter a name:");
Scanf("%s", name);
Printf("%s %s\n", name, "course");
return 0;
}
```

O/P: Enter a name: abc
abc course

[10.2] Library functions

Strlen in C reference
www.cplusplus.com

<String.h> <math.h> → Sqrt

Strlen

```
size_t strlen (const char *str);
```
fn declaration

Returns the length of C string str.

Char mystr[100] = "test string";

t	e	s	t		s	t	r	i	n	g	\0

defines an array of characters with a size of 100 chars but the C string with which mystr has been initialized has a length of only 11 characters.

∴ While sizeof(mystr) evaluates to 100
strlen(mystr) returns 11

```
#include <stdio.h>
#include <string.h>
int main()
{
 Char szInput[256];
 Printf("Enter a sentence:");
  gets(szInput); // scanf("%s", szInput);
 Printf("The sentence entered is %u characters long.\n",
        (unsigned) strlen(szInput));
 return 0;
}
```

O/p: Enter a sentence: just testing

```
#include <stdio.h>
#include <string.h>
unsigned int astrlen (char str[])
{
  int i=0;
 while(str[i]!='\0')
  i++;
  return i;
}
int main()
{
  char str[50];
 Printf("enter a string:");
  gets(str);
 Printf("Length of string is: %u \n", astrlen(str));
  return 0;
}
```

0	1	2	3	4	5	6
a	b	c	d	e	f	\0

O/P: Length of the string is: 6

Strcmp: Compares the C string str1 to the C string str2.

```
int strcmp (const char * str1, const char * str2);
```

This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until the terminating null-character is reached.

⇒ This function performs a binary comparison of the characters. For a function that takes into account locale-specific rules.

Ex:

"abcdef" "abdef"

< 0

0

> 0 "abdef" "abcd"

```
#include <stdio.h>
#include <string.h>
int main()
{
 char key[] = "apple";
 char buffer[80];
 do{
  Printf("Guess my favorite fruit?");
  fflush(stdout);
  Scanf("%79s", buffer);
  }
  While(strcmp(key, buffer) != 0);
  Puts("correct answer!");
  return 0;
  }
```

apple, orange

user enter : orange

O/P:

Guess my favorite fruit? orange
Guess my favorite fruit? apple
Correct answer!

Strcpy

```
Char *Strcpy = (Char *dest, Const Char *source);
```

⇒ Copies the C string pointed by source into the array pointed by destination. Including terminating null character (and stopping at that point)

⇒ To avoid overflows, the size of the array pointed by destination shall be long enough to contain the same string as source (including terminating null character), and should not overlap in memory with source.

⇒ Destination is returned.

```
#include <stdio.h>
#include <string.h>
int main()
{
Char str1[] = "sample string";
Char str2[] = [40];
Char str3[] = [40];
Strcpy (str2, str1);
Strcpy (str3, "copy successful");
Printf("str1: %s\n str2: %s\n str3: %s\n", str1, str2, str3);
return 0;
}
```

} When not assigning, we need to use strcpy for assigning.

O/P:

Str1: sample string
Str2: Sample string
Str3: Copy successful

Str2[0] = 'a' ✓, Str[1] = 'b' ✓

Str2 = "abcdef" ✗ Not valid

Strcat : for concatenation

① Appends a copy of source string to the destination string.

```
#include <stdio.h>
#include <string.h>
int main()
{
 char str[80];
 strcpy(str, "these");   these\0
 strcat(str, "strings");   remove \0 and add strings\0
 strcat(str, "are");
 strcat(str, "concatenated");
 puts(str);   // puts means print
 return 0;
}
```

```
char *strcat(char *dest, const char *source);
```

for, while, arrays, pointer

② terminating null character in destination is over-written by the first charcter of source.

③ Null character is included at the end of the new string form by the concatenation of both in the destination.

Strncpy

char * strncpy (char * destination, const char * source, size_t num)

⇒ Copies the first num character of source to dest.

⇒ If the end of source c string (which is signaled by a null character) is found before num characters have bee copied.

⇒ Destination is padded with zeros until a total of num characters have been copied. written to it.

⇒ No Null-character is implicitly appended at the end of destination if source is longer than num. Thus in this case destination is not considered a null terminated c string (reading it as such would overflow)

⇒ destination and source shall not overlap (

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[] = "To be or not to be";
char str2[40];
char str3[40];
strncpy (str2, str1, sizeof(str2));
strncpy (str3, str2, 5);
str3[5] = '\0';
puts (str1);
puts (str2);
puts (str3);
```

O/P: To be or not to be
To be or not to be
To be

```
#include<stdio.h>
#include<string.h>
int main()
{
 char str1[20];
 Char str2[20];
 Strcpy (str1, "To be");
 Strcpy (str2, "or not to be");
 Strncat (str1, str2, 6);
 Puts (str1);
 return 0;
}
```

O/P : To be or not

Strstr

```
const char *strstr (const char *str1, const char *str2);
      char *strstr( char *str1, const char *str2);
```

Locate Substring

Returns a pointer to the first occurence of str2 to str1, or a null pointer if str2 is not part of str1.

The matching process does not include the terminating null-characters, but it stops there.

```
char *strstr ( const char * , const char *);
```

```
#include <stdio.h>
#include <string.h>
int main()
{
char str[] = "this is a simple string";
char * pch ;
pch = strstr (str, "simple");
strncpy (pch, "sample", 6);
puts (str);
return 0;
}
```

simple → sample

copy only 6 characters

O/P: this is a sample string

Word

Search + Replace
F R

Strtok

String tokenizer

, space

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
char str[] = "- This, a sample string.";
char * pch;
printf ("Splitting string \"%s\" into tokens:\n", str);
pch = strtok (str, " ,.-");
while (pch != NULL)
{
printf ("%s\n", pch);
pch = strtok (NULL, " ,.-");
}
return 0;
}
```

pch → This
pch → a

O/P: Splitting string "- This, a sample string." into tokens:

This
a
sample
string

Sprintf

Write formatted output to sized buffer

```
#Include <Stdio.h>
int main()
{
 char buffer[50];
 int n, a=5, b=3;
 n = Sprintf (buffer, "%d plus %d is %d", a, b, a+b);
 Printf ("[%S] is a string %d chars long \n", buffer, n);
 return 0;
}
```

SScanf

: Instead of taking input from keyboard, take it from other strings.

WAP to convert strings to integer and float values.

```
#include <stdio.h>
int main()
{
 Char Str1[10] = "1000";
 Char Str2[10] = "125.50";
 int x; float y;
 SScanf (Str1, "%d", &x);
 SScanf (Str2, "%f", &y);
 Printf("value of x=%d, value of y=%.2f\n", x, y);
 return 0;
}
```

O/P: value of x = 1000, value of y = 125.50

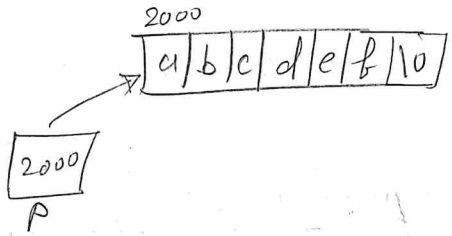
⇒ Covered main major functions of string.h

<math.h> functions math.h c reference

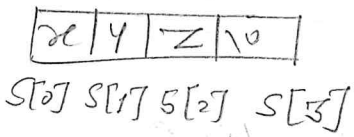
1. double acos (double x)
2. double asin (double x)
3. double atan (double x)
4. double atan2 (double x)
5. double cos (double x)
6. double cosh (double x)
7. double sin (double x)
8. double sinh (double x)
9. double tanh (double x)
10. double exp (double x)
11. double frexp (double x, int * exponent).
12. double ldexp (double x, int exponent)
13. double log (double x) (base e)
14. double log10 (double x)
15. double modf (double x, double * integer)
16. double Pow (double x, double y);
17. double sqrt (double x)
18. double ceil (double x)
19. double fabs (double x) return absolute of x
20. double floor (double x)
21. double ldexp (double x, double y)

Pointers, Arrays and Strings [10.3]

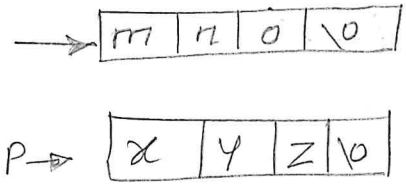

```
char *P = "abcdef";
```



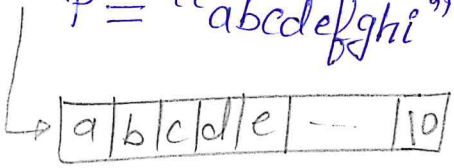

```
char S[] = "xyz";
```




```
char = "mno";
P = "xyz";
```




```
P = "abcdefghi"
```

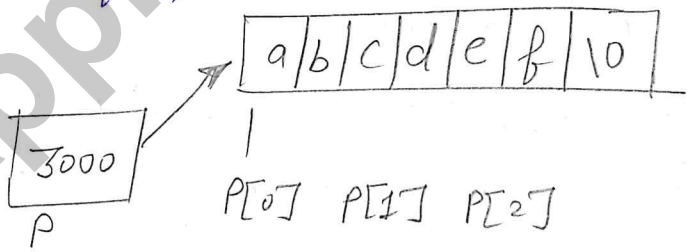



```
Scanf("%s", S);
```

abc ✓
abcd ✗

→ `char *P = "abcdef";`

`P[2] = a;`




```
Scanf("%s", P);
Strcpy(P, "abc"); ✗ Not allowed bcoz we are trying to put const
char *P;
Scanf("%s", P);
P = (char *) malloc(20);
```

Array of Strings

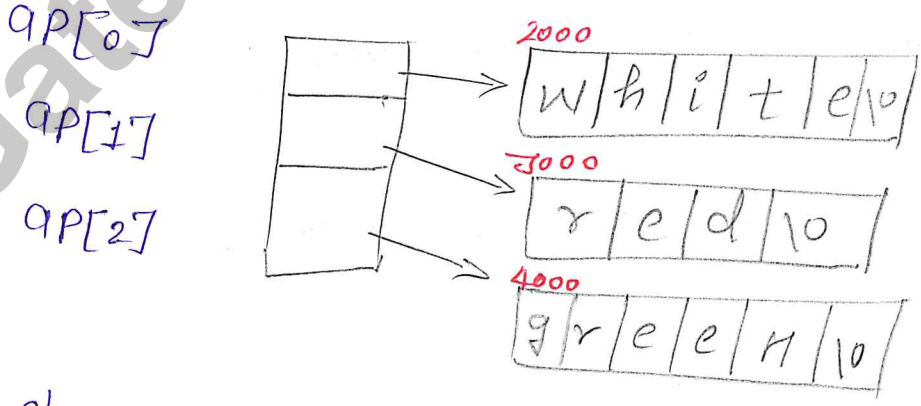
Char a[5][10] = { "White", "Red", "green", "blue", "yellow" }

	0	1	2	3	4	5	6	7	8	9
a[0] → 0	W	h	i	t	e	\0				
1	R	e	d	\0						
2	g	r	e	e	n	\0				
3	b	l	u	e	\0					
a[4] → 4	y	e	l	l	o	\0				

Char a[5][10] = { { 'W', 'h', 'i', 't', e },
{ 'R', 'e', 'd' },
{ 'g', 'r', 'e', 'e', 'n' }, { 'b', 'l', 'u', 'e' },
{ 'y', 'e', 'l', 'l', 'o' } }

Array of pointers to strings

Char *ap[ ] = { "White", "red", "green" };



Char a[5][10]; → for strings already Space allotted

Char *ap[5]; → space for strings is not allocated

a[0] = "jan"

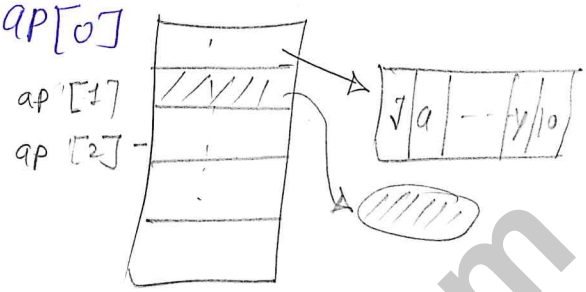
I am creating in memory JAN will be creating and try to allocate pointers to this memory

const

✗

Strcpy (a[0], "jan");

a[0] = "january";



Strcpy (ap[i], "feb"); ✗

scanf ("%s", a[2]); ✓ take the input & store it

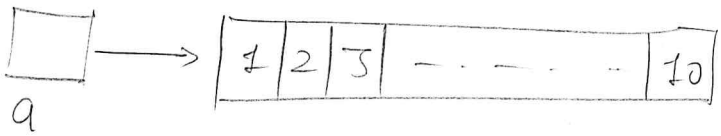
scanf ("%s", ap[2]); ✗ As there is not space allocated for ap[2]

ap[2] = (char *) malloc (20);
scanf ("%s", ap[20]);

⇒ If you print "abc". it is going to point constant in memory.

[10.4] Pointers to Array revisited

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};



int * const

a[2] = *(a+2); value stored at (a+2)

② Pointer to array (1D)

```
int (*P)[10];
```

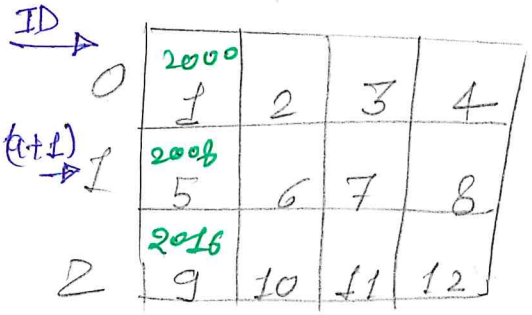
Datatype → `int *P[10];` → array of int ptrs

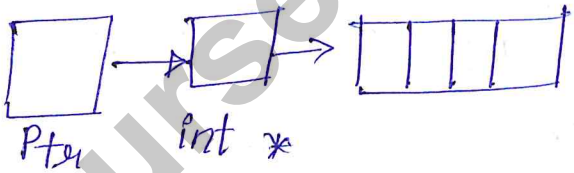
pointer to int

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

int ** Ptr

Double ptr





a ; Ptr to 1D array

a → 1D → (2000)

a+1 → 1D → 2008

(a+2) → 1D → 2016

*(a+1) ⇒ int * = 2008

*(int **) ⇒ int *

*(int *) ⇒ value

a[1][0] = *(*(a+1)+0)

a[i][j] = *(*(a+i)+j)

Consider C Program.

```
#include <stdio.h>
int main()
{
  int a[3][4][2] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
                    16,17,18,19,20};
  Printf("%d", *(*(a+2)-3))[2]);
  return 0;
}
```

What is output of code.

Soln:

a → a is pointer to 2D array

a+0

1	2
3	4
5	6
7	8

a+1

9	10
11	12
13	14
15	16

a+2

17	18
19	20
–	–
–	–

(1) a → address of (1)

(2) a+2 →

(3) *(a+2) → pointer to 1D array = a[2]
int **

(4) (*a+2) − 3 ⇒ pointer to 1D array
Jrows back → address of (11)

(5) *(*(a+2)−3)[2] ≡ *((*(a+2)−3)+2)

= *(Pointer to 1D array + 2 2 rows)

= 15

What is output of C code.

```
#include <stdio.h>
int main()
{
char arr[2][3] = { 'A', 'P', 'P', 'L', 'E' 'E' 'D',
                   'C', 'O', 'U', 'R', 'S' 'E' };
Char (*P)[2][3][3] = &arr ;
Printf("%d", (*(*(*P)[0]+5)) - (*(*(*P)[1]-3)));
return 0;
}
```

Sol'n:

*P[0]

A	P	P
L	I	E
D		C

*P[1]

O	U	R
S	E	\0
\0	\0	\0

+P = Pointer to 2D

*P[0] = *(*(P)+0

(*P)[0]+5 − **(*P)[1]-3

E − D = 1

Ptr to 1D Array → int **

Ptr to 2D Array → int ***

Ptr to 3D Array → int ****

Dereferencing

= *(int ***)

= (int **)

Chapter-12 Structures and Unions

[12.1] Structures & Unions

Array: collection of similar items → homogeneous

```
int a[10];
```

Student → logical entity / object (real-world)

name, rno, marks

String int float

```
struct student {
    char name[30];
    int rno;
    float marks;
} s1, s2, s3;
```

(struct: keyword; student: identifier; user defined data type)

} members / attributes

Collection of the heterogeneous members

all of them are attributes of a real world entity.

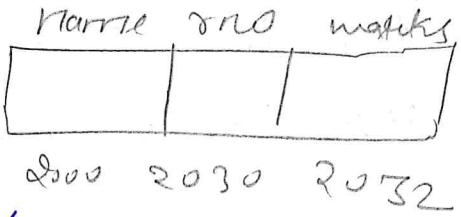
⇒ Declare a variable:

① with in definition

② `struct student s1, s2, s3;`

Memory

`struct student s1;`



⇒ all attributes or members of struct are stored contiguously

Initialize Structure :

(1) Init along with definition of the struct.

(2) Struct Student S2 = {"Neelima", 1, 98.0};
Decl Init

(3) Struct Student S3 = {"Naveen"}; rno & marks → init to 0

(4) Struct Student {
≡
float marks = 90.0; ✗ NOT valid
No m/r
}

Access :

Struct Student S1;

name	rno	marks
30B	2B	4B

✗ S1.name = "Muruli"

because Muruli is a constant

Strcpy (S1.name, "Muruli");

S1.rno = 2;

S1.marks = 95.0;

↑ dot operator for accessing elements

WAP to print the values of structure member.

```
#include <stdio.h>
#include <string.h>
Struct student
{
   char name[20];
   int rollno;
   float marks;
};
int main()
{
 Struct student str1 = {"satish", 25, 68};
 Struct student stu2, stud3;
 Strcpy(stud3.name, "Naveen");
  stu2.rollno = 26;
  stu2.marks = 98;
Printf("enter name, rollno., and marks for stud3:");
Scanf("%s %d %f", stud3.name, &stud3.rollno, &stud3.marks);
Printf("stud1: %s %d %.2f \n", stud1.name, stud1.rollno,
                                stud1.marks);
Printf("stud2: %s %d %.2f \n", stud2.name, stud2.rollno, stud2.marks);
Printf("stud3: %s %d %.2f \n", stud3.name, stud3.rollno, stud3.marks);
return 0;
```

user-defined Global data types

Assignment:

$S1 = S2;$

```
Struct student
int a, b
a = 10
b = a;
```

(1) Unary, relational, arithmetic operations are not allowed bitwise → Not allowed on struct variables.

(2) attributes/members of struct — st.rollno

S1 & S2 ✗ S1 > S2 ✗ S1 + S2 ✗ Not allowed

-S1; ✗

⇒ sizeof

✓ sizeof (struct student);

✓ sizeof (S1);

⇒ Arrays

struct student sa[10];

Access → sa[0].name;
sa[0].rollno;
:
sa[i].name;

int a[10];
for accessing a[0], a[1]

⇒ struct student a[3] = {
{"veerma", 6, 96},
{"satish", 2, 98.0},
{"Naveen", 3, 99.0}
};

⇒ It is same as initializing an array.

[12.2] Let's dive into Structures

1) Nested structure : Structure with in Struct

```
Struct student {
        char name[30];
        int rno;
        Structure date {
                int day;
                int month;
                int year;
        } dob;
} s1, s2;
```

definition of Struct Student

definition of

⇒ dot operator has the same precedence as parenthesis

s1.dob.day ⇒ ((s1.dob).day) L to R

(.) dot operator : use to access members of structures.

(→) Arrow operator : is powerful w.r.t pointers to structure.

(2) Another way of writing

```
Struct date {
    ==
};
```

```
Struct student {
    ==
    Struct date dob;
    ==
};
```

2) Pointers to Structure :

```
Struct student s1, *P;
P = &s1;
(*P).name
```

name, rno, marks

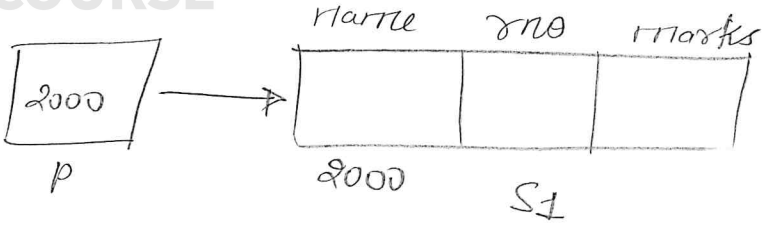
s1.name

s1.rno

dot has higher precedence over *

*P.name ⇒ *(P.name) ✗

If want to access name using pointer (*P).name



(*P).rno

Complex Syntax

P→name ≡ (*P).name

Much more neater, Less error prone syntax

(2) Ptr to an attribute

int *Ptr = &S1.rno;

(3) Pointer inside structure : (&, *, ., → operators)

```
Struct Student {
        char name[20];
        int *P;
        } S1, *SP;
```

S1 ↓ Student structure variable

SP ↳ pointer to the Student Structure

SP = &S1;

S1.P

*S.P = *(S.P) dot has higher precedence

*SP→P ⇒ *(SP→P)

WAP to understand pointers to structures.

```
#include<stdio.h>
Struct Student
{
   Char name[20];
    int rollno;
    int marks;
};
Int main()
{
Struct Student stud = {"Neelima", 25, 68};
Struct Student *Ptr = &stud;   → pointing to address of
Printf("Name - %s\t", Ptr → name);
Printf("Rollno - %d\t", Ptr → rollno);
Printf("Marks - %d\t", Ptr → marks);
return 0;
}
```

④ Structures & Functions:

⇒ Structures are user-defined

1) Passing Struct members as arguments.

```
#include<stdio.h>
#include<string.h>
Struct Student {
        Char name[20];
        int rollno.;
        int marks;
        };
void print(char name[], int rollno, int marks);
int main()
{
```

```
Struct Student Stud1 = {"Neelima", 1, 87};
Struct Student Stud2;
Strcpy (Stud.name, "Naveen");
Stud2.rollno = 2;
Stud2.marks = 90;
Printf (Stud1.name, Stud1.rollno, Stud1.marks);
Printf (Stud2.name, Stud2.rollno, Stud2.marks);
return 0;
}
void fprint (char name[], int rollno, int marks)
{
  printf ("Name - %s\t", name);
  Printf ("Rollno - %d\t", rollno);
  Printf ("Marks - %d\t", marks);
}
```

(2) Pass Structure variable as argument to function:

```
void fprint (struct student);
  Printf (Stud1);
  Printf (Stud2);
  return 0;
}
void fprint (struct student stud)
{
  =
}
```

(3) Passing pointer to a structure as an argument

⇒ Can pass structure by reference

```
void print (struct student *);
void inc_marks (struct student *);
int main ()
{
 struct student stud1 = {"Satish", 1, 87};
 struct student stud2 = {"Naveen", 2, 90};
 inc_marks (&stud1);
 inc_marks (&stud2);
 printf (&stud1);
 printf (&stud2);
 return 0;
}
void inc_marks (struct student *studptr)
{
  (studptr -> marks)++;
}
void print (struct student *studptr)
{
 printf ("Name - %s\t", studptr -> name);
 printf ("Rollno - %d\t", studptr -> rollno);
 printf ("marks - %d\t", studptr -> maks);
```

any changes made here will be visible

(4) Returning a structure itself

#WAP to understand how a structure variable is returned from a function

```
#include <stdio.h>
Struct student
{
   char name[20];
   int rollno;
   int marks;
};
void print (struct student);
Struct student change (struct student
int main()
{
 struct student stud1 = {"Satish", 20, 87};
 Struct student stud2 = {"Naveen", 21, 90};
  stud1 = change (stud1);
  stud2 = change (stud2)
  Printf (stud1);
  Printf (stud2);
  return 0;
}
 Struct student change (struct student stud)
 {
  stud.marks = stud.marks + 5;
  stud.rollno = stud.rollno - 10;
   return stud;  — returning variable of type Struct
}
void print (struct student stud)
{
  Printf ("Name - %.s\t", stud.name);
  Printf ("Rollno - %.d\t", stud.rollno);
  Printf ("Marks - %.d\t", stud.marks);
```

WAP to understand how an array of structures is sent to a function

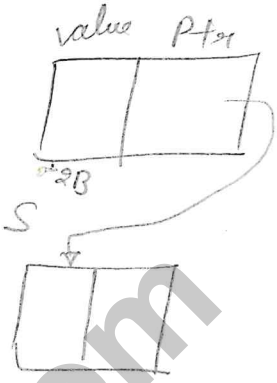
```
#include <stdio.h>
Struct student
{
    char name[20];
    int rollno;
    int marks;
};
void print (struct student);
void dec-marks (struct student stuarr[]);
int main ()
{ int i;
 Struct student stuarr[3] ={
                    {"Neelima", 12, 98};
                    {"Naveen", 11, 97};
                    {"Subbu", 13, 89};
                   };

 dec-marks (stuarr);
 for (i=0; i<3; i++)
  Print (stuarr[i]);
  return 0;
}
void dec-marks (struct student stuarr[])
{
 int i;
 for (i=0; i<3; i++)
  stuarr[i].marks = stuarr[i].marks - 10;
}
 void Print (struct student stud)
{ Printf ("Name - %s\t", stud.name);
  Printf ("Rollno - %d\t", stud.rollno);
} Printf ("Marks - %d\t", stud.marks);
```

Self Referential Structures

```
Struct node {
        Int value;
        Struct node *Ptr;
} S, S1;

S.Ptr = &S1;
```



⇒ self-referential structures are important for Linked-Lists.

[12.3] Unions and typedef

Unions: user-defined data type

↓ keyword

```
union result {
        Int marks;   - 2B
        Char grade;  - 1B
        float percentile; - 4B
} r;
```

⇒ Create memory of 4B not 7B, because 4B is max of these three numbers (2B, 1B, 4B).

⇒ I can either store marks or grade or percentile. I can't store all 3. like structure.

Number → any numeric → int → long → float → double

```
Union Number {
    int i;
    long l;
    float f;
    double d;
} n;
```

```
n.i = 10;
n.l = 100l;
n.f = 10.0f;
n.d = 10.0ff;
```

WAP to compare the memory allocated for a Union and structure variable.

```
#include <stdio.h>
Struct Stag {
        char c;
        int i;
        float f;
        };
Union Utag {
        char c;
        int i;
        float f;
        };
int main(void)
{
  Union Utag Uvar;
  Struct Stag Svar;
Printf("Size of Svar = %d\n", sizeof(Svar));
Printf("Address of Svar = %P\n", &Svar);
Printf("Address of members: %P %P %P\n",
        &Svar.c, &Svar.i, &Svar.f);
```

```
Printf("size of Uvar = %u\n", sizeof(Uvar));
Printf("Address of Uvar: %p\t", &Uvar);
Printf("Address of members: %p %p %p\n",
        &Uvar.c, &Uvar.i, &Uvar.f);
return 0;
}
```

O/P:

size of Svar = 12
Address of Svar = 0x---700
Address of members = 0x--700 0x--704 0x--708

size of Uvar = 4
Address of Uvar = 0x---6f0
Address of members = 0x---6f0 0x--6f0 0x--6f0

How do you Initialize Union:

only initialize the first member of union or first attribute of Union. You can not initialize more than that.

Union result r1 = {89};

```
int main()
{
  Union result
  {
    int marks;
    Char grade;
    float per;
  } res;
res.marks = 90;
Printf("Marks: %d\n", res.marks);
res.grade = 'A';
Printf("Grade: %c\n", res.grade);
res.per = 85.5
Printf("Percentage: %d\n", res.per);
```

⇒ Can use union inside struct.

⇒ Unions can be nested.

⇒ Arrays of Unions.

⇒ fn arg & return type in unions just like struct

⇒ pointers to unions.

⇒ Unions can be self referenced.

typedef

→ It is very useful in structures and unions.

```
typedef int num ;  // redefining int as num
num a,b;  <=>  int a;b;

Struct student S1;
typedef struct student stud ;  // user-defined datatype
                               redefining this as stud
Stud S1 // readable/ Crisp/ clean
```

Chapter – 13 Files

[13.1] Introduction to files :

Everything is created stored in the RAM

variable → RAM → As a program terminate, whatever is stored in RAM is lost
MM

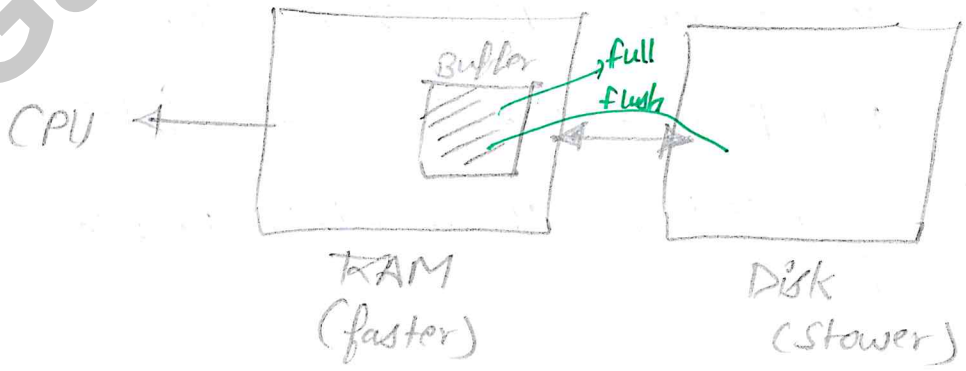
CPU ⟷ RAM ⟷ disk : Permanent storage of data
HDD
SSD (Solid state drive)
SD-card

There are two methods / modes of storage :

1. Text → human-readable, end of file (ASCII : 26) Ctrl+Z
   Portable (ASCII, Unicode) Ctrl+D
2. Binary → system (.dll, .exe)
   RAM → disk
   faster

Ex: (My name is reeliing) → f.txt

Buffer : Some space with in the RAM (temporary space)



abcdefghi\n[EOF] more efficient

as storing a then b --- if want to store 'l' as buffer is full. It will flush.

⇒ youtube video, which is not yet fully there. It is stored or buffered then you can watch

fflush() – file flush in stdio.h

∴ If something is half full. you can ask to flush it forcefully using fflush.

① Step1 ① opening
② read/write/appending
③ close

① opening : <stdio.h> (File) structure

↳ name, status, buffer size, current position

FILE *fp; typedef struct FILE;

fp = fopen("file1.txt", "w");

↳ name of file FILENAME_MAX

full path – C:\Document\file1.text

text

"w" – want to open my file in write mode.

a – append

r – read

"w+" – write + read (previous data is erased)

"r+" – read + write (not erased, append)

"a+" : append + read

Binary

"wb" ~ binary

"ab" — append mode binary file

ERRORS

fp = fopen ("file1", "w");

fp == NULL → error → No space on disk
No permission (Linux)
doesn't exist file r/a mode

Closing a FILE

int fclose (FILE *fp)
↳ all the file buffer has flushed → disk buffers are freed.
returns 0 (success)
−1 (EOF) stdio.h

Written by OS & Controlled by Program
↑ Error
↑ const

stdin → keyboard
stdout → monitor
stderr → monitor
} predefined const FILE * in stdio.h

int fcloseall (void) → all the file pointers in program are closed

```
main()
FILE *fp = fopen(—,—);
{
   ≡
   fclose(fp);   // all the buffers are flushed
}
```
data loss if don't write.

[13.2] FILE I/O


```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fptr; //creating file pointer
    int ch;
    if((fptr = fopen("test", "w"))==NULL)
    {
        printf("file doesnt exists\n");
        exit(1);
    }
    printf("enter text:\n");
    while((ch==getchar())!=EOF); //getting characters to keyboard
        fputc(ch, fptr);
        fclose(fptr);
        return 0;
    }
```

opening file test in write mode & assigning it to file pointer. if NULL then not exist

fputc takes parameter

Ctrl+Z or Ctrl+D

int fputc(int char, FILE *stream)

⇒ Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.

Parameters:

Char: Character to be written. This is passed as its int promotion.

Stream: pointer to a file object that identifies the stream where the character is to be written

Return value: → If there are no error, same characters that has been written is returned.

→ If an error occures, EOF is returned and error indicator is set.

```
#include <stdio.h>
int main()
{
  FILE *fp;
  int ch;
  fp = fopen("file.txt", "w+");
  for(ch=33; ch<=100; ch++){
        fputc(ch, fp);
  }
  fclose(fp);
  return 0;
}
```

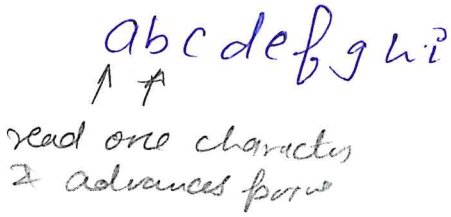
WAP to get the use of fgetc() get character from file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  FILE *p;
  char c;
  if((p=fopen("test","r"))==NULL)   → reading text file
  {
    printf("error in opening file\n");
        exit(1);
  }
while((c=fgetc(p))!=EOF)
    printf("%c",c);
    fclose(p);
    return 0;
}
```

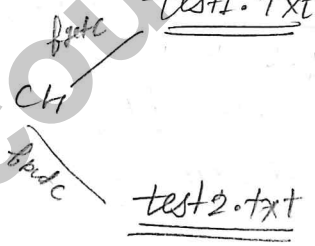
fgetc & fputc

int fgetc(FILE *Stream) : gets the next character (an unsigned char) from the specified stream

⇒ advances position indicator for the stream.



how fgetc and fputc can be used. ⇒ If want to copy one file to other.




```
int main()
{
FILE *sptr, *dptr;
Char ch;
if((sptr = fopen("test1.txt","r"))==NULL)
 {
  Printf("Error in opening source file\n");
   exit(1);
 }
 if((dptr = fopen("test2.txt","w"))==NULL)
 {
  Printf("Error in opening destination file\n");
    exit(1);
  }
 while((ch=fgetc(sptr)!=EOF)
     fputc(ch,dptr);
 fclose(sptr); fclose(dptr);
  return 0;
}
```

WAP to understand fputs (read strings)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  FILE *fptr;
  char str[80];
if((fptr=fopen("test","w"))==NULL)
{
 Printf("Error in opening file\n");
   exit(1);
 Printf("Enther the text\n");
Printf("to stop entering, press Cntrol+d/cntrl+Z\n");
 While(gets(str)!=NULL)
   fput(str,fptr);
   fclose(fptr);
   return 0;
}
```

reads from std i/p/keyb
Presenter - for going to next line

WAP to understand fgets

```
#include <stdio.h>
#include <stdlib.h>
 int main()
{
  FILE *fptr;
  Char   str[80];
  if((fptr=fopen("test","r")==NULL)
   {Printf("Error in opening file\n");
     exit(1);
   }
  While(fgets(str,80,fptr)!=NULL)
      Puts(str);
      fclose(fptr);
 return 0;
```

length of variable & file pointer
size at most 80

fgets

char *fgets(char *str, int n, FILE *stream)

⇒ Reads a line from the specified stream and stores it into the string pointed to by str.

It stops when either (n-1) characters are read, the new line character is read, or the end-of-file is reached, whichever comes first

fputs

int fputs(const char *str, FILE *stream)

Write a string to a specified stream up to but not including NULL character.

printf → stdout

scanf → stdin

fprintf → formatted file i/o
sprintf
sscanf

WAP to understand the use of fprintf()

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  FILE *fp;
  char name[10];
  int id;
  if((fp=fopen("test","w"))==NULL)
  {
    printf("Error in opening file\n");
    exit(1);
  }
  printf("Enter your name & id");
```

```
scanf("%s %d", name, &id);
fprintf(fp "My name is %s and id is %d", name, id);
fclose(fp);
return 0;
}
```

file pointer

fscanf

keyboard → scanf → var
file → fscanf → var
String → sscanf

```
#include <stdio.h>
#include <stdlib.h>
struct student{
        char name[20];
        float marks;
} stud;
int main()
{
  FILE *fp;
  if ((fp = fopen("test", "r")) == NULL)
    { printf("Error in opening file \n");
    exit(1);
  printf("NAME \t MARKS \n");
  while (fscanf(fp, "%s %f", stud.name, &stud.marks) != EOF)
  printf("%s \t %f \n", stud.name, stud.marks);
  fclose(fp);
  return 0;
}
```

Block input output

⇒ fwrite , fread

⇒ fwrite

This library function writes data from the array pointed to, by ptr to the given stream

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

(const void *ptr: array; size_t size: size of each array; FILE *stream: store it)

ptr: pointer to array of elements to be written.

size: size in bytes of each element to be written.

nmemb: No. of elements, each one of size of size bytes.

stream: pointer to a FILE object that specifies an output stream.

size_t : unsigned it

function returns total no. of elements successfully returned as size_t object

otherwise it will return error.

```
#include <stdio.h>
int main(){
FILE *fp;
Char str[] = "this is AAIC";
fp = fopen("file.txt", "w");
fwrite(str, 1, sizeof(str), fp);
fclose(fp);
, return 0.
```

fread : This C library function reads data from the given stream into the array pointed to by ptr.

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

(void * : Can point to any data type)

Ptr : pointer to block of mem/r with a min size of size * nmemb bytes.

size : size in bytes of each element to be read.

nmemb : This is number of elements, each one with a size of size bytes.

Stream : This is pointer to a FILE object specifies an input stream.

Block → array

fseek : This C library function sets the file position of the stream to the given offset.

int fseek(FILE *stream, long int offset, int whence)

stream : pointer to file object that identifies stream.

offset – No. of bytes to offset from whence.

whence : position from where offset is added. specified

SEEK_SET : Beginning of file

SEEK_CUR : Current position of file pointer

SEEK_END : End of file

func^n returns 0 if successful

Non-zero . if else

WAP for usage of fseek() function

```
#include <stdio.h>
int main()
{
FILE *fp;
fp=fopen("file.txt","w+");
fput("This is Appliedgate",fp);
fseek(fp,7,SEEK_SET);
fput("C programming lang",fp);
fclose(fp);
return 0;
}
```

wand to move 7 bytes ahead

O/P: This is C programming language

fseek helps to perform random access

abcdefgh

ftell

returns the current file position of the given stream.

```
long int ftell ( FILE *stream)
```

a b c d e f g h i
1 2 3 4 5 6 7 8 9 10

— returns 10

rewind :

Sets the position of file to the beginning of the file of the given stream.

```
void rewind (FILE *stream)
```

Ex:

abcdefgh ↑ → after rewind it will start pointing to start.

⇒ Moving the control with in file.

Remove

Deletes the given filename so that it is no longer accessible.

```
int remove ( const char *filename)
```

fflush() : This C library function flushes the output buffer of a stream.

```
int fflush(FILE *stream)
```

Stream - pointer to a file object that specifies a buffered stream.

Return : returns 0 if success

If error occurs, EOF is returned. error indicator is set (i.e. feof)

Chapter-15 Miscellaneous topics in C

[15.1] Enumerators : enum keyword in C.

⇒ Just like Struct, Union, enum is also a user-defined data-type.

Month Jan, Feb, March ....

(1) enum month { Jan, Feb, March, Apr, May }; // Defining (internally 0, 1, 2, 3, 4)
keyword identifier identifiers

enum month m1, m2; } Declaring a var
m1 = Jan (m1 = 0)

(2) enum month { Jan, feb = 4, March, Apr, May = 10 };
0 4 5 6 10

(3) printf("%s", m1); ✗ internally m1 is integer & here I am treating it as string

(4) enum set1 { abc, def, ghi };
enum set2 { abc, xyz, mno } ✗ Whatever you are using in one enum can not be used in the same scope.
int def = 0; ✗

⇒ internally helps to increase code readability.

[15.2] Function Pointers (Pointers to function)

⇒ code is also placed in the memory

each function → place/location in memory

address

```
int f()          |   int main()
{                |   {
 ≡               |    ≡
}                |   printf("%u", f);
                 |   }
```

fn name

address of the actual fn

Pointer to a function :

```
float func(int, int); // declaration of fun
float (*fp)(int, int); // fp is a pointer to a function
fp = func;             that takes 2 int argument
                       & returns a float
```

address of function

① `float a = func(10, 20);`

② `float a = (*fp)(10, 20);`

Pass a function address as an argument to another fn.

```
void f1(int);
void f(int, void(*fp)(int));
```

2nd arg is fun pointer, Pointing to a func with one int arg & return type as void

```
main()
{
  ---
  ===
f(10, f1);
}
```

↳ address of function f1

```
void f1(int a)
{
==
}
void f(int x, void (*f
Cint))
{
==
}
```

Array to function pointer

```
float (*fp[4])(float, float);
```

fp is an array of function pointer where each func has return type float & 2 float argument.

```
float add(float, float);
float sub(float, float);
float mul(float, float);
float div(float, float);
```

```
fp[0] = add;
fp[1] = sub;
fp[2] = mul;
fp[3] = div;
```

} ans

```
float (*fp[])(float, float) = {add, sub, mul, div};
```

Declaration — assignment

(1) int *f(void) → fn that returns int * & takes no args

(2) int (*fp)(void) → fn pointer

(3) int (*fp[4])(void) → array of fn pointer where each fn has no arg

④ int *(*fp)(void) → fn pointer to a fn that returns an integer ptr

⑤ int *(*fp[4])(int *a, int b)

↑ return type

array of fn pointer