→ Data processing [process of converting i/p to o/p with the help of software programs]. Memory representation:-

→ Computer work:

User
↓↑
Application
↓↑
Operating System
↓↑
Hardware

1. Text segment [read only segment]
2. Initialized data segment → global & static variables that are initialized
3. uninitialized data segment ↓ global & static variables that are not initialized.
4. stack [used to store local variables]
5. heap
↓
dynamic memory allocation

→ Program development cycle
→ Problem definition
→ Problem analysis
→ Algorithm development
→ Coding ~~debugging~~ & documentation
→ Testing & debugging
→ maintenance.

→ Flow chart:-
- ⬭ → start or stop
- ▱ → print statement
- ◇ → if condition.
- ▭ → rectangle

formulas

→ C - Dennis M. Ritchie
Bell laboratories, 1972
→ unix operating system → (which starts when you turn on computer) → It manages and allocates declarative system resources in a sufficient and secure manner.
→ named after B so it C.
→

Features:-
simple syntax
Easy to learn
easy to code
POP

Comment:-
/* - - → multiline
- - */
// - single line

Tokens:-
Smallest individual parts.
→ variables
→ datatypes
→ Identifers
→ constants
→ keywords
→ special symbols.

Source code
↓ compiler → compiler
Assembly source code → Assembler
↓ → Assembler converts assembly program to object
Object code
↓ Linker
executable code → It combines all object codes and execute
↓ loader
Execution

Types of variables:-
1. Camel case (Except first word all are capital)
2. Pascal case (Except All words start with capital)
3. Snake case
↓ separated by underscore

Variable:-
Memory location
it will change while execution, so it called as variable.
→ variables must be declared while before using.
→ If we don't declare variables then we get error.
→ Commas are not allowed
→ special symbols except underscore.
→ space also not allowed.

Identifers:-
→ An identifier can only have alphanumeric and underscore.
→ Identifer means name of (a-z), (A-z) are underscore
→ case sensitive in C.
→ 31 characters of an identifer.

Identifier and variable:-

→ identifier is a name used to identify a variable, function, array, structure or any other user defined item.

→ A variable storage location that holds a value.

Keywords C:-

32 keywords

Reserved words

Auto	Do	goto	signed	unsigned	
break	double	If	sizeof	void	
case	else	int	static	Volatile	
Char	enum	long	struct	while.	
const	extern	register	switch		
continue	float	return	typedef		
default	for	short	union		

Constants in C:-

→ Fixed values

→ Integer decimals

→ octal. [must start 0]

→ hexadecimal. [must start 0x or 0X]

→ Escape characters:-

We are not able to represent all the data in normal ways like ♡

→ so we use

printf("\"\U2665cats\"");

\n → new line

\r → Carriage return →

\v → vertical tab

\t → tab

\f → formfeed → move to nextline

\a → alert (beepsound)

\b → backspace.

formfeed:-

form feed

vertical tab

form feed

?! → question mark.

octal:-

#include<stdio.h>

printf("\072");

\uhhhh or \Uhhhhh

Constants:-

→ #define max 10 → It is better to take const variable in capital.

→ const

Data types:-

→ size qualifier [short, long]

→ Sign qualifer (Consigned, signed)

4-types:-

→ primitive (or) primary (or) Built in data type.

→ Non-primitive or user defined or secondary.

→ void.

→ Enum

primitive data types:-

Int [16bits or 2bytes]

float [4bytes]

Char [1byte]

Long int:-

32bytes or 4bytes

unsigned int:-

0. to 65535

short int:-

1byte.

unsigned short int:-

8bits or 1byte.

Long int:-

32byte or 4byte

unsigned long int

Float

double

long double

#include<stdio.h>

int main()

{

printf("hello\rworld");

return 0;

}

Null character:-

printf("unstoppable\0 continued");

I am unstoppable → After this everything will remove.

Non-primitive:
Structures, pointers, arrays, strings.
These are derived from the primitive data types.

enum:-
It is a key word,
syntax:-
enum enum-name{value1, value2, --}
→ If we want to give we can specify value by 1. Also.
↙ by default value starts with 0, and increment

Type conversion or Type casting:-
Implicit[automatically]
explicit[we have to mention]
Implicit:- lower data type is converted to higher data type.
Explicit:- higher data type is converted into lower. here we have to mention, because data loss will happen.
(type_name) expression.

Storage classes:-
Every variable must have <storage class datatype

→ storage class explains how long it stays in existence.
→ they are used to determine lifetime, visibility, memory location and initial values.
→ Lifetime means time between creation and destruction
4 types → Auto, static, register, extern

Auto:-
visibility - within block
same as local variable
auto data-type variable_nam

→ there no need to mention auto, storage
→ 3. Register:- place - RAM.
Register:- Instead of RAM these are stored in register.
register data-type variable_name;
→ definitely not a rule that they will be stored in registers, if place is not there then this or get stored in RAM and behaves as same as local variable.
→ used for faster access like counters.
Maximum size of variable = Maximum size of register.

External or global:-
→ stored in RAM
→ until the program ends lifetime will be there.
→ declared using extern keyword.
→ we can access extern variables in other files also
→ This is used whenever we have multiple numbers of files. same variable will be used in multiple files then we will declare global variables.

Static:-
→ local static variables.
→ Global static variables.

operators
→ 3-types { unary, Binary, Ternary

Unary operators:- which acts upon operands.
→ unary operator:- which is having only single operand
Binary operators:- which acts upon 2-operands.
Ternary:- which acts upon 3-operands.
→ Arithematic (+, -, *, /, %)
→ Relational (==, =, >, <, <=, >=)
→ Shift operator (left shif or right shift)
→ logical operator [&&, ||, !]
→ Bitwise [&, |, ^, ~)
→ increment or decrement { post → a++, pre → ++a
→ ~ → complement
-(n+1) ⇒ ~6 ⇒ -(6+1) ⇒ -7

& → 1000 & 0100 = 0000
| → 1000 | 0100 = 1100
^ → 00001100 ^ 00011001 = 00010101

Ternary or conditional operator:-
1. ?
2. :
Condition ? exp1 : exp2;
true
False
If condition is true exp1 evaluvates
condition2 is false exp2 evaluvates

Miscellanous:-
comma, sizeof,
↓ It

Increment or decrement
→ We can't use increment on constant variables.
→ whenever more than one format specifier is directly or indirectly related to same variable, then we need to evaluate the expression from right to left.

```
#include<stdio.h>
int main() {
int i=1;
Printf("%d %d %d", i, ++i, ++i); }
```

3 3 1

Control statements:-
→ if
→ if else
→ else if
→ Nested if

switch:-
```
Switch(expression)
  case value1:
    break;
  Case value2:
    break;
```

Rules:-
→ Switch expression must be of an integer or character.
→ Break statement in switch Case is not must. If there is no break statement, found in the case, all the cases will be executed between pres then fall through.

Loops:-
→ Entry Controlled [do, while]
→ Exit Controlled [while, for]

```
do {
  Statement; increment; }
while (Condition);
```

Jump or loop control or unconditional statements:-

→ Break → If we want to terminate program.
→ Continue → If we want to skip any iteration
→ Goto → Forward branching
→ Backward branching

Functions (or) Module (or) subroutines:-

finite set of instructions that together perform a task.

1. Function declaration.
2. Function Call
3. Function definition

Advantage:- Code reusability

function declaration:-

Return-type function_name (argument list);

```
int add(int a, int b);
add(10,40);
int add(int 10, int 40){
return result; }
```

2 types:-
→ user-defined
→ predefined

4 types:-
→ function with arguments and with return value.
→ function without argument and without return value.
→ function with arguments and without return value.
→ function without arguments and function without return value.

Call by parameters:-

→ When we change the values in formal parameters, then it will effect the actual parameters are called as call by reference.
→ Here address is passing.

function without return type:-

```
int function_name(int);
function_name(a);
int function_name(int a)
{
statements;
return type;
}
```

→ parameters are not important in function declaration, but their type is important.

→ function with no arguments no return type:-

```
void function_name();
function_name();
void function_name(){
Body of function }
```

→ function with no arguments & return value

```
int function_name();
function_name();
int function_name(){
return value; }
```

function with arguments & no return value

```
void function_name(int);
function_name(a);
void function(int a){
= }
```

parameters:-
→ Actual → which are declared in function call
→ Formal → which are declared in function definition

Call by value:- (formal parameters)
If we change values and it will not effect the actual parameters is called by call by value.

→ Here in call by value the values get stored in stack, after creating or after out of the loop it will destroy the loop.
→ Whereas here we will pass address.
→ If you change the value of function parameter, it is changed for the current function only.
→ Actual parameters:-
Constants, variables or expressions.
→ Formal parameters:-
→ variables
→ expressions & constants not be allowed.

→ Arrays:-
Array is collection of similar data-type and stored in continous memory.
→ lowest address corresponds to first element and highest address last element.
→ Array of characters is string.
→ Each element (ID) data item in array is called element.
→ Each of elements have share variable name but different locations
→ It can be used to represent multiple data structures like linked lists, stacks, queues, ~~sorting~~ trees, graphs etc.
→ We must know in advance how many elements we have to store
→ Static
→ Fixed size.
#define size 10;
→ No. of elements are less than size, remaining all put to zero.
→ No. of element more than size, then will show error.
→

→ single dimension
→ Two dimensional
→ multi dimensional.

Single:-
→ single subscript
→ linear form
No of bytes required for single dimension =
size of datatype * size of subscript

Two-dimensional:-
→ 2 subscripts
→ matrix
→ No. of bytes required = No. of rows * no. of columns * size of data type.

Structures:-
→ Structures used to store for the different data types.
→ Struct tagname { → tag
datatype member1; → members
Datatype member2;
Datatype member3;
};
Struct student {
char name[64];
Char course[12];
int age;
int year;
};
→ Each variable declared in Structure is member.
→ Name given to structure is called tag.
→ Like primary variables structure variables can also be initialed when they are declared.
→ Structure templates locally or globally.
→ is having less than values automatically set to

s1.name[];
s1->rollno;

→ By using sizeof operator we can find size.
→ sizeof(s1);
→ For accessing members of structures we use .
→ For accessing members of union same we use .
→ For accessing pointers in structures we use ->
→ For accessing pointers in union we use ->

Unions:-

Unions are same as structures, but here memory allocating is different.
→ The size of the union will be the size of the largest datatype.

Strings:-

→ group of characters, here we don't have any string datatype.
→ char name[10];
char b[9]={'c', 'o', 'm', 'p', 'u', 't', 'e', 'r'};
char b[]="computer"

Library functions:-

→ strlen → strcat.
→ strcpy (string1, string2)
→ strlwr
→ strupr
→ strcmp (str1, str2)
→ strrev
(String1, string2)
↓ source string
↓
Here string1 is destination

Strcmp:-

str1 == str2 → returns 0
str1 > str2 returns 1

→ Here we are not random value.

Typedef:-

→ It is a keyword that allows the programmer to create a new data type name for the existing data type.
→ aliasing name. It is useful mainly in

Bitfields:-

→ Here the length or width must be less than or equal the size of datatype.
→ Reduces memory consumption.

Errors:-

→ syntax error → easily corrected or debugged.
→ Run-time error.
→ Linker error
→ Logical error
→ Semantic error.

int a;
Int a;

→ syntax errors occured due to mistakes while typing or do not follow the syntax of the specified language.

Run time errors:-

→ Run-time errors we are not able to find in compile time
→ like 2/0.

Linker error:-

It occurs when executable file of the program is not created.

logical errors:-

Even if the syntax and other factors are correct, we may not get the desired results due to logical issues.

Semantic error:-

when a sentence is syntatically correct, but has no meaning semantic error occur. this is similar to grammatical errors. i.e if an expression is entered on left side of the assignment.

operator semantic error occurs

for(i=0; i<=5; i++);
↓
Here it has to loop 5-times, but only once it will create, because of semicolon.

i/p and o/p functions in C:-

1. Formatted
2. Unformatted

Formatted:-

It represents all data type

Unformatted:- printf(), scanf()

It represents only character & string

printf() → o/p data

scanf() → Take i/p data

Unformated:-

Characters:-

→ getchar() → used to read a character from the standard input

→ putchar() → used to display a character to standard output.

→ getch()

→ putch()

Strings:-

→ gets() → used for accepting any string from standard input

→ puts() → used to display a string or character array

→ getch() → read and display the character

→ getch() → only read the single character but not display

→ cgets() → read a string from the console

→ cputs() → display the string to the console

getchar():-

single character given to computer using input library.

char variable = getchar();

↓

It reads single character.

→ It does not take any arguments, but we have to given ()

getchar()

```
Char ch;
printf("Enter any character");
ch=getchar();
if(isalpha(ch)>0)
printf("It is a alphabet %c\n", ch);
else if(isdigit(ch)>0)
printf("It is a digit %c\n", ch);
```

Enter digit: abc => a.

putchar():-

used to display one character at a time on the standard o/p device. This function does the reverse operation of single character input

putchar(character variable)

```
char ch;
printf("Enter any alphabet in lower or uppercase");
ch=getchar();
if(islower(ch))
putchar(toupper(ch));
else
putchar(tolower(ch));
getch();
```

gets()

→ Group of characters

→ gets(char type of array variable)

```
printf("Enter string name");
gets(str);
```

puts():- used to display string

puts(char type of array variable)

```
puts("Enter String");
gets(str);
```

dynamic memory allocation:-

→ memory will be allocated at the run time.
→ these are used for allocating and freeing memory.
→ defined in stdlib.h.
→ malloc() (single)
calloc() [multiple]
realloc() [reallocates]
free() [free]
→ memory allocated at the compile time. [static]
→ allocates at run time.
→ heap is used to store dynamic memory, during execution the heap changes.
→ malloc()
memory allocation
single block of requested memory at runtime.
→ and returns pointer type of void
→ It is having garbage value initially()
→ If it fails to allocate enough space then it will return a null pointer.

```
ptr=(cast_type*)malloc(byte-size)
x=(int x) malloc(100*sizeof(int));
free(x)
```

↓
After allocating definitely we have to clear the space using free.
→ It is used for structures.

```
#include<stdlib.h>
int main()
{
int num,i,*ptr,sum=0;
printf("Enter no-of elements");
scanf("%d",&n);
ptr=(int*)malloc(num*sizeof(int));
if(ptr==NULL){
printf("Error; memory not
allocated");
exit(0);
}
printf("Enter elements of
array");
for(i=0;i<num;i++)
{
scanf("%d",ptr+i);
sum=sum+*(ptr+i);
}
printf("sum=%d",sum);
free(ptr);
return 0;
}
```

→ malloc is used to dynamically allocate a single large block of contigous memory according to size.
→ malloc doesn't have an idea of what it is pointing to.
→ It merely allocates memory requested by the user without knowing the type of data to be stored inside the memory.
int *ptr=(int*) malloc(4)
malloc allocates 4 bytes of memory in heap and address of the first byte is stored in the pointer ptr.

Calloc :-

→ Continous memory allocation.
→ Mainly used for arrays.
→ Allocates multiple block of request memory.
→ If it fails to locate enough space it will give error.
→ Sets values to zero by default.

```
ptr=(cast_type*) calloc (number,
        element_size)
```

Calloc → requires 2 arguments
- count → provide no. of elements
- size type → data type size.

→ memory allocated by calloc is initialized to zero
→ whereas malloc will allocate some garbage value
→ calloc → clear allocation.
→ used to create continous memory of arrays

```
int *arr
arr = (int*) calloc(10, sizeof(int));
char *str;
str = (char*) calloc(50, sizeof(char));
```

```
#include<stdlib.h>
int main(){
int num, i, *ptr, sum=0;
Printf("no. of elements");
Scanf("%d", &num);
ptr=(int*)calloc(num, sizeof(int));
if(ptr==NULL){
    Printf("error");
    exit(0);
}
printf("Elements of array");
for(i=0; i<num; ++i){
    Scanf("%d", ptr+i);
    sum=sum+*(ptr+i);
}
printf("sum=%d", sum);
free(ptr);
return 0;
}
```

realloc()

changes memory size that is already located.

→ Previous memory space is insufficient or more than required you can change the previously allocated memory size using realloc()

→ Initial value is garabage.

→ 2 types of arguments

```
int *ptr, i, n1, n2;
printf("Enter the size of array");
scanf("%d", &n1);
ptr = (int*) malloc(n1 * sizeof(int));
printf("Address of previously allocated memory");
for(i=0; i<n1; ++i)
    printf("%u\t", ptr+i);
printf("Enter new size of array");
scanf("%d", &n2);
ptr = realloc(ptr, n2);
for(i=0; i<n2; i++)
    printf("%u\t", ptr+i);
return 0;
}
```

malloc():-

```
#include<stdio.h>
#include<stdlib.h>
int main() {
  int i, n;
  printf("Enter the number of integers");
  scanf("%d", &n);
  int *ptr = (int*) malloc(n * sizeof(int));
  if(ptr == NULL) {
    printf("memory not available");
    exit(1); }
  for(i=0; i<n; i++) {
    printf("Enter an integer");
    scanf("%d", ptr+i);
  }
  for(i=0; i<n; i++) {
    printf("%d", *(ptr+i));
  return 0;
  }
```

→ used to change the size of the memory block without losing the old data.

→ on failure realloc returns NULL

```
ptr = (int*) realloc(ptr, 2*sizeof(int));
```

→ And this function moves the contents of the old blocks to a new blocks, data is not loss

(this space will allocate)

```
#include<stdio.h>
#include<conio.h>
void main() {
FILE *ptr;
```

free():-

Memory allocated in heap despite released automatically ~~the~~ after using memory. The space remains there can't be used.

Volatile:-

→ Volatile ensures immediate visibility across threads.

→ volatile does not guarantee atomicity of compound operations.

→ This informs the compiler that the variable should not be optimized or cached in registers, as the value might be modified externally such as by hardware, other threads

→ It is our responsibility to inform the compiler that the value of a particular variable may change through some external process also.

→ We are instructing the compiler to turn off the optimization process for that variable i.e we are forcing compiler to read the value of that variable from memory only, each time it is encountered in program.

getche():- reads a single character directly from screen without echoing. [for password purpose we will use] → show or display
→ getch means here it will be like that until we enter any character [here tab also it will take as a character]
→ If we enter any character any character it will go to program.
→ If we don't use getche then the program o/p will display once and it will automatically close.

getc:-
used to accept single character from string

putc:-
This function is used to display a single character in a character variable to standard o/p device.
putc(character variable);

return 0:-
It will tell the that the program has completed all its task and terminated succesfully and the hardware resources that has been allocated and now free and allocate other
→ Here in place of 0 we can place anything either positive or negative but zero will send a successful and error termination message.

ceil()
a=3.4
ceil(a) ⇒ 4.0000
floor(a) ; 3.00

Infinite loop:-
for(;;)
↓ ctrl + C

Pointers:-
→ pointer a variable that stores address of another variable of same data type.
→ It is a derived datatype.
→ It supports dynamic memory location.
→ It reduces length and program execution of time.

Null pointer:-
→ It is is always a good practice to assign a NULL value to pointer variable in case you do not have exact address
→ done at the time of compilation process.
→ A pointer that is assigned NULL is called NULL pointer.
data-type *pointer_variable -name;

int *p;
int* p → means here address belong to the datatype int.
int *ptr;
ptr = &a; → declaration
→ Initilization
int *ptr = &a;
→ normal variables store values whereas pointer a variable point to address of variable.
& → get address
* → used to get value.
We can able to perform subtraction on pointers, we are not able to perform Addition, multiplication

divisions.

```
int a, *p;
a=10
p=&a;
printf("%d", *p)→value
printf("%u", &a)→address
printf("%u", p);→ print
                address
printf("%d", *& a);→ points value
```

Move to next location:-

```
#include<stdio.h>
Const int MAX=3;
int main() {
int var[]={10, 100, 200};
int i, *ptr;
ptr= var;
for(i=0;i<max;i++)
   {
printf("Address of var
      %d = %x\n", i, ptr);
printf("value of %d = %d\n", i, *ptr);
ptr++;
  }
```

Decrementing:-

```
ptr= &var[MAX-1];
for(i=MAX;i>0;i--)
   {
printf("%d = %x", i, ptr);
Printf("%d=%d", i, *ptr);
ptr--;
  }
```

fseek(fp,0L,0); → beginning file
(same like rewind);
fseek(fp,0L,1): → current poistion
↓ rarely used.
fseek(fp,0L,2); → go to end of file.

rewind(): - puts or places the file pointer to the beginning of file, irrespective of where it is present right-now.

rewind(fp); → file pointer as an argument.

ftell(): - returns current file poistion of specified stream. we can use ftell(), to get total size of a file after moving file pointer to end of file.

n=ftell(fp)
→ n would give reltive offset (in bytes)

Charater I/o funtions:-

fputc:- used to write a charater in to the file.

getc():- used to a read charater from a file that has been opened in read mode.

open():- used to create a new file or to open existing file.

close():- is used to close already opened file.

fopen=
FILE *p=fopen("location", "mode");
FILE *fp=fopen("c:\\rob.", "r");
fputc(variable/value, filepointer);
fputc('P', ptr);

```
#include<conio.h>
int main() {
  FILE *ptr;
  char ch='P';
ptr=fopen("c:\\robo\\xyz.txt",
      "w");
if(ptr==NULL) {
 printf("File unable to
     open"); }
else {
 printf("File opened
     sucessfully");
  fputc(ch, ptr);
fputc('a', ptr);
fputc('b', ptr);
 fputc('\0', ptr));
 fclose(ptr);
  getch();
}
```

fgetc():-

```
#include<stdio.h>
#include<conio.h>
int main() {
 FILE *ptr;
 char ch;
ptr=fopen("c:\\robo\\xyz.
   txt", "r");
if(ptr==NULL) {
 printf("File unable to
   open"); }
else {
 printf("File content is:");
 while((ch=fgetc(ptr))!
     =EOF) {
 printf("%c", ch);
}
fclose(ptr);
 getch(); }
```

fputs():- ~~writes a line~~ function writes a line of characters into file.

→ line of characters into file.

```
#include <stdio.h>
int main(){
 FILE *fp;
fp=fopen("myfile.txt","w");
fputs("hello C programming",fp);
fclose(fp);
}
```

fgets():- reads a line of characters from file.

```
#include <stdio.h>
int main(){
 FILE *fp;
 char text[300];
 fp=fopen("myfile2.txt","r");
 printf("%s",fgets(text,200,fp)); fclose(fp);
 getch(); }
```

getw and putw():- used for integer values, useful to deal only integer data. ⇒ putw(integer, fp);

```
int x=5;
putw(x,fp); → getw(); → getting or reading integer value from file
x=getw(fp);
```

File poisting functions:-

fseek():- used to set the file pointer to the specified offset. used to write data to a specified location.

fseek(FILE *stream, → pointer to file
long int offset, int whence)
↓ position of record to found. ↓ where the offset starts

fseek(file pointer, offset, poistion);
↓ number or variable of type long.
if offset +ve → moves forward, ~~else~~ and negative means move backwards
position can take one of 3 values:- 0, 1, 2
beginning ← 0 ↓ 1 current → 2 end

set ~~end~~ starts offset from beginning of file

end starts offset from seek_cur → current location of

→ When we write program and when we terminate entire data. Will be lost storing data in file will reserve your data. Even if we terminates the program.
→ If we lost, if we have to type all the data it takes lot of time.
→ we can access the data, you can easily access the contents of file using few commands in C.
→ we can easily move data from one computer to other.

File i/p & o/p
formatted:-
fscanf()
fprintf()
unformatted:-
i/p functions:- gutc(), getw(), fread()
o/p functions:- putc(), putw(), fwrite().

→ Files is a collection of data, that is stored on secondary devices like hard disk.

2 types → text files
→ Binary files.

Text files:-
→ easily understanble
→ easily access
→ text format, minimum effort, less security

Binary files:-
→ .bin files
→ more security
→ not able to access easily
→ not understandble
→ more security,

File operations:-
→ Naming an existing file
→ opening an existing file
→ reading
→ writing
→ close

→ Declaring a file:-
→ Declare file
→ Declare file pointer
→ file using fopen function
→ process
→ close

Declaration:-
~~when~~ we need to work with a file, we need declare as pointer,

```
file *fp;
fp=fopen("fileopen", "r");
fp=fopen("e-programming.txt", "r")
```

Closing:-
fclose()
fclose(ptr);

r → opens a text file in read mode
w → open this in write mode
a → append
r+ → both write and read
w+ ~~both~~ but read first
w+ both write and read → but write first
a+ → both r and w.
rb → opens a binary file in read mode.
wb → opens a binary file in write mode

wb+ → both write and read
ab+ → both write and read

difference b/w v write and append mode:-

→ Both files a new file is created when existing not open.
→ when we give append the cursor will move to end of file.
→ write will erase the whatever there and again it starts.

```
fprintf(fp, "control string", list);
fprintf(fp, "%s", name, age);
fscanf(fp, "control string, list);
fscanf(fp, "%s%d", &name, &age);
```

↓ used to read list of items from a file.

Used to write a list of items to a file.

EOF → End of File

```
#include<stdio.h>
{
FILE *fp;
fp=fopen("file.txt","w");
fprintf("Hello file by
        fprintf");
fclose(fp);
}
```

2)
```
#include<stdio.h>
int main(){
  int num;
  FILE *fptr;
  fptr=fopen("C:\\program.txt","w");
  if(fptr==NULL){
    printf("Error");
    exit();
  }
  printf("Enter the num");
  scanf("%d",&num);
  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

putc:-

Used to a character in to file. only works character data type

```
putc(ch,fp);
fputc('a',fp);
```
→ file
↓ character

fgetc()

→ returns a single character from file.

```
char ch;
ch=getc(fp);
```

```
#include<stdio.h>
int main(){
 FILE *fp;
 char c;
 clrscr();
fp=fopen("myfile.txt","r");
while((c=fgetc(fp))!=EOF){
  printf("%c",c);
}
fclose(fp);
getch();
}
```