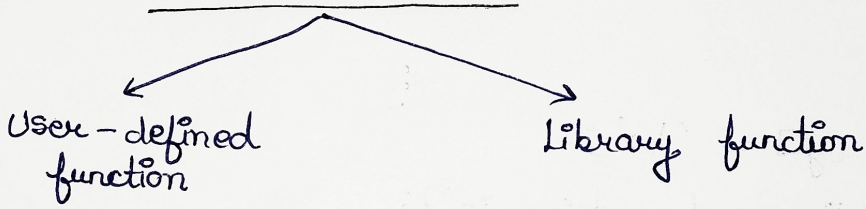
→ Strcat (str1 , str2) :-

⇒
```
# include <string.h>
# include <stdio.h>
Void main()
{
char Str1 = "Hello";
char Str2 = "Arpit";
Strcat (Str1, Str2);
printf (" strings after concatenation is :%s", str1);
}
```

Functions...



⇒ A function is a group of statements that performs a specific task and is relatively independent of the remaining code.

Function are used to organise program into smaller and independent unit.

→ Advantages :-

⇒ 1. Reduction in code Redundancy.
2. Enable code reuse.
3. Better readability.
4. Information hiding.
5. Improved debugging and Testing.
6. Program becomes easy to manage.

→ Difference b/w user-defined function and library function.

User-defined function

=> i) It is defined by the user according to the requirements of the application to be designed.

ii) User-defined function should be declared defined and called in order to be used in the program.

iii) For a user-defined functions programmer has to right the complete logic of that function before using it.

iv) Function name of a user defined function can easily be changed in the program

v) User-defined function is part of the 'C' language.

vi) Using more user-defined functions increases program complexity

Library function

i) Library functions are pre-defined in a header file and can be used in the program as per requirements of the application to be designed.

ii) These functions can simply be used by including respective header file in the program.

iii) E.g => Strcmp (str1, str2) can be used by including string.h in the program.

iv) Programmer does not need to write any logic related to the function.

v) Function name of library function cannot be modified by the user.

vi) Library function is part of the 'C' header file.

vii) Library functions make it simple and easy to design the program.

→ Function without any output / without any returntype / void function.

→ Syntax for function declaration:-

⇒ returntype functionname (datatype 1 [argument 1], datatype 2 [argument 2], ----------, datatype n [argument n]);

→ Syntax for function definition:-

⇒ returntype functionname (datatype 1 argument 1, ----, datatype n argument n)

{

// Function body

}

→ Syntax for function call:-

⇒ functionname (argument 1, argument 2, ----, argument n);

• Example of a Function with no input and no output / No arguments and no returntype / Void return type.

```
#include <stdio.h>
Void pattern()
{
 pattern();
 pattern();
 pattern();
}
Void pattern()
{
int i,j;
for(i=1; i<10; i++)
{
for(j=1; j<10; j++)
```

```
{
if (i==1 || i==9 || j==1 || j==9)
printf ("*");
}
printf ("\n");
}
}
```

- Example of a Function with one input and no output / no output / no arguments and no returntype / Void return type :-

```
=> #include <stdio.h>
Void pattern (int);
main()
{
pattern (5);
pattern (4);
pattern (1);
}
int i,j;
for (i=1; i<=x; i++)
{
for (j=1; j<10; j++)
{
if (i==1 || i==9 || j==1 || j==9)
printf ("*");
}
printf ("*");
}
}
```

```
* * * * * * * * *
*               *
*               *
*               *
*               *
*               *
*               *
*               *
* * * * * * * * *
```

- Example of a function with inputs and no output / no arguments and no returntype / Void return type :-

```
=># include <stdio.h>
Void patter (int , int , char);
main ()
{
int , rows, columns ;
pattern (5,4, '#');
printf ("Enter number of rows and columns");
scanf ("%d %d", &rows, &columns);
printf (" Enter symbol");
scanf ("%c", &ch);
pattern ( rows, columns, ch);
}
Void pattern (int r , int c , char s)
{
int i,j ;
for (i=1 ; i<=r ; i++)
{
for (j=1 ; j<=c ; j++)
{
if (i==1 || i==9 || j==1 || j==9)
printf ("%c",s);
}
printf ("\n");
}
}
```

→ Swap two integers :-

```
=># include <stdio.h>
Void swap (int , int);
main ()
{
```

```
int x, y;
printf ("Enter two numbers");
scanf ("%d %d", &x, &y);
swap (x, y);
swap (30, 101);
}
Void swap()
{
```

Functions :-

1. WAP to find whether a no. is even or odd using function

```
#include <stdio.h>
Void function (int);
Void main()
{
int a;
printf ("Enter a number");
scanf ("%d", &a);
function (a);
}
Void function (int a)
{
if (a%2==0)
{
printf ("%d is even", a);
}
else
{
printf ("%d is odd", a);
}
}
```

2. WAP to find factorial of a number using function :-

```
=># Include <stdio.h>
  Void fact (int);
  Void main ()
 {
 int a, f;
printf ("Enter a value");
scanf ("%d", &a);
 f = fact (a);
printf ("%d", f);
}
int fact (int a)
{
int i, f;
for (i=a; i=>1, i--);
f= f * i;
return f;
}
```

Recursion...

=> When a function calls itself, it is recursion

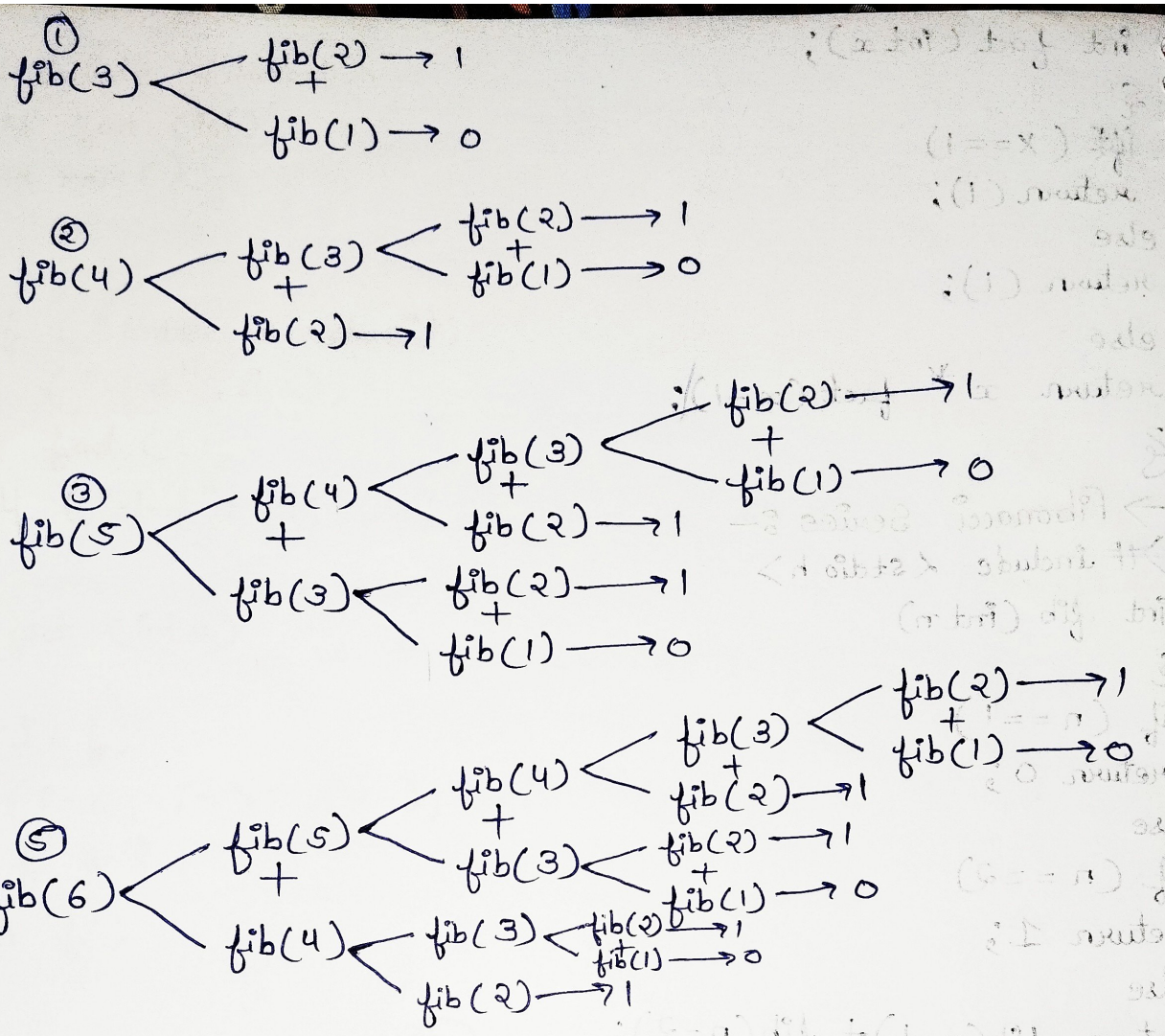
→ Example :- Factorial

```
=># include <stdio.h>
  int fact (int);
  Void main ()
 {
 int a, f;
printf ("Enter a number");
scanf ("%d", &a);
 f = fact (a);
printf ("%d", f);
}
```

```
int fact (int x);
{
if (x==1)
return (1);
else
return (1);
else
return x * fact (x-1);
}
```

→ Fibonacci Series :-

```
#include <stdio.h>
int fib (int n)
{
if (n==1)
return 0;
else
if (n==2)
return 1;
else
return fib(n-1)+ fib(n-2);
}
Void main ()
{
int a, i, f;
printf ("Enter a number");
scanf (".%d", &a);
for (i=1 ; i<=a ; i++)
{
f = fib (i);
printf (".%d", f);
}
```

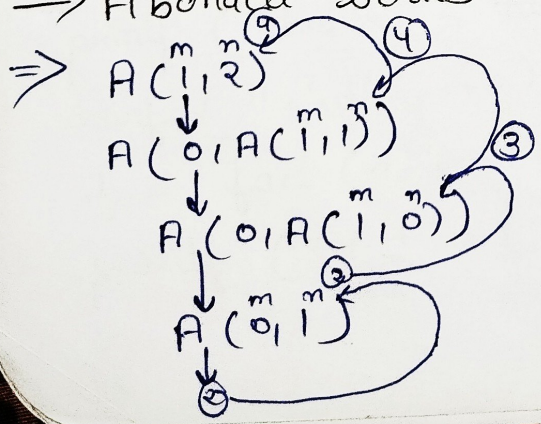


⇒ Ackerman's Function :-

$$\Rightarrow A(m,n) = \begin{cases} n+1 & , \text{ if } m=0 \\ A(m-1,1) & , \text{ if } m>0 \text{ and } n=0 \\ A(m-1), A(m,n-1) & , \text{ if } m>0 \text{ and } n>0 \end{cases}$$

(where m and n are non-negative numbers)

→ Fibonacci Series :-



```
#include <stdio.h>
int A (int, int)
Void main()
{
int m, n, sol;
printf ("Enter two numbers");
scanf ("%d %d", &m, &n);
sol = A(m,n);
printf = ("%d", sol);
}
int A (int m,n)
{
if (m==0)
 return n+1
else
if (m>0 && n==0)
 return A(m-1, 1)
else
 (m>0 && n>0)
return A(m-1, A(m-1), A(m-1, A(m,n-1))
}
```

30/11/23

☆ Base and power :-

```
=>#include <stdio.h>
long int pow (int, int);
Void main()
{
int b, p;
printf (" Enter base value");
scanf ("%d", &b);
printf (" Enter power");
scanf ("%d", &p);
printf ("%d raised to power %d is %ld", b, p, pow(b,p));
```

actual arguments

```
long int pow (int x, int y)        formal arguments
{
long int r = 1;
for (int i = 1; i<=y ; i++)
r = r * x;
return r;
}
```

★ **Argument :-**

=> Argument is basically a value in the form of variable, constant or expression which can be passed to the function as input so that by working on this input function may perform the desired task.

1. **Formal argument :-**

=> It is a variable or expression mentioned in function definition. Scope of formal argument is limited to the function definition only.

=> Datatype of formal arguments must be mentioned while defining the function.

2. **Actual arguments :-**

=> It is the variable, constant or an expression contained in a function call that replaces the formal argument which is a part of function definition. It is not required to mention datatype of actual arguments during function call.

★ **Return statements :-**

=> Return statement returns the control to the calling function after ending the execution of the function. In the calling function execution resumes at the point immediately following the call. Return statement may

return a value to the calling function based on the return type of the function in which this return statement is return.

If no return statement appear in a function definition control automatically returns to the calling function after the last statement of the called function is executed.

★ Recursive functions :-

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

fact(n) - n * fact(n-1)

↓

n-1 * fact(n-2)

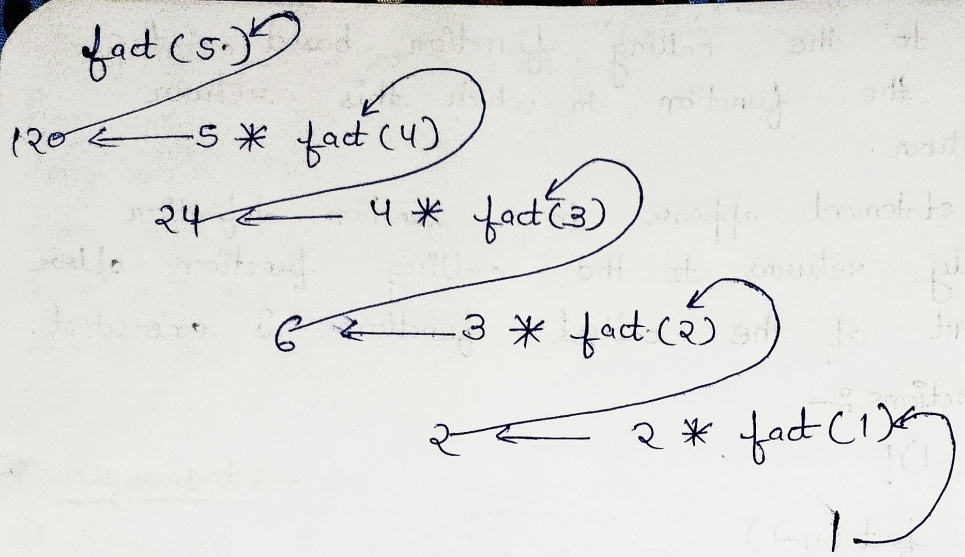
Sum(n) = n + Sum(n-1)

Sum of digits,

Sum of digits(n) = (n%10 + sum of digits(n/10))

Example :-

```
=> # include <stdio.h>
long int fact (int);
Void main ( )
{
printf ("Eactorial of 9 is %ld", fact (9));
}
long int fact (int n)
{
long int f;
if ((n==0||n==1))
return 1;
else
f = n * fact (n-1);
return f;
}
```



★ Recursive function / recursion :-

⇒ A function which calls itself in its function definition to solve any problem is called a recursive function and this technique of solving problem using recursive function is called recursion.

Example :-

A(2, 2)

↓

A(1) , A(2, 1)

↓

A(1 , (2, 0)) = 3

↓

A(1, 1) = 3

↓

A(0), A(1, 0) = 3

↓

A(0, 1) = 2

↓

2

A(1, 3) = 5

|

A(0, A(1, 2)) = 5

|

A(0, A(1, 1)) = 4

|

A(0, A(1, 0)) = 3

|

A(0, 1) = 2

A(1, 5) = 7

|

A(0, A(1, 4)) = 7

|

A(0, A(1, 3)) = 6

|

A(0, A(1, 2)) = 5

|

A(0, A(1, 1)) = 4

|

A(0, A(1, 0)) = 2

|

A(0, 1) = 2

★ Ackermann :-

```
=> # include <stdio.h>
int ackermann (int , int );
Void main ()
{
int x , y;
printf (" Enter two numbers" );
scanf ("%d%d", &x ,&y );
if (x>=0 && y>=0)
{
printf (" Ackermann function for %d and %d gives %d", x, y,
        ackermann (x, y));
}
int ackermann (int m , int n)
{
if (m==0)
return n+1;
else if (m>0 &&n=0)
return ackermann (m-1 , 1)
else if (m>0 && n>0)
return ackermann (m-1 , ackermann (m, n-1));
}
```

★ Sum of digits of a number using recursion :-

```
=> # include <stdio.h>
int sum of digits (int);
Void main ()
{
int x;
printf ("Sum of digits of 3481 is %d", sum of digits (3481));
printf ("Enter a number");
scanf ("%d", &x);
printf (" Sum of digits of %d is %d", x , sum of digits (x));
```

```
int sum of digits (int n)
{
if (n==0)
return 0;
else
return (n%10) + Sum of digits (n/10);
}
}
```

★ Local Variable & Global variable :-

=>
```
#include <stdio.h>
Void display();
Void main()
{
int x = 10; // local Variable
display ();
}
Void display ()
{
x++;
printf("%d", x);
}
```
(45)

```
# include <stdio.h>
int x; // Global Variable
Void display ();
Void main()
{
x = 10;
display ();
}
Void display ()
{
x++;
printf("%d", x);
}
```
(11)

→ Based on their scope variables are classified as local and global variable :-

=> Global variables are declared outside of all the functions in the program and can be accessed globally in the entire program.

Local variable is declared its size and specific function and can be accessed only within the function in which it is declared.

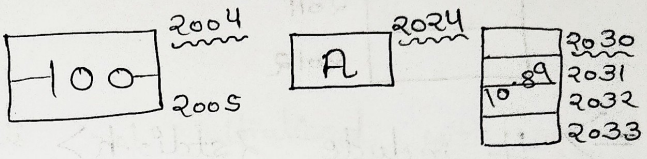
Pointers...

⇒ Pointer is that which stores memory address of another variables.

→ Declare of a pointer variable :-

⇒ < Datatype of variable to which it point > *pointername;

```
int *ptr;
char *ptr1;
float *ptr2;
```



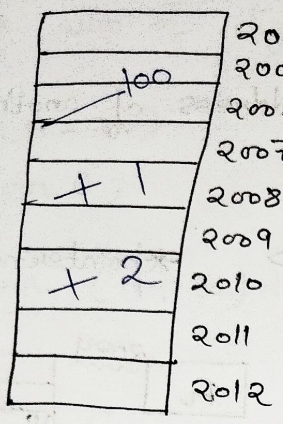
⇒
```
#include <stdio.h>
int main()
{
char ch = 'A';
int x = 100;
float y = 10.89;
int *p1;
char *p2;
float *p3;
p1 = &x;  // 2004
p2 = &ch; // 2024
p3 = &y;  // 2030
printf("Memory Address of x is %p", p1);
printf("Value Stored at p1 is %d", *p1);
printf("Memory address of ch is %p", p2);
printf("Value stored at p2 is %c", ch);
printf("Memory address of y is %p", p3);
printf("Value stored at p3 is %f", *p3);
}
```

Decimal ⇒ 0-9
Hexadecimal ⇒ 0-9, A, B, C, D, E, F

2 operators :-
i) & - address
ii) * - Indirection / Dereference

Operators...

- Reference / Address operator (&)
- Dereference / Indirection operator (*)



⇒ p1 + 2

⇒ p1 + 2 * size of (int)

p1 + 2 * 2 = 4

⇒

```
# include <stdio.h>
Void main ( )
{
int x=10,* px ;
char ch='x',* py;
float f=34,* pz ;
printf ("Memory address of x is %u, px);
px = px+4;
printf ("After jumping 4 memory locations px gives %u", px);
printf (" Value at px is %d", * px);
printf (" Memory address of y is %u", py );
printf (" py decrement by 2 gives %u", py-2);
printf (" Memory address of z is %u", pz)
printf (" pz incremented by 2 gives %u", pz+2);
}
```

→ Types of pointer :-

1.) Wild Pointer
2.) NULL Pointer
3.) Void Pointer

Example:-

```
# include <stdio.h>
Void main()
{
int *p; // pointer declared
```

=> 1. Wild pointer:- / Bad pointer:-

=> A pointer variable which is not initialized with any variable address or we can say an uninitialized pointer variable is called a wild pointer / bad pointer. Without any initialization it points to a random memory location. During implementation it is recommeded to initialize all the pointer variables with any variable address or make it null.

2. NULL Pointer:-

=> A pointer variable which is initialize to NULL to avoid any random memory allocation just like in wild pointer is called NULL Pointer. It doesn't store any address of a variable.

=> float * ptr = NULL;

Declaration of NULL Pointer

datatype * pointername = NULL;

3. Void pointer:-

A void pointer can access or manipulate any kind of datatype by using void pointer, we can hold address of a variable of any data type

Declaration of a void pointer

Void * pointername;

Example:-

```
# include <stdio.h>
  Void main ()
{
int a = 50;
char b = "#";
Void * ptr;
int * p;
ptr = &a;
printf ("Memory address of a is %u", ptr);
printf ("Value stored at ptr is %d", *((int*) ptr));
ptr = &b;
printf ("Memory address of b is %u", ptr);
printf ("Value stored at ptr is %c", *((char*) ptr));
* ptr = "h";
printf("Value stored at ptr is %c", *((char *) ptr));
}
```

Dereferencing a Void Pointer

```
*((type *) vptr);
```

→ Accessing array elements using pointer.

```
=> # include <stdio.h>
   Void main ()
{
int x [5] = { 0, -10, 8, 15, 33 };
printf ("Value of x %u", x);   // Base address of array 2016
printf ("Memory address of x[0] is %u", &x[0]); // 2016
printf ("Memory address of x [2] is %u", &x[2]); // 2020
printf ("Value stored at * x is %d", * (x-3)); // 0
printf ("Value stored at * (x+3) is %d", * (x+3)); // 15.
```

```
printf (" Value stored at x[4] is %d ", 4[x];
printf (" Value stored at x[4] is %d", *(x*4));
   *(x+3) = 75;      // x[3] = 75
printf (" Value stored at x[3] is %d", x(x+3));
}
```

→ Chain of pointers :-

```
=> #include <stdio.h>
   Void main ()
{
int x = 100, *p1, **p2, ***p3, ****p4;
p1 = &x;
p2 = &p1;
p3 = &p2;
p4 = &p3;
printf (" p1 = %u \t *p1 = %u\n", p1, *p1);
printf (" p2 = %u \t *p2 = %u\n", p2, *p2);
printf (" p3 = %u \t *p3 = %u\n", p3, *p3);
printf (" p4 = %u \t *p4 = %u\n", p4, *p4);
}
```

→ Pointer to array :-

```
=> #include <stdio.h>
   Void main()
{
int *p;
int (*ptr)[5];
int x[5] = { 2, 4, 5, 6, 7 };
int a[5] = {12, 3, 4, 90, 122 }
p = a;
ptr = &x;
```

```
printf ("p = %u    ptr = %u\n", p, ptr);
printf ("Value at p = %u  *ptr = %u  **ptr = %d\n",
        *p, *ptr, **ptr);
p++;
ptr ++;
printf ("p = %u    ptr=%u \n", p, ptr);
printf ("Value at p= %u  value at ptr = %u", *p, **ptr);
}
```

→ pointer to array;

```
=> #include <stdio.h>
   Void main ()
{
int *p;
int (*ptr) [5];
int x [5] = { 2, 4, 5, 6, 7 };
int a [5] = { 12, 3, 4, 90, 122 };
p = a ;
ptr = &x;
printf ("p = %u    ptr = %u\n", p, ptr);
printf ("Value at p = %u  *ptr = %u  **ptr = %d\n", *p,
        *ptr, **ptr);
p++;
ptr ++;
printf ("p = %u    ptr = %u\n", p, ptr);
printf (" Value at p = %u  value at ptr = %u", *p, **ptr);
}
```

→ Array of pointers:-

```
#include <stdio.h>
Void main()
{
int *ptr[5], i;
int x[5] = {-10, -20, 0, 10, 20};
for (i=0; i<5; i++)
    ptr[i] = &x[i];
for (i=0; i<5; i++)
{
printf("ptr[%d]= %u\t *ptr[%d] = %d\n", i, ptr[i], i,
        *ptr[i]);
}
}
```

Function call...

Call by value

```
#include <stdio.h>
Void swap (int, int);
Void main()
{
int a=10, b=20;
swap (a, b);
printf ("After swapping a=%d b=%d", a, b);
}
Void swap (int x, int y)
{
x = x+y;
y = x-y;
x = x-y;
}
```

Call by Reference

```
#include <stdio.h>
Void swap(int *x, int *y);
Void main()
{
int a =10, b = 20;
swap (&a, &b);
printf ("After swapping a=%d
        b=%d", a, b);
}
Void swap(int *x, int *y)
{
*x = *x + *y;   10+20=30
*y = *x - *y;   10
*x = *x - *y;   30-10=20
}
```

=> #define PI 3.14

```
#include <stdio.h>
Void showcircle (float, float * a, float * c);
Void main()
{
float radius, area, circle;
printf ("Enter radius of the circle");
scanf ("%f", &radius); // 7
show circle (radius, & area, & circle);
printf ("Area of circle is %f", area);
printf ("Circumference of circle is %f", circle);
}
Void showcircle (float r, float *a, float *c)
{
 *a = PI * r * r;
 *c = 2* PI * r;
}
```

area = 154	1024
area = 44	1032

Question paper solutions...

Q2. (a) What do you understand by loops ? Discuss 'for' loop and its syntax with example.

⇒ Loop is used to execute the block of code several times according to the condition given in the loop. It means it executes the same code multiple times so it saves code and also helps to transverse the elements of an array.

→ There are three types of loops :-

1. while loop
2. do - while loop
3. for loop

For loop :-

⇒ It also executes the code until condition is false. In this three parameters are given that is :-

- Initialization
- Condition
- Increment / Decrement

Syntax :-

⇒ for (initialization ; test expression ; update expression)
① ② ④

```
{
    // loop body → runtime order
    ③
}
```

For example :-

→ WAP to display first 10 natural numbers

```
# include <stdio.h>
int main()
{
int a;
for (a=1 ; a<=10 ; a++)
{
printf ("%d\n", a);
}
return 0;
}
```

memory → output

1 → 1
2 → 2
3 → 3
4 → 4
5 → 5
6 → 6
7 → 7
8 → 8
9 → 9
10 → 10
11 → Loop end

Another example :-

```
# include <stdio.h>
Void main()
{
int i;
```

```
for (i=20 ; i<25 ; i++)
{
printf ("%d" , i);
}
}
```

Output :-

=> 20 21 22 23 24

=> It initialize the variable i to 20, sets the condition to i < 25, and increments i by 1 in each iteration. The code inside the for loop is executed as long as the condition i < 25 is true.

When you run the code, it will print the numbers from 20 to 24 on the console, separated by a space.

(b) Write a program that reads a number and determines if the number is zero, positive or negative

=> A number is negative if it is less than 0 i.e. num < 0.

A number is said positive if it is greater than 0 i.e. num > 0.

We will use above logic inside if to check number for negative, positive or zero.

1. Input a number from user in some variable say num.
2. Check if (num < 0), then number is negative.
3. Check if (num > 0), then number is positive.

4. Check if (num == 0), then number is zero.

Let's check the number is positive, negative or zero using simple 'if';

```
# include <stdio.h>
 int main()
{
 int num;
printf ("Enter the number");
scanf ("%d", &num);
 if (num > 0)
{
printf ("Number is positive");
}
if (num < 0)
{
printf("Number is negative");
}
if (num == 0)
{
 printf ("Number is zero");
}
return 0;
}
```

Output :-

1. Enter the number : 10
   Number is positive

2. Enter the number : −3
Number is negative

3. Enter the number : 0
Number is zero.

⇒ In the above code, first check condition for positive number, then if a number is not positive number it might be negative or zero. Then check condition for negative number. Finally if a number is neither positive nor negative then definitely it is zero. There is no need to check zero condition explicitly.

3. (a) Differentiate b/w "while" and "do-while" loop with example.

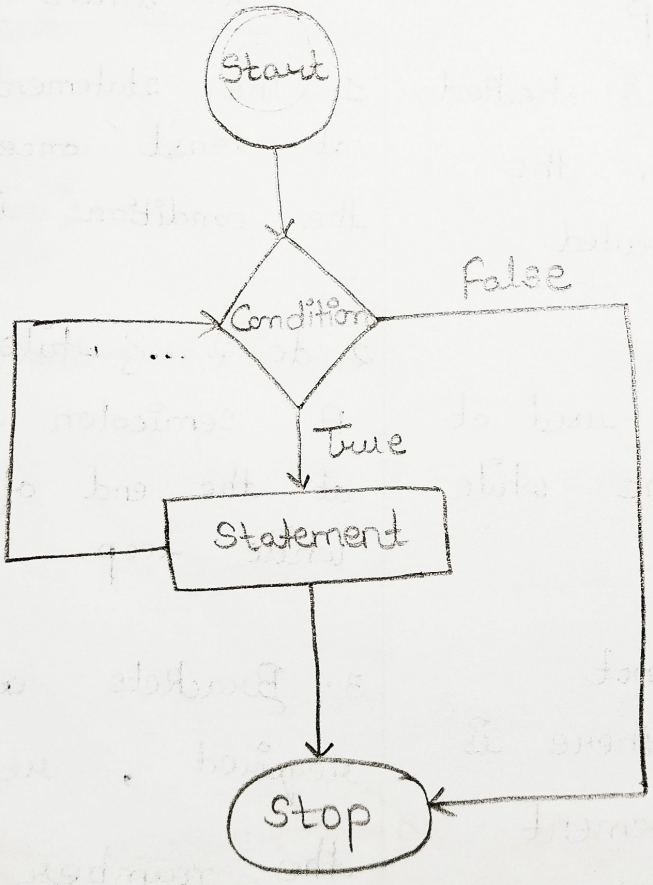
⇒

while loop	Do- while loop
1. The condition is checked first, and then the statement is executed.	1. The statement is executed at least once before the condition is checked.
2. while (condition) No semicolon is used at the end of the while loop.	2. do {.....} while (condition); A semicolon is used at the end of the Do while loop.
3. Brackets are not required if there is only one statement.	3. Brackets are always required, regardless of the number of statements.

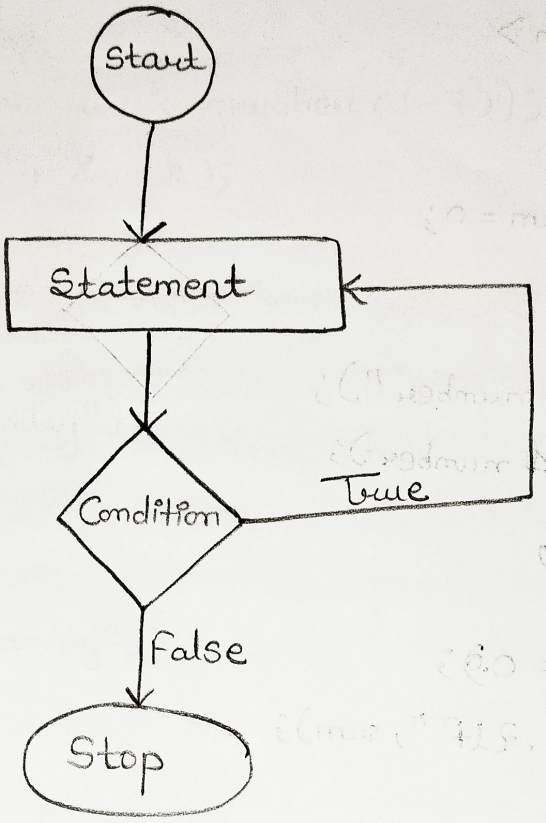
4. A variable must be initialized in the condition before the loop is executed.	4. The variable can be initialized before or within the loop.
5. The while loop is an entry - controlled loop.	5. The do - while loop is an exit - controlled loop.

6. Syntax :
=> while (Condition)
{
//loop body
}
	6. Syntax :
	=> do
	{
	//loop body
	}
	while (Condition);

→ Flowchart for while loop :-



→ Flowchart for do- while loop :-



Example of while loop ;

```
=> #include <stdio.h>
int main()
{
int count=1;
while
(count <=4)
{
printf ("%d", count);
count ++;
}
return 0;
}
```

Output :-

=> 1 2 3 4

Example of do - while loop :-

```
# include <stdio.h>
  int main()
{
double number, sum=0;
do
{
printf("Enter a number");
scanf("%lf", &number);
sum+= number;
}
while (number != 0.0);
printf("sum = %.2lf", sum);
return 0;
}
```

Output :-

=> Enter a number : 1.5
Enter a number : 2.4
Enter a number : -3.4
Enter a number : 4.2
Enter a number : 0
Sum = 4.70

(b) write a program to print the days of a week by using switch statement.

```
=> # include <stdio.h>
   int main()
```

```
{
int week;
printf ("Enter week number (1-7));
scanf ("%d", &week);
switch (week)
{
Case 1:
  printf ("Monday");
  break;
Case 2:
  printf ("Tuesday");
  break;
Case 3:
  printf ("Wednesday");
  break;
Case 4:
  printf ("Thursday");
  break;
Case 5:
  printf ("Friday");
  break;
Case 6:
  printf ("Saturday");
  break;
Case 7:
  printf ("Sunday");
  break;
default :
```

```
    printf ("Invalid input! Please enter week number
            between 1-7");
    }
    return 0;
}
```

Logic to print day of week name using Switch Case...

=> 1. Input day number from user. Store it in some variable say week.

2. Switch the value of week i.e. 1 to 7. Therefore write 7 Case inside switch. In addition, add default cases as an else block.

3. There can be seven possible values of week i.e. 1 to 7. Therefore write 7 case inside switch. In addition, add default case as an else block.

4. For e Case 1: print "Monday", for Case 2: printf "Tuesday" and so on. Print "Sunday" for Case 7:.

5. If any case does not matches then, for default: case print "Invalid week number".

Output :-

=> Enter the day of week : 1
Monday

Enter the day of week : 2
Tuesday

Enter the day of week : 3
Wednesday
Enter the day of week : 4
Thursday
Enter the day of week : 5
Friday
Enter the day of week : 6
Saturday
Enter the day of week : 7
Sunday

Structures...

=> Declaration of structure :-

Syntax :-

```
Structure statement
{
datatype 1   member1;
datatype 2   member2;
   ---
   ---
datatype n   member n
};
```

where n members are part of the structure.

-> WAP to create a user defined datatype date (by using pointer)

=>
```
# include <stdio.h>
# include <string.h>
```

```
Struct date
{
 int day;
 char *month;
 int year;
};
Void main ( )
{
Struct date d1 = { 23, "Jan", 2011 };
 Struct date d2;
 d2.day = 10;
Strcpy (d2.month, "Nov");
 d2.year = 2024;
printf ("Date 1: %d %s %d", d1.day, d1.month, d1.year);
printf ("\n Date 2: %d %s %d", d2.day, d2.month, d2.year);
}
```

- dot operator :- To access the member of structure.

=> Structure is a user-defined datatype in C language which is used to store a collection of different types of data under a single variable name. It is in many respects similar to an array only difference is that array is a collection of homogeneous data elements or similar type of data elements where as structure is a collection of heterogeneous data elements or different types of data element.

Struct keyword is used to declare a structure and variables declare inside declaration block of structure are called members of the structure.

→ WAP to create a structure store date related to students

```
=># include <stdio.h>
. # include <string.h>
  Struct student
  {
    char name [10]
    int roll no. ;
    char section [10];
  };
    Void main ()
  {
  struct student S1 = {"Pratiksha", 26347", AIML2" };
   struct Student S2 , S3;
   strcpy (S2 name , "Sanya");
   S2 roll no. = 26377;
  Strcpy (S2. Section , " CSE 1");
  printf ("Enter name of student 3");
  gets (S3 . name);
printf ("Enter roll no. of Students 3);
scanf (" %d", &S3. roll no);
printf (" Enter section of student 3");
gets ( S3. section);
printf (" Student 1 \n");
```

```
printf ("Name : %s  Roll no : %d  Section : %s", S1.name,
        S1.roll no , S1. section);
printf ("Student 2 \n");
printf ("Name : %s  Roll no %d  Section %s \n", S2 name,
        S2 roll no, S2 section);
printf ("Student 3 \n");
printf ("Name : %s  Roll no %d  Section %s \n", S3 name,
        S3 roll no , S3 section);
}
```

→ Array of structure :-

→ WAP to create an array of structures

```
=> # include <stdio.h>
   # include <string.h>
    Struct Student
   {
    char name [10];
    int roll no ;
    char Section [10];
   };
    Void main ( )
  {
   struct Student Std [5];
   for (int i=0, i<=5 ; i++)
  {
  printf ("Enter data for student %d", i+1);
  printf ("Enter name of student %d", i+1);
  gets (std [i], name);
```

```
printf ("Enter roll no. of student %d", i+1);
scanf ("%d", &std[i].rollno);
printf ("Enter section of student %d", i+1);
gets (std[i].section);
}
for (int j=0; j<5; j++)
{
printf ("student %d", j+1);
printf ("\n Name : %s Roll no %d section %s", std[j].
        name, std[j].rollno, std[j].section);
}
}
```

Que. What do you mean function call by value and function call by reference. Differentiate b/w the two with the help of an example.

→ Calling a function :-

⇒ When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed (or when its function-ending closing brace is reached), it returns the program control back to the main program.

Control of the program is transferred to the user-defined function by calling it.

→ Syntax of function call :-

⇒ function name (argument 1, argument 2, ---);

In the above example, the function call is made using addNumbers (n1, n2); statement inside the main() function.

→ Functions can be invoked in two way:-

1.) Call by Value

2.) Call by reference

1. Call by value :-

=> The value of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

2. Call by reference :-

=> It copies the address of an argument into the formal parameter. In this method the address is used to access the actual argument used in the function call. It means that changes made in the parameter alter the passing argument.

Call by Value	Call by reference
=>i) While calling a function, we pass values of variable to it. Such functions are known as "Call by value".	i) While calling a function, instead of passing the values of variables, we pass address of variables to the function known as "Call by reference".
ii) In this method, the value of each variable in calling function is copied	ii) In this method, the address of actual variables in the calling function

into corresponding dummy variables of the called function.	are copied into the dummy variables of the called function.
iii) Actual and formal arguments are created at the different memory location.	iii) Actual and formal arguments are created at the same memory location.

Example :- Call by Value...
Swap the values of the two variables.

```
#include <stdio.h>
Void swap (int , int)
int main ()
{
int a = 10;
int b = 20;
printf (" Before swapping the values in main a = %d, b = %d \n",
        a, b);
swap (a, b);
printf (" After swapping values in main a = %d, b = %d \n",
        a, b);
}
Void swap (int a, int b)
{
int temp;
temp = a;
a = b;
b = temp;
printf (" After swapping values in function a = %d, b = %d
        \n", a, b);
}
```

⇒ Output :-

Before swapping the values in main
a = 10, b = 20
After swapping values in function
a = 20, b = 10
After swapping values in main
a = 10, b = 20

Example :- Call by reference...

Swapping the values of a two variables

=>
```
#include <stdio.h>
Void swap (int *, int *);
int main()
{
int a = 10;
int b = 20;
printf ("Before swapping the values in main a = %d, b = %d
\n", a, b);
swap (&a, &b);
printf ("After swapping values in main a = %d, b = %d \n",
a, b);
}
Void swap (int *a, int * b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
printf ("After swapping values in function a = %d, b = %d
\n", *a, *b);
}
```

Output :-

=> Before swapping the values in main a = 10, b = 20

After swapping values in functions a = 20, b = 10

After swapping values in main a = 20, b = 10

File Handling...

- Opening a file
- Reading a file
- Writing a file
- Updating / Appending a file's content.
- Changing position for reading / writing a file.
- closing a file.

Modes...
- r (read)
- w (write)
- a (append)
- r+ (read and write)
- w+ (read and write)
- a+ (read and append)

```
=> # include <stdio.h>
   Void main()
   {
    FILE *fp;
fp = fopen("C:\abc\a.txt", "w");
if (fp == NULL)
{
printf("Cannot create file");
exit(1);
}
fprintf(fp, "HELLO");
fclose();
}
```

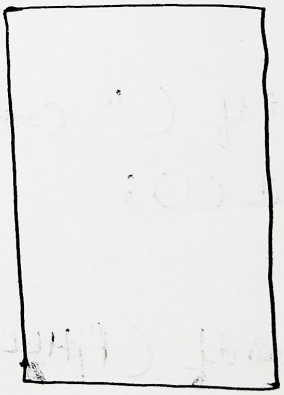
Another example :- WAP to write array elements to the file.

```
=># include <stdio.h>
```

```
Void main ()
{
int x[10];
FILE *fp;
fp = fopen ("b.txt", "w");
if (fp == NULL)
{
printf (" Cannot open the file ");
exit(1);
}
for (int i=0; i<10; i++)
{
printf ("Enter a number");
scanf (" %d", &x[i]);
fprintf (fp, "%d\n", x[i]);
}
fclose();
}
```

→ WAP to read Contents from a file

```
⇒ # include <stdio.h>
  Void main ()
  {
  char ch;
  FILE *fp;
fp = fopen ("b.txt", "r");
if (fp == NULL)
```



```
{
printf ("Cannot open the file");
exit(1);
}
ch = fgetc (fp);
while (ch != EOF)
{
printf ("%c", ch);
ch = fgetc (ch);
}
fclose (fp);
}
```

=> FILE is a structure defined in Header file <stdio.h> and objects of type FILE Contain FILE related information like mode of opening, size of file, place a FILE from where next read operation would start pointer to associated buffer if any error indicator that records whether a read or right has occur and a end of FILE indicator record wether the end of FILE is reach.

-> WAP to copy contents of one file to another

```
=> #include <stdio.h>
#include <stdlib.h>
Void main( )
{
FILE *fp1, *fp2;
```

```
fp1 = fopen ("file1.txt", "r");
fp2 = fopen ("file2.txt", "w");
if (fb1 == NULL || fb2 == NULL)
{
printf ("Cannot copy the Contents");
exit(1)
}
ch = fgetc (fp1); // Read a character from fp1
while (ch != EOF)
{
// fprintf (fp2, "%c", ch);
fputc (ch, fb2);
ch = fgetc (ch);
}
fclose (fp1);
fclose (fp2);
}
```

→ WAP to Count number of characters and words in a file

```
=> #include <stdio.h>
#include <std.h>
Void main ()
{
```

```
char ch; count ch=0, count w=0;
FILE * fp1
fp1 = fopen ("file1.txt", "r");
if (fp1 == NULL)
{
printf ("Cannot open the file");
exit(1)
}
ch = fgetc (fp1);
while (ch != EOF)
{
Count ch++;
if(ch==' '||ch=='\n'||ch=='\t'||ch==','||ch==';'||
   ch==':'||ch=='.')
Count w++;
ch = fgetc (fp1);
}
if (count ch>0)
Count w++;
else
printf ("Empty file");
printf ("No. of characters are %d", count ch);
printf ("No. of words are %d", Count w);
fclose (fp1);
}
```

Dynamic Memory Allocation...

→ Library function used for dynamic memory allocation:-

⟶ malloc ( )

⟶ calloc ( )

⟶ realloc ( )

⟶ free ( )

→ Static memory allocation:-

⇒ The memory allocation done while declaring an array of fix size or for any fixed number of variables is static memory allocation and size of allocated memory cannot be charged during program execution.

→ Dynamic memory allocation:-

⇒ The process of allocating memory at the time of execution is called dynamic memory allocation. The allocation and release of memory can be done with the help of some library functions declared in <alloc.h> and <stdlib.h>.

Pointers play an important role in dynamic memory allocation as we can access dynamically allocated memory only through pointers.

1. Malloc function:

⇒ It is used to allocate memory dynamically and it stands for memory allocation. The argument size passed to the function specifies number of bytes to be

allocated on success malloc function returns a pointer to the first byte of allocated memory. The return pointer is of type void which can be typecast to appropriate type of pointer. Malloc function is generally used as;

```
malloc (size in bytes);
```

e.g.:→

```
datatype *ptr;
ptr = (datatype *)malloc (size);
int *p;
p = (int *)malloc (5 * size of (int));
```

Malloc function returns a NULL pointer if it is not able to allocate requested amount of memory.

Malloc function does not initialize memory allocated during execution. This memory allocated : Carries garbage values.

For example:-

```
# include <stdio.h>
# include <stdlib.h>
# include <alloc.h>
  Void main()
{
int *p, n, // p is a dynamic array
printf ("Enter number of elements");
scanf ("%d", &n);
p = (int *)malloc (n* size of (int));
printf ("Enter elements for dynamic array");
```

```
for (int i=0; i<n; i++)
{
printf ("Element %d", i+1);
scanf ("%d", p+i);     // &p[i]
}
for (int i=0; i<n; i++)
{
printf ("Element %d: %d\n", i+1, *(p+i));
}
free(p);
}
```

→ Calloc :-

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
Void main()
{
int *p, n; // p is a dynamic array
printf ("Enter number of elements");
scanf ("%d", &n);
p = (int *) calloc (n, sign of (int));
printf ("Enter elements for dynamic array");
for (int i=0; i<n; i++)
{
```

```
printf ("Element %d", i+1);
scanf ("%d", p+i);
}
for (int i=0; i<n; i++)
{
printf ("Element %d : %d\n", i+1, *(p+i));
}
free (p);
}
```

⇒ Calloc stands for Contiguous Allocation.
Calloc function is used to allocate multiple blocks of memory of given size. It is some what similar to malloc accept for following differences.

(i) It takes two arguments unlike malloc which takes one argument. First argument specifies number of blocks and second argument specifies the size of each block.

(ii) Memory allocated by malloc contains garbage values while memory allocated by calloc is initializes to zero.

(iii) Malloc is used to allocate single block of memories where calloc is used to allocate multiple blocks of memory.

(iv) Malloc is used for simple data structures whereas calloc is used for Complex data structures.

⇒ datatype *p;
p = (datatype *) calloc (no. of blocks, size of each block);

⇒ Free function :-

⇒ free (p);

⇒ The dynamically allocated memory is not automatically released. It will exist till the end of the program. In order to release this memory so that it can be used for some other application we can use function free (p), malloc function allocate memory from a memory are called heap. When free function is called this memory is released back to one heap. So that it can be used for some other purpose.

→ Realloc function :-

⇒ Realloc function can be used to increase or decrease the memory allocated by malloc function or calloc function, This function alters the size of the memory block without loosing the stored data.

This function take two argument :-

(i) Pointer to the block of memory allocated by malloc or calloc.

(ii) New size of the memory to be allocated. Realloc function is generally used as,

datatype * ptr;

ptr = (datatype *) realloc (pointer to memory block, New size);

For example :-

```
# include <stdio.h>
# include <alloc.h>
# include <stdlib.h>
```

```
Void main()
{
int *p, sum=0;
printf("Enter 5 elements");
p = (int *)malloc(5 * size of (int));
for (int i=0; i<5; i++)
scanf("%d", p+i);
for (int i=0; i<5; i++)
{
printf("Element %d: %d", i+1, *(p+i));
sum += (p+i);
}
printf("Sum of dynamic array elements %d", sum);
p = (int x) realloc (p, 15 * size of (int));
printf("Enter 10 more elements");
for (int i=5; i<15; i++)
{
scanf("%d", p+i);
}
for (i=0; i<15; i++)
{
printf("Element %d: %d", i+1, *(p+i));
printf("\n");
}
free(p);
}
```

if (p == NULL) →
{
printf("Sufficient space not available");
exit(1);
}

★ Self-Referential Structures :-

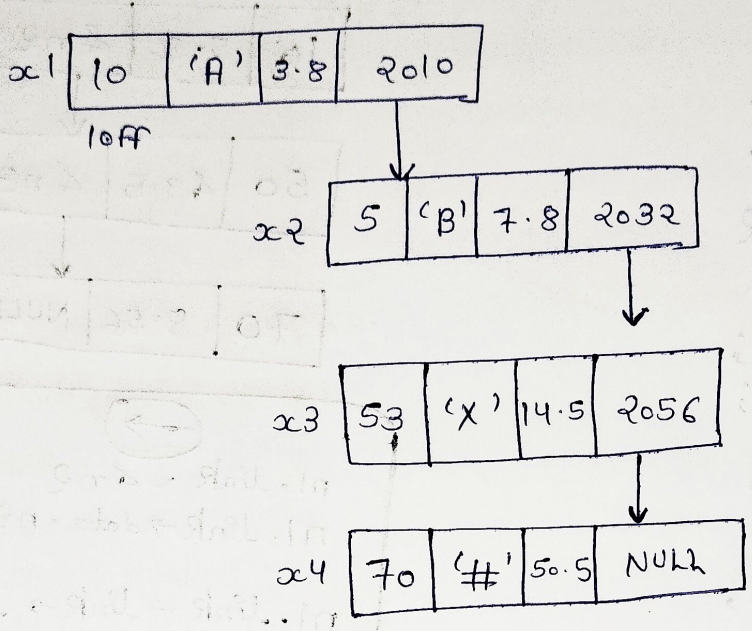
Syntax :-

```
Struct  Structurename
{
   datatype1   member1 ;
   datatype 2   member2 ;
      =
      =
      =
   datatype n    member n ;
   Struct   Structurename   * pointer 1 ;
      =
      =
      =
   Struct   structurename   * pointer n ;
};
```

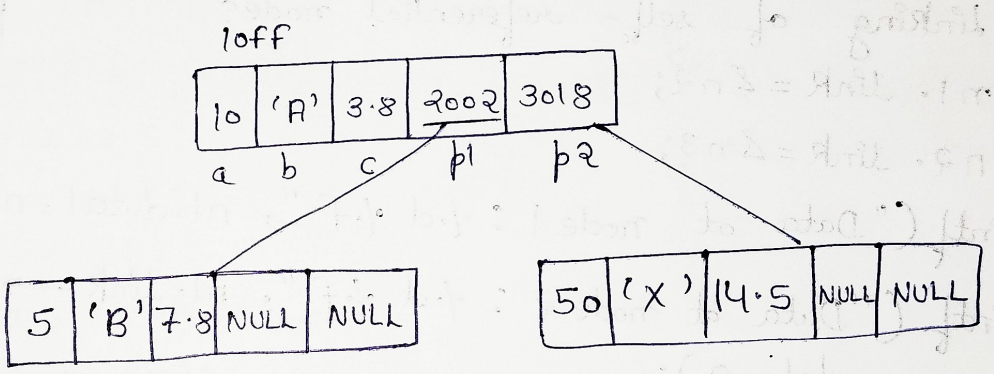
⇒ A structure that Contains pointers to structures of its own type is known as self-referential structure. This type of structures are helpful in implementing data structures and linked lists and trees.

⇒
```
Struct  abc
{
  int a ;
  char b ;
  float c ;
struct abc *p ; };
```

// Example of self - referential structure.

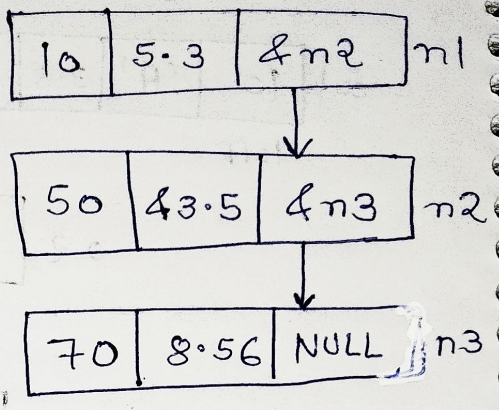


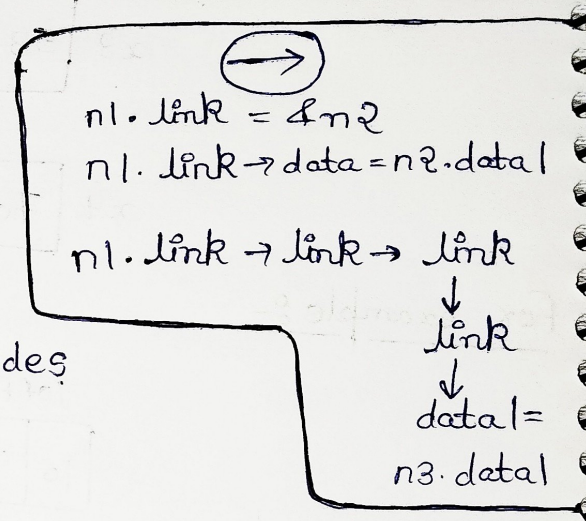
For example :-




```
=> # include <stdio.h>
   Struct node
   {
    int data1;
    float data2;
    Struct node *link;
   };
Void main()
{
 Struct node n1, n2, n3;
```

```
// Initialize node n1
   n1.data1 = 10;
   n1.data2 = 5.3;
   n1.link = NULL;
// Initialize node 2
   n2.data1 = 50;
   n2.data2 = 83.5;
   n2.link = NULL;
// Initialize node 3
   n3.data1 = 70;
   n3.data2 = 8056;
   n3.link = NULL;
// Linking of self-referential nodes
   n1.link = &n2;
   n2.link = &n3;
printf("Data at node1 : %d %f", n1.data1, n1.data2);
printf("Data at node 2 : %d %f", n1.link -> data1, n1.link ->
        data 2);
printf("Data at node 3 : %d %f", n1.link -> link -> data1,
        n1.link -> link -> data2);
}
```





→ Self-referential structures using two pointers.

```
=> # include <stdio.h>
   Struct node
{
 int data1;
float data2;
```

```
    struct node * link1;
    struct node * link2;
};
Void main ()
{
struct node n1, n2, n3;
// Initialize node n1
 n1. data1 = 10;
 n1. data2 = 5.3;
 n1. link1 = NULL;
 n1. link2 = NULL;
// Initialize node 2
 n2. data1 = 50;
 n2. data2 = 83.5;
 n2. link1 = NULL;
 n3. data1 = 70;
 n3. data2 = 8.56;
 n3. link1 = &n2;
 n3. link2 = NULL;
 n1. link1 = &n2;
 n1. link2 = &n3;
printf ("Date at node 1: %d %f ", n1. data1 , n1. data2);
printf ("Data at node 2: %d %f ", n1. link1 → data1 , n1. link1→
                                                    data2);
printf ("Data at node 3: %d %f", n2. link→ data1 , n2. link2 →
                                                    data2);
}
```