II SEM III-IV UNIT
INTRODUCTION TO PROGRAMMING

Array :-

Variables of basic data type can store only a single val at a time. eg → int a ;

a can store one integer value at a time.

These types of variables are called scalar variables.

Declaring array :-

Syntax:- Basic DataType arrayname [size];

eg → int as [50]; → Array of integer type to store 50 values.

& float fa [100]; → Array of float type to store 100 values.

Need of an array :-

If in a program requirement comes to store 20 or 100 integer values at a time the declaration would be :

int a1, a2, a3 . . . . , a100;

Using these variables would be even more difficult. so its better to use a single variable name for all 100 values & the difference would be only for indexing.

In an array first value is always stored at "0" index & last value at "size-1"

eg:- int a[20];

so value would be from a[0] to a[19]

0 → Lower Bound 19 → Upper Bound

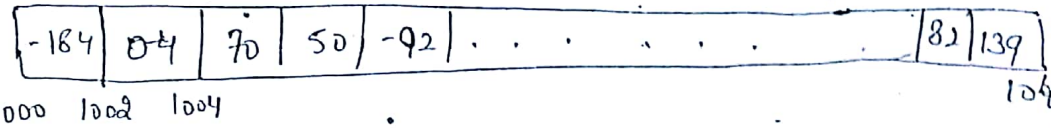
Memory Representation of an Array :-

int a[20];

After declaring an array, 40 bytes get immediately reserved in memory, 2 bytes for each of integer.

As array is not initialized, so all values values will get garbage (Wast) value.

All the array elements would always be present in contigious mememory locations



20 integer variables
& 40 bytes of memory
& Garbage values.

Array Initialization :-

```
int ar[5] = {2, 4, 6, 8, 10};
int n[] = {2, 4, 6, 8};
```

→ If the array elements are not initialized, they will get garbage value by default.

But in above case array values will get these above values.

→ If the array is initialized when it is declared, mention the dimension of the array is optional as in 2nd example above.

Accessing Array Elements :-

Single operations, which involve entire array, are not permitted in C. So operations could only be carried out on element-by-element basis. This is usually done within a loop.

eg → Store 20 as 3rd value of variable

```
so  a[2] = 20;
      ↑
    index
```

0 index is for 1st value
1 index is for 2nd value
.
.
.
99 index for 100th value

```
a[2] = a[0] + a[1];
```

Input:- int a[5];

```
printf("Enter values");
for (i=0 ; i<5 ; i++)
{
    scanf ("%d", &a[i]);
}
```

↳ index from 0 to 4

Similarly for processing & output we can use loops.

Important points about Array :-

* An array can store multiple elements at same time.
* All the array elements is of same type.
* First element in the array is at 0 index & last element is at size-1 index.
  i.e lower Bound = 0 & Upper Bound = size-1
* All the array elements are always stored in contiguous memory locations.

Allowed operations for an array :-

→
```
ar[i] = ar[i] + 1;
   OR
ar[i] += 1;
   OR
ar[i]++;
```
} To increment ith value of ar

→
```
ar[i] += n;
ar[i] = ar[i] + n;
```

→ ar[k] = ar[i]

→ br[k] = ar[i]

Example:- To calculate total marks & percentage of six subjects

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int m[6], i, tm;
    float per;
    clrscr();
    printf("Enter marks in 6 subjects");
    for(i=0; i<6; i++)
    {
        scanf("%d", &m[i]);
    }
    tm=0;
    for(i=0; i<6; i++)
    {
        tm = tm + m[i];
    }
    per = tm/6.0;
    printf("Total marks = %d", tm);
    printf("Percentage = %d", per);
    getch();
```

Multi Dimensional Array :-

Array with more than one dimension are called multi-dimensional array. It could be 2-dimensional, 3-dimensional

Declaration :-

2-D :- datatype arrayname [size 1] [size 2];

eg → int ar [3] [2];
- 3 → Row
- 2 → Column

3-D :- datatype arrayname [size1] [size2] [size3];

int ar [3] [2] [4];

Initialization :-

```
int ar[3][2] = {
                { 2, 3 },
                { 4, 7 },
                { 9, 11 },
              };
```

OR

```
int ar[3][2] = { 2, 3,
                 4, 7,
                 9, 11 };
```

OR int ar[3][2] = { 2, 3, 4, 7, 9, 11 };

Total No. of Elements :-

2D :- size1 x size 2 ⇒ eg → int a[5][2] ⇒ No of Elements = 10

3D :- size1 x size 2 x size 3

Memory Representation of 2-D Array :-

arr[0][0] = 2 → arr[0][1] = 3

arr[1][0] = 4 → arr[1][1] = 7

arr[2][0] = 9 → arr[2][1] = 11

i.e

Column no →	Column
2 | 3	row 0
4 | 7	row 1
9 | 11	row 2

In Computer Memory

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]
2	3	4	7	9	11
← row 0 →		← row 1 →		← row 2 →	

Accessing Multidimensional Array :-

To access multidimensional array we need nested loops - (Inner & outer). Outer loop is for row and inner loop is for column.

eg → to take input of a 2D array arr[3][2]

```
for (i=0; i<3; i++) → outer loop
{
    for (j=0; j<2; j++) → inner loop   i → Row
    {                                  j → Column
        scanf ("%d", &arr[i][j]);
    }
}
```

In the above example first inner loop will complete for value of i = 0 i.e

for i = 0 → j = 0, j = 1

i = 1 → j = 0, j = 1

for i = 2 → j = 0, j = 1

mple:-

WAP to store 2 matrices of order 3x2 & Display Sum of these 2 matrices.

Matrix of order 3x2 → $\begin{bmatrix} a[0][0] & a[0][1] \\ a[1][0] & a[1][1] \\ a[2][0] & a[2][1] \end{bmatrix}$

```
#include <stdio.h>
#include <conio.h>
Void main()
{
    int a[3][2], b[3][2], c[3][2], i, j;
    clrscr();
    printf("Enter elements of 1st matrix");
    for(i=0; i<3; i++)
    {
        for(j=0; j<2; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    printf("\n Enter elements of 2nd matrix");
    for(i=0; i<3; i++)
    {
        for(j=0; j<2; j++)
        {
            scanf("%d", &b[i][j]);
        }
    }
```

a → 1st matrix
b → 2nd matrix
c → Sum of 1st & 2nd

→ Input of 1st matrix

Input of 2nd matrix

```
for (i=0; i<3 ; i++)
{
   for ( j=0; j<2 ; j++)
   {
      c[i][j] = a[i][j] + b[i][j];
   }
}
for (i=0; i<3 ; i++)
{  printf("\n");
   for (j=0; j<2 ; j++)
   {
      printf ("\t %d", c[i][j]);
   }
}
getch();
}
```

Processing i.e Sum of a & b matrix

Output of c matrix (Sum)

V.G

Example :- WAP to calculate multiplication of 2 matrices

Example :- WAP to calculate transpose of a matrix.

Multiplication of 2 matrices

Matrix 1 is of order $m \times n$

Matrix 2 is of order $p \times q$

multiplication of Matrix 1 & 2 is possible only when $\boxed{n = p}$ & output would be of $\boxed{m \times q}$ order

i.e matrix 1 → 3×2 (2 same)

matrix 2 → 2×4

output matrix → 3×4

Prog:-

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int a[5][5], b[5][5], c[5][5];
  int i, j, k, m, n, p, q;
  clrscr();
  printf("Enter order of 1st matrix");
  scanf("%d %d", &m, &n);
  printf("Enter order of 2nd matrix");
  scanf("%d %d", &p, &q);
  if (n != p)
  { printf("Multiplication not possible");
    exit(0);
  }
```

```
printf("\nEnter elements of 1st matrix");          → Input 1st matrix
for (i=0; i<m; i++)
{
  for (j=0; j<n; j++)
  {
    scanf("%d", &a[i][j]);
  }
}
printf("\n Enter elements of 2nd matrix");          → Input of 2nd matrix
for (i=0; i<p; i++)
{
  for (j=0; j<q; j++)
  {
    scanf("%d", &b[i][j]);
  }
}

for (i=0; i<m; i++)                                  → Processing
{
  for (j=0; j<q; j++)
  {
    c[i][j]=0;
    for (k=0; k<n; k++)
      c[i][j]= c[i][j] + a[i][k]*b[j][k];
  }
}

for (i=0; i<m; i++)                                  → output
{ printf("\n");
  for (j=0; j<q; j++)
  {
    printf("\t %d", c[i][j]);
  }
}
getch();
}
```

Transpose of a Matrix :- Transpose would only be calculated for mxm matrix

Matrix 1 → $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ Transpose → $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

Prog :-

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int a[3][3], b[3][3], i, j;
   clrscr();
   printf("Enter elements of 1st matrix");
   for(i=0; i<3; i++)                    → Input
   {
      for(j=0; j<3; j++)
      {
         scanf("%d", &a[i][j]);
      }
   }
   for(i=0; i<3; i++)                    → Processing
   {
      for(j=0; j<3; j++)
      {
         b[i][j] = a[j][i];
      }
   }
   printf("\nTranspose of Matrix");
   for(i=0; i<3; i++)
   {  printf("\n");
      for(j=0; j<3; j++)
      {  printf("\t%d", b[i][j]);
      }
   }
   getch();
}
```

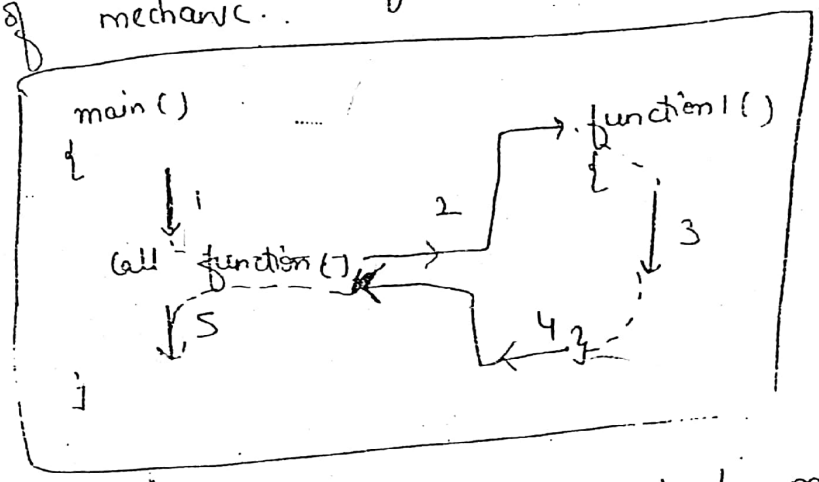
FUNCTIONS

A function is a self-contained block of statements performing a particular task.

Using a function is somewhat like hiring person for a particular job.

eg→ Suppose we want servicing of our vehicle we will give it to service station. dont need to give any instruction about it to do as mechanic knows his job and our vehicle back to us after servicing

Similarly functions performs the job of mechanic.



(→) Arrow signifies the control movement Whenever the need comes to execute statem given in function we call it so control shifted to that function. All the statemei function gets executed & then control is back to calling function statements & remaining statements are resumed so these statements are executed again.

Types

Types of functions :-

Functions in C are categorized into following two types :

→ In-built function (Library function) :-

The functions that are available in library & i.e created by C manufacturer are called library or system Defined functions.

eg → getch() & clrscr() → functions available in "conio.h" header file.

Definations of these functions is already available we just need to call these functions when needed.

→ User-Defined functions :-

The functions defined by the user according to their requirements are called user-defined functions the defination of function is been given by the user and calls the function whenever needed

eg → Sum(), Factorial() etc.

Need of functions :-

Functions basically breaks a big program into easy manageable chunks and hence provides following advantages :

1) Code Reuse :-

If we need to do a task many times in a program then same set of statements defining that function must be replicated at various points so better way is to create a function containing all those statements and just call whenever needed so code written once can be reused many times.

2) Functions make our code shorter and hence more readable & easy to understand.

3) It's easy to find mistakes in a shorter coding & have to correct a mistake only at one place instead of everywhere.

4) A function can be reused in multiple programs so are useful in code sharing.

5) Functions can be used to protect data. Local data of a function can be used only within a function & hence being protected from other function

Parts

1) Function

2) Function

3)

Parts of function (User- Defined function):-

1) Function Prototype / Signature / Declaration
2) Function Calling / Invoking
3) Function Defination

```
#include <stdio.h>
void main()
{
    <variable declaration>;
    <Function declaration>;          — (1)
    .
    .
    .
    <Function Calling>;              — (2)
    .
    .
}
Return Type Function ( )             } — (3)
{
    <Function Defination>;           }
}
```

1) Function Declaration :-

This statement basically identifies a function with name, list of arguments, type of value returned. This statement is specified in the starting where the variables are declared.

Syntax:- `Return Type Function_Name();`

OR

`Return Type function_name (DataType1, DataType2 ...);`

OR

`Return Type function_name (DataType1 variable1, DataType2 variable2 ...);`

Function_name → could be anything according to user wish.

(...) → In brackets we specify either (type) or (type &variable name) of the values being passed from main function to that particular function. These variables are called arguments or Parameters. No. of values could passed as parameters.

Return_Type → This specifies the data type of value being passed from this function to main function after execution of a statements of function. This could be int, float, char.

We can return only a single value

eg →

Suppose we want to add 2 no's so name would be Sum.

Parameters passed would be 2 int or float value

Return Type would be int or float value

```
int Sum (int, int);
```

OR

```
int Sum (int a; int b);
```

~~2) Function Calling~~ :-

→ Compiler must know details of a function before it is being used. Therefore function is declared prior to its use to allow compiler to check types of arguments & returning variable.

If a function is not

2) Function Calling :-

Function Calling is a single statement which replaces all lines which defines the actual task. Therefore this is the actual point where we want to accomplish that task. While invoking/Calling a function, the control program control is passed to the function defination. Once the function completes its task, the program control is passed back to calling statement.

Syntax :-

```
function_name (variable1 , variable2 . . .);
```

↳ If Return Type is void

OR

```
variable name = function_name (variable1 . . . . );
```

↳ If function returns something

OR

```
function_name ();
```

↳ If function does not return any & does not pass any parameter

3) Function Defination :-

The collection of program statements c that describes the specific task done by the function is called function defination. It consists → function Header & → function body

Syntax :-

```
function ReturnType FunctionName (DataType1 variable1, DataType2 variable2)
Header   {
           statement1;
           statement2;
function   return (variablename);
body     }
```

Return Type could be

→ int

→ float

→ char

→ void (Returns nothing)

eg → int Sum (int a, int b)
{
int c;
c = a + b;
return (c);
}

→ Example of function def. which takes 2 integers as formal parameters in it & returns an integer value.

eg → void printno (int n)
{
printf (" The number to be printed is %d ", n);
}

→ Example of function definition having 1 formal parameter & returns nothing so void

eg → void print ()
{
printf ("Hello ");
}

→ Example of function definition which has no formal parameters & returns nothing so void.

Program :-

WAP to compute average of 3 no's using functions

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a, b, c;
    float avg;
    clrscr();
    float average(int , int , int);
    printf("\n Enter 3 integer numbers");
    scanf("%d %d %d", &a, &b, &c);
    avg = average(a, b, c);
    printf("Average = %f", avg);
    getch();
}
float average(int a, int b, int c)
{
    float d;
    d = (a+b+c)/3.0;
    return(d);
}
```

→ function declaration signature

Arguments or Actual parameter

→ function Calling / invoking

formal parameter

function Definition

Program:-

WAP to find factorial of a no:

factorial → 5! = 5 × 4 × 3 × 2 × 1 = 120

Program of factorial

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n, f;
    int fact(int);
    clrscr();
    printf("Enter an integer number");
    scanf("%d", &n);
    f = fact(n);
    printf("\n Factorial = %d", f);
    getch();
}
int fact(int a)
{
    int f=1;
    while(a>=1)
    {
        f = f * a;
        a--;
    }
    return(f);
```

Scanned by CamScanner

Important points about function :-

→ There could be any number of functions in a program.

→ A single function can be called multiple times.

→ A function can be called through
  → main function
  → Any other function
  → itself

→ A function can accept multiple parameters/variables passed from calling function to called).

→ A function can only return a single value to the calling function.

→ No two functions can have same name.

→ Order of arguments should be same in calling statement & function defination statement.

variables used during Calling statement are called Actual parameter or arguments.

→ variables used during defination are called formal parameters.

→ variables are not actually passed to the function but a copy of it is being transferred to the function & so changes made during function defination are not reflected back in the calling function.

Functions & Arrays:-

Array can also be passed as arguments of functions. When entire array is an argument of a function, only address of the array is passed and not the copy of array (in case of a variable copy of variable is passed to the function). This implies that during its execution the function has the ability to modify the contents of the array that is specified as function arguments.

Function declaration could be done in any of the following ways:

→ float average (int []); // [] signifies array

→ float average (int a[]); // specifying the name of array

→ float average (int a[10]); // specifying name & size as well

→ float average (int *a); // * signifies address

Program

Function Calling:-

```
avg = average (a);
```

Function Definition:-

```
float average (int a[])
{
    :
}
```

OR

```
float average (int a[10])
{
    :
}
```

WAP to find average of 10 numbers stored in an array using function.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[10];
    float avg;
    float average (int i);
    clrscr();
    printf("Enter 10 integer numbers");
    for(i=0; i<10; i++)
    {
        scanf("%d", &a[i]);
    }
    avg = average (a);
    printf("Average = %f", avg);
    getch();
}
float average (int a[])
{
    float d=0;
    for(i=0; i<10; i++)
    {
        d = d + a[i];
    }
    d = d/10.0;
    return(d);
}
```

→ function Declaration

→ function calling
a[] → actual formal parameter
OR
Arguments

→ function Definition
a[] is formal parameter

Recursion :-

Defination - When a function call itself then it is called recursion. In recursive function we must include a condition so that it won't execute infinite times.

OR A recursive function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-calling.

Program:-

WAP to find the factorial of a number using recursive function.

```
#include <stdio.h>
void main()
{
    int n, f;
    int fact(int n);
    printf("\n Enter an integer number");
    scanf("%d", &n);
    f = fact(n);
    printf("\n Factorial = %d", f);
}
int fact(int num)
{
    if (num == 0)
        return 1;
    else
        return (num * fact(num-1));
}
```

Recursive calling of function fact

→ Let number be 3.

Then in fact function sequence of events would be -

num is (3 == 0) ?

False.

Then 3 * fact (2)

2 == 0 ? False

Then 3 * 2 * fact (1)

1 == 0 ? False

Then return 3 * 2 * 1 = 6 → Answer

Factorial could be computed as recursive function because the same function could be used repeatedly just by decrementing the number as -

$6! = 6 \times 5! = 6 \times 5 \times 4! = 6 \times 5 \times 4 \times 3! = 6 \times 5 \times 4 \times 3 \times 2!$

$= 6 \times 5 \times 4 \times 3 \times 2 \times 1! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$

Example of Recursion :-

→ WAP to compute a raise to the power b (a^b) using recursive function.

Strings OR Character Array :-

Data Type "char" can store only a single alph at a time like 'a' or 'b' etc but can not st multiple alphabets so for that we create an arr of characters. The combination or sequence of characte is also called "string".

A character array is always terminated by null character (\0) → (Backslash & zero)

Declaring a string :-

```
char str[20];   // str is character array which
                   can store maximum 19 character
                   because last character is '\0'
```

String Initialization :-

```
char str[25] = "Programming Language";
```

OR

```
char str[25] = { 'P','r','o','g','r','a','m','m','i','n','g',
                 ' ','L','a','n','g','u','a','g','e','\0' };
```

OR

```
char str[] = "Programming Language";
```

String Input :-

```
scanf ("%s", str);
```

↳ s is used for string

For character array we don't need a loop like a normal array to take input.

The problem in above statement is that it takes input of various characters without space i.e no space is allowed.

eg → If user inputs

"First" → as input it is accepted

but if user inputs

"First Last" → only 'First' will be considered input & "Last" is after space so rejected.

Therefore to accept space as well we need -

```
gets (str);
```

gets is a function to accept no. characters with space.

This takes the start address of an area of memory to hold The complete input line is read in & stored in memory string

Output :-

String functions :-

Various mathematical, logical, comparison operations can be applied to strings. like -

```
char str1, str2;
S1 = S2;
S1 < S2;
S1 + S2;
```

→ Not Possible OR Gives Error

To perform all these operations, various built-in functions are defined by C developer in header file "String.h" & those functions are

1) Strlen() :-

Function used to compute length of character array

eg → `char str[] = "INDIA";`

```
l = strlen(str);
printf("%d", l);
```

O/P → 5

2) Strlwr() :-

Function to convert a string into lower case

eg → `char ch[] = "INDIA";`

```
strlwr(ch);
printf("%s", ch);
```

O/P → india

3) Strupr() :- Function to convert a string into upper case.

eg → `char ch[] = "India";`

```
strupr(ch);
printf("%s", ch);
```

O/P → INDIA

4) Strrev() :-

Features

4) Strrev() :-

Function used to reverse characters of a string.

eg → char ch[] = "INDIA";

strrev(ch);

printf("%s", ch); O/P → AIDNI

5) Strcpy() :-

Function used to copy all characters of one string to another.

eg → char s1[] = "INDIA";

char s2[10];

strcpy(s2, s1); // s1 is copied in s2

printf("%s", s2); O/P → INDIA

6) Strcat() :- Function used to concatenate (merge) 2 strings.

eg → char s1[10] = "LAN";

char s2[10] = "GUAGE";

strcat(s1, s2); // Merge s1 & s2, & result is stored in s1.

printf("%s", s1); O/P → LANGUAGE

7) Strcmp() :- Function used to compare 2 character strings.

Returns 0 if both are equal

" +ve if 1st string is greater

" -ve if 2nd string is greater

eg → char ch1[10] = "ABC"; char ch2[10] = "PQR";

char ch3[10] = "XYZ"; char ch4[10] = "ABC";

i = strcmp(ch1, ch4); O/P → 0

i = strcmp(ch1, ch2); O/P → -ve

Program:- WAP to accept a string. check whether string is palindrome or not.

Note:- Palindrome means on reversing the string we will get the same string.

eg → str = "MADAM", ← same

Reverse of str is "MADAM"

Both are same so string is Palindrome

If str = "INDIA" ← NOT same

Reverse is "AIDNI"

str is not palindrome.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
    char s1[20], s2[20];
    int n;
    printf("Enter a string");
    gets(s1);
    strcpy(s2, s1);
    strrev(s2);
    n = strcmp(s1, s2);
    if (n == 0)
        printf("\nString is Palindrome");
    else
        printf("\n String is not Palindrome");
```

Program :- WAP to calculate length of string without string built-in function.

```
#include <stdio.h>
void main()
{
    char s1[20];
    int l;
    printf("Enter a string");
    gets(s1);
    l=0;
    while (s1[l] != '\0')
    {
        l++;
    }
    printf("length = %d", l);
}
```

Program → WAP to reverse a string without using built-in function.

```
#include <stdio.h>
void main()
{
    char s1[20], s2[20];
    int a, b, l;
    printf("Enter a string");
    gets(s1);
    l=0;
    while (s1[l] != '\0')
    {
        l++;
    }
    a=0;
    for (b= l-1; b>=0; b--)
    {
        s2[a] = s1[b];
        a++;
    }
    s2[a] = '\0';
    printf("Reversed string = %s", s2);
}
```

Program → WAP to concatenate two strings into a single string without using built-in functions.

```
#include <stdio.h>
void main()
{
  char s1[30], s2[20], s3[40];
  int i, j;
  printf("Enter 1st string");
  gets(s1);
  printf("Enter 2nd string");
  gets(s2);
  j=0;
  for (i=0; s1[i]!='\0', i++)
  {
    s3[j] = s1[i];
    j++;
  }
  for (i=0; s2[i]!='\0'; i++)
  {
    s3[j] = s2[i];
    j++;
  }
  ch3[j] = '\0';
printf("Concatenated String = %s", s3);
}
```

Arrays of Strings :- OR 2-D character Array

A single character array can store string but to store multiple strings we array of character array (string).

Declare :-

```
char S[5][30];
```

5 → rowsize i.e no. of strings

30 → Column Size i.e size of a string

This can store :-

"ABC", "XYZ", "PQR", "DEF", "LMN"

5 strings of max. 30 size.

Initialization :-

```
char S[5][10] = { "Cow", "Cat", "Dog", "Rat"
                  "Goat" };
```

Memory Representation :-

S[0]	C	o	w	\0						
S[1]	C	a	t	\0						
S[2]	D	o	g	\0						
S[3]	R	a	t	\0						
S[4]	G	o	a	t	\0					

Program :-

#include

Void

Program :- WAP to enter 5 strings & display them

```
#include <stdio.h>
void main()
{
    char s[5][10];
    int i;
    printf("Enter 5 strings");
    for (i=0; i<5; i++)
    {
        scanf ("%s", s[i]);
    }
    for (i=0; i<5; i++)
    {
        printf ("\t %s", s[i]);
    }
}
```

Structure :-

An array stores multiple elements of same type like int, float or char. But what if we want to store a record which stores roll no, name & percentage. Suppose if we try to create 4 arrays for storing 4 types of records.

Roll No[]	Name[]	Address[]	Percentage[]
1	Abc	A123	79
2	xyz	x456	85

So 1st record would be accessed as

rollno[0], name[0], address[0], percentage[0]

which shows that these are different i.e not record of a single student and even processing would also be very much difficult.

A new concept has been introduced in C to store multiple records containing multiple elements of different data types & that is Structure.

Def:- A structure is a user defined data type that store multiple values of different type under a single unit. All the structure elements are stored in sequence in memory. A structure is declared using keyword 'struct'.

Declaring a structure :-

Syntax:-

```
struct [structure name]
{
[var-decl-1];
[var-decl-2];
};
```

[tag name]

void main
{

Declaring a structure :-

Syntax :-

```
Struct <structure_name>
{
   <var-decl-1>;
   <var-decl-2>;
};
void main()
{
  Struct <structure-name> <str-variable>;
   :
}
```

Tag Name → <structure_name>
{ <var-decl-1>; <var-decl-2>; } → Members
<str-variable> → Structure variable

eg →

```
Struct student
{
   int rn;
   char nm[20];
   char add[30];
   float per;
};
void main()
{
  Struct student str;
   :
}
```

OR

Syntax :-

```
Struct <structure-name>
{
   <var-decl-1>;
   <var-decl-2>;
} <str-variable>;
void main()
{
   :
}
```

eg →

```
Struct student
{
   int rn;
   char nm[20];
   char add[30];
   float per;
}str;
void main()
{
   :
}
```

* Accessing Structure Members :-

We can refer to any structure element using structure variable as-

Syntax:- `<structure variable> . <element> ;`

eg → stu.rn;
stu.nm;

* Initialization of Structure :-

eg →

```
Struct student
{
   int rn;
   char nm[20];
   char add[20];
   float per;
} stu = { 1, "abc", "a123", 79 };
void main()
{
   :
}
```

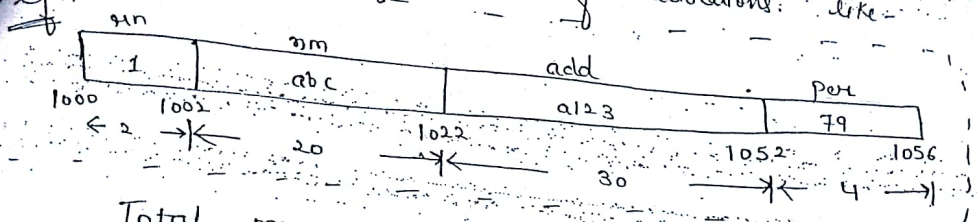
OR

```
Struct student
{
   int rn;
   char nm[20];
   char add[20];
   float per;
};
void main()
{
Struct Student stu =
   { 1, "abc", "a123", 79
   :
}
```

Memory Allocation by structure :-

All the elements of a structure are stored in contiguous memory allocations. like -

eg →



Total memory allocated by structure =

$2 + 20 + 30 + 4$

$= 56$

This can be calculated in a program with the help of a function →

eg → `Sizeof (stu);` → 56

OR

`Size of (struct student);` → 56

Array & Structure :-

We can also create array of structure to store multiple records.

eg :-

```
Struct student
{
    int rn;
    char nm[20];
    char add[30];
    float per;
};
void main()
{
struct student stu[5];
    :
}
```

OR

```
Struct student
{
    int rn;
    char nm[20];
    char cold[30];
    float per;
} stu[5];
void main()
{
    :
}
```

Program :- WAP to store multipe items record as - item code, item, name, item price, item net price

Item net price = Item price - discount. &

discount = 10 % → Item price > 1000

5 % otherwise

Source Code :-

```
#include <stdio.h>
struct item
{
    int ic, ip;
    char inm[20];
    float inp;
};
```

WAP to create a structure & display

Source Code :-

#include <str

{

Aim :- WAP to create a structure student, enter values into a structure & display the record.

Source Code :-

```
#include <stdio.h>
void main()
{
  struct student
  {
    int rn;
    char nm[20];
    char add[30];
    float per;
  };
  void main()
  {
    printf ("\nEnter Roll No");
    scanf ("%d", &rn);
    fflush (stdin);
    printf ("Enter Name");
    gets(&nm);  ---> fflush (stdin);
    printf ("Enter Address");
    gets (&add);
    printf ("Enter Percentage");
    scanf ("%f", &s.per);
    printf ("Roll No = %d", s.rn);
    printf ("Name = %s", s.nm);
    printf ("Address = %s", s.add);
    printf ("Percentage = %f", s.per);
```

2.

another eg?

```
void main()
{
  struct item it[10];
  int k;
  float d;
  for (k=0; k<10; k++)
  {
    printf("\n Enter Item Code, Name & Price");
    scanf("%d", &it[k].ic);
    fflush(stdin);
    gets(it[k].inm);
    scanf("%d", &it[k].ip);
  }
  for(k=0; k<10; k++)
  {
    if (it[k].ip > 1000)
      d = 0.1;
    else
      d = 0.05;
    it[k].inp = it[k].ip - d * it[k].ip;
  }
  for (k=0; k<10; k++)
  {
    printf("\n Item Code = %d", it[k].ic);
    printf("\n Item Name = %s", it[k].inm);
    printf("\n Item Price = %d", it[k].ip);
    printf("\n Item Net Price = %f", it[k].inp);
  }
```

When a structure is used inside another structure then it is called nested structure

eg →
```
struct address
{
    int hno;
    char area[20];
    char city[20];
};
struct student
{
    int rn;
    float per;
    char nm[20];
    struct address add;
};
void main()
{
  struct student stu;
  scanf("%d", stu.rn);
  scanf("%d", stu.add.hno);
  gets(stu.add.area);
  printf("House No = %d", stu.add.hno);
  printf("Area = %s", stu.add.area);
  printf("Roll No = %d", stu.rn);
}
```

Structure & functions

We can also pass structure to a function. For this we don't need to pass individual variables but structure variable is passed.

eg →

```
#include <stdio.h>
Struct employee
{
    int ec;
    char enm[20];
    int es;
    float ns;
};
void main()
{
    Struct employee emp;
    void compute (Struct employee);
    printf ("\n Enter Employee Code, Name & Basic Salary");
    scanf ("%d", &emp.ec);
    fflush (stdin);
    gets (emp.enm);
    scanf ("%d", &emp.es);
    Compute (&emp);
}
void compute (Struct employee emp)
{
    float da, hra;
    da = 0.20 * emp.bs;
    hra = 0.10 * emp.bs;
    emp.ns = emp.bs + da + hra;
    printf ("\n Net Salary = %f", emp.ns);
}
```

An array of structure can also be passed through a function.

eg →

Structure & Pointers :-

We can also store address of various structure elements in pointer by only storing address of structure variable in pointer.

eg → Declaration

```
Struct student
{
    int rn;
    char nm[20];
    char add[30];
    float per;
} stu;
void main()
{
Struct student *ptr;
    ptr = &stu;
    :
}
```

OR

```
Struct student
{
    int rn;
    char nm[20];
    char add[30];
    float per;
} *ptr;
Void main()
{
    :
}
```

Accessing elements through Pointer :-

```
ptr -> rn;
ptr -> nm;
```

→ Symbol is combination of hyphen (-) & greater than (>)

Union

A union is a structure all of whose members share the same storage. The amount of storage allocated for a union is sufficient to hold its largest member. At any given time, only one member of the union may actually reside in that storage. The way in which a union's storage is accessed depends on the member name that is employed during the access.

A union is identified in C through the use of keyword "union" in place of the keyword "struct". Virtually all other methods for declaring and accessing unions are identical to those for structures.

* Declaration :-

```
union student
{
    int rn;
    char nm[20];
    char add[30];
    float per;
};
void main()
{
    union student s;
    :
}
```

OR

```
union student
{
    int rn;
    char nm[20];
    char add[30];
    float per;
} s;
void main()
{
    :
}
```

If the same structure & union of same variables are created. Then:

Structure →	sizeof(str);	→ 56	(Total size of all members)
Union →	sizeof(s);	→ 30	(Maximum size from all variables)

Difference b/w Structure & Union:

1. Union occupy less space & Structure occupy more space for same no. of elements
2. In Union maximum size from all variables would be taken to allocate memory & In structure sum of all members memory is allocated.
3. With union we can refer to only one element at a time. but With structure we can refer to all elements simultaneously.

Enumerated data type

Enumerated data type helps us to attach string with numeric value for improving the readability of the program. An enumerated data type is created by using keyword enum.

Syntax:-

```
enum tag_name { member 1, member 2 ..... } variable 1, variable 2 ... ;
```

Here tag_name or variable may be ommited OR both may be present.
But atleast one of them must exist.

`enum tag_name` → specifies user-defined type.

variables of this user defined type can be declared like this:

```
enum tag_name var1, var2 ;
```

These variables can basically point to any member of the enum like this:

```
var2 = member1 ;
```

Eg →

```
enum days {Mon, Tues, Wed, Thurs, Fri, Sat, Sun};
enum Start ;
Start = Tues ;
```

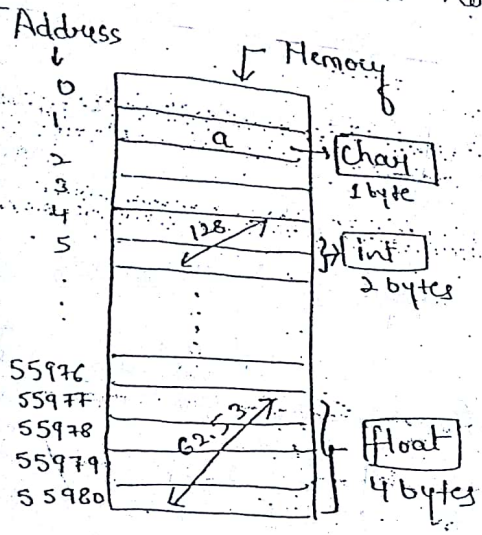
All the members are assigned integer values starting from 0. But can also change this -

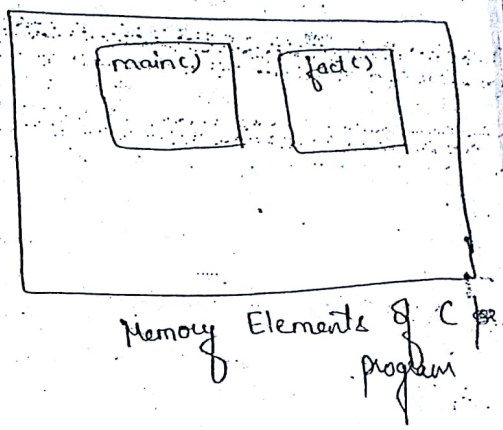
```
enum Colors { red=1, green, Blue };
```

Ho Red=1, green = 2 & Blue = 3

POINTERS

Let us have a look how variables are stored in memory





Memory Elements of C program

To access a value of a variable we can

→ Use variable name like int a ; a → variable name

→ Use Address of variable i.e address of a

Pointer def :- Pointers are data items used to store address of any variable.

i → variable name
5 → value
25798 → Address

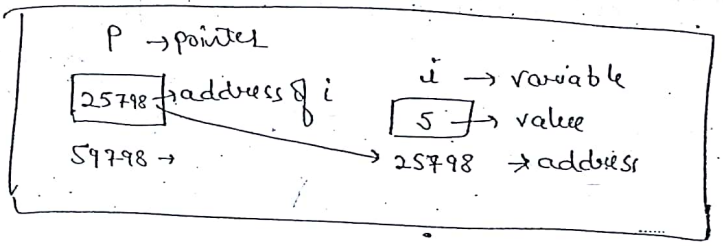
p → pointer name
25798 → Address of i
59798 → Address of pointer variable

As you can see pointer variable contains address of a variable rather than any integer float or character value.

As pointer stores the address of a variable so we can use a pointer to access value instead of using its name.

The name pointer is given for the simple reason that, by storing an address they 'point' to a particular point in memory. The pointer itself holds the address & that address holds a value.

eg →



* Declaring a pointer :-

Syntax :- `datatype * pointer_variable;`

Here datatype is type of variable whose address is to be stored in pointer.

eg → `int *ip;` // ip pointer stores address of & (integer) whose value

`float *fp;`

`char *cp;`

* **Assigning address of variable to pointer :-**

After declaration of a pointer, it contain address 0 or some random address so if use this, we will get unpredictable result.

Next step after declaration is assigning the address of ~~that~~ variable to pointer.

Syntax :-

```
pointer_name = & variable_name ;
```

& (Ampersand) symbol is used to refer address.

eg →

```
int i ;
int * ip ;      → Declaring a pointer.
ip = & i ;      → Assigning address of (integer) to ip (pointer).
```

OR

```
int * ip = & i ;   → combining declaration & assignment of address
```

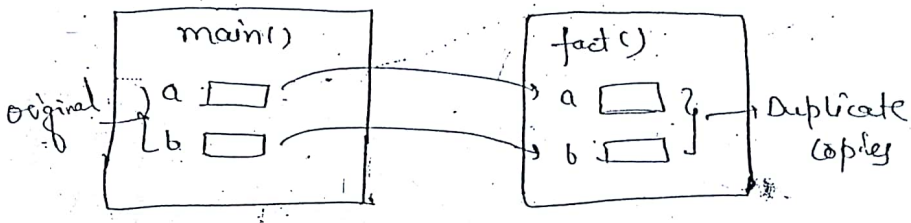
Value at that address :-

```
printf ( " %d ", & * ip)
                  ↳ value at address stored in ip
```

Passing Pointer to a function

In functions, when we pass [illegible] that method is called CALL BY VALUE but if pointers are passed in functions, the method is called CALL BY REFRENCE (Refrence means pointing the address).

In Call By Value, method original variables are not passed to the function that means we only provide a duplicate copy of variable like -

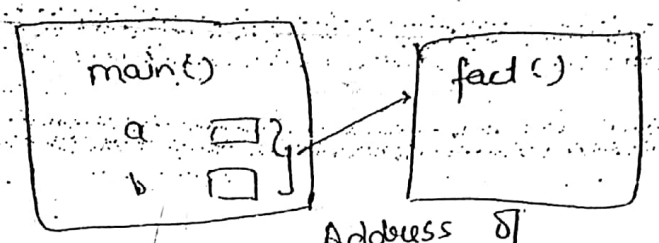


As duplicate copies are passed so any changes made in that variable are not reflected back.

eg, changes made in a fact function is not reflected back in a main function.

In Call By

In Call By Reference method, original copies of variables are passed to the function so any changes made are reflected back.



Address of these variable are being provided to fact.

Program :- WAP to swap two no's using Call by Value & Call By Reference

Call By Value :-

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a, b; void swap (int, int);
    clrscr();
    printf ("\nEnter 2 integer no's to be swapped
    scanf (" %d %d", &a, &b);
    printf ("\nValues Before swapping");
    printf (" %d\t%d", a, b);
→   swap (a,b);
    printf ("\nValues After swapping");
    printf (" %d\t%d", a, b);
```

```
void swap (int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
```

a = 10

t = a → t

a = b → a

b = t → b

so now a = 50 &

i.e values are swapped or interchan

O/P:-

Enter 2 integer no's to be swapped

10

50

Values Before Swapping

10

50

Values After Swapping

10

50

In above program values of a & b are being swapped in "swap" function but these values are not reflected in main because here we used call by value i.e duplicate copies are being provided & therefore original values remains unchanged.

Call By Re

#incl

Call By Reference :-

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a, b;
    void swap (int *, int *);
    clrscr();
    printf("Enter 2 integer no's");
    scanf ("%d %d", &a, &b);
    printf ("\nBefore Swapping");
    printf ("%d \t %d", a, b);
    swap (&a, &b);
    printf ("\n After Swapping");
    printf ("%d \t %d", a, b);
    getch();
}
```

→ Pointers are passed to functions i.e addresses of variable

→ Calling swap function

```
void swap (int *ap, int *bp)
{
    int t;
    t = *ap;
    *ap = *bp;
    *bp = t;
}
```

→ ap, bp → integer pointers containing address of a & b.

→ storing value at address stored in ap pointer to t variable.

Pointers

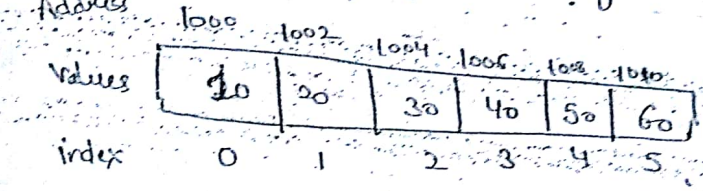
Output :-

```
Enter 2 integer No
10
50
Before swapping
10
50
After swapping
50
10
```

Here we used Call By Reference i.e. passed the address of variables to function that means provided the original copy to the function hence changes made in swap function are reflected in main function.

Pointers & Arrays

Arrays are data item used to store multiple values of same type under a single name



& pointers are used to store address.

so in Array we don't create the same number of pointer variable as the size of array. i.e for storing address of every different element, a different pointer is not used. Instead, a single pointer is sufficient for storing the address of 1st element of pointer & as in array all elements are stored contiguously so address of next element can be obtained by adding size of data type (i.e 2 for int, 4 for float) in 1st element address.

eg → ptr → pointer

→ address of 1st element

15928

To get address of 2nd element → 1000 + 2 = 1002

3rd element → 1000 + 4 = 1004

Functions Returning Pointers

```
ptr = &ar[0];
```

eg:-

```
printf ("%d", ar[0]);          10
printf ("%u", &ar[0]);         1000
printf ("%u", ar);             1000
printf ("%u", ar+1);           1002
printf ("%u", ptr);            1000
printf ("%u", *ptr);           10
printf ("%u", ptr+1);          1002
```

Program:- WAP to store 5 elements in an array. Display them with & without pointers.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[5];
    int *ptr;
    printf ("Enter 5 elements");
    for(i=0, i<5; i++)
        scanf ("%d", &a[i]);
    printf ("Without pointers");
    for (i=0; i<5; i++)
        printf ("\n %d", ar[i]);
    ptr = &ar[0];
    printf ("With pointers");
    for (i=0; i<5; i++)
    {
        printf ("\nValue = %d", *(ptr+i));
        printf ("\n Address = %u", (ptr+i));
    }
}
```

(ptr+1) → address of next element

int

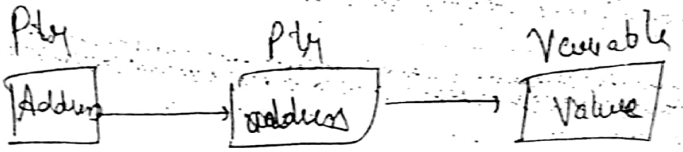
Next element address

Functions Returning Pointers :-

```
int *address()
{
    int a;
    int *p;
    p = &a;
    return p;
}
void main()
{
    int *q;
    q = address();
    printf("%u", q);
}
```

Chain of Pointers

A pointer to a pointer is a form multiple indirection, or chain of pointers.



Q →

```
int var;
int *ptr;
int **pptr;
var = 3000;
ptr = &var;
pptr = &ptr;
```

Void pointers :-

Pointer Increments & Scale Factors :-

Data Type	Scale Factor
short	2
int	2
long	4
char	1
float	4
double	8
long double	10
void	0

It means short type pointer variable updates pointer by a factor of 2 similarly float by 4 ..

Pointer Expressions :-

```
int arr = { 10, 20, 30, 40, 50 };
```

arr[0] → 10

```
int *p;
p = &arr;
```

o/p

→ printf("%u", p); — Address of 1st element of array

→ printf("%u", p+1); — Address of 2nd element of array

→ printf("%d", *p) — Value of 1st element

→ printf("%d", *(p+1)); — Value of 2nd element

1) Incrementing / Decrementing pointer :-

p++
++p
} - Incrementing pointer to contain the address of next element

arr

10	20	30	40	50

1000

p

1000

2058

After Incrementing (p+1)

p

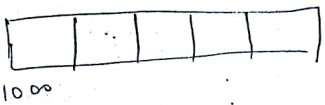
1002

2058

p--
--p
} decrementing pointer to contain the

2) Adding integer pointer with number :-

```
int a[5];
int *p;
p = &a;
p = p + 3;
```



p = 1000
p = 1006

3) Subtracting integer value from pointer :-

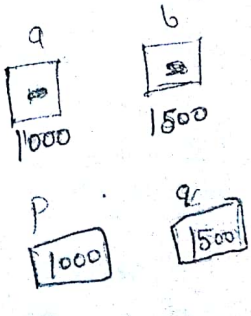
In above eq

p = p - 2

as because p was 1006
now p = 1002

4) Differencing Pointers :- (Subtracting 2 pointers)

```
int a, b;
int *p, *q;
p = &a;
q = &b;
printf("%d", q-p);
```



→ 500

5) Comparing 2 pointers :-

Considering Previous example

```
if (p > q)
    printf ("p is far from 0");
else
    printf ("0 is far from p");
```

Following operators can be used on operators

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equals
!=	Not equal

6) Divide & Multiply operations :

```
q = p/4;
```

or

```
q = p * 4;
```

} - Not possible

This will give an error Illegal use of pointer

Pointer & Structure

(1) Struct Student
{
int rin;
char nm[20];
float per;
} stu;

→ Way of creating a structure to hold multiple items with same nam

(2) Struct student * ptr;] → declaring a pointer of structure type

(3) p = &stu;] → Assigning address of stu to pointer

(4) ptr → rin;] → To Access any element of structure through pointer.

as with structure we use (.)

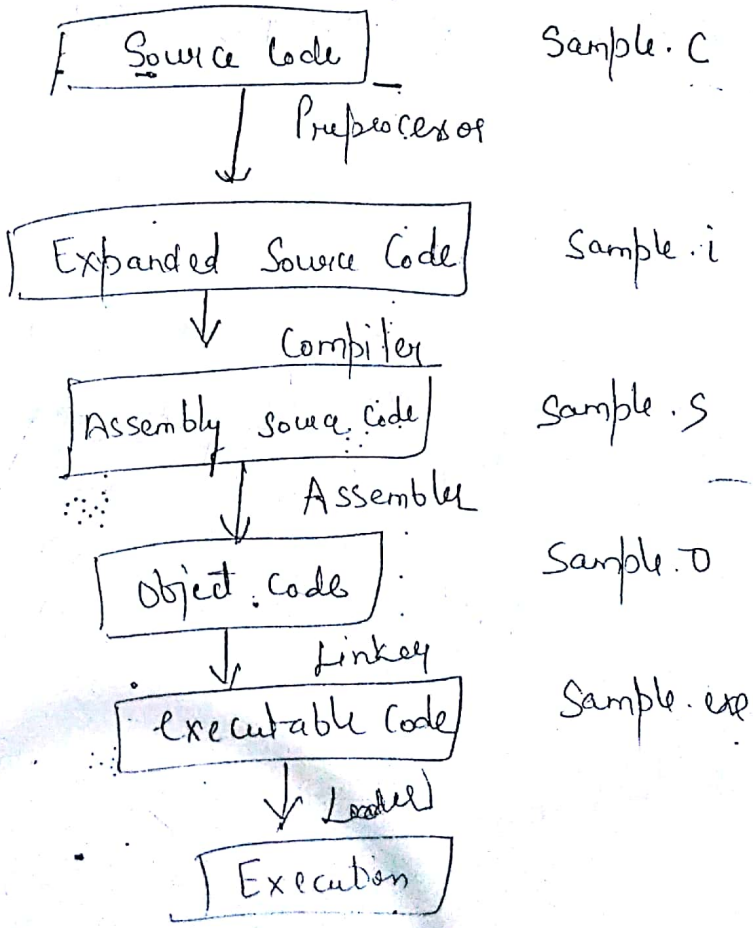
stu.rin;

here (→) symbol is used.

ptr → per;

Preprocessor

- Macro — #define
- File Inclusion — #include
- Conditional Compilation — #ifdef, #endif, #if, #else, #ifndef
- other directive



#if , #else , #endif

if clause statements executes if given condition is True

otherwise else clause statements executes.

eg→

```
#include <stdio.h>
#define a 100
void main()
{
#if (a == 100)
    printf ("A is equal to 100");
#else
    printf ("A is not equal to 100");
#endif
return 0;
}
```

O/P → A is equal to 100

→ "#ifdef" ... directive checks whether particular macro is defined or not. If it defined "if" clause statements execute otherwise "else" clause

eg →

```
#include <stdio.h>
#define RAJU 10
int main ()
{
  #ifdef RAJU
    printf ("Raju is defined");
  #else
    printf ("Raju is not defined");
  #endif
  return 0;
}
```

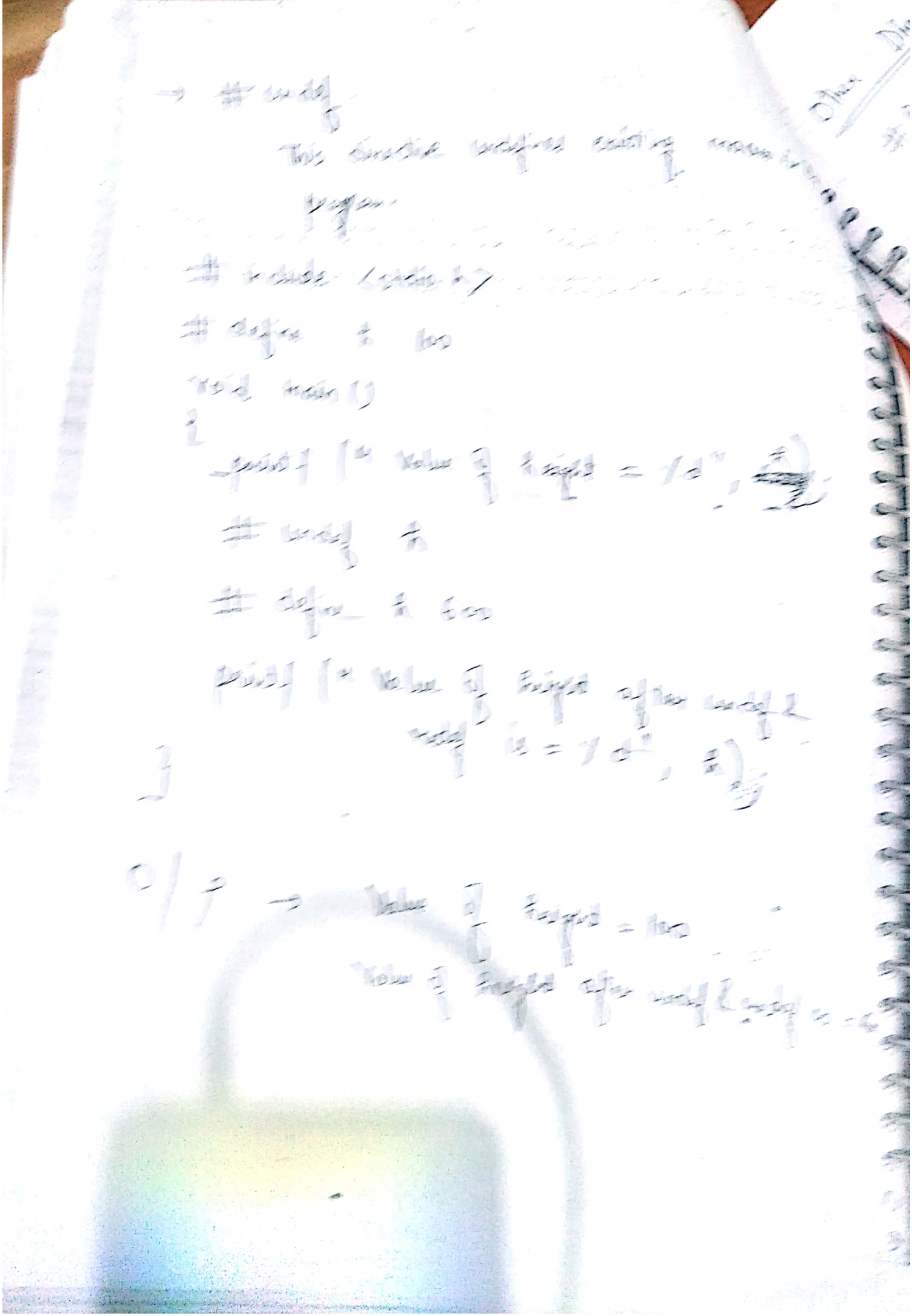
O/P → Raju is defined

#ifndef" exactly acts as reverse as if-def directi
if macro not defined - "if" clause execute
Otherwise, else clause executed

```
#include <stdio.h>
#define RAJU 100
int main()
{
    #ifndef SELVA
        printf ("Selva is not defined");
    #else
        printf ("Selva is defined");
    #endif
    return 0;
}
```

O/P → Selva is not defined

particular if it is not, statements execute



Scanned by CamScanner

Other Directive

Pragma - Used to call a function before & after main function

eg.
```
#include <stdio.h>
void function1();
void function2();
#pragma startup function1
#pragma exit function2
void main()
{
    printf("Main function");
}
void function1()
{
    printf("function 1 is called before main function");
}
void function2()
{
    printf("Function 2 is called after main");
}
```

o/p - function 1 is called before main function
main function
function 2 is called after main

Pragma Command	Description
# Pragma startup <fun name>	This directive executes function before main
# Pragma exit <fun name>	This directive executes function after main i.e just before the termination of the program
# pragma warn -rvl	If function does not return value, then warnings are suppressed while compiling
# pragma warn par	If function does not use passed function parameter, then warnings are suppressed
# pragma warn -rch	If non reachable code is written inside a program, such warnings are suppressed.

Preprocessor Directive Block :-

C preprocessor is a program that processes our source program before it is passed to the compiler. Before a C program is compiled it is passed through another program called "Preprocessor". C Program is often known as "Source Code". The preprocessor works on the source code and creates "Expanded Source Code". Suppose if program is saved with name prg1.c, then the expanded source code gets saved with name prg1.I. Now this expanded source code is sent to the compiler for compilation.

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begins with a # symbol. The directives are most often placed at the beginning of a program.

Various Preprocessor Directives are :-

a) Macro Expansion
b) File Inclusion
c) Conditional Compilation
d) Miscellaneous Directives

Macro Expansion:

Macro expansion or substitution is done by using #define preprocessor directive.

Syntax:-

```
# define <identifier> <value>
```

eg →

```
# define MAX 25
```

This statement is called 'macro definition' or just 'macro'. During preprocessing, the preprocessor replaces every occurance of MAX in the program with a value 25.

eg →

```
# include <stdio.h>
# define MAX 25

void main()
{
    int i;
    for (i = 0; i < MAX; i++)
    {
        printf ("\n %d", i);
    }
}
```

When we compile this program, before the source code passes to the compiler, it is examined by the C processor for any macro definations. When it see the "# define" directive it goes through the entire program in search

places the macro template with the appropriate macro expansion. Only after this procedure, program is handed over to the compiler.

Advantages of Using Macro :-

1) It makes the program easy to read

eg → the phrase "\x1B[2J" causes the screen to clear. but most of us are not aware about it so using a macro for this phrase is easy to read & understand.

```
#define CLEARSCREEN "\x1B[2J"
```

2) Most important advantage of using macro is suppose a value is being used multiple times in a program & let it gets changed so we need to change that value on all of its occurences in our program which is very much time consuming and if we forget one location our program will be erroneous. so better way is to create macro and using this we just need to change value while defining macro and whole program will get changed.

eg.
```
#include <stdio.h>
#define MAX 10
void main()
{
    int i;
```

```
for(i=0; i< MAX ; i++)
{
    // Input Block
}
for (i=0 ; i<MAX ; i++)
{
        // Processing Block
}
for (i=0; i< Max ; i++)
{
        // Output Block
}
}
```

Now Suppose the value of MAXimum elements of array has been changed from 10 to 20 so we just need to change one statement.

→ `#define MAX 20`

Some Important points about Macro :-

1) There is no semicolon after macro defination.
2) It is declared in the beginning of the program before ~~void~~ main function.
3) There may or may not be space between # space define.
4) The name of identifier should be in capital to distinguish it from normal variable.

```
# define ARANGE ( a>25 AND a<50)
```

Macro could be used to replace an entire C statement as

```
# define DISPLAY (" Any Printing Statement ");
```

Macro can have arguments, just as functions can -

eg→

```
# include <stdio.h>
# define AREA(x) (3.14 * x * x)
void main()
{
    float R1 = 5.21 , R2 = 7.92 ; a;
    a = AREA (R1);
    printf (" \n Area of circle = %f ", a);
    a = AREA (R2);
    printf (" \n Area of circle = %f ", a);
}
```

Macro Versus Functions

As like above macro, a function can be written to calculate area. Though macro calls are 'like' function calls, but both are not same.

In a macro call, the preprocessor replaces the macro identifier template with its expansion. But in a function call, the control is passed to a function along with arguments, calculations are performed in function & result (if any) is returned back from the function.

File Inclusion :-

The second preprocessor directive is # include. This causes one file to be included in another file and hence can use the function declared in one file in some other file without redefining it. This could be look like this :-

Syntax :-

```
# include "filename"
# include <filename>
```

Using above syntax in the begining of program causes the entire contents of the filename to be inserted into the source code.

Advantage :-

1) There are some functions & some macro definations that we need almost in all the progeams we write like :- input, output functions so these functions are stored in a file and that file is included in every program & we can call those functions easily without the need of defining them again.

2) If we have a very large program, the code is divided into several different files, each containing a set of related functions. It is a good progeamming practice to keep different sections of a large program seperate. These files are included at the beginning of main program file.

include <file name>

eg → #include <stdio.h>

This syntax is used for referring to the standard system header files. Using < > brackets compiler begins searching for the file in the standard INCLUDE directory.

Header files basically contains the declaration of various functions while Library files contain corresponding defination. We just need to include the header file in the beginning of any program using "include" preprocessor directive and this header file automatically invoke respective library file.

eg → for calculating square root of a no. we have a function "sqrt()".
its declaration is in header file math.h
its defination is in library file CS.LIB

include " file name "

eg → #include "myfile.h"

This Syntax is used for local header files (ie created by user) but can be used for standard header files. Using " " compiler begins searching for the file in current working directory as well as standard include directory.

Let Us C

Page 256

Conditional Compilation

```
#ifdef LABEL
    Sta 1;
    Sta 2;
#endif
```

Sta 1 & Sta 2 will get compiled only if LABEL has been defined

```
#define DEF
main()
{
    int i=2;
#ifdef DEF
    printf("\n Square=%d", i*i);
#else
    printf("\n i=%d", i);
#endif
}
```

```
#define MESS(m) printf("\n%s",m);
main()
{
    MESS("Printer error");
}
```

```
#define COND (a>=65 && a<=90)
main()
{
    if (COND)
    else
}
```

Storage classes

Till now we declare a variable using data type which is not sufficient to answer the questions

→ Where the variable would be stored (memory or Register)
→ How long the variable would exist (life)?
→ What would be the region of existence of a variable (Scope)
→ What are the default values of a variables?

To answer all these questions we have storage classes. C provides four storage classes specifiers

→ Automatic ✓
→ External ✓
→ Static ✓
→ Register ✓

Syntax:- <storage class specifier> <data type> <variable name>;

eg → static int a;

We have not used any of the storage class in our program till now. Thus the variables have default storage class.

The description of all types of storage classes are given on next page -

1) Automatic Storage Class

By default any variable declared in any function are automatic. We also use keyword 'auto' to explicitly specify its storage class.

eg→ auto char c;
auto float f;

Features of a variable defined to have an automatic storage class are:

Storage	→ Memory
Default Initial Value	→ Any unpredictable value which is often called garbage value.
Scope	→ Local to the block in which the variable is defined.
Life	→ Till the control remains within the block in which variable is defined

eg→

```
void main()
{   int a;
    void function();
    printf("%d", a);        → 5779
    a=10;
    printf("%d", a);        → 10
    function();
}
void function()
{   int x=5;
    printf("%d", x);        → 5
    printf("%d", a);        → Error → Scope
```

O/P

(Garbage Value) Initial Value

External Storage class OR GLOBAL STORAGE CLASS

An external variable is declared

→ declaring a variable outside all functions

→ declaring a function variable within a function using keyword 'extern'

Features of external class variable :

Storage	→ Memory
Default Initial Value	→ Zero
Scope	→ Global
Life	→ As long as the program's execution doesn't come to an end

eg →

```
#include <stdio.h>
int x = 10;                          → External variable
void main()
{
    int a = 5;                       → Automatic variable
    void function1();
    void function2();
    printf("%d", a);                 → Prints 5
    printf("%d", x);                 → Prints 10
    function1();
    printf("%d", x);                 → Prints 20 as lifetime of x is whole program
    function2();
}
void function1()
{
    x = 20;
    printf("%d", x);                 → Prints 20
}
void function2()
{
    int x = 70;                      → External variable is overridden
}
```

(iii) Static

This storage class takes the features of local and global variables. These type of variables are created using 'static' keyword.

eg → `static int i;`

Features of static variables are :

Storage	→	Memory
Default Initial Value	→	Zero
Scope	→	Local to the block in which the variable is defined.
Life	→	Value of the variable persists b/w different function calls

eg →

Automic Storage class

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

O/P:-

```
1
1
1
```

Static Storage class

```
#include <stdio.h>
void main()
{
 void incr();
 incr();
 incr();
 incr();
}
void incr()
{
 static int a=1;
 printf("\n %d", a);
  a= a+1;
}
```

```
1
2
3
```

As because value of Static program will remain for the duration of whole

Register Storage Class :-

A register storage class variable is declared using keyword 'register'.

Features of register storage class variable are :-

Storage	→ CPU Registers
Default Initial Value	→ Garbage Value
Scope	→ Local to the block in which variable is defined.
Life	↳ Till the control remains within the block in which the variable is defined

The main advantage of storing a variable in register instead is that a value stored in a register can always be accessed faster than the one stored in memory & hence increases the processing speed. Therefore, if a variable is used at many places in a program it is better to declare its storage class as register.

eg:-
```
#include <stdio.h>
void main()
{
    register int i;          // i is stored in register rather than memory.
    for(i=1 ; i<=10 ; i++)
        printf("\n%d", i);
}
```

Here, i is declared as register storage class variable, but still it is not sure that value of i would be stored in a register. The reason is there are very limited number of CPU registers, they may be busy doing some other work so in that case i will [illegible]

Bit Wise Programming :-

One of the unique feature of C programming language is that we can perform low level programming on it along with high level programming. Therefore, C language is also called middle-level programming.

Bit-wise operators :-

C provides us a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. These operators can operate upon int and char but not on float & double.

Bit-wise operators are categorized into three types :-

i) Bit-wise logical operators

ii) Bit-wise complement operators

iii) Bit-wise shift operators

Arithmetic operators have higher precedence than bitwise operators. Various Bitwise operators with their symbols are shown below.

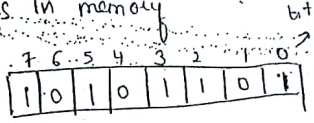
operator	Description	
&	Bit wise AND	} logical
|	Bitwise OR	
^	Bitwise XOR [Exclusive OR]	
~	Bitwise Complement	→ Complement
<<	Bitwise shift left	} Shift
>>	Bitwise shift Right	

Bit-wise logical operators :-

a) AND :- This operator is represented as &
This operator works on two operands. Both the operands must be of the same type (either char or int). Truth table of AND is shown below :-

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

Bits in memory



Uses of AND operator :-

1) To check whether a particular bit is ON or OFF

eg → A → 10101101

Check whether bit number 4 is ON (1) OR OFF (0)

So $2^4 * 1 = 16$ which is represented as

00010000

So 10101101 - Original bit
00010000 - AND MASK
-------- AND
00000000

↳ In output we got 0 so 4th bit in original bit it zero i.e OFF

but if we get 16 as output then bit would be ON.

```
#include <stdio.h>
void main()
{
    int i = 65, j;
    printf ("\n Value of i = %d", i);
    j = i & 32;          // to check 5th bit
    if (j == 0)
        printf ("\n Fifth bit is OFF");
    else
        printf ("\n Fifth bit is ON");
    j = i & 64;          // To check 6th bit
    if (j == 0)
        printf ("\n Sixth bit is OFF");
    else
        printf ("\n Sixth bit is ON");
}
```

O/P:-

```
Value of i = 65
Fifth bit is OFF
Sixth bit is ON
```

The example of this use is to check the attribute of a file stored in a directory. Meaning of each bit in the attribute byte is shown in the table

Bit Number								Meaning
7	6	5	4	3	2	1	0	
.	.	.	.	.	.	.	1	Read Only
.	.	.	.	.	.	1	.	Hidden
.	.	.	.	.	1	.	.	System
.	.	.	.	1	.	.	.	Volume Label Entry
.	.	.	1	.	.	.	.	Sub-directory Entry
.	.	1	.	.	.	.	.	Archive bit
.	1	.	.	.	.	.	.	Unused
1	.	.	.	.	.	.	.	Unused

Suppose To check whether a file is system file or not then check its 2nd bit in attribute byte with the help of previous program

2) Changing The Status of the bit :-

i.e switch off a particular bit in a byte.

eg → Original Byte is - 00000111 , To turn off bit Number @ 2

Take AND MASK as 11111011

i.e 2nd bit as 0 and all other as 1

Bit - Wise

```
00000 111   → original bit pattern
1111 011    → AND MASK
-----------
00000011    → Output
```

Here second bit is switch off.

OR :-

This is represented as | OR operator works on two operands. This returns 1 if either of the two bits is 1.

Truthtable of OR :-

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

Uses of OR :-

1) To Set a particular bit to ON

eg. Suppose original bit pattern is 00010111

To put ON bit number 93 take OR mask as:

00001000

Now apply OR operation

```
000 10111   → original
00001000    → OR MASK
-----------
00011111
   └→ 3rd bit is ON.
```

Bit-wise Complement Operator:

The operator that complement (invert) the bit is called Bit-wise complement operator. This operator is represented as ~ This works on a single operand and hence a unary operator.

eg → a = 12 i.e 12 → 0000 1100

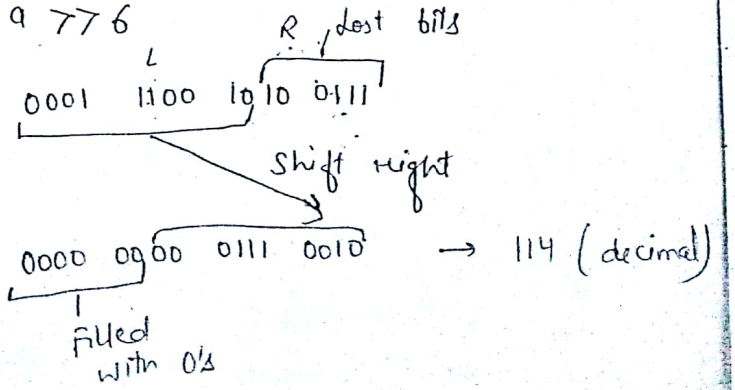
b = ~a i.e 243 → 1111 0011

(iii) **Bit-wise shift operators:-**

Right shift:-

The bit wise right shift operator causes all the bits in the first operand to be shifted to the right by the number of positions indicated by the second operand. The rightmost bits in the original bit pattern will be lost. The leftmost bit positions that become vacant will be padded with zeroes.

eg → b = a >> 6



XOR :-

XOR (Exclusive OR) is represented as ^. This operator works on 2 operands. This returns 1 if only one of the bits is 1 i.e both bits should be different.

Truth table of XOR is:

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

Use of XOR :-

XOR operator is used to toggle a bit ON or OFF.
A number XORed with another number twice gives the original number.

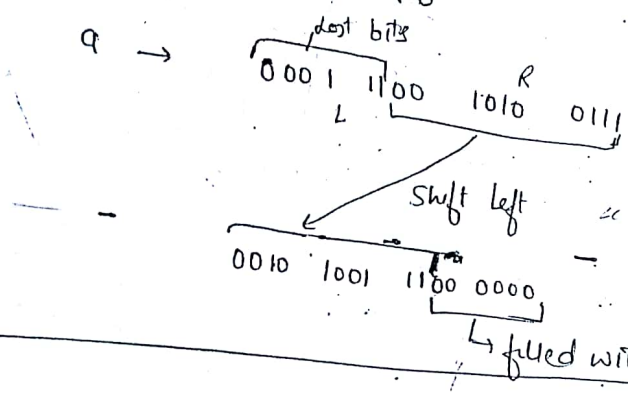
```
#include <stdio.h>
void main()
{
    int b = 50;
    b = b ^ 12;
    printf("\n %d", b);
    b = b ^ 12;
    printf("\n %d", b);
}
```

O/P → 62
50 → get the 50 after XORing it with 12 twice.

Left Shift :-

The bitwise left shift operation causes all the bits in the first operand to be shifted to the left by the number of positions indicated by the second operand. The most significant bits are lost as the number moves left, and the vacated least significant bits are zero.

b = a << 6



Bit fields in Structures :-

If in a program a variable is to take only two values 1 and 0, we really need only a single bit to store it.

0-3 value : → 2 bits are sufficient

0-7 — → 3 —

So why waste an entire integer when one or two or three bits are sufficient? Suppose there are several variable whose maximum values are small enough to pack them into a single memory locations, we can

integer.

eg → To store following record of an employee

a) Gender : Male or female

b) Marital Status : married, ~~or~~ single, divorced or widowed

c) Hobbies : Anyone from 8

d) Scheme : Anyone from 15

So we need 1 bit to store Gender

— 2 — Marital Status

— 3 — Hobby

— 4 — Scheme

Together we need 10 bits, which means we can pack all this information into a single integer.

So declare a structure :

```
Struct employee
{
    unsigned gender : 1;
    unsigned mar_stat : 2;
    unsigned hobby : 3;
    unsigned scheme : 4;
};
```

The colon (:) in the above declaration tells the compiler that we are talking about bit fields and the number after it tells how many bits to allot for the field.

```
#include <stdio.h>
#define MALE 0;
#define FEMALE 1;
#define SINGLE 0;
#define MARRIED 1;
#define DIVORCED 2;
#define WIDOWED 3;

Void main()
{
    Struct employee
    {
        unsigned gender : 1;
        unsigned mar_stat : 2;
        unsigned hobby : 3;
        unsigned scheme : 4;
    };
    struct employee e;
    e.gender = MALE;
    e.mar_stat = DIVORCED;
    printf ("\n Gender = %d", e.gender);
    printf ("\n Maital Status = %d", e.mar_status);
    printf ("\n Bytes occupied by e = %d", sizeof(e));
}
```

O/P →

```
Gender = 0
Maital Status = 2
```

Types of Errors -

1) → Compile - Time

2) → Run - Time

1) → The errors that occur in a program during compilation time are called compile time errors. Due to these errors we can't continue further. Therefore we must first correct them to obtain the output

eg → missing semicolon ;
missing curly brackets { }
Syntax Error
Use single quote for string

2) → The errors which occur in a program at run time are called Run Time errors. These errors occurs due to a statement which is impossible for the compiler to solve.

Ex. Divide by zero, logical Errors

Dynamic Memory Allocation

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr=(cast-type*)malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of n elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int n,i,*ptr,sum=0;
  printf("Enter number of elements: ");
  scanf("%d",&n);
  ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
  if(ptr==NULL)
  {
    printf("Error! memory not allocated.");
    exit(0);
  }
  printf("Enter elements of array: ");
  for(i=0;i<n;++i)
  {
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
  }
  printf("Sum=%d",sum);
  free(ptr);
```

return 0;
}

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
      printf("Error! memory not allocated.");
      exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

ptr=realloc(ptr,newsize);

Here, ptr is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    printf("Enter 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("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    for(i=0;i<n2;++i)
        printf("%u\t",ptr+i);
    return 0;
}
```

Standard Library Functions

... functions that are built in the memory ... computer for performing any specific task are called *standard library or build-in functions*. We can use these functions by using appropriate header file. The functions that accept input & display outputs are called **I/O** functions.

Types I/O functions: All the input output functions are categorized into following 2 types:

1. Console I/O functions
2. Disk I/O functions

Console I/O functions: The functions that accept data from standard input device (keyboard) & display output on standard output device (monitor) are called console I/O functions. These functions are further classified into following 2 categories:

1. Formatted I/O functions
2. Unformatted I/O functions

1. Formatted I/O functions: The functions that accept data & display result at user defined location & widths are called formatted I/O functions. Different types of formatted functions are: scanf( )-for input, printf( )-for output.

The printf( ) function: This function is used to display format output on the screen.

Syntax:
printf("<control string>", arg1,arg2,...,argn);
The control string consists of following three parts:
1. Characters that will be printed as they are.
2. Conversion (format) specifications that define the output format for each item (%d,%f,%c,%s).
3. Escape sequence characters begin with \ sign.

Ex:
```
int a=7,b=5;
printf("%d%d",a,b);          O/P: 75
printf("%d\t%d",a,b);        O/P: 7       5
printf("a=%d\tb=%d",a,b);    O/P: a=7     b=5
```
The general syntax for integer number is %ww where w specifies the maximum width.

Ex: int n=345;

printf("%d",n);	O/P:			3	4	5
printf("%5d",n)	O/P:			3	4	5
printf("%05d",n)	O/P:	0	0	3	4	5
printf("%-5d",n)	O/P:	3	4	5		

Note: The minus (-) sign indicates that output will display on the left side which is right aligned by default. The zero (0) is used to fill zeros in place of leading or trailing blanks.

The general syntax for float number is %ww.pf where w specifies the maximum width & p is the precision (no. of decimal places).

Ex: float p=3.1415;

printf("%f",p);	O/P: 3.141500				
printf("%4.2f",p);	O/P:	3	.	1	4

The general syntax for displaying strings %ws where w specifies maximum width.

Ex: char ch[20]="India";
```
printf("%s",ch);      O/P: India
printf("%7s",ch);     O/P:  India
printf("%-7s",ch);    O/P: India
printf("%2s",ch);     O/P: India
```

Note: If width specified by the user is less than the actual width of a variable then complete value will display.

The scanf( ) function: This function is used to accept any kind of input from the user.

Syntax:
scanf("<control string>",&arg1,&arg2);

```
Ex: int a,b,c;
scanf("%d%d",&a,&b);          I/p: 1234 56
printf("%d %d",a,b);          O/P: 1234 56
scanf("%2d%d",&a,&c);         I/p: 1234 56
printf("%d %d",a,b);          O/P: 12 34
scanf("%d%*d%d",&a,&b,&c);    I/p: 12 34 56 78 90
printf("%d %d %d",a,b,c);     O/P: 12 56 90
```
Where * symbol is used to skip the input value.

The sprintf( ) function: This function is similar to printf( ) function except that it send the output to a string instead of monitor screen.

The sscanf( ) function: This function is similar to scanf( ) function except that it accept data from a string instead of keyboard.

Ex:
```
void main( )
{
int a=7; char ch='A'; float pi=3.14; char str[20];
sprintf(str,"%d%c%f",a,ch,pi);
printf("%s",str);                    O/P: 7A3.14
sscanf(str,"%d%c%f",&a,&ch,&pi);
printf(%d %c %f,a,ch,pi);            O/P: 7 A 3.14
}
```

2. Unformatted I/O functions: The functions that accept data & display result at computer defined location & widths are called unformatted I/O functions. Different types of unformatted functions are: getchar( ), getche( ), getch( ), gets( ), putchar( ), putch( ), puts( ).

The getchar( ): This function is used to accept a character from the standard input device.

Syntax:

<varname>=getchar( );

Ex: char ch;
ch=getchar( );

Note: There is another function called fgetchar() that perform the same function.

The putchar( ): This function is used to display a character on the standard output device.

Syntax:

putchar(<varname>);

Ex: char ch='A';
putchar(ch);

Note: There is another function called fputchar() that perform the same function.

Prog: WAP to accept a character. Convert it into lowercase if it is in uppercase & vice-versa.

Sol:

```
#include<ctype.h>
#include<stdio.h>
void main( )
{
char ch;
printf("\n Enter a character");
ch=getchar( );
if(islower(ch))
putchar(toupper(ch));
else
putchar(tolower(ch));
}
```

The getche( ): This function is used to accept a character from the standard input device & echo the input character on the screen.

Syntax:

<varname>=getche( );

The getch( ): This function is used to accept a character from the standard input device but not echo the input character on the screen.

Syntax:

<varname>=getch( );

File handling in C

Till now we are using Input-Output functions that accept input data from standard input device (Keyboard) & display output on the standard output device (monitor). But C language also provides us some disk I/O functions that helps user to accept input data from file and write result onto the file.

File/Disk I/O functions: Different types of disk I/O functions are: fopen( ), fclose( ), getc( ), putc( ), getw( ), putw( ), fprintf( ), fscanf( ).

Note: All the above functions required file pointer to access any specific file. A file pointer is created by FILE data type as: FILE *fp;

The fopen( ) function: This function is used to open a file in C program. We can read or write into the file only after file is opened.

Syntax:

<filepointer>= fopen("<filename>","<file mode>");

Where filename is the name of file that we want to open & file mode specifies the purpose of opening the file. File modes are divided into following three categories.

r - Open the file in read mode
w - Open the file in write mode
a - Open the file in append mode

Ex: fp=fopen("Student","w");

The fclose( ) function: This function is used to close any opened file. We can re-open file only after closing it. Syntax: fclose(<filepointer>);

Ex: fclose(fp);

The getc( ) function: This function is used to read a single character from the file.

Syntax: <varname>=getc(<filepointer>);

Ex: ch=getc(fp);

The putc( ) function: This function is used to write a single character onto the file.

Syntax: putc(<varname>,<filepointer>);

Ex: putc(ch,fp);

Prog: WAP to accept some characters from keyboard until user press new line character. Transfer all the characters onto the file.

Sol:

```
#include<stdio.h>
void main( )
{
FILE *fp;    char ch;
fp=fopen("test","w");
printf("\n Enter a character");
ch=getchar( );
while(ch!='\n')
{
putc(ch,fp);
printf("\n Enter another character");
ch=getchar( );
}
fclose(fp);
}
```

Prog: WAP to accept all the characters from the file. Display all the characters.

Sol:

```
#include<stdio.h>
void main( )
{
FILE *fp;    char ch;
fp=fopen("test","r");
ch=getc(fp);
while(ch!=EOF)
{
printf("%c",ch);
ch=getc(fp);
}
fclose(fp); }
```

getw( ) fuqetion: This function is used to an integer from the file.
Syntax: <varname> getw(<filepointer>);
Ex: n=getw(fp);
The putw( ) function: This function is used to write an integer onto the file.
Syntax: putw(<varname>,<filepointer>);
Ex: putc(n,fp);

Prog: WAP to accept 10 integers from keyboard. Transfer all the even numbers in the file EVEN file & odd numbers in ODD file.
Sol: #include<stdio.h>
void main( )
{
FILE *fp,*f1,*f2; int n,i;
fp=fopen("Numbers","w");
for(i=1; i<=10; i++)
{
printf("\n Enter a number");
scanf("%d",&n);
putw(n,fp);
}
fclose(fp);
fp=fopen("Numbers","r");
f1=fopen("EVEN","w");
f2=fopen("ODD","w");
for(i=1; i<=10; i++)
{ n=getw(fp);
if(n%2==0)
putw(n,f1);
else
putw(n,f2);
}
fclose(fp); fclose(f1); fclose(f2);
}

Prog: WAP to contents of a file from another file.
Sol: #include<stdio.h>
void main( )
{
FILE *f1,*f2; char ch;
f1=fopen("abc","r");
f2=fopen("xyz","w");
ch=getc(f1);
while(ch!=EOF)
{
putc(ch,f2);
ch=getc(f1);
}
fclose(f1); fclose(f2);
}

The fprintf( ) function: This function is used to write different types of data (float,int,char & string) onto the file. Syntax:
fprintf(fp,"<control string>",<var1>,<var2>);

The fscanf( ) function: This function is used to different types of data (float,int,char & m the file. Syntax:
<controlstring>",&<var1>,&<var2>);

Prog: Define a structure item for storing item no.,item name ,item price for three items. Transfer all the items in a file. Reopen the file display for all the items.
Sol: #include<stdio.h>
void main( )
{
int rn; char nm[20];
float per; FILE *fp;
fp=fopen("Student","w");
printf("\n Enter rollno,name & percentage");
scanf("%d%s%f",&rn,&nm,&per);
fprintf(fp,"%d%s%f",rn,nm,per);
fclose(fp);
fp=fopen("Student","r");
fscanf(fp,"%d%s%f",&rn,&nm,&per);
printf("\n name=%s",nm);
printf("\n rollno=%d",rn);
printf("\n percentage=%f",per);
fclose(fp); }

The fwrite( ) function: This function is used to write some block of data (array or structure) onto the file. Syntax:
fwrite(<pointer var>,size of one of item>,<no of items>,<fp>);
Where pointer variable represents the pointer of an array or structure, size represents the size of array or structure, no.of items represent the no of items to be appended & fp represents the file pointer

The fread( ) function: This function is used to read some block of data (array or structure) from the file. Syntax:
fread(<pointer var>,size of one of item>,<no of items>,<fp>);
Where pointer variable represents the pointer of an array or structure, size represents the size of array or structure, no.of items represent the no of items to be appended & fp represents the file pointer

Prog: Define a structure item for storing item no.,item name ,item price for three items. Transfer all the items in a file. Reopen the file display for all the items.
Sol: #include<stdio.h>
struct Item
{ int in;
char nm[20];
float ip;
};
void main( )
{
struct Item it; FILE *fp;
fp=fopen("Items","w");
printf("\n Enter item number,name & price");
scanf("%d%s%f",&it.in,&it.nm,&it.ip);
fwrite(&it,sizeof(it),1,fp);
fclose(fp);
fp=fopen("Items","r");
fread(&it,sizeof(it),1,fp);
printf("\n Item name=%s",it.nm);
printf("\n Item no=%d",it.in);
printf("\n Item price=%f",it.ip);
fclose(fp); }

FILE HANDLING

A file represents a sequence of bytes, does not matter if it is a text file or binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through important calls for the file management.

Opening Files

You can use the **fopen()** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

```
FILE *fopen( const char * filename, const char * mode );
```

Here, **filename** is string literal, which you will use to name your file and access **mode** can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:

```
int fclose( FILE *fp );
```

The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

Writing a File

Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string s to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example:

Make sure you have **/tmp** directory available, if its not then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

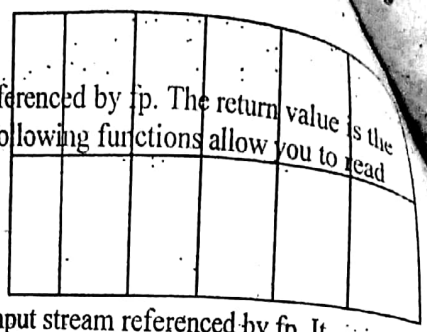
When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns **EOF**. The following functions allow you to read a string from a stream:



```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>

main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...

3: This is testing for fputs...
```

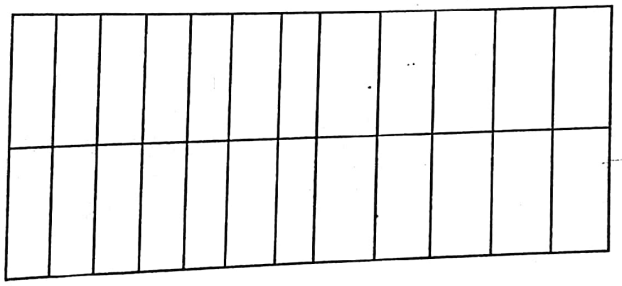
Let's see a little more detail about what happened here. First **fscanf()** method read just **This** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

Binary I/O Functions

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.



Error handling

The standard I/O functions maintain two indicators with each open stream to show the end-of-file and error status of the stream. These can be interrogated and set by the following functions:

```
#include <stdio.h>

void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

Clearerr clears the error and EOF indicators for the stream.

Feof returns non-zero if the stream's EOF indicator is set, zero otherwise.

Ferror returns non-zero if the stream's error indicator is set, zero otherwise.

Perror prints a single-line error message on the program's standard output, prefixed by the string pointed to by s, with a colon and a space appended. The error message is determined by the value of errno and is intended to give some explanation of the condition causing the error. For example, this program produces the error message shown:

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

main(){

    fclose(stdout);
    if(fgetc(stdout) >= 0){
        fprintf(stderr, "What - no error!\n");
        exit(EXIT_FAILURE);
    }
    perror("fgetc");
    exit(EXIT_SUCCESS);
}

/* Result.*/
fgetc: Bad file number
```

feof function

```
if(feof(fp))
printf("End of data.\n");
```

- Would display the message on reaching the end of file condition (File opened for reading)
  - Reports the status of the file indicated.
  - Takes file pointer as its argument.
  - Returns non zero integer if an error has been detected up to that point, during processing
  - Otherwise returns zero.

ferror function

```
if(ferror(fp) !=0)
printf("An error has occured.\n");
```

- Would display the error message if reading is not successful.

Program to illustrate error handling functions:

Code:

```
main()
{
char *filename;
FILE *fp1,*fp2;
int i,num;
fp1 = fopen("TEST","w");
for(i=0;i<=100;i+=10)
 putw(i,fp1);
fclose(fp1);
fp1 = fopen("TEST","r");
for(i=1;i<=10;i++){
 num=getw(fp1);
if(feof(fp1)) {
 printf("\nRan out of data.\n");
 break; }
```

```
    else
        printf("%d\n",num);
    }
    fclose(fp1);
    system("PAUSE");
    }
```

Random access to files

- Functions that help in accessing only a particular part of the file:
  (not reading & writing sequentially).
- ftell
- fseek
- rewind

- ftell :
  - Takes a file pointer.
  - Returns a number of type ***long***, that corresponds to the current position.
  - Useful in saving the current position of a file, which can be used later in the program.

n = ftell (fp);

  - n would give the relative offset (in bytes) of the current position
  - i.e., n bytes have already been read (or written).

- **fseek:**
  - Used to move the file position to a desired location within the file.

fseek (file_pointer, offset, position);

offset -> number of positions to be moved from location specified by position.(number or variable of type long)

position -> integer number.

- position:

0	Beginning of file
1	Current position
2	End of file

- Offset:

positive -> meaning move forwards

negative -> meaning move backwards

Statement	Meaning
fseek(fp,0L,0)	Go to beginning (similar to rewind).
fseek(fp,0L,1)	Stay at the current position (rarely used).
fseek(fp,0L,2)	Go to the end of the file, past the last character of the f
fseek(fp,m,0)	Move to (m+1)th byte in the file.
fseek(fp,m,1)	Go to forward by m bytes.
fseek(fp,-m,1)	Go backward by m bytes from the current position.

- **fseek:**
  - Returns zero, when operation is successful.
  - Returns -1 when attempted to move file pointer beyond the file boundaries (error occurs).

Program that uses functions ftell and fseek:

Code:

```
main()
{
FILE *fp;
long int n;
char c;
fp=fopen("Random","w");
while((c=getchar())!=EOF)
 putc(c,fp);
printf("No. of characters entered = %ld\n",ftell(fp));
fclose(fp);
fopen("Random","r");
n=0L;
while(feof(fp)==0){
fseek(fp,n,0);   //position to (n+1)th character
printf("Position of %c is  %ld\n", getc(fp),ftell(fp));
n++;}
fclose(fp);
system("PAUSE");
}
```

Command line arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, otherwise and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   printf("Program name %s\n", argv[0]);

   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"

Programm name ./a.out
The argument supplied is testing1 testing2
```