C Language

where we can write c pgm's :-
1) Text Editors - Notepad, editplus
2) C-editors - TC, Borland 'C', Quick 'C'
3) IDE's → VSCODE → To use this 1st we install GCC Compiler.

c lag support ASCII code System
unicode

variable

Variable is the name of memory location where data is stored
→ variable is used to store the value
→ Syntax for declaring variable :

datatype var_name = value ;

int i = 100;

Rules for declaring variables/Identifiers
1) It can start with underscore and alphabets
2) It can't start with digit
3) keywords should not declared as Identifiers
4) white spaces are not allowed b/w words (or) characters : int i ;

C language support 3 types of extensions :
1) abc.c = source code (editable)
2) abc.exe = executable file (Non-editable)
3) abc.obj = obj code (byte code) : C, OS

C language is middle level procedure oriented & structured pgming language
→ If any pgming language having modularity features is called procedure oriented
→ Top-down approach with block format is called structured oriented

Compilation process in c language

* It has 4 steps
1) preprocessing 2) compiler 3) Assembling 4) linker
→ preprocessor takes source code & removes all the comments. it take directives like stdio.h, inter predators & saved in a file
→ the preprocessor code compiles & converts into assembly code by using assembler reconvert assembly code into object code
→ By using linker we combine object code of the out pgm & object code of directives.

Datatype

1) primary Datatype : the data will be stored directly in the memory location at compile time
→ the data is stored in stack memory
ex : int, float, char etc

2) Derived Datatype : the address will be stored at run time.
→ the data is stored in Heap memory
ex : Array, pointers.

3) User defined Datatype : struct, unions
4) Enumberation Datatype : enum
5) void datatype : it is used to pass any type of value at run time
6) boolean Datatype : it stores only 2 value i.e, true (or) false

Datatypes size & format specifier

Type Size

Type	size (bytes)	range	Control string
1) char or signed char	1	-128 to 127	%c
2) unsigned char	1	0 to 255	%c
3) int / signed int	2	-32768 to 32767	%d (or) %i
4) unsigned int	2	0 to 65535	%u
5) short int / signed short int	1	-128 to 127	%d (or) %i
6) unsigned short int	1	0 to 255	%d (or) %i
7) long int / signed long int	4	-2147483648 to 2147483647	%ld
8) unsigned long int	4	0 to 4294967295	%lu
9) float	4	3.4E-38 to 3.4E+38	%f (or) %g
10) Double	8	1.7E-308 to 1.7E+308	%lf
11) long double	10	3.4E-4932 to 1.1E+4932	%Lf

character set

→ The characters which are used to form words, digits, symbols & whitespaces.

letters like A-Z (or) a-z

digits like 0-9

special characters = ; semicolon, comma, period, question mark, slash, black slash, vertical bar, exclamation mark, Hide (~), underscore (_), dollar ($), percentage sign (%), caret (^), asterisk (*), opening angle bracket (<), closing angle bracket (>), left parenthesis ("("), right parenthesis (")"), left bracket ("["), right bracket ("]"), Number sign (#), left brace ("{"), right brace ("}")

white spaces ---, blank space, new line, horizontal line, carriage return

Types of variables

1) Global Variable: the variable which is declared outside a block (or) outside a function is known as Global Variable.

→ Any function in the pgm can change the value of the global variable.

→ the scope of the Global Variable is outside of function (or) within the pgm.

→ Redeclaration of the global variable is possible when first declaration does not leads to initialization.

```
int i;
int i=10;
void main()
{
int i;
int i=10;
printf("%d",i);
}
```

error

2) local variable: the variable which is declared within the function (or) block is called local variable.

→ the scope of local variable is within the function (or) block

→ Before we use local variable it should be initialized

3) Auto variable (Automatic variable):

→ It is also called as local variable. Every variable declared inside block by default it is Automatic variable. It is declared by using "Auto" keyword

→ local variable is a scope & Automatic variable is a storage class

```
void main()
{
auto int i=10; local
```

declared
is
value of
side of
possible
initialize
declared
d
in the
d be
ory
it it is
sing
riable

4) static variable :- the variable which is declared with "static" keyword is called static keyword variable
→ In the static variable the previous value will be stored previously within the function clause.

```
void main()
{
    int a=10;
    static int b=20;
    a=a+1;
    b=b+1;
    printf("%d,%d,%d",a,b)
}
```

main()
o/p :- a=11 b=20
main() a=11 b=21
main() a=11 b=22

5) External (external variable :- it is also called as "Global variable".
→ It is declared by using extern keyword
→ External variable can be accessible in multiple files within the application

I/O fuctions :-

C language support formatted & Unformatted I/O functions.

1) formatted I/O fuctions :-
formatted fuctions are present (or) accept data in specific format by using format specifiers
1) printf() → it is formatted o/p fuction ; returntype = int
syntax : returntype printf("formatstring", variable)
2) scanf() → it is formatted i/p fuction
syntax : scanf("formatstring", &variablename)

2) Unformatted I/O fuctions
it is divided into 2 types :-
1) character fuctions :-
2) string fuctions

character functions :-
→ getchar(), getche(), getch(), these are i/p unformatted fuctions.

→ putchar(), putch() Here are o/p unformatted fuctions

String fuctions

puts() → o/p fuction
gets() → i/p fuction

structure of the c-pgm

1) Documentation section : Description (or) summary of the pgm.
→ this section is omitted by using comments
→ c lang. support 2 types of comments :-
1) single line comment (//)
2) Multi line comment (/* ... */)
2) Linkage section : Linking header files into c-pgm. like <stdio.h>, <conio.h>
3) Definition section : In this section we write macros & preprocessor directives
4) Main section : It is starting point of execution of c-pgm.

Keywords

→ the words which have predefined meaning & recognized by the c-compiler.
→ keywords should not be declared as identifiers (or) variables.
→ C lang support 32 keywords : auto, enum, extern, break, case, char, continue, default, double, else, float, for, goto, int, long, register, return, short, sign, sizeof, static, struct, union, switch, typedef, unsign, void, volatile, while, if, do, constant

~~Folders~~

Format specifiers:

→ By using format specifiers we specify what type of data to be printed to the console to compiler.

%d, %i = integer, %c = char, %f = float

```
int i=10;
pf("%d the value of i=%d", i);
```

getbackvalue, value

pf("the value of i=%d", i++); o/p = 10, we don't get error

→ for the variable you can assign expression

→ on the RHS we can have value, expression, variable

→ or the LHS we should have only variable

scanf()

→ this function is used to read the i/p value from the keyboard dynamically at the run time. It will accept multiple arguments

Syntax: scanf("formatted string", &variablelist)

Write a pgm sum of the 2 no's accept the i/p values from keyboard

```
#include<stdio.h>
void main()
{
int a,b,sum;
printf("enter 2 no's");
scanf("%d%d",&a,&b)
sum=a+b;
printf("%d", sum) (or)
JPP("sum of %d & %d = %d", a,b,sum)
```

Integer (signed int)

* int (signed int) (signed short)

format specifier - %d%i

size - 2bytes (32bits) & bytes (64bit)

0 → +ve no's
1 → -ve no's

1 byte = 8 bits
2 byte = 16 bits

Formula :- -2^{n-1} to $2^{n-1}-1$

unsigned : 0, +ve
signed : -ve, +ve, 0

Macros

→ For every datatype we have macros to get max & min value
Ex: INT_MAX, INT_MIN

→ All the datatype macros are available in <limits.h> header file except for float datatype → <float.h>

unsigned integer :
format specifier = %u
size = 2 bytes
macros = UINT_MAX, UINT_MIN

long int
format specifier = %li (or) %ld
size = 4 bytes (32 bitos).
macros = LONG_MAX, LONG_MIN

unsigned long int :
format specifier = %lu
size = 4 bytes (32 bitos)
macros = ULONG_MAX, ULONG_MIN

char datatype
format specifier = %c
size = 1 byte
macros = CHAR_MAX, CHAR_MIN

float datatype
format specifier = %f, or %e or %g
size = 4 bytes, range = $-3.4*10^{38}$ to $3.4*10^{38}$
macros = FLT_MAX, FLT_MIN

double datatype

Format specifier = %lf

size = 8 bytes

range = -1.7 * 10(308) to +1.7 * 10(308)

macros = DBL_MAX, DBL_MIN

long double datatype

format specifier = %lf (or) %Lf

size = 10 byte

range = -3.4 * 10(4932) to 3.4 * 10(4932)

macros = LDBL_MAX, LDBL_MIN

pgm

```
int a=10;
printf("%d", a);
```

o/p = 10.

Escape sequences

By using escape sequences we can customize the o/p

→ It is also called as boundaries (or) delimiters

~~1) '\n' (newline)~~

→ we write escape sequences in printf() function

→ Escape sequences start with backslash (\) followed by the character.

→ It occupies "1" byte of space

Ex: \n = new line (or) next line

\t = tab

\b = backspace

\r = carriage return

\f = form feed

\h = horizontal

\v = vertical

\nn = octal

\hh = hexadecimal

\a = null

\a = alarm beep

```
#include<stdio.h>
#include<limits.h>
void main()
{
printf("%d", INT_MAX);
printf("%c", CHAR_MIN);
}
```

2) printf("welcome to \n c");

o/p welcome to
in c

It is printed as it is bcoz of delimiter predefined meaning

\t (tab)

→ tab will move from one tab to another tab not by character.

→ for one tab should have the 8 characters

Ex:-

pf("1234\t5") → o/p: 1234 ... 5

pf("1234\t\t\t6") → o/p: 1234 ... 5 6

\b (backspace)

→ It moves the cursor one position back

pf("123\b5678") → o/p: 1235678

pf("12345678\b") → o/p: 12345678

\r (carriage return)

→ It makes cursor to move first position in the same line.

pf("Anu\r") → o/p: Anu

pf("Anu\rc") → cnu

pf("Anu\rtai") → tai

pf("\rabc") / pf("\bsi") / pf("\rta") → o/p: tai

→ pf("welcome to \"c\""); → o/p: welcome to "c"

→ we can't use double quotes in double quotes

→ pf("welcome to "c") → welcom

operators

→ Operator is a symbol which is used to perform operations on the operands.

Types of operators:-

1) Arithmetic operators.

→ These operators are used to perform mathematical operations like +, -, *, /, %. +, - -

(Q) (R)

2) Relational operators

→ these operators are used to compare the values

→ Return type = boolean value i.e. True (or) False

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

3) logical operators
→ These operators are used to combine the multiple conditions.
→ Return type is boolean value
Ex: &&, ||, Not

4) Assignment operators
→ there are 2 types of assignment operators :-
1) Simple Assignment operator (=)
2) Compound Assignment operator → are a mix assignment operator with other operator like +=, -=, %=, *=, etc.
Ex: a+=10 ⇒ a=a+10.

5) Conditional operator (question mark operator, Ternary operator)

```
a>b?a:b;
```

6) Special operators
1) sizeof

7) Bitwise operators
→ these operators are used to perform operations on the bits
Types of Bitwise operators
1) Bitwise AND
2) Bitwise OR
3) Bitwise X-OR
4) Complement
5) leftshift
6) rightshift

Increment & Decrement operators
* ~~Incre~~ these operators are called unary operators
→ Increment operator (++) → it has two forms
1) preincrement
2) it is incremented by value 1 & later we perform operations.

z = ++a;
a = a+1; // a = 10+1
z = a; // z = 11 = 1

2) post increment
→ first we perform operations later we incremented by '1'

a = 10
z = a++
z = a, z = 10
a = a+1; a = 10+1 = 11

#include<stdio.h>
void main()
{
int i = 1;
printf("%d %d %d", i, ++i, i++)
}
o/p : 3, 3, 1
3 ← 3 ← 1

→ int i = 0, j = 0;
j = ++i + i++;
printf("%d", j); o/p : 2, 2
pf("%d", i);

logical operators
→ these operators are used to combine the multiple conditions
1) AND (&&) → if any of the condition is false result is false
2) OR (||) → If any of the condition is true result is true
3) NOT (!) it is the complement

Bitwise operators
→ these operators will perform operations on bits
Ex : AND

12 & 25

2 | 12
2 | 6 — 0
2 | 3 — 0
1 — 1
(00001100)₂ ← 12

2 | 25
2 | 12 — 1
2 | 6 — 0
2 | 3 — 0
1 — 1
(00011001)₂ ← 25

00001100
00011001
00001000

1000 = 2^3 + 2^0 + 2^1 + 2^0 x 0

12 || 25
↑12
(00001100)$_2$
00001100
00011001
00001101

(00011001)$_2$
$1\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$
$=29_{10}$

X-OR (^).

00001100
00011001
00001101

$1\times2^4+0+1\times2^2+0+1\times2^0$
$=16+4+1=21$

→ If both are different then result is true otherwise result is false

left shift operator (<<)

→ It moves one position towards left side.

eg: 5<<1 → a=12, b=25
pf("%d", a^b)

00000101
11111011
00001010 = 10.

Right shift operator (>>)

eg: 10>>1
00001010
00000101

1's Complement

pf("%d", ~5); →o/p :-6

complement
↑5
-(x+1)
-(5+1) = -6.

eg :- 33<<2

∴ formula for leftshift : N (2^{pos})
= 33 (2^2)
= 132

eg: 33>>2

∴ formula for rightshift = $\frac{N}{2^{pos}} = \frac{33}{2^2} = \frac{33}{4}$

Special operators

1) Comma operator :
2) type casting operator
3) Reference operator (&)
4) Dereference operator (*)
5) double pointer (pointer to pointer)
6) sizeof()

Comma operator

→ these operator execute first operand the result is discarded & second operand result is evaluated.

Eg: int i=(10,20); [∴ the last value will be considered & displayed]
printf("%d",i); o/p :- 20
Nodes file

Eg: int f1()
{
return 10;
}
int f2()
{
return 33;
}
void main()
{
int k=(f1(),f2()); o/p :- 33
printf("%d",k);
}

Type of operator

→ it is used to convert one datatype to another datatype

1) Implicit : Automatically converting one datatype to another datatype

2) Explicit : Manually converting one datatype to another datatype. syntax : (datatype) var_name;

Pgm : int i=17,j=5;
float f;
f=(float) i/j;
pf("%f",f);

Eg

Reference operator (or) Address of operator (&)

int i=10;
pf("%d",&i); o/p : 64223800 (Address)

sizeof() operator

→ By using this operator we can get size of given variable.

pf("%d",sizeof(int)); → 4
pf("%d",sizeof(char)); → 1

Deference operator (*)

→ It returns the value stored in a variable pointed by specified pointer

```
int k =10;
int *p;
pf("%d",k); → 10
pf("%u",&k); → 6842200
p=&k;
pf("Address = %d",p); → 6842200
pf("Value = %d",*p); → 10.
```

Double pointer (pointer to pointer)

→ pointers are used to store the address of a given variable of similar datatype.
→ If we want to store address of pointer variable then we use double pointer (**p) i.e., pointer to pointer variable

Conditional operator (? operator) (ternary operator)

syntax: condition ? T : F

→ It is alternate to if else condition.

write a pgm to find greatest among 2 no's by using ternary operator

```
int a,b,d;
pf("enter 1st no");
sf("%d",&a);
pf("enter 2nd no");
sf("%d",&b);
d=a>b?a:b;
pf("%d",d); → 20
```

```
int a,b,k;
pf("enter 3 no's");
sf("%d%d%d",&a,&b,&c);
big = i>j && i>k ? i :
      j>k ? j : k
```

finding biggest among 4 no's

(a>b)(a>c)(a

(a>b&&a>c&&a>d)?a:(b>c&&b>d)?b:(c>d)?c:d;

Constant

→ C lang. support 2 ways to declare a constant
1) By using constant key
2) By using #define preprocessor directive
→ the value which can't be change throughout the pgm is called constant

By using const key

Ex: const float pi = 3.14;
pi = 11.23; → error
pf("%f", pi); → 3.14

→ By using #define

Syntax: #define token value

Ex:
```
#include<stdio.h>
#define pi 3.14
void main()
{
pf("%f", pi);
}
```

Control flow statements:

→ By using CFS's we control the execution of the pgm
→ By using these stmts we can specify the order in which the stmts should be executed
→ we are having 2 types of condition stmts:
1) if 2) switch

Types of if :- 1) simple if, 2) if else 3) Nested if
4) if-else if-else

2) Iterative stmts (or) loops : 1) for loop → entry control loop
1) simple for loop 2) labelled for loop
3) Enhanced for loop 4) Infinite for loop.

2) while loop :
→ it is 2 types
1) simple while loop 2) Infinite while loop

3) Do-while loop

-> It is called as exit control loop

It is a type : 1) simple do-while loop 2) Infinite do-while loop

Jump (or) transfer stmts :-

1) Break 2) Continue 3) return 4) go to

5) exit

Conditional stmts :

-> for diff conditions we perform diff. actions

simple if :

syntax : if <conditions>
{
stmt;
}

if else

syntax : if <conditions>
{
stmt;
}
else
{
stmt;
}

if else if else

syntax :- if <conditions>
{
stmt;
}
else if <conditions>
{
stmt;
}
else
{
stmt;
}

write a pgm to find greatest among 3 no's. accept i/p values from keyboard

```
#include<stdio.h>
void main()
{
int a,b,c;
pf("enter 3 no's");
sf("%d%d%d",&a,&b,&c);
```

```
if((a>b)&&(a>c))
{
  printf("largest no is=%d", a)
}
else if (b>c)
{
  printf("largest no is=%d", b);
}
else
{
  printf("largest no is=%d", c);
}
}
```

switch

→ From the multiple options if we want to return only one value then go with switch case

syntax: switch(variable/expression)
{
case caselabel(: → it should be int & char not float, string
  stmt;
  break;
case 2:
  stmt
  break;
default:
  stmt;
  break;
}

* we can't write variable name as a caselabel
* char. expression, constants, symbols, are allowed to write in caselabel.

case 'c', case 'a', case 'abc', case "abc",
✓ ✓ ✗ ✗

Rules for declaring switch case:

→ Expression in the switch stmt should be always int const & char const.
→ No real no's are used in the expression
→ Default can be placed anywhere
→ Default is optional
→ caselabel should be terminated with the colon
→ caselabel can't be duplicate
→ caselabel can be return in expression
Ex: switch(x>1), switch(x+10), switch(x),
switch(x+1.1), switch(f(c)), case 1, 2, 3:
✗

Iterative starts

→ If the process is repeating for multiple times then we go with Iterative starts

→ C lang support 3 types of iterative starts.

1) for loop 2) while loop 3) Do-while loop

for loop: It is used when we know the fixed no. of iterations, in advance ① ② ⑤ ④ ⑦

syntax: for (initialization; condition; inc/dec)
{
stmts; ⑨ ⑥
}

ex: for (i=1; i<=10; i++)
{
pf("%d", i); o/p: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
}

→ In for loop everything is optional

→ for loop rotates in anticlockwise direction

→ In for loop we can do multiple initializations, conditions, inc/dec. we

→ when you are doing multiple initializations all the do variable datatypes should be similar

pgr for switch case

```
int a = 3.8;
switch(a)
{
default:
print("invalid");
break;
case 1:
pf("one");
break;
case 2:
pf("two");
break;
case 2+1:
pf("three");
} break;
```

ex: int a = 4/2;
switch(a)
{
=
=
=
=
case 2*2:
pf("one");
break
case 8/2: pf("two"); break;
case 12%4: pf("three"); break;
}

Pgm

```
int x=0,i,k
for(k=0;i=5;k<5;i>0;k=k+2,i--)
{
x++;
pf("%d%d%d%d",k,i,x);
}
```

o/p : 051
242 → 88
433

051
242
433 → ,(comma)
624
815

Infinite loop

```
for(;;)
{
pf("Pen");
}
```

```
int i;
for(i=1;i<=10;i++)
{
int i=20;
pf("%d",i);
}
```

o/p :- 20,20,20,20,20,20,20,20,20,20

Nested for loop

→ the for loop inside the another for loop is called Nested for loop.

```
for(int i=1;i<=10;i++)
{
for(int j=1;j<=10;j++)
{
pf("%d%d",i,j);
}
```

o/p
1 1
1 2
2 1
2 2
upto
10 9
10 10

while loop :-

→ It is used when we don't know the fixed no. of iterations

→ C lang support 2 types of while loops :-

1) simple while loop 2) Infinite while loop

syntax :-

```
initialization
while (condition)
{
stmts;
inc/dec;
}
```

Ex: int i=1;

```
while (i<=10)
{
pf("%d\n",i);
i=i+1;
}
```

```
int a=5;
while(a) (or) false
{
pf("%d\n",a);
a=a-1;
}
pf("a=%d",a+10);
}
```

```
int a,b;
a=b=1;
while(a)
{
a=b<=3;
b=b+1;
pf("%d%d\n",a,b);
}
pf("%d%d%d\n",a+10,b+10);
}
```

o/p:
```
5
4
3
2
1
10
```

o/p:
```
2
3
4
5
10 15
```

Infinite while loop

→ the while loop which don't have any condition

```
ex: int i=1;
    while(1)
    {
    pf("%d",i);
    i=i+1;
    }
```

do-while loop

→ this also called as exit control loop

→ if the condition is false also atleast one iteration will be executed

```
ex: int i=1;
    do
    {
    pf("%d",i);
    i=i+1;
    while(i<10);
    }
```

o/p: 1

Jump control / loop control statements

1) **Break** : It is an conditional control stmt

→ Break is used in switch & inside the loop

→ Break can't be used if only if condition is executed

Ex:-

```
for(int i=1; i<=10; i++)
{
pf("%d\n", i);
if(i==5)
break;
}
```

o/p :
1
2
3
4
5

Ex:- int i=1;

```
while(i<=10)
{
if(i>4)
break;
pf("%d", i);
i++;
}
pf("%d", i+10);
```

o/p :
1
2
3
4
15

Continue

→ the given condition is skipped & next iteration will continue

```
for(int i=1; i<=10; i++)
{
pf("%d", i); → o/p : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
if(i==5)
continue;
pf("%d", i) → o/p : 1, 2, 3, 4, 6, 7, 8, 9, 10
}
```

→ int i=10;

```
while(i<=10)
{
i=i+2;
if(i>40 && i<60)
continue;
pf("%d", i);
}
```

o/p : 12

Goto

→ Goto is a keyword which is used to jump one part of the pgm to another part

→ Goto is followed by the identifier i.e, label

Syntax: start 1;
start 2;
goto label;
start 3;
label:
start 4;
start 5;

```
→ printf("A");
  printf("B");
  goto xyz;
    printf("C"); o/p: ABDE
xyz:
  printf("D");
  pf("E");
```

→ Calling the label is always optional

write a pgm display the even no's from 2 to 20 by using goto start.

```
int i=2;          o/p: 2, 4, 6, 8, 10, 12, 14 ... 20
EVEN:
  pf("%d", i);
  i=i+2;
  if(i<=20)
goto EVEN;
```

without calling label name

```
Ex: ABC:
  pf("A")
  pf("B")
```

```
→ pf("A");
  goto xyz;
    pf("B");
ABC:
    pf("D");
xyz:
    pf("C");
goto ABC;
```

→ pf & sf will return a value
→ pf will return by default integer value
→ sf will return integer values based on no. of values you are submitting.

Return strmt

A function may return or may not return a value.

→ In a pg function will return only one value.

→ Return keyword should be last strmt inside the function

→ After return keyword if we write any strmts those strmts will not be considered

Ex:
```
int f()
{
return 10;
}
void main()
{
int i=f();
pf("%d", i);
pf("%d", f());
}
```

exit

→ exit function is available under <stdlib.h>

→ By using this function we are exiting from the pgm

Ex:
```
#include<stdlib.h>
void main()
{
exit(0);
}
```

Arrays:

→ Array is a collection of similar datatype elements

→ Arrays are derived datatypes

→ Arrays are "zero" based index

→ Arrays are dynamic in nature

→ Arrays are fixed in size

Types of Arrays

1) Single / 1D array
2) 2D array
3) Multi Dimensional array

1) 1D array

→ It this elements will be arranged either in row/col format

→ Syntax for declaring.

datatype arrayname[size]; → size is mandatory

Ex: int arr[5];

statement → int arr[] = {1,2,3}; / char arr2 = ['a','b'];
int arr[4] = {1,2,3,4};/

int arr[]; → error

int arr[-2]; → error

→ const int a=5; ✓ | int a=5;
int arr[a]; | int arr[a]; ✗

write a pgm to insert the elements into 1D array & display

int arr[4];

Pf("%d", arr] → getting the address of array

Single Dimensional

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int i=011;   → octal (1x8^1 + 1x8^0)   olp: 9   8+1=9
Pf("%d",i);
}
```

→ int arr[3];
Pf("enter elements");
for(int i=0;i<3;i++)
{
sf("%d", &arr[i]);
}
Pf("displaying elements");
for(int i=0;i<3;i++)
{
Pf("%d\n", arr[i]);
}

olp: enter elements: 1,2,3
display elements 1,2,3

```
int arr[10];
int size;
Pf("enter size");
sf("%d", &size);
for(int i=0;i<size;i++)
{
Pf("enter element a[%d]=", i);
sf("%d", &arr[i]);
}
Pf("display elements");
for(int i=0;i<3;i++)
{
Pf("%d", arr[i]);
}
```

```
int i=3, x;
int arr[] = {10,20,30,40,50}
x = i*arr[i] + 2*arr[i-1] + 3*arr[i-2];
arr[i-1];
```

Pf("%d", x) [∵ arr[-1] = 0]

olp: -12 [∵ arr[-3] = -2]

1*(-2) + 2*(-2) + 3*(-2)
-2 - 4 - 6 → -12

→ int arr[5]={5,1,15,20,25};
int i,j,k=1,m;
i=++arr[1];
j=arr[1]++;
m=arr[i++];
pf("%d%d%d%d",i,j,m,i[arr]); → or arr[i]
}

i=3
j=1
i=2
m=15
i[arr]=20

2 Dimensional array

→ In 2D array elements will be arranged in the form of rows & Columns..
→ It is also called as "array of arrays"
→ int arr[3][3]; ✓ arr[3][]; ✗ arr[][3]; ✓
arr[][]; ✗
→ row size is optional & Col size is mandatory.

Eg: {1,2,3} → 1D, {{ }} → 2D

Ex: int arr[][3]={1,2,3,4}

{1,2,3}{4,0,0}

123
400

Ex:

int arr[3][3]={{1,2,3},{4,5,6},{7,8,9}};
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
pf("%d",arr[i][j]);
}
pf("\n");
}

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

int a[3];
pf("%d",a[0]);

int a[5]={1,2,3};
pf("%d",a[3]);

0	1	2	3	4
1	2	3	0	0

o/p: 0

→ int a[2][3] = {1, 2, 3, 4};
pf("%d", &arr); → add of array
pf("%d", &arr); → add of array
pf("%d", *arr); → 1D add of array
pf("%d", **arr); → 1st element of array

→ char ch[] = "pencil";
char ch[] = {'p', 'c', 'n'}

→ char ch[] = "python"; → o/p: python
char ch[] = {'p', 'y', 't', '\0', 'h', 'o', 'n'}; → o/p: pyt
int i = 0;
while(ch[i] != '\0')
{
pf("%d", ch[i]);
i = i + 1;
}

→ passing array as a parameter to a function:-
→ to the function we can pass array, pointer, structure as a parameter

→ Ex: void f1(int arr[]);
void main
{
int arr[] = {1, 2, 3, 4, 5};
f1(arr);
}
void f1(int arr[]) o/p: 1 2 3 4 5
{
for(int i=0; i<5; i++)
{
pf("%d", arr[i]);
}

functions

→ function is a group of stmts under a name is called as a function.
→ the main advantage of function is code reuseability
→ clang support 2 types of functions:
1) pre-defined / built in functions :- the functions which comes automatically with C.

2) User Defined functions:-
→ the functions which are created by the user
→ main() is a user-defined function.

The precedence & associativity of various operators in C

operator	Description	precedence	Associativity
()[]	functional Call square bracket	1	L-R (left to right)
+, -, ++, --, !, ~, *, &, size of	Unary plus, Unary minus, Inc/Dec, Not operator Complement pointer operator Address operator size of operator	2	R-L (Right to left)
*, /, %	Multiplication, Division, modulo div	3	L-R
+, -	Add, sub	4	L-R
<< >>	left shift, right shift	5	L-R
< <= > >=	Relational operator	6	L-R
== !=	Equals to, Not equal to	7	L-R
&	Bitwise AND	8	L-R
^	Bitwise X-OR	9	L-R
|	Bitwise OR	10	L-R
&&	logical AND	11	L-R
||	logical OR	12	L-R
? :	Conditional	13	L-R
=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Assignment operator	14	L-R

steps for creating a function (components in the function)

1) function declaration (function prototype)
2) calling the function
3) creating the function (called function)

```
#include<stdio.h>
void f1();
void main()
{
f1(); → o/p: welcome to f1()
}
void f1()
{
pf("welcome to f1()");
}
```

properties / diff types of args

→ parameters are the i/p to the function
→ parameter provides data communication b/w calling function & called function
→ there are 2 types of parameters:
1) Actual parameters 2) formal parameters

Actual parameters: the parameters which specified in calling function are called Actual parameters
→ Actual parameters can be variables, expression, constant values & address.

formal parameter
→ the parameter specified in called function are called formal parameters
→ formal parameters can be only variables.

ex:
```
void main()
{
sum(a,b);
}
void sum(int i, int j)
{
}
```

a,b are actual parameters
i,j are formal parameters

Types of functions

1) No arg, no return type
2) arg with no return type
3) no arg with return type
4) arg with return type

Arg with no return type

```
void sum(int, int); (sum(int, float);
void main()
{
  sum(10,20); or sum(11,33.23);
}
void sum(int i, int j)
{
int sum = i+j;
pf("%d", sum);
}
```

```
o/p: 30.6float sum = i+j;
     pf("%f", sum);
```

Arg with return type

```
int f1(int, int);
void main()
{
  int add = f1(10,20);
  pf("%d", add);
}
int f1(int i, int j)
{
  int sum = i+j;
  return sum;
}
```

o/p :- 30

no arg with return type

```
int f1();
void main()
{
  int add = sum();
  pf("%d", add);
}
int f1()
{
  int i=10, int j=20;
  int sum = i+j;
  return sum;
}
```

Recursive functions

→ the function which call by itself is called Recursive function

→ Advantage is to shorten the code

- write a pgm to find factorial of a given no by using recursive function.

```
#include<stdio.h>
unsigned int factorial(unsigned int i)
{
if(i<=1)
{
return 2;
}
return i*factorial(i-1);
}
void main()
{
int i=5;
i=factorial(i);
pf("%d",i);
}
```

```
#define x 5+2
void main()
{
int i;
i=x*x*x;
pf("%d",i);
}
```

→ 5+2*5+2*5+2
= 5+10+10+2
= 27 o/p

```
void main()
{
void f1(); → function
{ declaration in
f1(); main()
}
void f1()
{
pf("Hi");
}
```

```
→ #include<stdio.h>
void pointer()
{
pf("Hello")
}
void main()
{
printf();→ error
}
```

→ syntax

→ predefined functions can be used as function name but we can't call the function

Storage classes

→ storage classes specifies storage area, scope (visibility of the variable), lifetime, default value.
→ there are 2 types of storage classes
1) Automatic storage class 2) static storage class
1) Automatic storage class
→ Automatic storage class variables are created automatically & destroyed automatically
→ it is stored in stack area of data segment
→ under this we have 2 types
1) auto variable (automatic variable)
→ it is also called as local variable
2) register variable
→ it is stored in CPU register
→ it requires more memory
→ if the variables are repeating continuosly use register variables.
→ these variables are faster than other variables.
→ they don't support pointers.

static storage classes :
→ it will be created only once
→ there are 2 types of variables
1) static 2) extern
→ these variables are stored inside static area

Type	scope	life	default value
auto	within the body (fun() (block	body	garbage value
static	within the fun()	program	zero (0)
extern	program	application	zero (0)
register	body	body	garbage value

working with auto variables

```
void ~~naire~~ abc()
{
auto int a=5;
  --a;
  pf("%d", a);
}
void naire()
{
  abc();
  abc();
  abc();
}
```

→ void naire()
```
{
auto int a=5;
pf("%d", a);
{
 int a=10;
 --a;
 pf("%d", a);
}
++a
pf("%d", a);
}
```

working with static variables

```
void abc()
{
int a=5;
static int s=5;
++a;
++s;
pf("%d%d%d", a, s);
}
void main()
{
abc();
abc();
abc();
}
```

→ the scope of static variable within the function

```
void abc()
{
auto int a=10;
  --a;
  pf("%d", a);
}
void naire()
{
int a=5;
  ++a;
abc(); → 9
abc(); → 9
abc(); → 9
++a;
pf("%d", a); → 7
}
```

preprocessor

→ preprocessing is a pgm which will be executed automatically.

→ preprocessing is under the control of preprocessing directives

→ In c lng. we have diff. types of preprocessing directives

1) Macro substitution directive

Ex: #define

2) File inclusion directive

Ex: #include

3) Conditional compilation directive

Ex: #if, #elif, #else, #ifdef, #undef, #endif, #ifndef

4) Mislenious directives

Ex: #error, #pragma, #line

macros

→ we can write the directive anywhere in the c pgm but define the directive before we create the first function.

syntax:

#directive identifier replacement text

Ex:

```
#include<stdio.h> (or) "stdio.h"
#define x 10 → Constant
void main()
{
++x; → error
pf("%d", x); → 10
}
```

```
#define x (3+2)
#define y (3+2)
void main()
{
int c = x * y;
pf("%d", c); → 25
}
```

Function macros

```
#define max(x,y) (x>y)?x:y
#define sum(x,y) x+y
void main()
{
int s;
s=sum(11,22);
pf("%d", s); → 33
}
```

```
(or)
int max(x,y)
{
if(x>y)
return x;
else
return y;
}
```

```
#include<stdio.h>
#define f1() {\
for(int i=1; i<=10; i++)\
pf("%d", i);\
}
void main()
{
f1();
}
```

write a pgm to find cube of a given no. by using macros.

```
#define cube(x) x*x*x
void main()
{
 int s;
 s=cube(3);
 pf("\n%d", s);
}
```

File inclusion directive

→ By using this preprocessor directive we can include a file into another file

→ By using this preprocessor directive we can include ".h (Header file)

→ Header file is a collection of predefined functions, global variables, constant values, predefined datatypes, predefined structures, predefined macros.

Ex:
```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
#include<stdarg.h>
#include<process.h>
```

```
#include<limit.h>
#include<dos.h>
#include<ctype.h>
#include<dir.h>
#include<graphic.h>
```

condition compilation directive

1) #if

```
void main()
{
 pf("A");
#if 5<8!=1
 pf("C");
 pf("D");
#endif
 pf("B");
}
```

o/p :- A
B

→ #if define & #ifndefine these two are called as macro testing conditional compilation preprocessor directives.

```
#define NIT
void main()
{
pf("hii");
#ifdef NIT
pf("hello");
pf("NIT");
#endif
```

o/p: hii
hello
NIT.

Mislanious directives

1) #error → this directive is used for indicating the errors
→ the compiler gives fatal error
→ if #error is found it skips further compilation process
2) #pragma → this directive gives additional info. to the compiler
3) → #pragma startup & #pragma exit → these two directives specifies when function should be started & ended.

syntax: # pragma token

```
#include<stdio.h>
void f1();
void f2();
#pragma startup f1()
#pragma exit f2()
void f1()
{
pf("f1()");
}
void f2()
{
pf("f2()");
}
void main()
{
pf("main");
}
```

o/p: f1()
main
f2()

→ #include<stdio.h>
#define INPUT
void main()
{
int a=0;
#ifdef INPUT / #ifndef INPUT
a=2;
#else
pf("enter value for a");
sf("%d",&a);
#endif
pf("value=%d",a);→2 / → enter value of a=10
}
10

structures

→ structure is a combination of diff. datatype elements
→ structure is a user-defined datatype
→ structure is a combination of primary & derived datatype
→ By using "struct" keyword we create a structure & it should be terminated by semicolon
→ ~~the smallest size of~~
Syntax for creating structure :- structure structname
{
Datatype member;
- - - -
};

Declaring structure variable

→ with the help of structure variable we can assign values to the members of the structure & we can access members of the structure
→ there are 2 ways to create struct variable :-
1) At the time of creating the structure
eg) struct emp
{
int id;
float sal;
}e1;
2) creating the struct variable inside the main
void main()
{
struct emp e1;
}

```
struct emp
{
 int id;
float sal;
}e1;
void main()
{
struct emp e1={50,5000};
   e1.id=50;
   e1.sal=5000;
```

Accessing the members of structure

→ there are 2 ways to access the members of structure

1) By using dot (.) → if we are using normal datatypes. ~~##~~ use dot operator.

→ if we are using pointers use "→" operator

```
struct emp
{
 int height;
 int weight;
 int age;
float sal;
}e1;
void main()
{
 struct emp e1={6,75,23,5000};
 pf("%d",e1.height);
 pf("%d",e1.weight);
 pf("%d",e1.age);
 pf("%f",e1.sal);
}
```

write a pgm create a structure with structure members

id, name, sal

```
#include<string.h>
struct emp
{
 int id;
string char name; char name[10];
 float sal;
}e1,e2,e3;
void main()
{
 e1.id=100;
 strcpy(e1.name,"isa");
 e1.sal=6000;
 pf("%s",e1.name);
 pf("%d",e1.id);
```

```
pf("%f",e1.sal);
```

or

```
struct emp e1={23,"isa",2000};
```

structure
datatypes.

rembers

;2000];

Nested structure

→ the structure inside the another structure is called Nested structure.

→ there are 2 ways to created Nested structure.

1) By creating the structures separately
2) By embeding one structure inside another structure

Ex:
```
#include<stdio.h>
    struct x
    {
    int a;
    };
    struct y
    {
     int b;
     struct x x1;
    };
    struct z
    {
    int c;
    struct y y1;
};
void main()
{
 struct z z1;
 z1.c=33;
 z1.y1.b=22;
 z1.y1.x1.a=32;
 pf("%d",z1.c);
pf("%d",z1.y1.b);
pf("%d",z1.y1.x1.a);
}
```

structure to the pointers ~~like primitive datatypes~~

→ like primitive datatypes
→ we can have pointer to the structure by using arrow (→) operator. we can access rembers of the structure
→ we can create the structure variable globally

```
struct emp
{
 int x,y;
};
void main()
{
struct emp e1={x=100,y=200};
pf("%d%d",e1.x,e1.y);→x=100, y=200
struct emp e2={y=300};
pf("%d%d",e2.y,e2.x);→x=0 y=300
```

```
struct emp
{
 int x,y;
};
void main()
{
struct emp e1={11,22};
struct emp *ptr=&e1;
pf("%d%d",ptr→x,ptr→y);
```

op:- 11,22

→we can't perform arithmetic operators on structure variables.

```
 struct emp
 {
 int x;
 };
void main()
{
struct emp e1,e2,e3;
e1.x=11;
e2.x=22;
e3.x=e1.x+e2.x;
pf("%d",e3.x);
}
```

union:

→ union is a collection of diff. datatype (elements)
→ union is a derived datatype
→ union data members will share single memory location which is equal to the size of largest data member

→ By using union keyword we create union.

```
union emp
{
int id;
char ch[20];
float sal;
}e1;
void main()
{
union emp e1={20,"isa",5000};
pf("%d %s %f",e1.id,e1.ch,e1.sal);
pf("%d",e1.id)
```

we can't access all members at a time but we can access one at a time

because data members will store single mem. location

Files

→ Files are used to store data permanently in the secondary storage device.

There are 2 types of files

1) text file 2) Binary file

1) Text files: these files are used to store character data.

→ Default extension is ".txt".

→ Binary files are used to store binary data (0 & 1) like images, audio files & video files.

→ extension for the binary file is ".bin"

→ File I/O

formatted i/p function :-

→ steps for processing the file :

1) Create the file pointer : FILE *ptr;

→ By using the file pointer data will be communicated b/w file & the program

2) opening the file.

fopen("filename","mode"); ex: fopen("abc.txt","r");

→ Before we perform any operations on the file, file should be opened by using "fopen()", it takes two arguments 1st is filename & 2nd is mode.

3) process the data

→ By performing read & write operation

4) closing the file:
→ closing of file can be done by using "fclose()"
ex: fclose(fp);

formatted file I/O function:
→ fprintf() is the formatted file o/p function
→ fscanf() is the formatted file i/p fection

unformatted file I/O fection
→ unformatted file o/p fections
1) getc() 2) getw() 3) fread()
→ unformatted file o/p fections
1) putc() 2) putw() 3) fwrite()

~~steps~~ streams
→ stream is a flow of data (or) sequence of bytes after "c" pgm is started.
→ os is responsible for opening three streams
1) stdin 2) stderr 3) stdout
↓ related to keyboard ↓ related to monitor

modes
1) w → opening the file in the write mode
→ If the file doesn't exist it will create a file
→ If the file already exist old data will replace with new data
2) r → opening the file in read mode
→ If

3) a →

4) r+ → oper the file in read & write mode
5) w+ → oper the file in write & read mode
6) a+ → oper the file in append & read mode

functions

1) getc() → get the character from a file. ex: getc(fp);
2) putc() → writing the character to a file. ex: putc(c, fp);
3) fprintf() → writing set of data to the file.
   ex: fprintf(fp, "control string", list);
4) fscanf() → reading set of data from the file.
5) getw() → getting int value from file
   int value
6) putw() → writing int value to a file
7) EOF → end of the file

→ forward / → backward

```
void main()
{
FILE *fp;                          // alternate solution ( or ) /a
fp=fopen("d:\abc.txt", "w");
fprintf(fp, "hello file");
fclose(fp);
}
```

1) putc()

ex:- putc('a', fp);

<ctype.h>

→ this header file contain functions, which accepts character as a i/p of one byte & returns integer value as o/p either '0' or 'a'

→ int isalnum(char)
→ int islower(char)
→ int isupper(char)
→ int isdigit(char)
→ int isalpha(char)

```
#include<ctype.h>
#include<stdio.h>, void main(){
char a;
pf("enter the character");
sf("%c", &a);
if(islower(a))
{
pf("it is lower");
if(isalpha(a))
pf("it is alpha");
```

```
#include<ctype.h>
main()
{
char ch[] = "iso";
pf("%s", ch); puts(ch);
int i=0;
while(ch[i] != '\0')
{
pf("%c", ch[i]);
}
```

strings

→ string is a collection of characters / it is a array of characters / sequence of characters
→ c lang supports string functions in <string.h> header file

functions

1) strcpy (target, source)
2) strcat (target, source)
→ it combines source to the target
3) strrev (source)
4) strupr (source)
5) strlwr (source)
6) ~~strlen~~ int strlen (source)
→ it returns no. of characters given in string
7) ~~strcmp~~ int strcmp (str1, str2)
→ if both strings are equal then "0"
→ if 1st string is greater then 2nd string then "1"
→ if 1st string is less then 2nd string then "-1"

rules

→ no. of characters
→ order of the character (it will compare 1st character if it is same then next characters)
→ Based on ASCII code value

8) stricmp (s1, s2)
ignore case sensitive

→ char s1[10] = "pascal";
→ char s2[10] = "windows";
strcpy(s2+4, s1+2);
puts(s1); → pascal
puts(s2); → windscal

→ char s1[20] = "hello891011121314"
strrev(s1+5); o/p: hello98765111098
puts(s1);