1. Introduction
2. Data types in C
3. Operator
4. Control statements
5. Functions
6. Recursion
7. Pointer
8. Array
9. String
10. Storage class
11. Files

Introduction

<Q> What is embedded system?

- It is a computer(up/uc) based subsystem in a larger system

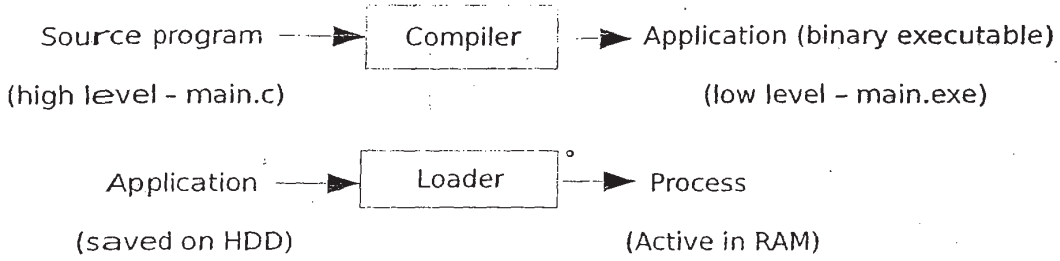
- combination of H/w and s/w which design specific task.

Program : Set of instructions with respect to a particular programming language

Programming language : Set of rules to write a program

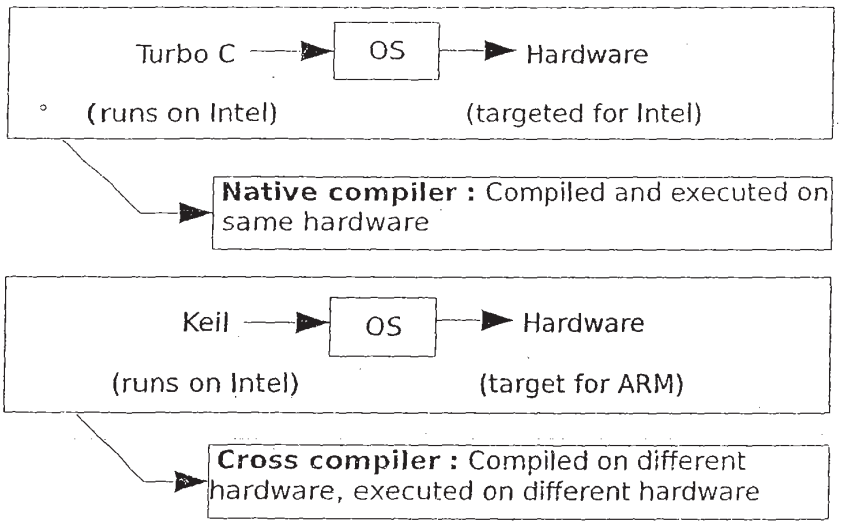
Program, Application and Process

Program :- Source file (main.c)
Application :- Executable file saved on Hard drive (main.exe)
Process :- Application which is active in RAM



<Q> **Benefits** of programming in high level language.

- Easy debugging
- Easy coding
- Re usability



<Q> Why C is preferred over C++ and Java in cross compilers?

- Executables (binaries) compiled in C compilers are compact (smaller) in size than executables (binaries) compiled in C++ and Java compilers.

Introduction to Linux

UNIX – command user interface
LINUX – sophisticated graphical user interface

Portable commands for UNIX/LINUX :

`pwd`	prints current working directory
`clear (ctrl + l)`	clear screen
`poweroff`	shut down
`exit`	log off from current user
`mkdir [dir]`	makes new directory
`rmdir [dir]`	removes empty directory only
`cd ..`	go to parent directory
`cd [dir]`	enters into typed directory i.e. makes typed directory as working directory
`cd`	makes working directory as home directory of user (jumps to home directory)
`ls`	list all files and directories present in that directory
`ls -a`	list all files and directories including hidden files
`ls -l`	long list
`ls /`	root directory files listed
`ls /home`	home directory files listed
`ls [dir]`	files within that directory listed
`cp [source file] [destination file]`	copies file in same folder
`cp [source file] [destination directory]`	copies mentioned source file in mentioned directory
`rm [file]`	deletes file, not used for directory
`rm prog*`	deletes all files starting with prog
`rm *prog`	deletes all files ending with prog
`rm -r [dir]`	deletes directory recursively (including files and subdirectories within that directory also)

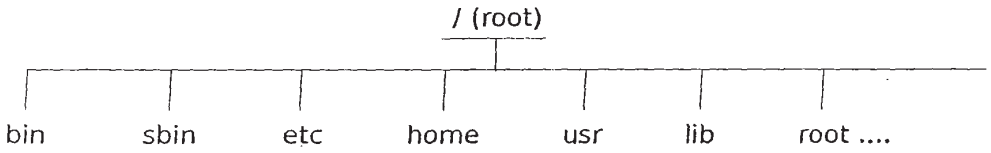
NOTE : when in empty folder, entered command `ls -a` returns with : . ..
. - links to itself
.. - links to parent directory
. & .. are called links, every operating system folder contains it.

NOTE : Hidden files in linux starts with . (dot)

Usage of `cp`: if in Linux file system, following folder order is present, home>v14he2>day1 then if your working directory is day1, then to copy a file named test.c located in directory day1 in directory v14he2,

```
cp test.c ../test.c       //if you want to rename destination file
          or
cp test.c /home/v14he2    //copies file with same name if
                            destination file name is not specified
          or
cp test.c ../             //copies file with same name
```

UNIX file system:



bin sbin /usr/bin /usr/sbin	|	contains executables (commands)
dev	>	device driver files
etc	>	system maintenance files and admin related files
home	>	contains home folders for every user
lib	>	contains library files (.dll and .so)
root	>	home for admin (root)

LINUX Command prompt:

user@host:[WD]$

e.g.

chintan@chintan-pc:~$	// indicates home folder for user chintan
chintan@chintan-pc:~/cprog$	// (after cd cprog/ command)
chintan@chintan-pc:/home$	// home directory
chintan@chintan-pc:/$	// root directory

in place of [WD]:

~ // indicates you are in home directory
/ // indicates you are in root directory

at the end of command prompt:

$ // normal user
// root user (admin)

VI Editor Commands:

`vi test.c` // creates a file with a name test.c and opens it in vi editor if file doesn't present with same name, if file with name test.c present already then shows its contents in vi editor

i // enter in insert mode, press esc to come outside of insert mode

```
/* commands outside insert mode */
```

w // advance to next word
e // advance to end of word
b // move to previous word

yy // copy line, press **p** to paste it after below line where cursor currently is
nyy // copy n lines after cursor
dd // delete line, press **p** to paste deleted line
ndd // deletes n lines after cursor

u // undo last
U // undo all changes to current line
o // add new line below cursor line
O // add new line above curser line

/word // move to occurrence of "word", to search next occurrence press **n**
:w // save changes
:w [new] // save changes to new file named "new"
:wq // save and quit
:q // quit
:q! // quit without saving changes

GCC (GNU Compiler Collection):

```
gcc test.c      |
    or          | compiles file test.c with default output file named a.out
cc test.c       |
```

```
gcc test.c -o test.exe    // compiles file with custom output file name
```

For executing binary file created by cc command,
write entire path of binary - e.g. **`/home/chintan/test.exe`**
or
go to that directory and type - **`./test.exe`**
here . indicates link to directory itself

if we write only executable's name even if we are in that function, it shows error and doesn't execute. We have to compulsory provide link to its own directory in order to execute that binary.

`a.out: command not found` // error displayed when we try to a.out without providing link i.e. executing with a.out in place of ./a.out

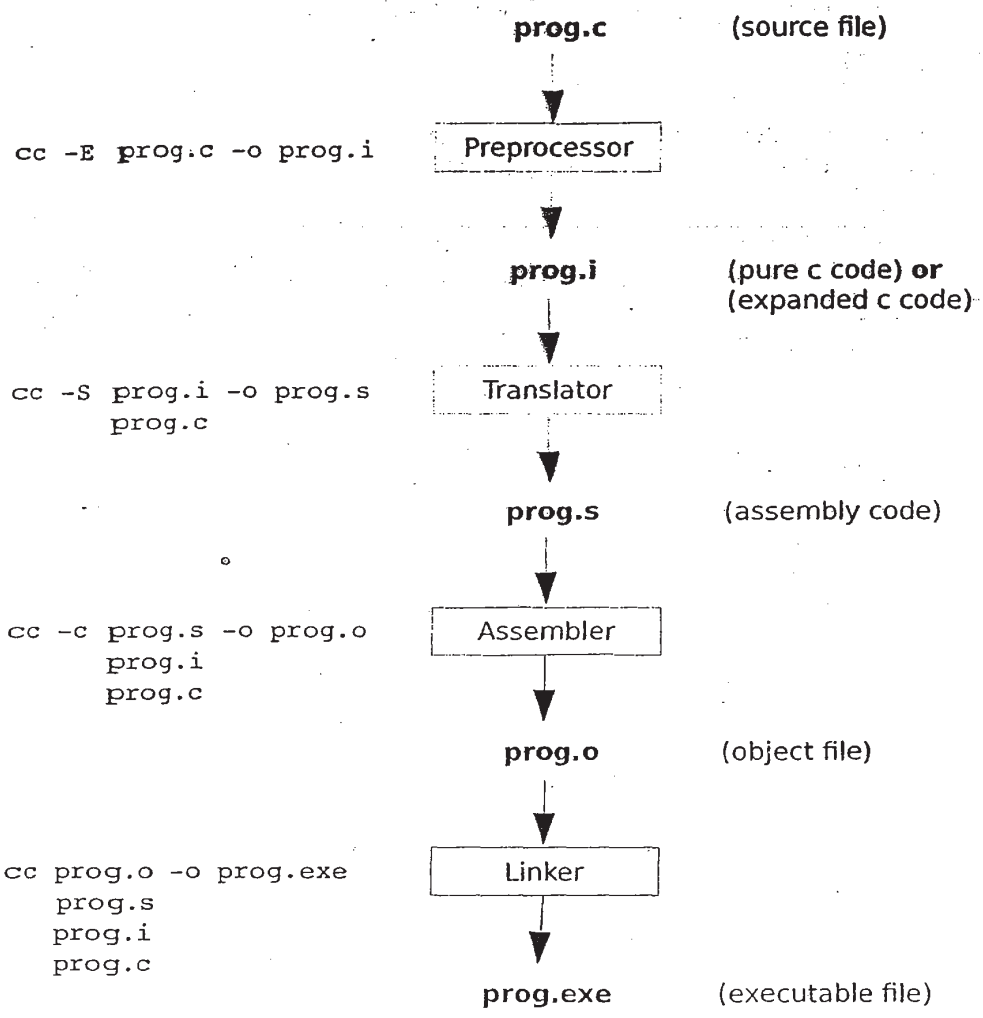
> **NOTE :** Commands of linux are nothing but executable files which are present in system path. System path contains address of directories where commands are present. When we write a command in command prompt (often called shell ot bash - bourne again shell) it searches for that command in directories present in system path. So that we do not have to write command's entire path to execute it. But in case of our user developed binaries, we have to provide link to its directory or write entire path in order to execute it. When we dont provide path or link, shell assumes that user is wanting to execute that command and searches for it in system path, but as that command is not found, it shows error message as shown above.

To see system path, type `echo $PATH` in command prompt.

> **NOTE :** LINUX/UNIX are extention independent but gcc is not. That means input files used with gcc must be having proper extention as they should be having, like for c files : test.c

Compiler Fundamentals

Compilation stages :



Preprocessing :

- header files are replaced by its contents by preprocessor (in place of #include statements, file's contents will be replaced and #include statements will be removed)
- comments are removed
- macro replacement is done by preprocessor

output of preprocessor is called pure c code (expanded c code). Because it will be having only c statements, preprocessor directives and comments are removed.

NOTE : Header file contains declarations of library function but function body is present in library file of that function

NOTE : Header files are written at the top of the program because it contains declarations of library functions which are used in our program, and functions must be declared first before they are called.

NOTE : It is not necessary to include header files, but then we will have to define all the functions we use in our program by ourself.

Translating :

- checks for syntax errors
- if any errors are not found, converts source code into assembly code

Assembling :

- converts assembly code into binary code

Linking :

- links calling function to called function (called function mapping)
- creates _start functions, which called main function in our program and _start also contains proper exit procedure

NOTE : Giving intermediate filenames extentions like .i .o .s is necessary, as they will be used by compiler as inputs. As GCC is not extention independent, it doesnt accept input files with any other extention than it should be having.

NOTE : In GCC function declaration is not strict (undeclared functions is not an error in gcc). ISO-C has made it a strict rule that any function used in the program must be declared. No declaration of function results in error.

Disassembler :

To see contents of binary file, disassemblers or hex editors are used. Linux has inbuilt disassembler called **objdump**.
To disassemble a executable or object file in Linux, following commands should be used.

```
objdump -D prog.exe/prog.o
```

ut using this command, it shows disassembles file contents in command rompt, if we want to view disassembled file in pages,

```
objdump -D prog.exe | less
```

Prog>
include<stdio.h>

```
main()
{
     printf("in main\n");
}
printf("after main\n");
```

o/p > syntax error, translating error (every statements should be in function body)

<Prog>

```
#include<stdio.h>
main()
{
     printf("in main");
}
f()
{
     printf("in f");
}
```

o/p > in main

<Q>. in above program, will f()'s body be present in executable?

Although function f() is not called from the program, using disassembler if we see its contents using command `objdump -D prog.exe | less`, we can see that f() is present in executable, but it is not called from main.

<Prog>

```
#include<stdio.h>
main()
{
     printf("in main\n");
     f();
}
f()
{
     printf("in f\n");
}
```

o/p > in main
in f

<Prog>

```
#include<stdio.h>
f1()
{
     printf("in f1\n");
}
main()
{
```

```
        f1();
        printf("in main\n");
        f2();
}
f2()
{
      printf("in f2\n");
}
```

```
o/p >  in f1
       in main
       in f2
```

NOTE : NO matter in what order functions are present in program & in what order in executable, _start function created by linker in executable calls main first. When main() returns a value, it is collected by _start function. _start also contains proper exit procedure. _start is an entry point. So when loader loads application into RAM, execution starts from _start function.

<Prog>

```
#include<stdio.h>
f1()
{
        printf("in f1\n");
}
first()
{
        f1();
        printf("in first\n");
        f2();
}
f2()
{
        printf("in f2\n");
}
```

```
o/p > /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-Linux-
      gnu/crt1.o: In function `_start':
      (.text+0x20): undefined reference to `main'
      collect2: error: ld returned 1 exit status
```

its linking stage error. Error message doesn't show in which line of program error has occurred because its linker stage error and linker only knows object file and doesn't know source file.)

-> To compile a program without main() function, we will have to create executable without a _start function. For that an option is available in gcc.

```
cc -nostartfiles prog.c
```

Compiling above program using switch -nostartfiles, it compiles and gives a warning as shown below.

```
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting
to 0000000000400340
```

--> Here if we see contents of executable of above program, we can see that 0000000000400340 is f1()'s address. This means that when a program is compiled without creating _start function, its execution starts with the function which is first in the executable. In program's executable f1() is present first.

--> Now f1() function has return statement in executable, but as no exit procedure is written, when executed gives runtime error segmentation fault.

```
o/p > in f1
      Segmentation fault (core dumped)
```

here only f1() executes, as compiler tries to execute return statement, it is segmentation fault because no exit procedure is written

<Prog>

```
#include<stdio.h>
first()
{
        f1();
        printf("in first\n");
        f2();
}
f1()
{
        printf("in f1\n");
}
f2()
{
        printf("in f2\n");
}
compiling program with cc -nostartfiles prog.c

o/p > in f1
      in first
      in f2
      Segmentation fault (core dumped)
```

if function first is present at top of all the functions, then execution starts from there, so it calls f1 and f2. But again due to no exit procedure is written, it is segmentation fault.

NOTE : We can introduce proper exit procedure in the program by writing exit (0);

<Prog>

```
#include<stdio.h>
first ()
{
        f1();
        printf("in first\n");
        f2();
        exit(0);
}
f1()
{
        printf("in f1\n");
}
f2()
{
        printf("in f2\n");
}
```

compiling program using cc -nostartfiles prog.c

gives us two warnings :

```
test1.c: In function 'first':
test1.c:7:2: warning: incompatible implicit declaration of built-
in function 'exit' [enabled by default]
  exit(0);
  ^
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting
to 0000000000400390
```

first warning is due to the fact that exit() function is present in stdlib.h header file, so #include<stdlib.h> should be included first in the program.

```
>/p > in f1
      in first
      in f2
```

NOTE : Writng main() function in program is not mandatory as just now we have seen it that without main() function programs can be written and executed also. But if main() is not present in our program, execution starts from first ever function available in executable. So writing a program without main() function, programmer will have to set every function in order, but if program is lenthy, this process becomes tiresome. So not to do so we write our programs using main() function. So that _start function will jump directly to main function no matter where main is present in our program. So using main() in our program ensures that we dont have to worry about the orders of functions present in our program.

NOTE : exit() function will work with any integer value. It is not mandatory to write exit(0); in our program, similarly in main() function, writing return 0; is not mandatory, we can write any value we want, it works perfectly.

Usage of -C switch with cc command :

```
--> cc -C prog.i -o prog.exe
       prog.s
       prog.o
```

using -C switch in cc command, it creates executable file from any intermediate file of different stages of compiler only. i.e. it takes only intermediate files like prog.i, prog.s, prog.o as its input. -c switch will not work with source files like prog.c

<Q> In c program development, what kind of errors do you encounter?

1). Compile time error
2). Run time error

Compile time errors :

preprocessing error :	invalid header file names invalid comments (nested comments)
translating error :	syntax error
assembling error :	not present (translator converts source file to assembly file, so if there is no error in translating stage, its output assembly code will be error free. That assembly code is used by assembler to generate object code. So assembly code is always error free
linking error :	no main() defined function called but not defined

Run time errors :

--> segmentation fault (due to trying to access or modify that portion of memory to which our application doesn't have permission to)
--> floating point exception (dividing by 0)

<Ex> `printf("hello","hi",1);`
o/p > `hello` (only 1st string considered)

NOTE : Declaration of the function just before function call is valid in c

NOTE : if only string is present in printf()'s like `printf("chintan");` then it is converted into puts() by compiler. We can see this by using disassembler on executable file. Similarly if only one character is present in printf() like `printf("c");` then it will be converted into putchar() by compiler.

This conversion takes place because printf() is a very complicated function And requires more time to execute than puts() or putchar(). So to increase Execution speed, compiler automatically does this kind of conversions.

Data types in C

Token : Each meaningful smallest information to compiler is called token.
e.g. operators, constants, keywords, identifiers, separators.

Basic data types :

Integral type	Real type
- char	- float
- short int	- double
- long int	- long double
- long long int	

Data :
- constant **or** - variable

Constant :

character type : 'a', 'S', 'A', '#'
integer constant : 5, -10, 0x12, 0b1010, 0567
string constant : "ABC", "1234"
real constant : 5.6, 5.6e+5

Variable :

syntax : `datatype var_name;`

var_name : cant be a keyword
must start with underscore or alpha bate
cant start with a number
cant contain special characters other that underscore

In C, upper case and lower case letters are significant. i.e. variable `chintan` and variable `CHINTAN` are two different variables.

--> Memory is allocated by every declared variable by compiler

char	:	1 byte
short int	:	2 bytes , 2
long int	:	4 bytes , 8
long long int	:	8 bytes , 8
float	:	4 bytes
double	:	8 bytes
long double	:	10 bytes , 12

NOTE : Integer's default size varies by compiler. gcc uses long int by default in place of int, turbo c uses short int by default in place of int.

NOTE : #include, #define are not tokens but they are preprocessor directives Compilation is basically translation, and preprocessor directives starting with # will be removed by preprocessor & will not be present in translation stage.

<Ex>

```
main()
{
      char v;    // declaration
      v='A';     // assignment
      putchar(v);
}
```

<Q> What is lvalue error?

--> It is assignment error. In assignment statement, on left side variable must be present who is having locations to which right side operand can be stored, if not then lvalue error occurs.

e.g.
```
main()
{
      'a'='b';
}
```

as 'a' is constant, and no memory is allocated for constants by compiler, so when we try to assign 'b' to 'a' it is lvalue error like shown below occur.

```
test.c: In function 'main':
test.c:3:4: error: lvalue required as left operand of assignment
  'a'='b';
     ^
```

Format specifiers for printf() & scanf()

char	:	%c
int	:	%d
unsigned int	:	%u
short int	:	%hd
unsigned short int	:	%hu
long int	:	%ld
unsigned long int	:	%lu
long long int	:	%lld
unsigned long long int	:	%llu
float	:	%f
double	:	%lf
long double	:	%llf
octal	:	%o
hex	:	%x
exponential	:	%e
string	:	%s
pointer	:	%p

ASCII Codes of some characters

A - Z : 65 to 90
a - z : 97 to 122
0 - 9 : 48 to 57

--> char v='a'; // **initialization** = declaration + assignment

<Prog>

```
#include<stdio.h>
main()
{
      int c,d,e,f;
      printf("c=%d d=%d\n e=%d\n f=%d\n",c,d,e,f);
      c=10;
      d=20;
      e=30;
      f=40;
      printf("c=%d d=%d\n e=%d\n f=%d\n",c,d,e,f);
}
o/p >  c=794093616 d=32767 e=0 f=0
       c=10 d=20 e=30 f=40
```

when a variable is not been initialized, when printed, it contains some garbage values. When values are assigned to then, then it prints values assigned to them. So in our program if our need is that a variable should not be containing garbage value, then it should be initialized first.

NOTE : printf() doesnt print values. It only prints string. e.g. if we try to print 100 using format specifier %d, then integer 100 is converted first into string "100", and then string "100" is printed on screen.

<Prog>

```
#include<stdio.h>
main()
{
        char v;
        v=getchar();
        printf("v = %c %d\n",v,v);
        v=getchar();
        printf("v = %c %d\n",v,v);
}
o/p > a
      v = a 97
      v =
        10
```

When program is executed, getchar() waits for user to input a character. When we press any character, it will not take it until we hit enter. When from common line inputs are given, they are present in input buffer. So if we have entered character 'a' and hit enter, then 'a' and '\n' is present in input buffer. First getchar() takes a from input buffer, but '\n' is still present. So 2nd getchar() takes it and printf a newline in place of printing \n. Its ASCII value is 10.

<Prog>
```
#include<stdio.h>
main( )
{
        char v=100;
        printf("v = %c\n",v);
}
```
o/p > v = d

<Prog>
```
#include<stdio.h>
main( )
{
        char v='d';
        printf("v = %c %d\n",v,v);
        v=v*2;
        printf("v = %c %d\n",v,v);
}
```
o/p > v = d 100
v = ? -56

strange character appears in second printf()'s execution because characters having ASCII values only up to 127 are printable. Values greater than 127 will result in strange characters. Default type of char is signed. So -56 is displayed in place of 200.

200 -> 1 1 0 0 1 0 0 0

negative numbers are stored in 2's compliment format. As there is 1 at MSB, it indicates minus sign, so it understands that a negative number was stored there after converting it into its 2's compliment form. To represent it in decimal form, again 2's compliment is taken and minus sign is put before the number.

2's comp 0f 200 -> 0 0 1 1 1 0 0 0 --> -56

Range of char
unsigned char : 0 to 255
0 : 0 0 0 0 0 0 0 0
255 : 1 1 1 1 1 1 1 1

signed char : -128 to 127
0 : 0 0 0 0 0 0 0 0
127 : 0 1 1 1 1 1 1 1
-1 : 1 1 1 1 1 1 1 1
-128 : 1 0 0 0 0 0 0 0

NOTE : Signed type is default for all untill we declare variable as unsigned

Type Modifiers :
- signed
- unsigned
- long
- short

Type Qualifiers :
- const
- volatile

<Prog>
```
#include<stdio.h>
main()
{
        unsigned short int a=-1;
        unsigned long int b=-1;
        printf("a=%hu\n",a);
        printf("b=%lu\n",b);
}
```
o/p > a=65535
b=4294967295

saving -1 in unsigned integers, prints highest value of its range

Range of short int
unsigned : 0 to 65535
signed : -32768 to 32767

NOTE :						
Highest	+	1	-->	Lowest		
Lowest	-	1	-->	Highest		
Highest	+	x	-->	Lowest	+	(x-1)
Lowest	-	x	-->	Highest	-	(x-1)

<Prog>
```
#include<stdio.h>
main()
{
        unsigned short int a;
        short int b;
        a=b=-5;
        printf("a=%hu\n",a);
        printf("b=%hd\n",b);
```

o/p > a=65531 // lowest – x = highest – (x-1) --> 0-5=65535-(5-1)
b=-5

<Prog>
```
#include<stdio.h>
main()
{
        unsigned short int a=32767;
        short int b;
        a=b=a+3;
        printf("a=%hu\n",a);
        printf("b=%hd\n",b);
}
```
o/p > a=32770
b=-32766

<Q> using particular format specifier for that data type is mandatory or not?
--> It is not mandatory but assume that unsigned short int's data is printed using %hd (short int) format specifier. If value stored in variable is less than 32767, then there will not be any problem. But if value is greater than 32767, then printing it using %hd will show negative value, but as our variable is unsigned short int, it should be positive. To avoid such kind of bugs in a program, its a good practice to use proper format specifiers.

<Prog>
```
#include<stdio.h>
main()
{
        char a;
        scanf("%d",&a);
        printf("a = %c %d\n",a,a);
}
```
o/p > 97
a = a 97

when compiled, gives us warning, but such kind of usages of format specifiers are impractical and of no use. On some system, it also gives segmentation fault because %d specifier is used with scanf, it understands that integer data has been entered. As default int's size is 4 bytes, it tries to save input data in 4 bytes starting from the variable's address we passed to scanf(), but as only 1 byte is allocated for char type variable a. So it is segmentation fault.

```
test.c: In function 'main':
test.c:5:2: warning: format '%d' expects argument of type 'int *',
but argument 2 has type 'char *' [-Wformat=]
  scanf("%d",&a);
  ^
```

<Prog>
```
#include<stdio.h>
main()
{
        char a;
        scanf("%hhd",&a);
```

```
        printf("a = %c %d\n",a,a);
}
o/p >  67
       a = C 67
```

if %hhd format specifier is used in place of %d, then program compiles without any kind of error or warning. Thats because short short integers are of 1 byte. And char variable used is also of 1 byte. So its able to store the value in variable without any trouble. But such kind of practice should be avoided, because gcc is not very strict compiler but a strict compiler like turbo c may give error.

Real data

```
float  a;
double  b;
long double c;
```

IEEE-754 (floating point number representation)

Every floating point number has got 3 parts

	float (32-bit)	double (64-bit)
sign	1	1
exponent	8	11
significand	23	52

float -->

31	30 ... 23	22 ... 0

double-->

63	62 ... 52	51 ... 0

Base exponent is not having 0 value for exp=0, but it has some default values for zero exponent. And for other exponent values , its biased values will be stored.

Base exponent default values for exp=0
float --> 127
double --> 1023

consider for float data,
if exp=2 then base exp = 127+2 = 129
if exp=-2 then base exp = 127-2 = 125

Binary based exponential format :

= **1.bbbbbb e ±d** // ± - sign, bbbbbb - significand, +d - exponent

Binary data for real values are not stored in this format,its for our understanding. This is to understand the process of how real data will be actually stored in memory.

<Ex> How 5.5 will be stored in memory?

$(5.5)_{10} = (101.1)_2$

$+ 1.011 \text{ e } +2$

sign - 0 ; significand – 011 ; exponent – 127+2 = 129 (for float)
exponent – 1023+2 = 1025 (for double)

0	10000001	01100000000000000000000	⟶ 5.5 float

0	10000000001	011000000000.....0000	⟶ 5.5 double

<Ex> 0.25 in float

$(0.25)_{10} = (0.01)_2$

$+ 1.0 \text{ e } -2$

sign - 0 ; significand – 00 ; exponent – 127-2 = 125 (for float)

0	01111101	00000000000000000000000	⟶ 0.25 float

<Ex> 23.4 in float

$(23.4)_{10} = (10111.\overline{0110})_2$

$+ 1.01110\overline{0110} \text{ e } +4$

sign – 0 ; significand – $01110\overline{0110}$; exponent – 127+4 = 131 (float)
exponent – 1023+4 = 1027 (double)

0	10000011	01110110011001100110011	⟶ 23.4 float

0	10000000011	011101100110.....0110	⟶ 23.4 double

<Prog>

```
#include<stdio.h>
main()
{
        float a=5000000000.5;
        printf("a = %f\n",a);
        a=a*2.0;
        printf("a = %f\n",a);
}
```

```
o/p > a = 5000000000.000000
      a = 10000000000.000000
```

<Q> Float variable's memory is of 4 byte, so how come it is able to store and display values even greater that 2^{32} ?
--> from IEEE-754 format, we know that base exponent value for float is 127. Base exponent for float is having 8 bits so,

max number which can be added to base --> 255-127 = 128
min number which can be subtracted from base --> 127

so floating point range according to these,
1 bbbb....b . bbbb...b
  128   127

$2^{128} . 1/(2^{127})$
thats why float is able to store such huge amount compared to integer variable.

Operators

Operator : Operator is a symbol meaningful to compiler

Classification :

Based on number of operands
- Unary operator
- Binary operator
- Ternary operator

Based on type of operation
- Assignment
- Arithmetic
- Logical
- Relational
- Bitwise

Assignment Operator :

```
syntax :    var = var;
            var = const;

const = var;   // lvalue error.

--> a=b=c=20;
```

if all operators are having same priority, associativity rule comes into play.
For assignment associativity --> right to left

statement `a=b=c=20;` will take below steps to execute,
c=20
b=c
a=b

```
-> a=b=30=c=20;      // lvalue error.
```

Arithmetic Operator :

*, /, % --> high precedence
+, - --> low precedence

Associativity : left to right among equal precedence

<Ex>

```
 nt a=5,b=2;
int c;
=a/b;     // o/p > 2
=a%b;     // o/p > 1
```

<Ex>

```
float  a=5.0,b=2.0,c;
c=a/b;      // o/p > 2.5
c=a%b;      // o/p > syntax error
```

NOTE : modulo division % is not applicable on real data, it is only applicable on integer data

<Prog>

```
#include<stdio.h>
main()
{
      int a=5,b=2;
      float c;
      c=a/b;
      printf("c=%f\n",c);
}
```

o/p > c=2.000000

c=a/b, first a/b will be evaluated to 2, then 2 will be assigned to c, but as c is float variable, 2 is converted to 2.0 and assigned to c.

Arithmetic operator is having high precedence than assignment operator.

Type casting

Two operands using arithmetic operator must be of same type. (e.g. int/int, float*float). If not then compiler converts one operand to other's type. It is temporary conversion for performing that operation only. Original data is unaffected by type casting.

Implicit type casting

Type casting done by compiler is called implicit type casting. Implicit type casting is upper type casting i.e. promoting to upper type.

char & int - char converted to int type
int & float - int converted to float type
float & double - float converted to double

<Prog>

```
#include<stdio.h>
main()
{
      int a=5;
      float b=2.0;
      float c;
      c=a/b;
      printf("%f\n",c);
}
```

```
o/p > c=2.5     // a/b = 5.0/2.0 (integer 5 type casted to float
                   5.0)
                // c = 2.5
```

Explicit type casting
if variable type is changed by user, then it is called explicit type casting.
e.g. int a;
```
     (float)b;  // explicit type casting
```

<Prog>
```
#include<stdio.h>
main()
{
     int a=5;
     float b=2;
     float c;
     c=a/(int)b;
     printf("%f\n",c);
}
o/p > c=2.0     // a/(int)b --> 5/2 --> 2 (∵ both int operands)
                // 2 will be assigned to c, but c is float, c=2.0
```

Relational Operator

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equality
!=	inequality

syntax : `operand1<op>operand2; // e.g. a==b;`

Here operands can be constants, variables or expressions
Result of these expressions can be either true or false
true - 1
false - 0

unsigned int → only (+)
signed int → (-) & (+)

<Prog>
```
 include<stdio.h>
main()
{
     unsigned int a=5;
     int b=-2,c;
     c=a>b;
     printf("c=%d\n",c);
```

```
/p > c=0
```

If one signed and other unsigned operands are present in expression, then signed is implicitly type casted into unsigned.

c=a>b --> b is type casted to unsigned, so -2 in unsigned will be 4G-2, so final expression will become 5>(4G-2)--> false (∴ c=0)

<Prog>

```
#include<stdio.h>
main()
{
      int a=5;
      float b=5.5;
      int c;
      c=b>a;
      printf("c=%d\n",c);
}
o/p > c=1
```

<Prog>

```
#include<stdio.h>
main()
{
      int a=5;
      float b=5.5;
      int c;
      c=(int)b>a;     // (int)b>a --> 5>5 --> false
      printf("c=%d\n",c);
}
o/p > c=0
```

NOTE : All real constants are double type by default & integer constants are signed by default

<Prog>

```
#include<stdio.h>
main()
{
      float b=5.4;
      int c;
      c = b==5.4;
      printf("c=%d\n",c);
}
o/p > c=0
```

As real constant 5.4 is double, b will be type casted into double. But this conversion is temporary and 5.4 is compared with double equivalent of b. b's value or type will not be changed in this type conversion.

$(23.4)_{10} = (0101.\overline{0110})_2$ --> $1.01\overline{0110}$ e +2

0	10000001	01011001100110011001100	► b

0	10000000001	0101100110011001100110.....0110	► 5.4 double

0	10000000001	01011001100110011001000....0000	► b's double equivalent

When after floating point, repeating bits are there in its double equivalent as shown above (0110 bits are repeated in significand), b's double equivalent will be placed in its double significand as it is, float's bits are shown by underline in its double equivalent, and then remaining bits will be filled by 0. it is clear that b==5.4 is false.

but if significand is such that after floating point, non repeating values appear, then actual double and float's double equivalent will be equal. In such cases result will be true.

<Ex>
```
float b=5.5;
int a;
a = b==5.5;    // true, c=1
```

<Ex>
```
float b=5.4;
int a;
a = b==5.4f;   // true, c=1
```

<Prog>
```
#include<stdio.h>
main()
{
     float a=5.4;
     double b=5.4;
     c = a==(float)b;
     printf("c=%d\n",c);
}
o/p > c=1
```

<Prog>
```
 include<stdio.h>
main()
{
     float a=5.4;
     double b=5.4;
     c = (double)a==b;
     printf("c=%d\n",c);
}
o/p > c=0
```

Logical operator

&& - logical AND | --> binary operators
|| - logical OR |
! - logical NOT --> unary operator

<Ex>

```
int b=30, c=0, d;
d=b&&c;    // d=0
d=b||c;    // d=1
d=!c;      // d=1
```

Here, any nonzero (positive or negative) value is considered as 1 or TRUE, 0 is considered as 0 or FALSE.

<Ex>

```
int a=20, b=30, c=0, d;
d=(a>0)&&(b>0);      // d=1
d=(a<0)&&(b<0);      // d=0
```

<Prog>

```
#include<stdio.h>
main()
{
     int a=2, b=0, c=-3, d;
     d=(a=b)&&(b=c);
     printf("%d %d %d %d\n",a,b,c,d);
}
o/p > 0 0 -3 0
```

NOTE : gcc compiler adopts optimization methods in order to execute the program faster.

in above expression, with logical AND, if first operand is 0, then second operand will be skipped without executing or checking its condition.

<Prog>

```
#include<stdio.h>
main()
{
     int a=2, b=0, c=-3, d;
     d=(a=b)||(b=c);
     printf("%d %d %d %d\n",a,b,c,d);
}
o/p > 0 -3 -3 1
```

similarly with logical OR, if first operand is 1, then second operand will be skipped.

<Prog>

```
#include<stdio.h>
main( )
{
      int a=2, b=0, c=-3, d;
      d=(c=b)||(b=a)||(a=c);
      printf("%d %d %d %d\n",a,b,c,d);
}
```

o/p > 2 2 0 1

first (c=b) || (b=a) --> 0 || 2 --> 1
then 1 || (a=c) --> 1

a	b	c	d	
2	0	-3		
2	0	0		(c=b)
2	2	0		(b=a)
2	2	0	1	

<Prog>

```
#include<stdio.h>
main()
{
      int a=2, b=0, c=-3, d;
      d=(a=b)&&(b=c)&&(c=a);
      printf("%d %d %d %d\n",a,b,c,d);
}
```

o/p > 0 0 -3 0

first (a=b) && (b=c) --> 0 && (b=c) --> 0
then 0 && (c=a) --> 0

a	b	c	d
2	0	-3	
0	0	-3	0

<Prog>

```
#include<stdio.h>
main()
{
      int a=2, b=0, c=-3, d;
      d=(a=b)&&(b=c)||(c=a);
      printf("%d %d %d %d\n",a,b,c,d);
}
```

o/p > 0 0 0 0

first (a=b) && (b=c) --> 0 && (b=c) --> 0
then 0 || (c=a) --> 0 || 0 --> 0

a	b	c	d
2	0	-3	
0	0	-3	
0	0	0	0

<Prog>

```
#include<stdio.h>
main()
{
      int a=2, b=0, c=-3, d;
      d=(a=b)||(b=c)||(c=a);
      printf("%d %d %d %d\n",a,b,c,d);
```

o/p > 0 -3 0 0

	a	b	c	d
first (a=b) || (b=c) --> 0 || -3 --> 1	2	0	-3	
then 1 && (c=a) --> 1 && 0 -->	0	-3	-3	
	0	-3	0	0

Bitwise operator

& : bitwise AND
| : bitwise OR
^ : bitwise Ex-OR
<< : left shift
>> : right shift
~ : complement

NOTE : Bitwise operators are not applicable on real data. It is only applicable on integer data or characters.

<Prog>

```
#include<stdio.h>
main()
{
     int a=20, b=10, c;
     c=a&b;
     printf("%d\n",c);
     c=a|b;
     printf("%d\n",c);
     c=a^b;
     printf("%d\n",c);
}
o/p > 0
      30
      30
```

To explain bitwise operation results, last significant 8 bits are considered.

```
a     -->   0 0 0 1 0 1 0 0   -->   20
b     -->   0 0 0 0 1 0 1 0   -->   10
d     -->   0 0 0 0 0 0 0 0   -->   a&b = 0
d     -->   0 0 0 1 1 1 1 0   -->   a|b = 30
d     -->   0 0 0 1 1 1 1 0   -->   a^b = 30
```

<Ex>
-1 & 50

```
-1     -->   1 1 1 1 1 1 1 1
50     -->   0 0 1 1 0 0 1 0
result -->   0 0 1 1 0 0 1 0
```

-1 ANDed with number = number itself

<Ex>
0 | 30

0 --> 00000000
30 --> 00011110
result --> 00011110

0 ORed with number = number

<Ex>
-1 | 30 = -1

<Ex>
0 & 30 = 0

<Prog>
```
#include<stdio.h>
main()
{
        int a=20,b=10;
        b=a*b/a=b;
        printf("%d %d\n",a,b);
}
```
o/p > lvalue error

Arithmetic operator is having high precedence than assignment operator. So, arithmetic operations will be done from left to right.
First a*b --> result, then, result/a --> result1, then b=result1=b

assignment operations will be done from right to left
First result1=b, --> lvalue error

```
a=b+c;     // valid
a+b=c;     // invalid
a+(b=c);   // valid, but result will not be stored anywhere,
              useless statement.
```

<Prog>
```
#include<stdio.h>
main()
{
     int a=2,b=1,c=3;
     c+=b+=a+=5;
     printf("%d %d %d\n",a,b,c);
```

o/p > 7 8 11

first a+=5 --> a=a+5 --> a=7
then b+=a --> b=b+a --> b=8
then c+=b --> c=c+b --> c=11

<Prog>

```
#include<stdio.h>
main( )
{
     int a=22,b=11;
     a^=b;
     printf("%d %d\n",a,b);
}
o/p >  29 11
```

```
a     -->   00010110   -->   22
b     -->   00001011   -->   11
a^b   -->   00011101   -->   29
```

Swapping algorithms :

<Prog>

```
#include<stdio.h>   // swapping method-1
main()
{
     int a=20,b=10;
     b=a*b/(a=b);
     printf("a=%d b=%d\n",a,b);
}
o/p >  a=10 b=20
```

b=a*b/(a=b) --> b=200/(a=b) --> b=200/10 (a=10) --> b=20

But this method will not work if multiplication of two integer values exceed default range.

<Prog>

```
#include<stdio.h>   // swapping method-2 (using ex-or)
main()
{
     int a=22,b=11;
     a^=b^=a^=b;
     printf("a=%d b=%d\n",a,b);
}
o/p > a=11 b=22。
```

This method swaps bitwise, so it can swap any value which is within the range of integer.

First a^=b --> a=a^b --> 22^11=29 (a=29)
then b^=a --> b=b^a --> 11^29=22 (b=22)
then a^=b --> a=a^b --> 29^22=11 (a=11)

<Prog>

```
#include<stdio.h>   // Swapping method-3 (using temp variable)
main( )
{
      int a=22,b=11,temp;
      temp=a;
      a=b;
      b=temp;
      printf("a=%d b=%d\n",a,b);
}
o/p >  a=11 b=22
```

As this method stores values in third temporary variables, this method can also swipe any values up to default range of integer.

<Prog>

```
#include<stdio.h>   // Swapping method-4
main()
{
      int a=22,b=11;
      a=a+b;
      b=a-b;
      a=a-b;
      printf("a=%d b=%d\n",a,b);
}
o/p > a=11 b=22
```

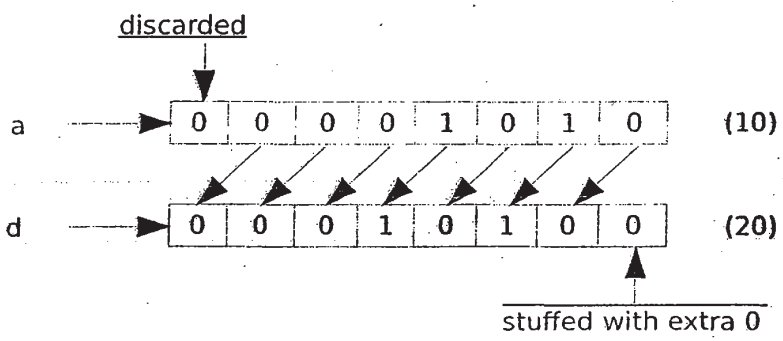
a=a+b --> a=33, b=11
b=a-b --> a=33, b=22
a=a-b --> a=11, b=22

this method will work only when addition of two integers is within its default range.

Method-2 (Ex-OR)	Method-3 (temp variable)
--> executes in 6 operations a^=b^=a^=b; a=a^b; // first ^ then = b=b^a; a=a^b;	--> executes in 3 operations temp=a; a=b; b=temp;
--> doesn't require any third variable	--> requires third variable
--> won't work on real data (because bitwise operations are not compatible with real data)	--> will work on real data

<Ex>
```
d=a<< 2;    // left shift by 1
            // equivalent to multiplication by 2
```

suppose a=10



d=a<<1 ; // d=ax2^1 = ax2
d=a<<2 ; // d=ax2^2 = ax4
d=a<<3 ; // d=ax2^3 = ax8

NOTE : In bitwise operations, value of a will remain unchanged, result after shifting will be stored in another variable

d=a>>1 ; // d=a/2^1 = a/2
d=a>>2 ; // d=a/2^2 = a/4

<Ex>

```
--> printf("%d",1<<5);          // o/p > (1x2^5) = 32
--> printf("%d",1<<10);         // o/p > (1x2^10) = 1024
--> printf("%d",1<<20);         // o/p > (1x2^20) = 1M
--> printf("%d",1<<30);         // o/p > (1x2^30) = 1G
--> printf("%d",1<<31);         // o/p > (1x2^31) = -2G
```

1<<31 : will have 1 in only 31st bit which is sign bit, so it will be equivalent to -2G

```
--> printf("%d",1<<33);         // o/p > 0
```

gives warning

```
test.c: In function 'main':
test.c:6:2: warning: left shift count >= width of type
[enabled by default]
 printf("%d\n",1<<33);
  ^
```

```
--> int a=1,b=33;
    printf("%d\n",a<<b); // o/p > 2
```

beyond 31 shifts in integer variable, we cant judge it will be rotation or shifting

<Ex>

```
int a=10,b=50,c=3,d;
a=a<<c;              // a=a<<3, ax2 --> a (20)
                                ax2 --> a (40)
                                ax2 --> a (80)
printf("%d\n",a);

b>>=c ;              // b=b/2, b/2 --> b (25)
                              b/2 --> b (12)
                              b/2 --> b (6)
printf("%d\n",b);
```

o/p > 80
6

<Ex>

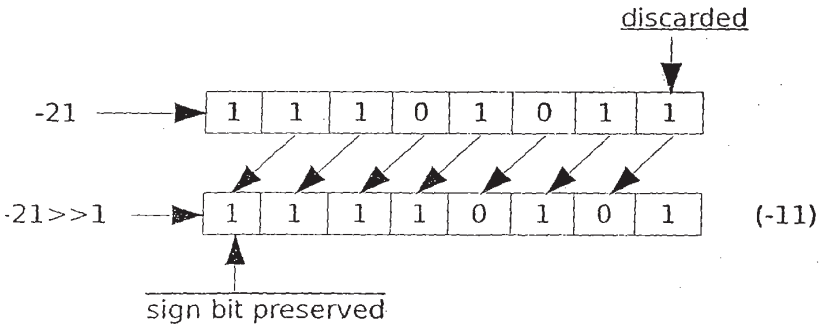
```
int a=-10, b=-50, c,d;
c=a<<2 ;
d=b>>1 ;
printf("%d %d\n",c,d);
```

o/p > -40 -25

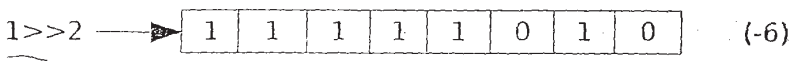
<Ex>

```
printf("%d\n",-21>>1);
printf("%d\n",-20>>1);
printf("%d\n",-19>>1);
```

o/p > -11
-10
-10



NOTE : In negative number shifting operatons, sign bit is preserved by sign copy mechanism



For odd number :
-21>>1 : expectation = -10
but answer = -10-1
=-11

Similarly for -19>>1 : expectation = -9
but ans = -9-1
= -10

for even numbers
-20>>1 : expectation = -10
answer = -10

<Ex>

```
-50>>3      --> -50/2 = -25
                -25/2 = -13 (-12-1)
                -13/2 = -7  (-6-1)

-5>>3       --> -5/2 = -3
                -3/2 = -2
                -2/2 = -1

-3>>3       --> -3/2 = -2
                -2/2 = -1
                -1/2 = -1 (0-1)
```

NOTE : -50>>3 : it will not shifted right 3 at once, but it will be shifted once 3 times

<Ex>

```
int a=10,b=-50,c,d;
c=~a;
d=~b;
printf("%d %d %d %d\n",a,b,c,d);

o/p > 10 -50 -11 49
```

a 00001010 (10)
~a 11110101 (-11) --> ~a
b 11001110 (-50)
~b 00110001 (49) --> ~b

1's comp(x) = 2's comp(x) -1
= -x-1
1's comp(-x) = 2's comp(-x) -1
= x-1

<Ex>
~(456) = -456-1 = -457
~(-45) = 45-1 = 44
~(-1) = 1-1 = 0
~(0) = 0-1 = -1
~(1) = -1-1 = -2

<Prog>
```
#include<stdio.h>
main( )
{
     int v=50;
     printf("%d\n",(v>>3)&1);
}
```

o/p > 0

```
50          --> 00110010
50>>3       --> 00000110
(50>>3)&1   --> 00000110
                00000001
                00000000 --> 0
```

similarly if
(v>>4)&1 --> o/p > 1

```
(v>>4)&1  --> 00000011
              00000001
              00000001 --> 1
```

NOTE : Bitwise operations are faster than arithmatic operation

sizeof( )

```
syntax :    sizeof(int/float/double);     |
            sizeof(variable);             | returns bytes occupied
            sizeof(constant);             |
```

<Ex>
```
int i;
char ch;
printf("%d",sizeof(i));       // o/p > 4
printf("%d",sizeof(ch));      // o/p > 1
```

<Ex>
```
printf("%d",sizeof(char));    // o/p >1
printf("%d",sizeof('c'));     // o/p > 4
```

--> here 'c' is char constant and it should return with 1, but char constants are represented by its ASCII value, which is an integer value. So sizeof() operator used with character constant returns integer's default size, which is 4 in gcc.

<Ex>
```
printf("%d",sizeof(50));      // o/p > 4
printf("%d",sizeof(5.0));     // o/p > 8
printf("%d",sizeof(5.0f));    // o/p > 4
```

--> default real constants are of double type so sizeof(5.0) will return with 8 value.

NOTE : sizeof() is only operator which is keyword. It looks like function but it is not a function

<Ex>
```
sizeof("char");      // o/p > 5
sizeof("c");         // o/p > 2
sizeof("50");        // o/p > 3
sizeof("5.0");       // o/p > 4
```

--> every string constant contains null character at its end, so sizeof() with string constant returns with total char in string + 1 null character

Increment / Decrement

v++ --> post increment
++v --> pre increment
v-- --> post decrement
--v --> pre decrement

NOTE : Increment decrement operators are not allowed on constant.

<Ex>
```
int a=1;
int b;
printf("%d %d\n",a,b);   // o/p > a=1, b=garbage

b=a++;
printf("%d %d\n",a,b);   // o/p > a=2, b=1

b=++a;
printf("%d %d\n",a,b);   // o/p > a=3, b=3

b=a--;
printf("%d %d\n",a,b);   // o/p > a=2, b=3
```

```
b=--a ;
printf("%d %d\n",a,b);   // o/p > a=1, b=1
```

b=a++; first b=a then a++
b=++a; first a incremented then b=a
b=a--; first b=a then a--
b=--a; first a decremented then b=a

<Ex>

```
int a=1,b=1,c;

c=a++ + b++;
printf("%d %d %d",a,b,c);      // o/p > a=2, b=2, c=2

b=++a + ++c;
printf("%d %d %d",a,b,c);      // o/p > a=3, b=6, c=3

a=b-- - --c;
printf("%d %d %d",a,b,c);      // o/p > a=8, b=5, c=2
```

<Ex>

```
int a=1,b=1,c=1;
a=b++ + ++c + a++;
printf("%d %d %d",a,b,c);      // o/p > a=5, b=2, b=2
```

--> here 4 will be assigned to a, but after assignment, one post increment on a is done so value of a=5

<Ex>

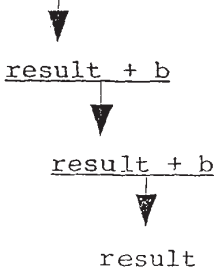
```
int a=1,b=1;
a=b++ + b++ + b++ + b++;
printf("%d %d",a,b);           // o/p > a=4, b=5
```

<Ex>

```
int a=1,b=1;
a=++b + ++b + ++b + ++b;       // o/p > a=15, b=5
```

--> here actual operation will be

```
a= b + b + b + b        // arithmetic operation execution : left to right
```



consider first b+b, after pre-increment value of first value of first b is 2 and second b is 3. but addition is being on same variable and same variable can not have two different values. So latest value of b will be considered.

a=3+3
6 + ++b(4)
10 + ++b(5)
15 --> assigned to a

these kind of statements are not used practically, so they are not standardized. Different compilers behave differently for it. Behavior of gcc is explained above, but turbo c behaves different.

In turbo c --> a=b+b+b+b is considered as a=4*b(4 times incremented) so a will be 20 in turbo c

Conditional operator

syntax : operand 1 ? operand 2 : operand 3;

test condition	statement on true	statement on false

test condition, statement on true and statement on false can be : variable expression, constant and function call

Conditional operator is only ternary operator in c.

NOTE : Keywords are not allowed as any part of the operands

<Ex>

```
int a,b,c;
printf("enter int for a & b:");
scanf("%d %d",&a,&b);
(a==b) ? printf("equal\n") : printf("not equal\n");

o/p > if a=b=12 --> equal
      if a=12, b=13 --> not equal
```

<Ex>

```
int a,b,c;
printf("enter a & b:");
scanf("%d %d",&a,&b);
c=(a>b)?a:b;
printf("c=%d",c);
```

```
o/p > if a=5,b=4 --> c=5 (true, so a value)
      if a=3,b=4 --> c=4 (false, so b value)
      if a=3,b=3 --> c=3 (false, so b value)
```

Testing the bit in data

```
c=(a>>b)&1;          | both are equal, only two answers are possible, in both
c=((a>>b)&1)?1:0;    | case, either 0 or 1 are possible results

c=(a&(1<<b));  // returns integer values. (not only 0 and 1 but any integer)
c=(a&(1<<b)?1:0);   // returns only 1 or 0
```

these all statements are used to test bth bit in data a, whether it is 0 or 1.

in case of `a&(1<<b);` if a=20 and b=2

```
a -->       00010100
1<<b -->    00000100
            00000100 (4) --> returns any integer value
```

in all other statements only 1 or 0 is possible result.
e.g. `(a>>b)&1;` if a=20 and b=2

```
a>>b -->    00000101
1 -->       00000001
            00000001 (1)
```

<Prog>

```
#include<stdio.h>
main()
{
     int a,b;
     printf("enter a & b");
     scanf("%d %d",&a,&b);
     (a==b) ? printf("eq") : (a>b) ? printf("a") : printf("b");
}
```

--> if more than one conditional operators are present, then compiler automatically groups them as shown below.

1 ? (eq) : c2 ? (a>b) : (a<b) ;

NOTE : Grouping is done right to left, but execution is from left to right.

conditions are as follows,

A ? B ? C ? D : E : F : G ;
hen compiler will do grouping as follows

```
A? B ? C ? D : E : F : G;
```

first of all, A will be checked because execution is from left to right. if
A is false then G is executed, but if
A is true then B will be checked, if
B is false then F will be executed, if
B is true then C will be checked. If
C is true then D and if false then E will be executed.

NOTE : Grouping is done in order to divide complex expressions in single simplified steps. It is mandatory and done by compiler wherever needed.

<Assignment> Scan a character and determine its type whether its upper case, lowercase digit or special character using conditional operator.

```
#include<stdio.h>
main()
{
        char ch;
        printf("Enter a character:");
        scanf("%c",&ch);

        ((ch>96)&&(ch<123))?printf("lower case\n"):
        ((ch>64)&&(ch<91))?printf("upper case\n"):
        ((ch>47)&&(ch<56))?printf("digit\n"):
        printf("special character\n");
}
```

<Assignment> Scan 3 integers and find relation between them. Find which is higher, all three, two of them or only one using only conditional operator.

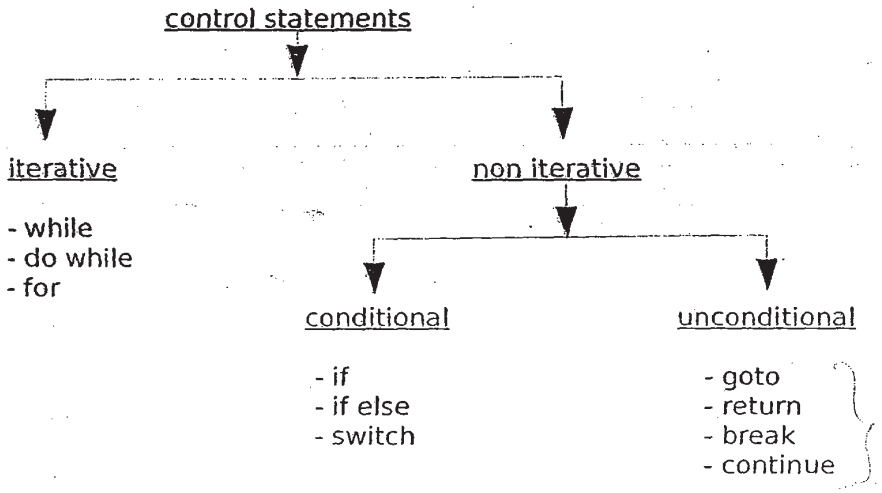
```
#include<stdio.h>
main()
{
        int a,b,c;
        printf("Enter three integers a,b & c:");
        scanf("%d %d %d",&a,&b,&c);

        ((a>b)&&(a>c))?printf("a max\n"):
        ((b>a)&&(b>c))?printf("b max\n"):
        ((c>a)&&(c>b))?printf("c max\n"):
        ((a==b)&&(a>c))?printf("a & b max\n"):
        ((b==c)&&(b>a))?printf("b & c max\n"):
        ((c==a)&&(c>b))?printf("c & a max\n"):
        printf("a,b & c max\n");
}
```

<Assignment> Scan 3 integers and find complete relation between them using conditional operator only.

```
#include<stdio.h>
main( )
{
        int a,b,c;
        printf("Enter three integers a,b & c:");
        scanf("%d %d %d",&a,&b,&c);

        ((a==b)&&(a>c))?printf("a=b and both high\n"):
        ((a==b)&&(a<c))?printf("a=b and c high\n"):
        ((b==c)&&(b>a))?printf("b=c and both high\n"):
        ((b==c)&&(b<a))?printf("b=c and a high\n"):
        ((c==a)&&(c>b))?printf("c=a and both high\n"):
        ((c==a)&&(c<b))?printf("c=a and b high\n"):
        ((a>b)&&(a>c))?printf("a high\n"):
        ((b>c)&&(b>a))?printf("b high\n"):
        ((c>a)&&(c>b))?printf("c high\n"):
        printf("a=b=c\n");
}
```

Control Statements

Control statements are of two types
1). Iterative (Repetitive)
2). Non-iterative (Non-repetitive)



If :

syntax :

```
if(boolean condition)
{
     statement;
}
```

only if block

```
if(boolean condition)
{
     statement;
}
else
{
     statement;
}
```

if else blocks

here boolean condition can be : constant, variable, expression

--> when if or else block contains only one statement, writing braces is not necessary

--> in conditional operators, in true or false statements we can not use keywords. Operands of conditional operator can not be keywords. So goto, return, break, continue are not allowed in conditional operator but they are allowed in if or else statements.

<Prog> enter a character and determine whether lowercase, uppercase, digit or special character using if.

```
#include<stdio.h>
main( )
{
        char ch;
        puts("Enter a character:");
        ch=getchar();

        if((ch>='a')&&(ch<='z'))
                printf("lowercase\n");
        if((ch>='A')&&(ch<='Z'))
                printf("uppercase\n");
        if((ch>='0')&&(ch<='9'))
                printf("digit\n");
        if(!(((ch>='a')&&(ch<='z'))||((ch>='A')&&(ch<='Z'))||
        ((ch>='0')&&(ch<='9'))))
        printf("special character\n");
}
```

in above program only one statement will be executed from 4 conditions, but all conditions will be checked. If we don't want to check conditions after one condition is true, then else if ladder should be used.

Else if ladder :

syntax :

```
if(condition)
{
     statement;
}
else if(condition)
{
     statement;
}
else if(condition)
{
     statement;
}
else
{
     statement;
}
```

NOTE : if in else if ladder, else is forgotten, it is not syntax error but bug appears in program. else if ladder is not neccessarily ended with else

else if ladder should be used when all the conditions are related. i.e. in any case only one condition will be true from all conditions.

<Prog> enter a character and determine whether lowercase, uppercase, digit or special character using else if ladder.

```
#include<stdio.h>
main()
{
        char ch;
        printf("Enter a character:");
        scanf("%c",&ch);

        if((ch>='a')&&(ch<='z'))
                printf("lowercase\n");
        else if((ch>='A')&&(ch<='Z'))
                printf("uppercase\n");
        else if((ch>='0')&&(ch<='9'))
                printf("digit\n");
        else
                printf("special character\n");
}
```

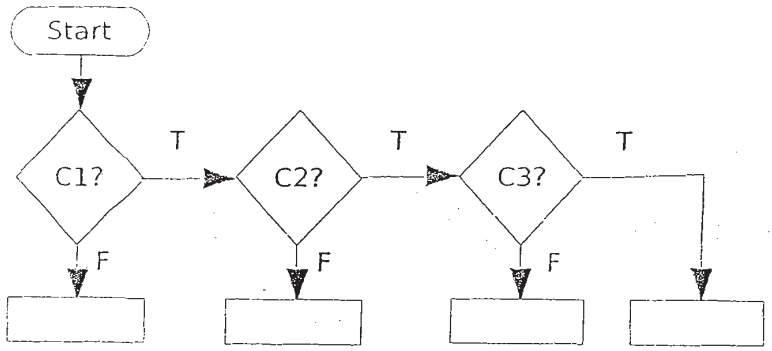
in above program, if lower case is entered, other conditions are not checked. If upper case is entered then only first two conditions are checked, others are skipped.

Nested if :

syntax :

```
if(condition)
{
     if(condition)
     {
          if(condition)
          {
               statement;
          }
     }
}
```

flow chart :



<Prog> Write a program to input integer data, test bit value of given position.

```
#include<stdio.h>
main()
{
        int data,bit;
        printf("Enter data:");
        scanf("%d",&data);
INPUT:  printf("Enter bit:");
        scanf("%d",&bit);

        if((bit>=0)&&(bit<=31))
        {
                if((data>>bit)&1)
                        printf("set\n");
                else
                        printf("clear\n");
        }
        else
        {
                printf("Invalid bit, Enter again\n");
                goto INPUT;
        }
}
```

`if(((data>>bit)&1)==1)` also valid in above program in place of `if((data>>bit)&1)` . But `if(((1<<bit)&data)==1)` will not work because answer of expression will be 32 in case of data=50 and bit=5. So answer should be set but as (32==1) is false, answer will be clear.

`if((1<<bit)&data)` will work because expression will give non-zero answer but it will be considered as 1, because if only accepts boolean conditions.

goto :

syntax :

```
main()
{
      statement;
      :
      :
      goto END;
      :
      :
END: ;
}
```

goto is unconditional jump. Use of goto is limited to the function only in which it is used. Multiple functional goto jump is invalid.

Labels should be used with statement in program. If label is used at the end of

the program, as shown in above syntax, then a blank statement (only semicolon ;) must be there after label.

<Prog> print binary of an integer value entered.

```
#include<stdio.h>
main( )
{
        int data,bit;
        printf("Enter data:");
        scanf("%d",&data);
        bit=31;
TEST:   if((data>>bit)&1)
                printf("1");
        else
                printf("0");
        bit--;
        if(bit>=0)
                goto TEST;
        printf("\n");
}
```

```
o/p > Enter data:10
      00000000000000000000000000001010
```

<Assignment> Write a program to calculate percentage scored by a student whose 5 subjects marks are entered by user. Find the gradation. (using 3 methods - if, else if ladder, nested if)

```
#include<stdio.h>
main()
{
        int a,b,c,d,e;
        float p;
        printf("Enter 5 subject marks:");
        scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
        p=((a+b+c+d+e)/5);
        printf("scored percentage : %f\n",p);
        printf("earned grade : ");

                        /* using if */

        if((p>=0)&&(p<40))
                printf("fail\n");
        if((p>=40)&&(p<50))
                printf("3rd class\n");
        if((p>=50)&&(p<60))
                printf("2nd class\n");
        if((p>=60)&&(p<75))
                printf("1st class\n");
```

```
 if((p>=75)&&(p<=100))
        printf("distinction\n");

            /* using else if ladder */

if((p>=0)&&(p<40))
        printf("fail\n");
 else if((p>=40)&&(p<50))
        printf("3rd class\n");
 else if((p>=50)&&(p<60))
        printf("2nd class\n");
 else if((p>=60)&&(p<75))
        printf("1st class\n");
 else
        printf("distinction\n");

              /* using nested if */

 if(p>40)
 {
      if(p>50)
      {
           if(p>60)
           {
                if(p>75)
                     printf("distinction\n");
                else
                     printf("1st class\n");
           }
           else
                printf("2nd class\n");
       }
       else
            printf("3rd class\n");
 }
 else
      printf("fail\n");

}
```

```
o/p > Enter 5 subject marks:78 67 75 56 69
      scored percentage : 69.000000
      earned grade : 1st class
```

<Assignment> write a program to print all the characters and their respective ASCII equivalent (without using looping statements)

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

```
{
        int data=0;
        printf("Character\t ASCII\n");
LOOP:   printf("%c\t %d\n",data,data);
        data++;
        if(data<128)
                goto LOOP;
}
```

<Assignment> Write a program to print characters, its ASCII value and its binary value. (without using looping statements)

```
#include<stdio.h>
main()
{
        int data=0,bit;
        printf("Character\t ASCII\t Binary\n");
LOOP:   printf("%c\t %d\t ",data,data);
        bit=7;
BIN:    if((data>>bit)&1)
                printf("1");
        else
                printf("0");
        bit--;
        if(bit>=0)
                goto BIN;
        printf("\n");
        data++;
        if(data<128)
                goto LOOP;
}
```

Switch :

```
syntax :    switch(expression)
            {
                 case 1:   statement;
                           break;
                 case 2:   statement;
                           break;
                 :
                 :
                 default : statement;
            }
```

switch expression can be : constant, variable, expression.

--> Switch expression should be integer expression yielding integer or character only. In case values real values are not allowed. Case values should be constant. Constant expression can be used as case value but variable can not be used to represent case value.

--> Writing default is not mandatory. If any of case doesn't match with value of expression in switch, then default statements will be executed.

--> We can write default at the top of the switch block, but then we will have to use break at the end of default statements

--> writing break is not mandatory in case statements. It is not syntax error but if no break is written then all case statements following the matched case will be executed.

--> Writing curly braces in case statements are not necessary.

--> Switch, case and default are keywords only used in switch.

--> Goto is allowed in switch. But it should not be used with case labels. We can not make jump to any case using goto statement.

<Ex>

```
int v=65;
switch(v)
{
     case 'A': printf("v is A\n");
               break;
     case 65 : printf("v is 65\n);
               break;
}
```

```
o/p > test.c: In function 'main':
      test.c:9:3: error: duplicate case value
         case 65 : printf("you are in case 65\n");
         ^
      test.c:7:3: error: previously used here
         case 'A' : printf("you are in case A\n");
         ^
```

syntax error because both cases are same. 'A' is represented by its ASCII value 65.

<Prog> gradation using switch case

```
#include<stdio.h>
main()
{
     int p;
     printf("enter percentage:");
     scanf("%d",&p);
```

```
        switch(p)
        {
              case 1:
              case 2:
              case 3:
              :
              :
              :
              case 40: printf("fail");      break;
              case 41:
              :
              :
              case 50: printf("3rd class");      break;
              :
              :
        }
}
```

first of all we can not take percentage as real data. Only integer values are allowed as percentage. Another problem is that we will have to write cases for all possible percentage values as shown above which is tiresome. In such cases else if ladder is more preferable

--> when no statements are written for the case, its next case statements are executed. So for the case 1 to 40, case 40 statements will be executed.

Switch execution or why switch?

break

break is an unconditional control statement used inside loops or switch. It passes the control to the end of the block.

<Ex>

```
main()
{
      int v;
      scanf("%d",&v);
      if(v<0)
            break;
      printf("v=%d",v);
}
```

```
test.c: In function 'main':
test.c:7:3: error: break statement not within loop or switch
   break;
   ^
```

here error is shown because break is not used within any loop or switch.

<Prog> set a particular bit in given data.

```
#include<stdio.h>
main()
{
        int data,bit;
        printf("enter data & bit:");
        scanf("%d %d",&data,&bit);
        data=data|(1<<bit);
        printf("data=%d\n",data);
}
```

```
o/p > enter data & bit:10 2
      data=14
```

```
10          00001010
1<<2        00000100
20|(1<<2)   00001110   (2nd bit set in data 10)

25          00011001
1<<3        00001000
20|(1<<3)   00011001   (data remains as it is because bit is already set)
```

<Prog> clear a particular bit in given data

```
data=data&~(1<<bit);
```

```
o/p > enter data & bit:25 3
      data=17
```

```
25           00011001
-(1<<3)      11110111
20&~(1<<3)   00010001   (3rd bit cleared in data 25)
```

<Prog> complement a particular bit in given data

```
lata=data^(1<<bit);
```

```
/p > enter data & bit:20 3
     data=28
```

```
0          00010100
<<3        00001000
0^(1<<3)   00011100   (3rd bit complemented in data 20)
```

Prog> convert lower case to upper case and vice versa.

```
include<stdio.h>
ain()
```

```
{
        char ch;
        printf("enter an alpha bate:");
        scanf("%c",&ch);
        if((ch>='A')&&(ch<='Z'))    | in place of if else block,
                ch+=32;             | ch^=(1<<5); or
        else                        | ch^=32;
                ch-=32;             | could be used
        printf("ch=%c\n",ch);
}

o/p >  enter an alpha bate:A
       ch=a

       enter an alpha bate:a
       ch=A
```

<Assignment> using switch case, write a menu driven program which displays given menu after supplying integer data and a valid bit position. After selecting menu action it should do that operation and data and again display menu.

Menu :
t/T : test bit
s/S : set bit
c/C : clear bit
m/M : compliment bit
i/I : input again
e/E : exit
enter your choice :

```
#include<stdio.h>
main()
{
        int data,bit,result;
        char input;
INPUT:  printf("Enter Data:");
        scanf("%d",&data);
BIT:    printf("Enter Bit:");
        scanf("%d",&bit);
        if(!((bit>=0)&&(bit<=31)))
        {
                printf("Enter valid bit position\n");
                goto BIT;
        }
LOOP:   printf("\nMENU\n");
        printf("t/T\t Test bit\n");
        printf("s/S\t Set bit\n");
        printf("c/C\t Clear bit\n");
        printf("m/M\t Complement bit\n");
        printf("i/I\t Input again\n");
```

```
        printf("e/E\t Exit\n");
        printf("Enter your choice:\n");
        scanf(" %c",&input);
        switch(input)
        {
                case 't':
                case 'T': result=((data>>bit)&1);
                          printf("result=%d\n",result);
                          goto LOOP;
                case 's':
                case 'S': result=(data|(1<<bit));
                          printf("result=%d\n",result);
                          goto LOOP;
                case 'c':
                case 'C': result=(data&~(1<<bit));
                          printf("result=%d\n",result);
                          goto LOOP;
                case 'm':
                case 'M': result=(data^(1<<bit));
                          printf("result=%d\n",result);
                          goto LOOP;
                case 'i':
                case 'I': goto INPUT;

                case 'e':
                case 'E': return;
        }
}
```

--> if you observe, in above program `scanf(" %c",&input);` is used in place of `scanf("%c",&input);` there doesn't seem difference between these two sentence but `scanf(" %c",&input);` statement is used due to a reason.

--> when we scan a data using scanf function, while executing, it waits for user to enter data until we hit return (↵),once return is hit, data is supplied to variable.

--> Scanf and getchar function's input behavior is called canonical mode, in which scanf or getchar doesn't automatically accept data until we hit return. This kind of behavior is there because it assumes that after typing the data, user might want to edit that data again. So until we hit enter, data is not supplied to variable.

--> But there is a problem with this mechanism, whichever data we enter is present in input buffer, from where scanf or getchar scans appropriate data according to format specifier used. Hitting return (↵) is considered as if we supplied character '\n' in input buffer.

--> when we use scanf with %d, '\n' present in input buffer doesn't create problem because scanf with %d only accepts integer values and not character but when we use scanf with %c or getchar to scanf a character, then '\n'

present in input buffer creates a problem.

--> consider above assignment,
`scanf ("%d",&data);` when we supply data, again bit variable is scanned using scanf with %d, `scanf("%d",&bit);` so in scanning of bit variable '\n' doesn't create a problem.
But while entering bit, suppose we want to supply 3 to bit. So we will input 3 and hit enter, then in input buffer '3' and '\n' both will be present. '3' will be scanned by scanf with %d and supplied to bit. But \'n' will be still present in input buffer.

--> now if we use a scanf with %c or getchar to scan a character, it will scan '\n' character from input buffer and assign it to the variable for which we were scanning. If we print that variable using %c, then a new line will be printed. If we print that variable using %d, then ASCII value of '\n' which is 10 will be displayed.

Remedy :

--> so to absorb useless '\n' present in input buffer there are several methods.
- use a dummy getchar before scanning character
- use scanf statements like shown below

```
scanf(" %c",&input);
scanf("\n%c",&input);
scanf("\t%c",&input);
```

- use fflush() function anywhere in program where we want to clear input buffer, simply write `fflush();` to flush input buffer.

scanf() & getchar() behavior

if 3 successive getchar() or scanf with %c are present in program, and we enter bcd and hit enter to first getchar or scanf, then 'a','b' & 'c' will be taken from nput buffer by 3 getchar. And 'd' and '\n' will still be present in input buffer.

f we input 12 34 56 to scanf with %d, then only 12 will be accepted by scanf. 34 and 56 will stay in input buffer and if next scanf is present, it will scan 34 rom input buffer.

NOTE : scanf only accepts string. When with scanf with %d, if we provide 11, hen it is not integer 11 but string "11" will be supplied to scanf, scanf will hen convert string "11" to integer 11 and supply it to variable. Simlarly printf nly prints string, data will be converted into string according to format pecifier and then printed on screen.

LOOP

While

syntax :
```
while(boolean condition)
{
    statement;
}
```

when curly braces are not given, next statement after while will be considered loop body.

```
while( 1)   // infinite loop
{
    statement;
}
```

in while if we leave parentheses empty, its syntax error
```
while( )    //  syntax error
```

```
while(a);       // if a nonzero, then infinite loop
{
}
```

<Ex> when this loop will terminate?
```
#include<stdio.h>
main()
{
        short int v=1;
        while(v)
        {
                printf("in while %d\n",v);
                v++;
        }
        printf("after while %d\n",v);
}
```

first of all v will be increased to maximum positive value of short integer which is 32767, then it will be -32768 and start increasing. When it will become 0, loop will terminate.

<Ex> will ch be 128 ever in below example? When will loop terminate?
```
#include<stdio.h>
main()
{
        char ch=0;
        while(ch<=127)
        {
                printf("in while %d\n",ch);
                ch++;
```

```
        }
}
```

this will be infinite loop, because once ch value is increased up to 127, then incrementing ch further it will become -128. now -128<=127 is true because integer constants are signed type by default. So from -128 it will become 0 and then 1 27. thus loop will never terminated and ch value will never become 128.
remedy : we should use unsigned char in this kind of situation.

<Prog>

```
#include<stdio.h>
main()
{
        int a=5;
        while(a--)
                printf("in loop: a=%d\n",a);
        printf("after loop: a=%d\n",a);
}
```

```
o/p >  in loop: a=4
       in loop: a=3
       in loop: a=2
       in loop: a=1
       in loop: a=0
       after loop: a=-1
```

a	printf	
5	4	|
4	3	|
3	2	| in loop
2	1	|
1	0	|
0	-1	| after loop

<Prog>

```
#include<stdio.h>
main()
{
        int a=5,b=-3;
        while(a-- && b++)
                printf("in loop: a=%d b=%d\n",a,b);
        printf("after loop: a=%d b=%d\n",a,b);
}
```

```
o/p > in loop: a=4 b=-2
      in loop: a=3 b=-1
      in loop: a=2 b=0
      after loop: a=1 b=1
```

a	b	pf(a)	pf(b)	
5	-3	4	-2	|
4	-2	3	-1	| in loop
3	-1	2	0	|
2	0	1	1	| after loop

<Prog>

```
#include<stdio.h>
main()
{
        int a=-2,b=-4;
        while(++a && --b)
                printf("in loop: a=%d b=%d\n",a,b);
        printf("after loop: a=%d b=%d\n",a,b);
}
```

o/p > in loop: a=-1 b=-5
after loop: a=0 b=-5

a	b	pf(a)	pf(b)	
-1	3	-1	3	in loop
0		0	3	after loop

↓

Optimization will take place

⟶ when multiple statements are grouped using comma (,) , then group executes from left to right but group's result will be right most statement's result.

<Ex>

```
int a=2,b=4,c=0;
c=a,b=c;
```

⟶ c=a,b=c; valid, b and c value will be 0 after this statement. In the right hand side, two statements are grouped using comma. So execution will be from left to right. So first a will be executed, but only a is of no meaning, so then b=c will be executed, and then right most result, b is assigned to c(c=b)

<Ex>

```
int a=2,b=4,c=0;
c=(a,b=0);
```

⟶ c=(a,b=0); c will be 0

<Ex>

```
int a=2,b=4,c=0;
c=(a++,b=a+5,a+b);
```

⟶ c=(a++,b=a+5,a+b); first a++, so a will be 3, then b=a+5, so b will be 8 and a+b will be 11 which will be stored in c. So c will be 11.

<Ex>

```
int a=-2,b=4;
while(0,a>b,++a,--b)
{
     statement;
}
```

here loop will execute until b becomes 0, because group will be executed from left to right but right most statement will be group's result. If in above example semicolon is present in place of comma, it will be syntax error.

<Prog>

```
#include<stdio.h>
main()
{
        int a=1,b=1;
        while(a<=5)
        {
```

```
                printf("a=%d ",a);
                while(b<=5)
                {
                        printf("b=%d ",b);
                        b++;
                }
                printf("\n");
                a++;
        }
        printf("after loop: a=%d b=%d\n",a,b);
}
```

```
o/p >  a=1 b=1 b=2 b=3 b=4 b=5
       a=2
       a=3
       a=4
       a=5
       after loop: a=6 b=6
```

if we initialize inner loop's control variable b every time before inner loop, then it will execute every time otherwise it will execute only once.

<Prog>

```
#include<stdio.h>
main()
{
        int a=1,b=1;
        while(a<=5)
        {
                b=1;
                while(b<=5)
                {
                        printf("%3d",a*b);
                        b++;
                }
                printf("\n");
                a++;
        }
}
```

```
o/p >    1  2  3  4  5
         2  4  6  8 10
         3  6  9 12 15
         4  8 12 16 20
         5 10 15 20 25
```

`printf("%3d",a*b);` %3d used to make proper alignment in printing, it will allocate 3 character's space for one integer to be printed on screen.

<Assignment> print all characters, its ASCII values and its binary values using while loop.

```
#include<stdio.h>
main( )
{
        int data,bit;
        printf("symbol\tASCII\tbinary\n");
        data=0;
        while(data<=127)
        {
                printf("%c\t%d\t",data,data);
                bit=7;
                while(bit>=0)
                {
                        printf("%d",(data>>bit)&1);
                        bit--;
                }
                printf("\n");
                data++;
        }
}
```

For

syntax : `for( initialization ; test ; operation on )`
`of control variable` `condition` `control variable`

```
for(i=0;i<=5;i++)
    ①  ②   ③
{
    printf("%d",i);④
}
```

--> order of execution : ①→②→④→③

looping statements

above for loop is equivalent to,

```
① i=1;
② for(;i<=5;)
  {
④   printf("%d",i);
③   i++;
  }
```

```
for(i<=5)        //  syntax error
```
in for statement, two semicolons (;) are mandatory

```
for(;; )         // infinite loop
{
     statement;
}

for(;i ;)        // infinite loop
{
     statement;
}
```

<Prog>

```
#include<stdio.h>
main()
{
        int a,b;
        for(a=1;a<=5;a++)
             for(b=1;b<=10;b++)
                  printf("%3d",a*b);
}
```

--> if in condition statement of for loop, many statements are written separated by comma (,) then right most will be considered.

<Prog>

```
#include<stdio.h>
main()
{
        int a,b;
        for(a=1;a<=5;a++,printf("\n"))
                for(b=1;b<=5;b++)
                        printf("%d ",a);
}
```

```
o/p > 1  1  1  1  1
      2  2  2  2  2
      3  3  3  3  3
      4  4  4  4  4
      5  5  5  5  5
```

<Prog>

```
#include<stdio.h>
main()
{
        int a,b;
        for(a=1;a<=5;a++,printf("\n"))
```

```
                for(b=1;b<=a;b++)
                        printf("%d ",a);
}
```

```
o/p >  1
       2 2
       3 3 3
       4 4 4 4
       5 5 5 5 5
```

if in above program b variable is printed instead of a

```
o/p >  1
       1 2
       1 2 3
       1 2 3 4
       1 2 3 4 5
```

if in above program instead of `printf("%d",a);` statement, `printf("* ");` is used then

```
o/p >  *
       * *
       * * *
       * * * *
       * * * * *
```

<Prog>

```
#include<stdio.h>
main()
{
        int a,b;
        for(a=1;a<=5;a++,printf("\n"))
        {
                for(b=1;b<=5-a;b++)
                        printf(" ");
                for(b=1;b<=a;b++)
                        printf("* ");
        }
}
```

```
o/p >      *
          * *
         * * *
        * * * *
       * * * * *
```

above program can be made compact and easy by using conditional operator.

```
#include<stdio.h>
main()
{
        int a,b;
        for(a=1;a<=5;a++,printf("\n"))
                for(b=5;b>=1;b--)
                        (b>a)?printf(" "):printf("* ");
}
```

NOTE : in pattern printing programs like shown above, remember few things to decrease complexity of program.

First of all : use one variable (e.g. a) for rows
use one variable (e.g. b) for columns

Determine number of rows first, like in above example it is 5. always start variable for rows (a) with 1 and keep on increasing it.

If you want to leave more spaces in first row and then in next rows, number of spaces will be decreased (like in above example). Start b with 5 and keep decreasing it. If number of spaces in each row has to be increased row by row, then start b with 1 and keep inceasing it.

Having done this you have to only think about condition for particular pattern

<Assignment> write a program to reverse bits of a given integer number.

```
include<stdio.h>
main()
{
    int data,bit,i,j;
    printf("Enter data:");
    scanf("%d",&data);

    //printf binary of original data
    for(bit=31;bit>=0;bit--)
        printf("%d",(data>>bit)&1);
    printf("\n");

    //reverse bits
    for(i=31,j=0;i>j;i--,j++)
    {
        if(((data>>i)&1) != ((data>>j)&1))
        {
            data^=(1<<i);
            data^=(1<<j);
        }
    }
```

```
        //print binary of modified data
        for(bit=31;bit>=0;bit--)
             printf("%d",(data>>bit)&1);
        printf("\n");
}
```

```
o/p >  Enter data:10
       00000000000000000000000000001010
       01010000000000000000000000000000
```

in bit reverse bits should be swapped like shown below

```
7 - 0 |  --> swapping pairs.
6 - 1 |  (check if ith and jth bits are not equal, if not then complement both,
5 - 2 |  otherwise do nothing)
4 - 3 |  --> testing condition : (i>j)
 i   j
--  ++
```

<Assignment> count digits of an integer number

```
#include<stdio.h>
main()
{
     int data,i,count=0;
     printf("Enter data:");
     scanf("%d",&data);
     if(data==0)
          printf("count=1\n");
     for(i=data;i;i=i/10)
          count++;
     printf("count=%d\n",count);
}
```

```
o/p > Enter data:123
      count=3
```

<Assignment> write a program to calculate sum of digits in a given integer

```
#include<stdio.h>
main()
{
     int data,i,sum=0;
     printf("Enter data:");
     scanf("%d",&data);
     for(i=data;i;i=i/10)
          sum+=i%10;
     printf("sum=%d\n",sum);
}
```

```
o/p > Enter data:123
      sum=6
```

<Assignment> reverse digits of given integer

```
#include<stdio.h>
main( )
{
      int data,i,sum=0;
      printf("Enter data:");
      scanf("%d",&data);
      for(i=data;i;i=i/10)
            sum=sum*10 + i%10;
      printf("Reversed number=%d\n",sum);
}
```

```
o/p >  Enter data:123
       Reversed number=321
```

<Assignment> find factorial of given integer

```
#include<stdio.h>
main()
{
      int data,i,fact=1;
      printf("Enter data:");
      scanf("%d",&data);
      for(i=data;i;i--)
            fact*=i;
      printf("Factorial=%d\n",fact);
}
```

```
o/p > Enter data:5
      Factorial=120
```

<Assignment> write a program to print Fibonacci series up to given max value

```
#include<stdio.h>
main()
{
      int a=0,b=1,c,max;
      printf("Enter max value:");
      scanf("%d",&max);
      printf("0, 1, ");
      for(c=a+b;(c=a+b)<=max;a=b,b=c)
            printf("%d, ",c);
}
```

```
o/p > Enter max value:25
      0, 1, 1, 2, 3, 5, 8, 13, 21,
```

<Assignment> input two integers x and y, calculate x^y

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

```
        int x,y,result,i;
        printf("input two integers x and y:");
        scanf("%d %d",&x,&y);
        for(i=y,result=1;i;i--)
             result=result*x;
        printf("%d^%d is %d\n",x,y,result);
}

o/p >  input two integers x and y:2 3
       2^3 is 8
```

<Assignment> input two integers x and y, and check if x is power of y or not.

```
#include<stdio.h>
main()
{
     int x,y,flag,count=0,temp;
     printf("Enter two integers x and y:");
     scanf("%d %d",&x,&y);
     if(y==x)
     {
          flag=1;
          goto END;
     }
     for(temp=x;temp>1;temp=temp/y)
     {
          if(temp%y==0)
          {
               count++;
               continue;
          }
          else
          {
               flag=1;
               break;
          }
     }
END: if(flag==1)
     {
          if(x==y)
               printf("%d is 1 power of %d\n",x,y);
          else
               printf("%d is not power of %d\n",x,y);
     }
     else
          printf("%d is %d power of %d\n",x,count,y);
}

o/p > Enter two integers x and y:8 2
      8 is 3 power of 2
```

<Assignment> input an integer and test if it is prime or not

```
#include<stdio.h>
#include<math.h>
main( )
{
      int num,s,i;
      printf("Enter num:");
      scanf("%d",&num);
      s=sqrt(num);
      for(i=2;i<=s;i++)
      {
            if((num%i)==0)
                 break;
      }
      if(i==s+1)
            printf("%d is prime\n",num);
      else
            printf("%d is not prime\n",num);
}
```

```
o/p > Enter num:13
       13 is prime
```

if we try to compile above code using cc prime.c it gives linking error shown below

```
/tmp/cc4DDjtM.o: In function `main':
prime.c:(.text+0x35): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

this is because math.h functions are present in different library

libc.so --> stdio.h functions present
libm.so --> math.h functions present

compiling above program with `cc prime.c` command, compiler searches for called functions in libc.so by default and links them. But sqrt function is present in different place so we have to tell compiler to search for libm.so library and link sqrt function. For that we have to use command

```
cc test.c -lm
```

similarly if function is present in different library named libname.so, then we will have to compile the source code using command

```
cc prime.c -lname
```

- l name
↙ ↘
lib name.so

<Assignment> print all the prime numbers within given range.

```
#include<stdio.h>
#include<math.h>
main()
{
     int num,s,i,max,min;
     printf("Enter range : min and max values");
     scanf("%d %d",&min,&max);
     for(num=min;num<=max;num++)
     {
          s=sqrt(num);
          for(i=2;i<=s;i++)
          {
               if((num%i)==0)
                    break;
          }
          if(i==s+1)
               printf("%d, ",num);
     }
}
```

```
o/p > Enter range : min and max values0 50
       1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
```

<Assignment> write a program to print multiplication table in given range.

```
#include<stdio.h>
main()
{
        int min,max,sum=0,temp,i,j;
        printf("Enter min and max values:");
        scanf("%d %d",&min,&max);

 // count digits of max num for proper alignment in printing
        for(temp=max*10;temp;temp=temp/10)
                sum++;

        for(i=min;i<=max;i++)
        {
                printf("%*d : ",sum-1,i);
                for(j=1;j<=10;j++)
                        printf("%*d ",sum,i*j);
                printf("\n");
        }
}
```

```
o/p > Enter min and max values:10 12
      10 :  10  20  30  40  50  60  70  80  90 100
      11 :  11  22  33  44  55  66  77  88  99 110
      12 :  12  24  36  48  60  72  84  96 108 120
```

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=1;j<=5;j++)
                        (j<i)?printf(" "):printf("* ");
}
```

o/p >

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

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=1;j<=5;j++)
                        (j<i)?printf("  "):printf("* ");
}
```

o/p >

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

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j,flag=1;
        for(i=1;i<=5;i++,printf("\n"))
        {
                if(i%2==0)
                        flag=0;
                else
                        flag=1;
                for(j=1;j<=i;j++)
                {
                        printf("%d ",flag);
                        flag=!flag;
                }
        }
}
```

o/p > 1
0 1
1 0 1
0 1 0 1
1 0 1 0 1

<Assignment>

```
#include<stdio.h>
main( )
{
        int i,j,flag=65;
        for(i=1;i<=5;i++,printf("\n"))
        {
                for(j=5,flag=65;j>=i;j--)
                {
                        printf("%c ",flag);
                        flag++;
                }
        }
}
```

o/p > A B C D E
A B C D
A B C
A B
A

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=1;i<=5;i++,printf("\n") )
                for(j=5;j>=1;j--)
                        (j>i)?printf(" "):printf("* ");
        for(i=1;i<=4;i++,printf("\n"))
                for(j=1;j<=5;j++)
                        (j<=i)?printf(" "):printf("* ");
}
```

o/p >

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

<Assignment>

```
#include<stdio.h>
main( )
{
        int i,j,k;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=5;j>=1;j--)
                        (j>i)?printf("  "):printf("* ");
        for(i=1;i<=4;i++,printf("\n"))
                for(j=1;j<=5;j++)
                        (j<=i)?printf("  "):printf("* ");
}
```

o/p >

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

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=1;j<=i;j++)
                        printf("* ");
        for(i=1;i<=4;i++,printf("\n"))
                for(j=5;j>i;j--)
                        printf("* ");
}
```

o/p >

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

<Assignment>

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

```
        int i,j;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=1;j<=5;j++)
                        (j<i)?printf(" "):printf("* ");
        for(i=1;i<=4;i++,printf("\n"))
                for(j=5;j>=1;j--)
                        (j>i+1)?printf(" "):printf("* ");
}
```

```
o/p >  * * * * *
        * * * *
         * * *
          * *
           *
          * *
         * * *
        * * * *
       * * * * *
```

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=1;j<=5;j++)
                        (j<i)?printf("   "):printf("* ");
        for(i=1;i<=4;i++,printf("\n"))
                for(j=5;j>=1;j--)
                        (j>i+1)?printf("   "):printf("* ");
}
```

```
o/p >  * * * * *
        * * * *
          * * *
            * *
              *
            * *
          * * *
        * * * *
      * * * * *
```

<Assignment>

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=1;i<=5;i++,printf("\n"))
                for(j=5;j>=i;j--)
                        printf("* ");
```

```
        for(i=1;i<=4;i++,printf("\n"))
                for(j=1;j<=i+1;j++)
                        printf("* ");
}
```

```
o/p >  * * * * *
       * * * *
       * * *
       * *
       *
       * *
       * * *
       * * * *
       * * * * *
```

do while

```
yntax :     do
            {
                 statement;
            }while(condition);       // ; is necessary
```

r loop's body contains only one statement, then writing curly braces is not mandatory

```
do

    printf("in loop");
hile(1);                // infinite loop

)

    printf("in loop");
hile(0);                // will execute at least once
```

while	do while
> while is entry controlled loop (in hich condition is checked first and en loop body is executed according that condition)	--> do while is exit controlled loop (in which loop body is executed first and then condition is checked for further iterations
> minimum number of times while op will execute is 0	--> minimum number of times do while will execute is 1

If numbers from 1 to 5 are to be printed,

```
using  do while
int i=1;
do
{
     printf("%d ",i);
     i++;
}while(i<=5);

using  while
int i=1;
while(i<=5)
{
     printf("%d ",i);
     i++;
}

similar structure like do while using while loop
int i=1;
while(1)
{
     printf("%d ",i);
     i++;
     if(i>5)
          break;
}
```

--> do while loop will be helpful when loop body should be executed at least once irrespective of its condition

for example, in counting of digits, if 0 is supplied, number of digits should be 1

```
using do while loop
int num,cnt=0;      // works for number 0 also
do
{
     cnt++;
     num=num/10;
}while(num);

using while loop
int num,cnt=0;      // not applicable if number is 0
while(num)
{
     num=num/10;
     cnt++;
}
```

continue

⟶ unconditional control statement

⟶ can only be used inside loop

⟶ if executed, passes the control to the starting of the loop by skipping the statements below it

usage :

```
for(i=1;i<=20;i++)
{
      if(i%3==0)
            continue;
      printf("%d ",i);
}
```

o/p > 1 2 4 5 7 8 10 11 13 14 16 17 19 20

In this loop, when continue is executed, statements below the body are skipped, but i++ is executed after continue and then test condition is checked and body is executed according to it.

--> if in while loop, operation on control variable is after continue statement, then loop enters in infinite loop.

```
i=1;
while(i<=20)
{
      if(i%3==0)
            continue;       // will enter in infinite loop when i=3
      printf("%d ",i);
      i++;
}
```

-> in do while continue statement passes the control to while statement

```
i=1;
do
{
      if(i%3==0)
            continue;       // will enter in infinite loop when i=3
      printf(%d ",i);
      i++;
 while(i<=20);
```

Functions

Benefits of using functions:

- Easy debugging
- Re usability of code
- Readability
- Easy understanding of code

<Prog>

```
#include<stdio.h>
#include<math.h>
void testPrime(void)          //testPrime > called function
{
     int v,s,i;
     printf("Enter integer:");
     scanf("%d",&v);
     s=sqrt(v);
     for(i=2;i<=s;i++)
     {
          if(v%i==0)
              break;
     }
     if(i==(s+1))
          printf("prime");
     else
          printf("not prime");
}
main()                        //main > calling function
{
     testPrime();
}
```

NOTE : If no return type is specified in a function, default return type int will be used.

<Prog>

```
#include<stdio.h>
#include<math.h>
void testPrime(int v)
{
     int s,i;
     s=sqrt(v);
     for(i=2;i<=s;i++)
     {
          if(v%i==0)
              break;
     }
```

```
        if(i==(s+1))
                printf("prime");
        else
                printf("not prime");
}
main()
{
        int v;
        printf("Enter integer");
        scanf("%d",&v);
        testPrime(v);
}
```

here, **v** is local variable in main, so it is unknown to function testPrime. However if we want variable v's value to be used in testPrime function we have to pass it. This is called data passing concept.

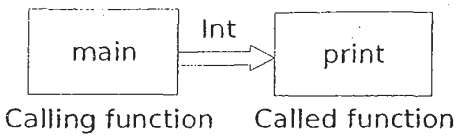
Function data passing:

<Prog>

```
main()
{
        int a;
        scanf("%d",&a);
        print();
}
print()
{
        printf("%d",a);
}
```

```
O/P :-- Syntax error because variable a is not declared in print()
function.
```

Function data passing : Local data of calling function is not visible to called function. So we have to pass data to called function.



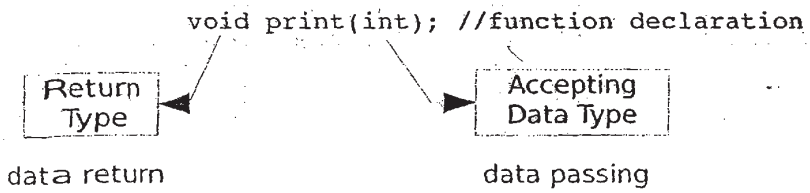
To pass the data;

```
printf(a); //here a – actual argument
```

passed variable should be received in called function, so one variable variable is declared in called function to receive passed data by calling function,

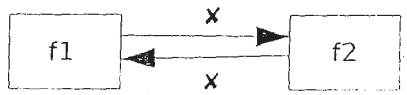
```
print(int a) //here a – formal argument
{
      –
      –
}
```

In above program, copy of a is passed to print(). It is called call by value method of data passing. Here variable a in main() and print() are different, addresses of variable a in both functions are also different.

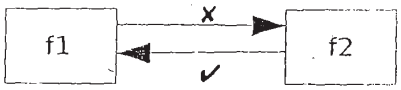



```
print(a);         //function call

void print(int a) //Function definition
{
      –
      –
}
```

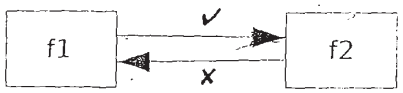
NOTE : Declarations never execute, the are statements for the copiler to know about the type of functions or variables



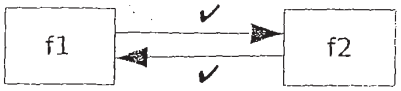
void function_name(void);



type function_name(void);



void function_name(type);



type function_name(type);

<Prog> : Print prime numbers between given range using function.

```
#include<stdio.h>
#include<math.h>
int testPrime(int);      //function declaration
```

```
main( )
{
      int v,min,max;
      printf("enter range");
      scanf("%d %d",&min,&max);
      for(v=min;v<=max;v++)
      {
           if(testPrime(v)==1)
                printf("%d, ",v);
      }
}
int testPrime(int v);
{
     int s,flag=0;
     s=sqrt(v);
     for(i=2;i<=s;i++)
     {
          if(v%i==0)
               break;
     }
     if(i==(s+1))
          flag=1;
     return flag;
}
```

NOTE : sqrt() accepts double data and returns double data

<Prog>

```
#include<stdio.h>
main()
{
     int a=5;
     print(++a);    // value 6 is passed
}
print(int a)
{
     printf("%d",a); // prints 6
}
```

o/p >> 6

<Prog>

```
#include<stdio.h>
main()
{
     int a=5;
     print(a);
     printf(%d\t",a); //prints 5
}
```

```
print (int a)
{
     printf("%d\t",++a); //prints 6
}

o/p >> 6  5
```

<Prog>

```
#include<stdio.h>
main()
{
     int a=5;
     print(a++);          //5 is passed to print()
     printf("%d\t",a);    //prints 6
}
print( int a)
{
     printf("%d\t",a);    //prints 5
}

o/p >>  5   6
```

<Prog>

```
#include<stdio.h>
void print(int,int);
main()
{
     int a=10,b=20;
     print(a,b);
}
print(int x,int y)
{
     printf("%d, %d",x,y);    // prints 10, 20
}
```

```
o/p >> 10, 20
```

in above program, y receives b's copy first, and then x receives x's copy.

NOTE: Passed data is received in called function from right to left by default.

NOTE: Any number of data can be passed but only one data can be received.

NOTE: Return type of a function can be float also.

<Prog>

```
include<stdio.h>
ain()
```

```
{
      int a=10,b=20;
      print(a=b,a==b);    //(a==b)->0 will be passed first,
                          //then (a=b)->20 will be passed
}
print(int x,int y)
{
     printf("%d, %d",x,y); //prints 20, 0
}
```

o/p >> 20, 0

first of all result of expression a==b will be passed to y, which is 0; then in a=b passing, first of all b's value will be assigned to a, then a's value which is 20, will be passed to x.

<Prog>

```
#include<stdio.h>
void print(int x,int y,int z)
{
     printf("%d, %d, %d",x,y,z);
}
main()
{
     int a=10,b=20;
     print(a,a=b,a=0); //values passed should be like this
                       //first a=0 -> 0
                       //second a=b -> 20
                       //third a -> 20
}
```

o/p > 20, 20, 20

in such case, it is compiler's choice to pass which values of a. x,y and z will receive value of a. So compiler will pass final value of a in print()'s all arguments. a's final value is 20 after operation a=b; so 20 will be passed to x,y and z. This method of passing final value of variable to the arguments which are receiving same variable is called **optimization**.

NOTE : Turbo C doesn't do optimization, GCC does that.

<Prog>

```
#include<stdio.h>
void print(int x,int y,int z)
{
     printf("%d, %d, %d",x,y,z);
}
main()
```

```
{
      int a=10,b=20;
      print(a=b,a==b,a=0);      //values passed should be like this
                                //first a=0 -> 0 passed to z
                                //second a==b -> 0 passed to y
                                //third a=b -> 20 passed to x
}
```

o/p > 20, 0, 20

here x and z receives copy of same variable a. So compiler will pass latest value of a to both arguments, which is 20. so z will receive 20 in place of 0.

<Prog>

```
#include<stdio.h>
void print(int x,int y,int z)
{
      printf("%d, %d, %d",x,y,z);
}
main()
{
      int a,b;
      printf("Enter a &b:");
      scanf("%d %d",&a,&b);  //assume a=10 and b=20 entered

      print(a=b,a=0,a=b+10); //here assuming optimizing compiler,
                             //a=b+10 -> 30, a=0 -> 0, a=b -> 20;
                             //final value of a=20 should be passed
                             //to x,y and z.
}
```

o/p > 20, 0, 30

if scanf() statement is used to enter values of variables, compiler doesn't know what values will be entered by user at runtime. So compiler **will not do optimization** in programs in which **scanf** is used. Because scanf() statement requires address of variables, so when address of variable is used in program, compiler will not optimize it. If scanf is not used in above program then o/p > 20, 20, 20 because compiler will do optimization.

NOTE : scanf() doesn't allow optimization to be done by compiler

NOTE : Function declaration is not necessary when function definition is written above the call

<Prog>

```
#include<stdio.h>
void print(int x,int y,int z)
{
```

```
    printf("%d, %d, %d",x,y,z);
}
main( )
{
    volatile int a=5,b=6; //notice volatile type qualifier used
    print(a=b,a=0,a=20);
}

o/p >  6, 0, 20
```

when **volatile type qualifier** is used, then **compiler is informed not to take advantage of CPU registers** and every usage of variable will be from its actual location. Taking advantage of CPU register is one of the way of optimizing the code by compiler. In which if a variable is to be successively used for some time, instead of reading that variable from memory every time, its copy is saved in CPU register and operations on that variable is done from register. This is so because operations on register are faster that operations on memory.

```
--> Printf("%d",a); //here printf has actually two arguments, one
is                  //string "%d" and other is variable a.
```

<Ex>

```
main()
{
    volatile int a=5,b=6;
    printf("%d %d %d",a=1,a=2,a=3);
}

o/p > 1 2 3
```

<Ex>

```
main()
{
    int a=5,b=6;   //volatile not used, optimization will be done.
    printf("%d %d %d",a=1,a=2,a=3);
}
o/p > 1 1 1
```

all arguments receives variable a, so first of all after a=3, a value will be 3. similarly after a=2 and a=1; final value of a will be 1, which will be supplied to all arguments.

<Ex>

```
main()
{
    int a=5;
    printf("%d %d %d",++a,++a,++a);
}                        8   7   6      // 8 value will be passed
```

o/p > 8 8 8

<Ex>

```
main( )
{
     int a=5;
     printf("%d %d %d",a++,a++,a++);
}                              7   6   5
```

o/p > 7 6 5

<Ex>

```
main()
{
     int a=5;                          a     a  a
     printf("%d %d %d %d %d",++a,a++,a,++a,a++);
}                                    9   7  7  7   5
```

o/p > 9 7 9 9 5

in above three examples, one thing is clear that whenever post increment is present in functions arguments, it doesn't allow optimization to be done. ++a & a will optimized (i.e. wait for the latest value of a), where as in a++ a will be passed first and then incremented, so not optimized.

<Ex>

```
main()
{
     int a=1;
     printf("%d %d %d %d %d",a++,++a,++a,a++,a);
}                                4    4   3   1  1
```

o/p > 4 5 5 1 5

in above program, final value of a will be 5 after left most a++. so ++a and a will receive final copy of a which is 5. where as a++ will receive its actual copies.

NOTE: getchar() accepting type is void, but return type is integer.

<Q> What is the difference between void main() and int main()?

- main() is a user defined function, but main's specialty is main can have various type of data acceptance and data return. main() function is designed to receive command line data.

<Ex>

```
#include<stdio.h>
void main(int v)
{
     printf("v=%d\n",v);
```

```
}

o/p > v=1       // ./a.out - total one string supplied at command
                   line -> "a.out"

    > v=5       // ./a.out 11 22 33 44 - total 5 strings supplied
                   -> "a.out","11","22","33","44"

    > v=4       // ./a.out 11 22 "33 44" - total 4 strings supplied
                   -> "a.out","11","22","33 44"
```

return; -> return statement passed the control to calling function.

e.g.

```
void main()
{
     -
     -
     return;   //control return - control is returned to calling
               //function
}
```

<Prog>

```
#include<stdio.h>
int main()
{
     printf("in main\n");
     return;   //should be error, because return type is specified
               //as int, but here no data is returned from main.
               //gcc allows it and it is not error in gcc.
}
```

Will print garbage

NOTE : if main() function returns any data, it returns to _start function, which is calling function of main(). If main() returns data to _start function or not, it is useless and of practically no use, because it is not user defined function. It is created by linker. So gcc developers have developed gcc that way that, returning value to _start is not necessary.

NOTE : return data type of function can be int, char, float or double

<Q> In some programs where int main() is used, **return 0;** statement is used. Similarly to introduce exit procedure at some part of program, **exit(0);** is used. So why 0 is used with these two statements and not any other integer?

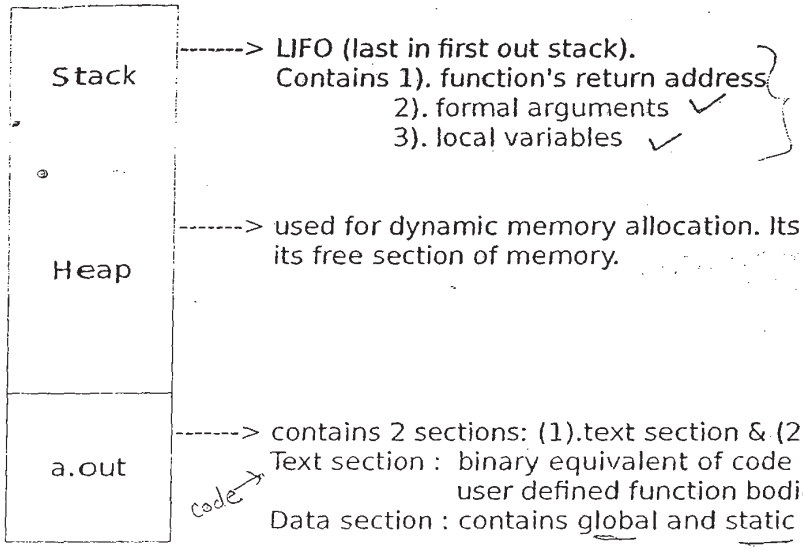
- Operating system should know that when a process terminates, it is success termination or failure termination. So for that, processes are designed to return values to OS so that it could know about the

termination.
0 -> success termination
1 -> failure termination

Returning data from main doesn't matter to us, because it returns data to _start function which is not created by user. But it matters to the process receiving it.

Memory partitions of a Process

Before execution of program starts, stack is allocated for it by operating system. Stack size depends on operating system.



Whenever a function is called, stack frame for it is created in stack section. Stack frame for a function contains local variables defined in that function, formal arguments, and function's return address.

ole of stack while using function

Prog>

```
include<stdio.h>
nt g;
oid f1(void)

    g++;
    printf("f1: g=%d at %u\n",g,&g);

ain()
```

```
{
      int v;
      printf("enter v:");
      scanf("%d",&v);
      if(v<0)
           f1();
      printf("main: v=%d at %u\n",v,&v);
      printf("main: g=%d at %u\n",g,&g);
}

o/p >  enter v:-1
       in f1: g=1 at 6295628
       in main: v=-1 at 2839265036
       in main: g=1 at 6295628
```

when compiled, gives us warning:

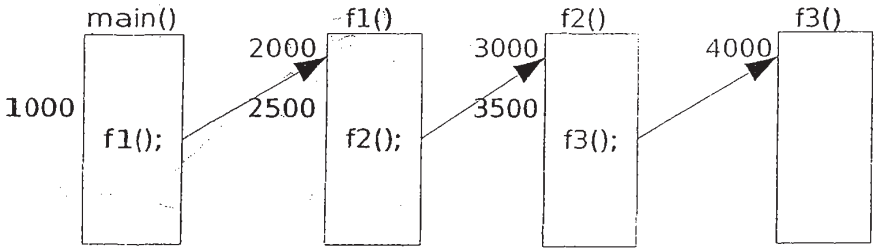
```
 warning: format '%u' expects argument of type 'unsigned int', but
argument 3 has type 'int *' [-Wformat=]
  printf("in f1: g=%d at %u\n",g,&g);
  ^
```

this is because %u specifier is used for printing address, %p should be used. But using %p address is printed in hex format, so to print address in decimal format %u is used.

-> Now in above program, when executed, main's stack is present as program execution starts from main. But main's stack is not created at the instant when program is executed. _start function is executed first and it calls main. When main is called from _start, then its stack frame is created.

-> Even if function f1()'s call is depended on value of variable v in main, it will be in life. Because function's body will be present in text section of executable file. So when program is executed, whole binary file is brought to RAM. So it will be present in memory throughout the execution of the program. f1()'s life is application's life.

>> stack frames creation

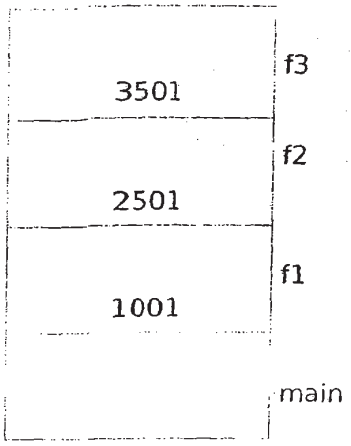


suppose we have four functions in our program. f1() is called from main(), f2() i called from f1(), f3() is called from f2().

Now if f1()'s body starts at the address 2000, then main() must know this address in order to make a jump to that location. i.e. reference of address of f1

must known to main. This is called **function mapping (linking).**

Now if in above examples, all the functions are called then its stack frames will be created like shown in figure.



When program execution starts, main's stack frame is created. Local variables, formal arguments and return address are present in stack frame. When f1() is called from main, f1's stack frame is created. And return address is pushed in stack as shown. In the same way when f2 and f3 are called, their stack frames are created. When a function collapse, its stack frame also collapses. That means, local variables present in that function, formal arguments also collapse. When function is called again, its stack frame is created again, meaning local variables and formal arguments will also be created again.

When function is called, first of all stack frame for it is created, then address of next instruction of calling function is pushed in called function's stack and jump is made to that function's location. When function collapse, program counter is made as return address of f3, and thus functions are called.

-> If there is a local variable v1 in function f1(). Then if f1() is called from main, it will be present in its stack frame. So v only comes in life when f1() is called and destroyed when f1() collapses and returns to main. If f1() is not called, v will never come in life. But f1() will be in life whether it is called or not.

-> When executable is brought to memory, f1, main, global and static variable's life starts.

-> If f1() is not called, then g value will be 0. if f1() called, g will be increased by 1, and it will be 1.

-> If one more variable v is present in f1(). It is different from main's v because for both functions different stack frames are created, so that their addresses will also be different. So both v variables are visible in its own function only.

NOTE : Data section is lower in memory, stack is higher

NOTE : Global variables are by default initialized to 0

<Q> Why scanf() used call by reference mechanism while printf() doesn't?

printf() function doesn't need to modify contents of data in order to print variable's data on screen, so call by value mechanism is used in printf() where, copy of variables are passed. There is no need to access actual data in printf().

But in case of scanf(), scanf() is capable of receiving multiple data from standard input, and storing them in multiple variable, for that scanf() needs to access original variables, which can be achieved by call be reference mechanism. If in scanf() call by value would have been used, then after modifying input data, modified data should be returned, but as return statement can return only one data, it is not possible to use call by value in scanf().

<**Prog**> How to verify stack is up going or down going?

```
#include<stdio.h>
void f2(int a)
{
     printf("f2: a=%d at %u\n",a,&a);
     a=a+10;
     printf("f2 returning: a=%d at %u\n",a,&a);
}
void f1(int a)
{
     printf("f1: a=%d at %u\n",a,&a);
     f2(a++);
     printf("f1 returning: a=%d at %u\n",a,&a);
}
main()
{
     int a=1;
     printf("main: a=%d at %u\n",a,&a);
     f1(a+1);
     printf("main returning: a=%d at %u\n",a,&a);
}
o/p > main: a=1 at 763118044
      f1: a=2 at 763118012
      f2: a=2 at 763117980
      f2 returning: a=12 at 763117980
      f1 returning: a=3 at 763118012
      main returning: a=1 at 763118044
```

>> From above outputs it is clear that **stack is down growing**. Because main's local variable a is at upper most location in stack, then f1's a is below main's a and f2'a is at lower most location.

GDB (GNU Debugger) :

To make a.out debugable, `// cc -g test.c`

to open debugable executable in debugger `// gdb a.out`

GDB Commands :

`break/b <line number or function name>` : sets breakpoint in the line where executable statement lies.

```
e.g. break 1
     break main
     b main
```

`run/r` : to start executing a.out in gdb

`next/n` : executes current line

`print/p ch` : prints value of variable ch

`backtrace/bt` : stack check

`continue/c` : if no other break points available, executes whole program

`quit/q` : quit gdb session

SIZE Command :

command : `size a.out`

size command tells us that how much RAM application will occupy when executed. A sample size command output is shown below

text	data	bss	dec	hex	filename
1564	560	8	2132	854	a.out

bss - block started by symbol (uninitialized data section), also a data section

NM Command :

command : `nm a.out / nm test.o`

nm command is used to list symbols from executables or object files

symbol list means > function name
global variable name
static variable name

sample output of nm command used on object file test.o compiled from above stack example,

```
00000000000046 T f1
00000000000000 T f2
00000000000093 T main
                 U printf
```

U	Undefined symbols	
T	Text section	
B	BSS section	// strong symbol
D	Data section	// strong symbol
c	common symbol	// weak symbol

if nm command is used on executable a.out compiled from test.c shows all the symbols which are added in executable by linker.

NOTE : If we want to see user defined symbols only it is recommended to use nm command on object files, which contains user defined symbols only.

- **Scanf(), printf()** are undefined because they are not defined in our program, they are defined in library functions.
- whatever functions are defined in our program, they are listed as defined.

NOTE : When a.out is executed, entire executable is brought to memory. main(), f1(), f2() are in life that time. When application's life begins, all these function's life begins. So these function's life is application's life. But functions like printf(), scanf() are not in life that time, because they are not present in executable but they are present in library functions and are called from program

NOTE : main()'s local variable's llife doesnt start immediately as the application is executed. First of all _start function is executed and _start makes a call to main(). When main() is called from _start, its local variable's life starts.

Pointers

Pointer : A variable which holds address.

<Prog>

```
#include<stdio.h>
main( )
{
     unsigned int a,b;
     a=10;b=20;
     printf("a=%u at %u\n",a,&a);
     printf("b=%u at %u\n",b,&b);

     a=&b;
     printf("a=%u at %u\n",a,&a);
     printf("b=%u at %u\n",b,&b);
}
```

```
o/p >  a=10 at 1131388152
       b=20 at 1131388156
       a=1131388156 at 1131388152
       b=20 at 1131388156
```

here, & is reference operator, so address of b is stored in a by statement a=&b;
so after assigning address of b into a, printing a will show address of b.

NOTE : Pointers are also called indirections.

<Prog>

```
#include<stdio.h>
main()
{
     unsigned int a=10,b=20;
     a=&b;
     *a=50;
     printf("b=%d\n",b); //Error because dereference operator * is
                         //used on non-pointer variable.
}
```

```
o/p > shows an error
     error: invalid type argument of unary '*' (have 'int')
     *a=50;
     ^
```

To declare a variable as a pointer;

```
int *a;       //declaration of a pointer; * is called declarator of pointer
```

To store address of a variable in pointer;
a=&b;

To change the value of variable of which pointer variable is pointing
*a=50;

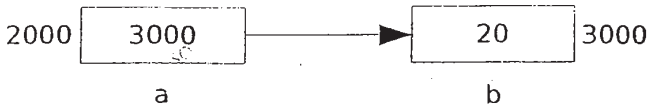
* : dereference operator //using dereference operator we get data from that address
& : reference operator //using reference operator, we put address in pointer

Let us consider an example,

<Prog>

```
#include<stdio.h>
main( )
{
     unsigned int *a,b;  //*a declares variable a as pointer
     a=10;          //gives us warning because 10 is not address
     b=20;

     a=&b;          //stores address of variable b in variable a
     *a=50;         //*a used to dereference address stored in it
     printf("b=%d\n",b);
}
```

```
warning: assignment makes pointer from integer without a cast
[enabled by default]
  a=10;
    ^
o/p > b=50
```



Reference of variable b is stored in a.
Dereference of a is b.

in above example, a is declared pointer variable by using int *a; to store another variable's address in pointer a, use statement a=&b; assume that if b's address is 3000; then a will be containing value 3000. Now to change value of b using pointer a, we have to use dereference of a using *a=50; after this statement, which ever location address pointer a is containing, its value will be changed to 50. so b will be changed to 50 from 20.

NOTE : Reference is only one byte's location. Only base (starting) address of Variable. Not all bytes address.

NOTE : printf("%d",*&a) or printf("%d",*(&a)); prints data of integer a.

NOTE : In GCC pointer variables are of 4 bytes by default. In 64 but systems Pointer variable is of 8 bytes.

<Prog>

```
#include<stdio.h>
main()
{
      int i=400;
      char c='a';
      double d=23.4;

      int *ip=&i;     //equivalent to
                      //int *ip;
                      //ip=&i;
      int *cp=&c;
      int *dp=&d;

      ++(*ip);        //increments whole data even if it is
                      //represented by more than one bytes.
      ++(*cp);
      ++(*dp);
      printf("%d %c %lf\n",i,c,d);
}
```

```
o/p > 401 b 24.4
```

<Q> If pointers are used for storing address and are of same size (4 bytes), then what is the significance of pointer to be int, char or double?

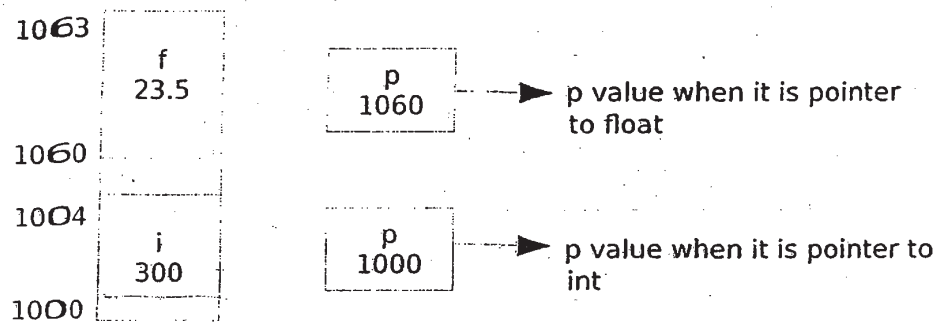
- char *cp; //is a pointer to character, means what type of data it is pointing

consider following example;

<Prog>

```
#include<stdio.h>
main()
{
      float f=23.5;
      int i=300;
      char *p=&f;
      (*p)++;
      printf("f = %f\n",f);
      p=&i;
      (*p)++;
      printf("i = %d\n",i);
```

```
o/p >  f = 23.500002
       i = 301
```



in above example, char pointer is used to store address of float variable. So it will store base address of that variable, but as pointer is of char type it represents only char data, that means it will consider only one bytes data to which it is containing address. So if we edit its data using (*p)++ operation, it only considers base address of that variable and considering it as char data increments only one byte's data.

Now if we try to print edited data, what data will be displayed depends on CPU architecture. It may be :

(1). Little endian CPU
(2). Big endian CPU

in little endian CPU, in case of double is will be significand's lower bits. So incrementing it by 1 will not have that much significance on our data. But in case of integer, it will be lower byte, which when incremented, increments whole data as a result.

but putting CPU architecture's discussion aside, let us consider another example.

<Prog>

```
#include<stdio.h>
main()
{
      float f=23.5;
      int i=300;
      float *p=&f;
      int *q=&i;
      (*p)++;
      printf("f = %f\n",f);
      (*q)++;
      printf("i = %d\n",i);
}
```

```
o/p > f = 24.500000
      i = 301
```

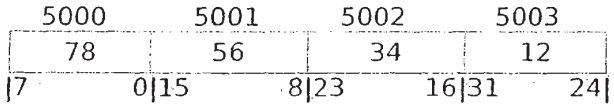
using pointers of specific type means that it is pointing to that type of data. i.e. if we use integer pointer, compiler means that pointer is pointing to integer data and while executing (*q)++, fetches 4 bytes of data starting from the base address to which pointer is pointing, and does increment operation on the whole data instead of single byte like in case of char pointer.

Float pointer also works the same way. Compiler understands that it is pointing to float data and fetches whole 4 byte data while performing increment operation.

CPU Architecture:

Little endian CPU : (lower byte first)

if we consider an integer and try to store hex data 0x12345678 in integer variable then it will be stored in memory like below if memory locations from 5000 to 5003 are allocated to that variable.

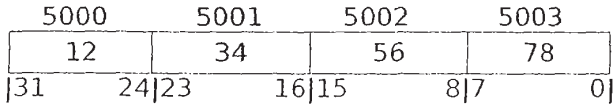


L.E

in little endian PC, lower byte (i.e. bits 7 to 0) which is data 12 is stored first in location 5000, then bits 15 to 8 (34) gets stored in location 5001 and successively, bits 31 to 24 (78) gets stored in location 5003.

Big endian CPU : (upper byte first)

in big endian CPU, hex data 0x12345678 will be stored in memory like shown below.



B.E

in big endian CPU, lower byte (bits 7 to 0) will go in higher location which is 5003, then bits 15 to 8 will go in 5002, and higher byte (bits 31 to 24) will go in 5000.

Pointer type casting : `*(<type>*)p;`

e.g. if int pointer p is there, and you want to type cast it to float type, then

```
-> int *p;
>> *(float*)p; //type casted to float pointer
```

```
/*  To know about byte ordering, consider following examples*/

<Prog>
#include<stdio.h>
main( )
{
      unsigned int i=0x12345678,j;
      char *p=&i;

      j=*p;
      printf("char pointer: j=%x\n",j);
      j=*(short int*)p;
      printf("short int pointer: j=%x\n",j);

      j=*(unsigned int*)p;
      printf("unsigned int  pointer: j=%x\n",j);
}

o/p >  char pointer: j=78
       short int pointer: j=5678
       unsigned int  pointer: j=12345678
```

first of all, char pointer's dereference is assigned to j; so it will be containing only base byte's data. Our PC is little endian, so lower byte of entered hex data 78 will be stored first in base address. So char pointer will fetch only that byte and assign it to j. Now how that result will be stored in j is also interesting. 78 data will be stored in its base address. And then successive 3 bytes will be made 0. for example if j's base address is 5000, then data 0x78 will be stored in j like this.

5000	5001	5002	5003
78	00	00	00
|7 0|15	8|23	16|31	24|

Now using statement `j=*(short int*)p;` char pointer is type casted to short int type pointer. So it will be pointing to short int data which is of 2 bytes. So it will fetch first two bytes starting from base address of i, and store it in variable j. Now j will be having data equivalent to 0x5678, which will be stored in j like shown below.

5000	5001	5002	5003
78	56	00	00
|7 0|15	8|23	16|31	24|

similarly, using statement `j=*(unsigned int*)p;` char pointer is type casted to unsigned int type. Which will fetch whole 4 bytes data from dereference of p, which is integer i itself. So j will be having data 0x12345678.

<Prog >

```
#include<stdio.h>
main()
{
      float f=34.25;
      char *p=&f;
      printf("%d\n",*p);
      p++;
      printf("%d\n",*p);
      p++;
      printf("%d\n",*p);
      p++;
      printf("%d\n",*p);
}
```

```
o/p > 0
      0
      9
      66
```

incrementing char pointer using p++ will increment its address by 1, because as its char pointer, after incrementing it by 1, it will point to next chat after it, which is 1 byte after current byte.

if pointer is of int type, incrementing it will point to the next integer. As integers are of four byte, pointer will be having location of next int. For example, if integer pointer is pointing to location 1000, that means that pointer it pointing to the integer whose base address is 1000. next integer after it will be starting from 1004, so after incrementing int pointer, its value will be 1004.

in case of double pointer, if will point next double when it is incremented. So location after 8 bytes from base address will be pointed by it.

<Prog>

```
#include<stdio.h>
main()
{
      float f=34.25;
      char *p=&f;
      int i;
      for(i=0;i<=3;i++)
      {
            printf("%d\n",*(p+i));
      }
```

```
o/p > 0
      0
      9
      66
```

`for(i=0;i<=3;i++)` will print from lower location, which will not be correct byte ordering if PC is little endian.

In little endian PC, lower byte will be stored in lower location. So to print our data from higher byte to lower byte for better understanding, we will have to print if from higher location to lower location. To print so for loop should be starting from 3 and ending with 0. for big endian pcs this is correct method to print bytes in correct ordering.

For little endian, for loop should be like `for(i=3;i>=0;i--)`

--> consider `*(p+i)`, if i is 3, and p is integer pointer containing base address of integer which is 1000, then `*(p+3)` will point to next 3rd integer data, whose base address will be 1012.

<Prog>

```
#include<stdio.h>   /* To print data in binary format */
main()
{
     float f;
     char *p=&f,ch;
     int i,j;
     printf("enter real data:");
     scanf("%f",&f);
     for(i=3;i>=0;i--)
     {
          ch=*(p+i);
          for(j=7;j>=0;j--)
               printf("%d",(ch>>j)&1);
     }
}
```

```
o/p > Enter real data : 13.4
      01000001010101100110011001100110
```

<Prog>

```
#include<stdio.h>   //Minimized Program using integer pointer
main()
{
     float f;
     int *p=&f,v,i;
     printf("enter real data:");
     scanf("%f",&f);
     v=*p;
     for(i=31;i>=0;i--)
          printf("%d",(v>>i)&1);
}
```

```
o/p > Enter real data : 13.4
      01000001010101100110011001100110
```

```
ptr   // contains address of data
*ptr  // contents of data
&ptr  // address of pointer
```

<Prog>

```
#include<stdio.h>
main( )
{
      char *p1;
      int *p2;
      int v=0x01020304;
      p1=&v;
      p2=&v          // equivalent to p2=p1;
      printf("p1=%u\n",p1);
      printf("p2=%u\n",p2);
      printf("&p1=%u\n",&p1);
      printf("&p2=%u\n",&p2);
      printf("*p1=%d\n",*p1);
      printf("*p2=%d\n",*p2);
}
```

```
o/p > p1=4278534124
      p2=4278534124
      &p1=4278534128
      &p2=4278534136
      *p1=4
      *p2=1020304
```

<Assignment> Write a program to print binary equivalent of double (using char pointer or short int pointer, int pointer and long long int pointer)

```
// Using char pointer
#include<stdio.h>
main()
{
     double d;
     int i,j;
     printf("enter real data:");
     scanf("%lf",&d);
     char *p=&d,cp;
     for(i=7;i>=0;i--)
     {
          cp=*(p+i);
          for(j=7;j>=0;j--)
               printf("%d",(cp>>j)&1);
     }
}
```

```
o/p > enter real data:23.4
01000000001101110110011001100110011001100110011001100110011001100110
```

```
// Using int pointer
#include<stdio.h>
main( )
{
     double d;
     int i,j;
     printf("enter real data:");
     scanf("%lf",&d);
     int *p=&d,ip;
     for(i=1;i>=0;i--)
     {
          ip=*(p+i);
          for(j=31;j>=0;j--)
               printf("%d",(ip>>j)&1);
     }
}
```

```
// Using long long int pointer
#include<stdio.h>
main()
{
     double d;
     int i,j;
     printf("enter real data:");
     scanf("%lf",&d);
     long long int *p=&d,ip;
     ip=*(p+i);
     for(j=63;j>=0;j--)
          printf("%d",(ip>>j)&1);
}
```

<Assignment> Write a program to input a short integer, convert the given data into big endian byte ordering, if current CPU is little endian. If CPU is big endian, do nothing.

```
#include<stdio.h>
main()
{
     short int data,temp=1;
     printf("enter data:");
     scanf("%hd",&data);
     char *p,cp;
     p=&temp;
     if(*p)
     {
          p=&data;
          cp=*p;
```

```
            *p=*(p+1);
            *(p+1)=cp;
      }
      printf("data : %hd\n",data);
}
```

```
o/p >  enter data:1
       data : 256
```

in little endian PC, 1 will be stored in 1st location and then in next byte 0 will be stored, when converted in big endian, byte positions gets swapped, so that new data will be 256.

little endian data : 00000001 00000000
big endian data : 00000000 00000001

when printed the converted data, little endian CPU will treat higher locations as higher byte, so weightage of 1 increases from 2^0 to 2^8 which is 256.

```
                        /* other method */

#include<stdio.h>
short int littleToBig(short int data)
{
      data=(data>>8)|(data<<8);
      return data;
}
main()
{
      short int data,temp=1;
      printf("enter data:");
      scanf("%hd",&data);
      char *p=&temp;
      if(*p)
            data=littleToBig(data);
      printf("data : %hd\n",data);
}
```

```
o/p > enter data:1
      data : 256
```

```
                    /* using call by reference */

#include<stdio.h>
void littleToBig(char *p)
{
     char cp=*p;
     *p=*(p+1);
     *(p+1)=cp;
}
```

```
main( )
{
      short int data,temp=1;
      printf("enter data:");
      scanf("%hd",&data);
      char *p=&temp;
      if(*p)
           littleToBig(&data);
      printf("data : %hd\n",data);
}
```

o/p > same as above

as we are passing address of short int variable, it should be received by called function by short int type pointer, but in our application as we need char pointer to swap data, char pointer is used, for which compiler gives us warning for that.

```
test.c:2:6: note: expected 'char *' but argument is of type 'short
int *'
 void littleToBig(char *p)
      ^
```

<Ex> Little endian to Big endian for integer data

```
#include<stdio.h>
void littleToBig(char *p)
{
      char ch;
      ch=*p;
      *p=*(p+3);
      *(p+3)=ch;
      ch=*(p+1);
      *(p+1)=*(p+2);
      *(p+2)=ch;
}
main()
{
      int a=0x1345678,temp=1;
      char *p;
      printf("original data=%x %d\n",a,a);
      p=&temp;
      if(*p)
      {
           littleToBig(&a);
           printf("big endian converted data=%x %d\n",a,a);
      }
      else
           printf("already big endian, no need to convert\n");
}
```

```
o/p >  original data=1345678 20207224
       big endian converted data=78563401 2018915329
```

Generic Pointer :

Generic Pointer : a pointer variable which is declared as void type.

```
syntax : void *ptr;      // also of 4 bytes

*ptr;      //syntax error, type should be provided to pointer
void v;    //syntax error, void type variable is not possible
```

if we directly dereference generic pointer, then its syntax error because compiler will not know how many bytes to fetch from memory(dereference).

<Ex>

```
#include<stdio.h>
main()
{
     char c1='A',c2;
     int i1=1234,i2;
     double d1=123.4,d2;

               /* if char pointer is used to fetch data */

     char *ptr;
     ptr=&c1;
     c2=*ptr;

     ptr=&i1;
     i2=*(int *)ptr;

     ptr=&d1;
     d2=*(double *)ptr;

           /* if void pointer is used to fetch data */

     void *ptr;
     ptr=&c1;
     c2=*ptr;

     ptr=&i1;
     i2=*(int *)ptr;

     ptr=&d1;
     d2=*(double *)ptr;

     printf("%c %d %lf\n",c2,i2,d2);
```

```
}
```

in above program, in both the cases, char pointer is used or void pointer is used, answer will be same.

<Q> Benefits of generic pointer.
If we forget to type case pointer according to type of the data to be fetched by pointer, then bug appears in the program. But if generic pointer is used, then we have to compulsorily type cast it according to the type of the data. If we forget to do so, then its syntax error in place of bug. So using generic pointer reminders programmer to typecast the pointer while dereferencing.

Type Qualifier : Const

<Ex>

```
#include<stdio.h>
main( )
{
        const int i;    //read only variable
        int j;
        printf("i=%d at %u\n",i,&i);
        printf("j=%d at %u\n",j,&j);
}
```

```
o/p >  i=134513771 at 3212850904
       j=-1216917504 at 3212850908
```

from above program, we can see that both i and j are in stack section.
But in above program if we use `i=10;` then its error as shown below.

```
test.c: In function 'main':
test.c:6:2: error: assignment of read-only variable 'i'
  i=10;
  ^
```

<Ex>

```
#include<stdio.h>
main()
{
        const int i=10;
        int j;
        j=i*2;
        printf("i=%d at %u\n",i,&i);
        printf("j=%d at %u\n",j,&j);
}
```

```
o/p > i=10 at 3216242216
      j=20 at 3216242220
```

from above example it is clear that only initialization is possible on const variables, further modification using assignment operator is not allowed. So conclusion from above example is that read only variables must be initialized.

<Q> why at all we need const type qualifier? If it allocates memory to the variable and we cant modify it then why even allocate it memory?

--> best example of such case is using pi value in our program. Suppose we need a precise value of pi in our program, then instead of writing long constant value every time in our program it is better solution to store it in variable and use the variable.

Now if in my program, more than one function needs pi value, then we should declare variable pi as global variable. But a careless programming can change the value of pi in any of the function and such unknown modification may result in bug.

So in order to prevent such modifications, const type qualifier is used.

<Ex>

```
#include<stdio.h>
main()
{
        const int i=10;
        int j,*p;
        printf("i=%d\n",i);
        printf("j=%d\n",j);
        j=i*2;
        p=&i;            //assignment happens runtime

        *p=j*2;          //user knows p points to const type
                         //variable, compiler doesn't
        printf("i=%d\n",i);
        printf("j=%d\n",j);
}
```

```
o/p > i=10
      j=134513835 (garbage)
      i=40
      j=20
```

note that j gets value of i*2 at runtime and not at compile time even if value of is known at compile time.

Conclusion : const type variables can be modified by using pointer.

<Ex>

```
#include<stdio.h>
main()
{
        int i=10;
```

```
        const int *p;
        p=&i;
        *p=20;
        printf("i=%d\n",i);
}
```

```
test.c: In function 'main':
test.c:7:2: error: assignment of read-only location '*p'
  *p=20;
  ^
```

by statement `const int *p;` compiler understands that p is a pointer pointing to constant variable. So `*p=20;` is not allowed even if variable which p is pointing is not constant.

```
int *p1;
```

p1 is a variable pointer pointing to variable integer.

```
p1=&var;  //allowed
*p1=10 ;  //allowed
```

```
const int *p2;
```

p2 is a variable pointer pointing to constant variable of integer type.

```
p2=&var;  //allowed
*p2=10 ;  //not allowed
```

```
int *const p3;
```

p3 is a constant pointer pointing to a variable integer.

```
p3=&var;  //not allowed
*p3=10;   //allowed
```

```
const int *const p4;
```

p4 is a constant pointer pointing to constant variable of integer type.

```
p4=&var;  //not allowed
*p4=10;   //not allowed
```

OS Memory Protection :

if we assign some random address directly to the pointer and try to access or modify value at that location using pointer, then if the memory location assigned is not within permissible range of memory, it will be segmentation fault. This is because OS denies the access to the portion of memory where user is not allowed to access or modify. This concept is called operating system's memory protection concept.

Storage Class

Syntax :
```
storage_specifier datatype var_name/fun_name;
```

--> For usage on **variables**, 4 storage class are possible

- Auto |
- Register | <-- all are keywords, but not datatype
- Extern |
- Static |

e.g. - `static int s;`

--> For usage on functions, 2 storage class are possible

- Static
- Extern

e.g. `static int f1(void);`

--> Storage class for all identifiers are existing.

--> If we don't provide storage class to variables which are declared inside functions (**local variables**), its storage class is **auto** by default.

-> If variable is declared outside functions, (**global variables**), its storage class is **extern** by default.

--> If storage class is not defined for any **function**, its storage class is **extern** by default.

--> Storage class decides :

- memory of identifier
- life of identifier
- scope of identifier
- default initial value (D.I.V.) of identifier

Auto : (default for local variables)

Memory : stack
Life : starts with the function call, gets destroyed with return
Scope : visible to the function where it is declared
D.I.V. : undefined / garbage

Register

Memory : preferably CPU register, but guaranteed to be in stack
Life : same as auto
Scope : same as auto
D.I.V. : same as auto

Extern: (default for global variables)

Memory : data section
Life : starts with application execution, terminates with application exit
Scope : visible to all functions present in executable
D.I.V. : 0

Static :

Memory : same as extern
Life : same as extern
Scope : visible to the functions where declared. Has internal linkage
D.I.V. : same as extern

Register Type :

<Ex>

```
#include<stdio.h>
register int v;     // syntax error
main()
{
     _
     _
}
```

Syntax error because, variable is declared as global variable, which are present in data section. But using register storage specifier with it, it indicates that variable should be in CPU register, so there is conflict of storage class.

NOTE : Pointers can not be used with register type variable, because CPU registers dont have address and register type variables are in CPU registers.

<Ex>

```
#include<stdio.h>
main()
{
     register int v;
     printf("%d",v);     //prints garbage value
     printf("%u",&v);    //syntax error, because CPU registers
don't                          have addresses
}
```

addresses of variable which are in data, heap or stack only is allowed to be printed (i.e. variable present in RAM only).

- If much many variables (more number of variables than there are registers available in CPU) are declared using register as shown below

```
register int v1,v2,...;
```

then compiler ignores such request. Because CPU registers are for

shared usages among all other running processes. One process can not hold CPU register for long time. And all variables will be allocated memory in stack.

<Ex>

```
#include<stdio.h>
int g;                  // global variables - extern by default
void print(int x)       // formal arguments - register by default
{
     printf("%d\n",x);
}
main()
{
     int v;             // local variables – auto by default
     printf("Enter int for v:");
     scanf("%d",&v);
     print(v);
}
```

<Ex>

```
register int i;
for(i=0;i<=1000000;i++)

{          | to execute loop  again and again, i will be in CPU
     -     | register. And through out the execution it will be in
     -     | CPU register.
}
```

in such kind of operations, compiler keeps variable in register because in every loop iteration, CPU has to access the variable and do some kind of operation on variable. So if variable is in stack, it takes more time to fetch variable every time. In such cases variable will be in register because, operations on register are faster than operations on memory.

NOTE : CPU registers can never be occupied by only one application Throughout life of that application

- Variable should be using register or not depends upon compiler. Some compilers like gcc, when does optimization on code, may use CPU register for auto variables if necessary for faster execution of process. When compiler tries to optimize the code, it keeps a copy of variable in CPU register until operations upon it are completed.

Extern Type :

<Ex>

```
#include<stdio.h>
extern int v;        // pure declaration but not definition
inc()
{
     v++;
}
main()
{
     inc();
     printf("v=%d\n",v);
}
```

```
o/p > /tmp/ccZcJZtw.o: In function `inc':
      test.c:(.text+0x6): undefined reference to `v'
      test.c:(.text+0xf): undefined reference to `v'
      /tmp/ccZcJZtw.o: In function `main':
      test.c:(.text+0x25): undefined reference to `v'
      collect2: error: ld returned 1 exit status
```

Linking error is there because statement, `extern int v;` is pure declaration but no initialization. There is error in linking stage so, up to assembling stage there is no error. So if we compile object file from above program using -c switch and use nm command for it, we can see that variable v is undefined – U.

If `int v;` had been used in our program, it is declaration plus definition (because global variables are initialized to 0 by default). So in such case there will not be any linking error.

```
extern int v;   // pure declaration
int v;          // declaration + definition with default value 0
static int v;   // declaration + definition with default value 0
auto int v;     // declaration + definition with default garbage value
```

--> if we have creates 3 files named main.c, f1.c, f2.c as shown below

main.c

```
#include<stdio.h>
extern int v;
main()
{
     void f1(void);
     f1();
     printf("v=%d\n",v);
}
```

f1.c

```
void f1(void)
{
     extern int v;
     void f2(void);
```

```
        f2();
        v=v+10;
}
```

f2.c

```
void f2(void)
{
        extern int v;
        v=v+5;
}
```

if we compile executables of above programs using -c switch with cc command, and then try to link all three object files main.o, f1.o, f2.o using **`cc main.o f1.o f2.o`**, then its linking error. Its because v is undefined in all three object files. We can check it using nm command on object files. Nowhere v has been initialized, so it will be linking error.

```
nm main.o :                  U f1
            0000000000000000 T main
                             U printf
                             U v

nm f1.o :   0000000000000000 T f1
                             U f2
                             U v

nm f2.o :   0000000000000000 T f2
                             U v
```

if we make another file named data.c like shown below,

data.c

```
int v=20;
```

if we use nm command with data.o, it shows v is defined.

nm data.o : 0000000000000000 D v

now compiling above object files with command **`cc main.o f1.o f2.0 data.o`** then there is no linking error because v is defined in data.c

variable v's linkage with other files f1.c and f2.c is called external linkage. **Extern storage qualifier supports external linkage.** That means, extern variable named v of one program can be linked with variable v in another programs.

--> now if all programs are rewritten as shown below

main.c

```
#include<stdio.h>
int v;
main()
{
     void f1(void);
     f1();
     printf("main: v=%d\n",v);
}
```

f1.c

```
void f1(void)
{
     int v;
     void f2(void);
     f2();
     v=v+10;
     printf("f1: v=%d\n",v);
}
```

f2.c

```
void f2(void)
{
     int v;
     v=v+5;
     printf("f2: v=%d\n",v);
}
```

compiling using **cc main.o f1.o f2.o**

```
o/p > f2: v=5
      f1: v=10
      main: v=0
```

now there is no linking error. When we use nm commands with these program's object files,

```
nm main.o :                    U f1
           0000000000000000 T main
                               U printf
           0000000000000004 C v

nm f1.o :  0000000000000000 T f1
                               U f2
                               U printf

nm f2.o :  0000000000000000 T f2
                               U printf
```

variable v is common symbol in main.o, common symbols are considered as weak symbols. Weak symbols are that symbols which are not been initialized. But v in main.c is having default value 0, so all other weak symbols will use

VIGNAN - R4
BHAVISH

that value.

NOTE : Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.

```
extern int v;  // undefined
int v=20;      // strong symbol
int v;         // weak symbol
```

NOTE : D/B : Strong symbol
c : weak symbol

--> if in main.c in place of `int v;` statement, `int v=10;` is present, then using nm command we can see that v is now strong symbol listed as B (indicates BSS section)

- weak symbols are treated by compiler as undefined
- if strong symbols are present in more than one file, it will be linking error
- if strong symbol is present in one source file and in other files weak symbols are present, then weak symbols will be treated with strong symbol's value
- if we declare a variable as extern in one function, then in order to use that variable in other functions, we will have to declare that variable as extern in all other functions. To avoid such situations, extern variables should be declared outside and above every functions so that all functions can use it with only one declaration.

<Ex>

```
#include<stdio.h>
int g;                    // weak symbol
void print(void)
{
     printf("print: g=%d\n",g);
     g++;
}
main()
{
     int v=1;
     g=g+10;
     printf("enter v:");
     scanf("%d",%v);
     if(v<0)
          print();
     printf("main: g=%d\n",g):
}
int g=5;  // or     g=5; // strong symbol
o/p > enter v:-1
      print: g=15
      main: g=16
```

here statement `int g=5;` will never execute but will be used by compiler to make an arrangement so that when g comes into life, its initial value is 5.

If `int g;` is commented, then it will be syntax error because g is used on program but not declared.

If in above program, if `extern int g;` is used above main, and below main, `int g=5;` is commented then it is linking error because `extern int g;` is pure declaration not definition, so g will be undefined symbol in the program.

If above main, `extern int g;` // undefined symbol
and below the main, `int g;` // weak symbol
then there is no error because if there are no other strong symbols are present in program then undefined symbols will use weak symbol's default value, which in this case will be 0.

Static Type :

<Ex>

```
#include<stdio.h>
void print(void)
{
        static int s=10;
        s++;
        printf("s=%d\n",s);
}
main()
{
        int v;
        printf("Enter value of v:");
        scanf("%d",&v);
        if(v<0)
        {
                print();
                print();
                print();
        }
}
```

in above program, print()'s call is depended on which value is entered by user. If print() function is not called, then s variable will be in life because it is present in data section.

<Q> Even though a statement `static int s=10;` is present in print() function, why variable v doesn't contain 10 every time and retain its value, but only first time value 10 is assigned to v?

<Ex>

```
#include<stdio.h>
void print(void)
{
      static int v=10;
      int a=10;
      a++; // stack
      v++; // data section
      printf("v=%d\n",v);
      printf("a=%d\n",a):
}
main( )
{
      print();
      print();
      print();
}
```

```
o/p >  v=11
       a=11
       v=12
       a=11
       v=13
       a=11
```

--> `static int v=10;` is initialization, so 10 value will not be assigned to v at runtime, it will be used by compiler to make an arrangement for v's initial value to be 10 when execution starts. But in case of `int a=10;` value 10 will be assigned to variable a at runtime.

In print()'s execution of three times, variable a will be created and collapsed 3 times, but variable v will be created and collapsed only 1 time.

This is because variable a is present in function's stack. So when function is called 3 times, its stack is also created and collapsed 3 times. So every time a function is called, a new variable a will be created, but variable v is present in data section. So it doesn't collapse, but through function's call, its value is updated according to the operation done on variable v.

NOTE : Llinkages are only for function, static and global variables. Not for auto or register variables

Static - internal linkage (can only be used in source file in which it is declared)
Extern - external linkage (can be used in other source files also)
Auto - no linkage (for local variables there is no linkage)

<Prog>

```
#include<stdio.h>
void print(void)
{
        static int s=10;
```

```
        s++;
        printf("in print : s=%d\n",s);
}
main()
{
        static int s;
        print();
        print();
        print();
        printf("in main : v=%d\n",s);
}
```

```
o/p >  in print : s=11
       in print : s=12
       in print : s=13
       in main : v=0
```

here both s variables are present in data section but both's are having different addresses, so main's s is different from print's s.

--> if in above program, static variable s is declared globally, its scope is within whole file.

```
o/p >  in print : s=11
       in print : s=12
       in print : s=13
       in main : v=13
```

--> if in another source file f.c, static int s is present, then we cant link our file's static variable with f.c's static variable s. Both variables are different and having internal linkages with respect to that source file only. External linking is not allowed on static variable. Its allowed only on extern variables.

```
o/p >  in print : s=11    |
       in print : s=12    |    this is test.c's variable
       in print : s=13    |
       in main : s=0      // this is f.c's variable
```

--> if in f.c in place of static variable, extern variable is present, then it will be linking error like shown below.

```
/tmp/ccBSipSU.o: In function `f':
f.c:(.text+0x7): undefined reference to `s'
collect2: error: ld returned 1 exit status
```

--> if in f.c `int s;` is written in place of static int s; then there will be no error

```
o/p > in print : s=11
      in print : s=12
      in print : s=13
      in main : s=0
```

Some facts about storage class :

- if in main.c, f1.c, f2.c, extern int v is present, v is initialized in main.c, and we want to change v's value in f1.c then we cant use initialization again, we will have to use assignment.
- If in one file, 2 functions are present, like main() and f1(). `extern int g;` is present in main(), then it cant be accessed in f1() until we declare g in f1() also using `extern int g`. To be used in all the functions, extern variables must be declared globally.
- We cant initialize extern variable inside any function. It must be initialized outside the function only. Same way if we have declared extern variable inside the function and try to assign values directly, its error.

<Prog>

```
#include<stdio.h>
int v=10;           // global variable – data section
f()
{
     v++;           // global v --> from 10 to 11
}
main()
{
     int v=10;      // local v – stack section (auto)
     f();
     printf("in main : v=%d\n",v);
}
```

```
o/p > in main : v=10
```

we can declare global and local variable with using same name for both. Both variable will be different from each other.

<Prog>

```
#include<stdio.h>
int v=10;                          // scope – global (extern)
f2()
{
        printf("in f2 : v=%d\n",v);     // prints global v
}
f1()
{
        static int v;          // scope – within f1() only
        v++;
        f2();                  // static v from 0 to 1
        printf("in f1 : v=%d\n",v);     // prints static v
}
```

```
main()
{
        int v=20;              // scope – within main only
        f1();
        printf("in main : v=%d\n",v);   // prints local v
}
```

```
o/p > in f2 : v=10
      in f1 : v=1
      in main : v=20
```

--> if in main()
`static int v;` and `int v;` both are present, it is syntax error. Because in a function two variables cant hold one name even if their storages are different from each other.

<Prog>

```
#include<stdio.h>
int v=10;
main()
{
        int v=1;
        printf("in main : v=%d\n",v);
        {
                v++;
                printf("in block : v=%d\n",v);
        }
        printf("in main : v=%d\n",v);
}
```

```
o/p > in main : v=1
      in block : v=2
      in main : v=2
```

even though global variable is present, if a local variable is created in main with same name, it will be accessed in main, not the global variable.

--> if in block, `int v=10;` is present, then v is visible inside block only

```
o/p > in main : v=1
in block : v=11
in main : v=1
```

<Prog>

```
#include<stdio.h>
int v=10;
main()
{
        int v=1;
        switch(v)
        {
                int g=10;
```

```
                case 1: printf("g=%d\n",g);
                        break;
        }
}
```

o/p > g=-1217564672 (garbage)

int g=10; will not be executed in any case, but compiler acknowledges its declaration, but in case of local variables initialization is done at runtime. So g will be created once program execution starts and initialized with garbage value.

```
--> static int f1(int);        // static function
```

static function can only be used by other functions of that source file only, it cant be used outside that source file

```
--> extern void f1(int);       // extern function
```

extern function can also be used outside the source file in which it is defined. All functions are extern by default until we declare it as static.

NOTE : printf() and scanf() are extern functions, so that we can use them from any source files

Recursion

if function calls itself, its **recursive function**.

<Ex>
```
#include<stdio.h>
main( )
{
        printf("in main\n");
        main();
        printf("returning from main\n");
        return;
}
```

in above program, main will be called and new stack frame will be created every time it is called. Program execution stops only when stack is overflow by main's stack frames. Program never terminates normally. In recursive functions, we must control repetitions or how many times function will be called, otherwise stack will be overflowed.

<Ex>
```
#include<stdio.h>
main()
{
        int v=0;
        v++;
        printf("in main : v=%d\n",v);
        if(v<=5)
                main();
        printf("returning from main\n");
        return;
}
```

if a local variable is used to control repetitions like shown above, it will not work because first v will be 0, then it will be increased to 1. then condition will be checked, as it is true it will call main again. A new stack frame will be created for main, a new variable v will be created and initialized to 0, again its value will be incremented to 1. so in such way main will be called until stack is overflow.

-> if static variable is used in place of local variable, then repetitions can be controlled.

<Ex>
```
#include<stdio.h>
main()
{
    ①       static int v=0;
    ②       v++;
```

```
    ③       printf("in main : v=%d\n",v);
    ④       if(v<=5)
    ⑤               main();
    ⑥       printf("returning from main\n");
    ⑦       return;
}
```

```
o/p > in main : v=1
      in main : v=2
      in main : v=3
      in main : v=4
      in main : v=5
      in main : v=6
      returning from main
      returning from main
      returning from main
      returning from main
      returning from main
      returning from main
```

above program can be understood well with help of the diagram of stack frames creation like shown below.

V=0/1	2, 3, 4, 5, 6, 7
V=1/2	2, 3, 4, 5, 6, 7
V=2/3	2, 3, 4, 5, 6, 7
V=3/4	2, 3, 4, 5, 6, 7
V=4/5	2, 3, 4, 5, 6, 7
V=5/6	2, 3, 4, 6, 7

In above diagram, all the stack frames of main and in each stack frames, values of variable v is shown. On the side of stack frames, executed statement's number are shown using underline. These are statements are executed in previous main before calling a new main function. Remaining statements in previous main function will be executed once a newly called main executes and returns first.

<Ex>

```
#include<stdio.h>
main()
{
    ①   int i;
    ②   static int v;
    ③   i=v;
    ④   v++;
```

```
⑤     printf("in main : v=%d\n",v);
⑥     if(v<4)
⑦             main();
⑧     printf("returning : i=%d\n",i);
}
```

```
o/p > in main : v=1
      in main : v=2
      in main : v=3
      in main : v=4
      returning : i=3
      returning : i=2
      returning : i=1
      returning : i=0
```

i=0 v=0/1	3, 4, 5, 6, 7, 8
i=1 v=1/2	3, 4, 5, 6, 7, 8
i=2 v=2/3	3, 4, 5, 6, 7, 8
i=3 v=3/4	3, 4, 5, 6, 8

<Ex>

```
#include<stdio.h>
void f(int v)
{
        printf("in f : v=%d\n",v);
        if(v)
                f(v-1);
        printf("returning : v=%d\n",v);
}
main()
{
        f(3);
}
```

```
o/p > in f : v=3
      in f : v=2
      in f : v=1
      in f : v=0
      returning : v=0
      returning : v=1
      returning : v=2
      returning : v=3
```

main

f v=3

f v=2

f v=1

f v=0

<Ex>

```
#include<stdio.h>
void f(int v)
{
        printf("in f : v=%d\n",v);
        if(v)
                f(--v);
        printf("returning : v=%d\n",v);
}
main()
{
        f(3);
}
```

```
o/p > in f : v=3
      in f : v=2
      in f : v=1
      in f : v=0
      returning : v=0
      returning : v=0
      returning : v=1
      returning : v=2
```

main

f v=3/2

f v=2/1

f v=1/0

f v=0

<Ex>

```
#include<stdio.h>
void f(int v)
{
        printf("in f : v=%d\n",v);
        if(v)
                f(v--);
        printf("returning : v=%d\n",v);
}
main()
{
        f(3);
}
```

```
o/p > stack overflow
```

<Prog> Find factorial of given number using recursion.

```
#include<stdio.h>
int fact(int n)
{
        if(n<=1)
                return 1;
        return (n*fact(n-1));
}
main()
{
```

```
        int n,f;
        printf("Enter n :");
        scanf("%d",&n);
        f=fact(n);
        printf("factorial of %d is %d\n",n,f);
}
```

o/p > Enter n :4
factorial of 4 is 24

f=fact(4)	f=24
n=4	ret 4*fact(3)=4*6=24
n=3	ret 3*fact(2)=3*2=6
n=2	ret 2*fact(1)=2*1=2
n=1	ret 1

<Q> For a program like finding factorial, using loop is better method or using recursion is better method?

--> in recursion - stack may overflow if many number of calls happen.
- execution is slower because of push and pop operations

in loop - execution faster because no push and pop operations
- no possibility of stack overflow.

<Q> For embedded system programming, recursion is not preferred. Why?

--> Because in recursion, every time function is called, new stack frame is created. And in embedded systems, stack is limited. so...

<Prog> Write a program to calculate sum of digits using recursion

```
#include<stdio.h>
int sum(int data)
{
        if(data)
                return ((data%10) + sum(data/10));
}
main()
{
        int data;
        printf("Enter a number :");
        scanf("%d",&data);
        printf("sum of digits = %d\n",sum(data));
}
```

while (n>0)
{
r = n % 10 <-
n = n/10 <- n pass & every time changes, so call
a = a+r
}

o/p > Enter a number :12345
sum of digits = 15

<Prog> W.A.P to print Fibonacci value of given number

```
/* Fibonacci values
pos value
 1 -> 0
 2 -> 1
 3 -> 1
 4 -> 2
 5 -> 3
 6 -> 5
 7 -> 8   */

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

```
o/p > Enter a number :7
      Fibonacci value = 8
```

<Prog> Fibonacci series printing

```
#include<stdio.h> number :7
int fib(int data)
{
        if(data==1)
                return 0;
        else if(data==2)
                return 1;
        else
                return (fib(data-2) + fib(data-1));
}
main()
{
        int i,data;
        printf("Enter a number :");
        scanf("%d",&data);
```

```
        for(i=1;i<=data;i++)
                printf("%d, ",fib(i));
        printf("\n");
```

```
/p > Enter a number :10
     0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

<Prog> Reverse a number using recursive functions

```
include<stdio.h>
int pwr(int n)

        if(n>1)
                return (10*pwr(--n));
}
int rev(int data,int n)
{
        if(data)
                return (pwr(n)*(data%10) + rev(data/10,--n));
}
nt dig(int data)
{
        if(data)
                return (1+dig(data/10));
}
main()
{
        int i,data;
        printf("Enter a number :");
        scanf("%d",&data);
        printf("reversed data = %d\n",rev(data,dig(data)));
```

```
/p > Enter a number :12345
     reversed data = 54321
```

```
char * getReverse(char str[])
{
    static int i = 0;
    static char rev[MAX];

    if (*str)
    {
        getReverse(str+1);
        rev[i++] = *str;
    }
    return rev;
}
```

everse string

```
har* str( char *s)

   if (*s)
   {
      str(s+1);
      Pf("%c", *s);
    }
}
```

verse Number 1234 → 4321

```
sum=0, r;

erse(int n)

if(n)
{
   r=n%10;
   sum=sum*10+r;
   reverse(n/10);
 }
else
 {
   return sum;
 }
 return sum;


char * str(char *s, int i, int j)
   {
       char temp;
       temp=s[i];
       s[i]=s[j];
       s[j]=temp;
   if(i>j)
      return s;
   str(s, ++i, --j);
```

Array

Array : collection of variables (members/data) of same type present in adjacent memory. It is an derived data type.

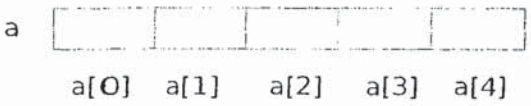
syntax : `datatype array_name[size];`

compiler doesn't have any default size for array, it is user defined.

```
char a[];       //syntax error
char a[v];      //syntax error, because variable are modifiable at
                  runtime but array size should be constant.
```

```
char a[]={'a','b','c','d','e'};
```
if array is initialized, then writing size in brackets is optional. Compiler automatically determines size of an array.

--> `char a[5];`

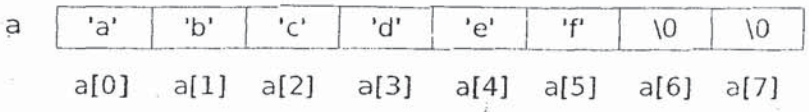


all members in array will be having name 'a' but they can be accessed by using their indexes starting from 0 to size-1. And all elements will be having garbage value when array is not initialized.

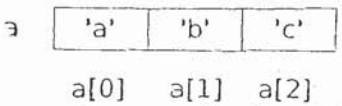
--> If initialization is done and size is not provided by user, its size will be number of elements which are used to initialize array.
e.g.
```
char a[ ]={'a','b','c','d','e','f'};     //array size will be 6
```

--> If array is partially initialized, then remaining data will be having null in case of char type array) or zeros (in case of int arrays).

e.g. `char a[8]={'a','b','c','d','e','f'};`



--> `char a[3]={'a','b','c','d','e','f'};`



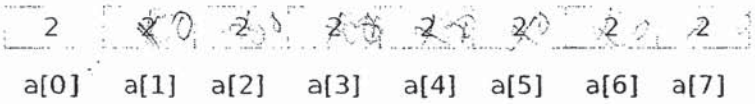
In such cases, it will not be any error. Compiler may generate warning like shown below.

```
test.c: In function 'main':
test.c:4:2: warning: excess elements in array initializer [enabled
by default]
  char a[3]={'a','b','c','d','e','f'};
  ^
```

<Q> Why array is called derived data type?
Compiler doesn't know size of array, user has to provide it so it is (user defined / derived) data type. Array is also called as data structure, as data is stored in adjacent memory blocks and we can save hundreds of data in array with help of loops.

--> if array size is defined and array is initialized with only one element, then all of its members will be having that value.

e.g. `int a[8]={2};` // all members of array a will be 2



<Ex>

```
#include<stdio.h>
main()
{
        int a[3]={11,22,33,44,55};
        int i,n;
        n=sizeof(a)/sizeof(a[0]);       // n=3 (12/4)
        for(i=0;i<n;i++)
                printf("a[%d]=%d\n",i,a[i]);
}
```

```
o/p > a[0]=11
      a[1]=22
      a[2]=33
```

--> if `int a[5]={11,22,33};` (partially initialized array) present in program,

```
o/p > a[0]=11
      a[1]=22
      a[2]=33
      a[3]=0
      a[4]=0
```

--> if `int a[3];` (uninitialized array) present in program,

```
o/p > a[0]=-1216839680   (garbage)
      a[1]=134513787     (garbage)
      a[2]=-1217093632   (garbage)
```

<Ex>

```
#incl ude<stdio.h>
main( )
{
        int a[5],b[5],c[5];
        int i,n;
        n=sizeof(a)/sizeof(a[0]);
        for(i=0;i<n;i++)
                a[i]=10+i;
        for(i=0;i<n;i++)
                b[i]=a[i]+10;
        for(i=0;i<n;i++)
                c[i]=a[i]+b[i];

        for(i=0;i<n;i++)
                printf("a[%d]=%d b[%d]=%d c[%d]=
%d\n", i,a[i],i,b[i],i,c[i]);
}
```

```
o/p > a[0]=10 b[0]=20 c[0]=30
      a[1]=11 b[1]=21 c[1]=32
      a[2]=12 b[2]=22 c[2]=34
      a[3]=13 b[3]=23 c[3]=36
      a[4]=14 b[4]=24 c[4]=38
```

--> now if we print using following statements

```
printf("a[8]=%d\n",a[8]);
printf("a[12]=%d\n",a[12]);
printf("b[-4]=%d\n",b[-4]);
printf("c[-7]=%d\n",c[-7]);
```

```
o/p > a[8]=23
      a[12]=34
      b[-4]=11
      c[-7]=13
```

<Q> if 8, 12, -4, -8 are not valid indexes for the array of size 5, how it is able to print data using those indexes? How compiler treats them?

First of all, **array's name represents its base address** (i.e address of its first element in memory).

```
printf("a : %u\n",a);
printf("b : %u\n",b);
printf("c : %u\n",c);
```

```
o/p > a : 3213583316
      b : 3213583336
      c : 3213583356
```

*a,*b and *c fetches first element of array a,b and c respectively

```
printf("*a : %d\n",*a);
```

```
printf("*b : %d\n",*b);
printf("*c : %d\n",*c);
o/p > *a : 10
      *b : 20
      *c : 30
```

now a[0] is basically dereference operation, **a[0] is *(a+0).** Using subscript is dereference operation. For easiness we use index to access elements of array but actually it is dereference.

So from the base locations of array a,b and c, it is clear that they are in consecutive memory blocks in our programs, indexes which is bigger than size of array is able to fetch data from memory. But this wont be case every time and careless dereferencing by using index, we can try to access the portion of memory which we are not authorized to do, and result in memory fault or segmentation fault.

Now for easiness of calculations, consider a's base address to be 1000. so b's base address will be 1020 and c's base address will be 1040.

a[8] = *(a+8) = 1032, which is address of 4th element of array b.
b[3]=23, so a[8]=23

now similar way, c[-8] = *(c-8) =1012, which is address of 3rd element of array a, a[2]=13, so c[-8]=13.

a[13] or b[-3] doesn't show error although it is invalid index of array because basically a[13] is *(a+13) which is valid. Similarly b[-3] is *(b-3).

<Q> Why index starts with zero (0)?
--> Name of the array represents base address of array, so to get first element of array *a, to get second element from array *(a+1) should be done.

To get data of first element, you have to go through no displacement with dereference which is *(a+0) --> a[0]

<Prog> input 5 integers and find the highest data

```
#include<stdio.h>
main()
{
        int a[5];
        int i,n,max;
        n=sizeof(a)/sizeof(a[0]);
        printf("Input 5 integers :");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        max=a[0];
        for(i=1;i<n;i++)
                if(a[i]>max)
                        max=a[i];
        printf("max number = %d\n",max);
```

```
}
o/p >  Input 5 integers :10 50 30 20 40
       max number = 50
```

<Assignment> W.A.P. to input 10 integers and find the index of highest and lowest data

```
#include<stdio.h>
main()
{
      int a[10];
      int i,n,max,min,max_index=0,min_index=0;
      n=sizeof(a)/sizeof(a[0]);
      printf("Input 10 integers :");
      for(i=0;i<n;i++)
            scanf("%d",&a[i]);
      max=min=a[0];
      for(i=1;i<n;i++)
      {
            if(a[i]>max)
            {
                  max=a[i];
                  max_index=i;
            }
            if(a[i]<min)
            {
                  min=a[i];
                  min_index=i;
            }
      }
      printf("max number = %d at index %d\n",max,max_index);
      printf("min number = %d at index %d\n",min,min_index);
}

o/p > Input 10 integers :12 98 34 46 23 75 87 56 43 86
      max number = 98 at index 1
      min number = 12 at index 0
```

<Assignment> W.A.P to initialize an array of 10 characters. Count number of alpha bates, characters and special characters.

```
#include<stdio.h>
main()
{
        char a[10]={'a','A','1','&','r','R','%','6','q','7'};
        int i,n,alpha=0,num=0,special=0;
        n=sizeof(a)/sizeof(a[0]);
        for(i=0;i<n;i++)
              printf("%c ",a[i]);
        for(i=0;i<n;i++)
        {
              if(((a[i]>='a')&&(a[i]<='z'))||
```

```
((a[i]>='A')&&(a[i]<='Z')))
                        alpha++;
                else if((a[i]>='0')&&(a[i]<='9'))
                        num++;
                else
                        special++;
        }
        printf("\nAlphabates = %d\n",alpha);
        printf("Numbers = %d\n",num);
        printf("Special characters = %d\n",special);
}
```

```
o/p > a A 1 & r R % 6 q 7
      Alphabates = 5
      Numbers = 3
      Special characters = 2
```

Random number generation

rand() function is used to generate random number, it is useful when we want to check algorithms designed on a considerably big array. Assigning different values and checking our program is tiresome job, so instead using rand() function, random numbers can be generated and can be assigned to array.

```
rand()%100;       // will generate random numbers between o to 99
rand()%90 + 10    // same as above

rand()%100 + 100      // 100 to 199
```

--> but this functions generate same set of numbers every time.

```
#include<stdio.h>
main()
{
        int a[10],i;
        for(i=0;i<10;i++)
                a[i]=rand()%101;
        for(i=0;i<10;i++)
                printf("%d ",a[i]);

}
```

```
o/p > 32 32 54 12 52 56 8 30 44 94      (1st run)
      32 32 54 12 52 56 8 30 44 94      (2nd run)
```

--> So to generate different set of random numbers every time srand() function should be used in the program as stated below.

```
#include<stdio.h>
main()
{
        int a[10],i;
```

```
        srand(getpid());            // or srand(time(0));
        for(i=0;i<10;i++)
                a[i]=rand()%101;
        for(i=0;i<10;i++)
                printf("%d ",a[i]);

}
```

```
o/p >  94 21 41 51 61 19 19 89 33 49     (1st run)
       53 56 97 77 19 23 59 15 5 0       (2nd run)
       95 38 78 20 38 73 2 37 25 86      (3rd run)
       22 19 95 82 90 68 68 62 64 88     (4th run)
```

<Assignment> Find higher number from array and all its indexes.

```
#include<stdio.h>
main()
{
        int a[10],i,max=0;
        srand(time(0));
        for(i=0;i<10;i++)
        {
                a[i]=rand()%100;
                printf("%d  ",a[i]);
                if(a[i]>max)
                        max=a[i];
        }
        printf("\nmax = %d\n",max);
        printf("indexes = ");
        for(i=0;i<10;i++)
                if(a[i]==max)
                        printf("%d ",i);
        printf("\n");
}
```

```
o/p > 60  6  31  36  76  28  95  44  10  55
      max = 95
      indexes = 6
```

<Assignment> W.A.P to input 25 characters and count number of vowels.

```
#include<stdio.h>
main()
{
        int i,cnt=0;
        char a[25];
        srand(time(0));
        for(i=0;i<25;i++)
        {
                a[i]=rand()%26+97;
                printf("%c  ",a[i]);
                if((a[i]=='a')||(a[i]=='e')||(a[i]=='i')||
```

```
(a[i]=='o')||(a[i]=='u'))
                    cnt++;
        }
        printf("\nnumber of vowels = %d\n",cnt);
}
```

```
o/p > j  o  q  g  v  i  g  j  k  s  u  v  w  i  f  a  o  k  j  b
      p  e  i  u  a
      number of vowels = 10
```

Passing array to functions :

suppose we want to pass array to functions for some operations on it, we should pass its base address and number of components, so that function can read array elements by dereferencing on base address of array up to number of element of times.

Consider following application. Writing codes for every requirement in main makes main function lengthy, so usage of functions is better option here.

<Prog> Write a program to input 10 integers in array and find maximum and minimum numbers and all its indexes.

```
#include<stdio.h>
void input(int a[],int n)
{
     int i;
     for(i=0;i<n;i++)
          a[i]=rand()%100;
}
void print(int *a,int n)
{
     int i;
     for(i=0;i<n;i++)
          printf("%d  ",a[i]);
}
void find_indexes(int a[],int n,int num)
{
     int i;
     for(i=0;i<n;i++)
          if(a[i]==num)
               printf("%d ",i);
}
void find_max_min(int a[],int n)
{
     int i,max=a[0],min=a[0];
     for(i=0;i<n;i++)
     {
          if(a[i]>max)
               max=a[i];
          if(a[i]<min)
```

```
                min=a[i];
        }
        printf("\nmax = %d\n",max);
        printf("indexes = ");
        find_indexes(a,n,max);
        printf("\nmin = %d\n",min);
        printf("indexes = ");
        find_indexes(a,n,min);
}
main( )
{
      srand(time(0));
      int a[10]={0},n;
      n=sizeof(a)/sizeof(a[0]);
      input(a,n);
      print(a,n);
      find_max_min(a,n);
}
```

```
o/p > 6   68   88   0   77   14   15   17   0   43
       max = 88
       indexes = 2
       min = 0
       indexes = 3 8
```

now observe input and print function's formal arguments in above program. Although base address of a is passed to both input() and print() from main, in input() it is received by int a[] and in print() it is received by int *a. Here a[] is same as *a. But for better understanding that to input function array's base address is passed and not any integer's address, a[] is used instead of *a.

Returning array from functions :

if we want to return array from function, we should return array's base address. But size of array to be returned should be known to calling function as we can return only one variable from function. So if address is returned, size can not be returned along with it.

Consider following example first,

```
<Ex>
#include<stdio.h>
int *f(void)
{
        int v=5;
        printf("in f : &v=%u v=%d\n",&v,v);
        return &v;
}
main()
{
        int *p;
```

```
        p=f();
        printf("in main : p=%u *p=%d\n",p,*p);
}

o/p >  in f : &v=3216201164 v=5
       in main : p=3216201164 *p=5
```

<Q> even if function f()'s stack frame had been collapsed, how main was able to dereference it and get exact value of local variable v of function f()?
--> if function collapses, its stack frame is collapsed but contents that were present in memory portion of that stack frame is not destroyed. They remain as it is until another function is called. If another function is called, then and then old data of that memory portion is over written with new data. Thats why when variable v's address was returned to main, main was able to dereference its exact value.

Now consider another example

<Ex>

```
#include<stdio.h>
void dummy(void)
{
        int a[100]={0};
}
int *f(void)
{
        int v=5;
        printf("in f : &v=%u v=%d\n",&v,v);
        return &v;
}
main()
{
        int *p;
        p=f();
        dummy();
        printf("in main : p=%u *p=%d\n",p,*p);
}

o/p > in f : &v=3219538140 v=5
      in main : p=3219538140 *p=0
```

in this example, after f() function is called, a dummy function is called in which an array of 100 integers is declared and initialized to 0. so after main's stack frame in memory, 400 bytes will be containing 0. so when we try to dereference the value of variable v of f() function, it gives 0 instead of 5.
so, **a safe method of getting function's data after function collapses is declaring function's variables as static. In that case even after function collapses, its variables will remain safe in data section.**

So in program of getting maximum and minimum numbers from array and all minimum number's indexes, if all indexes of minimum number should be known to main, then if more than one indexes are found, all indexes can not be

returned to main. In that case a static array of indexes should be declared and its base address should be passed. Its illustrated below.

<Ex>

```
#include<stdio.h>
int *find_min_index(int a[],int n)
{
        int i,j,min=20;
        static int min_index[8];
        for(i=0;i<8;i++)
                min_index[i]=-1;
        for(i=0,j=0;i<8;i++)
                if(a[i]==min)
                        min_index[j++]=i;
        return (min_index);
}
main()
{
        int a[]={50,20,30,80,20,30,20,90},i,*p,n=8;
        p=find_min_index(a,n);
        printf("min number : %d\n",a[p[0]]);
        printf("min indexes : ");
        for(i=0;p[i]!=-1;i++)
                printf("%d ",p[i]);
}
```

```
o/p > min number : 20
      min indexes : 1 4 6
```

<Assignment> Assign 10 unique random alpha bates to array.

```
#include<stdio.h>
int test_unique(char *p,int i)
{
     int j;
     for(j=0;j<i;j++)
     {
          if(p[i]==p[j])
               return 0;
     }
     return 1;
}
main()
{
     srand(time(0));
     char str[10];
     int i;
     for(i=0;i<10;i++)
     {
          str[i]=rand()%26+65;
          if(!test_unique(str,i))
               i--;
```

```
        }
        for(i=0;i<10;i++)
            printf("%c ",str[i]);
}
```

o/p > S T R F C E P W Y L

<Assignment> Get random 25 number of any character data and count number of alpha bates, numeric data and special characters.

```
#include<stdio.h>
void input(char a[],int n)
{
      int i;
      for(i=0;i<n;i++)
            a[i]=rand()%128;
}
void print(char a[],int n)
{
      int i;
      for(i=0;i<n;i++)
            printf("a[%d]=%c\n",i,a[i]);
}
void count(char a[],int n)
{
      int i,alpha=0,num=0,sp=0;
      for(i=0;i<n;i++)
      {
            if(((a[i]>='a')&&(a[i]<='z'))||
((a[i]>='A')&&(a[i]<='Z')))
                  alpha++;
            else if((a[i]>='0')&&(a[i]<='9'))
                  num++;
            else
                  sp++;
      }
      printf("Alphabates : %d\nNumericals : %d\nSpecial
characters :%d\n",alpha,num,sp);
}
main()
{
      srand(time(0));
      char a[25];
      input(a,25);
      print(a,25);
      count(a,25);
}
```

but in above example, count() function prints counts of alpha bates, numericals and special characters. If we want that these counts should be known in main, then there are three ways.

1.) Create variables in main for all counts and pass by reference to count so that count() can modify values of variables as shown below.

```
void  count(char a[],int n,int *p1,int *p2,int *p3)
{
        int i;
        for(i=0;i<n;i++)
        {
                if(((a[i]>='a')&&(a[i]<='z'))||
((a[i ]>='A')&&(a[i]<='Z')))
                        (*p1)++;
                else if((a[i]>='0')&&(a[i]<='9'))
                        (*p2)++;
                else
                        (*p3)++;
        }
}
main()
{
        int c1=0,c2=0,c3=0;
        srand(time(0));
        char a[25];
        input(a,25);
        print(a,25);
        count(a,25,&c1,&c2,&c3);
        printf("Alphabates : %d\nNumericals : %d\nSpecial
characters :%d\n",c1,c2,c3);
}
```

in above program, observe that to increment values of c1,c2 and c3 pointers are used in count() function and pointer's dereference is incremented `(*p1)++;` without brackets, `*p1++;` only p1 will increment.

2.) create static array of 3 elements in main for three different counts and pass to count() for modification in it.

```
void count(char a[],int n,int cnt[])
{
        int i;
        for(i=0;i<n;i++)
        {
                if(((a[i]>='a')&&(a[i]<='z'))||
((a[i]>='A')&&(a[i]<='Z')))
                        cnt[0]++;
                else if((a[i]>='0')&&(a[i]<='9'))
                        cnt[1]++;
                else
                        cnt[2]++;
        }
}
main()
{
```

```
        static int cnt[3]={0};
        srand(time(0));
        char a[25];
        input(a,25);
        print(a,25);
        count(a,25,cnt);
        printf("Alphabates : %d\nNumericals : %d\nSpecial
characters :%d\n",cnt[0],cnt[1],cnt[2]);
}
```

3.) create static array in count and return its address to main.

```
int *count(char a[],int n)
{
        int i;
        static int cnt[3]={0};
        for(i=0;i<n;i++)
        {
                if(((a[i]>='a')&&(a[i]<='z'))||
((a[i]>='A')&&(a[i]<='Z')))
                        cnt[0]++;
                else if((a[i]>='0')&&(a[i]<='9'))
                        cnt[1]++;
                else
                        cnt[2]++;
        }
        return cnt;
}
main()
{
        int *p;
        srand(time(0));
        char a[25];
        input(a,25);
        print(a,25);
        p=count(a,25);
        printf("Alphabates : %d\nNumericals : %d\nSpecial
characters :%d\n",p[0],p[1],p[2]);
}
```

--> if function is to be declared in program, declaration like,
`void print(char [], int);` is invalid. Because we can not write empty brackets in declarations, we also have to write variable name in declaration. If you don't want to write variable name in declaration, declare function like this,
`void print(char *, int);` `// valid`

NOTE : When pointers are involved, we should not use `a^=b^=a^=b;` kind of statements for swaping data of arrays.

<Ex>

```
int a[5]={10,20,30,40,50};
int b[5];
b=a;              // not valid
printf("%d %d %d %d %d\n",b[0],b[1],b[2],b[3],b[4]);
```

```
o/p > test.c: In function 'main':
      test.c:6:3: error: incompatible types when assigning to type
      'int[5]' from type 'int *'
        b=a;
          ^
```

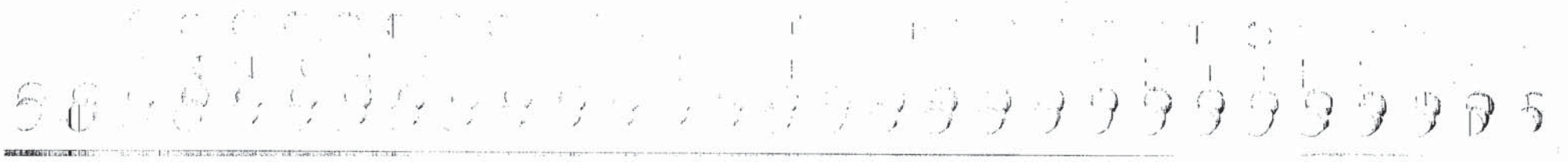
--> array name represents its base address and it is fixed in memory. We can change data of array by traversing using base address of array but we can not change base address of array itself. Because array is an collection of data, so that data can be modified but array itself can not.

```
--> printf("%u\n",a);          // prints base address of array

--> printf("%u\n",++a);
```

```
o/p > test.c: In function 'main':
      test.c:6:16: error: lvalue required as increment operand
        printf("%u\n",++a);
                      ^
```

--> array's usage is like a pointer but array is not a pointer.

--> only method existing to copy one array into another is member by member using a loop.



Strings

Strings : strings are basically an array which is collections of characters.

```
<Ex>
#include<stdio.h>
main()
{
        char s[100]={'a','b','c','d','e'};
        int n,i;
        n=sizeof(s)/sizeof(s[0]);
        for(i=0;i<n;i++)
              printf("%c",s[i]);
        printf("\n");
}
```

now in above example, although we have supplied only 5 characters in string loop will execute 100 times.

Note that when an character array is partially initialized, remaining elements are made NULL by compiler. So in array of 100 characters in above example, compiler only initializes first 5 characters and make remaining elements NULL.

NULL ('\0') is an important character in strings. In all strings , NULL indicates end of string.

So above program can be minimized by using below code.

```
#include<stdio.h>
main()
{
        char s[100]={'a','b','c','d','e'};
        int i;
        for(i=0;s[i];i++)
              printf("%c",s[i]);
        printf("\n");
}
```

but in above program, printf() is still called 5 times. Loop terminates when NULL is found in string. But to print whole string with one printf() call, printf() with format specifier %s should be used. Print() with %s is used for printing strings on screen and stops printing only when NULL is found in string. And having NULL based operations, there is no need of passing number of elements to any function. Same way to printf() also only base address of string is supplied

```
printf("%s\n",s);   //prints all valid characters from location
                      represented by s up to NULL.
```

Characteristics of NULL character :

NULL character assignment,

```
char ch = 0;        // integer constant assignment
char ch = '\0';     // character constant assignment
```

- here both assignments are same and indicated that NULL is assigned to variable ch because all characters are stored in memory by their ASCII values. And NULL character's ASCII value is 0.

<Ex>

```
#include<stdio.h>
main()
{
        char ch;
        ch=0;
        printf("%c",ch);
}
```

o/p > nothing gets printed on screen

NOTE : NULL can not be input from keyboard nor it can be printed on screen.

<Ex>

```
#include<stdio.h>
main()
{
        char s[10]={'a','b','c','d','e'};
        printf("%s\n",s);
        s[3]='\0';              //'a','b','c','NULL','e'
        printf("%s\n",s);
        s[1]=0;                 //'a','NULL','c','NULL','e'
        printf("%s\n",s);
        printf("%s\n",s+1);     //points to s[1], which is NULL
        printf("%s\n",s+2);     //points to s[2] which is 'c'
        printf("%s\n",s+4);     //points to s[4] which is 'e'
}
```

```
o/p > abcde
      abc
      a

      c
      e
```

<Ex>

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

```
        char s[10]={'a','b','c','d','e'};
        char str[10]={'s','=','%','s','\n'};
        printf(str,s);
        printf(str,"xyz");
}
```

```
o/p > s=abcde
      s=xyz
```

<Q> Even though printf's first argument is string, how come printf is able to print data correctly with its first argument as base address of array?

→ in GCC string constants are saved in text section, and represents base address of string present in text section. So when a string constant like "s=%s\n" is passed to printf, it is actually string stored in text section and in program, string constant is represented by its base address of text section. So actually printf()'s first argument is base address of string and not string itself. When base address of string present in text section is given, printf can dereference it and read string from that address and according to format specifiers present in string, prints data on console output. So thats why when we save percentage specifier's string in another array, printf() is able to print outputs correctly.

Conclusion from above discussion : printf() and scanf()'s first argument is address, not string.

We know that in memory, stack is higher and then data section and text section is present. But data section is present above text (code) section. So we can verify that string constants are stored in text section by below code.

<Ex>

```
#include<stdio.h>
main()
{
        int i;
        static int j;
        printf("i : %u\n",&i);
        printf("j : %u\n",&j);
        printf("string constant : %u\n","xyz");
}
```

```
o/p > i : 3215951964
      j : 134520868
      string constant : 134513936
```

variable i is local variable of main() which will be in stack, variable j is static variable which will be present in data section, which is much lower address than stack. String constant's address is lower than address of variable j. It is text section's address.

<Ex>

```
#include<stdio.h>
main()
{
        char s1[10]={'a','b','c','d','e'};
        char s2[10]="chintan";
        printf("s1 : %s\n",s1);
        printf("s2 : %s\n",s2);
}
```

```
o/p > s1 : abcde
      s2 : chintan
```

in above example, s1 and s2 both are of 10 characters. So when s1 is initialized with 5 characters, compiler fills NULL in remaining 5 characters, so printf() works perfectly. In case of s2, it is initialized by string constant. When an character array is initialized with string constant, whole string gets stored in array with NULL at end. So to initialize strings string constants should be used like in case of s2 instead of separate characters used to initialize s1.

Now consider following example,

<Ex>

```
#include<stdio.h>
main()
{
        char s1[5]={'a','b','c','d','e'};
        char s2[20]="chintan";
        char s3[10]="patel";
        printf("s1 : %s\n",s1);
        printf("s2 : %s\n",s2);
}
```

```
o/p > s1 : abcdepatel
      s2 : chintan
```

<Q> Why "patel" also got printed with "abcde" when only s1 string is printed using %s?

→ printf() with %s is a NULL based operation, when we pass string s1's base address to printf, it doesn't know how many characters are present in s1 or what is the size of s1. It only searches for a NULL in s1 and when found, it stops printing. But intentionally we have made string s1 of size 5 and initialized it with 5 characters so there will be no NULL present at the end of s1. In s2 and s3, NULL is guaranteed at lase because size of s2 and s3 is bigger than characters used to initialized it. So there is space for NULL at the end and as they are initialized with string constant, NULL is guaranteed at the end.

Now see base addresses of strings s1, s2 and s3 by using below statements,

```
printf("s1 is at %u\n",s1);
printf("s2 is at %u\n",s2);
printf("s3 is at %u\n",s3);

o/p > s1 is at 3215562521
      s2 is at 3215562536
      s3 is at 3215562526
```

from addresses it is clear that s1 s2 and s3 are present in contagious memory blocks and they are present in order, s1 --> s3 --> s2. Now arrays are present in which order is compiler's choice.

Now when s1 is tried to print with %s, it keeps on searching NULL, but as there is no NULL at last of s1, it also enters in array s3 as they are contagious in memory. So at the end of s3 there is NULL present. So printf() stops printing in encounter of NULL at the end of s3. And then s2 is printed as it is as NULL is present at the end of s2.

Conclusion : in such kind of strings, there should be always one extra space provided for NULL, otherwise unpredictable results may displayed when string is printed by using printf() with %s.

Now consider another example,

<Ex>

```
#include<stdio.h>
main()
{
        char s1[]={'a','b','c','d','e'};
        char s2[]="chintan";
        char *s3="patel";
        printf("%d %d %d\n",sizeof(s1),sizeof(s2),sizeof(s3));
        printf("s1 : %u &s1 : %u\n",s1,&s1);
        printf("s2 : %u &s2 : %u\n",s2,&s2);
        printf("s3 : %u &s3 : %u\n",s3,&s3);
}

o/p > 5 8 4
      s1 : 3216764975 &s1 : 3216764975
      s2 : 3216764980 &s2 : 3216764980
      s3 : 134514128 &s3 : 3216764968
```

→ if size of array is not provide then compiler automatically decides the size when array has been initialized. So in case of s1, no null will be provided at the end so its size is 5 bytes. In s2 null will be provided at its end so its size is 8 bytes. But s3 is pointer, and any pointer in c is of 4 bytes (in 32 bit GCC compiler, in 64 bit GCC it is of 8 bytes). So string will be stored in code section and its address will be supplied to s3.

→ s1, s2 and s3 will be in stack section. s1 and s2 will be containing characters but s3 will be containing text area's address.

Now in above code if we write,

```
s1[0]=s2[0];
s2[0]=s3[0];
printf("%s\n%s\n%s\n",s1,s2,s3);

o/p >  cbcdephintan
       phintan
       patel
```

but if following code is written,

```
s1[0]=s2[0];
s2[0]=s3[0];
s3[0]=s1[0];
printf("%s\n%s\n%s\n",s1,s2,s3);

o/p > segmentation fault
```

its segmentation fault because s1 ad s2 are present in stack and its contents are modifiable by user. But although s3 is present in stack, string pointed by it is present in code section which is read only section. So any attempt to modify anything in read only section will result in segmentation fault.

Getting string inputs from keyboard :

<Ex>

```
#include<stdio.h>
main()
{
        char s[100];
        printf("Enter a string :");
        scanf("%s",&s);
        printf("You have entered : %s\n",s);
}

o/p > Enter a string :chintan patel
      You have entered : chintan
```

here expectation was to input "chintan patel" as whole string in s. But scanf() considers space as data separator by default. So anything separated by space will be considered two different data to scanf.

<Q> So what should be done to input strings with space?

→ there are two ways. We can make scanf to scan string up to new line is supplied. This can be achieved by following statement.

1.) `scanf("%[^\n]s",s);`

here `[^\n]` indicates that scanf keeps on accepting data until new line is supplied.

Similarly, `scanf("%[^.]s",s);`
here `[^.]` indicates that data will be accepted until `'.'` Is not supplied

2.) `gets(s);`

gets() function accepts one string up to new line by default.

When string with scanf() and gets() string is entered and enter is hit, it puts entered in given location but new line is not put, instead NULL is placed at the last.

Finding length of string & size of string :

<Ex>

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[100];
        printf("Enter a string :");
        gets(s);
        printf("size of string : %d\n",sizeof(s));
        printf("length of string : %d\n",strlen(s));
}
```

```
o/p > Enter a string :chintan
      size of string : 100
      length of string : 7
```

sizeof(s) - gives total size allocated in memory for s.
strlen(s) - gives total characters present in string s without NULL.

strlen() :

→ declaration :

```
size_t strlen(const char *s);
```

→ The strlen() function calculates the length of the string s, excluding the terminating null byte ('\0').

→ strlen() user defined :

```
size_t Ustrlen(const char *p)
{
        size_t count=0;
        int i;
        for(i=0;p[i];i++)
                count++;
        return count;
}
```

<Ex>

```
#include<stdio.h>
#include<string.h>
main( )
{
        char s[10];
        printf("Enter a string :");
        gets(s);
        printf("Entered string : %s\n",s);
}
```

```
o/p >  Enter a string :hello friends, how are you?
       Entered string : hello friends, how are you?
       *** stack smashing detected ***: ./a.out terminated
       Aborted (core dumped)
```

if like shown above, if we enter data more than size of the string, then scanf() of gets() doesn't report error about it because size of array is now known to them. Only argument supplied to both function is starting address of string from where scanf() or gets() will start storing entered data.

If we supply more data than size of string, then scanf() or gets() will try to store it beyond allocated memory of string where it is not allowed to do so, so a stack smashing will be detected by OS and it will report the error and abort the application execution.

<Ex>

```
#include<stdio.h>
main()
{
        char str[10]="chintan";
        char *p="patel";
        str=p;
        printf("%s\n",str);
}
```

```
o/p > test.c: In function 'main':
      test.c:6:5: error: incompatible types when assigning to type
      'char[10]' from type 'char *'
        str=p;
            ^
```

→ if in above program, in place of str=p; , p=str; would had been written then it is completely valid statement, its because p is a pointer, it is initialized by the base address of string present in text section. But later base address of string str is assigned to it. Although we loose track of string present in text section and will never able to use it again in our program, but it is syntactically valid.

<Assignment> WAP to declare an array of 100 characters and input a string and do following
(1). input a character and search number of occurrences.

(2). reverse a string and print it.
(3). convert all lower case to upper case.

1) input a char and find no. Of occurrences

```
#include<stdio.h>
main()
{
        char str[100],ch;
        int i,cnt=0;;
        printf("Enter a string : ");
        gets(str);
        printf("Input a character to search : ");
        scanf("%c",&ch);
        for(i=0;str[i];i++)
        {
                if(str[i]==ch)
                        cnt++;
        }
        printf("character %c occurs %d times\n",ch,cnt);
}
```

```
o/p > Enter a string : chintan patel
      Input a character to search : a
      character a occurs 2 times
```

2) reverse string and print it

```
#include<stdio.h>
main()
{
        char str[100],ch;
        int i,cnt=0;
        printf("Enter a string : ");
        gets(str);
        for(i=0;str[i];i++);        //find length of string
        cnt=i;
        printf("%d\n",cnt);
        for(i=0;i<=cnt/2;i++)
        {
                ch=str[i];
                str[i]=str[cnt-i-1];
                str[cnt-i-1]=ch;
        }
        printf("reversed string : %s\n",str);
}
```

3) convert all lower case to upper case

```
#include<stdio.h>
main()
{
        char str[100];
```

```
        int i;
        printf("Enter a string : ");
        gets(str);
        for(i=0;str[i];i++)
        {
                if((str[i]>='a')&&(str[i]<='z'))
                        str[i]-=32;
        }
        printf("string : %s\n",str);
}
```

```
o/p > Enter a string : ChInTan PatEl
      string : CHINTAN PATEL
```

String Copy :

for copying one string into another, there are two ways.
1). copying member by member of one string into another
2). by using library function strcpy()

1) member by member transfer

```
#include<stdio.h>
main()
{
        char s1[100],s2[100];
        int i;
        printf("Enter string for s1 : ");
        gets(s1);
        for(i=0;s1[i];i++)
                s2[i]=s1[i];
        s2[i]=0;
        printf("s2 : %s\n",s2);
}
```

2) using built in function strcpy()

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100];
        printf("Enter string s1 : ");
        gets(s1);
        strcpy(s2,s1);
        printf("s2 : %s\n",s2);
}
```

Function strcpy() declaration is present in string.h header file. So in order to use it string.h header file should be included in our program.

→ strcpy(s2,s1); // copies all character from s1 into s2 including the null character

strcpy() :

→ **declaration :**

```
char *strcpy(char *dest, const char *src);
```

<Ex>

```
strcpy(s1,"abcdefgh");   // s1 : abcdefgh
strcpy(s2,s1+3);         // s2 : defgh
strcpy(s1,s2+1);         // s1 : efgh
```

<Ex>

```
strcpy(s1,"abc");              // s1 : abc
strcpy(s2,"01234");            // s2 : 01234
strcpy(s1+strlen(s1),s2);      // s1 : abc01234
strcpy(s2+strlen(s2),s1);      // s2 : 01234abc01234
```

<Ex>

```
strcpy(s1,"abcdefgh");   // s1 : abcdefgh\0
strcpy(s1,s1+2);         // s1 : cdefgh\0h\0
strcpy(s1+1,s1+2);       // s1 : cefgh\0\0h\0
printf("s1 : %s\n",s1); // s1 : cefgh
```

<Ex>

```
strcpy(s1,"abcdefgh");
strcpy(s1+4,s1+3);
```

→ in this kind of operation, unpredicted behavior may be seen, strcpy() stops copying elements only after it copies null from source. But in above example, null is not copied from source to destination. So it doesn't stop until it reaches out of the array size and results in segmentation fault.

→ return : return a pointer to the destination string

→ strcpy() should not be used when source and destination address are present in overlapping memory area. e.g. are in same string.

→ **strcpy() user defined :**

```
char *Ustrcpy(char *dest,const char *src)
{
        int i;
        for(i=0;src[i];i++)
                dest[i]=src[i];
        dest[i]=0;
        return dest;
}
```

<Q> What is size_t?

→ This is an unsigned integer type used to represent the sizes of objects. The result of the sizeof operator is of this type. On systems using the GNU C Library, this will be `unsigned int` or `unsigned long int`.
Usage : size_t is preferred to declare any arguments or variables that holds sizes of objects

strncpy() :

→ declaration :

```
char *strncpy(char *dest, const char *src, size_t n);
```

<Ex>

```
strcpy(s1,"abcdefgh");
strcpy(s2,"012345678");
strncpy(s2+2,s1+3,3);     // s2 : 01def5678
```

→ If there is no null byte among the first n bytes of source, the string placed in destination will not be null-terminated. If the length of source is less than n, strncpy() writes additional null bytes to destination to ensure that a total of n bytes are written.

→ One valid (and intended) use of strncpy() is to copy a C string to a fixed length buffer while ensuring both that the buffer is not over flowed and that unused bytes in the target buffer are zeroed out.

→ If there is no terminating null byte in the first n bytes of src, strncpy() produces an unterminated string in dest. You can force termination using something like the following:

```
strncpy(dest, str, n);
if (n > 0)
        dest[n - 1]= '\0';
```

→ return : return a pointer to the destination string

→ strncpy() user defined :

```
char * Ustrncpy(char *dest, const char *src, size_t n)
{
      int i;

      for (i=0;i<n && src[i];i++)
            dest[i]=src[i];
      for (;i<n;i++)
            dest[i]='\0';
      return dest;
}
```

memmove() :

→ declaration :

```
void *memmove(void *dest, const void *src, size_t n);
```

→ memmove requires third argument as how many bytes to be copied, same as in strncpy().

→ for copying strings in overlapping areas, memmove() function is used.

→ using memmove to copy string, bytes in source are first copied into a temporary array that does not overlap source or destination, and the bytes are then copied from the temporary array to destination.

<Ex>
```
strcpy(s1,"abcdefgh");
memmove(s1+4,s1+3,4);
printf("s1 : %s\n",s1);

o/p >  s1 : abcddefg
```

→ **memmove() user defined :**
```
void *Umemmove(void *dest,const void *src,size_t n)
{
        char temp[n+1];
        int i;
        for(i=0;i<n;i++)
                temp[i]=((char *)src)[i];
        for(i=0;i<n;i++)
                ((char *)dest)[i]=temp[i];
        return dest;
}
```

→ return : returns a pointer to destination

strcat() :

→ **declaration :**
```
char *strcat(char *dest, const char *src);
```

→ The strcat() function appends the src string to the dest string, overwriting the terminating null byte ('\0') at the end of dest, and then adds a terminating null byte. The strings may not overlap, and the dest string must have enough space for the result. If dest is not large enough, program behavior is unpredictable. As with strcat(), the resulting string in dest is always null terminated.

<Ex>
```
strcpy(s1,"abcdefgh");
strcat(s1,"xyz");              // s1 : abcdefghxyz
```

→ in above example, equivalent statement like `strcat(s1,"xyz");` can be also implemented using strcpy as shown below
```
strcpy(s1+strlen(s1),"xyz");
```

<Ex>
```
strcpy(s1,"xyz");
srtcat("abc",xyz");
strcat(s1,"123");
```

→ no compile time error is reported in this program because strcat()'s arguments are addresses. So addresses of strings of text section is passed to strcat(). So strcat() will try to modify data in text section which is prohibited so segmentation fault will be reported.

<Ex>
```
strcpy(s1,"abc");
strcat(s1+1,"123");       // s1 : abc123
```

<Ex>
```
strcpy(s1,"abc";
strcat(s1+5,"123");       // s1 : abc
```

→ will search for null from s1+5 location from which null is not guaranteed to be present, we can get segmentation fault if null is not found in the user space memory.

→ **strcat() user defined :**
```
char *Ustrcat(char *dest, const char *src)
{
        int len=strlen(dest);
        int i;
        for (i=0;src[i];i++)
                dest[len+i]=src[i];
        dest[len+i]='\0';
        return dest;
}
```

→ return : pointer to the destination string

strncat() :

→ **declaration :**
```
char *strncat(char *dest, const char *src, size_t n);
```

→ The strncat() function is similar to strcat(), except that it will use at most n bytes from source; and source does not need to be null-terminated if it contains n or more bytes. If source contains n or more bytes, strncat() writes n+1 bytes to destination (n from source plus the terminating null byte). Therefore, the size of destination must be at least strlen(destination)+n+1.

→ **strncat() user defined :**
```
char *strncat(char *dest, const char *src, size_t n)
```

```
{
        int len=strlen(dest);
        int i;
        for (i=0;i<n && src[i];i++)
                dest[len+i]=src[i];
        dest[len+i]='\0';
              return dest;
}
```

→ return : returns pointer to the resulting destination string.

<Assignment> Input 3 strings and store in s1,s2,s3. Copy all the 3 strings into s4, one after another separated by space.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[10],s2[10],s3[10],s4[10];
        printf("Enter s1 : ");
        gets(s1);
        printf("Enter s2 : ");
        gets(s2);
        printf("Enter s3 : ");
        gets(s3);
        strcpy(s4,s1);
        strcat(s4," ");
        strcat(s4,s2);
        strcat(s4," ");
        strcat(s4,s3);
        printf("String s4 : %s\n",s4);
}
```

```
o/p > Enter s1 : abc
      Enter s2 : xyz
      Enter s3 : 123
      String s4 : abc xyz 123
```

<Assignment> input 2 strings s1 and s2. Insert s2 in given position in s1.
1). using temp array
2). without using temp array

```
1) using temp array
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],s2[20],temp[20];
        int pos;
```

```
        printf("Enter s1 : ");
        gets(s1);
        printf("Enter s2 : ");
        gets(s2);
        printf("Enter position : ");
        scanf("%d",&pos);
        strcpy(temp,s1+pos);
        strcpy(s1+pos,s2);
        strcat(s1,temp);
        printf("modified s1 : %s\n",s1);
}
```

2) without using temp array

```
        memmove(s1+pos+strlen(s2),s1+pos,strlen(s1+pos)+1);
        memmove(s1+pos,s2,strlen(s2));
        printf("modified s1 : %s\n",s1);
```

```
o/p > Enter s1 : chintan
      Enter s2 : patel
      Enter position : 3
      modified s1 : chipatelntan
```

<Assignment> WAP to input a string and remove all non alpha bates.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],s2[20];
        int i,len;
        printf("Enter s1 : ");
        gets(s1);
        for(i=0;s1[i];i++)
        {
                if(((s1[i]>='a')&&(s1[i]<='z'))||
((s1[i]>='A')&&(s1[i]<='Z')))
                        continue;
                else
                {
                        memmove(s1+i,s1+i+1,strlen(s1+i));
                        i--;
                }
        }
        printf("modified s1 : %s\n",s1);
}
```

```
o/p > Enter s1 : c7h&i.n,t5a) n
      modified s1 : chintan
```

<Assignment> WAP to input a string and a character and remove all occurrences of the character from the string.

```
#include<stdio.h>
#include<string.h>
main( )
{
        char s1[20],ch;
        int i;
        printf("Enter s1 : ");
        gets(s1);
        printf("Enter character : ");
        scanf("%c",&ch);
        for(i=0;s1[i];i++)
        {
                if(s1[i]==ch)
                {
                        memmove(s1+i,s1+i+1,strlen(s1+i+1)+1);
                        i--;
                }
        }
        printf("modified s1 : %s\n",s1);
}
```

```
o/p > Enter s1 : embedded
      Enter character : e
      modified s1 : mbddd
```

→ instead of checking every character in string one by one in our string to search for a particular character in a string, there is an inbuilt function available : strchr()

strchr() :

→ **declaration :**

```
char *strchr(const char *s, int c);
```

→ The strchr() function returns a pointer to the first occurrence of the character c in the string s. Here "character" means "byte"; these functions do not work with wide or multi byte characters.

→ The terminating null byte is considered part of the string, so that if c is specified as '\0', these functions return a pointer to the terminator.

<Ex>

```
char s[20]="chintan",ch='i';
if(strchr(s,ch)==NULL)
        printf("Not found\n");
```

```
else
        printf("Found at index %u\n",strchr(s,ch)-s);

o/p > Found at index 2
```

→ but here calling strchr is called every time whenever location of the character 'i' in string "chintan" is needed, instead of that, we can collect return of strchr in a character pointer and use that pointer when needed as shown below.

<Ex>

```
char s[20]="chintan",ch='i',*p;
p=strchr(s,ch);
if(p==NULL)
        printf("Not found\n");
else
        printf("Found at index %u\n",p-s);

o/p > Found at index 2
```

→ strchr() user defined :

```
char *Ustrchr(const char *s,int c)
{
        int i;
        for(i=0;s[i];i++)
        {
                if(s[i]==c)
                        return s+i;
        }
        return NULL;
}
```

→ Note that we are passing character to be searched to strchr() but in function data passing, character's ASCII values are passed so characters are promoted to integer. Thats why in formal arguments of strchr(), int is used instead of char.

→ **return :** The strchr() function return a pointer to the first occurrence of matched character or NULL if the character is not found.

<Assignment> WAP to input a string and a character and remove all occurrences of the character from the string.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],ch,*p;
        int i;
        printf("Enter s1 : ");
```

```
        gets(s1);
        printf("Enter character : ");
        scanf("%c",&ch);
        while(p=strchr(s1,p))
                memmove(p,p+1,strlen(p+1)+1);
        printf("modified s1 : %s\n",s1);
}
```

<Assignment> Remove extra repetitions of all characters in a given string.

```
#include<stdio.h>
#include<string.h>
main( )
{
        char s[20],*p;
        int i;
        printf("Enter string : ");
        gets(s);
        for(i=0;s[i];i++)
        {
                while(p=strchr(s+i+1,s[i]))
                        memmove(p,p+1,strlen(p+1)+1);
        }
        puts(s);
}
```

```
o/p > Enter string : embedded
      Modified string : embd
```

<Assignment> WAP to input a string and find out how many characters are repeated.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[20],*p,temp[20];
        int i,cnt=0;
        printf("Enter string : ");
        gets(s);
        strcpy(temp,s);
        for(i=0;temp[i];i++)
        {
                if(strchr(temp+i+1,temp[i]))
                {
                        cnt++;
                        while(p=strchr(temp+i+1,temp[i]))
                                memmove(p,p+1,strlen(p+1)+1);
                }
        }
        printf("No. of characters repeating : %d\n",cnt);
}
```

```
o/p > Enter string : embedded
      No. of characters repeating : 2
```

strrchr() :

→ declaration :

```
char *strrchr(const char *s, int c);
```

→ The strrchr() function returns a pointer to the last occurrence of the character c in the string s.

→ strrchr() user defined :

```
char *Ustrrchr(const char *s,int c)
{
        int i,l=-1;
        for(i=0;s[i];i++)
        {
                if(s[i]==c)
                        l=i;
        }
        if(l>=0)
                return s+l;
        else
                return NULL;
}
```

→ **return** : The strchr() function return a pointer to the last occurrence of matched character or NULL if the character is not found.

strstr() :

→ declaration :

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

→ The strstr() function finds the first occurrence of the substring needle in the string haystack. The terminating null bytes ('\0') are not compared.

<Ex>

```
char s1[20]="abcabcdefbcdebcd",s2[20]="bcd",*p;
int i;
p=strstr(s1,s2);
if(p)
        printf("Found at %u\n",p-s1);
else
        printf("Not found\n");
```

```
o/p > Found at 4
```

<Ex>

```
char s1[20]="abcabcdefbcdebcd",s2[20]="bcd",*p;
int i,cnt=0;
p=s1;
while(p=strstr(p,s2))
{
        cnt++;
        p++;
}
printf("String s2 occures %d times in s1\n",cnt);
```

o/p > String s2 occures 3 times in s1

→ **strstr() user defined :**

```
char *Ustrstr(const char *haystack,const char *needle)
{
        int i,j;
        for(i=0;haystack[i];i++)
        {
                if(haystack[i]==needle[0])
                {
                        for(j=1;needle[j];j++)
                        {
                                if(haystack[i+j]!=needle[j])
                                        break;
                        }
                        if(needle[j]=='\0')
                                return haystack+i;
                }
        }
        return NULL;
}
```

→ **return :** return a pointer to the beginning of the substring, or NULL if the substring is not found.

<Assignment> remove all occurrences of s2 from s1.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        while(p=strstr(s1,s2))
                memmove(p,p+strlen(s2),strlen(p+strlen(s2))+1);
        printf("s1 : %s\n",s1);
```

```
o/p > Enter string s1 : chintanpatelchintan
      Enter string s2 : patel
      s1 : chintanchintan
```

<Assignment> Reverse s2 in s1

```
#include<stdio.h>
#include<string.h>
main( )
{
        char s1[100],s2[100],*p,ch;
        int i,j;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        while(p=strstr(s1,s2))
        {
                j=strlen(s2)-1;
                for(i=0;i<j;i++,j--)
                {
                        ch=*(p+i);
                        *(p+i)=*(p+j);
                        *(p+j)=ch;
                }
        }
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintanpatel chintanpatel
      Enter string s2 : patel
      s1 : chintanletap chintanletap
```

<Assignment> Remove extra consecutive blank spaces.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[100],*p,*q,ch=32;
        printf("Enter string : ");
        gets(s);
        p=s;
        while(p=strchr(p,ch))
        {
                q=++p;
                while(*p==ch)
                        p++;
                memmove(q,p,strlen(p)+1);
        }
        printf("s : %s\n",s);
}
```

```
o/p >  Enter string : chintan     patel     patel
       s  : chintan patel patel
```

<Assignment> Hide s2 in s1

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p;
        int i;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        while(p=strstr(s1,s2))
        {
                j=strlen(s2);
                for(i=0;i<j;i++)
                        s1[(p-s1)+i]='*';
        }
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintanpatelchintan
      Enter string s2 : patel
      s1 : chintan*****chintan
```

→ this assignment can also be done by built in function memset().

<Assignment> Hide s2 in s1

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        p=s1;
        while(p=strstr(p,s2))
        {
                memset(p,'*',strlen(s2));
                p++;
        }
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintan patel chintan
```

```
Enter string s2 : patel
s1 : chintan ***** chintan
```

memset() :

→ **declaration :**

```
void *memset(void *s, int c, size_t n);
```

→ The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.

→ **memset() user defined :**

```
void *Umemset(void *s,int c,size_t n)
{
        while(n--)
        {
                *(char *)s=c;
                ((char *)s)++;
        }
        return s;
}
```

→ return : returns pointer to memory area s.

strcmp() :

→ **declaration :**

```
int strcmp(const char *s1, const char *s2);
```

<Ex>

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100];
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        if(strcmp(s1,s2)==0)
                printf("Equal\n");
        else
                printf("Not Equal\n");
}
```

```
o/p > Enter string s1 : chintan
      Enter string s2 : patel
      Not Equal
```

→ The strcmp() function compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

→ strcmp() user defined :

```
int Ustrcmp(const char *s1,const char *s2)
{
        while(*s1 && *s2)
        {
                if(*s1 != *s2)
                        break;
                s1++;
                s2++;
        }
        if(*s1 == *s2)
                return 0;
        else
        {
                if(*s1 > *s2)
                        return 1;
                else
                        return -1;
        }
}
```

→ return : returns an integer less than, equal to or greater than zero, if s1 is found respectively to be less than, to match or to be greater than s2.

strncpy() :

→ declaration :

```
int strncmp(const char *s1, const char *s2, size_t n);
```

→ The strncmp() function is similar, except it compares the only first (at most) n bytes of s1 and s2.

→ return : returns an integer less than, equal to or greater than zero, if fisrt n bytes of s1 are found respectively to be less than, to match or to be greater than first n bytes of s2.

<**Assignment**> reverse given string

```
#include<stdio.h>
#include<string.h>
void strrev(char *p)
{
        int n=strlen(p);
        char *q,temp;
        for(q=p+n-1;p<q;p++,q--)
```

```
        {
                temp=*p;
                *p=*q;
                *q=temp;
        }
}
main()
{
        char s[100];
        printf("Enter string : ");
        gets(s);
        strrev(s);
        printf("s : %s\n",s);
}

o/p > Enter string : chintan
      s : natnihc
```

<Assignment> implement your own strrstr()

```
#include<stdio.h>
char * Ustrrstr(char s1[], char s2[])
{
        int i,j;
        char *p=NULL;
        for(i=0;s1[i];i++,j=1)
        {
                if(s1[i]==s2[0])
                {
                        for(j=1;s2[j];j++)
                        {
                                if(s1[i+j]!=s2[j])
                                        break;
                        }
                        if(s2[j]=='\0')
                                p=s1+i;
                }
        }
        if(p)
                return p;
        else
                return NULL;
}
main()
{
        char s1[20],s2[20],*p;
        puts("enter s1:");
        gets(s1);
        puts("enter s2:");
        gets(s2);
        p=Ustrrstr(s1,s2);
```

```
        if(p)
        {
                printf("found at index=%d\n",p-s1);
        }
        else
                printf("not found\n");
}
```

File based program

1) Read file data.
   --> ./a.out file_name

2) Write some information into file.
   --> ./a.out file_name

3) Implement the copy command.
   --> ./a.out file1_name file2_name

4) Count the number of occurance the string in a file.
   --> ./a.out file_name string

5) Find the string in file and hide.
   --> ./a.out file_name string

6) Replace the one string to other string in file.
   --> ./a.out file_name str1 str2

7) Implement the grep command.
   --> ./a.out file_name string

8) Count the number of words,charactor,line -->> wc command.
   --> ./a.out file_name

9) In main file odd number of word store in one file and even number of word store in other file.
   --> ./a.out file1 file2(odd) file3(even)

10) In main file odd number of line store in one file and even number of line store in other file.
   --> ./a.out file1 file2(odd) file3(even)

11) In file size of line is must be more than 5 byte consider otherwise this line can be remove and copy to other file.
   --> ./a.out file1_name file2_name

12) Merge the n number of file.
--> ./a.out file1 file2 file3.....filen

13) Implement sort command.
--> ./a.out file_name

14) Input two file name and also enter the integer than after the program should insert the contain of f2 in given position in f1.
--> ./a.out file1 file2 position

1) read the file data

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);

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

main(int n,char **p)
{
      FILE *fp;
      char ch;

      if(n != 2)
      {
            printf("Error ./a.out file_name\n");
            return;
      }

      fp=fopen(p[1],"r");

      if(fp==NULL)
      {
            printf("Error ./a.out file_name\n");
            return;
      }

      while((ch=fgetc(fp)) != EOF)
      {
            printf("%c",ch);
      }

      fclose(fp);
}
```

2) write some information into the file

// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);

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

main(int n,char **p)
{
      FILE *fp;
      char ch;

      if(n != 2)
      {
            printf("Error ./a.out file_name\n");
            return;
      }

      fp=fopen(p[1],"w");

      printf("Enter the data into the file: and last (ctrl + d) \n");

      while((ch=getchar()) != EOF)
      {
            fputc(ch,fp);
      }

      fclose(fp);
}
```

3) implement copy command

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);

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

void *copytobuf(char *,int);

main(int n,char **p)
{
      char *buf;
      int i;
      buf=copytobuf(p[1],n);
      puts(buf);

      FILE *fp;
      fp=fopen(p[2],"w");

      fputs(buf,fp);
      fclose(fp);
}
void *copytobuf(char *name,int n)
{
      FILE *fp;
      char *buf;
      int size;

      if(n!=3)
      {
            printf("error ./a.out file_name file_name\n");
            exit(0);
      }

      fp=fopen(name,"r");

      if(fp==NULL)
      {
            printf("Error\n");
            exit(0);
      }

      fseek(fp,0,2);
```

```
    size=ftell(fp)+1;
    fseek(fp,0,0);

    buf=calloc(1,size);

    fread(buf,size-1,1,fp);

    fclose(fp);
    return buf;
}
```

4) count the number of occurance the string in a file.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);

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

void *copytobuf(char *,int);

main(int n,char **p)
{
     char *buf,*ptr;

     buf=copytobuf(p[1],n);
     //puts(buf);

     int cnt=0;

     ptr=buf;

     while(ptr=strstr(ptr,p[2]))
     {
          cnt++;
          ptr=ptr+strlen(p[2]);
     }
     printf("cnt = %d \n",cnt);
}
void *copytobuf(char *name,int n)
{
     FILE *fp;
     char *buf;
     int size;

     if(n!=3)
     {
          printf("error ./a.out file_name file_name\n");
          exit(0);
     }

     fp=fopen(name,"r");

     if(fp==NULL)
```

```
    {
        printf("Error\n");
        exit(0);
    }

    fseek(fp,0,2);
    size=ftell(fp)+1;
    fseek(fp,0,0);

    buf=calloc(1,size);

    fread(buf,size-1,1,fp);

    fclose(fp);
    return buf;
}
```

5) find the string in the file and hide it.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);

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

void *copytobuf(char *,int);

main(int n,char **p)
{
      char *buf,*ptr;
      FILE *fp;
      buf=copytobuf(p[1],n);
      puts(buf);

      ptr=buf;

      while(ptr=strstr(buf,p[2]))
      {
            memset(ptr,'*',strlen(p[2]));
            ptr=ptr+strlen(p[2]);
      }
      puts(buf);
      fp=fopen(p[1],"w");
      fputs(buf,fp);
      fclose(fp);
}
void *copytobuf(char *name,int n)
{
      FILE **fp;
      char *buf;
      int size;

      if(n!=3)
      {
            printf("error ./a.out file_name string\n");
            exit(0);
      }

      fp=fopen(name,"r");
```

```
    if(fp==NULL)
    {
        printf("Error\n");
        exit(0);
    }

    fseek(fp,0,2);
    size=ftell(fp)+1;
    fseek(fp,0,0);

    buf=calloc(1,size);

    fread(buf,size-1,1,fp);

    fclose(fp);
    return buf;
}
```

6) replace the one string to other string in file.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);

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

void *copytobuf(char *,int);
void *replace(char *,char *,char *,char *);
main(int n,char **p)
{
     char *buf;
     FILE *fp;

     buf=copytobuf(p[1],n);
     puts(buf);

     buf=replace(buf,p[1],p[2],p[3]);
     puts(buf);

     fp=fopen(p[1],"w");
     fputs(buf,fp);
     fclose(fp);
}

void *replace(char *buf,char *name,char *s1,char *s2)
{
     char *ptr;

     int l1,l2;
     l1=strlen(s1);
     l2=strlen(s2);
//   puts(buf);
     ptr=buf;

     while(ptr=strstr(buf,s1))
     {
       memmove(ptr,ptr+l1,strlen(ptr+l1)+1);
     //   puts(buf);
         buf=realloc(buf,strlen(buf)+l2+1);
         memmove(ptr+l2,ptr,strlen(ptr)+1);
     //   puts(buf);
```

```
            strncpy(ptr,s2,l2);
            ptr=ptr+strlen(s1);
        }
//      puts(buf);
        return buf;
}
void *copytobuf(char *name,int n)
{
        FILE *fp;
        char *buf;
        int size;

        if(n!=4)
        {
                printf("error ./a.out file_name str1 str2\n");
                exit(0);
        }

        fp=fopen(name,"r");

        if(fp==NULL)
        {
                printf("Error\n");
                exit(0);
        }

        fseek(fp,0,2);
        size=ftell(fp)+1;
        fseek(fp,0,0);

        buf=calloc(1,size);

        fread(buf,size-1,1,fp);

        fclose(fp);
        return buf;
}
```

7) implement the grep command.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);
// char *fgets(char *ptr,int size,FILE *fp);

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

main(int n,char **p)
{
      FILE *fp;
      char str[100];
      int cnt=0;

      if(n != 3)
      {
            printf("error ./a.out file_name string\n");
            return;
      }

      fp=fopen(p[1],"r");

      while(fgets(str,100,fp))
      {
            cnt++;
            if(strstr(str,p[2]))
            {
                  printf("%d %s\n",cnt,str);
            }
      }
      fclose(fp);
}
```

8) Count the number of line,charactor and words --->> WC command.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);
// char *fgets(char *ptr,int size,FILE *fp);

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

main(int n,char **p)
{
     FILE *fp;
     char ch,str[100];
     int chr=0,word=0,line=0;

     if(n != 2)
     {
          printf("error ./a.out file_name\n");
          return;
     }

     fp=fopen(p[1],"r");
     if(fp==NULL)
     {
          printf("file does not exist\n");
          return;
     }

     while((ch=fgetc(fp)) != EOF)
     {
          chr++;
          if(ch == '\n')
          {
               line++;
          }
     }

     fseek(fp,0,0);
     while(fscanf(fp,"%s",str) != EOF)
     word++;

     printf("charctor:%d\n",chr);
```

```
        printf("line:%d\n",line);
        printf("word:%d\n",word);
        fclose(fp);
}
```

9) In main file odd number of word store in one file and even number of word store in other file.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);
// char *fgets(char *ptr,int size,FILE *fp);

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

main(int n,char **p)
{
     FILE *fp1,*fp2,*fp3;
     char ch,str[100];
     int cnt=0;

     if(n != 4)
     {
          printf("error ./a.out file1 file2 file3\n");
          return;
     }

     fp1=fopen(p[1],"r");
     fp2=fopen(p[2],"w");
     fp3=fopen(p[3],"w");

     if(fp1==NULL)
     {
          printf("file does not exist\n");
          return;
     }

     while(fscanf(fp1,"%s ",str) != EOF)
     {
          cnt++;
          if(cnt%2==0)
          fprintf(fp2,"%s ",str);
          else
          fprintf(fp3,"%s ",str);
     }
     fclose(fp1);
     fclose(fp2);
     fclose(fp3);
```

10) In main file odd number of line store in one file and even number of line store in other file.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);
// char *fgets(char *ptr,int size,FILE *fp);

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

main(int n,char **p)
{
      FILE *fp1,*fp2,*fp3;
      char str[100];
      int cnt=0;

      if(n != 4)
      {
            printf("error ./a.out file1 file2 file3\n");
            return;
      }

      fp1=fopen(p[1],"r");
      fp2=fopen(p[2],"w");
      fp3=fopen(p[3],"w");

      if(fp1==NULL)
      {
            printf("file does not exist\n");
            return;
      }

      while(fgets(str,100,fp1))
      {
            cnt++;
            if(cnt%2==0)
            fprintf(fp2,"%s",str);
            else
            fprintf(fp3,"%s",str);
      }
      fclose(fp1);
      fclose(fp2);
      fclose(fp3);
```

11) In main file size of line is must be more than 5 byte otherwise this line can be remove and copy to other file.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);
// char *fgets(char *ptr,int size,FILE *fp);

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

main(int n,char **p)
{
     FILE *fp1,*fp2;
     char str[100];
     int i,cnt=0;

     if(n != 3)
     {
          printf("error ./a.out file1 file2\n");
          return;
     }

     fp1=fopen(p[1],"r");
     fp2=fopen(p[2],"w");

     if(fp1==NULL)
     {
          printf("file does not exist\n");
          return;
     }

     while(fgets(str,100,fp1))
     {
          for(i=0;str[i];i++);
          if(i>5)
          {
               fprintf(fp2,"%s",str);
          }
     }
     fclose(fp1);
     fclose(fp2);
```

}

12) mearge the n number of file.

```
// int fgetc(FILE *fp);
// int fputc(int ch,FILE *fp);
// int fputs(const char *ptr,FILE *fp);
// int fread(void *buf,int size,int n,FILE *fp);
// char *fgets(char *ptr,int size,FILE *fp);

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

main(int n,char **p)
{
      FILE *fp1,*fp2;
      char str[100];
      int i;

      for(i=1;i<n-1;i++)
      {
            fp1=fopen(p[i],"r");
            fp2=fopen(p[i+1],"w");

            while(fgets(str,100,fp1))
            {
                  fprintf(fp2,"%s",str);
            }
            fclose(fp1);
            fclose(fp2);
      }
}
```

13) Implement sort command.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void  *buf_copy(char*,int);

main( int n,char **p)
{
      char *buf,temp[100];
      FILE *fp;
      int cnt=0,j,i=0,size=0;
      char (*q)[20]=NULL;

      fp=fopen(p[1],"r");

      buf=buf_copy(p[1],n);
      puts(buf);

/*************** count the line in file ***************/

      printf(">>>>>>>>>>>>\n");
      while(fgets(temp,100,fp))
      {
            q=realloc(q,(cnt+1)*20);
            strcpy(q[i],temp);
            printf("%s",q[i]);
            cnt++;
            i++;
      }

/*********************  sort *************************/

      for(i=cnt-1;i>0;i--)
      {
            for(j=0;j<i;j++)
            {
                  if(strcmp(q[j],q[j+1])>0)
                  {
                        strcpy(temp,q[j]);
                        strcpy(q[j],q[j+1]);
                        strcpy(q[j+1],temp);
                  }
            }
      }
      for(i=0;i<cnt;i++)
```

```
        printf("%s",q[i]);

        fclose(fp);
}
void  *buf_copy(char *name,int n)
{
        int size;
        char *buf;
        FILE *fp;

        if(n!=2)
        {
                printf("Error\n");
                return;
        }

        fp=fopen(name,"r");

        if(fp==NULL)
        {
                printf("Error wr--->>>./a.out <file_name>\n");
                return;
        }

/*******************  Find the Size ******************/

        fseek(fp,0,2);
        size=ftell(fp)+1;
        fseek(fp,0,0);
        //rewind(fp);

/************ allocate the size to buffer *****************/

        buf=calloc(1,size);

/*************** copy file into the buffer ****************/

        fread(buf,size-1,1,fp);
        fclose(fp);
        return(buf);
}
```

14) Input two file name and also enter the position than after the program should insert the contain of f2 int given position in f1.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void  *buf_copy(char*,int);

main(int n,char **p)
{
      char *buf1,*buf2,temp[100];
      FILE *fp;
      int a,d,l1,l2;

      buf1=buf_copy(p[1],n);
      buf2=buf_copy(p[2],n);

      a=atoi(p[3]);

      l1=strlen(buf1);
      l2=strlen(buf2);

      buf1=realloc(buf1,l1+l2+1);

      memmove(buf1+a+l2,buf1+a,strlen(buf1+a)+1);
      memmove(buf1+a,buf2,l2);

      fp=fopen(p[1],"w");
      fputs(buf1,fp);

      fclose(fp);
}

void *buf_copy(char *name,int n)
{
      char *buf;
      FILE *fp;
      int size;

      if(n!=4)
      {
            printf("Error\n");
            return;
      }

      fp=fopen(name,"r");
```

```
        if(fp==NULL)
        {
                printf("Error wr--->>>./a.out <file_name>\n");
                return;
        }

/***** ************** Find the Size ******************/

        fseek(fp,0,2);
        size=ftell(fp)+1;
        fseek(fp,0,0);


        printf("size:%d\n",size);

/********** ailocate the size to buffer *************/

        buf=calloc(1,size);

/*************** copy file into the buffer *************/

        fread(buf,size-1,1,fp);

//      puts(buf);
        fclose(fp);
        return(buf);
}
```

Strlen
Strcmp

strlen

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

// int strlen(char const *string);

int str_len(char*);

main()
{
     char str[100];
     int p;
     printf("Enter the string:");
     scanf("%[^\n]s",str);
     p=str_len(str);
     printf("the length of string is:%d\n",str_len(str));
}

int str_len(char *p)
{
     int i;

     for(i=0;p[i];i++);
     return i;
}
```

```
{
char *p=str;
int cnt=0;
while(*p!='\0')
{
cnt++;
p++;
}
return cnt;
}
```

O/P: String: Dakshan
length: 7

strcopy

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

// char *strcopy(char *s1,const char *s2);


char  *str_copy(char*,char*);

main()
{
     char s1[100],s2[100];

     printf("Enter the string1:");
     scanf(" %[^\n]s",s1);

     printf("Enter the string2:");
     scanf(" %[^\n]s",s2);

     printf("the string s1 is %s\n",str_copy(s1,s2));

     puts(s1);
     puts(s2);
}

char *str_copy(char *s1,char *s2)
{
     int i;

     for(i=0;s2[i];i++)
     {
          s1[i]=s2[i];
     }
     s1[i]='\0';
     return s1;
}
```

s1 : Deepam
s2 : Joshi
Output s1 = Joshi

strncopy

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

// char *strncpy(char *s1,const char *s2,int n);
// destination string must be large


char  *str_ncpy(char*,char*,int);

main()
{
     char s1[100],s2[100];
     int n;

     printf("Enter the string1:");
     scanf(" %[^\n]s",s1);

     printf("Enter the string2:");
     scanf(" %[^\n]s",s2);

     printf("how many char copy in s2 string:");
     scanf("%d",&n);

     printf("the string s1 is %s\n",str_ncpy(s1,s2,n));

//   puts(s1);
//   puts(s2);
}

char *str_ncpy(char *s1,char *s2,int n)
{
     int i;

     for(i=0;s2[i];i++)
     {
          if(i<n)
          s1[i]=s2[i];
     }
     for(i;i<n;i++)
     {
          s1[i]='\0';
     }
     return s1;
```

strcmp

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

// int strcmp(char const *s1,char const *s2);
// s1>s2 --->>> 1   >0
// s1<s2 --->>> -1
// s1==s2 --->>> 0

int str_cmp(char*,char*);

main()
{
     char s1[100],s2[100];
     int p;

     printf("Enter the string1:");
     scanf(" %[^\n]s",s1);

     printf("Enter the string2:");
     scanf(" %[^\n]s",s2);

     p=str_cmp(s1,s2);

     if(p==0)
     printf("both same string\n");
     else if(p>0)
     printf("s1>s2 not same\n");
     else
     printf("s1<s2 not same\n");
}

int str_cmp(char *s1,char *s2)
{
     int i;

     for(i=0;s1[i] == s2[i] ;i++)
     {
          if(s1[i] != s2[i])
          {
               break;
          }
     }

     if(s1[i]==s2[i])
```

```
        return 0;

        else if(s1[i]>s2[i])
        return 1;

        else
        return -1;
}
```

strncmp

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

int strncmp(char *s1,const char *s2,int n);
// s1>s2 --->>> 1
// s1<s2 --->>> -1
// s1==s2 --->>> 0
// not index wise ..... only lenth

int str_ncmp(char*,char*,int);

main()
{
     char s1[100],s2[100];
     int n,p;

     printf("Enter the string1:");
     scanf(" %[^\n]s",s1);

     printf("Enter the string2:");
     scanf(" %[^\n]s",s2);

     printf("Enter number u want to compare :");
     scanf("%d",&n);

     p=str_ncmp(s1,s2,n);

     if(p==0)
     printf("same\n");
     else if(p>0)
     printf("s1>s2 not same\n");
     else
     printf("s1<s2 not same\n");
}

int str_ncmp(char *s1,char *s2,int n)
{
     int i;

     for(i=0;i<n;i++)
     {
          if(s1[i]!=s2[i])
          break;
     }
}
```

```
    }

    if(s1[i]==s2[i])
    return 0;

    else if(s1[i]>s2[i])
    return 1;

    else
    return -1;
}
```

strcat

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

char *strcat(char *s1,const char *s2);

```
char *str_cat(char*,char*);

main()
{
     char s1[100],s2[100];

     printf("Enter the string1:");
     scanf(" %[^\n]s",s1);

     printf("Enter the string2:");
     scanf(" %[^\n]s",s2);

     printf("the string s1 is %s\n",str_cat(s1,s2));

//   puts(s1);
//   puts(s2);
}

char *str_cat(char *s1,char *s2)
{
     int i,j;

     for(i=0;s1[i];i++);
     printf("i:%d\n",i);

     for(j=0;s2[j];i++,j++)
     {
          s1[i]=s2[j];
     }
     s1[i]='\0';
     return s1;
```

```
int len = strlen(s1);
int i;
for(i=0; s2[i]; i++)
   s1[len+i] = s2[i];
 s1[len+i] = '\0';
return s1;
}
```

s1 = Question
s2 = Jeshi
new string: Question Jeshi

strncat

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

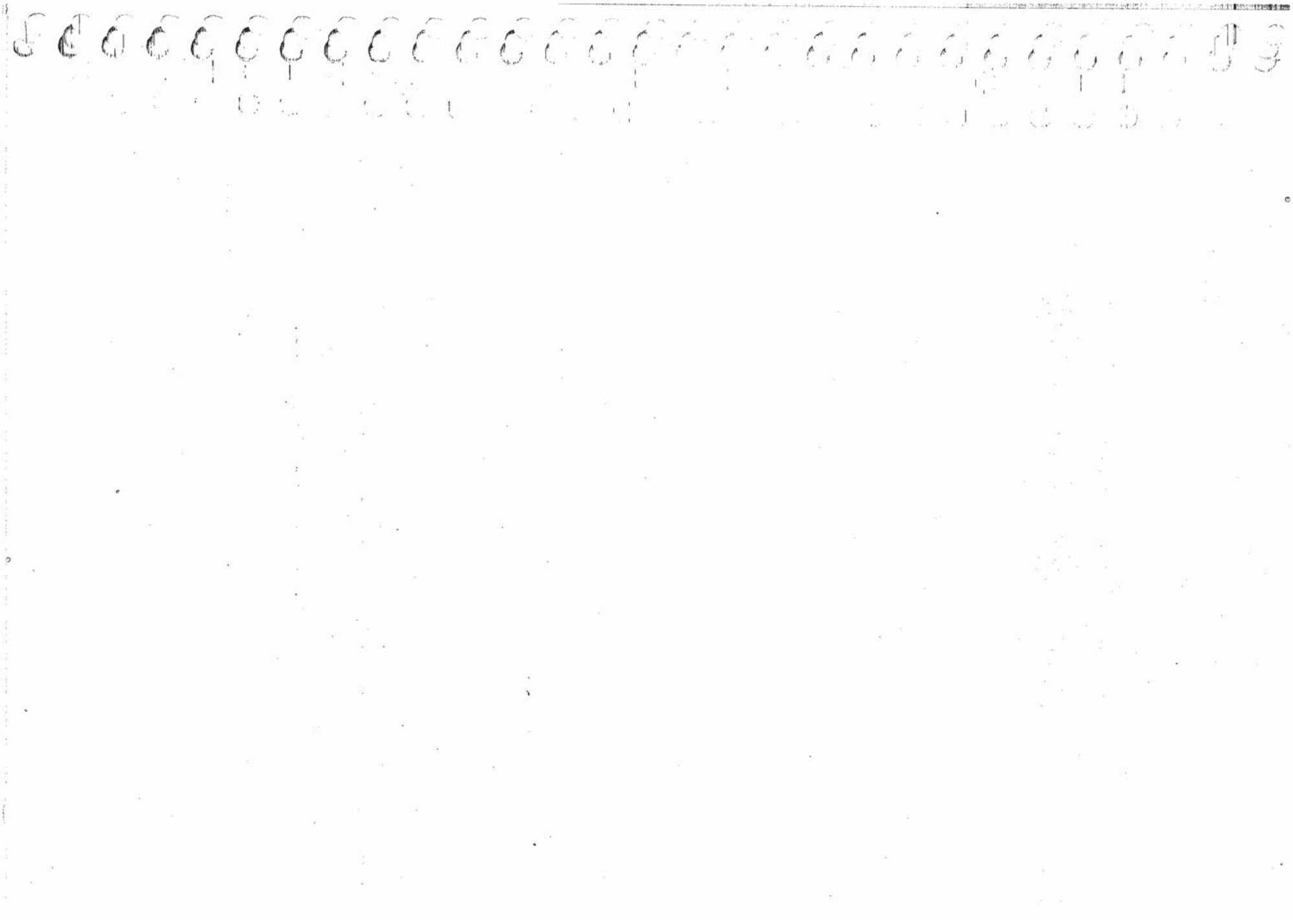
char *strncat(char *s1,const char *s2,int n)

```
char  *str_ncat(char*,char*,int);

main( )
{
      char s1[100],s2[100];
      int n;

      printf("Enter the string1:");
      scanf(" %[^\n]s",s1);

      printf("Enter the string2:");
      scanf(" %[^\n]s",s2);

      printf("how many char concate in s1 string:");
      scanf("%d",&n);

      printf("the string s1 is %s\n",str_ncat(s1,s2,n));

      puts(s1);
      puts(s2);
}

char *str_ncat(char *s1,char *s2,int n)
{
      int i,j;

      for(i=0;s1[i];i++);

      for(j=0;j<n;i++,j++)
      {
            s1[i]=s2[j];
      }
      s1[i]='\0';
      return s1;
}
```

```
int len = strlen(s1);
int i;
for(i=0;i<n && s2[i];i++)
{
    s1[len+i] = s2[i];
}
s1[len+i] = '\0';
return s1;
```

strchr

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

char *strchr(const char *s,int ch);

char *str_chr(char*,int);

main( )
{
     char s[100],ch,*p;

     printf("Enter the string1:");
     scanf(" %[^\n]s",s);

     printf("Enter the charctor:");
     scanf(" %c",&ch);

     p=str_chr(s,ch);

     if(p>0)
     printf("found at %d\n",p-s);
     else
     printf("not found\n");
}

char *str_chr(char *s,int ch)
{
     int i;

     for(i=0;s[i];i++)
     {
          if(s[i]==ch)
          return (s+i);
     }
     return 0;
}
```



strrchr

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

char *strrchr(const char *s,int ch);
last occurance find

```
char *str_rchr(char*,int);

main()
{
      char s[100],ch,*p;

      printf("Enter the string1:");
      scanf(" %[^\n]s",s);

      printf("Enter the charctor:");
      scanf(" %c",&ch);

      p=str_rchr(s,ch);

      if(p>0)
      printf("found at %d\n",p-s);
      else
      printf("not found\n");
}

char *str_rchr(char *s,int ch)
{
      int i,q=0;
      char *p;
      p=s;
      for(i=0;s[i];i++)
      {
            if(s[i]==ch)
            q=p-s;

            p++;
      }
      if(q<=0)
      return 0;
      else
      return s+q;
}
```

```
for(i=0; s[i]; i++)
  {
     if (s[i]==ch)
     p= sti
```

```
#include<stdio.h>
#include<string.h>
char *usersstrstr(char *s1,char *s2);
main()
{
   char s1[]="abcdqbcdef";
   char s2[]="bcde";
   char *ptr;
   ptr = usersstrstr(s1,s2);
   if(ptr==NULL)
      printf("not found\n");
   else
      printf("found");
   printf("found at %d\n",ptr-s1);
}
char *usersstrstr(char *s1,char *s2)
{
   int i,j,n;
   n=strlen(s1);
   for(i=0;s1[i];i++)
   {
      if(s1[i]==s2[0])
      {
         for(j=1;s2[j];j++)
         {
            if(s1[i+j]!=s2[j])
               break;
         }
         if(s2[j]=='\0')
            return(s1+i);
      }
   }
   return NULL;
```

strstr

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

char *strstr(const char *s1,const char *s2);

```
char *str_str(char*,char*);

main()
{
     char s1[100],s2[100],*p;

     printf("Enter the string1:");
     scanf(" %[^\n]s",s1);

     printf("Enter the string2:");
     scanf(" %[^\n]s",s2);

     p=str_str(s1,s2);

     if(p==NULL)
     printf("not found\n");
     else
     printf("found at %d\n",p-s1);
}
char* str_str(char*p,char*q)
{
     int i=0,k,j;

     for(i=0;p[i];i++)
     {
          if(p[i]==q[0])
          {
          k=1;
               for(j=i+1;q[k];j++)
               {
                    if(p[j]==q[k])
                    {
                         k++;
                    }
                    else
                    {
                         break;
                    }
               }
```

vivek vek

```
        if(q[k]=='\0')
        {
            return p+i;
        }
    }
}
```

int m = strlen(s2); c = 0;

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

{

while(*s1)

{

p=s2, c=0

while (*s1 == *p)

{

c++;

p++;

s1++

if(c==m)

return ~~p-c+1~~ s1-c

}

s1++;

}

} return null;

mem cpy

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

void *memcpy(void *s1,const void *s2,int n);

```
void  *mem_cpy(void*,void*,int);

main()
{
      char s1[100],s2[100],*p;
      int n;

      printf("Enter the string1:");
      scanf(" %[^\n]s",s1);

      printf("Enter the string2:");
      scanf(" %[^\n]s",s2);

      printf("Enter value of n:");
      scanf("%d",&n);

      p=mem_cpy(s1,s2,n);

      puts(s1);
      puts(s2);
}

void *mem_cpy(void *s1,void *s2,int n)
{
      int i;
      char *p=s1;
      char *q=s2;

      for(i=0;i<n;i++)
      {
            p[i]=q[i];
      }
      return p;
```

memmove

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

void *memmove(void *s1,const void *s2,int n);

```
void *mem_move(void*,void*,int);

main()
{
      char s1[100],s2[100],*p;
      int n;

      printf("Enter the string1:");
      scanf(" %[^\n]s",s1);

      printf("Enter the string2:");
      scanf(" %[^\n]s",s2);

      printf("Enter value of n:");
      scanf("%d",&n);

      p=mem_move(s1,s2,n);

      puts(s1);
      puts(s2);
}

void *mem_move(void *s1,void *s2,int n)
```

for(i=0; i<n; i++)

{

s[i] = s[j]

s1++

s2++

}

return s1;

[illegible]

memset

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

void *memset(void *s1,int ch,int n);

```
void  *mem_set(void*,int,int);

main()
{
      char s1[100],ch,*p;
      int n;

      printf("Enter the string1:");
      scanf(" %[^\n]s",s1);

      printf("Enter the charctor:");
      scanf(" %s",&ch);

      printf("Enter value of n:");
      scanf("%d",&n);

      p=mem_set(s1,ch,n);

      puts(s1);
}

void *mem_set(void *s1,int ch,int n)
{
      int i;
      char *s=s1;

      for(i=0;i<n;i++)
      {
            s[i]=ch;
      }
      return s;
```

w.a.p. to count the number of charactor if charactor more than one time.

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

main()
{
     char ch,*p,s[100];
     int l=0,cnt=0,i,q;

     printf("Enter string:");
     scanf("%[^\n]s",s);

     p=s;
     for(i=0;s[i];i++)
     {
          p=s;
          ch=s[i];

          cnt=0;
          q=0;

          while(p=strchr(p,ch))
          {
               cnt++;
               q=p-s;
               memmove(s+q,s+q+1,strlen(s+q+1)+1);
          }

          if(cnt>1)
          {
               l++;
               i--;
          }
     }
     printf("l=%d\n",l);
```

Find the maximum and second maximum number in array.

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

main()
{
     int a[5],n,i,temp=0,b,max=0;

     n=sizeof(a)/sizeof(a[0]);

     for(i=0;i<n;i++)
     scanf("%d",&a[i]);

     for(i=0;i<n;i++)
     printf("%d ",a[i]);

     printf("\n");

     for(i=0;i<n;i++)
     {
          if(temp<a[i])
          {
               temp=a[i];
               b=i;
          }
     }
     printf("temp=%d\n",temp);

     for(i=0;i<n;i++)
     {
          if(max<a[i] && b != i)
          max=a[i];
     }
     printf("second max=%d\n",max);
}
```

```
for(i=0;i<8;i++)
{
     if(a[i]> temp)
     {
          prev = temp;
          temp = a[i];
     }
     elseif (prev<a[i])
     {
          prev=a[i];
     }
}
```