DELTA Pg No.
Date / /

Software – As per the industrial standard a digitalized (GUI) automated system is called software.
When the software where is providing graphically user interface then it is called digitalized.
Without human interaction if the process is completed then it is called automated system.

A software is nothing but a collection of programs.
A program is a set of instructions which is designed for a particular task.
A number of programs combining together like a single unit it is called software tool or software component
Software are classified into two types –

1) System software
2) Application software

(1) **System software** – The software design for general purpose & does not having any limitation it is called system software.
System softwares are classified into 3 types

a) OS – Dos, window, linux, Unix
b) Translators – Compiler, interpreter, assemblers
c) Packages – Linker, loader, editor.

DELTA Pg No.
Date / /

(2) Application Software – This software is design for specific task only it is called application softwar.

Ex:- application softwares are classified into two types

① Application Packages - ex- MS office, Oracle

② Special Purpose softwares – ex- Tally

– MS office is a microsoft product which ~~can~~ maintain the information in document format.

– Oracle is a database which maintain the information in document format.

– By using Tally we can maintain the information of accounts

⟹ A Computer is an electronic device which accept instruction from user and according to user sended instruction it produce result.

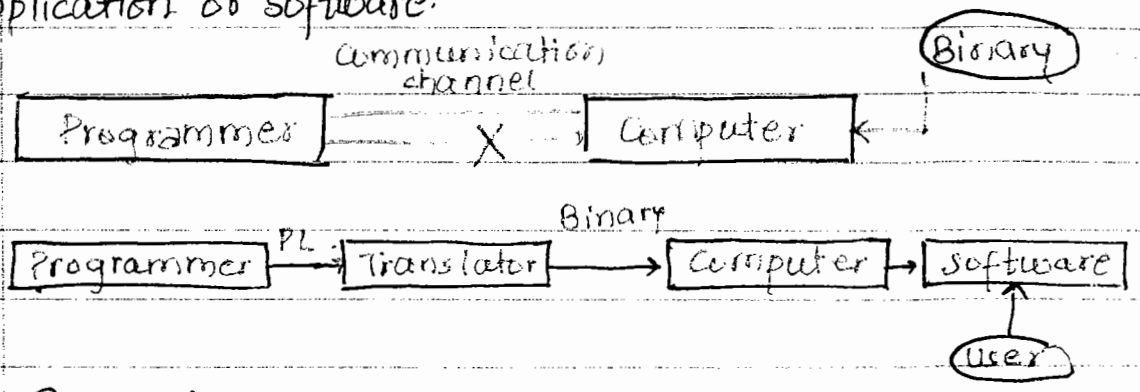
Computer knows only one language i.e binary lang

– Ans a programmer when we required to interact with a computer we need a communication channel called programming language

A programming language is a special kind of instructions which is used to communicate with computer.

Ans a programmer if we knows the programming language then it is not possible to interact with computer because computer can understand binary code only.

In above case, recommended to use translator. As a programmer if the instruction came programming lang., translator will converts programming lang. code into binary format and according to even binary instructions we will get an application or software.



Translators -

A Translator is a system software which converts programming lang. code into binary format.

Translators are classified into 3 types -

① Compiler
② Interpreter
③ Assembler

(1) Compiler - It is a system software which converts programming lang. code into binary format in a single step except those lines are having error.

(2) Interpreter - It is a system software which converts programming lang. code into binary format step by step ie line by line compilation takes place.
(When 1st error is occur it stop compilation process)

(3) Assembler- By using assembler we can convert assembly language instructions into binary formats.

As per the performance wise, recommended to use compiler

As per the development wise recommended to use an interpreter

PROGRAMMING LANGUAGE

A programming lang. is a special kind of instructions which is used to communicate with computer.

Programming lang. is classified into two types-

(1) High level Programming lang.
(2) Low level Programming lang.

(1) HIGH LEVEL PROGRAMMING LANGUAGE

which programming lang. syntactically similar to english and easy to understand it is called high level P.L

By using high level P.L we are developing user interface application.

Ex:- C, C++, VC++, java, C#, swift, Objective C, D- language

(2) LOW LEVEL PROGRAMMING LANGUAGE

This programming lang. is also called assembly lang.

C Programming Language

(1) It is high level procedure oriented structured programming lang.

(2) Which P.L is syntactically similar to english and easy to understand is called high level P.L

(3) When the programming language supports module or function implementation then it is called procedure oriented lang.

(4) Top down approach in the form of blocks is called structured Programming language

HISTORY OF C

(1) The programming language term is started in the year of 1950s with the lang. called FORTRAN

(2) From FORTRAN lang. another programming lang. is developed called ALGOL (Algorithmic language)

(3) The beginning of C is started in the year of 1968 with the language called BCPL Martin Richard (Basic Combine Programming Lang.)

(4) In the year of 1970s from BCPL another Programming language is developed by Ken Thompson it is called B-language (Basic language)

(5) In the year of 1972 Dennis Ritchie developed C-programming language at AT and T Bell laboratories for developing system software.

(6) In the year of 1978, Ritchie and Kernighan released next version of C language

DELTA Pg No.
Date / /

"K and R-C"

(7) In the year of 1988, ANSI is released next version of C Language called "ANSI-C" (American National Standard Institute)

ASCII → American Standard code for information Interchange.

(8) In the year of 2000. ISO standaid C is released called "C 99"

(9) On 8th of Dec 2011, latest version of C is released with the name called "C11" which having actual name has '

(10) In alphabetical order only the name C-Language is given

(11) For giving the name has C++ there is a reason nothing but post features of C-language.

Applications of C

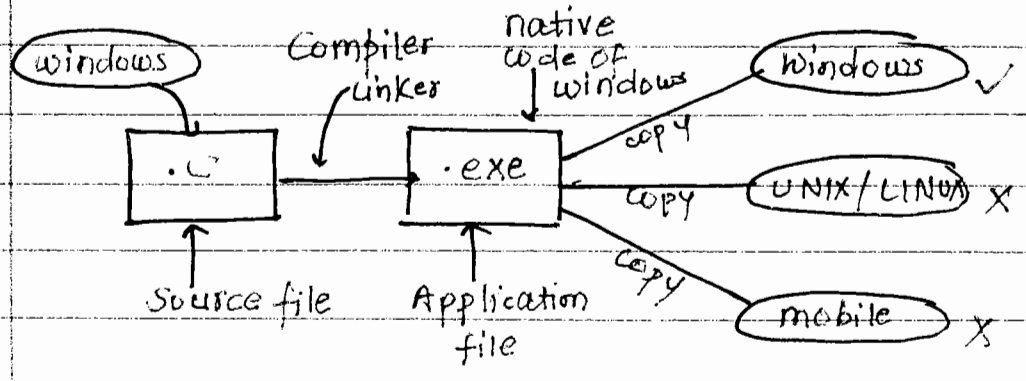
(1) C-programming Lang. can be used for different type of application like

- System software development i.e Operating System and Compiler.
- Application software development and EXCEL sheets
- Graphics related applications i.e PC and mobile games
- Any kind of mathematicatical expressions can be evaluated.

DELTA Pg No. Date / /

Advantages of C

(1) Portability - It is a procedure of carrying the instructions from one system to another system.



(1) As per above observation when we are copying .exe file to any other computers which contains Windows Operating system then it works properly. Because native code is same.

(2) Same .exe file if we are copying to any other computer which contains UNIX / LINUX OS then it doesn't works because native code is different. So this behaviour is called Platform dependency.

Platform dependent, Independent

After developing any application on a specific OS if we are able to execute on same OS then it is called Platform dependent.

→ If we are able to run on multiple OS then it is called platform independent

C Programming language

- Platform dependent, mission independent P.L i.e it doesnot depend on hardwired component of a system.

Source code :- Set of high level programming lang. and programming instruction

Object code :- Compiled format data of source code is called object code.

Native code :- Unix instructions of a OS is native code.

Byte code :- Java and C# related compiled code are called byte code.

Source code and byte code are platform independent
Object code and native code are platform dependent

- C programming doesn't supports cross platform applications.
- Modularity :- When the application is developing in same modules or in functions then it is called modularity.
- Mid-level :- C programming lang. is called middle level programming language. Because it can support high-level language features in the combination of assembly language also.
- Simple :- C programming lang. syntactically similar to english and limited concepts are available

28/5/2015

DELTA Pg No. Date / /

Characteristics of C

To develop a program what fundament components are required those are called characteristics of C.

In C prog lang. we having 6 characteristics i.e

→ Operator
→ Keywords
→ Seperators
→ Constants
→ Pre-define functions
→ Syntax

(1) Keyword - It is a reseive word, some meaning is already available to this word & that meaning automatically recognizable by compiler.
In C prog. lang. we are having 32 Keywords ex:- if, else, void, for, break - - - etc.

(2) Operator - It is a special kind of symbol which performs a particular task.
In C prog. lang. we are having 44 operators ex- +, -, * ...

3) Seperators - By using seperator, we can seperate, an individual unit called token.
In C prog. lang. total no. of seperators are 14. ex:- , ; : ' ' " " { } space, etc

4) Constants - It is a fixed one never change during the execution of program
In C, C constants are classified into 2 types -
1) Alpha numeric constants
2) Numeric constants

DELTA Pg No.
Date / /

(i) **Alpha numeric constant**:- By using this constant we can represent alphabets and 0-9 no.s

Alpha numeric constants are classified into two types -

a) Character constant

b) String constant

Any data if we are representing in single quotes then it is called character constant.

ex - 'A', 'd', '+', '5', '#'

When we are representing the data within the double quotes then it is called string constant.

ex:- "freshjobs" "Hello"

Under alpha numeric constants we having only one type of data values i.e char.

In C programming lang., total no. of characters are 256.

(ii) **Numeric Constant**:- By using numeric constant we can represent value type data Numeric constant are classified into 2 types -

a) Integer

b) Float

When we are representing the numeric values without any fraction parts then it is called integer. Ex- 12, -12, 48, -48, 1234;

When we are representing the numeric values with fractional parts then it is called float

Ex- 12.3, 14.85, 98.10

Note - char, int and float are called basic datatype or basic data elements bcz any data is a combination of these 3 types of constants.

(5) Pre-define function - These all are set of preimplemented functions which are available along with the compiler.

When we required to perform any specific task then we need to call pre-define function. ex- printf(), scanf(), strcopy(), textbackground(), gettime(), setdata()

The basic syntax of C lang. is every statement should ends with ;

Operators :- It is a special kind of symbol which perform a particular task.

In C programming lang. we are having 44 operators and depends on no. of operands this operators are classified into 3 types

- Unary Operator
- Binary Operator
- Ternary Operator

When we are working with unary operator it require only one argument or operand.

Binary takes 2 operands and ternary is required 3 operands.

When we are evaluating any expression what input data we are passing it is called operand, which symbol we are using it is called operator.

DELTA Pg No.
Date / /

Assignment Operator :-

→ It is a binary Operator. 1)

→ Binary operator means require 2 arguments i.e left, right side arguments. 2)

→ By using assignment operator we can assign right side value to left side variable

→ When we are working with binary operator if any one of the argument is missing, it gives an error 3)

→ When we are working with assignment operator right side argument can be variable type or constant type left side argument must be variable type only. 4)

Syntax :- L = R ;
L: variable, R: var or constant

Ex :- 1. a = 100; ✓
2. a = 100; Error statement missing
3. a = 12.86; ✓
4. a = 'D' ✓
5. a = ; Error R value require (Binary)
6. a = 120; Error (Binary)
7. 100 = 200; Error L side should be var.
8. a = 100;
b = a; ✓

29

Arithmetic Operator :-

When we required to evaluate basic mathematical expressions then we required to use arithmetic operators.

- Arithmetic operators are of 2 types -
  - Unary arithmetic Operators ++, --
  - Binary Arithmetic Operators +, -, *, /, %

DELTA Pg No. Date / /

* Binary Arithmetic Operators –

1) When we are working with these operators we required 2 operands. If any one of the arg. is missing it gives an error.

2) In implementation when we are evaluating any expression if that expression contains multiple operators then inorder to evaluate that expression we require to follows priority of the operator.

3) According to priority which operators are having highest priority it should be evaluate first, which operator contains least priority it should be evaluated at last.

4) When equal priority is occured, if it is binary left to right, if it is unary, right to left required to evaluate.

1. * / %
2. + –
3. =

29/5/2015

1. a = 2+5; 7 (return values)
2. a = 5+3-4; 4
   8-4
3. a = 5-4+3; 4
   1+3
4. a = 2*3 + 2*3; 12
   6 + 6
5. a = 2+3*2+3; 11
   2+6+3
6. a = 5 +; Error
7. a = 8 -; Error
8. a = +5; 5
9. a = -8; -8

→ When we are working with +, - symbols then it works like a binary and unary operator also.

→ If the symbol is available before the operands then it is unary operator which indicate sign of the value.

→ When the symbol is available after the operands then it is binary arithmetic operators which require 2 arguments.

→ Always expressions required to evaluate acc. to operand type only.

→ Always operator behaviour is operand dependent only i.e what type of data we are passing depends only on input types it works.

→ If both arguments are integer then return value is int.

→ Any one of the argument is float or both are float then return value is float type only.

→ Output sign will depends on numerator value sign and denominator value also i.e If any of the argument is negative then return value is negative, if both arguments are return value is positive.

→ In division operator when the numerator value is less than of denominator operator then return value is zero if both are integer.

Expression	Return Values
5/2	2
5.0/2	2.5
5/2.0	2.5
5.0/2.0	2.5
2/5	0
2.0/5	0.4
-5/2	-2
5/-2	-2

DELTA | Pg No. | Date / /

Modulus Operators (%) :-

This operator returns remainder value.

➤ Remainder value means the part which is not divisible in modulus operator - Return value sign depends only on numeric value.

➤ When the numerator value is less than half denominator value then return value is numerator value only

```
a= 15 % 3 ;    0
a= 21 % 2 ;    1
a = 12 / 8 ;   4
a = 16 % 1 ;   0
a = 27 % 7 ;   6
```

Value	Return Value
47 % 5	2
47 % -5	-2
-47 % -5	2
5 % 2	1
2 % 5	2

```
a = 12.0 + 5/6 % 2 ;
```

When we are working with modulus operator, 2 arguments are required and bo

DELTA Pg No.
Date / /

→ We can't apply it for float datatype

→ In implementation when we require to calculate remainder value of float datatype then go for fmod() or fmod() function which is declared in <stdlib.h>

Syntax :-

```
value = fmod (n, d);
// return type is double datatype
value = fmod (n, d);
// return type is long double datatype
```

```
a = 158 % 10 ;   8
a = 158 / 10 ;   15
a = 104 % 10 ;   4
a = 1048 / 10 ;  104
a = 86 % 10 ;    6
a = 23 % 10 ;    3
a = 23 / 10 ;    2
```

→ % 10 always provide last digits of input value.

→ Divided by 10 always removes last digit of input values.

DELTA Pg No.
Date / /

Relational and logical Operators

→ In C and c++, all relational and logical operators return 1 or 0.

→ If expression is true then return value is one, if expression is false then return value is 0.

→ Every non-zero is called true, when the value become zero, it is false.

→ Relational Operators are :-

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

→ logical Operators are :-

&&, ||, !

Note :- ANSI-C prog. lang supports bool data-type so relational, logical operator returns TRUE or FALSE.

1.	()	Paranthesis
2.	+, --, !	Unary operator
3.	*, /, %	
4.	+, -	
5.	<, >, <=, >=	
6.	== , !=	
7.	&&	
8.	||	
9.	? :	Conditional operators.
10.	=	

04/6/2015

DELTA Pg No.
Date / /

Relational OPerators

1. a = 2>5 ; 0
2. a = 5<8 ; 1
3. a = 3>2>1; 0
   1>1
4. a = 3>2>0; 1
   1>0
5. a = 5<8>2<5; 1
   1>2<5;
   0<5

1. a = 5>5<=0; 1
   0<=0;
2. a = 8<=8>=1; 1
   1>=1
3. a = 5>(8<5)<5; 1
   5>0<5
   1<5

Whenever we have to keep one of the part in the expression with highest priority we keep it in paranthesis.

L = R ; → assignment
L == R ; compares
comparision = equals
L != R → not equals

1. a = 2 == 8 ; 0
2. a = 5 == 5 ; 1
3. a = 2<5 == 0 ; 0
   1 == 0
4. a = 5>2 == 2<8 ; 1
   1 == 1
5. a = 1>(5==5)<5 ; 1
   1>1<5;
   0<5

1. ( )
2. +, -, !
3. * / %
4. +, -
5. <, >, <=, >=
6. ==, !=
7. &&
8. ||
9. ? :
10. =

DELTA PG NO.
DATE / /

1 1. a = 5 !₀ = 5; 0

2. a = 2 ! = 8; 1

3. a = 2 > 5 !₀ = 5 < 8; 1

4. a = 1 !₀ = 5 < 8; 0

1 5. a = 1 ! = 5 > 8 == 2 < 5 !₀ = 1; 0

1 !₀ = 0 == 1 !₀ = 1

1 == 1 !₀ = 1

1 !₀ = 1

7 6. In implementation when we require to
D combine multiple expressions then
recommended to go for logical operators

• In C programming language, we have 3 logical operators i.e is Logical AND && - Binary
(ii) Logical OR || - Binary
(iii) Logical NOT !₀ - Unary

&&	||	!₀
T T → T	T T → T	T → F
T F → F	T F → T	F → T
F T → F	F T → T	
F F → F	F F → F	

Every non-zero should be called as 1.

DELTA Pg No.
Date / /

a	b	a && b	a || b	! a
1	1	1	1	0
0	1	0	1	1
1	0	0	1	0
0	0	0	0	1

1. a = 5>8 && 2<5 ; 0
   0 && 1
2. a = 8>5 && 2>8 ; 0
   1 && 0
3. a = 5>8 && 8<2; 0
   0 && 0
4. a = 5>2 && 2<8 ; 1
   1 && 1
5. a = 1 != 1<5 && 0 != 0<5 ; 0
   = 1 != 1 && 0 != 1
   0 && 1
6. a = 2>5 != 2<5 && 8>5 == 2<5 ; 1
   ⇒ 0 != 1 && 1 == 1
   1 && 1
7. ~~a = 2<5 ;~~

OR Combinations

1. a = 2<5 || 2>5 ; 1
   1 || 0
2. a = 5>8 || 5>2 ; 1
   0 || 1
3. a = 8<5 || 2>5 ; 0
   0 || 0

DELTA PAND
Date / /

4. a = 1 != 1 > 5 || 0 != 0 < 5; 1

1 != 0 || 0 != 1

1 || 1

5. a = 5 > 2 != 2 || 2 < 5 != 5; 1

1 != 2 || 1 != 5

1 || 1

NOT Combinations

1. a = !5 ; 0
2. a = !0; 1
3. a = !5 != 5; 1

0 != 5;

4. a = !(2 < 5 && 2 > 5); 1

!(1 && 0)

Points

1. When we are working with logical OR operator anyone of the expression is true or both expressions are true, then return value is 1.
2. In logical AND operator, both expressions or all expressions are true then only return value is 1.

DELTA | Pg No.
Date / /

- C Programming Language is called CASE Sensitive Language i.e. upper & lower case contents both are different
- In C Programming language all existing keywords, and pre-defined functions, all are available in lower case only.
- To write a C Program, we need **IDE** (Integrated Development Environment)

⇒ Generally IDE provides Editor, compiler & linker
Loader is also required - which is a part of OS

- Editor - provides workspace for typing the program.
- Compiler - will perform translations
- Linker - will combine object file into application file (OPJ → .exe)

⇒ For C programming Languages we having different types of IDEs.

for 32 bit
1. Turbo C++ v3.0 (DOS, XP, win7 - 32 bit, win8 - 32bit)
2. Borland Turbo C++ 4.5 (XP, win 7/8 - 32 bit)
3. Turbo C++ 5.06 (win 7/8 32/64 bit)
4. DevC++ 4.9.9.2 (Win 7/8 32 bit os)

for 64 bits
5. C-Free 4.0 (win 7/8 32 bit)
6. C-Free 5.0 (win 7/8 32 bit)
7. Dev-C++ 5.6.3 (win 7/8 32/64 bit)
8. Code Block (win 7/8 32/64 bit) Linux also

For embe

(1)

os

DELTA Pg No.
Date / /

For embedded
9. Cygwin (Win 7/8 32/64 bit) gcc compile for windows/linux
10. GCC compiler for linux in-built
11. KDeveloper for linux inbuilt in redhat linux

Diff. b/w Turbo C++ and GCC

① Data Types - Turbo C → int 2 bytes
GCC → int 4 bytes
Turbo C → support long, double
GCC → don't support

(2) Inc/dec

3) Pointer size (2B/4B/8B)

⇒ if we are using linux OS, gcc compiler is already installed.

05-6-15

1st Prog

```
void main()
{
    Print("Welcome");
}
```

O/P: Welcome

* When we are working with Turbo C compiler, by default standard I/o related and console I/o related predefined functions are supported by default. That's why it is not required to include stdio.h and contain header files.

DELTA Pg No.
Date / /

* When we are working with high and compilers, for every predefined fn, corresponding header file must be required to include or else compiler provides warning message.
* Generally main() fn, void is a keyword which indicates starting point of an app.
* '{' indicates dat instruction block is started, closing curly brace indicates that instruction block is ended.
* all the instructions must be required to place within the body only.

Printf():-

* It is predefined fn which is declared in <stdio.h>
* By using this predefined fn, we can print the data on console.
* When we are working with printf() function, it can take any no. of arguments but first arg. must be string constant & remaining arguments are seperated by comma.
* Within the " " doeuble quotes, whatever we pass, it brings it like that only, if any format specifies are there, then copy that type of data.

Syntax :-

```
int [cdecl] printf (* const char * format, ...);
```

- int → Return type
- printf → name of function
- * const char * format → String const type
- ... → Variable argument list

DELTA Pg No.
Date / /

FORMAT SPECIFIERS

(1) All format specifiers will decide that what type of data required to print on console.

- int → %d
- short → %hd
- char → %c
- long → %ld
- float → %f

* Designing a prog on linux OS.

(1) Open the terminal and type following command
vi file1.c

(2) To make typing enable just press i button & type prog.

```
#include <stdio.h>
int main()
{
    printf("Balututorials");
    return 0;
}
```

O/P: Balututorials

(3) Press Esc button, then write following command
:wq (writing quit)

(4) For compiling and linking the page, we are required to use following command
gcc -o file1 file1.c

(5) To load or execute d prog, we required to use following command:
./file1

DELTA Pg No.

① stdio.h provides standard I/O related predefined functions prototypes → just declaration not implementation

② .h file doesn't provides any implementation part of predefined functions, it provides only prototype i.e forward declaration of function.

③ void main() fn doesn't provide any exit status back to the OS.

④ int main() fn provide exit status back to the OS i.e success or failure

⑤ If we are required to provide exit status as success, then return value is 0 i.e return 0; or return EXIT-SUCCESS

⑥ When we are required to inform exit status or failure, then return value is 1 i.e return 1; or return EXIT-FAILURE

TOKEN

① Smallest part of programming or an individual part of programming is called

② A C prog. is a combination of tokens.

③ Tokens can be Keyword, operator, separator, constant and any other identifiers.

④ When we are working with tokens we can't split dat token or we can't break the token but b/w the tokens, we can keep any no. of spaces, tabs and newline characters

```
void main vo id m ain ()
{
    pri nt ("Welcome");
}
```

O/P: Error

```
void
main
(
)
{
    printf ("Hello");
}
```

O/P: Hello

DELTA PAGE NO. DATE / /

1. printf ("Hello"); Hello

2. printf ("@@welcome##"); @@welcome##

3. printf ("%d Welcome %d" 10,20); Error

function call missing

" " 10 not seperated by comma

4. printf ("%d welcome%d", 10,20,); Error

function call missing

20, seperator then, compiler waits for one more argument

5. printf ("%d welcome %d", 10,20) Error

Statement missing

6. printf ("%d welcome %d", 10, 20);

10 welcome 20

FAILURE

In printf statement when we are passing format specifier then at the time of execution automatically format specifiers are replaced with corresponding value.

1. printf ("%d%d%d", 10,20,30);

102030 no clarity

2. printf ("%d %d %d", 10,20,30);

10 20 30

3 printf ("%d, %d, %d", 10,20,30);

10, 20, 30

⟹ Within the double quotes of a printf everything is treated like normal characters only except format specifiers & special characters

DELTA Pg No. Date / /

10. printf ("2+3= %d", 2+3);
2+3= 5 ↳ expression will return value

11. printf ("%d %d %d", 100, 200);
100 200 0 → garbage / junk value

12. printf ("%d %d", 100, 200, 300);
100 200

⇒ In printf statement when we are passing an additional format specifier which doesn't having corresponding value then it bring some unknown or undefined value called garbage / junk.

⇒ In printf when we are passing an additional value which doesn't having corresponding format specifier then that value is ignored because format specifier only decide that what type of data is required to print on console.

13. printf ("Total salary = %d", 25,000);
Total salary = 25

14. printf ("Total salary = %d", 25000);
Total salary = 25000

15. printf ("Total salary = %d");
Total Salary = gr/ junk

16. printf ("%d %d", 2>5, 5<8);

O/P 0 1

→ void main()
{
a = 10;
printf ("a = %d", a);
}
Error : undefined symbol 'a'

→ void main()
{
int a; // variable declaration
a = 10; // Assignment
printf ("a = %d", a);
}
O/P : a=10

* VARIABLE Declarations

(i) Name of the memory location is called variable

(ii) Before using any variable in the program, it must be required to declare first

(iii) Declaration of variable means required to specify data type, name of the variable followed by same order.

(iv) In 'C' prog. lang. variables required to declare on top of the prog. after opening the body & writing first statement.

(v) In declaration of a variable, ~~existing~~ name of the variable must be required to start with alphabet or underscore only.

(vi) In declaration of the variable, existing key words, Operators, seperators, constants are not allowed

(vii) In declaration of the variable, max. length of variable name is 32 characters, after 32 characters compiler will be not consider remaining characters.

```
Data Type variable
or
DataType var1, var2, var3
```

- When we are using multiple variables of same data type then recommended to use comma as a seperator

- According to syntax, atleast single space must be required b/w datatype and name of the variable.

DELTA Pg No.
Date / /

Syntax to Initialize a Variable

Data Type variable = value;

Ex-

1. int a; Error
2. int a;
3. int a b c; Error
4. int a,b,c ;
5. int abc, d ;
6. int if ; Error
   ↳ Keyword
7. int If ; yes valid bcz C is case sensitive and all Keywords in same case.
8. int _if ; yes valid
9. int 1,2,3 ; Error
   constant does not allowed
10. int _1,_2,_3; Valid
11. int 1a, 1b, 1c ; Error
12. int a1, b1, c1 ; yes
13. int total-sal ; Error
    ↳ operator
14. int total_sal ; Yes
15. int printf; Valid

★ All Keywords are reserved words, bt all reserved words are not Keywords

DELTA Pg No.
Date / /

because -

Predefined reserved words are possible to redefine but Keywords are not possible to redefine.

like printf is keyword not defined in java, C#, etc.

As per the variable declarations for naming convension we required to follow 2 rules -

1. CAMEL NOTATION -

According to this notation first character of every word should be required in upper case & every subsequent word also is required to follow same condition.

eg: ~~Tota~~ int TotalSalary;

2. Hungarian Notation -

Acc. to this notation every variable should require a prefix which indicates what type of datatype is the variable

→ The basic difference b/w declaration and initialization of a variable is

In declaration of the variable after creating the memory it stores garbage value until we are assigning the data

In initialization of the variable after creating the m/m by default it stores what value is assigned

DELTA Pg No. Date / /

```
int a;
a=10;
```

a

garbage 10

```
int a=10;
```

a

10

Ex- 1. Open the terminal and write the following commands

vi file2.c

2. For enabling typing, press i button

```
#include<stdio.h>
int main()
{
  int i;
  float f
  i= 5/2;
  f= 5/2;
  printf("i=%d f=%f", i, f);
  return 0;
}
```

/press escape button then type following command [:wq]

/Compile & link : gcc -o file2 file2.c

Run/load : ./file2

o/p : [i=2 f=2.000000]

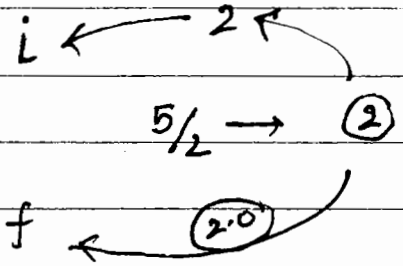
DELTA Pg No.
Date / /

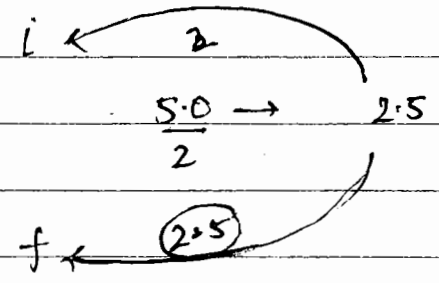
→ Operator behaviour is always operand dependent only i.e depends on input values only, behaviour of operator will be changed.

→ Return value behaviour is always variable dependent only i.e depends on variable type automatically return value will be changed.

→ When the operator is ~~ope~~ returning an integer value and we required to store in float variable then automatically return value will convert it into float format by adding zero

→ When the operator is returning float value and we are storing into int variable then decimal value only is assigned





value	int i;	float f;
—	gr	gr
5/2	2	2.0
5.0/2	2	2.5
5/2.0	2	2.5
5.0/2.0	2	2.5
2/5	0	0.0
2.0/5	0	0.4

Control Flow Statements

→ The execution flow of the prog. is under control of control flow statements.

→ In C prog. lang. control-flow statements are classified into 3 types

1) Selection statements
   ex :- else, if, elseif, switch
2) Iterative statements
   ex - while, for, do while
3) Jumping statements
   Ex - break, continue, goto

(1) Selection statements

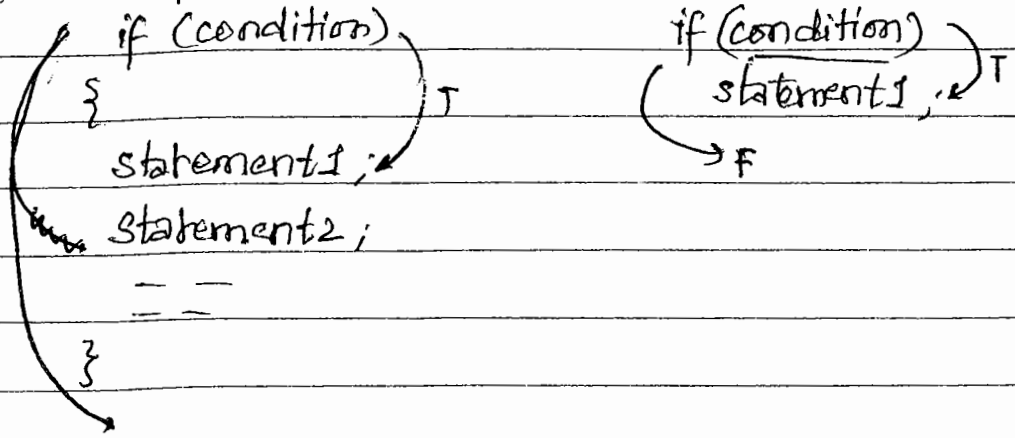
→ These are also called decision making statements

→ By using them, we can create conditional oriented block.

→ When we are working with selection statements if condition is true then block is executed if condition is false then corresponding block will be ignored

DELTA Pg No.
Date / /

Syntax to If :-



→ Constructing the body is always optional.

→ Body is recommended to use when we having multiple statements

→ For a single statement, it doesn't require to specify a body, if body is not mentioned then automatically scope is terminated with next semicolon.

* Else :-

→ else is a keyword, by using this keyword we can create alternate block of if condition

→ Using else is always optional, it is recommended to use when we having alternate block.

→ When we are working with if and else, only one block can be executed i.e when 'if' condition is false then only 'else' part is executed

Syntax

```
if (condition)
{
    statem
```

DELTA Pg No.
Date / /

Syntax

① In this
In 'if' - multiple statements
'else' - multiple statements

```
if (condition)
{
  statement 1
  statement 2
}
else
{
  statement 3
  statement 4
}
```

T → statement 1 → statement 2
F → else → statement 3 → statement 4

②

```
if (condition)
{
  statement 1
  statement 2
}
else
  statement 3
```

T → statement 1 → statement 2
F → else → statement 3

if → ms
else → s

③

```
if (condition)
  statement 1;
else
  statement 2;
```

T → statement 1;
F → statement 2;

if - s
else s

④

```
if (condition)
  statement 1;
else
{
  Statement 2;
  Statement 3;
}
```

T → statement 1;
F → else { Statement 2; Statement 3; }

if - s
else - ms

PROGRAM

```
void main()
{
  printf("A");
  printf("B");
  if (2 < 1 < 5)      // if (1 && 1)
  {
    printf("C");
    printf("D");
  }
}
```

WARNING MESSAGE
When

```
    printf ("Welcome");
}
```

O/P : ABNITC Welcome

Note: When we are constructing any conditions by using constant expression then compiler provides a WARNING MESSAGE i.e condition is always true or False

→ When the warnings are occured, it can be ignored if it is possible bcz prog. can be executed properly.

PROGRAM

(1)

```
void main()
{
  printf ("NIT");
if (1!=2<5 || 0==5<8)        // if (1!=1 || 0==1)
                                0 || 0 = 0
  {
    printf ("A");
    printf ("B");
  }
printf ("C");
}
```

F

O/P : NITC

(2)
```
Void main ()
{
 printf ("Hello");
 if (5<8 | = 1)        // (1 != 1)  ⇒(0 =1) False
   printf ("A");  → if scope is close here
 printf ("B");
 printf ("C");
}
```
O/P : HelloBC

→ When the body is not specified then automatically scope is terminated with next semicolon i.e within the condition only 1 statement will be placed

(3)
```
Void main()
{
 printf ("NIT");
 if (2<5 | = 2>5)      // (1 | = 0)  True
{
 printf ("A");
 printf ("B");
}
 else
   {
    printf("C");
    printf ("D");
  }
}
```
O/P: NITAB

DELTA Pg No.
Date / /

(4)
```
void rnain()
{ printf ("Welcome");
  if (!5)                    //0 → False
  {
    printf ("A");
    printf ("B");
  }
else
  {
    printf ("C");
    print ("D");
  }
}
```
O/P: Welcome CD

5)
```
void main()
  {
    printf("A");
    if (5>8!=1).          // if (0!=1)
    {
      printf ("B");
      printf ("C");
    }
else
  {
    printf ("D");  → else scope is close here.
    printf ("NIT");
}
```
O/P: ABCNIT

→ When the body is not specified for else part then automatically scope is terminated with next semicolon.

6)
```
Void main
{
  printf("A")
  if (
     printf("B");
     printf("C");
else
   {
     printf("Hi");
     printf("C");
   }
}
```

O/P: ~~ABC~~ Error, misplaced else

else scope starts only after if scope only.

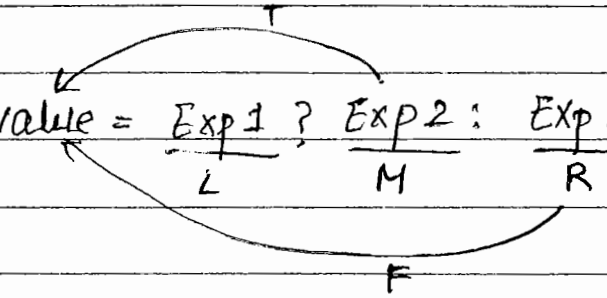
→ According to syntax of if-else when we are using elsepart it must be required to start after if scope only.

CONDITIONAL OPERATORS (?:) →

1. Conditional Operators are ternary category operators.
2. Ternary Category means it required 3 arguments, i.e left, middle & rightside arguments.
3. When we are working with conditional Operators, if cond^n is true, then returns with middle arg. If cond^n is false, then returns with rightside arg.

and left side argument is treated like condition.

Syntax:



4. According to syntax, if exp 1 is true or leftside value is non-zero, then exp2 or middle value is ~~written~~ returned.

5. If exp1 is false or leftside value is 0, then exp3 or right side value is returned.

6. When we are working with conditional operators, we require to satisfy following conditions —
   1) No. of ? and : should be equal.
   2) Every colon should match with just before ?.
   3) Every " " followed by ? only.

• int a;

1. a = 10 ? 20 : 30; O/P = 20

   a = 10 (L) ? 20 (M) : 30 (R) ;

2. a = 5 < 8 (0) = 1 ? 20 : 30; O/P = 30

3. a = 2 > 5 ? 10 : 20 : 30; O/P = Error

   No. of ? and : are equal

DELTA Pg No.
Date / /

4. a = 2<5!=1 ? 10 : 5>2 ? 20 : 30;
O/P : L 1 m 1 2 2 R

upto 1st ? → L
after 1st ? Before 1st : → m
after 1st : Before ; → R

1 != 1 | 5>2 ? 20 : 30
L m R

O/P = 20

* 1. a = 5>2 ? 2<5! = 0 ? 10 : 20 : 30;
L M R

a = 5>2 ? 2<5! = 0 ? 10 : 20 : 30;
1 2 2 1
L m R

O/P → 10

* int a
1. a = 5>8 ? 10 : 2<5 ! = 1 ? 20 : 5>2 ? 30 : 40;
L 1 m 1 R 2 2 3 3

L m R
L m R

O/P → 30

2<5! = 1 ? 20 : 5>2 ? 30 : 40
L m R

False → then right side.

5>2 ? 30 : 40
L m R

True then m (30)

DELTA Pg No.
Date / /

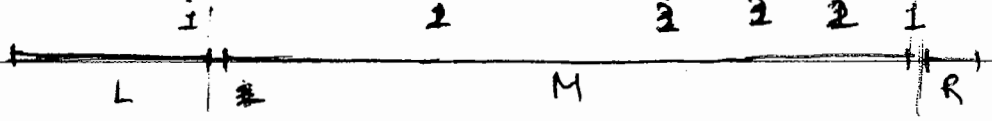
→ When we are working with multiple ? and : then initially we required to convert the expression into 3 arguments.

→ In order to convert the expression into 3 arguments, it is recommended to follow Numbering system

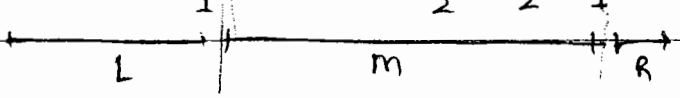
→ According to numbering process, for every ? one unique no. required to assign and for every ':' corresponding ? no. required to assigned.

→ Upto 1st question mark, it is called left argument after 1st ?, by 1st ? corresponding : should be middle argument and remaining complete part is Right side argument

1. a = 8>2 ? 2<5!=0 ? 5! = 5>2 ? 10 : 20 : 30 : 40;



2<5!=0 ? 5! = 5>2 ? 10 : 20 : 30



5! = 5>2 ? 10 : 20

L m R

5! = 1

O/P : 10

2. a = 5 > 8 ? 2 < 5 ? 10 : 20 : 2 > 5 ! = 0 ? 30 : 5 < 8 ? 2 > 5 ! = 0 ? 15 ? 40 : 8 ? 50 : 60 : 70 : 80 ;

1 2 2 1 3 3 4 5 6 6 7 7 5 4

L m R

L m R

5 < 8 ? 2 > 5 ! = 0 ? ! 5 ? 40 : 8 ? 50 : 60 : 70 : 80

L R

m

2 > 5 ! = 0 ? ! 5 ? 40 : 8 ? 50 : 60 : 70

L m R

O/P : 70

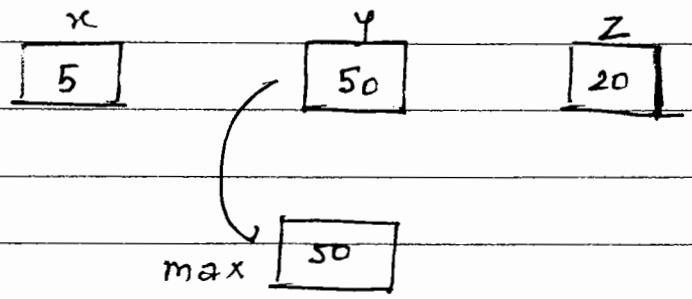
DELTA PG No. DATE / /

DELTA Pg No.
Date / /

1) The basic advantage of conditional operator is reducing coding part of the prog.

2) When we are reducing the coding part then it occupies less memory, so automatically performance will increase.

PROGRAM




```
void main()
{
  int x, y, z, max;
   x = 5; y = 50; z = 20;
if (x > y && x > z)
  {
    max = x;
  }
if (y > x && y > z)
  {
    max = y;
  }
if (z > x && z > y)
  {
   max = z;
  }
  printf ("max value is: %d", max);
}
```

O/P : Max value is: 50

- In implementation when interelated blocks are constructed independently then after completion of the requirement also, compiler checks remaining all cond's. So it is time taking process

- When interrelated blocks are occur then always recommended to create in optimal blocks by using else part.

- By using else part, we can create only 1 optimal block, if we required to create multiple blocks then recommended to go for nested if else

Note: In previous prog., in place of writing multiple condns we can create single statement which can provide max value i.e conditional operator required to use.

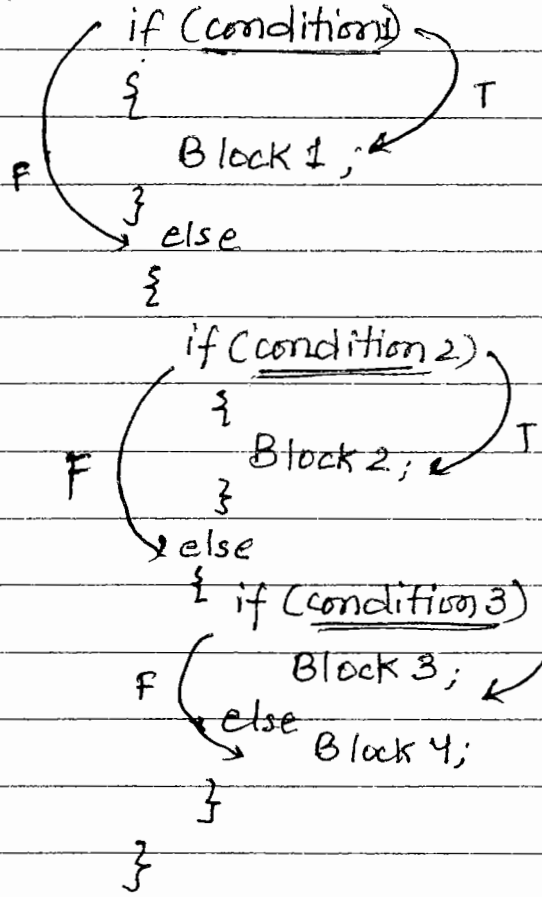
*
```
void main()
{
  int x, y, z, max;
  x=5; y=50; z=20;

  max = x>y&&x>z ? x : y>z ? y : z;

  printf("Max value is %d", max);
}
```

DELTA Pg No.
Date / /

NESTED If-else

* It is a procedure of constructing a condn within an existing conditional block.
* In C prog. lang., it is possible to place upto 255 nested blocks.

Syntax:

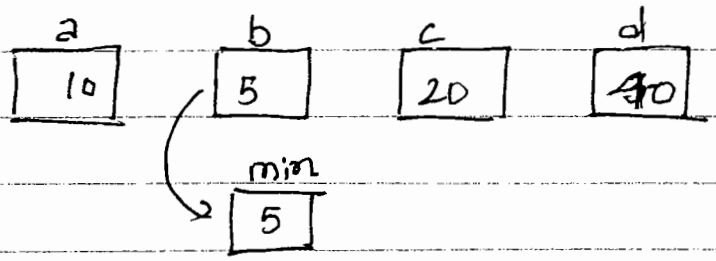


* According to syntax, if condition 1 is true then block 1 is executed, if it is false then control will pass to else part.
* Within the else part if condition 2 is true then block 2 will executed, if it is false then control will pass to nested else.
* Within the nested else, if condition 3 is true then block 3 will executed, if it is false then block 4 will be executed.

DELTA Pg No.
Date / /

* When we are working with nested if else at any point of time, only one block will be executed or can be executed.
* Nested concepts can be applied for if part and else part also
* If we are applying the nested concepts to if part then it is called nested if-else, if we are applying to else part, then it is called else if ladder

— Prog:




```
void main()
{
 int a,b,c, d , min ;
 clrscr();
 Printf ("Enter 4 values : ");
 Scanf (" %d %d %d %d ", &a, &b, &c, &d);
 if (a<b && a<c && a<d)
   {
    min = a;
    }
 else
    {
     if (b<c && b<d)
        {  min = b;
        }
```

```
else
  {
    if (c<d)
      min = c;
    else
      min = d;
  }
}
Printf ("min value is : %d", min);
getch();
}
```

O/P: Enter 4 values : 10 20 30 40
min value is : 10

* **Scanf**

- It is a predefined func^n which is declared in stdio.h
- By using scanf function we can read d data from users.
- When we are working with scanf function, it can take any no. of arguments, but first argument must be string const & remaining arguments r seperated with ',' .
- When we are working with scanf() func^n within the double quotes, we are required to pass proper format specifics only i.e. what type of data we are reading, same type of format specifier is required.
- Scanf() func^n will works with the help of call by address mechanism thats why every variable requires '&' symbol.

Syntax:-

int cdecl scanf (const char * format...);

* **clrscr**

- It is a predefined func^n which is declared in conio.h, by using this func^n we can clear the data from console.
- Using clrscr is always optional, it is recommended to place after declaration part only.

* **getch()**

→ It is a predefined func^n which is declared in conio.h, by using this func^n we can read a character from Keyboard.

→ Generally getch fun^s we are placing at end of the body because after printing the output, it can hold the screen until we are passing any character input.

Note: Conio.h is a compiler dependent header file i.e all the compilers doesn't supports conio.h

* **Comments**

→ When we are using comments in the prog. then that specific part of the prog. is ignored by compiler.

→ Generally comments are used to provide the description about the logic.

→ In C programming lang. we having 2 types of comments i.e

1) Single line
2) multi-line

DELTA Pg No.
Date / /

→ Single line comments can be provided by using //

→ Multi line comments can be provided by using /* ---- */

→ When we are coorking with multi line comments then nested comments are not possible i.e comments within the comment.

* In previous prog., in place of using nested if-else we can use single statement which can provide min. value also, i.e. conditional operators requried to use.

Prog:

```
void main()
{
 int a,b,c,d,min;
 clrscr();
 printf("Enter 4 values:");
 scanf("%d %d %d %d", &a, &b, &c, &d);
min = a<b && a<c && a<d ? a: d<c && b<d
      ? b: c<d ? c: d;
printf("min value is: %d", min);
getch();
}
```

O/P: Enter 4 values: 10 20 30 40
min value: 10

Ques Write a prog. to calculate electricity bill value by using following info

1) Serial no. consider as 4235
2) Tariff data is 1 – 50 1.75
51 – 150 3.75

DELTA PG NO.
DATE / /

//	151 – 250 5.00
	>= 251 6.50
3)	Previous reading value is 1234
4)	min. charged amt. is 40 rs.
5)	Service tax need to applied to 12.36%

~~void main()~~

```
#include <stdio.h>
#include <stdlib.h>
{
 int sno, cread, pread = 1234, nunits, t;
float ops;
 printf ("Enter sno:");
 scanf ("%d", &sno);
if (sno = 4321)
  {
   printf ("Enter current reading:");
   scanf ("%d" &cread);
   if (cread > = pread)
{
   nunits = cread - pread;
    if (nunits > 251)
   {
      t = nunits - 250
      ops = t* 6.50                    //4th
      ops = ops + 100*5.00;            //4+3rd
      ops = ops + 100*3.75;            //4+3+2nd
      ops = ops + 50*1.75;             //4+3+2+1st
   }
```

DELTA Pg No.
Date / /

```
else if (nunits >= 151)
  {
    t= nunits - 150
    rps = t * 5.00 // 3rd
    rps = rps + 100 * 3.75 ;  // 3+2nd
    rps = rps + 50 * 1.75   // 3+2+1st
}
else if (nunits >= 51)
    {
      t = nunits - 50;
          rps = t * 3.75;
          rps = rps + 50 * 1.75;
    }
    else
      rps = nunits * 1.75;
   if (rps < 40);      // Min amount
  rps = rps + ( rps * 12.36 / 100)
   printf ("Total Amount : %.2f \n", rps);
   }
else
   printf (invalid cread value\n");
else
   Printf (invalid sno\n");

return EXIT-SUCCESS; // retun 0;

}
```

// Compil & Link command : gcc -o ebill ebill.c

// Run / Load command : ./ebill

O/p : Enter sno : 4321

Enter current reading : 1234

Total Amount : 44.94

DELTA Pg No.
Date / /

* LOOPS

⇒ Set of instructions into the compiler to execute set of statements until the condition become false it is called loop.

⇒ Basic Purpose of loop is code repetition

⇒ In implementation when the repetitions are required then recommended to go for loops.

⇒ Generally iterative statements are called loop bcz way of the repetition forms a circle.

⇒ In 'C' prog. lang., loops are classified into 3 types
1) while loop
2) for loop
3) do-while loop

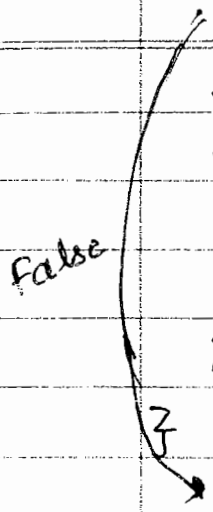
(1) While loop

- When we are working with while loop pre-checking process is occured i.e before execution of statements block condition part is executed.
- While loop always repeats in clock direction.

Syntax :-

```
Assignment;
while (condition)
{
```

True



→ Acc. to syntax "while" cond. is true then control will pass within the body.

→ After execution of the body, once again control will pass back to the cond^n and until the condition became false body will be repeated n no. of times.

→ when while condition become false, then control will pass outside of the body, if cond^n is not false, then it becomes an infinite loop.

```
void main()
{
  int i ;
  i = 1 ;
while (i<=10)
{
  printf ("%d", i);
  i = i+2 ;
}
}
```

O/P : 1 3 5 7 9

DELTA Pg No.
Date / /

* **for loop :-**

When we are working with for loop it contains 3 parts -

1) Initialisation
2) Condition
3) Iteration

Syntax :- for (initialization ; condition ; iteration)
{
statement block ;
}

- When we are working with for loop, always execution process will start from initialisation.
- Initialisation part will be executed only once when we are passing the control within the body first time.
- After execution of initialisation part, control will pass to condition, if condn evaluated is true, then control will pass to statement block.
- After execution of statement block, control will pass to iteration, from iteration once again it will pass block to condition.
- Always repetition will come b/w condn, statement block and iteration only.
- When we are working with for loop, everything is optional but mandatory to place 2 semicolons.

Pa

while () ⟶ error

for (; ;) ⟶ valid

- When the condition part is not given in for loop, then it repeats infinite times beoz condn part is replaced with non-zero value (True value).
- When we are working with for loop, it repeats in anticlock direction.
- Always prechecking process occurs when we are working with for loop i.e by execution of statement block condn part is executed.

while (0) ⟶ No repetition

for (;0,) ⟶ 1's it will repeat

- In for loop, when d condn part is replaced with constant 0 then it repeats once, bcz no. of instances became 1 at the time of compilation.

```
int i;
i = 0;
while (i)  -> No repetition
for (; i;) -> No rep.
```

↳ Condn part shud be const 0 or NULL not a variable.

Prog:

```
void main()
{
```

DELTA Pg No.
Date / /

```
int i;
for (i = 1; i <= 10; i = i+2)
printf ("%d", i);
}
```

O/P : 1 3 5 7 9

3) **do - while :-**

- In implementation when we required to repeat the statement block atleast once then go for do-while loop.
- When we are working with do-while loop, post checking process occurs, i.e after execution of statement block, condⁿ part is executed.
- When we are working with do-while loop, it repeats in clock direction.

Syntax :

```
Assignment ;
do
{
  statement 1 ;
  statement 2 ;
  statement 3 ;
  - - -
  - - -
  inc / dec ;
} while (condition);
```

- Acc. to syntax, semicolon must be required at the end of the body.

Prog:-

```
void main()
{
  int i;
  i = 1;
  do
  {
    printf("%d", i);
    i = i + 2;
  }
  while (i<=10);
}
```

O/P: 1 3 5 7 9

Working with while loop

(1) Increment order

- Assignment statement should contain min value
- Condition → max
- relation → < / <=
- control → Add (+)

Ex: 2 4 6 8 10 12 14 16 18 20

(2) Decrement Order

- Assignment → max
- Condition → min
- relation → > / >=
- Control → sub (-)

DELTA Pg No.
Date / /

Ex: 25 23 21 19 17 15 13 11 9 7 5 3 1

Task 1

2 4 6 8 10 12 14 16 18 20

```
void main()
{
   int i ;          // initially garbage value
   i = 2 ;
   while ( i<= 20)
   {
      printf("%d", i);
      printf("%d",
      i = i+2 ;
   }
}
```

Task 2

25 23 21 19 17 15 13 11 9 7 5 3 1

```
void main()
  {
    int i ;
     i= 25;
   while ( i>= 1)
     {
      Printf ("%d",i);
      i= i-2 ;
     }
  }
```

Prog :- Enter a value : 30

1 3 5 6 7 9 11 12 13 15 17 18 19 21 23 24

25 27 29 30.

```
void main()
{
  int i, a
  i = 1;
  Printf ("
          ", &a);
  while (i<=a)
  {
    printf("  ", i);
    i = i + 1.
    if (i%2==0)
    {
      continue;
    }
  }
}
```

OR

```
void main()
{
  int i, n, count = 0;
  clrscr();
  printf("Enter a value : ");
  scanf("%d", &n);
```

```
i = 1;
while (i<=n)
{
printf ("%d ", i);
count = count + 1;
if (count == 3 && i != n)
{
printf ("%d ", i+1);
count = 0;
}
i = i+2;
}
getch ();
```

Prog [Fibonacci Series]

Enter a value : 100

0 1 1 2 3 5 8 13 21 34 55 89

```
void main()
{
int i = 0;
int j = 1;
int a = 1;
printf ("%d %d", i, j);
a = 1;
while (a<=n)
{
i = j;
j = a;
```

```
int n, a;
Printf ("Enter a value: ");
Scanf ("%d", &n);
```

i → x2

j → x3

a → x5

```
        a = i+j;
      }
    }
```

x

Again

```
void main()
{
  int n,x,y,z;
  clrscr();
  printf("Enter a value: ");
  scanf("%d", &n);
  if (n>0)
{
   x=0;
   y=1
    printf("%d %d", x, y);
    z=1;     // x+y=z;
  while (z<=n)
  {
    printf("%d", z);
    x=y;
    y=z;
    z=x+y;
    }
  }
else
  printf("input value should be >0");
  getch();
}
```

DELTA Pg No. Date / /

Prog1: Write a Program to print first n no. of fibonacci values.

How many values you want to print? – 15

O/P : 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377.

Prog2: Write a prog to print fibonacci series b/w given two input values.

→ Enter 2 values: 10 200

13 21 34 55 89 144

→ Enter 2 values : 5 200

5 8 13 21 34 55 89 144

Prog 1

```
void main()
{
 Int n, x, y, z, count = 2;
 clrscr();
 printf("How many values you want to print ? :");
 scanf("%d", &n);
    x = 0;
    y = 1;
   printf("%d%d",x,y);
    z = 1;
while (count < n)
  {
   printf("%d", z);
   x = y;
   y = z;
   Z = x + y;
```

```
        count ++;
    }
    getch();
}
```

given

Auto-alignment concept

DELTA Pg No.
Date / /

Prog

```
Enter no. of rows : 10
Enter a value: 5
        5 * 1 = 5
        5 * 2 = 10
        5 * 3 = 15
        5 * 4 = 20
        5 * 5 = 25
        5 * 6 = 30
        5 * 7 = 35
        5 * 8 = 40
        5 * 9 = 45
        5 * 10 = 50
```

```
void main()
{
 int r, n, i;
 clrscr();
 printf("Enter no. of rows : ");
 scanf ("%d", &r);
 if (r>= 1 && r<= 25)
  {
    printf (" Enter a value: ");
    scanf ("%d", &n);
    i = 1;
    while (i<=r)
    {
      printf ("%d * %2d = %2d", n, i, n * i)
      i= i+1;
    }
}
```

Auto alignment

DELTA Pg No.
Date / /

* \n

- It is a special character which is used to print data in vertical format.
- When we are using %.2f format specifier it allows to print

Q Write a program to print all even nos b/w 2 even input values with irrespective of i/p data.

Case 1:

Enter 2 values : 10 20 (n1, n2)

10 12 14 16 18 20

or

Case 2: Enter 2 values : 9 20 (n1, n2)

10 12 14 16 18 20

Case 2: Enter 2 values : 20 10 (n1, n2)

20 18 16 14 12 10

or

Enter 2 values : 21 10 (n1, n2)

20 18 16 14 12 10

case 3: Enter 2 values : 10 10

Both values are same

```
else
    printf ("invalid row value (1-25)");
    getch ();
}
```

- In printf statement, when we are using "%2d" format specifier then it indicates that 2 digit decimal values required to print, if 2 digits are not occur then go for right alignment, i.e digits will be printed towards right side and space will be printed towards left side.

- When we are using "%-2d" format specifier then it indicates 2 digit decimal value required to print, if 2 digits are not occur then go for left alignment i.e digit will be printed towards left side and space will be printed towards right side

- When we are using "%5.3d" format specifier then it indicates 5 digit decimal value with right alignment but mandatory to print 3 digits. If 3 digits are not occur then fill with zeros.

- When we are using "%-5.3d" format specifier then it indicates 5 digit decimal value with left alignment but mandatory to print 3 digits, if 3 digits are not occur, then fill with zeros

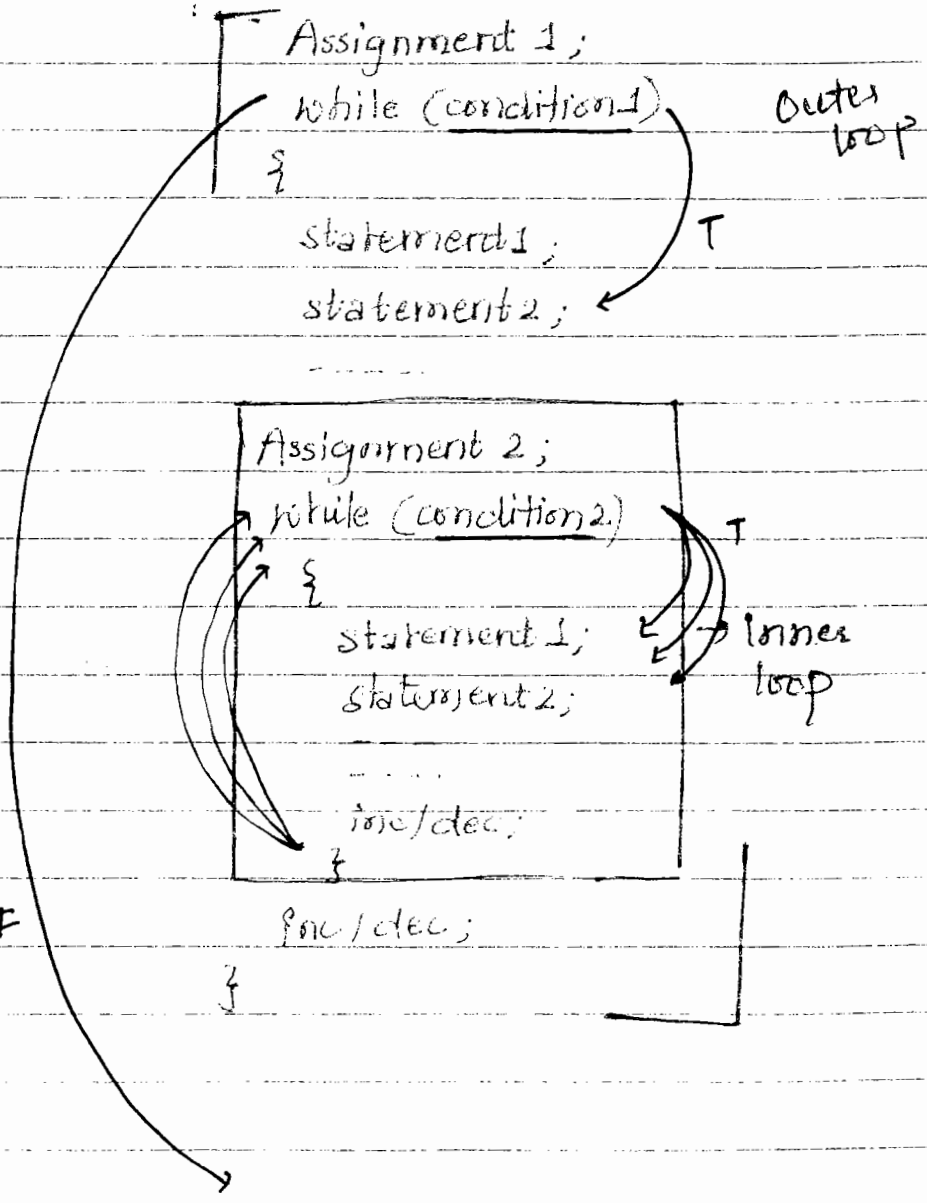
Auto fill

DELTA Pg No.
Date / /

* NESTED LOOPS

- It is a procedure of constructing a loop within an existing loop body.
- When the repetitions are required then go for loops, if complete loop body required to repeat n no. of times then go for nested loop.
- In C prog. lang., we can place upto 255 nested blocks.

Syntax:



DELTA Pg No.
Date / /

- When we are working with nested loops, always execution is started from outer loop condⁿ i.e Condⁿ 1
- When the outer loop condition is true, then control will pass to outer loop body.
- In order to execute the outer loop body, if any while statements occur those are called inner loops.
- When the inner loop occurs, we required to check inner loop condⁿ i.e condⁿ 2.
- If Inner loop condⁿ is true then control will pass within the inner loop body and until the inner loop condition becomes false, body is repeated n no. of times, when inner loop condition is false then control will pass to outer loop and until the outer loop condition became false body is repeated n no. of times.

Task

Prog:

Enter 2 values : 2 5

2*1 = 2	3*1 = 3	4*1 = 4	5*1 = 5
- - -	- - - -	- - -	- - -
2*10 = 20	3*10 = 30	4*10 = 40	5*10 = 50

Try

```
void main()
{
 int n, n1, n2, i;
 clrscr();
 printf("Enter 2 values:");
 scanf("%d %d", &n1, &n2);
```

```
i = 1;
while((i<=10))
{
  printf("\n");
  n = n1
  while(n<=n2)
  {
    printf("%3d * %2d = %2d", n, i, n*i);
    n = n+1;
  }
  i = i+1;
}
getch();
}
```

i: 1, 2, 3, ..., 10

2 * 1 = 2

2 x 2 = 4

n: 2, 5

Task 1

Enter a value : 6

```
1
1 2
1 2 3
1 2 3 4   → outer loop
1 2 3 4 5
1 2 3 4 5 6   → inner loop
```

Try.

DELTA Pg No.
Date / /

```
void main()
{
  int n,i,in;
  clrscr();
  printf("Enter a value:");
  scanf("%d",&n);
  i=1;
  while(i<=n)
  {
    printf("\n");
    in=1;
    while(in<=i)
    {
      printf("%d",in);
      in=in+1;
    }
    i=i+1;
  }
  getch();
}
```

- When we are working with pattern related programs then recommended to split the program into 3 partitions –
  1) Inner loop order
  2) outer loop order.
  3) making the inner loop & outer loop

→ In order to design inner loop logic, we required to consider only 1 row where we having max no. of elements.

→ In order to design outer loop logic, we required to consider changing value sequence of every row.

→ In order to make the relation b/w inner loop and outer loop we required to place outer loop variable in inner loop logic.

DELTA Page No. Date / /

- If startup values are changing then inner loop assignment contains outer loop variable
- If ending values are changing then inner loop condition contains outer loop variable.

Task 2:

Enter a value : 6

Decrement order

6 5 4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1

```
void main()
{
  int n, i, dn;
 clrscr();
 printf("Enter a value:");
 scanf("%d", &n);
 i=n;
 while (i>=1)
 {
   printf("\n");
   dn = i;
   while (dn>=1)
    {
      printf("%d", dn);
```

```
        dn = dn - 1;
    }
    i = i - 1;
  }
  getch();
}
```

Task3:

```
Enter a value : 6
1 2 3 4 5 6
2 3 4 5 6
3 4 5 6
4 5 6
5 6
6
```

Task4:

```
Enter a value : 6
6 5 4 3 2 1
6 5 4 3 2
6 5 4 3
6 5 4
6 5
6
```

Task5:

```
Enter a value : 6
1
1 *
1 * 3
1 * 3 *
1 * 3 * 5
1 * 3 * 5 *
```

DELTA Pg No.
Date / /

Task6: Enter a value: 6

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
```

Task7: Enter a value: 5

```
1
* *
1 2 3
* * * *
1 2 3 4 5
* * * * * *
```

Task8: Enter a value: 6

```
1
* 2
1 * 3
* 2 * 4
1 * 3 * 5
* 2 * 4 * 6
```

Task9: Enter a value: 8

```
*
* *
* 2 *
* 2 3 *
* 2 3 4 *
* 2 3 4 5 *
* 2 3 4 5 6 *
* * * * * * * *
```

Task 10: Enter a value: 6

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
0 1 0 1 0 1
```

Task 11: Enter a value: 6

```
1
2  3
4  5  6
7  8  9  10
11 12 13 14 15
16 17 18 19 20 21
```

Task 12 Enter a value: 6

```
1
2  7
3  8  12
4  9  13 16
5  10 14 17 19
6  11 15 18 20 21
```

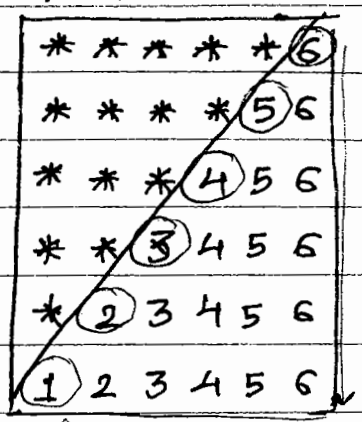
Task 13 Enter a value: 5

```
0
1  1
2  3  5
8  13 21 34
55 89 144 233 377
```

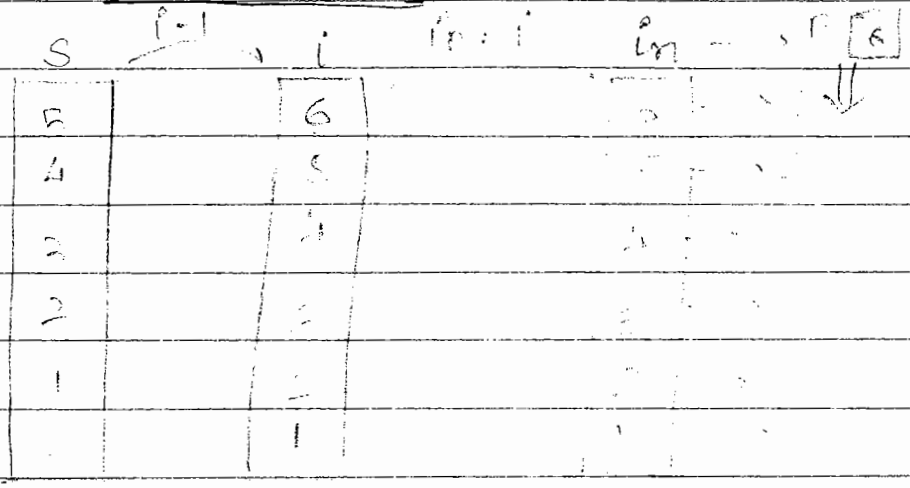
15/6/2015

DELTA | Pg No. | Date / /

Task 14: Enter a value : 6

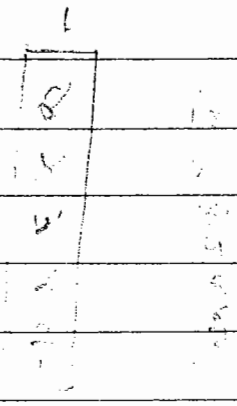


n=6

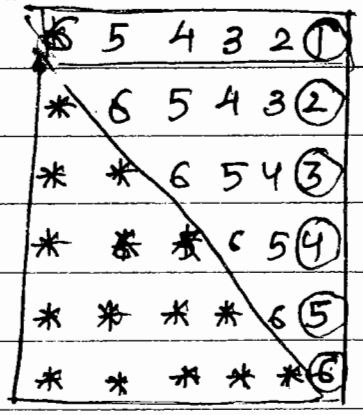


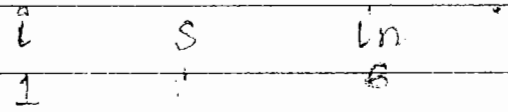

```
void main()
{
  int n, i, in, s;
  clrscr();
  printf("Enter a value : ");
  scanf("%d", &n);
  i = n;
  while (i>=1)
  {
    printf("\n");
    s = 1;                    // s = i-1;
    while (s<= i-1)           // while (s>=1)
    {
```

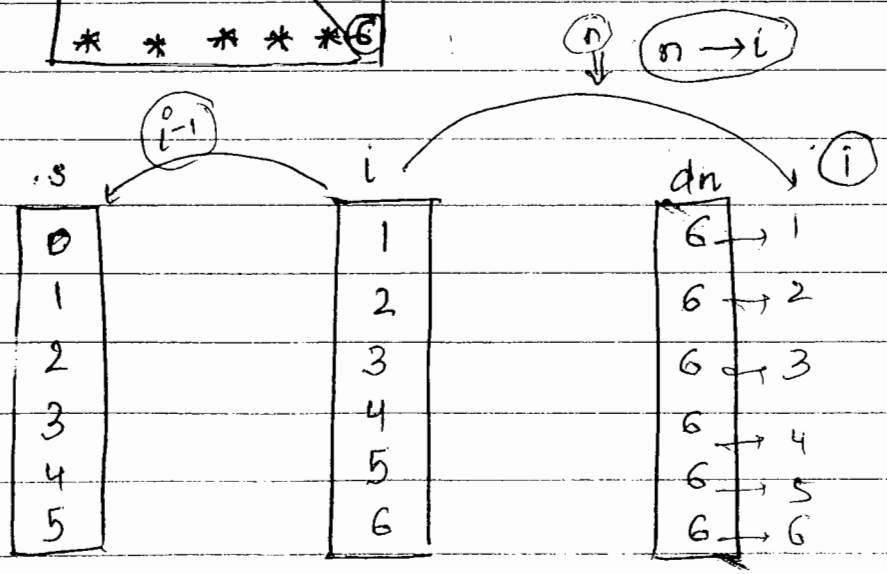
```
while (s <= i-1)
    printf ("*");
    s=s+1;          //s=s-1;
}
in = i;
while (in <= n)
{
    printf ("%d", in);
    in = in+1
}
i=i-1;
}
getch();
}
```



Task 15: Enter a value : 6



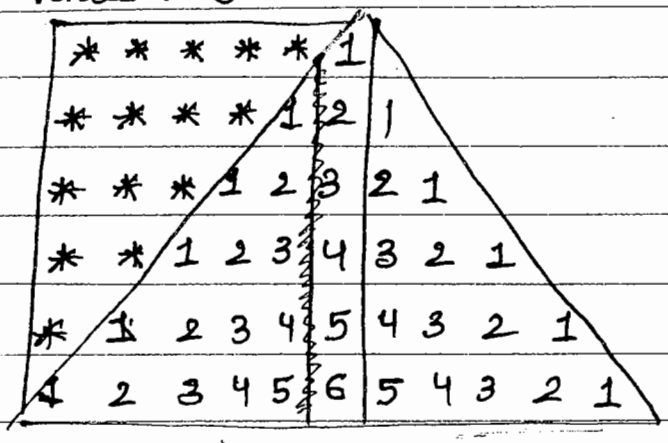


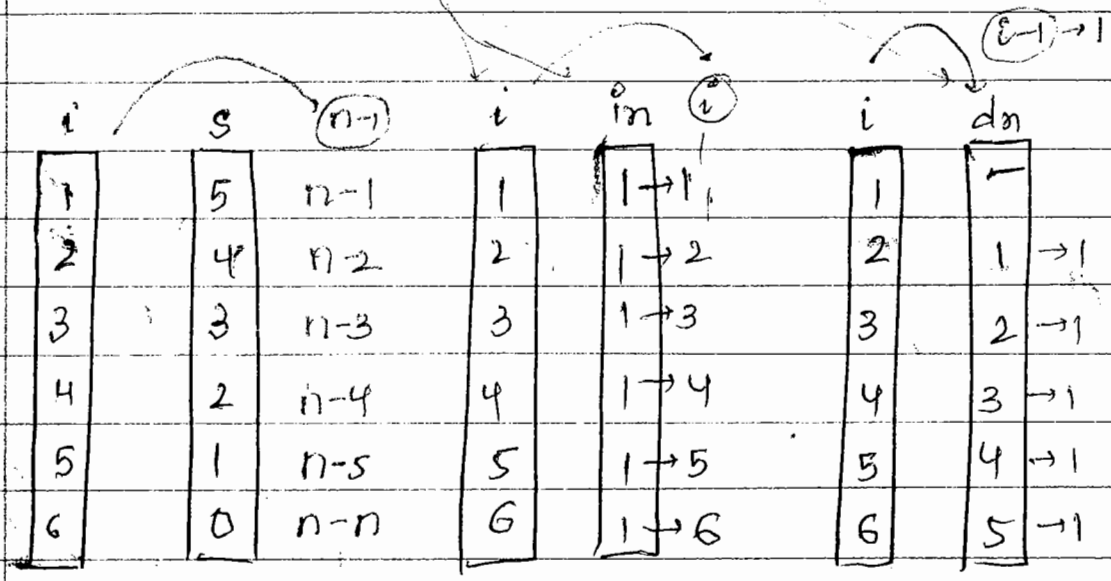


```
void main()
{
 int i, n, dn, s;
 clrscr();
 printf("Enter a value: ");
 scanf("%d", &n);
 i=1;
 while (i<=n)
  {
    printf("\n");
    s=i-1;
    while (s>=1)
    {
      printf("*");
      s=s-1;
    }
 dn=n;
 while (dn>=i)
  {
    printf("%d", dn);
     dn = dn - 1;
    }
     i = i+1;
  }
  getch();
}
```

DELTA | Pg No. | Date / /

Task 16. Enter a value : 6





Tas

```
void main()
 {
  int n, i, s, in, dn;
  clrscr();
  printf (" Enter a value : ");
  scanf ("%d", &n);
  i = 1;
  while ( i<=n)
{
  printf ("\n");
  s = 1;
```

Fa

```
while (in<=i)
{
 printf ("%d", in);
 in = in+1;
 }
 dn = i-1;
 while (dn>=1)
  {
   printf ("%d", dn);
   dn = dn-1;
  }
 i = i+1;
}
 getch();
}
```

Task17:

```
Enter a value: 6
6 5 4 3 2 1 2 3 4 5 6
* 6 5 4 3 2 3 4 5 6
* * 6 5 4 3 4 5 6
* * * 6 5 4 5 6
* * * * 6 5 6
* * * * * 6
```

DELTA Pg No.
Date / /

Task 18: Enter a value : 6

```
1 * * * * * * * * * * 1
1 2 * * * * * * * * 2 1
1 2 3 * * * * * * 3 2 1
1 2 3 4 * * * * 4 3 2 1
1 2 3 4 5 * * 5 4 3 2 1
1 2 3 4 5 6 6 5 4 3 2 1
```

Task 19: Enter a value : 6

```
1 2 3 4 5 6 6 5 4 3 2 1
1 2 3 4 5 * * 5 4 3 2 1
1 2 3 4 * * * * 4 3 2 1
1 2 3 * * * * * * 3 2 1
1 2 * * * * * * * * 2 1
1 * * * * * * * * * * 1
1 2 * * * * * * * * 2 1
1 2 3 * * * * * * 3 2 1
1 2 3 4 * * * * 4 3 2 1
1 2 3 4 5 * * 5 4 3 2 1
1 2 3 4 5 6 6 5 4 3 2 1
```

Task 20:

```
1           1
  1       1
    1   1
      1
    1   1
  1       1
1           1
```

DELTA Pg No.
Date / /

Task21:

```
1           1
  2       2
    3   3
      4
    5   5
  6       6
7           7
```

Task22: Enter a - value : 6

Task 22:-

```
void main()
{ int i, j, a, s, n;
clrscr();
printf("Enter a value");
scanf("%d", &n);
i=1;
```

```
while (i<=n)
{
  printf("\n");
   s= n-i
  while (s>=1)
   {
     printf("*");
       s--;
   }
  j=1;
  while (j<=i)
  {  printf ("%d", j);
       j++;
   }
  a = i-1
  while (a>=1)
  {  printf ("%d", a);
       a--;
    }
  i++;
}
  i=n-1;
 while (i>=1)
{
  printf ("\n");
  s=1
  while (s<=n-i)
    {
      printf ("*");
        s++;
    }
```

```
j=1;
while (j<=1)
 {
  printf ("%d", j);
   j++;
 }
 a= i-1;
  while (a>=1)
   {
    printf ("%d", a);
      a--;
   }
i--;
 }
  getch();
}
```

Unary Arithmetic Operators

- In implementation, when we require to modify initial value of a variable by 1, den go for increment, decrement operators i.e ++, --
- When we are working inc/dec operators then difference between existing value and new value is +1 or -1 only.
- Depending on the posn, these operators are classified into 2 types.

DELTA Pg No.
Date / /

Pre operators	Post operators
• When the symbol is available b4 d operand.	When the symbol is available after d operand.
• When we are working wid pre operators, data is required to modify by evaluating d expression.	
• When we are working wid post operators, then data is required to modify after evaluating d expression.	

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

Syntax :- `b = ++a;` pre increment.

First increment the value of a by 1, den evaluate d expression.

i.e `b = a;`

O/P :- a = 2 b = 2

Syntax 2 :- `b = a++;` post increment

First evaluate d expression, den increment d value of a by 1.

O/P :- a = 2, b = 1

Syntax 3 :- `b = --a;` pre decrement

First decrement d value of a by 1, den evaluate d expression.

O/P :- a = 0 b = 0

DELTA Pg No.
Date / /

Syntax 4:- b = a-- post decrement

First evaluate the expression, den decrement d value of a by 1.

O/P :- a=0 b=1

nd.

is

Order of priority :-

1. ( )
2. +, −, !, ++, -- (signs) pre
3. *, /, %
4. +, − (Arithmetic operations)
5. <, >, <=, >=
6. ==, !=
7. &&
8. ||
9. ?:
10. =
11. ++, -- post

⇒ The behaviour of inc/dec operators changes from compiler to compiler

→

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

O/P = 9

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

O/P = 9

DELTA | Pg No.
Date / /

- There is no diff b/w pre & post operators, until we are assigning the data to any other variable.

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

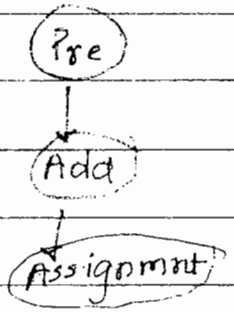
O/P :- 12

$a = ++a + ++a + ++a;$

$a = a + a + a;$

a

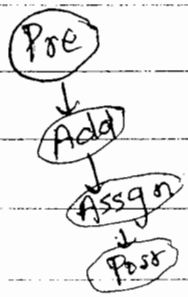
~~2~~ 3 4 ⇒ a=4



firstly evaluate all the pre operators then substitute the value.

Thus, a=12

→
```
void main()
{ int a;
   a=5;
   a = --a + a-- + --a;
   printf ("%d", a);
}
```



a
5 4 3

$a = a + a + a$

$3 + 3 + 3$

$a = 9$

a=8

a
10

DELTA Pg No.
Date / /

util
ble.

→
```
void main()
{ int a;
  a = 1;
  a = a++ + ++a + a++;
  printf ("%d", a);
```

$a = a + a + a$

$= 2 + 2 + 2$

$a = 6$ after post $a = 8$

→
```
void main()
{
  int a, b;
  a = b = 50;
  a = a++ + ++b;
  b = ++a + b++;
  printf ("a = %d  b = %d", a, b);
}
```

$a = a++ + ++b;$

$a = a + b$ ✓

$= 50 + 51;$

$= 101$

after post ⇒ $a = 102$

a	b
102	154

$b = a + b$

$b = 103 + 51$

$b = 154$ ⟶ after post = 155

DELTA Pg No.
Date / /

→
```
void main()
{
 int a, b;
 a = b = 5;

 a = a-- + --b;
 b = --a + b--;
 printf ("%d %d", a, b);
}
```

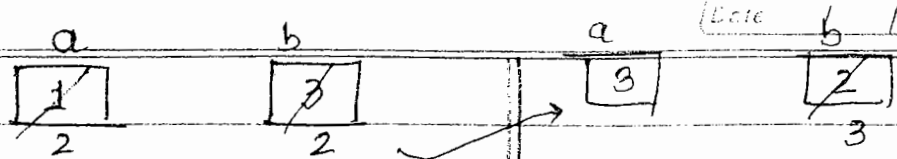
a = a + b
a = 5 + 4
a = 9 - 1 ⇒ a = 8 → Then
b = 4

b = a + b
b = 7 + 4
b = 11

O/P:- a = 7 b = 10 ← b = 10

→
```
void main()
 {
 int a, b;
 a = 1; b = 3;
 a = ++a - --b + a++;
 b = ++b - a-- + b++
 printf ("a = %d b = %d", a, b);
}
```

DELTA Pg No.
Date / /



$a = a - b + a$

$2 - 2 + 2$

$a = 2$

$b = b - a + b$

$b = 3 - 3 + 3$

$b = 3$

after post $\boxed{a = 3}$

After that $\boxed{b = 4}$

$a = 3$ but $a--$

$\boxed{a = 2}$

$\boxed{b = 4}$

→
```
void main()
{
int a,b,c
a=2, b=4, c=6;
a = ++a + b++ - --c;
b = a++ - --b + c--;
c = --a + ++b - c++;
printf ("%d %d %d", a,b,c);
}
```

For a

a	b	c
2	4	6

$a = a + b - c$

$a = 3 + 4 - 5$

$\boxed{a = 2}$

DELTA Pg No.
Date / /

for b :-

a	b	c
~~2~~ 3	5	~~5~~ 4

$b = a - b + c$

$= 2 - 4 + 5$

$\boxed{b = 3}$

for c :-

a	b	c
3	3	4

$c = a + b - c$

$= 2 + 4 - 4$

$\boxed{c = 2}$

DELTA | Pg No.
Date / /

II model

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

stack (LIFO) approach

passes this way.

⇒ printf is a predefined function, by using printf function, we can print the data on console.

⇒ When we are working with printf function, it works with the help of stack i.e LIFO approach.

⇒ When we are working with printf function, always arguments require to pass, from right to left and data is required to be printed from left to right

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

pre (+1) post (+1)

remains same

O/P :- 6 4 4

DELTA Pg No.
Date / /

→
```
void main ()
{
  int a;
  a=5;
  printf ("%d %d %d", a--, --a, a--);
  printf ("\n a=%d", --a);
}
```

pre-subtract two.

O/P :-

3	3	5

1

DELTA Pg No.
Date / /

→ void main()
{
int a;
a = 5;
printf ("%d %d %d", ++a, a=10, ++a);
}

O/P :- 11 10 6

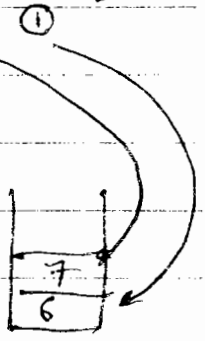
we can assign values inside printf

→ void main()
{
int a;
a = 5;
printf ("%d %d %d", --a, --a==3, --a);
}

O/P :- 2 1 4

→ void main()
{
int a;
a = 5;
printf ("%d %d %d", ++a, ++a);
}

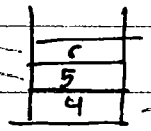
O/P: 7 6 gb



For printing, 7 goes first then 6, then gb.

DELTA Pg No.
Date / /

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

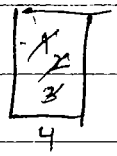


O/P :- 6 5

4 remains in stack.

```
void main()
{
 int a=1, b, c=1, d;
  int b = ++a * ++a * ++a;
  d = ++c * ++c * ++c;
 printf("a=%d b=%d c=%d d=%d", a, b,
                              c, d);
}
```

O/P : 4 24 4 64



int b = ++a * ++a * ++a ← initialisation

2 * 3 * 4

24

works acc. to stack

a=4 b=24

in stack pre post both are same.

d = ++c * ++c * ++c ← assignment

d = c * c * c

4 * 4 * 4

64

works acc. to register

a=4 d=64

DELTA Pg No.
Date / /

1. Any kind of expressions are evaluate in 2 locations i.e 1) stack evaluation
2) Register evaluation

2. Acc. to stack evaluation, pre and post operator both are having same priority

3. In stack evaluation, data required to substitute at the time of evaluating the expression only.

4. In Register evaluation, pre & post operator both are having diff. priority i.e pre operator having highest priority than post operator.

5. In Register evaluation, data required to substitute after modifying all three values.

→
```
void main()
{
 int a=1, c=1, d;
 int b= a++,* ++a * a++;
 d = c++ * ++c * c++;
printf("a=%d b=%d c=%d d=%d", a,b,c,d);
}
```

a = 4 b = 9 c = 4 d = 8

→
```
void main()
{
 int a=1, c=1, d;
 int b = ++a + a++ + ++a;
    d = ++c + c++ + ++c;
```

printf ("a = %d b = %d, c = %d d = %d", a,b,c,d);
}

O/P: 4 8 4 9

→ void main()
{
int a;
a = 1;
printf ("\n%d", ++a * ++a * ++a);
a = 1;
printf ("\n%d", a++ * ++a * a++);
a = 1;
printf ("\n%d", ++a * a++ * ++a);
a = 1
printf ("\n%d", a++ * a++ * a++);
a = 1
printf ("\n%d", a++ * a++ * ++a);
a = 1
printf ("\n%d", ++a * a++ * a++);
}

O/P :

// 2 * 3 * 4 = 24
// 1 * 3 * 3 = 9
// 2 * 2 * 4 = 16
// 1 * 2 * 3 = 6
// 1 * 2 * 4 = 8
// 2 * 2 * 3 = 12

d);

→ When we are working with printf statement, always evaluation is required to take place acc. to stack

→ In printf statement if we are passing single expression then evaluation is required to take place from left to right, multiple expression if we are passing right to left.

```
void main()
{
  int a, b;
  a=1; b=3;
  printf("\n%d %d", ++a* b++, a++*++b);
  a=2; b=4;
  printf("\n %d %d", --a + --b, a-- + b--);
  a=3; b=2;
  printf("\n%d %d", a-- + b++, ++a * b++);
}
```

O/P : 12 4
2 6
7 8

DELTA Pg No.
Date / /

INTEGER

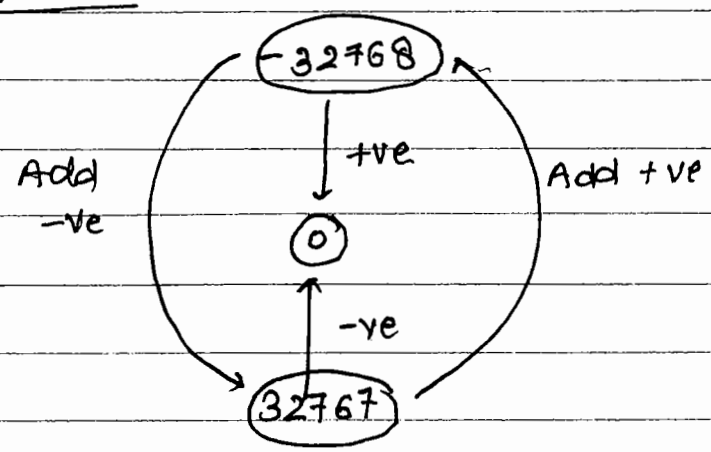
→ On doc based compiler, size of ~~com~~ integer is 2 bytes and range from -32768 to +32767

→ When we are working with integer variable then always data required to store within the limit only.

→ When we are crossing the limit, then automatically control will pass to opp. direc^{n}.

→ When we are crossing the max. +ve value, then it provides negative data, when we are crossing min. -ve value then it provides +ve data.

int i; / short i;



value	int value
32767	32767
32767 + 1	-32768
32767 + 2	-32767 (-32768 + 1)
32767 + 3	-32766 (-32768 + 2)
32767 + 11	-32758 (-32768 + 10)
-32768	-32768
-32768 - 1	+32767
-32768 - 2	+32766 (+32767 - 1)
-32768 - 3	+32765 (+32767 - 2)
-32768 - 11	+32757 (+32767 - 10)

→ V

DELTA Page No.
Date / /

→ Void main()
{
int a, b;
a = 200 * 200 / 200;
b = 200 / 200 * 200;
printf ("a = %d b = %d", a, b);
}

O/P : a = -127 b = 200

200 * 200

40000 -32768

32767 (n-1) + 7232

7233 -25536

a = 200 * 200 / 200
= -25536 / 200
= $\frac{-25536}{2} \times 10^{-2}$
= -12768×10^{-2}
= -127.68

a = -127

b = 200 * 200 * 200
) * 200

→ Void main()
{
int a, b;
a = 300 * 200 / 300;
b = 300 / 200 * 300;

DELTA Pg No.
Date / /

60000 → 32767 → 27233

-32768 → 27232 → -5536

$a = 360 * 200 / 300;$

$= -5536 / 300$

$= \frac{-5536}{3} \times 10^{-2}$

$= -1845$

$a = -18$

$b = 360 / 200 * 300;$

$= 1 * 300$

$b = 300$

→
```
void main()
int a;
a = 32767;
 if (++a < 32767)   // if (-32768 < 32767)
   printf ("Welcome %d", a);
 else
   printf ("Hello %d", a);
}
```

O/P: Welcome -32768

→
```
void main()
{ int i
 i = -32768;
  if (-i > -32768)   // if (32767 > -32768)
```

DELTA Pg No.
Date / /

```
    printf ("Welcome %d", i);
else
    printf ("Hello %d", i);
}
```

O/P : Welcome 32767

→
```
void main()
{ int i;
  i = 32767;
  if (i++ < 32767)        // if (32767 < 32767)
     printf ("Welcome %d", i);
  else
     printf ("Hello %d", i);
}
```

O/P: Hello -32768

→
```
void main()
{ int a, b;
  a = b = 1;
  while(a)
  { a = b++ <= 3
    printf ("\n %od %od", a, b);
  }
  printf ("\n a = %d  b = %d", a+10, b+10);
}
```

O/P: ~~1 1 / 1 2~~ a = 11 b = 13

O/P: 1 2
1 3
1 4
0 5
a = 10 b = 15

DELTA | Pg No.
Date / /

a = b++<=3

1 <= 3

∴ b = 2

a = 1 and b = 2

Then 2 <= 3 ∴ b = 3

a = 1 and b = 3

Then 3 <= 3 ∴ b = 4

a = 1 and b = 4

Then 4 <= 3 ∴ b = 5

and a = 0

a = 10 b = 15

→ void main()

{ int a, b;

a = b = 5;

while (a)

{ a = ++b <= 8;

printf ("\n%d %d", a, b);

}

printf ("\n a = %d b = %d", a+10, b+10);

}

a = ++b <= 8

5 <= 8

b = 6

a = 1 and b = 6

6 <= 8

O/P: 1 6

1 7

1 8

0 9

a = 10 b = 19

```
void main()
{
  int a;
  a = 10;
  a * 5;        → dummy statement
  printf("a%d", a);
}
```

O/P: a=10

→ When we are not collecting the value of any expression or if we are not using value of the xpression then it is called dummy statement.

→ When we are working with dummy statement compiler doesn't gives any error but it gives a WARNING msg i.e code has no effect.

```
void main()
{ int a;
  a = 5;
  a * 10;        → dummy statement
  printf("%d %d %d", a, a*2, a);
}
```

O/P: 5 10 5

```
void main()
{
  printf("A");
  if (578);      ← dummy condition
{
  printf("B");
  printf("C");
}
  printf("NIT");
}
```

It will not consider this condition.

O/P: ABC NIT

DELTA Pg No.
Date / /

- When we are placing the semicolon at end of the if then it is called dummy condition.
- When dummy conditions are created then compiler creates new body without any statements and current body comes outside of the condition.
- When we are working with dummy conditions, if the cond is true or false always correct body is executed.

```
void main()
{ printf ("A");
  if (8>5)
{ printf ("B");
  printf ("C");
} else ;          <-- dummy else
{
  printf ("NIT");
  printf ("D");
}
}
```

O/P: ABCNITD

→ When we are placing semicolon at end of else then it is called dummy else part.

→ When we are working with dummy else part then after execution of if part also else part is executed.

→
```
void main()
{ int i = ;
    i = 1;
```

DELTA | Pg No. | Date / /

```
while (i<=10)
  { printf ("%d", i);
    ++i;
  }
}
```

O/P : 1 2 3 4 5 6 7 8 9 10

→
```
void main()
{ int i;
i=1;
while (i<=10)
  printf ("%d ", i);
++i;
}
```

scope is close here

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

O/P : 1 1 1 1 ---- inf loop

→ When the body is not constructed for the loop then only 1 statement will be placed inside the body and until the condition became false single line statement only will be executed & whenever the condition will become false, den automatically control will pass outside d body. If it is not false, den it becomes infinite loop.

```
void main()
{ int i;
  i = 1;
  while (i<=10);
  { printf ("%d", i);
    ++i;
  }
}
```

dummy loop

O/P: No output with inf loop

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

DELTA | Pg No.
Date / /

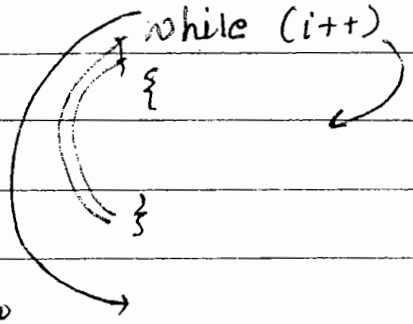
→ when we are placing the semicolon at end of the while then it became dummy loop.

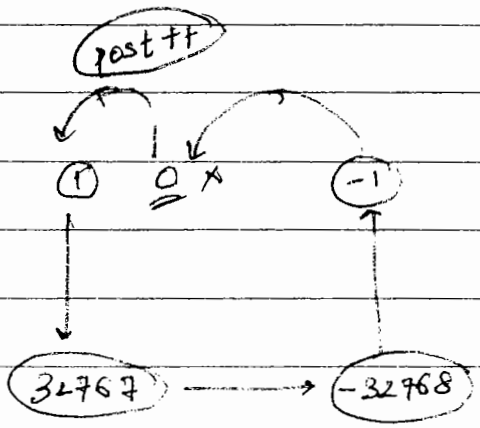
→ When the dummy loop is constructed then compiler creates empty body without any statements & until the loop condition became false, body is repeated n no. of times, if it is not false then it became infinite loop.

*
```
void main()
{
   int i;
   i=1;
   while(i++);
   printf("i=%d", i);
}
```

O/P: i=1 (65536)

i
1
2
3
4
5
8
32767 non-zero
-32768
-32767
-4
-3
-2
-1
0





32767
32768
+1
65536

→ void main()

```
{ int i;
  i = -1;
  while (i--> -32768);
  printf ("i=%d", i);
}
```

dummy loop

-1
-2

O/P: i = 32767

while (i--> -32768)
{
}

(-1)

-32768 > -32768

+32767

→ void main()

```
{ int i;
  i = -1;
  while (i--);
  printf ("i=%d", i);
}
```

post

O/P: i = -1 (65536)

→ Void main()

```
{ int i;
  i = 1;
  while (i++ < 32767);
  printf ("i = %d", i);
}
```

O/P: i = -32768

DELTA Pg No.
Date / /

break & Continue

1) break is a keyword, by using break keyword, we can terminate the loop body or switch body.
2) Using break is always optional but it should be required to place within the loop body or switch body only.
3) In implementation, where we knows the max. no. of repetitions but depends on the condition, if we require to stop the repetition process then go for break statement.
4) continue is a keyword, by using continue we can skip the statements from loop body.
Using continue is always optional, but it should be required to place within the loop body only.
5) In implementation, where we knows the max. no. of repetitions but depending on the condition if we require to skip the statements then go for continue.

→
```
void main ()
{
  Pnt i;
  i = 1;
  while (i<=10)
  {
   printf ("%d", i);
  if (i>3)
  break;
  ++i;
}
}
```

O/P: 1 2 3 4

→ In previous program, when if condition became true, break statement is executing, when the break statement is executed, then control is passed outside of the loop body.

→
```
void main()
{
 int i;
 i = 20;
 while (i>=2)
 {
  printf ("%d", i);
  i-=2;          // i = i-2
  if (i<=10)
  break;
 }
}
```

20 18 16 14 12

O/P : 20 18 16 14 12

→
```
void main()
{
  int i,
  i=1;
 while (i<=10)
  {
   printf ("%d", i);
  if (!i)
 break;
 i+=2;   // i = i+2;
 }
}
```

O/P:- 1 3 5 7 9

DELTA Pg No.
Date / /

→ There is no any mandatory conditions to execute break statement inside a loop body

→ Depending on cond^n ~~statement~~ states only break statement can be executed.

```
void main()
{
  int i;
  i=10;
  while (i>=2)
  {
    printf ("%d", i);
    if(i)
    break;
    i-=2;      //i=i-2;
  }
}
```

O/P: 10

→
```
void main()
{
  int i;
  i=5;
  while (i<=50)
{
  printf ("%d", i);
  if (i>=25);
    break;
  i+=5;
}
}
```

5

dummy condition

```
i=5;
while (i<=50)
{
  pf ("%d", i);
```

```
        if (i>=25)
        {
        }
        break;
        i+=5;
    }
```

O/P: 5

→ In previous prog. due to dummy condition, break statement is placing outside of the body that's why in order to execute loop body 1st time, automatically break statement is executed.

→
```
void main()
{
 int i;
 i=1;
 while (i<=10);        <- dummy loop
 {
   printf ("%d", i);
 if (i==5)
    break;
    ++i;
 }
}
```

O/P: Error misplaced break

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

→ Acc. to d syntax of the break, it should be required to place inside the loop body only, but in previous prog. due to dummy loop it is placing outside of the dummy loop.

DELTA | Pg No.
Date / /

→
```
void main()
{ int i;
  i=0;
  while (i<=40)
  {
    i+=2;    // i=i+2
    if (i>=10 && i<=30)
    continue;
    printf ("%d", i);
  }
}
```

O/P: 2 4 6 8 32 34 36 38 40 42

- When the continue statement is executing within the loop body then control will pass back to the cond", without executing remaining statements.

→
```
void main()
{
  int i;
  i=30;
  while (i>2)
  {
    i-=2;
  if (i>10 && i<20)
    continue;
    printf ("%d", i);
  }
}
```

O/P: 28 26 24 22 20 10 8 6 4 2

→
```
void main()
{
  int i;
  i=1;
  while (i<=25)
  {
    printf("%d", i);
    if (i>=5 && i<=15)
      continue;
    i+=2;
  }
}
```

O/P:- 1 3 5 5 5 5 — — inf. loop

- When we are working with continue statement, if increment statement or dec statement is skipping then it becomes inf. loop.

→
```
void main()
{
  int i;
  i=1;
  while (i<=30)
  {
    if (i>=9 && i<=25)
      continue;
    printf("%d", i);
    i+=2;  // i+2;
  }
}
```

O/P: No output with inf loop

DELTA Pg No.
Date / /

→
```
void main()
{
  int i;
  i=50;
while (i>=10);
{
  i-=2;  // i=i-2;
    if (i>=10 && i<=40)
continue;
  printf("%d", i);
}
}
```

dummy loop

O/P: Error misplaced continue

Acc. to syntax of continue, it should be required to place within the loop body only, but in previous prog due to dummy loop, it is placing outside the body.

Working with 'for' loop :-

When we are working with for loop it contains 3 parts i.e initialisation, condition, iteration.

→ void main()
{
int a;
a = 2
for(; a <= 10;)
{
printf("%d", a);
a += 2;
}
}

→ It will execute bcz in for loop everything is optional.
but mandatory to place 2 semicolons

O/P :- 2 4 6 8 10

→ void main()
{
int a, b;
for(a=1, b=10; a<=b; a++, b--)
printf("\n%d %d", a, b);
printf("\n a=%d b=%d", a+10, b+10);
}

O/P :-	1	10
	2	9
	3	8
	4	7
	5	6
	a=16	b=15

1 10
2 9
3 8
4 7
5 6
6<=5 Here condn - false
So 6+10 . 5+10

- When for loop is not having the body

In for loop, when we requied to place multiple initialisation & iteration, then recommended to use comma as a seperator.

→ void main()
{
int a, b;
for(a=b=5; a; printf("\n%d %d", a, b))
a = b-- >= 3;
printf("\n a=%d b=%d", a+10, b+10);
}

O/P :-	1	4
	1	3
	1	2
	0	1
	a=10	b=11

a: 5 4 3 2 1 0
b: 5>=3 ✓, 4>=3 ✓, 3>=3 ✓, 2>=3 ✗

→ Void main()

```
{
  int a, b;
  for (a=b=8; a; )
  {
    a = --b >= 5;
    printf("\n%d %d", a, b);
  }
  printf("\na = %d  b = %d", a+10, b+10)
}
```

a	b
~~8~~	~~8~~
1	~~7~~
1	~~6~~
1	~~5~~
0	~~4~~
a=10	b=14

O/P :-		
	1	7
	1	6
	1	5
	0	4
	a=10	b=14

* Perfect Number

Sum of all the factors of the no. should be = to input value then it is called Perfect Number.

Ex- 6 ↔ 1+2+3 → (6)

28 → 1+2+4+7+14 → (28)

496 → 1+2+4+8+16+31+62+124+248 = (496)

8128 → 1+2+4+8+16+32+64+127+254+508+1016+2032+4064 = (8128)

→ void main()

```
{
  int n, i, sum = 0
  clrscr();
  printf("Enter a value:");
  scanf("%d", &n);
for (i=1; i<=n/2; i++)
  {
    if (n%i==0)
      sum = sum + i;  // sum += i;
  }
if (sum==n && n>0)
  printf("\n%d IS PERFECT NUMBER", n);
else
  printf("\n%d IS NOT PERFECT NUMBER", n);
getch();
}
```

n	n/2
28	14

i	sum
~~1~~	0+1
~~2~~	1+2
~~3~~	3+4
~~4~~	7+7
~~5~~	14+14
~~6~~	(28) ✓
~~7~~	
~~8~~	
~~9~~	
~~10~~	
~~11~~	
~~12~~	
~~13~~	
~~14~~ (15)	

To get list of Perfect Numbers

```
void main()
{
 int n, n1, i, sum = 0;
 clrscr();
 printf("Enter a value : ");
 scanf("%d", &n1);
 for (n = 6; n <= n1; n += 2)
{ sum = 0;
  for (i = 1; i <= n/2; i++)
  { if (n % i == 0)
     sum = sum + i;  // sum += i;
  }
   if (sum = n)
     printf("\n %d. PERFECT NO : %d", ++count, n);
    if(count == 4)
     break;
  }
   getch();
}
```

O/P : Enter a value : 10000

1. PERFECT NO. : 6
2. PERFECT NO. : 28
3. PERFECT NO. : 496
4. PERECT NO. : 8128

Armstrong No.

Sum of every individual digit's cube value = no. is called armstrong Number.

There are 4 armstrong numbers

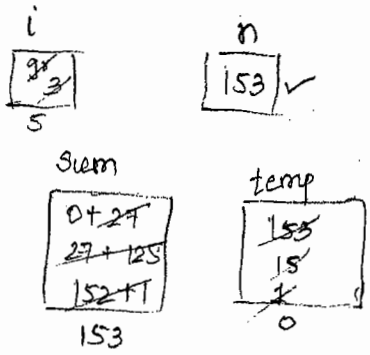
→ 153 → $1^3 + 5^3 + 3^3$ → $1 + 125 + 27$ } 153

→ 370 → $3^3 + 7^3$ → $27 + 343$ } 370

→ 371 → $3^3 + 7^3 + 1^3$ → $27 + 343 + 1$ } 371

→ 407 → $4^3 + 7^3$ → $64 + 343$ } 407

```
Void main()
{
 int n, temp, i;
 clrscr();
 printf("Enter a value:");
 scanf("%d", &n);
 for(temp=n; temp!=0; temp/=10)
 {
  i= temp%10;
  sum+=(i*i*i);    // sum+=pow(i,3);   <math.h>
 }
 if(n==sum && n>1)
  printf("\n%d IS ARMSTRONG NUMBER", n);
 else
  printf("\n%d is not ARMSTRONG NUMBER", n);
  getch();
}
```



O/P: Enter a value: 153
153 IS ARMSTRONG NUMBER

How to get list of numbers

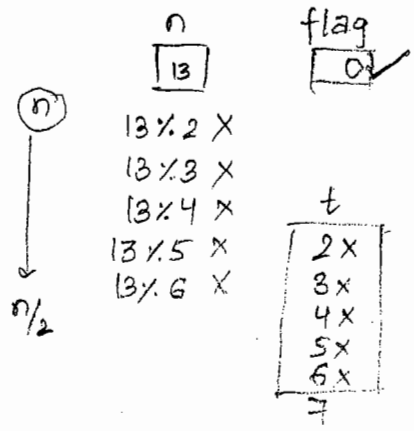
```
Void main()
{
 int n, n1, temp, i, sum=0;
 clrscr();
 printf("Enter a value: ");
 scanf("%d", &n1);
 for(n=153; n<=n1; n++)
 {
  sum=0;
  for(temp=n; temp!=0; temp/=10)
  {
   i= temp%10;
   sum+=(i*i*i);    //sum+=pow(i,3);  <math.h>
  }
  if(n==sum)
    printf("\n%d. ARMSTRONG NO.: %d", ++count, n);
  if(count==4)
    break;
  }
  getch();
}
```

O/P: Enter a value : 1500
1. ARMSTRONG NO. : 153
2. ARMSTRONG NO. : 370
3. ARMSTRONG NO. : 371
4. ARMSTRONG NO. : 407

* Prime No.

It is a no. which is not having any factors except 1 & itself.

```
void main()
{
 int n,t, flag=0;
 clrscr();
 printf("Enter a value : ");
 scanf("%d", &n);
 for(t=2; t<=n/2; t++)
 {
   if(n%t==0)
    {
      flag=1;
      break;
    }
 }
 if(flag==0)
    printf("\n%d IS PRIME NUMBER", n);
  else
    printf("\n%d IS NOT PRIME NUMBER", n);
  getch();
}
```



O/P: Enter a value : 13
13 IS PRIME NUMBER

```
Void main()
{
 long int n, n1, n2, t, count=0;
 int flag;
 clrscr();
 printf("Enter 2 values : ");
 scanf("%ld %ld", &n1, &n2);
 for(n=n1; n<=n2; n++)
 {
  flag=0;
  for(t=2; t<=n/2; t++)
  {
```

```
        if (n%b==0)
        {
          flag=1;
          break;
        }
    }
    if (flag==0)
       printf("\n%1d. PRIME NO: %1d", ++count, n);
  }
  getch();
}
```

O/P: Enter 2 values : 15 2170

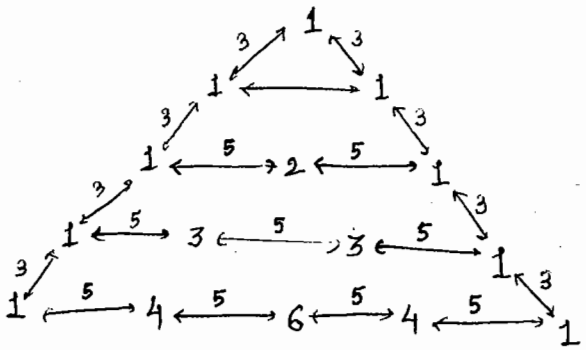
1. PRIME NO : 17
2. PRIME NO : 19
3. PRIME NO : 23

⋮

22/6

PASCAL TRIANGLE

Enter no. of rows = 5




```
void main()
  {
    int r, s, i, K, n;
     clrscr();
    printf("Enter no. of rows: ");
    scanf("%d", &r);
     i=0;
     while(i<r)
     {
        printf("\n")
        for(s=40-i*3; s>=1; s--)
           printf(" ");
           for(K=0; K<=i; K++)
             {    if(K==0)
                    n=1;
```

```
else
   n = ((i -K+1) * n) / K;
     printf ("%6d", n);
     }
        i= i+1;
     getch();
   }
}
```

```
i= 0;
while (i<r)
{
   printf ("\n");
   for (s= 40-i*3 ; s>=1 ; s--)
       printf (" ");
   for (K=0; K<=i; K++)
   {
      if (K==0)
          n=1;
      else
          n = ((i-K+1) * n)/K;
      printf ("%6d", n);
   }
   i= i+1;
}
```

r: 5

s: 41→1, 37→1, 34→1, 31→1, 28→1

i: 0, 1, 2, 3, 4, 5

K: 0, 1, 2, 3, 4

r: X, 2, 3, 4, 5

```
                    1
                  1   2   1
                1           1
              1   3       3   1
            1   4   6   4   1
```

INTERVIEW QUESTIONS

→ void main()
{
int a;
a = printf("Welcome");
printf("\n a = %d", a);
}

O/P :- Welcome
a = 7

It is possible to collect value from printf → integer values (17)

a = pf ("Welcome")
pf ("a = %d", a);

When we are working with printf function it returns an integer value i.e total no. of characters printed on console.

→ void main()
{
int a;
a = printf("%d Hello %d", 10, 200);
printf("\n a = %d", a);
}

O/P: 10 Hello 200
a = 12

(2(10) + 1(S) + 5(Hello) + 1(S) + 3(200))

→ void main()
{
int a;
a = printf("\n Welcome %d", printf("Naresh IT"));
printf("\n a = %d", a);
}

L ← R
10
9
R

O/P :- Naresh IT
Welcome 9
a = 10

(9(Naresh IT))
(10(1\n) + 7(Welcome) + 1(S) + 1(9))

→ When we are working with printf function always it executes from (R → L) only because it works with the help of stack.

→ When we are placing printf statement within the printf then from right side side 1st 'print' always executed first.

→ void main()

```
{
  int a;
  a = printf("One %d\n", printf("Two %d\n", printf("Three %d\n")));
    printf("a = %d", a);
}
```

O/P :-
Three
Two6
One5
a = 5

void main()

```
{
  int a;
  a = printf("One\n") + printf("Two\n") + printf("Three\n")
    printf("a = %d", a);
}
```

O/P :-
One 4
Two +4
Three +5+1
a = 14

→ void main()

```
{
  int a
  a = 5>2 ? printf("Hai") : printf("Bye");
  printf("\na = %d", a);
}
```

O/P : Hai
a = 3

a = 5>2 ? pf("Hai") : pf("Bye");
T

→ void main()

```
{
  int a;
  a = ! printf("Hi") ? printf("Bye") : printf("Hello");
    printf("\na = %d", a);
}
```

O/P : HiHello
a = 5

1) Hi ⇒ 2 value
2) Not operator

→ void main()
{
int a = 2;
a = printf("Hai") ? printf("NIT") : a = 20
pf("a = %d", a);
}

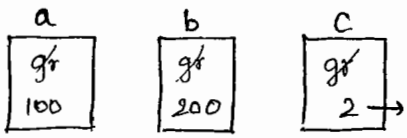
assignment operator i.e error

O/P: Error L value required

→ In conditional operator if right side expression contains assignment operator then it gives an error because syntax evaluation is right to left.

# void main()
{
int a, b, c;
c = scanf("%d %d", &a, &b);
pf("\na = %d", " b = %d", c = %d", a, b, c);
}
// input values 100 200

O/P:- a = 100 b = 200 c = 2



→ Total no. of values provided by user.

→ Scanf is predefined function, by using scanf function we can read the data from user.

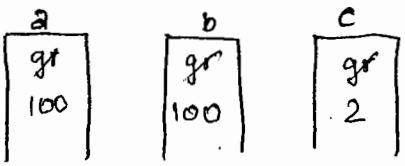
→ Scanf function returns an integer value ie total no. of input values provided by user

# void main()
{
int a, b, c;
a = b = 100;
c = scanf("%d %d", a, &b);
printf("\n a = %d b = %d c = %d", a, b, c);
}

Complete behaviour of scanf depends on format specifier not on argument.

// input values are 111 222

a = 700 b = 222 c = 2



→ Complete behaviour of scanf function will depends on format specifier only not on argument list.

→ As a programmer it is our responsibility to store the data proper by using ampercant "&" symbol.

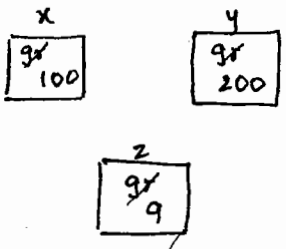
→ In scanf statement when we are not using "&" symbol then that variable value will not be updated with new value

```
void main()
{
  int a,b,c;
  a=10; b=20;
  c= scanf("%d %d %d");
  printf("\n a=%d  b=%d  c=%d", a,b,c);
}
```

O/P:- a=10 b=20 c=3

23/6

```
void main()
{
  int x,y,z;                                   L ← R
  z= printf("Welcome %d", scanf("%d %d", &x, &y));
     printf("\n x=%d  y=%d  z=%d", x,y,z);
}
// Input values are 100 200
```

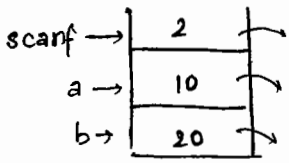


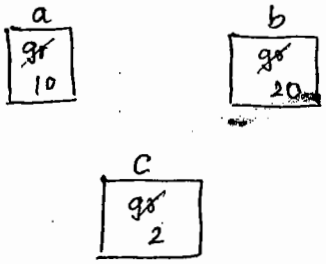
O/P:
```
Welcome 2
 x=100  y=200  z=9
```

```
void main()
{
  int a,b,c;
  a= 10; b= 20;
  c= printf("%d %d %d", scanf("%d %d %d", &a, &b);
     printf("\n a= %d  b=%d  c=%d", a,b,c);
} // input values are 111, 222
```

2 10 20

O/P:- a=111, b=222, c=7



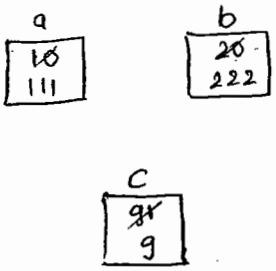


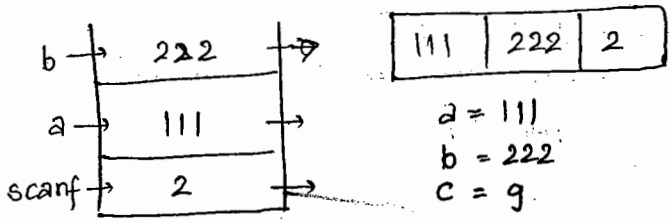
→ After passing the data into printf related stack then it is not possible to update by using scanf function.

→ Always scanf func^{n} will try to update actual memory locations only.

```
# void main()
  {
     int a, b, c ;
     a = 10 ; b = 20;
     c = printf ("%d %d %od", a,b, scanf("%d %d", &a, &b));
        printf ("\n a = %d  b = %d , a, b);
  } // Input values are 111 and 222.
```

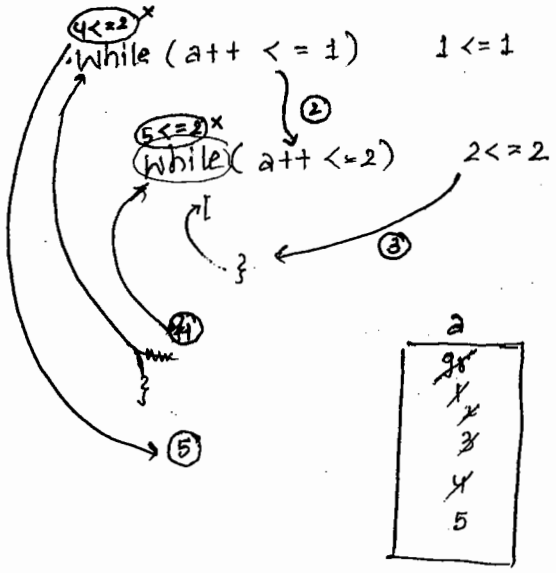
O/P :- a = 111, b = 222 c = 9





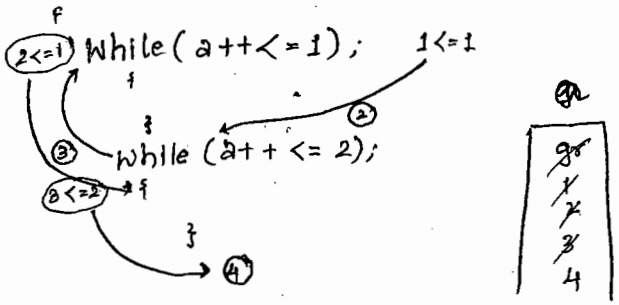

```
→ void main()
  {
     int a ;
     a = 1;
     while (a++ <= 1) ;
     while (a++<=2);
     printf ("a =%d", a);
  }
```

O/P :- a = 5



→ In Implementation when two whiles statements occur immediately without body, without statement and without semicolon then first while is called outer loop and second loop is called lower loop i.e according to nested loop body will be executed.

```
Void main()
{
  int a;
  a = 1;
  While (a++ <= 1);
  While (a++ <= 2);
  printf ("a=%d ", a);
}
```



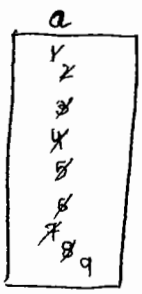
O/P: a = 4

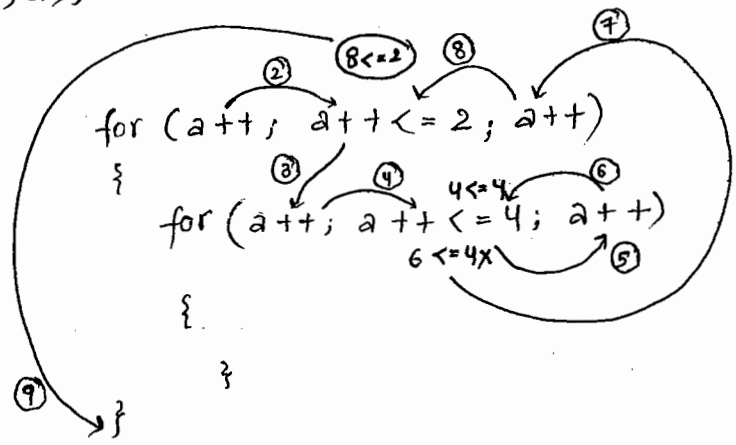
→ void main()

```
{
  int a;
  a = 1;
  for(a++; a++ <= 2; a++);
  for(a++; a++ <= 4; a++);
  printf("a = %d", a);
}
```

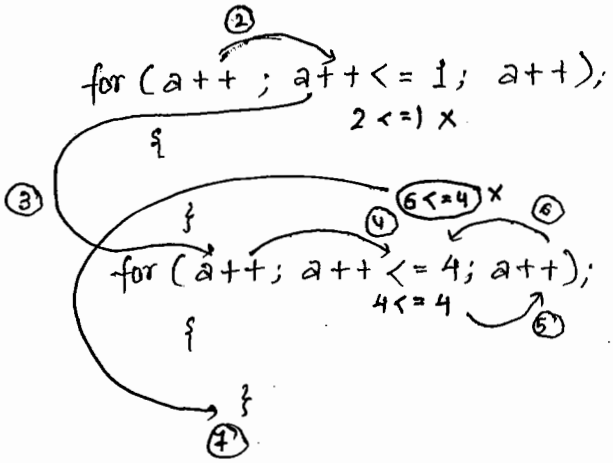
O/P: a = 9

⇒






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



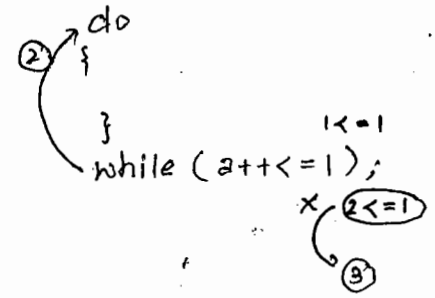
→ void main()
{
int a;
a = 1;
do
while (a++ <= 1);
printf ("a = %d", a);
}

```
do
while (a++<=1)
 {
 }
```

O/P :- Error - do statement must have while loop

→ When we require to create dummy - do while loop then recommended to place the semicolon ';' at the end of the do.

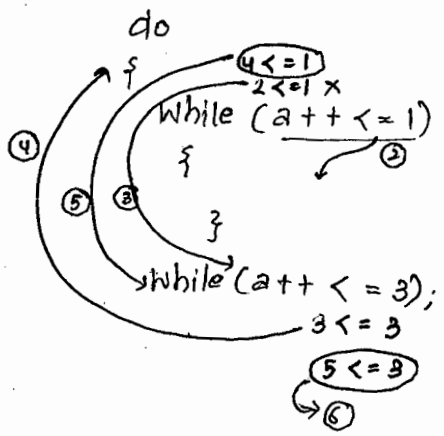
```
void main()
{
 int a;
 a = 1;
 do;
 while (a++ <= 1);
 printf ("a = %d", a);
}
```



O/P : a = 3

```
void main()
{
  int a;
  a = 1;
  do
  while (a++<=1);
  while (a++<=3);
  printf ("a = %d", a);
}
```

O/P :- a = 6



SWITCH (Switch);

→ Switch is a Keyword, by using this Keyword we can create selection statement with multiple choices.

→ Multiple choices can be constructed by using case keyword.

→ When we are working with switch statement it require a condition or expression of type and integer.

→ case keyword always required an integral constant expression or integral constant value.

Syntax :-

```
switch (condition / expression)
{
  case const 1: Block1;
                break;
  case const 2: Block 2;
                break;
  case const 3: Block 3;
                break;
  ---
  ---
  default: Block-n;
}
```

- When we are working with switch statement at the time of compilation condition or expression written value will map with case ~~sta~~ constant values.
- At the time of execution if matching case is available then control will pass to corresponding block, from that matching case upto Break everything will be executed.
- At the time of execution if matching case doesnot occur then control will pass to default block.
- default is a special kind of block which will be executed automatically when the matching case doesnot occur.
- By using nested if-else also, it is possible to create multiple blocks but we are required to create n no. of blocks den (n-1) conditions are mandatory to be created.
- When we are working with nested if-else at any point of time only one block can be executed but in switch statement, it is possible to execute multiple blocks by removing break statement b/w the blocks.
- Constructing the default is always optional, it is recommended to use when we are not handling all the blocks of switch statements.

→
```
void main() {
    int i;
    i = 2;
    switch (i)
    {
      case 1: printf("1");
      case 2: printf ("2");
      case 3: printf ("3");
      default: printf ("D");
    }
}
```

O/P :- 23D

→ void main()
```
{
  int i;
  i = 1;
  switch (i)
  {
    case 1 : printf ("A");
    Case 2 : printf ("B");
             break;
    case 3 : printf ("C");
             break;
    default : printf ("D");
  }
}
```

O/P :- A B

→ void main()
```
{
  int a;
  a = 5;
  switch (a)
  { case 1 : printf ("A");
             break;
    case 2 : printf ("2");
             break;
    case 3 : printf ("e");
             break;
    default : printf ("D");
  }
}
```

O/P :- D

→ void main
```
{
  int a;
  a = 3;
  switch (a)
  {
    case 1 : printf ("A");
             break;
    case 3 : printf ("B");
    case 2 : printf ("2");
             break;
```

O/P :- B2

```
        default : printf("D");
    }
}
```

- When we are working with switch statement, cases can be constructed randomly ie in any sequence cases can be designed.
- When we are constructing the cases randomly then from matching case upto break everything will executed in any sequence.

→
```
void main()
{
  float i;
  i=2;
  switch(i)
  {
    case 1 : printf("A");
             break;
    case 2 : printf("B");
             break;
    case 3 : printf("C");
             break;
    default : printf("D");
  }
}
```

O/P: Error (switch selection expression must be of integral type)

- When we are working with switch statement it is not applicable to float data type.

→
```
Void main()
{
  int f;
  f = 3.8;
  switch(f)
  {
    case 1 : printf("Apple");
             break;
    case 2 : printf("iphone");
             break;
    case 3 : printf("Apple TV");
             break;
    default : printf("ipad");
  }
}
```

O/P: Apple TV

→
```
void main()
{
  int a;
   a = 5;
   switch (a)
   {
    case 1 : printf ("A");
                 break;
    default : printf("D");
    case 2 : printf ("2");
                 break;
    case 3 : printf ("C");
  }
}
```

- When we are working with default, it can be placed anywhere within the switch body i.e beginning or middle or end of the body but generally recommended to place at the end of the body only.

24/Jun

→ void main()
{
int a,b,c,d;
a = 1; b = 2; c = 3;
d = c - a;
switch(d) → variable type
{
case a : printf ("A");
break;
case b : printf ("B");
break;
case c: printf ("C");
break;
default: printf ("D");
~~bre~~
}
}

case 'a';
case 'b':
case 'c':

O/P: Error Constant expression Required

- When we are working with case Keyword, it should be required, constant integral expression or constant integral value only i.e variable type data, we can't pass.

→ void main()
{
int a;
a = 8/4;
switch(a)
{
case 1+1: prinf ("A");
break; // Case 2;
case 4%2: printf ("B");
break; // Case 0;
case 2*3 : printf ("C");
break; // Case 6
default : printf ("D");
}
}

O/P: A

Within the switch statement when we are passing constant expression then it works acc. to written value.

→ void main()
```
{
  int a;
  a = 2<5;
  switch (a)
  {
    case 5%8 : printf("A");      // case 0:
               break;
    case 2<5 : printf("B");      // case 1:
               break;
    case 1!=2>5 : printf("C");   // case 1:
               break;
    case 2&&2 : printf("D");     // case 1:
  }
}
```

O/P : Error, Duplicate Case

- within the switch body cases must be unique i.e more than 1 case with same constant value not possible to create.

→ void main()
```
{
  int i;
  i = 2;
  switch (i)
  { case 1 : printf("A");
             break;
    case 2 : printf("B");
             continue;
    case 3 : printf("C");
             break;
    default: printf("D");
  }
}
```

O/P: Error misplaced continue

loop body [case, default] → switch body
loop body [break, continue]

→ void main()
```
{
  int a;
  a = 1;
  switch (a);   → dummy switch
}
```

```
    case 1 : printf ("A");
             break;
    case 2 : printf ("2");
             break;
    case 3 : printf ("C");
             break;
    default : printf ("D");
  }
}
```

O/P : Error misplaced case, default and break

→ When we are placing the semicolon at end of the switch, then it is called dummy switch.

→ When we are creating the dummy switch, automatically compiler creates new body without any statement and current body is outside of switch.

→ Acc. to syntax of case, default & break, it should be required to place within the body only.

goto

- goto is a keyword, by using this we can pass the control anywhere within the program.
- goto keyword always required an identifier called label.
- Any valid identifier followed by colon is called label.
- Generally goto statement is called unstructured programming statement bcoz it breaks the rule of structured prog. lang.
- In implementation, we required to take the repetition process, without using loop, then recommended to go for goto statement

Syntax:

```
    statement 1; ✓
    statement 2; ✓
goto LABEL;
    statement 3; X
LABEL:
    statement 4; ✓
    statement 5; ✓
```

→
```
void main()
{
  printf ("A");
  printf ("B");
goto ABC;
  printf ("Welcome");
ABC :
  printf ("C");
}
  printf ("D");
```

O/P : ABCD

→
```
void main()
{
  printf("A");
  printf("B");
ABC:
  printf("C");
  printf("D");
goto XYZ;
  printf("NIT");
XYZ:
  printf("X");
  printf("Y");
}
```

O/P: ABCDXY

- In order to execute the program, if the label is occured then automatically it will be executed.
- Creating the label is always optional, after creating the label, calling the label also optional. But if we are calling the label then it should be exist in the program

WAP to print all even nos from 2 to 20 without using loops.

```
void main()
{
  int i;
  i=2;
EVEN:
  printf("%d ", i);
  i+=2;
  if (i<=20)
goto EVEN;
}
```

O/P: 2 4 6 8 10 12 14 16 18 20

→
```
void main()
{
  printf("A");
goto XYZ;
  printf("NIT");
ABC:
  printf("B");
  printf("C");
XYZ:
  printf("X");
  printf("Y");
goto ABC;
}
```

O/P: AXYBCXYBCXYBC.... infinite loop

- When we are working with goto statement, if circle is created between the labels then it becomes infinite.

→
```
void main()
{
    printf ("A");
    printf ("B");
goto NIT;
    printf ("Welcome");
nit:
    printf ("C");
    printf ("D");
}
```

O/P: Error undefined label 'NIT'

- When we are working with goto statement, label works with the help of case sensitive, i.e upper & lower case both treated as different.

→ switch program

```
void main()
{
  int i;
  i = 2;
  switch (i)
  {
    case1 : printf ("A");
            break;
    case2 : printf ("B");
            break;
    case3 : printf ("C");
            break;
    default : printf ("D");
  }
}
```

label

O/P: D

- Acc. to syntax of the switch, case and constant value must required atleast single space.
- When the space is not given b/w case and constant value then it become label, dat's why default block is executed in prev. prog.

→
```
void main()
{
  int i;
  i = 2;
  switch (i)
{
  case 1 : printf ("A");
           break;
  case 2 : printf ("B");
           break;
```

```
        case 3 : printf ("C");
                 break;
        case default : printf ("D");
    }
}
```

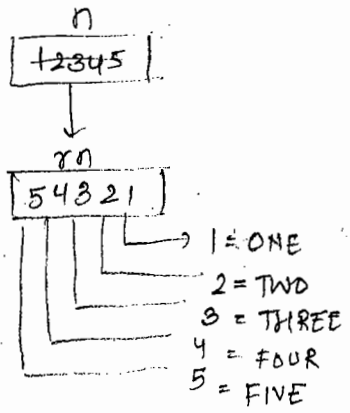
case ~~default~~ :
casedefault :
O/P : 2

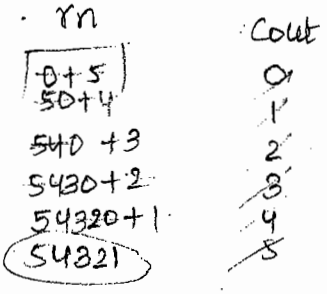
O/P : Error

Prog : Write prog.
Enter a value : 12345
ONE TWO THREE FOUR FIVE

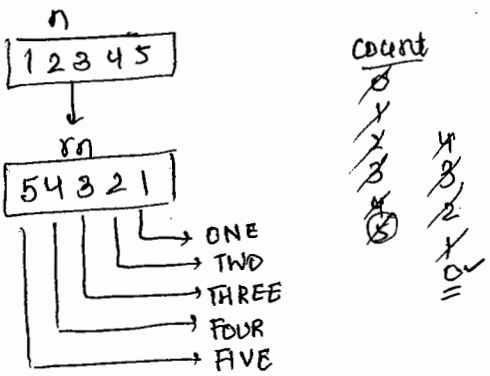
```
void main()
{
 long int n, rn;
 int count = 0;
 clscr();
 printf("Enter a value");
 scanf("%ld", &n);
 while(n)
{ rn = rn*10 + n%10;
  ++count;
  n = n/10;
}
while (rn)
 {  switch (rn % 10)
     {
        case 0: printf ("ZERO");
                 break;
        case 1 : printf ("ONE");
                 break;
        case 2 : printf ("TWO");
                  break;
        case 3 : printf ("THREE");
                  break;
        case 4 : printf ("FOUR");
                  break;
        Case 5 : printf ("FIVE");
                  break;

        Case 6 : printf ("SIX");
                  break;
        Case 7 : printf ("SEVEN");
                  break;
```

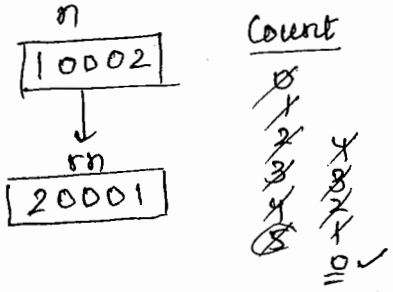
```
        case 8: printf ("EIGHT");
                break;
        case 9: printf ("NINE");
                break;
    }
    rn = rn/10;
    --count;
}
while (count > 0)
{ printf ("ZERO");
  --count;
}
getch();
}
```

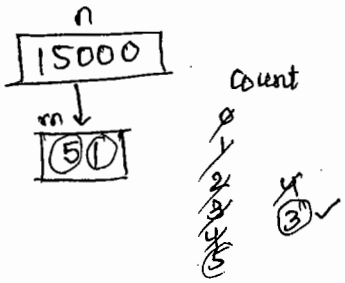


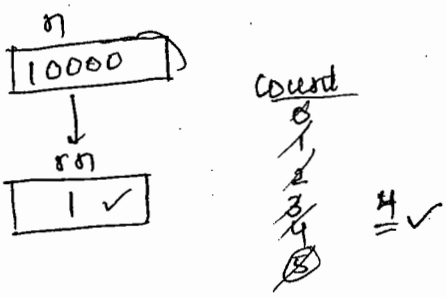


Other cases









25/Jun

Program - CALENDER

Enter year :
Enter month :
Enter day :
Weekday is :

```
Void main()
{
 int yy, mm, dd, nleap;
 long int dp;
 clrscr();
 do
   { printf("\n Enter year");
     scanf("%d", &yy);
   }
 while (yy<1);
 do
   { printf("Enter month : ");
     Scanf("%d", &mm);
   } while (mm<1 || mm > 12);
 do
  {
   printf("\n Enter Date : ");
   scanf("%d", &dd);
 } while (dd<1 || dd > 31);
 if ((mm==4 || mm== 6 || mm== 9 || mm== 11) && dd>30)
   {
     printf("\n invalid date");
     goto END;
   }
 if (yy%4 ==0 && yy % 100 != 0 || yy % 400 == 0)
   {
     if (mm= 2 && dd>29)
       { printf("\n invalid date");
         goto END;
       }
 }
  else
   {
     if (mm== 2 && dd >28)
       { printf("\n invalid date");
```

Here we want to repeat invalid case, so write invalid condition.

Here if we write (mm <1 && mm > 12);
Here condition not fully satisfy
Take -15
-15<1 but -15>12
True False

```
        goto END;
    }
}
nleap = (yy-1)/4 - (yy-1)/100 + (yy-1)/400;
```

every 4th year ⇒ every century year ⇒ every 400th year

```
dp = (yy-1) * 365L + nleap;
```

long for storing long value of int

```
switch (mm)
{
  case 12 : dp+ = 30;     // Nov. (Add)  (dp contains the value of dec
  case 11 : dp+ = 31;                      if it is of dec month)
  case 10 : dp+ = 30;
  case 9  : dp+ = 31;     // Reverse because
  case 8  : dp+ = 31;                say if d month is june
  case 7  : dp+ = 30;
  case 6  : dp+ = 31;     then 6+5+4+3+2+1   and since case
  case 5  : dp+ = 30;                ||          nature is ↓ ∴
  case 4  : dp+ = 31;                ↓           in reverse → in order
  case 3  : dp+= 28;      d date                 to add all d values
  case2 : dp+ =31;
  case1 : dp+=dd;
}
```

If cases like 1 2 3 4 5

Here if date is 25-Jun-05
Control goes to june
then it adds july, aug, sept --
which is wrong

⇒ if leap year

```
if ((yy%4==0 && yy%100!=0 || yy%400==0) && mm>2)
    ++dp;
printf ("\n%d/%d/%d  Weekday is : ", dd, mm, yy);
switch (dp%7)
{
  case 1 : printf ("Monday");
           break;
  case 2 : printf ("Tuesday");
           break;
  case 3 : printf ("Wednesday");
           break;
  case 4 : printf ("Thursday");
           break;
  case 5 : printf ("Friday");
           break;
  case 6 : printf ("Saturday");
           break;
  case 0 : printf ("Sunday");
}
END :
getch();
}
```

To find Weekday

* 01-01-01 ⟶ Monday

18 Jan-01 → 4 Thursday
18 % 7 = 4

14-Feb-01 → 3 Wednesday
31
+14
45
45 % 7 = 3

20-Mar-01 ⇒ 2 Tuesday
31
28
20
79 % 7

final now
yy
2015
J
mm
06
dd
current 25
2015

passed years (yy-1) * 365 + n leap
2014 * 365 + 488 = 31-Dec 2014

735598

2015
31
28
31
30
31
25

dp → 7,35,774

Jan, Mar, May, July, Aug, Oct, Dec } 31
April, June, Sept, Nov } 30
Feb 28/29

For leap year

1) % 4
2) after 100 yrs → X no leap yr
3) after 400 yrs → ✓ leap yr

100	1100
200	(1200)
300	1300
(400)	1400
500	1500
600	(1600)
700	1700
(800)	1800
900	1900
1000	(2000)

20 x 25 = (500)
3
503
- 20
483
+ 5
(488)

Program :- To get the weekday of 29th date of feb of the input year

```
void main()
{
   int yy, nleap;
   long int dp;
   clrscr();
   do
  {
   printf ("\n Enter year : ");
    scanf ("%d", &yy);
    } while (yy<1);
  if (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0)
 {
   nleap = (yy-1)/4 - (yy-1)/100 + (yy-1)/400;
     dp = (yy-1) * 365L + nleap;
      dp+ = 31;    // Jan
      dp+ = 29;    // Feb
      printf ("\n 29-Feb-%d Weekday is : ", yy);
      switch (dp % 7)
    {
      case 1: printf ("Monday");
              break;
      case 2: printf ("Tuesday");
              break;
      case 3: printf ("Wednesday");
              break;
      case 4: printf ("Thursday");
              break;
      case 5: printf ("friday");
              break;
      case 6: printf ("saturday");
              break;
       case 0 : printf("Sunday");

    }
 }
else
     {
       printf ("%d IS NOT LEAP YEAR");
      getch();
 }
```

DATA TYPES :-

Always data types will decide that what type of data is required to store in variable.

⇒ In 'c' programming lang. We have 3 types of basic data types i.e. :-

a) char
b) int and
c) float types

→ When the basic data type is not supporting user requirement then go-for primitive or predefined data types.

→ All primitive data types are constructed by extending size and range of basic data type.

→ In 'C' programming lang. we are having 9 types of primitive or pre-defined data types

Type	Size	Range	%	Ex :-
char	1 byte	-128 to 127	%c	'a' 'A' '#' '\n'
unsigned char	1 byte	0 to 255	~~%d~~	
int	2 bytes	-32768 to 32767	%d	-25 5 0 -5
unsigned int	2 bytes	0 to 65,535	%u	25u 32768u
long int	4 bytes	-2,147,483,648 to +2147,483,648	%ld	45L -5L 40000L
unsigned long	4 bytes	0 to 4,294,967,295	%lu	100Lu 51Lu
float	4 bytes	$\pm 3.4 * 10^{\pm 38}$	%f	-35f 7.5f
double	8 bytes	$\pm 1.7 * 10^{\pm 308}$	%lf	-3.5, 7.5
long double	10 bytes	$\pm 3.4 * 10^{\pm 4932}$	%Lf	-3.5L, 7.5L

→ The basic advantage of classifying this many types nothing but utilizing m/m more efficiently & inc. the performance

- In implementation, when we required to manipulate characters then it is recommended to go for char or unsigned char datatype.
- When we required the numeric operations from the range of -128 to +127 then go for char datatype in place of constructing an integer, in this case recommended to use %d format specifier.

→ When we required the numeric values from the range of 0 to 255 then go for unsigned char datatype in place of constructing an unsigned integer. In this case it is recommended to use %u or %d format specifier

→ For normal numeric operations go for an int type, if there is no -ve representation then go for unsigned int datatype like employee salary

→ By default, any type of decimal values are integer and real values are decimal.

→ short, long, signed & unsigned are called Qualifiers.

→ These all Qualifiers required to applied for an integral datatypes only i.e we can't apply for float, double & long double type.

→ Integral datatype means char, unsigned char, int, unsigned int, long int and unsigned long data types.

→ signed, unsigned are called signed specifiers

→ short, long are called size specifiers

```
void main()
{
  int i;
  long int l;
  float f;
  i = 32767+1;
  l = 32767+1;
  f = 32767+1;
  printf("%d %ld %f", i, l, f);
}
```

O/P :- -32768 -32768 -32768.000000

Automatic Operations on Data Type.

→ If both arguments are same type then return value is same type

→ If both arguments are diff. type then among those 2, which one will occupies max. value, will return.

int, int ----------> int

int, float ----------> float

int, char ----------> int

(int), long int ----------> long

same size: (short), long ----------> long

same size: (signed), unsigned ----------> unsigned

float, long ----------> float

float, double ----------> double

double, long double ----------> long double

→ When we are working with an integral value with the combination of float then always return value is float type only.

```
void main()
{
  int i;
  float f;
  long int l;
  i = 32767 + 1;
  l = 32767L + 1;   // L = (long) 32767 + 1;
  f = 32767.0f + 1; // f = (float) 32767 + 1;
  printf("%d %ld %f", i, l, f);
}
```

O/P:- -32768 32768 32768.000000

Type Casting

→ It is a procedure of converting one datatype values into another data type.

→ Type Casting can be performed by using (type) operator
(type) means name of the datatype.

→ In 'C' prog. lang. Type Casting is classified into 2 types.

1) Implicit type casting
2) Explicit type Casting

1) Implicit Type Casting - When we are converting low order data into high order datatype.

→ Implicit Type casting is under control of compiler.

→ As a programmer, it is not required to handle implicit type casting process.

Ex-
```
int i;
long int l;
i = 32414;
l = i;      L = (long) i;
```

2) Explicity Type Casting - When we are converting high order data type into low order datatype then it is called explicit Type Casting.

→ Explicit Type Casting is under control of programmer

→ As a programmer, mandatory to handle explicit type casting process or else data overflow is occur

Ex-
```
long double d;
long int L;
d = 123456789.987654321;
L = (long) d;
```

→ void main()
```
{
  float f;
  f = 2.9;   → by default it is double
  if (f == 2.9)      // if (float == double)  4B == 8B
    printf("Welcome");
  else
    printf ("Hello");          O/P :- Hello
}
```

→ When we are comparing float value with double then comparision will take place on binary representation, so 8 bytes data ≠ 4 bytes.

→ void main()
```
{
  float f;
  f = 3.7;
  if (f == 3.7f)   // if (f == (float) 3.7)
    printf ("Welcome");
  else
    printf ("Hello");          O/P :- Welcome
}
```

→ void main()
```
{
  int i;
  float f;
  i = 7;
  f = 7.0;
  if (i == f)
    printf ("Welcome");
  else
    printf ("Hello");          O/P :- Welcome
}
```

When we are working with float values or float variables, if the fractional part is zero, then it occupies the m/m in the form of integer that's why float data is equivalent to integer data type.

→ void main()
```
{
  float f;
  f = 3.5;
  if (f == 3.5)
    printf ("Welcome");
  else
    printf ("Hello");
}
```

O/P :- Welcome

When .1 .6 .2 .7 .3 .8 .4 .9 } float ≠ double

.0 .5 } float == double

```
void main()
{
  int i;
  unsigned int u;
  i = -1;
  u = 10;
  if (i > u)
    printf("Welcome");
  else
    printf("Hello");
}
```

O/P :- Welcome

i → signed value
u → unsigned value

if (i > u)
signed > unsigned

-1 (signed)

65535 (unsigned) > 10

NUMBER SYSTEM

Always number system will decide that how the numeric values required to store in m/m.

Number systems are classified into 4 types.

1) Decimal Number System
2) Octal Number System
3) Hexadecimal Number System
4) Binary Number System

1) Decimal Number System -

- The base value of decimal number system is 10 and the range from 0 to 9.
- By using decimal no. system, we can represent +ve & -ve values also.
- When we are working with decimal no. system, we required to use %d format specifier.

Eg :- 10, -20, 35, -40

2) Octal Number System

- The base value of octal no. system is 8 and the range from 0 to 7.
- If any numeric value is started with zero then it indicates octal data
- By using octal no. system, we can represent +ve data only.
- When we are working with octal no. system, we required to use %o format specifier

Eg :- 012, 034, 0765, 01457

3) Hexadecimal Number System

- The base value of Hexadecimal is 16 and the range from 0 to 9 A B C D E F.
- If any numeric value is started with 0x then it indicates hexadecimal value.
- By using hexadecimal no. system, we can't represent -ve data.
- When we are working with hexadecimal no. system, then we are required to use %x or %h format specifier

Eg :- 0X1234, 0XA1B2, 0XFEAB

	DECIMAL (10) 0-9 %d	OCTAL (8) 0-7 %0	HEXADECIMAL (16) 0-9ABCDEF %x
①	100	0144	0X64
②	127	0177	0X7F
③	32767	077777	0X7FFF
④	65535	0177777	0XFFFF

8 | 100
8 | 12 - 4
1 - 4

8 | 127
8 | 15 - 7
1 - 7

16 | 100
6 - 4

16 | 127
7 - 15 (F)

	DECIMAL (10) 0-9 %d	OCTAL (8) 0-7 %0	HEXADECIMAL (16) 0-9ABCDEF %x
⑤	10	012 $8^1 \times 1 + 8^0 \times 2$	0XA
⑥	83	0123 $8^2 \times 1 + 8^1 \times 2 + 8^0 \times 3$	0X53
⑦	Illegal ←	0128 → Max 0-7 0192	Illegal

161	0241	0XA1 $16^1 \times 10 + 16^0 \times 1$
291	0443	0X123 $16^2 \times 1 + 16^1 \times 2 + 16^0 \times 3$ 256 + 32 + 3
32768	0100000	0X8000

BINARY NUMBER SYSTEM

→ The base value of binary no system is 2 and the range is 0, 1.

→ When we are representing the data in m/m, then always it stores in the form of binary only.

→ As a programmer we cant pass the instruction and can't store the data directly in binary format.

→ There is no any special kind of format specifiers are available to print the data on console but logically it is possible using arrays.

int i = 0101 → dis is not binary to octal
↳ starting with 0

Decimal	Binary	
49	0011 0001	(32+16+1)
127	0111 1111	(64+32+16+8+4+2+1)
255	1111 1111	128 + 127 (2^8)
32768	1000 0000 0000 0000	
65535	1111 1111 1111 1111	

No. of bits	No. of Combination	max value	max value representation
1	2 (2^1)	1 (2^1-1)	0001
2	4 (2^2)	3 (2^2-1)	11
3	8 (2^3)	7 (2^3-1)	111
4	16 (2^4)	15 (2^4-1)	1111
5	32 (2^5)	31 (2^5-1)	0011 1111
6	64 (2^6)	63 (2^6-1)	111 1111
7	128 (2^7)	127 (2^7-1)	1111 1111
8	256 (2^8)	255/-1 (2^8-1)	1111 1111
16	65536 (2^{16})	65535/-1 ($2^{16}-1$)	1111 1111 1111 1111

int i ; On dos based compiler, size of integer is 2 bytes & the range from -32768 to 32767

→ For representing any integer value, we required 16 bit combination i.e 65536 representations.

→ Among those all representations, 50% provides +ve values & remaining 50% provides -ve value .

→ left side → most significant bit is called Sign bit

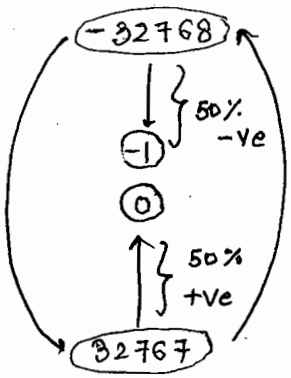
→ Always sign bit will decides the written value of binary representation i.e +ve or -ve.

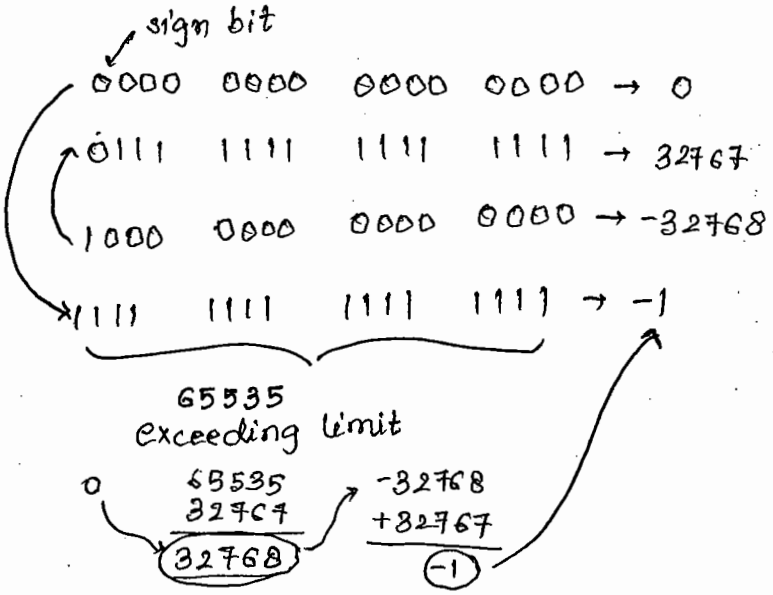
→ If sign bit is 0 and remaining all bits are zero then it gives min. value of +ve sign i.e zero

→ If sign bit is 0 and remaining all bits are ones then it gives max. value of +ve sign i.e 32767

→ If sign bit is 1 and remaining all bits are zeros then it gives min. value of -ve sign i.e -32768

→ If sign bit is 1 and remaining all bits are ones then it gives max. value of -ve sign i.e -1





32767 → 0111 1111 1111 1111
\+ 1 → 0000 0000 0000 0001

1000 0000 0000 0000 ⇒ -32768

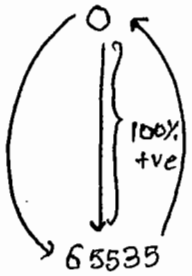
unsigned int u ;

→ The size of unsigned integer is 2 bytes and the range from 0 to 65535

→ For representing any unsigned integer value, we required 16 bit combination i.e 65536 representations.

→ Among those all representations, all provides +ve data only coz sign bit is not available in unsigned type.

→ When we are working with integer and unsigned integer, den always format specifier decides the return value of binary representation.



%d	Binary	%u
0	0000 0000 0000 0000	0
32767	0111 1111 1111 1111	32767
-32768	1000 000 000 000	32768
-1	1111 1111 1111 1111	65536

char ch ;

→ The size of character is 1 byte and the range from -128 to 127

→ For representing any character variable, we require 8 bit combinations i.e 256 representations.

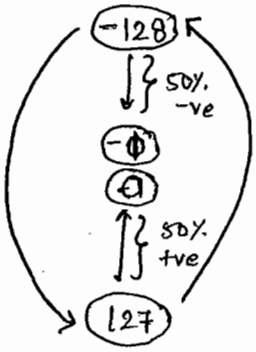
→ Among those all representations 50% provides +ve data & remaining 50% provides +ve data

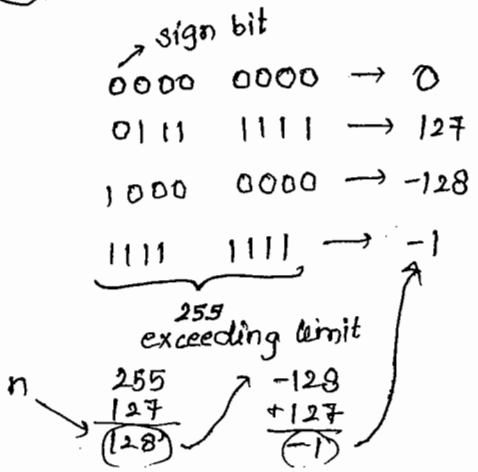
→ If sign bit is 0, and remaining all bits are zeros then it gives min. value of +ve sign i.e (0)

→ If sign bit is 0, and remaining all bits are ones then it gives max. value of +ve sign i.e (127)

→ If sign bit is 1, and remaining all bits are zeroes, then it gives min. value of -ve sign i.e (-128)

→ If sign bit is 1, and remaining all bits are ones, then it gives max. value of -ve sign i.e (-1)





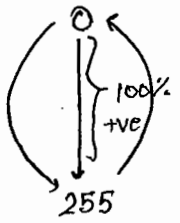
unsigned char ch ;

→ The size of unsigned char is 1 byte and range from 0 to 255

→ For representing any unsigned char, we required 8 bit combinations i.e 256 representations.

→ Among those all representations, all provides +ve data only.

→ When we are working with char & unsigned char, always datatype will decides the written value of binary representation bcoz both are having same format specifier.



0000 0000 → 0
0111 1111 → 127
1000 0000 → 128
1111 1111 → 255

1. printf ("%d %u", 65535u, -1);
O/P :- -1 65535

2. printf ("%d %u", 32768u, -32768); O/P :- -32768, 32768

3. printf ("%d %o %x", 65535u, 65535u, 65535u);
O/P :- -1 177777 FFFF

First count in binary forms and then solve

65535 → 1111 1111 1111 1111 (Octal)
1 7 7 7 7 7

65535 → 1111 1111 1111 1111 (Hexadecimal)
F F F F

For representing any octal data we need 3 binary bits only, for representing any hexadecimal value we require 4 binary bits.

4. printf ("%d %u %x", 076543, 076543, 076543);
↳ octal

076543
~~0111~~ ~~0110~~ ~~0101~~ ~~0100~~
111 110 101 100 011 octal
0111 1101 0110 0011 Hexa
7 D 6 3
2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^{9} 2^{8} 2^{7} 2^{6} 2^{5} 2^{4} 2^{3} 2^{2} 2^{1} 2^{0}

%x → 7D63
%d → 32099
%u → 32099

16384 + 8192 + 4096 + 2048 + 1024 + 256 + 64 + 32 + 2 + 1 = 32099

5. printf ("%d %u %o", 0XFABC, 0XFABC, 0XFABC);

0XFABC

F → 1111
A → 1010
B → 1011
C → 1100

} 16 bit

1 7 5 2 7 4

1111 1010 1011 1100

32768 + 16384 + 8192 + 4096 + 2048 + 512 + 128 + 32 + 16 + 8 + 4 = 64188

64188 − 32768 = 31420

32767 − 31421 = 1848

%u = 61488
%d = −1848
%o = 175274

BITWISE OPERATORS

1) In implementation when we required to manipulate the data on binary representation then go for bitwise operators.
2) Bitwise operators need to be applied for an integral datatype only i.e we can't applied for float, double and long double type.
3) Bit manipulation is always low level programming concept.
4) When we are working with bitwise operators directly manipulation will happen on m/m only.
5) In 'C' programming lang., we are having following operators.

~ ⟶ 1's complement
<< ⟶ bitwise left shift
>> ⟶ bitwise right shift
& ⟶ bitwise AND
^ ⟶ bitwise XOR
| ⟶ bitwise OR

(1) 1's complement -
This operator returns complement value of input data.
Complement means all 1's will be converted to zeros, all 0's → 1's

→ int a;

a) a = ~5

5 → 0000 0000 0000 0101
~5 → 1111 1111 1111 1010 (2² 2⁰) = −6 (−1−4−1)

65535 → 1111 1111 1111 1111
~5 → 1111 1111 1111 1010

65535 − 5 = 65530

65530 − 32767 = 32763 (n−1)

−32768 − 32762 = −6

(n)

b) a = ~239

Sign bit approach

239 → 0000 0000 1110 1111

~239 → 1111 1111 0001 0000 ($2^7\ 2^6\ 2^5$ / $2^3\ 2^2\ 2^1\ 2^0$)

sign bit

$(-1 - 1 - 2 - 4 - 8 - 32 - 64 - 128 = \boxed{-240})$ ($2^0\ 2^1\ 2^2\ 2^3\ 2^5\ 2^6\ 2^7$)

Calculation approach

65535 → 1111 1111 1111 1111

~239 → 1111 1111 0001 0000

65535 − 239 = 65296

65296 → 32767 (n−1) → 32529

−32768 → 32528 → −240

(c) a = ~0; −1

0 ---→ 0000 0000 0000 0000

~0 ---→ 1111 1111 1111 1111 (−1)

d) a = ~32767; −32768

32768 ---→ 0111 1111 1111 1111

~32767 ---→ 1000 0000 0000 0000 −32768

e) a = ~−19; 18

−1 → 1111 1111 1111 1111

−19 → 1111 1111 1110 1101 } From −1 to −19 we need to reduce 18 values

~−19 → 0000 0000 0001 0010 (18)

f) a = ~−128, 127

−1 → 1111 1111 1111 1111

−128 → 1111 1111 1000 0000 (−1 −64 −32 −16 −8 −4 −2 −1)

~−128 → 0000 0000 0111 1111 (127)

---

unsigned int u;

1. u = ~15 65520

15 → 0000 0000 0000 1111

~15 → 1111 1111 1111 0000 65520 (65535 − 15)

[1's complement value ≠ negation of given of given input value
means ~5 ≠ −5
2's complement value is always = negation of given input value
2's complement = 1's complement + 1]

Que What is the 2's complement value of 29?

Ans -29 (exactly negation value)

```
29 ----------> 0000 0000 0001 1101
~29 ---------> 1111 1111 1110 0010
+ 1 ---------> 0000 0000 0000 0001
               1111 1111 1110 0011   (-29)(-1-16-8-4)
```

LEFT-SHIFT OPERATORS

1) In left-shift operators, towards from left side n no. of bits required to draw & towards from right side (empty) places required to be filled with (zeros).

2) When we are dropping the bits towards from left side den all 1's will be shifted towards left side by n places, so automatically value is increased.

int a;

1. a = 10 << 1;

```
10 ---> 0000 0000 0000 1010
10<<1 -> 0000 0000 0001 0100   20(16+4)
```

2. a = 5<<2;

```
5 ---> 0000 0000 0000 0101
5<<2 -> 0000 0000 0001 0100   20(16+4)
```

3. a = 20<<3;

```
20 ---> 0000 0000 0001 0100
20<<3 -> 0000 0000 1010 0000   160(128+32)
```

4. a = 15<<4;

```
15 ---> 0000 0000 0000 1111
15<<4 -> 0000 0000 1111 0000   240 (128+64+32+16)
```

5. a = 1<<1 2

6. a = 100<<1; 200

7. a = 500<<1; 1000

8. a = 1500<<1; 3000

9. a = 1<<15; -32768

```
1 ---> 0000 0000 0000 0001
1<<15 -> 1000 0000 0000 0000   (-32768)
```

10. a = 32767 << 15 −32768

32767 → 0111 1111 1111 1111

32767<<15 →

11. a = 1 << 16; 0

12. a = 1234 << 16; 0

13. a = 32767 << 16; 0

* For any +ve integer, left shift (16) makes the value as zero bcoz +ve data representation is available in 16 bit towards from left side.

int a;

1. a = -10 << 1; −1 → 1111 1111 1111 1111
~9 << 1; −10 → 1111 1111 1111 0110

2. a = -10<<2; −10<<1 → 1111 1111 1110 1100 −20 (-1-16-2-1)
~9 <<2; −10<<2 → 1111 1111 1101 1000 −40 (-1-32-4-2-1)

3. a = -10<< 3; −10<<3 → 1111 1111 1011 0000 −80 (-1-64-8-4-2-1)
~9 <<3;

Value = x<<n; → valid upto n = 14.
x * 2^n

RIGHT-SHIFT OPERATOR

1) In right-shift operator, towards from right side n no. of bits required to draw & towards from left side empty places required to be filled with zeros for +ve no.s only.

2) In right-shift operator, for -ve no., empty places required to be filled with 1's so sign bit will not be modified & we will get -ve values only.

3) When we are dropping the bits towards from right side then all 1's will be shifted towards right side by n places, so automatically value is decreased. (for +ve only).

int a;

1. a = 10 >> 1; 10 → 0000 0000 0000 1010
10>>1 → 0000 0000 0000 0101 (5)

2. a = 15 >> 2; 15 → 0000 0000 0000 1111
15>>2 → 0000 0000 0000 0011 (3)

3. a = 20>>3; 20 → 0000 0000 0001 0100

4. a = 25>>4; 20>>3 → 0000 0000 0000 0010 (2)

25 → 0000 0000 0001 1001

25 >> 4 → 0000 0000 0000 0001 ①

5. a = 100 >> 1; 50

6. a = 1000 >> 1; 500

7. a = 1234 >> 1; 617

8. a = 1 >> 15; 0

9. a = 32767 >> 15; 0

⇒ For any positive integer, right-shift 15 makes the value as 0 because +ve data representation is available in 15 bits towards from right side.

int a;

Value = x >> n;
= $x/2^n$

1. a = -5 >> 1;

-4 >> 1;

-1 → 1111 1111 1111 1111

-5 → 1111 1111 1111 1011

-5 >> 1 → 1111 1111 1111 1101 · -3 (-1-2)

-5 >> 2 → 1111 1111 1111 1110 -2 (-1-1)

-5 >> 3 → 1111 1111 1111 1111 -1

Remaing Operators (BITWISE AND, OR, XOR)

a	b	a & b	a | b	a ∧ b
1	1	1	1	0
0	1	0	1	1
1	0	0	1	1
0	0	0	0	0

1. printf ("%d %d %d", 2&3, 2|3, 2∧3);

O/P : 2 3 1

2 → 0000 0000 0000 0010

3 → 0000 0000 0000 0011

2 & 3 → 0000 0000 0000 0010

2 | 3 → 0000 0000 0000 0011

2 ∧ 3 → 0000 0000 0000 0001

2. printf ("%d %d %d", 25&30, 25|30, 25^30);

```
25 → 0000 0000 0001 1001
30 → 0000 0000 0001 1110
-------------------------
25&30 → 0000 0000 0001 1000   (24)
25|30 → 0000 0000 0001 1111   (31)
25^30 → 0000 0000 0000 0111   (7)
```

3. printf ("%d %d %d", 0&-1, 0|-1, 0^-1)

```
0  → 0000 0000 0000 0000
-1 → 1111 1111 1111 1111
-------------------------
0&-1 → 0000 0000 0000 0000   (0)
0|-1 → 1111 1111 1111 1111   (-1)
0^-1 → 1111 1111 1111 1111   (-1)
```

OPERATORS IN 'C'

In 'c' prog. lang., We are having 44 operators, acc. to the priority those operators are- (45 operators but , is not opr, it is seperator so 44)

1. ( ), [ ] (subscript), → (Pointer to member), . (struct to member) — 1st category operators
   (Paranthesis)

2. +, -, ++, --, !, ~, *, &, size of, (type) → 2nd category
   (* ↓ 1st direction; & address of)

3. *, /, % — 3rd category

4. +, - — 4th category

5. << (L. shift), >> (R. shift) — 5th category

6. <, >, <=, >= — 6th category

7. !=, == — 7th category

8. & (Bitwise AND) — 8th category

9. ^ (B. XOR)

10. | (B. OR)

11. &&

12. ||

13. ?:

14. =, +=, -=, *=, /=, %=
    &=, |=, ^=, <<=, >>=

15. ,

Sizeof

→ sizeof is a operator cum Keyword in 'C' lang.

→ By using sizeof operator, we can find size of an I/P argument.

→ Sizeof operator is a unary operator which always requires 1 argument of type, of datatype & returns unsigned integer value which always greater than zero.

```
void main()
{ int i;
  char ch;
  float f;
  clrscr();
  printf ("\nsize of int : %u", sizeof(i));
  printf ("\nsize of char : %u", sizeof ch);
  printf ("\nsize of float : %u", size of f);
```

→ with paranthesis
→ without paranthesis
} Both way, u can call.

O/P :- size of int : 2
size of char : 1
size of float : 4

sizeof (int) ⟶ 2
sizeof (short) ⟶ 2
size of (signed) ⟶ 2
sizeof (unsigned) ⟶ 2
sizeof (long) ⟶ 4
sizeof (25) ⟶ 2
size of (25.0) ⟶ 2
size_of (25l) ⟶ 4

---

sizeof (char) ⟶ 1
sizeof ('A') ⟶ 2
char ch;
ch = 'A';
sizeof(ch) ⟶ 1

- sizeof character or character variable is 1 byte but sizeof character constant is 2 bytes

- By default character constant returns an integer value i.e Ascii value of a character constant that's why size of (Char constant) is (2 bytes).

sizeof (float) ⟶ 4
sizeof (double) ⟶ 8
sizeof (long double) ⟶ 10B
sizeof (short float) ⟶ Error
sizeof (signed double) ⟶ Error
sizeof (12.8) ⟶ 8 // double
sizeof (12.8f) ⟶ 4 // float
sizeof (12.8L) ⟶ 10 // long double
sizeof (12.0) ⟶ 8
sizeof (12.5) ⟶ 8

```
void main()
{ int i ;
  i=10;
  printf ("\n size 1 : %u ", sizeof (i));
  printf ("\n size 2: %u", size of (i/5.0));
  printf ("\n size3: %u", sizeof (i * 51));
  printf ("\n size4 : %u" , sizeof (i/2.0f));
  printf ("\n size5 : %u", sizeof (i = i* 10));
  printf ("\n i = %d", i);
}
```

O/P :
```
size 1: 2     // int
size 2: 8     // int/ double → double
size 3: 8     // int * long → long
size 4 : 4    // int / float → float
size 5 : 2    // int * int → int
 i = 10
```

- In sizeof Operator any kind of expressions can be evaluated except assignment.
- In sizeof Operator directly or indirectly, assignment related expressions are not evaluated

```
void main()
{
  int a ;
  a = 10;
  printf ("\n size of a : %u", size of (++a));     //a = a+1;
```

```
    printf ("\n a = %d", a);
}
```

O/P :- size of a : 2
a = 10

→
```
void main (){
    int i;
    i = -1;
    if (i > sizeof(i))      // Here signed is compared with unsigned
      printf ("Welcome");
    else printf ("Hello");
}
```

O/P :- Welcome

if (i > sizeof(i))

-1 (signed) → 65535 (unsigned) > 2

2 bytes (unsigned)

→
```
void main()
{
  int a, b, c;
  a = b = 1;
  c = ++a > 1 || ++b > 1;
  printf ("%d %d %d", a,b,c);
}
```

O/P :- 2 > 1 || right side will not be evaluated. | 2 | 1 | 1 |

- In logical OR operator, if anyone of the expression is true then return value is 1
- In logical OR operator, when left side expression is true right sigle will not be evaluated.

→
```
void main ()
{ int a,b,c;
  a = 1; b = 2;
  c = --a > 1 || ++b > 2;
  printf ("%d %d %d", a,b,c);
}
```

O/P :- | 0 | 3 | 1 |

→
```
void main ()
{
  int a,b,c,d;
  a = 1; b = 2; c = 3;
  d = ++a < 1 || --b < 2 || ++c > 3;
  printf ("%d %d %d %d", a,b,c,d);
```

O/P :- 2 1 3 1

In logical 'OR' operator, evaluation required to stop when expression becomes true

→ void main()
{
int a,b,c;
a=b=1;
c= ++a<1 && ++b>1;
printf("%d %d %d", a,b,c);
}

O/P:- 2 1 0

- When we are working with logical 'AND' operator both expressions or all expressions must be true, then only return value is (1)
- In logical 'AND' operator when left side expression is false then right side will not be evaluated.

→ Void main()
{
int a,b,c;
a=1; b=2; ——→ c= ++a >> 1 && --b<2;
printf("%d %d %d", a,b,c);
}

O/P:- 2 1 1

→ void main()
{
int a,b,c,d;
a=1; b=2; c=3;
d= ++a>1 && --b>2 && ++c>3; 2>1 && 1>2
printf("%d %d %d %d", a,b,c,d);
}

O/P:- 2 1 3 0

- In logical 'AND' operator, required to stop the evaluation when the expression became false.

OR, AND Combinations

→ void main()
{
int a,b,c,d;
a=1; b=2; c=3;
d= ++a>1 || --b<2 && ++c>3; 2>1 || (1<2) (True, X)
printf("%d %d %d %d", a,b,c,d);
}

O/P: 2 2 3 1

(1) exp1 || exp2 && exp3
when true ↓ this will not be evaluated

2) exp1 (False) || exp2 (False) && exp3 ✗;

3) exp1 || exp2 (True) && exp3 ✓;

→ void main()
{
int a, b, c, d
a = 1; b = 2; c = 3;
d = ++a < 1 && --b < 2 || ++c > 3; 2 < 1 &&
printf("%d %d %d %d", a, b, c, d);
}

O/P :- 2 2 4 1

1) exp1 (F) && exp2 || exp3;

2) exp1 (T) && exp2 (T) || exp3 ✗;

3) exp1 (T) && exp2 || exp3;
(exp2 → F)

2/7/2015.

POINTERS...

→ Pointer is a derived data type in 'C' which is constructed from fundamental datatype of 'C' language

→ Pointer is a variable which holds address of another variable.

Advantages of Pointer

1) Data Access
- 'C' programming language is a procedure-oriented lang. i.e applications are developed by using functions.
- From one func^n to another func^n, when we required to access the data, pointer is required

2) Memory Management
- By using pointers only, dynamic m/m allocation is possible.

3) Database/Datastructures
- Any kind of data structures required to develop by using pointers only, if datastructures are not available then database is not possible to create.

4) Performance
- By using pointers, we can increase the execution speed of the program.

→ When we are working with pointers, we required to use following operators:-

1. & :- address of operator
2. * :- Indirection operator or dereference operator or object at location or value at address.

→ Address of operator always returns base address of variable.
→ Starting cell address of any variable is called base address.
→ Indirection operator always returns value of a address; i.e what address we are passing, from that address corresponding value will be retrieved.

Syntax to create a pointer

```
Datatype *ptrName;
```

→ Acc. to syntax, Indirection operator must be required b/w datatype & ptrName.
→ Space is not mandatory to place in declaration of pointer variable.

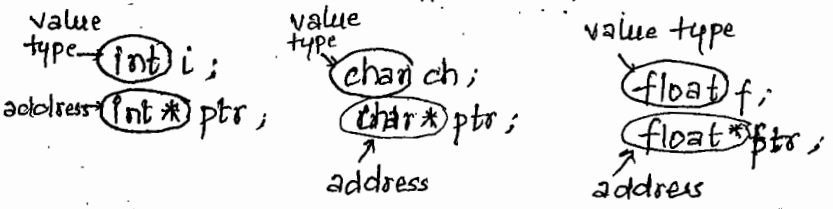
Syntax to initialise a pointer

```
Datatype variable;
Datatype * ptr = &variable;
```

⇒
```
void main()
{
  int i;
  int * ptr;
}
```

→ i is a variable of type integer, it is a value type variable which holds an integer value

→ ptr is a variable of type an int*, it is a address type variable which holds an integer variable address



Memory Management in TC3.0 (Memory Architecture)

→ TC3.0 designed on 8086 Architecture

→ The platform of 8086 Architecture is dos 16 bit OS

→ On this architecture, when we are designing a 'c' application, it occupies 1 Mb m/m only.

→ This complete 1 MB data is divided into 16 equivalent parts called segments

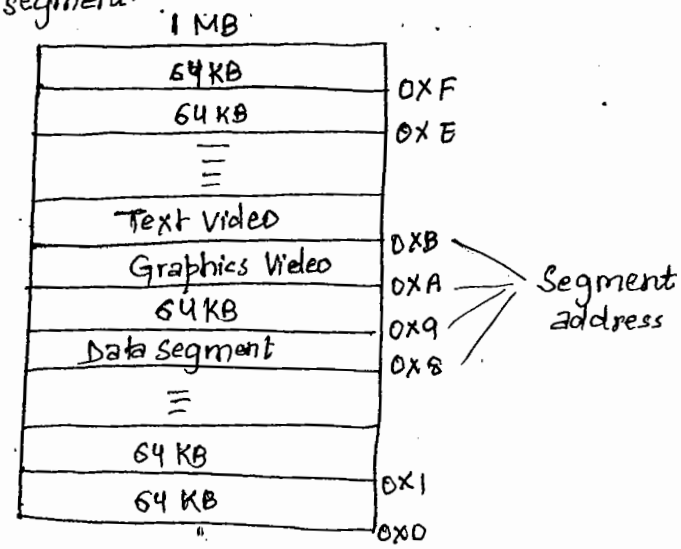
→ Each and every segment having a unique identification value called Segment address which starts from 0x0 and ends with 0xF.

→ Among those all segments, 9th segment is called data segment i.e 0x8, 11th segment is called Graphics Video Segment & next segment is called Text Video Segment.

→ Whenever we are creating any type of variable, then that variable will occupy the m/m within the data segment only.

→ Graphics related resources are available in Graphics Video segment & printing process will take place in text video segment.

TC-3.0
↓
8086
↓
Dos 16 bit OS
↓
1 MB
↓
16 segments
↓
64 KB



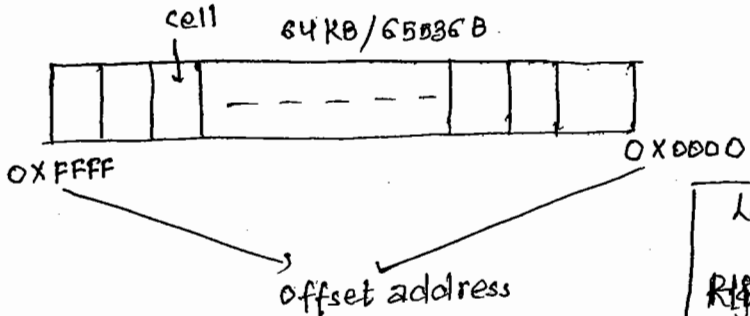
→ Each segment capacity is (64KB) only.

→ This complete (1KB) data is divided into small partitions called cells

→ Each cell capacity is 1 byte only.

→ Each and every cell having a unique identification value called offset Address which starts from 0X0000 and ends with 0X FFFF

Physical add. of a variable means combination of offset Address and Segment Address.



Low order - right side

High order - left side

always acc. to standard

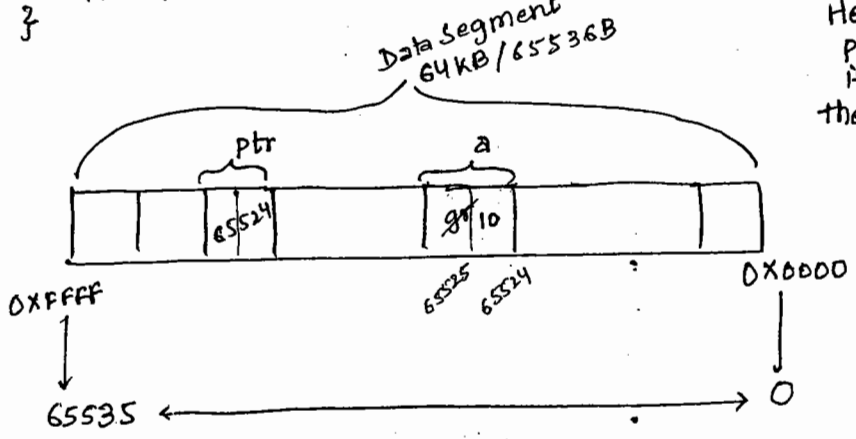
→
```
void main()
{
  int a;
  int *ptr;
  ptr= &a;
  a=10;
  printf("\n%p %p", &a, ptr);
  printf("\n%p %p", a, *ptr);
  *ptr = 20;   // a=20
  printf("\n%u %u", &a, ptr);
  printf("\n%d %d", a, *ptr);
}
```

%p → will print physical address only in 16 bit not in 32 bit

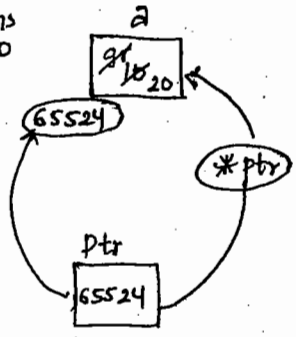
O/P :-

Imaginary output

Address (H)	Address (H)
10	10
Address (D)	Address (D)
20	20



Here in code
ptr = &a
if ptr contains address of a
then *ptr = 20
it contains value of 20



65535 → 1111 1111 1111 1111

65524 → 1111 1111 1111 0100
F F F 4

O/P :-

FFF4	FFF4
10	10
65524	65524
20	20

→ In implementation, when we required to print the addition of a variable then we are required to use %p, %x, %u, %lp or %lu format specifier.

→ %p, %x, %lp → will print the address in the form of (Hexadecimal.)

→ %u, %lu → will print the address in the form of (decimal.)

→ %p, %x, %u → will print 16 bit physical address i.e offset value only.

→ %lp, %lu → will print 32 bit physical address i.e segment address & offset address also

NOTE:- for printing the address of a variable, we can't use %d format specifier because-

1) Addresses are available in the form of Hexadecimal from the range of 0X0000 to 0XFFFF in decimal (0 to 65536) so this range is not compatible.

2) There is no any -ve values are available in addresses but %d will print -ve data also

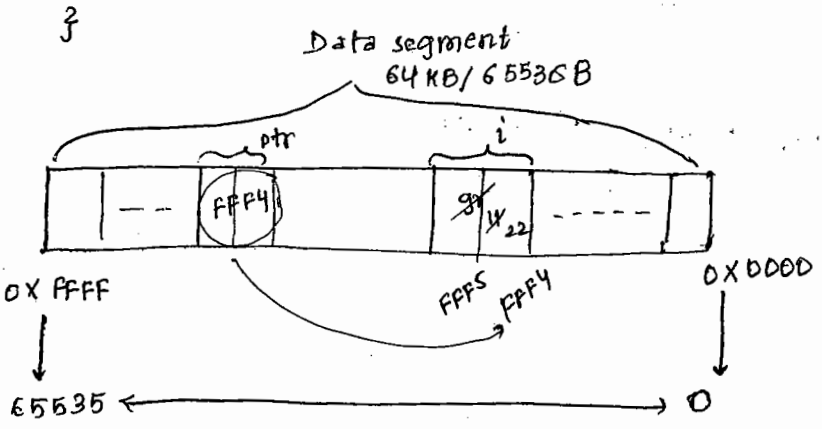
---

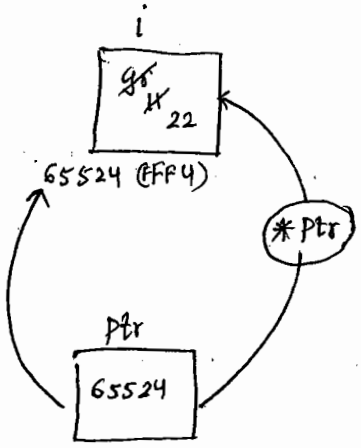
```
→ void main()
  {
     int i;
     int * ptr;
     ptr = &i;
     i = 11;
     printf("%p %p", &i, ptr);
     printf("\n%d . %d , i, *ptr);

    * ptr = 22;
     printf("\n %x %x", &i, ptr);
     printf("\n%d %d", i, *ptr);
  }
```

O/P :-

FFF4	FFF4
11	11
fff4	fff4
22	22

0X0000 → 0000 0000 0000 0000
0XFFFF → 1111 1111 1111 1111
0XFFF4 → 1111 1111 1111 0100





%p format specifier → prints the address in form of Hexadecimal & alphabets are printed in Upper case.

%x format specifier → prints the address in form of Hexadecimal & alphabets are printed in lower case.

→ Address of operators always returns base address i.e starting cell address of a variable.

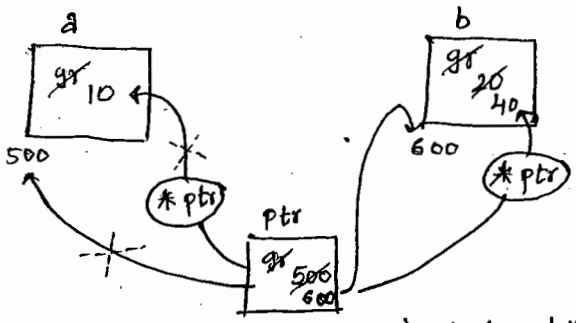
→ In previous prog, ptr holds an integer variable address thats why if cmd ptr in direction operator always returns value of the address.

→ i and *ptr values are same because ptr is holding address of i.

```
void main()
{ int a, b;
  int *ptr;
  ptr = &a;
  a = 10;
  b = 20;
  printf ("\n%d %d %d", a, b, *ptr);
  *ptr = 30;
   ptr = &b; - now pointer is pointing to b
   b = 40;
   printf("\n%d %d %d ", a, b, *ptr);
}
```

O/P :-

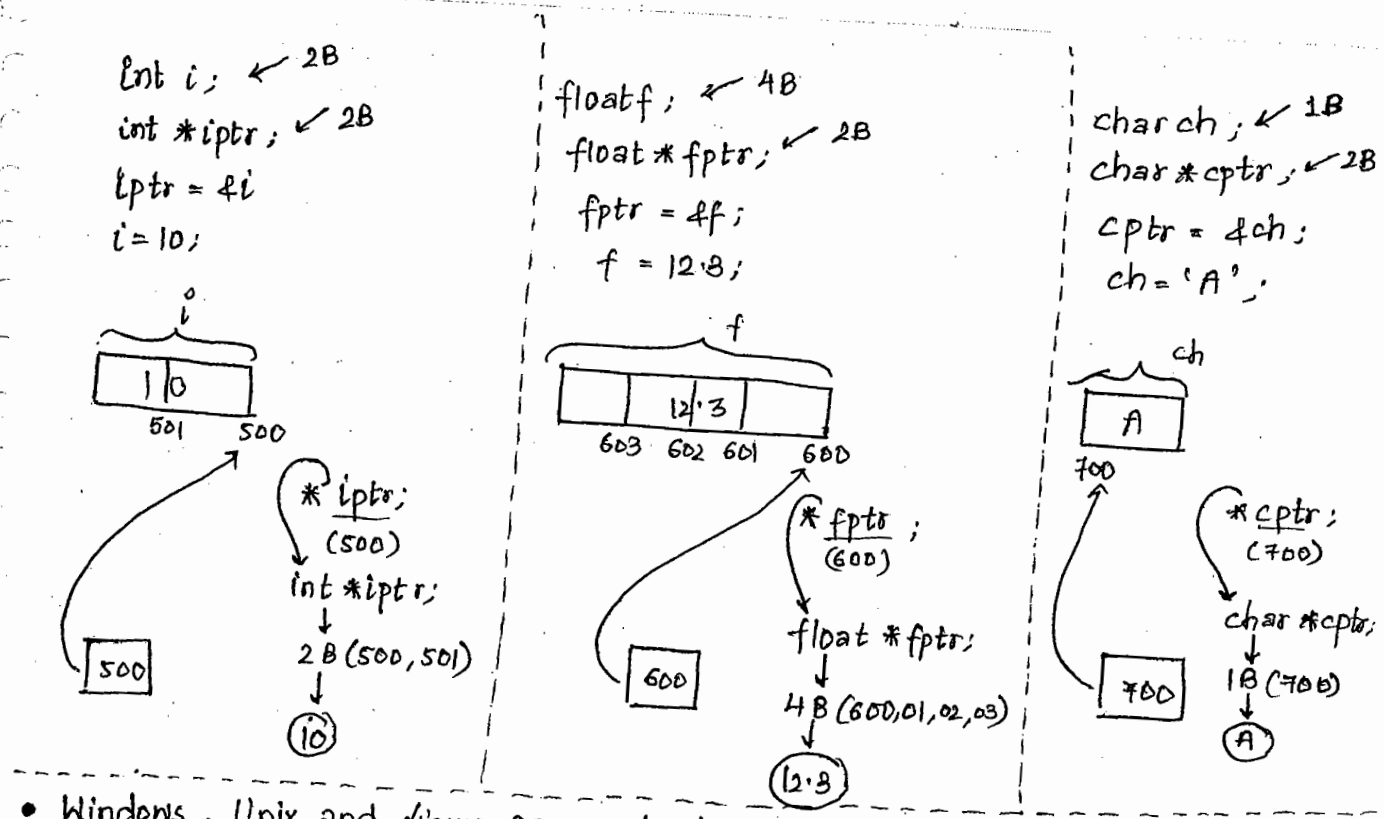
10	20	10
30	40	40



In implementation when we required to shift the pointer from one variable to another variable then we are required to reassign address of a variable to ptr

DATA ACCESS MECHANISM USING POINTERS

- Different types of variables are having diff. types of sizes bcoz internal content is diff.
- Any type of pointer having same size only because internal content is address which is common for any type of variable.
- DOS os doesn't having any m/m management that's why at run time addresses are limited.
- On DOS based compiler (TC-3.0, 8086 based, 16 bit compiler). The size of pointer is 2 bytes bcoz addresses are limited.



- Windows, Unix and Linux OS are having proper <u>m/m management</u> that's why addresses are not limited (depends on RAM capacity)
- On windows based compiler (TC-4.5, c-free, Devctt, gcc compiler, 32 bit compiler) The size of the pointer is <u>4 bytes</u> because addresses are unlimited.
- On 64-bit OS, when we are using 64 bit compiler then size of the pointer is 8 bytes.

32 bit OS, 16 bit compiler size of pointer is 2 bytes
64 bit OS, 16 bit compiler not compatible
only 32 bit compiler and 64 bit compiler will work.

↓ OS/compiler →	16 bit	32 bit	64 bit
16 bit	2B	2B	X
32 bit	X	4B	4B
64 bit	X	X	8B

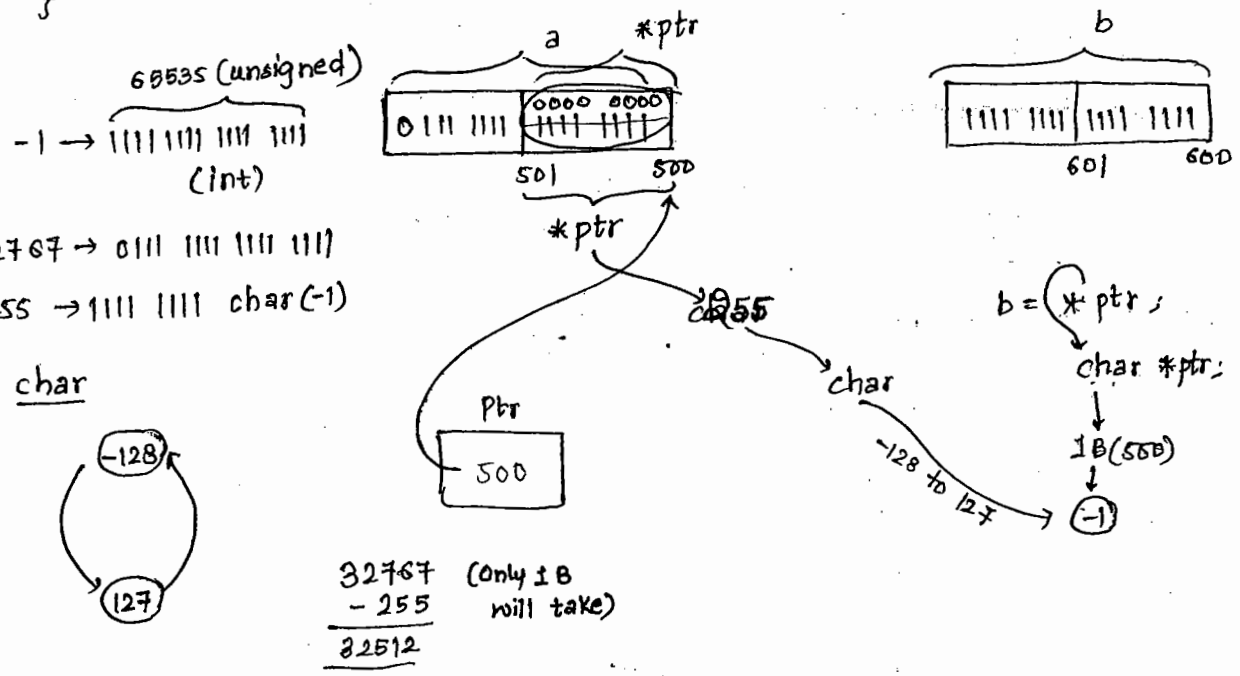
- Any type of pointer size is 2 bytes only because it maintains the offset address from the range of 0X0000 to 0XFFFF, so for holding this many address we require 2 B data
- Any type of pointer holds single cell info only i.e Base address but data can be present in multiple cells.

- On address when we are applying indirection operator then it defelence to pointer type, So depending on pointer type given base address (in no. of bytes will be accessed)
- In 'C' prog. lang, any type of pointer can holds, any kind of variable address because address doesn't maintain any type information.
- In implementation when we are manipulating an integer variable then go for an int*, float variable float* and character variable is char* because we can see the difference when we are applying indirection operator.

→
```
void main()
{
  int a, b;
  char * ptr;
  ptr = &a;
  a = 32767;
  b = * ptr;
  printf("\n%d %d %d", a, b, *ptr);

  * ptr = 0;
  printf("\n%d %d %d", a, b, *ptr);
}
```

O/P :=

32767	-1	-1
32512	0	0

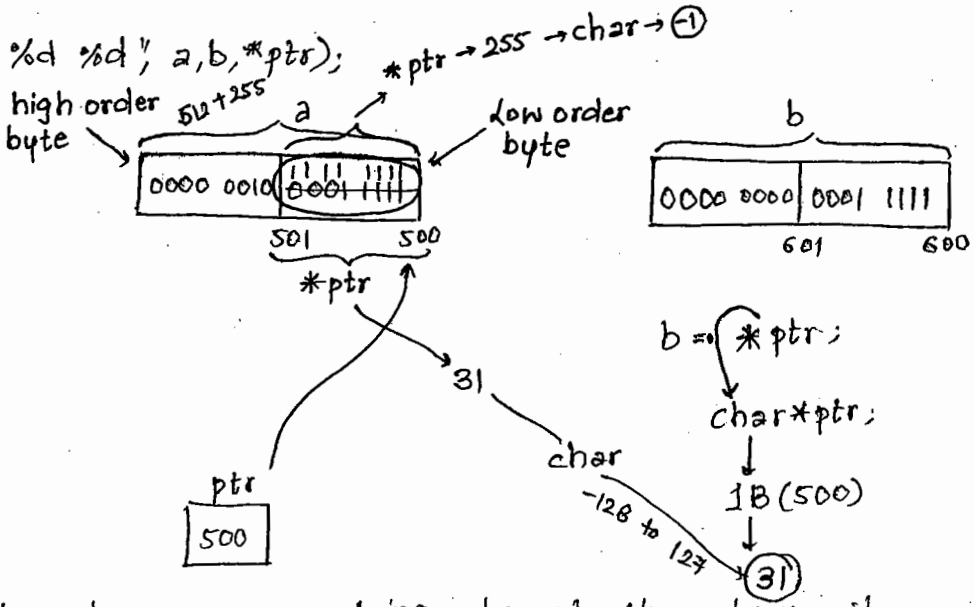


- On integer variable, when we are applying char pointer, then it can access and manipulate only (1Byte) data, because indirection operator behaviour is datatype dependent.

→
```
void main()
{ int a,b;
 char *ptr;
  ptr = &a;
  a = 548;
  b = *ptr;
  printf ("\n%d %d %d",a, b,*ptr);

 *ptr = 255;
  printf ("\n%d %d %d", a,b,*ptr);
}
```

O/P :-

548	31	31
767	31	-1



→ All integer variable, when we are applying char ptr then always it will access low order byte data only.

→ When we are working with 2 byte integer, then 1st byte is called low order byte and 2nd byte data is called high order byte.

→ When we are working with pointers, generally we are getting following errors :-

1. Suspicious pointer conversion :-
   - This warning msg occurs when we are assigning address of a variable in diff. type of pointer.
   - These conversions are not allowed in C++.

2. Non-portable pointer conversion :-
   This warning msg occurs when we are assigning value type data to a pointer.

Ex-
```
void main()
{
  int a;
  int *ptr;
   ptr = a
}
```

Arithmetic Operations On Pointers :-

```
int P1, P2;
int *P1, *P2;
P1 = &i1;
P2 = &i2;
```

1) P1 + P2 ; Error
2) P1 + 1 ; Next address
3) ++P1 ; } Next address
   P1++ ; }
4) P2 - P1 ; No. of elements
5) P2 - 1 ; Pre address
6) --P2 ; } Pre Address
   P2-- ; }

1) P1 * P2 ; } Error
   P1 * 2 ; }
2) P1 / P2 ; } Error
   P1/3 ; }
3) P1 % P2 ; } Error
   P1 % 2 ; }

Pointer Rules

Rule 1 :-

Address + Number = Address (Next Address)
Address - Number = Address (Pre Address)
Address ++ = Address (Next Address)
Address -- = Address (Pre Address)
++ Address = Address (Next Address)
-- Address = Address (Pre Address)

Rule 2 :-

Address - Address = Number (No. of elements)
= Size Diff / sizeof (Datatype)

int *p1 = (int*) 100
int *p2 = (int*) 200
p2 - p1 = 50
200 - 100 → 100 / size of (int)

Rule 3 :-

Address + Address = Illegal
Address * Address = Illegal
Address / Address = Illegal
Address % Address = Illegal

Rule 4 :-

We can't perform bitwise operation b/w 2 pointers like

Address & Address = Illegal

Address | Address = Illegal

Address ^ Address = Illegal

~ Address = Illegal

Rule 5 :-

We can use relational operator and conditional operator b/w two pointers (<, >, <=, >=, ==, !=, ?:)

Address > Address = T/F

Address >= Address = T/F

Rule 6 :- We can find size of a pointer using sizeof operator

Pointer-to Pointer

→ It is a procedure of holding the pointer address into another pointer variable.

→ In C prog. lang., pointer to pointer relations can be applied upto **12 stages.**

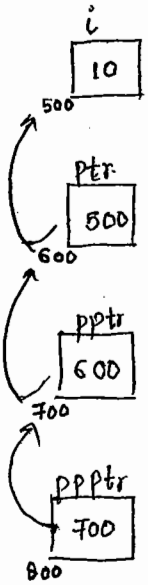
→ For a pointer variable, we can apply **12 indirection operators.**

→ When we are inc. pointer to pointer relations then performance will be decreased.

Syntax :-

pointer :	Datatype *ptr;
P2 pointer :	Datatype **ptr;
P2 P2 pointer :	Datatype ***ptr;
P2 P2 P2 p :	Datatype ****ptr;

```
void main()
{
  int i;
  int *ptr;
  int **pptr;
  int ***ppptr;
  ptr = &i;
  pptr = &ptr;
  ppptr = &pptr;
  i = 10;
}
```



① &i → 500

2) ptr → 500

3) i → 10

4) *ptr → 10

1) &ptr → 600

2) pptr → 600

3) *pptr → 500

4) **pptr → 10

*(600)

↓

*(500)

↓

(10)

1) &ptr → 700

2) ppptr → 700

3) *ppptr → 600

4) **pptr → 500

5) ***ptr → 10

*(700)

↓

*(600)

↓

*(500)

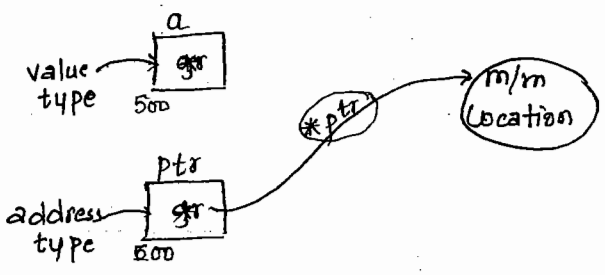
↓

(10)

→ Wild pointer :-

- Uninitialized pointer variable or the pointer which is not initiatialized with any variable add. is called wild pointer
- Wild pointer is also called bad pointer bcz without assigning any variable address, it is pointing to a m/m location

```
void main()
{
  int a;
  int *ptr; // Wild or bad pointer
}
```



→ When we are working with pointers, always recommended to initialise with any variable address or make it NULL.

NULL Pointer :-

- The pointer variable which is initialised with null value it is called null pointer
- Null pointer doesn't points to any m/m location until we are not assigning the address
- Sizeof Null pointer also is 2 bytes acc. to DOC compiler.

Syntax :

Datatype * ptr = NULL ; <stdio.h>

Eg :- int* ptr = NULL;

char*ptr = NULL;

float *ptr = NULL;

Datatype * ptr = (Datatype *) NULL;

Ex: int * ptr = (int *) NULL;
char * ptr = (char *) NULL;
float * ptr = (float *) NULL;

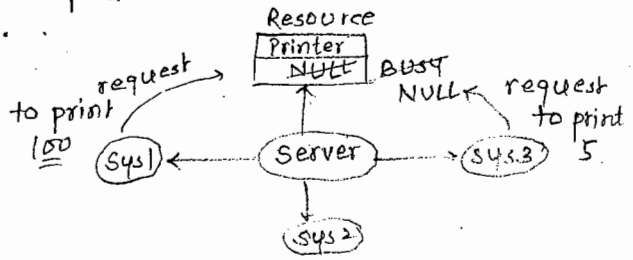
Datatype * ptr = (Datatype *) 0;

Ex: int * ptr = (int *) 0;
char * ptr = (char *) 0;
float * ptr = (float *) 0;

- Null pointer can be used like error code of a function
- NULL can be used like a constant integral value.

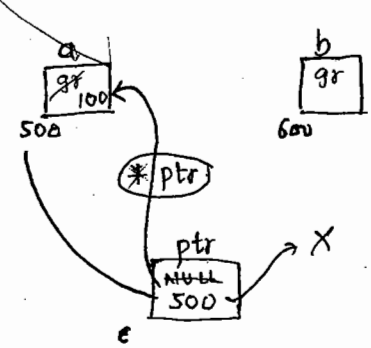
int x = NULL; // int x = 0;

- By using NULL, we can check the status of a resource, i.e it is busy with any process or free to utilize.




```
→ void main()
{
  int a, b;
  int *ptr = (int*)0;
            // int*ptr = NULL;   <stdio.h>
  if (ptr==0)
  {
    ptr = &a;
     a = 100;
  }
  if (ptr == (int *)0)
  {
    ptr = &b;
     b = 200;
  }
  printf (" Value of *ptr : %d", *ptr);
}
```

O/P: Value of *ptr = 100



1) Check whethe ptr is null or not

if (ptr == 0)
NULL == 0 True
{
ptr = &a
then put 500
In a = 100
}
then ptr ≠ NULL
not access b

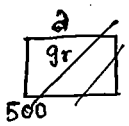
ptr
NULL
500

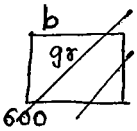
→ void main()

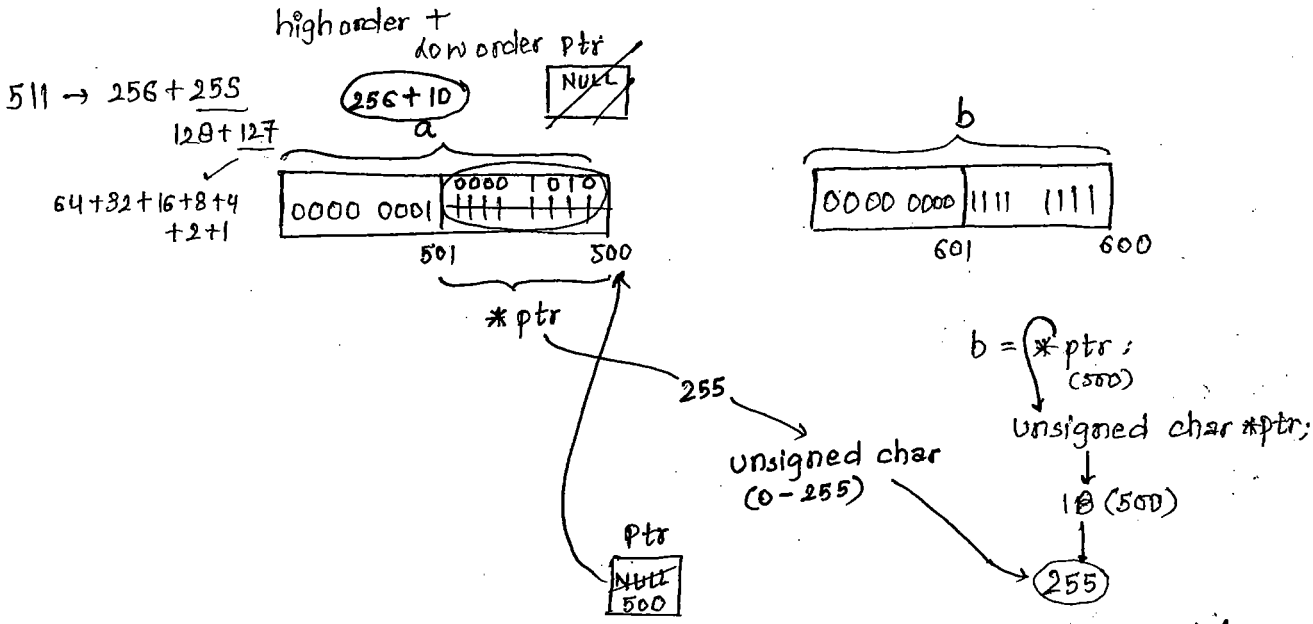
```
{ int a,b;
  unsigned char* ptr = (unsigned char*)0;
  ptr = &a;
  a = 511;
  b = *ptr;
  printf("\n%d %d %d", a, b, *ptr);
  *ptr = 10;
  printf("\n%d %d %d", a, b, *ptr);
}
```

O/P:

511	255	255
266	255	0







- On integer variable when we are applying unsigned character pointer then it can access and manipulate 1B data only bcz indirection operator behaviour is data type dependent.

- TC-3.0 designed on 8086 Architecture.
  In this Architecture m/m models are classified into 6 types i.e.
  (1) Tiny
  (2) Small
  (3) Medium
  (4) Compact
  (5) Large
  (6) Huge

→ By default, any application m/m model is small in TC 3-0

→ In TC-4.5 m/m models are classified into 4 types i.e.

- Small
- Medium
- Compact
- Large

→ By default, any application m/m model is large in TC 4.5

- Always m/m model will decides that what type of pointer required to create.
- Depending on m/m model pointers are classified into 3 types :-
  1. near pointer
  2. far pointer
  3. huge pointer

1) near pointer - The pointer variable which can handle only 1 segment of 1MB data it is called near pointer.

→ near pointer doesn't points to any other segments except data segment.

→ The size of near pointer is 2 bytes.

→ When we are incrementing the near pointer value then it inc. Offset address only

→ When we are applying the relational operators on near pointer then it compares Offset address only

→ By default, any type of pointer is near only.

→ When we are printing the address, by using near pointer, then we required to use %d or %x or %u format specifier only

→ By using near keyword, we can create near pointer.

2) far pointer - The pointer variable which can handle any segment of 1MB data, is called far pointer.

- When we are working with far pointer, it can handle any segment from the range of 0X0 - 0XF, but at a time only 1 segment.
- The size of far pointer is 4 bytes bcz it holds segment & offset add. also.
- When we are incrementing the far pointer value, then it increases offset address only.
- When we are applying relational operators on far pointer then it compares segment address along with offset address.
- When we are printing the address by ~~prin~~ using far pointer then we required to use %lp or %lu format specifier
- By using far keyword, we can create far pointer.

3) huge Pointer:- The pointer variable which can handle any segment of 1MB data is called huge pointer

- When we are working with huge pointer, it can handle any segment from the range of 0X0 – 0XF, but at a time only 1 segment.
- When we are working with huge pointer, then it occupies 4B of m/m & it holds segment & offset address also.
- When we are incrementing the huge pointer value, then it increase segment address along with offset address
- When we are comparing the huge pointer, it compares normalisation value.
- When we are printing the address by using huge pointer then we required to use %lp or %lu format specifier.
- By using huge Keyword, we can create huge pointer.

* Normalisation - is a process of converting (32 bit) physical bit into (20 bit) hexadecimal format.

IN DECIMAL

Physical address = (Segment address) * 16 + Offset Address

IN HEXADECIMAL

Physical address = (Segment Address) * 0X10 + Offset Address

What is the normalisation value of 0X12345678 huge address ?

huge address :- 0X12345678

Segment address :- 0X1234

Offset Address :- 0X5678

Physical address = (Segment Address) * 0X10 + Offset Address

= 0X1234 * 0X10 + 0X5678

= 0X12340 + 0X5678

= 0X179B8

In binary :- 0001 0111 1001 1011 1000

→ One's is moved to ten's
→ 10's to 100's
→ 100's to 1000's
→ 1000's to 10,000's

1 * 10 = 10
12 * 10 = 120
123 * 10 = 1230
1234 * 10 = 12340

01 * 010 = 010
012 * 010 = 0120
0123 * 010 = 01230

1 * 8 = 8
10 * 8 = 80

0X1 * 0X10
0X12 * 0X10
0X1234 * 0X10
= 0X12340

1 * 16 = 16
18 * 16 =

Void Pointer

→ Generic Pointer of C and C++ is called void Pointer

→ Generic Pointer means it can access and manipulate any kind of data properly. (multiple datatype, 1 pointer)

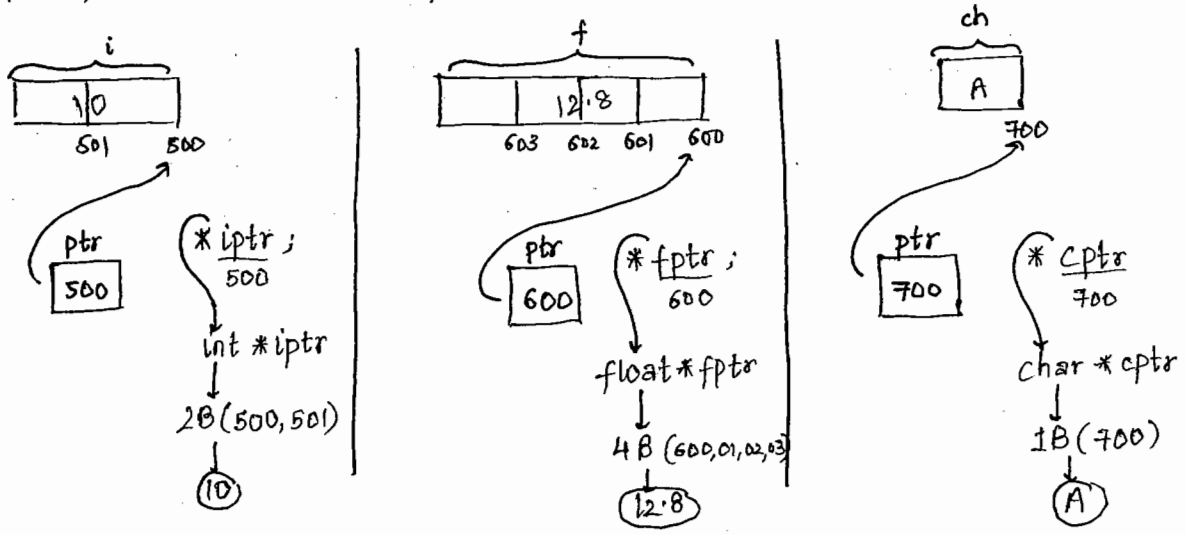
→ Size of void Pointer is 2 bytes

→ By using void Pointer, when we are accessing the data then we required to use Type Casting.

→ When we are working with void Pointer, Type specification will be decided at run time only.

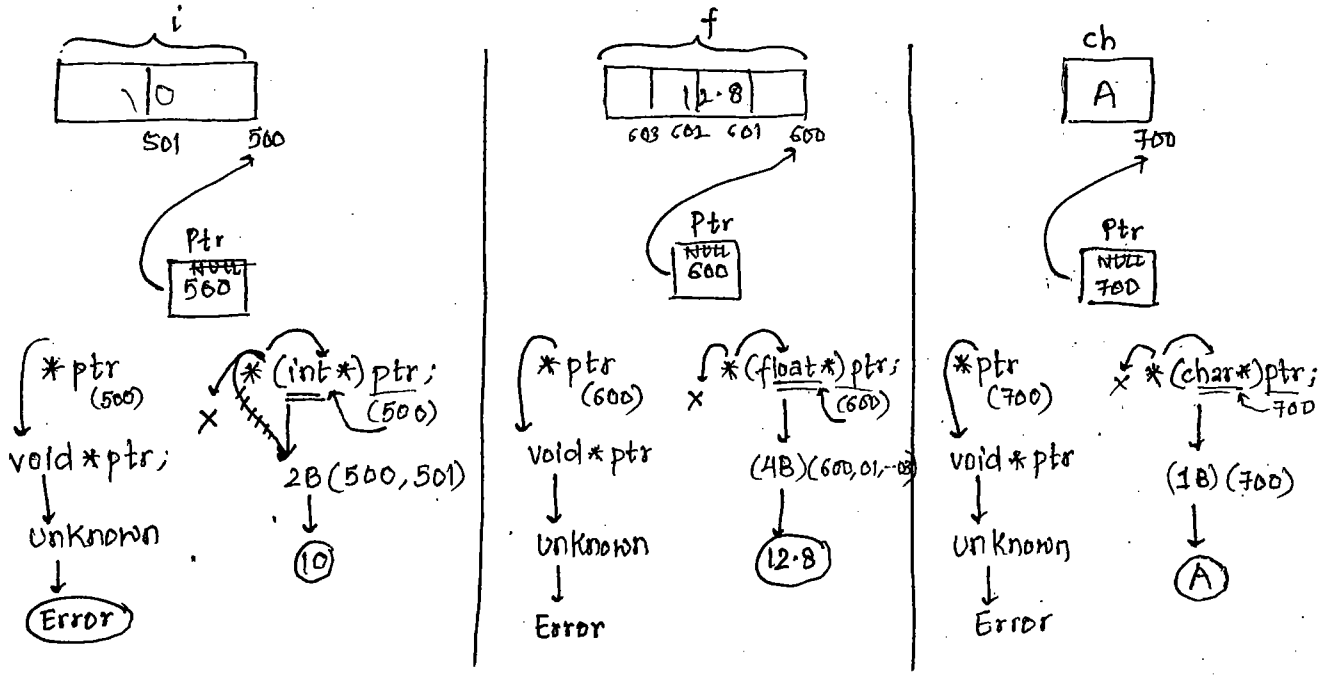
→ When we are working with void Pointer, arithmetic operations are not allowed, i.e. Incrementation and decrementation of pointer is restricted.

```
void main()
{
  int i;
  float f;
  char ch;
  int* iptr = (int*)0;
  float *fptr = (float*)0;
  char* cptr = (char*)0;

  iptr = &i;
  i = 10;
  printf("\n%d %d", i, *iptr);

  fptr = &f;
  f = 12.8;
  printf("\n%f %f", f, *fptr);

  cptr = &ch;
  ch = 'A';
  printf("\n%c %c", ch, *cptr);
}
```

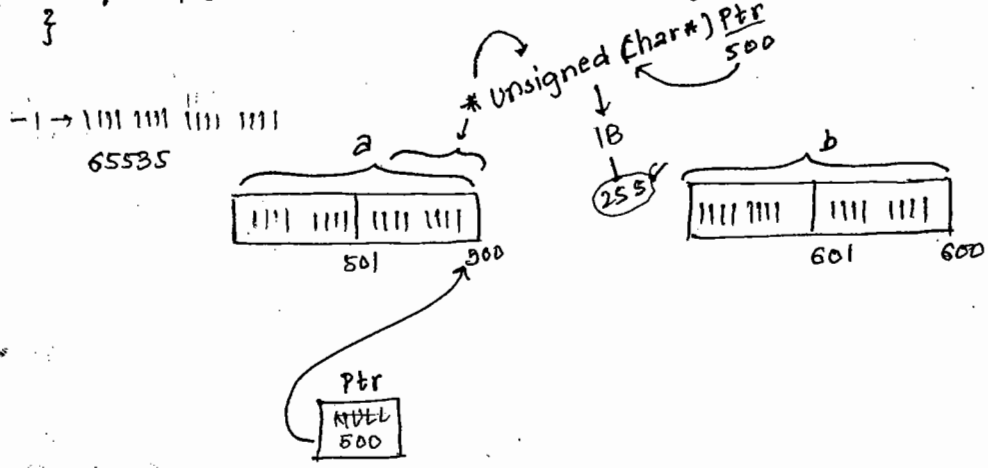


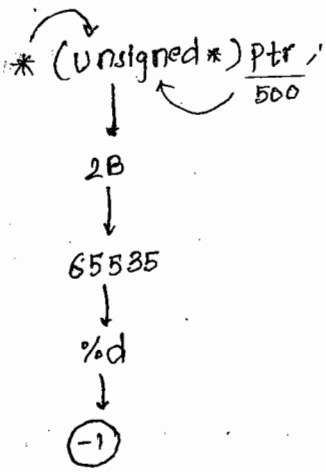
→ In previous prog., in place of constructing 3 types of pointers, we can create a single pointer variable which can access & manipulate any kind of variable properly. i.e void pointer required to use.

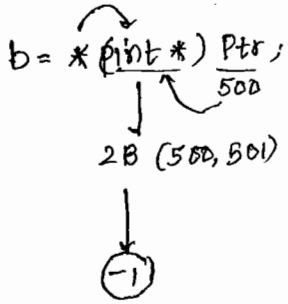
```
** void main()
   {
    int i;
    float f;
    char ch;
    void *ptr;

    ptr = &i;
    i = 10;                          // *(int*)ptr = 10;
    printf ("\n%d %d", i, *(int*)ptr);

    ptr = &f;
    f = 12.8;
    printf ("\n%f %f", f, *(float*)ptr);   // *(float*)ptr = 12.8;

    ptr = &ch;
    ch = 'A';
    printf ("\n%ch %c", c, *(char*)ptr);   // *(char*)ptr = 'A';
```

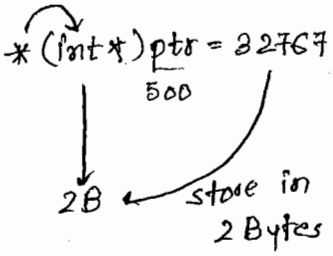


```
void main()
{
    int a,b;
    char* ptr;
    ptr = &a;
    a = -1;
    b = *(int*)ptr;
    printf("\n%d %d %d", a, b, *(unsigned*)ptr);
   *(int*)ptr = 32767;
    printf("\n%d %d %d", a, b, *(unsigned char*)ptr);
}
```





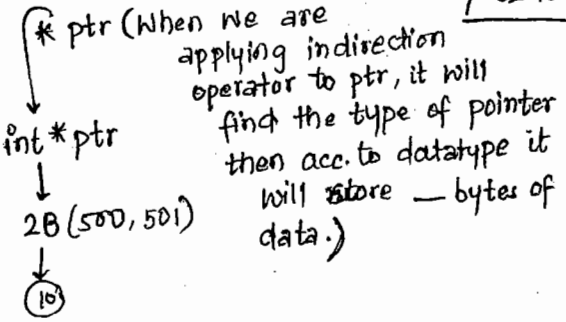




O/P :-

-1	-1	-1
32767	-1	255

Explaination



FUNCTIONS

Self contained block of 1 or more statements or a sub program which is designed for a particular task is known as function

ADVANTAGES

1) Module Approach :- By using functions we can develop the application in module format ie procedure oriented language concept.

2) Reusability :- By using functions we can create reusability blocks i.e
develop once and use Multiple times

3) By using functions, we can easily Debug the program.

4) Code Maintainence - When we are developing the application by using functions, then it is easy to maintain code for future enhancement.

- The basic purpose of function is Code reuse
- A 'C' prog. is a collection of functions.
- Always compilation process will take place from Top to Bottom.
- Execution process starts from main and ends with main only.
- When we are working with functions, functions can be defined randomly.
- From any func. we can invoke (call) any another function.
- When we are calling a function, which is defined later for avoiding the compilation error, we required to go for forward declaration

- Declaration of a function means required to specify return type, name of the function & parameter type information
- In Function Definition, first line is called function header / function prototype.
- Always function declaration must be required to match with function declaration.
- When the function is not returning any value then specify the return type as void.
- Void means nothing i.e no return type or value
- Default returntype of func^{n} is an int.
- Default parameter type of func is void

- When the function is returning the value, we required to specify the return type is other than void i.e what type of data it is returning, same type of return statement is required to be specified.
- When the function is returning the value, specifying the return statement is optional.
  In this case compiler will give a warning message i.e function should return a value.
- When the function is returning the value, collecting the value is always optional
  In this case compiler will not give any warning message or error.

Syntax :-

```
return_type FunctionName(parameters)
{
  function_statement;
  —
  —
  return statement;
}
```

- Acc. to syntax, specifying the return type, parameters and return statements are optional.
- All the rules of variable declarations are applicable to function name also.

Functions are classified into 2 types-

1) Library functions/ Built-in functions/ Pre-defined funcⁿs.
2) User-defined functions

1) **Library Functions** :- Library functions are set of pre implemented functions which are available along with compiler.

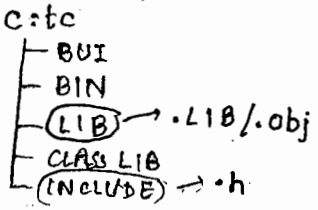
↦ The implementation part of library functions are available in .lib or .obj files which is available in C:\tc\LIB directory

↦ .lib or .obj files contains pre-compiled object code

↦ When we are working with pre-defined functions for avoiding the compilation error we required to go for forward declaration i.e prototype is required

↦ When we required to provide of prototype of pre-defined functions, then required to go for header-files

↦ .h files does not provide any implementation part of predefined functions, it provides only prototype, i.e forward declaration of function.



When we install C software, some pre-defined functions are automatically come with that in C: directory.

Limitations

- All predefined functions contain Limited task only i.e for what purpose func^{n} is developed, for same purpose we required to use.
- As a programmer, we doesn't having control on predefined functions.
- As a programmer, it is not possible to alter or modify the behaviour of any pre-defined functions.

eg:- printf(), scanf(), clrscr(), getch()
strcpy(), pow(), sqrt()

- When pre-defined functions are not supporting user requirement then go for user-defined functions

2) **User-defined Functions** :-

→ As per client or project requirements, the functions we are developing are called user-defined functions.

→ Always user-defined functions are client specific functions or project specific functions only.

→ As a programmer, we are having full control on user-defined ~~defined~~ functions.

→ As a programmer, it is possible to alter or modify the behaviour of any user-defined functions if it is required because coding part is available.

→ Depends on return type and parameter type.

→ User-defined functions are classified into 4 types -

1) No return type with no parameters
2) No return type with parameter
3) With return type without parameters
4) With return type with parameter.

NOTE: All predefined functions are user-defined functions only bcoz somewhere else another programmer developed these functions & we are using in the form of object code or compile code.

- Whenever we are using any functions in compiled format then it is called pre-defined functions.

```
void printf()
 {
 }
void main()
 {
   printf("Hello");
 }
```

O/P :- [Blank]

It is compiled & executed, don't give any error but don't access pre-defined function access only user-defined function.

- It is possible to place user-defined function name as similar to pre-defined function name also.
- When both are same then it is not possible to call predefined function.

About main() :-

- Main is an identifier in the program which indicates startup point of an Application (Every function name is identifier)
- Main is a user-defined functions with pre-defined signature for Linker.
- A Linker is a Assembly Language Program which always decides startup point of the program is main.

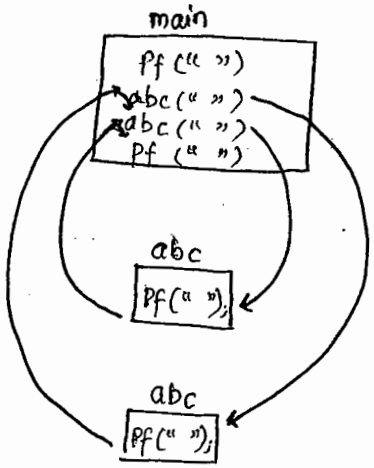
C	CPP	Java	C#
Void main()	int main()	p s v main()	void main()
{	{	{	{
}	}	}	}

- **It is possible to change name of the main function if it is required but to execute the program, we required to place one more function with the name called main()
- It is possible to develop a program without using main function also. In this case Compilation is success but Linking is failure.
- In any kind of Application, onky one ~~kind~~ of function is required to place with any 1 unique name i.e. multiple main functions are not possible.
- When we are developing .Lib/.obj files for other projects then doesn't required to include main function (reusable components for multiple projects)
- Generally main function doesn't returns any value, that's why return type of main function is void.
- In implementation, when we required to provide exit status of an application then recommended to specify the return type as an int.
- Void main function doesn't provides any exit status back to the OS.
- int main function provides exit status back to the OS i.e success or failure.
- When we required to inform the exit status as success, then return value is 0, i.e return 0; or return EXIT_SUCCESS.
- When we required to inform the exit status as failure, then return value is 1, i.e return 1 ; or return EXIT_FAILURE.
- When the return type of main funcn is integer, then it is possible to return the values from the range of -32768 to 32767 but except 0 and 1, remaining values doesn't having any meaning.
- When the user is explicitly terminating the program, then return value is -1

Prog1 →
```
void abc()      //&abc
{
 printf("Hello abc\n");
}
void main()     //&main
{
 printf("Hello main1\n");
 abc();         //calling  &abc();
 abc();         //calling  &abc();
 printf("Hello main2\n");
}
```

O/P:-
```
Hello main1
Hello abc
Hello abc
Hello main2
```

Compilation starts from Top to Bottom
While execution from main()



⇒ A 'C' program is a combination of pre-defined and User-defined functions Always compilation process starts from Top to Bottom and execution process starts on main() and ends with main() only.

⇒ In order to compile a program, if any function is occured then with that function name one unique identification value is created called address of func^n.

⇒ When we are calling any function, that calling statement is substituted with corresponding func^n address called **Binding process.**

⇒ With the help of Binding process only, compiler will recognize that which func^n required to call at the time of execution.

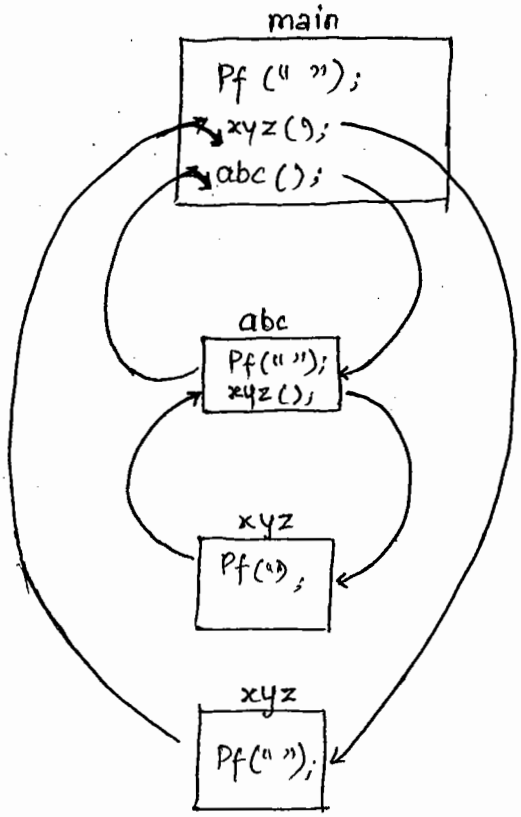
⇒ In 'C' prog. lang., always <u>static binding takes place</u> i.e Compile-Time Binding Process.

⇒ Dynamic Binding is a OOPL concept which works with the help of polymorphism.

Prog2 →
```
void xyz()          //&xyz
{
 printf("Welcome xyz\n");
}
void abc()          //&abc
{
 printf("Welcome abc\n");
 xyz();             //calling &xyz();
}
```

address is not given to printf because printf definition is not there all are user-defined functions

```
void main()        // &main
{
 printf (" Welcome main \n");
 xyz();            // calling  &xyz();
 abc();            // calling  &abc();
}
```



O/P:-
```
Welcome main
Welcome xyz
Welcome abc
Welcome xyz
```

- When we are working with functions, functions can be implemented randomly, i.e in any sequence functions can be defined.
- From any function we can call any other function
- After execution of any function, automatically control roill pass back to the calling place i.e from which location we called the func^n to same location it will pass.

Prog3 -
```
void main()
{
  printf (" Hello main \n");
  abc();
}
void abc()
 {
  printf (" Hello abc \n");
 }
```

O/P :- Error Type mismatch in redeclaration of 'abc'

Case 1:

```
Void abc()
{
}
Void main()
{
  abc();
}
Valid
```

1) Address
2) Type information
   - return type - Void
   - Parameters - void
   - No. of parameters - 0

Case 2:

```
Void main()
{
  abc();
}
Void abc()
{
}
Error
```

1) Address
2) Type information (default)
   - return type - (int)
   - Parameters - void
   - No. of Parameters - 0

On this, if we write int instead of void

Soln 1

```
void main()
{
  abc();
}
int abc()
{
}
valid
```

O/P :-

- When we are working with functions, at the time of compilation along with the address, Type of information also maintain by compiler i.e return type, parameter type and no. of parameters.
- If function is compiled before calling, then along with the address, Type information also available but if func^n is calling b4 compilation then address is not available and automatically compiler take default type information i.e return type - (int)
  parameter type - void
  no. of parameters - 0
- In previous prog., at the time of compilation, we are getting Type mismatch error because return type is expecting an int but actual return Type is void.
- When we are calling a func^n which is defined later for avoiding the compilation error, we required to provide forward declaration i.e prototype of the function.

- Forward declaration means required to specify return type, function name and parameter type info
- Forward declaration statement always provides Type information explicitly so compiler doesn't take default Type info.

Soln2
```
void main()
{
  void abc(void);   // declaration
  printf("Hello main\n");
  abc();
}
void abc()
{
  printf("Hello abc\n");
}
```
O/P:-
```
Hello main
Hello abc.
```

Prog4 :-
```
void xyz();   //global decla
void abc()
{
  printf("Welcome abc\n");
  xyz();
}
void main()
{
  printf("Hello main\n");
  xyz();
  abc();
}
void xyz()
{
  printf("Welcome xyz\n");
}
```
O/P :-
Error Type Mismatch in redeclaration of 'xyz'

- In order to call the xyz function in abc(), main(), we required to provide forward declaration of xyz() in abc(), main() also.
- In implementation, when we required to provide forward declaration more than once then recommended to go for Global declaration
- When we are declaring a func at top of the prog. before defining first function then it is called Global declaration.
- When the Global declaration is available then doesn't required to go for local declaration. If local declaration also available then Global declaration is ignored.

Soln
```
void xyz();     // global declaration
void abc()
{
  //void xyz();  // local declaration
   printf("Welcome abc\n");
   xyz();
}
void main()
{
  //void xyz();  // local declaration
   printf("Hello main\n");
   xyz();
   abc();
}
void xyz()
{
   printf("Welcome xyz\n");
}
```

O/P:-
```
Hello main
Welcome xyz
Welcome abc
Welcome xyz
```

STORAGE CLASSES

Storage Classes of C will provide following information to the compiler i.e

1. Storage area of a variable.
2. Scope of a variable i.e in which block that variable is visible.
3. Lifetime of a variable i.e how long that variable will be there in active mode.
4. Default value of a variable if it is not initialised

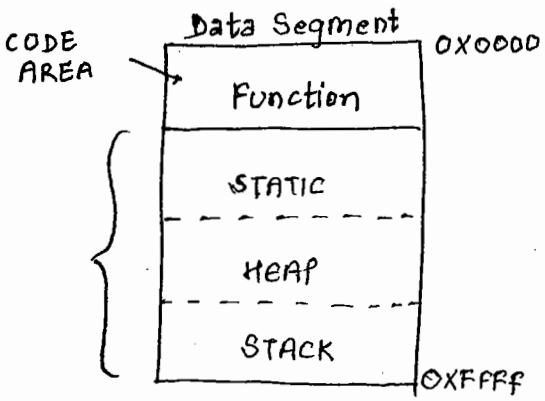
* depends on storage area and behaviour, storage classes are classified into 2 types -
  1. Automatic storage class -
  2. Static storage class

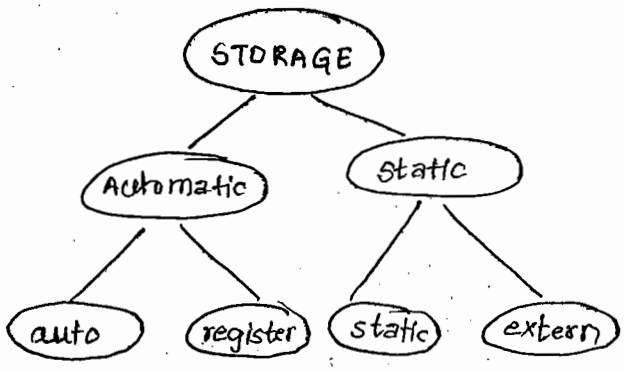
1) AUTOMATIC STORAGE CLASS

- Automatic Storage class Variables are created automatically and destroyed automatically.
- This storage class variables stored in stack area of Data Segment.
- Under automatic storage class, we are having 2 types of storage class specifiers auto, register

2) STATIC STORAGE CLASS

- This storage class variables are created only once and throughout the program it will be there in active mode.
- Static Storage Class variable are stored in static area of data segment.
- Under Static Storage Class, we are having 2 types of static storage specifiers i.e static, extern.





TYPE	SCOPE	LIFE	DEFAULT VALUE
auto	body	body	Garbage value
static	function	program	0
extern	Program	program (or) All functions	0
register	body	body	Garbage value

⇒ In 'C' prog. lang. we are having 4 types of scope
- body scope
- function scope
- file scope
- program scope

⇒ By default any variable ~~is~~ storage class specifier is auto within the body.

REGISTER VARIABLES

- It is a special kind of variables which stores in CPU register
- The basic advantage of register variable is it is faster than normal variables

- In implementation, when we are accessing a variable throughout the program n no. of times then go for register variable

Limitations

⇒ Register m/m is limited so it is not possible to create n no of register variables.

⇒ On register variables we can't apply POINTERS because register variables doesn't allows to access address.

NOTE :- Register storage class specifier just recommends to the compiler that variable need to be stored in CPU Register if memory is available or else store in stack area of data segment.

```
void main()
{
  register int r = 10;
  ++r;
  printf("\nr = %d", r);

  printf("\nEnter a value: ");
  scanf("%d", &r);
  ++r;
  printf("\nr = %d", r);
}
// Enter a value: 100
```

O/P :- Error must take address of a memory location.

In Interviews, they give

```
scanf("%d", r);
```

O/P :

```
r = 10
r = 12
```

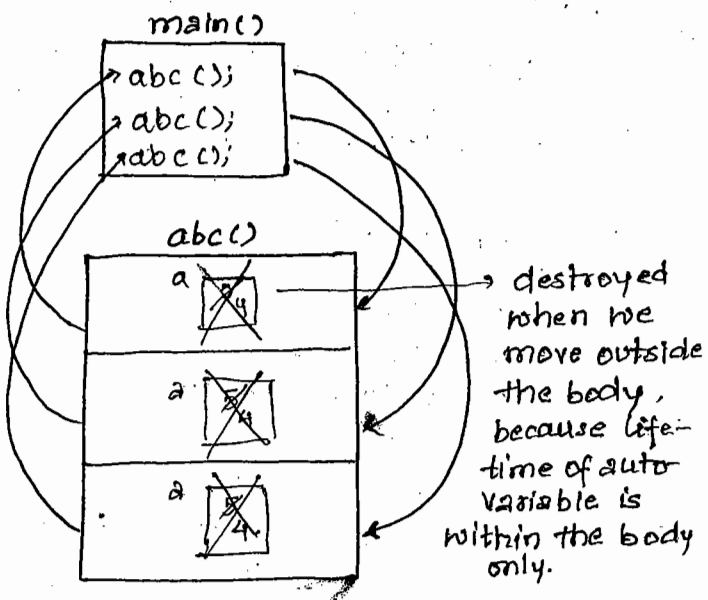
```
register int r = 10;
int *ptr;
ptr = r;          // ptr = &r;

*ptr = 100;
printf("%d", r);
```

Error

WORKING WITH AUTO VARIABLES

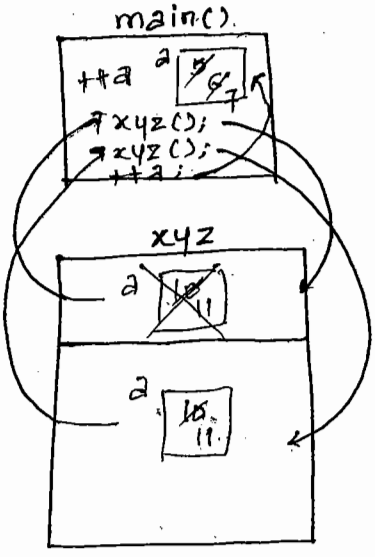
```
void abc()
 {
   int a=5;
    --a;
    printf("%d",a);
 }
void main()
 {
   abc();
   abc();
   abc();
 }
```

O/P :- 4 4 4



⇒ Acc. to storage classes of C, by default any type of variable storage class specifier is auto and lifetime of autovariable is restricted within the body that's why how many times we are calling the function that many times it is constructed.

```
void xyz()
 {
   auto int a = 10;
   ++a;
   printf("\na = %d", a);
 }
void main()
  {
     int a = 5;
     ++a;
     xyz();
     xyz();
     ++a;
     printf("\na = %d",a);
  }
```



O/P: a = 11
a = 11
a = 7

→ void abc()

```
{
  auto int a = 10;
  --a;
  printf("\na = %d", a);
}
void main()
{
  abc();
  abc();
  printf("\na = %d", a);
}
```

O/P:
```
Error
Undefined symbol 'a'
```

- Scope of the autovariable is restricted within the body only that's why abc() related data we can't access in main function.

→ void main()

```
{
  int a = 10;
  clrscr();
  int b = 20;
  ++a;
  ++b;
  printf("\na = %d  b = %d", a, b);
}
```

should be declared here.

O/P:
```
Error
declaration is not
allowed here.
```

- In 'C' prog. lang., variables required to declare on top of the program after opening the body before writing first statement

→ void main()

```
{
  auto int a = 5;
    int a = 10;
  ++a;
  printf("a = %d", a);
}
```

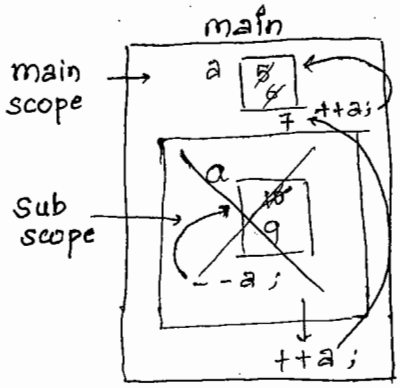
O/P: Error
Multiple declaration for 'a'

⇒ In any type of scope only one variable required to place with any one of unique name, Multiple variables, if we are creating with same name then it gives error

```
void main()
{
  auto int a = 5;
   ++a;
   printf ("\na = %d",a);
   {
      int a =10;
      --a;
      printf (" \na = %d", a);
   }
  ++a;
  printf ("\na = %d ",a);
}
```

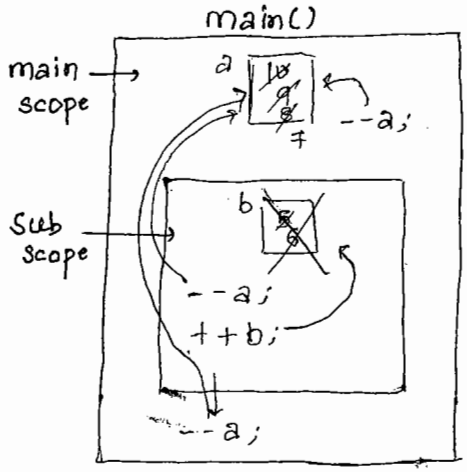
O/P :

```
a = 6
a = 9
a = 7
```



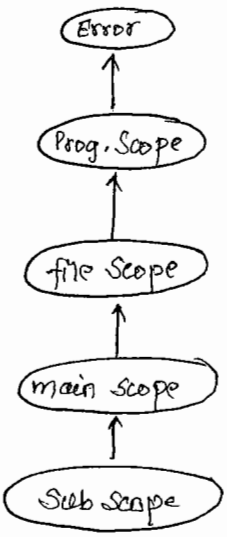
→

```
void main()
{
  int a = 10;
    --a;
   printf ("\na = %d", a);
   {
    auto int b = 5;
    --a;
    ++b;
    printf ("\na = %d  b = %d", a, b);
  }
   --a;
   printf ("\na = %d", a);
}
```



O/P :-

```
a = 9
a = 8    b = 6
a = 7
```



⇒ It is possible to access main scope variable directly in subscope, if subscope doesn't having same variable.

⇒ Acc. to K & R-C standard, previous prog. is error because it is not possible to create variables directly in sub-scope.

⇒ Acc. to NCC standard, it is possible to create variables anywhere within the prog. after opening the body.

→
```
void main()
{
 auto int a=5;
 ++a;
 printf("\na = %d ", a);
{
 int a = 10;
 auto int b = 20;
 ++a;
 ++b;
 printf ("\na = %d b=%d ", a,b);
}
++a;
++b;
printf ("\na = %d b= %d ", a,b);
}
```

O/P: Error undefined symbol 'b'.

It is not possible to access subscope variable in mainscope, if main scope doesn't having same variable.

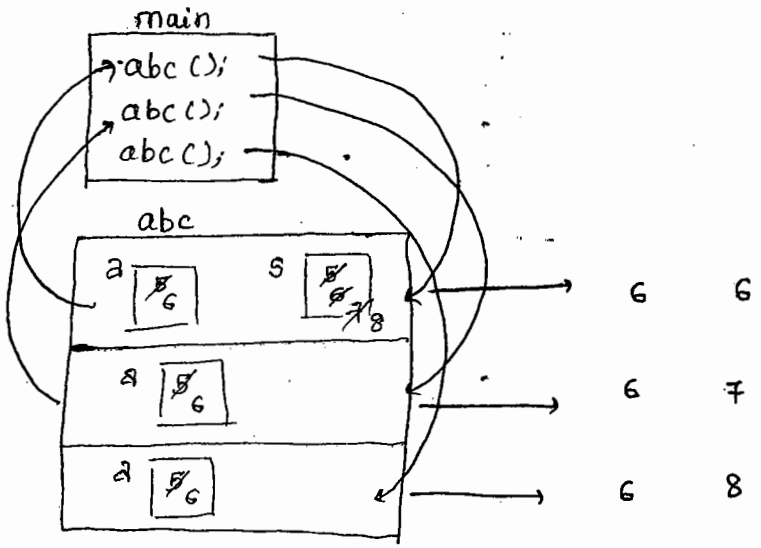
11/7/15

WORKING WITH STATIC VARIABLES

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

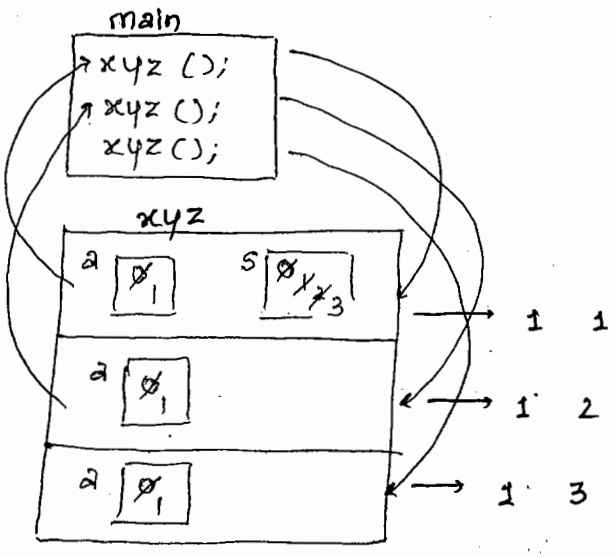
O/P :-

6	6
6	7
6	8



- When we are working with static variable, it is created only once when we are calling the function first time and throughout the program the variable will be there in active mode only

```
void xyz()
{
  auto int a=0;
  static int s;
  ++a;
  ++s;
  printf ("\n%d %d", a, s);
}
void main()
{
  xyz();
  xyz();
  xyz();
}
```



O/P:

1	1
1	2
1	3

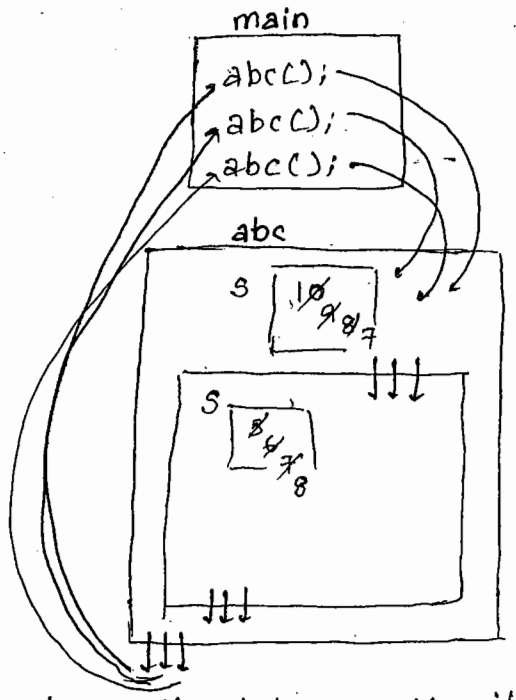
- When we are working with static variable, by default value of static variable is 0 if it is not initialized

→
```
void abc()
{
  static int s= 10;
  --s;
  printf ("\n static 1: %d", s);
    {
      static int s = 5;
      ++s;
      printf ("\n static 2: %d", s);
    }
}
```

O/P:-

```
static c1: 9
static c2: 6
static c1: 8
static c2: 7
static c1: 7
static c2: 8
```

```
void main()
{
 abc();
 abc();
 abc();
}
```



- Static c1 : 9
  Static c2 : 6

  Static c1 : 8
  Static c2 : 7

  Static c1 : 7
  static c2 : 8

- When we are working with static variable, it can be created within function scope and subscope/body/local also.
- When we are creating the static variable within the subscope then it is accessible in subscope only, if we are creating in functionscope then throughout the function it can be accessed.

→
```
void abc()
{
 static int s = 1947;
  ++s;
  printf("\ns = %d", s);
}
void main()
{
 abc();
 abc();
 printf("\nstatic data : %d", s);
}
```

o/p: Error Undefined symbol's:

⇒ When we are working with static variable, scope is restricted within the function only that's why abc function related data, we can't access in main function.

⇒ When we are working with auto variable, scope & lifetime both are restricted within the body only.

⇒ When we are working with static variable, scope is restricted within the function but lifetime is not restricted.

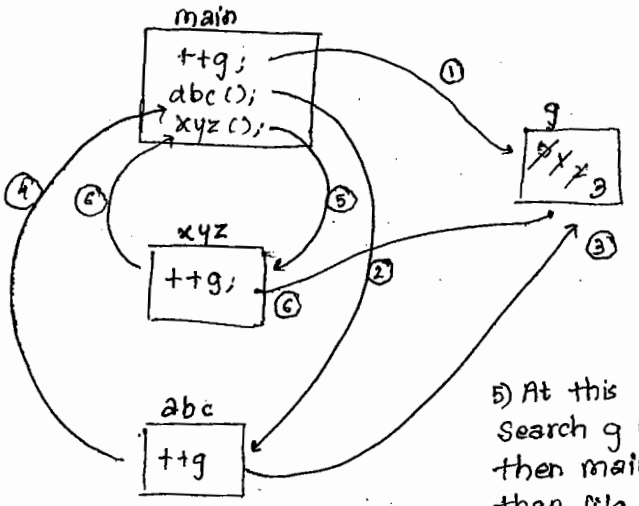
⇒ In implementation, when we required to access a data in more than function then go for extern variables, i.e Global variable is required.

⇒ When we are declaring a variable, outside the function, then it is called Global variable

⇒ When we are working with global variables then scope and lifetime is not restricted.

WORKING WITH GLOBAL VARIABLES :-

→
```
int g;  // global variable
void abc()
{
  ++g;
}
void xyz()
{
  ++g;
}
void main
{
  ++g;
  abc();
  xyz();
  printf("g = %d", g);
}
```



5) At this abc will search g in subscope then main scope then file scope

O/P : g = 3

⇒ By default, any type of variable storage Class Specifier is auto within the body, extern outside the body

⇒ When we are working with global variable, it can be created anywhere within the prog. i.e top or middle or end of the prog. also

→
```
void abc()
{
  ++g;      → this creates error
}
int g;
void xyz()
{
  ++g;
}
void main()
{
  ++g;
  abc();
  xyz();
  printf("g = %d", g);
}
```

O/P :- Error Undefined symbol 'g'

During compilation process, from Top to Bottom [++g] treated as local variable but there is no local and global variable defined. So throws error.

⇒ When we are working with Global variable, it can be created anywhere within the program but if we are accessing before creation then we will get an error.

⇒ In previous program, due to Top-Bottom compilation process, when we are compiling ++g statement in abc() then we will get an error because memory is not created for global variable.

⇒ To avoid the compilation error of global variable, we required to provide forward declaration by using extern Keyword.

Soln -
```
void abc()
{ extern int g;   // declaration
  ++g;
}
int g;
void xyz()        // defining
{ ++g;
}
void main()
{ ++g;
  abc();
  xyz();
  printf("g= %d", g);
}
```
O/P: g=3

NOTE: The basic difference between declaration and defination of Global variable is-

- In declaration of global variable, it doesn't occupies any physical memory, it avoids only compilation error.
- In defination of global variable, actual memory is constructed.

```
void abc()
{
  --g;
}
void xyz()
{
  --g;
}
  void main()
  { abc();
    xyz();
    --g;
    printf("g=%d", g);
  }
  int g=10;
```
O/P: Error, Undefined symbol 'g'.

⇒ In order to access global variable g, in abc(), xyz() and main(), we required to provide forward declaration of global variable in abc(), xyz() and main() also.

⇒ In implementation, when we required to provide declaration of a global variable more than once, we required to go for Global declaration.

⇒ When we are declaring a global variable, Top of the program, before defining first function then it is called Global declaration if the global declaration is available then doesn't required to go for local declaration, if local declaration also available then global declaration is ignored.

Soln :-
```
extern int g;  // global declaration
void abc()
{
  // extern int g; local declaration
  --g;
}
void xyz()
{
  // extern int g; local declaration
  --g;
}
void main()
{
  // extern int g; local declaration
  abc();
  xyz();
  --g;
  printf("g = %d", g);
}
int g = 10;
```

O/P: g = 7

→
```
extern int g;
void abc()
{
  ++g;
}
void main()
{
  ++g;
  abc();
  printf("g = %d", g);
}
```

O/P: Error Undefined Symbol -g (Linking Error)

- When we are working with global variable, if declaration statement is not available, we will get Error at the time of Compilation.
  If definition is not available, then we will get the error at the time of Linking

→
```
extern int g = 10;
void abc()
{
  --g;
}
void main()
{
  --g;
  abc();
  printf("g=%d", g);
}
int g;
```

O/P: g = 8

- Initialisation of the global variable can be placed in declaration and definition also.

→
```
extern int g = 5;
void xyz()
{
  ++g;
}
void main()
{
  ++g;
  xyz();
  printf("g=%d", g);
}
```

O/P: g = 7

- When declaration statement contains initialisation, then physical memory is constructed in declaration only. So doesn't required to go for definition. If definition also available then it will ignore automatically.

→
```
extern int g = 5;
void abc()
{
  ++g;
}
void main()
{
  abc();
  ++g;
  printf("g=%d", g);
}
int g = 10;
```

O/P: Error Variable 'g' is initialised more than once

→ In 'C' programming lang. when local variable and Global variables names are same, then we can't access global varial in local variable In C++, in order to access the variable we required to use scope resolution operator i.e ::

```
extern int g = 5;
void abc()
{
  int g = 420;
   ++g;
  printf ("\ng = %d", g);
}
void main()
 {  ++g
    abc();
    printf ("\ng = %d", g);
 }
```

O/P: 421
g=6

FORMAL ARGUMENTS, ACTUAL ARGUMENTS :-

- In function declarator or in function header, what variables we are creating those are called formal arguments, Parameters
- In function calling statement what data we are passing those are called
- In order to call a function if it is required specific no. of parameters then it is not possible to call the function with less than or more than no. of arguments.
- Where we are implementing the logic of the function it is called function definition.
- In function definition, 1st line is called function header / function declarator
- Where we are executing the logic of a function it is called function calling statement
- When we are providing type information explicitly then it is called function prototype or forward declaration.
- Always function declaration must be required to match with function declarator.

CALLING CONVERSIONS

→ As a programmer, it is our responsibility to indicate in which sequence parameters required to create.

→ Always Calling conversion will decides that in which sequence parameter created i.e left to Right or Right to left.

→ In 'C' prog. lang., calling conversions are classified into 2 types-

1) cdecl (_cdecl)
2) pascal (_pascal)

1) cdecl calling conversion - In this calling conversion, parameters are created towards from right to left

- Bydefault, any function calling conversion is cdecl.
- When we are working with cdecl calling conversion that recommed to place the function name in lower case.

2) pascal calling conversion - In this calling conversion, parameters are created towards from left to right

- In pascal calling conversion, recommeded to place ~~the calling conversion~~ the function name in UPPERCASE
- Generally in database pascal calling conversion is available like orade.

PARAMETER PASSING TECHNIQUES

In 'C' prog. lang. we are having 2 types of parameter passing Techniques-

1) Call by value
2) Call by address

1) Call by Value

- When we are calling a function by passing value type data then it is called Call by Value
- In call by value, actual arguments and formal arguments both are value type variables only.
- In call by value, if any modifications are occur on formal arguments then those modifications doesn't pass to actual arguments.

ex- printf(), pow(), sqrt(), textcolor() etc

2) Call by address

- When we are passing address type data to a function then it is called call by address.
- In call by address, actual arguments are address type and formal arguments are pointer type.
- In call by address, if any modifications occured on formal arguments then those changes will pass to actual arguments.

  eg:- scanf(), strcpy(), strupr() ... etc

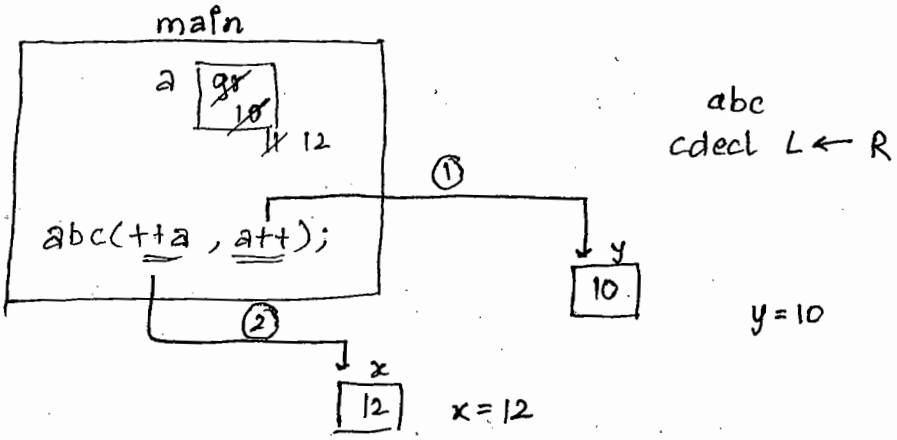
NOTE: • C prog. lang. doesn't supports **call by reference**

- Call by reference is a OOPL concept which is used to access the data by using reference type.

• C prog. lang. doesn't support reference type
- That's why call by reference is not possible.

```
void cdecl abc (int x, int y)
{
 printf ("\n x = %d  y = %d", x, y);
}
void main ()
{
 int a;
 a = 10;
 abc (++a, a++);   - Here 2 arguments need to be passed because
 printf ("\n a = %d", a)        of 2 parameters.
}
```

O/P: x = 12 y = 10
a = 12

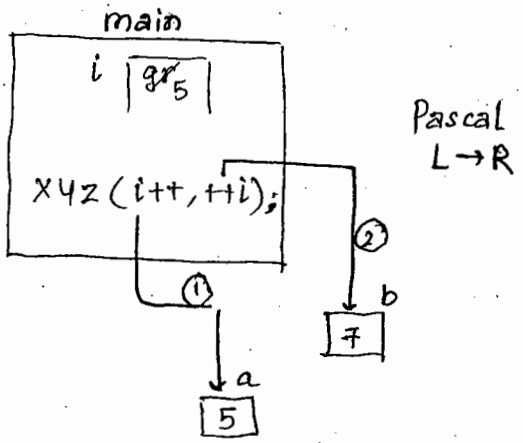


In this above prog.,

If cdecl is not then automatically by default it will take cdecl

→
```
void pascal XYZ (int a, int b)
{
  printf ("\n a = %d  b = %d ", a, b);
}
void main ()
{
  int i;
  i = 5;
  XYZ (i++, ++i);
  printf ("\n i = %d", i);
}
```

O/P:
```
a = 5  b = 7
i = 7
```



→
```
void swap()
{
  int t;
  t = a;
  a = b;
  b = t;
  printf ("\n a = %d  b = %d", a, b);
}
void main()
{
  int a, b;
  a = 10; b = 20;
  swap (a, b);
  printf ("\n a = %d  b = %d ", a+10, b+10);
}
```

O/P: Error undefined symbol 'a', 'b'

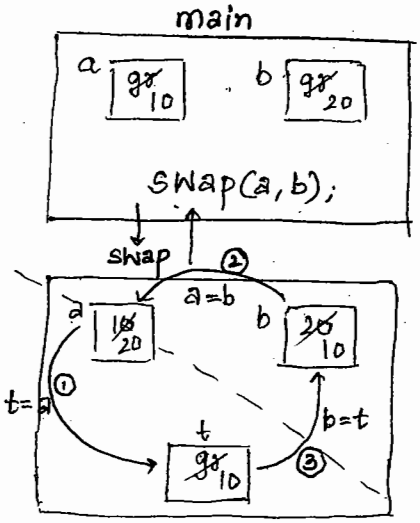
- In order to call the swap function it doesn't required any parameters but we are calling the function by sending 2 arguments
- Acc. to storage classes of 'C', a, b are autovariables, but we are trying to access in swap func^n ∴ Invalid

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

```
    printf("\na = %d  b = %d", a, b);
}
void main()
  { int a, b;
    a = 10;  b = 20;
    swap(a, b);
     printf("\na = %d  b = %d ", a, b);
  }
```

O/P:

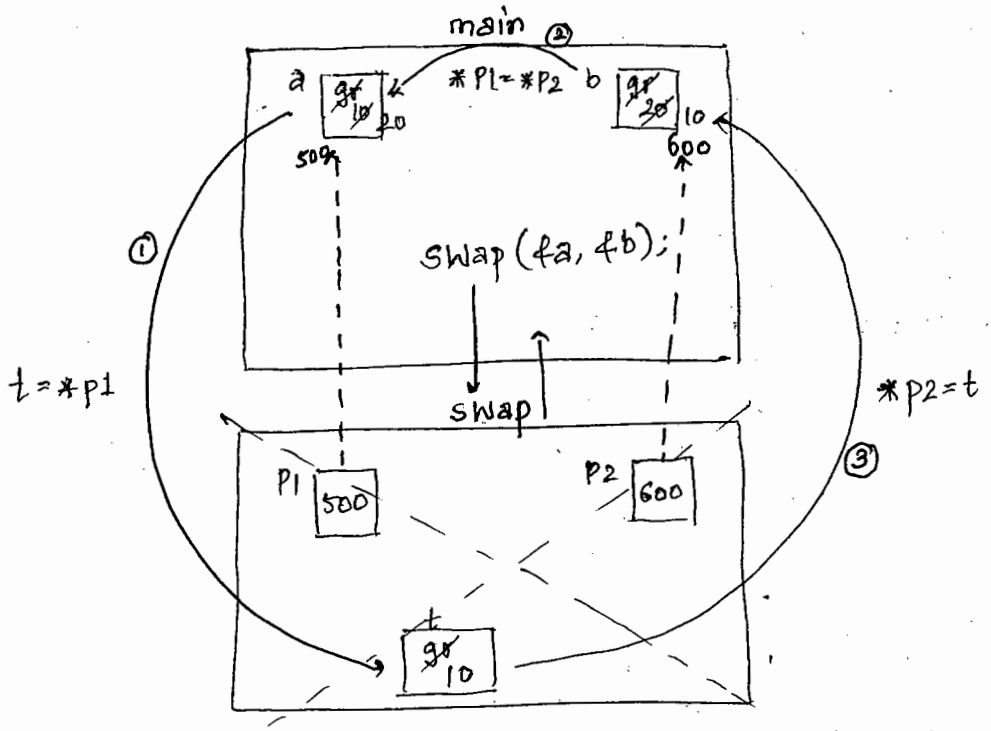
a = 20	b = 10	in swap
a = 10	b = 20	in main



- In previous prog., swap() is working with the help of call by value mechanism. That's why no any modification of swap() is passing back to main.
- In implementation, when we are expecting the modifications, then go for call by address.
- In 'C' prog. lang., call by address can be implemented by using POINTERS only

→ The basic advantage of pointer is accessing the data which is available outside of the function

→
```
void swap(int *p1, int *p2)
  { int t;
    t = *p1;          // t = a
    *p1 = *p2;        // a = b
    *p2 = t;          // b = t
     printf("\n Data in swap a = %d b = %d", *p1, *p2);
  }
void main()
  { int a, b;
    a = 10; b = 20
    swap(&a, &b);
    printf("\n Data in main a = %d  b = %d", a, b);
  }
```



- In previous program, array elements are passing by using call by address mechanism that's why all the modifications of swap() is passing back to main() function.
- The relation b/w formal arguments & parameter is always parameters are initialised with arguments.

14/7/2015.

```
int a=10, b=20;
a=a+b;
a=a-b
a=a-b;
```

O/P: a=20, b=10

a: 10 30 20 | b: 20 10

```
int a=10, b=20,
    c=30;
a=a+b+c;
b=a-(b+c);
c=a-(b+c);
d=a-(b+c);
```

a=30 b=10 c=20

O/P: a: 10 60 30 | b: 20 10 | c: 30 20

```
int a=10, b=20,
    c=30, d=40;
a=a+b+c+d;
b=a-(b+c+d);
c=a-(b+c+d);
d=a-(b+c+d);
a=a-(b+c+d);
```

O/P:- a: 10 100 40 | b: 20 10 | c: 30 20 | d: 40 30

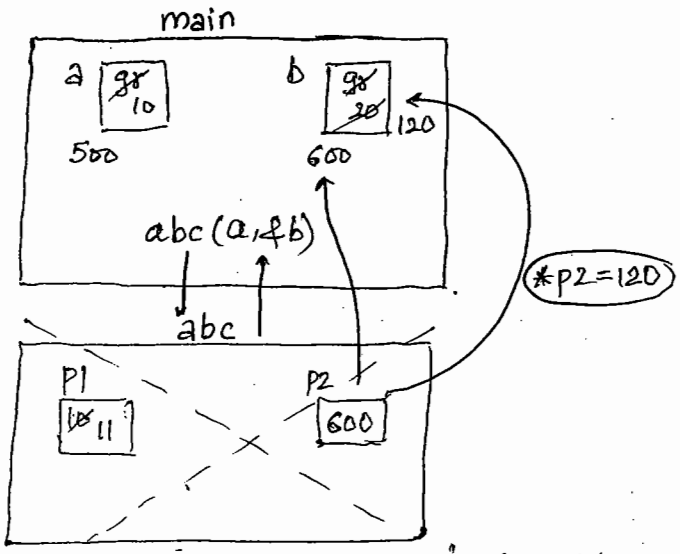
a=40 b=10 c=20
d=30

```
int a=10, b=20;
a=a^b;
b=a^b;
a=a^b
```

O/P:- a=20 b=10

→
```
void abc (int p1, int *p2)
{
 ++p1;
 *p2 = 120;
}
void main()
{
  int a, b;
  a = 10; b = 20;
  abc(a, &b)
  printf("\na = %d  b = %d ", a, b);
}
```

O/P: a = 10 b = 120



→ In a single func^n, it is possible to pass value & address at a time.
→ When we are passing value type data then it is not updated but if we are passing address type data then it is updated.

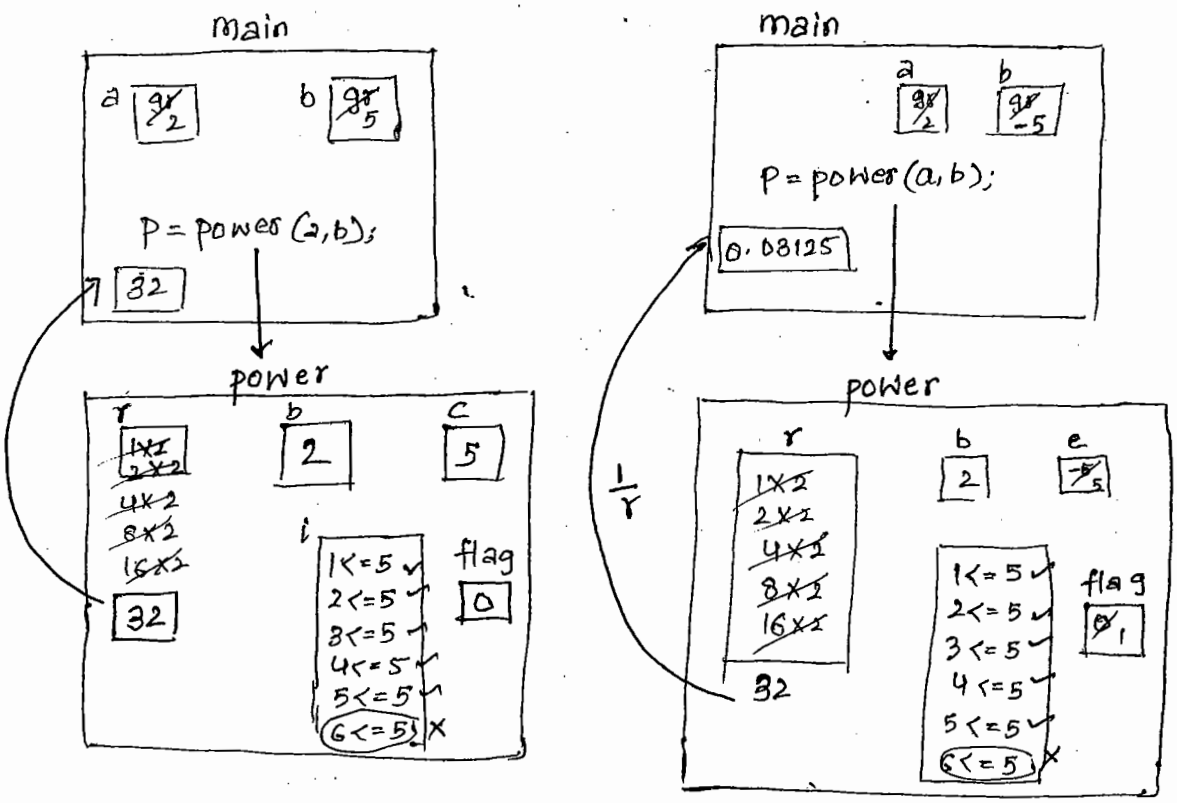
return by value

→ When the function is returning value type data then it is called return by value
→ When the function is not returning any values then specify the return type as void
→ Void means nothing i.e function doesn't return any value.
→ When the function is returning
i.e what type of data it is returning, same type of return statement required to specified
→ In implementation, when the function is returning an integer value, specify the return type as int, i.e function returning value type called return by value.

POWER PROGRAM -

```
* float power (int b, int e)
{
  int i, flag=0;
  float r=1;
  if (e<0)
  {
    flag=1;
    e=e*-1
  }
for (i=1; i<=e; i++)
  r=r*b;
  if (flag==0)
  return r;
else
    return (1/r);
}
```

```
void main()
{
  int a,b;
  float p;
  clrscr();
  printf ("Enter value of a:");
  scanf ("%d", &a);
  printf ("Enter value of b:");
  scanf ("%d", &b);
  p = power (a, b);      //p=pow(a, b);
                          <math.h>
  printf ("\n %d^%d value is %g",
                          a,b,p);
  getch();
}
```



→ return is a Keyword, by using return keyword we can pass the control back to the calling place with arguments or without arguments.

→ return is a exit control of a function which will stop the execution process of a function.

→ In a function it is possible to place any no. of return statements, but at any given point of time, only one return statement can be executed.

→ function can return only one value.

→ By using return statement, it is possible to return only one value.

→ From a function, when we required to return multiple values, then use call by address

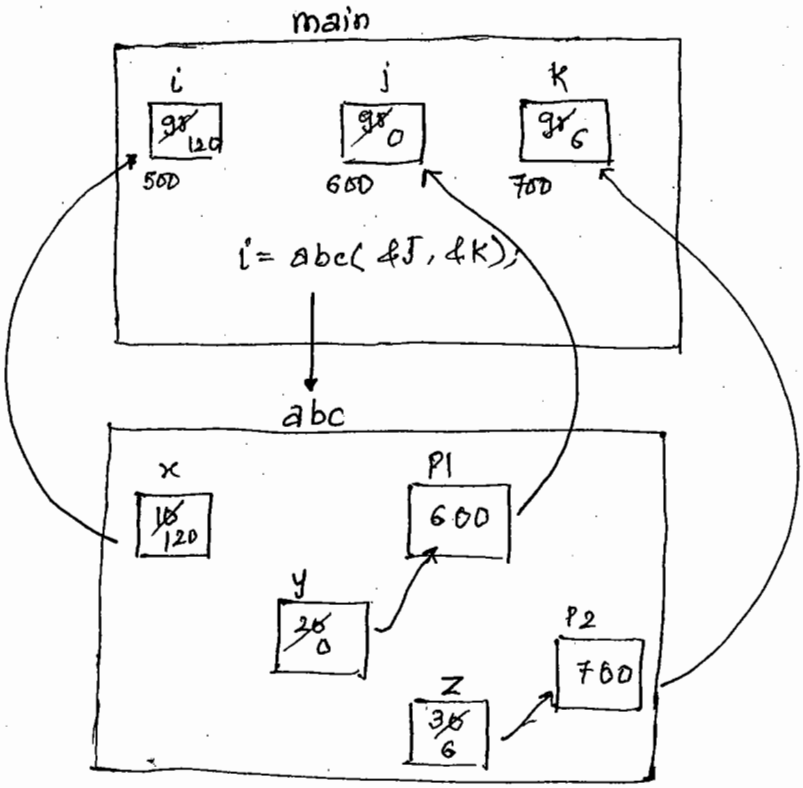
➤ By using call by address, it is not possible to return multiple values but it is possible to pass multiple values by using POINTER.

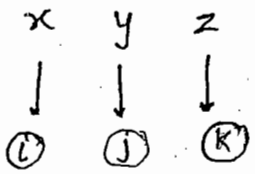
```
→ int max(int x, int y)
{
  if (x>y)
    return x;
  else
    return y;
}
Void main()
{
  int a, b, m;
  clrscr();
  printf("Enter 2 values:");
  scanf("%d %d", &a, &b);
  m = max(a, b);
  if(a!=b)
    printf("Max value is : %d", m);
  else
    printf("Both values are same");
  getch();
}
```

```
O/P:- Enter 2 values : 10 20
      Max value is : 20
```

```
int abc(int *p1, int*p2)
{
  int x=10, y=20, z=30;
  x* = 10+2;  // x = x*(10+2);
  y% = 1+3;  // y = y%(1+3);
  z/ = 2+3;  // z = z/(2+3);
  *p1 = y;  // j = y;
  *p2 = z;  // K = z;
  return x;
}
```

```
Void main()
{
  int i, j, k;
  i = abc(&j, &K);
  printf("i = %d j = %d K = %d", i, j, K);
}
```

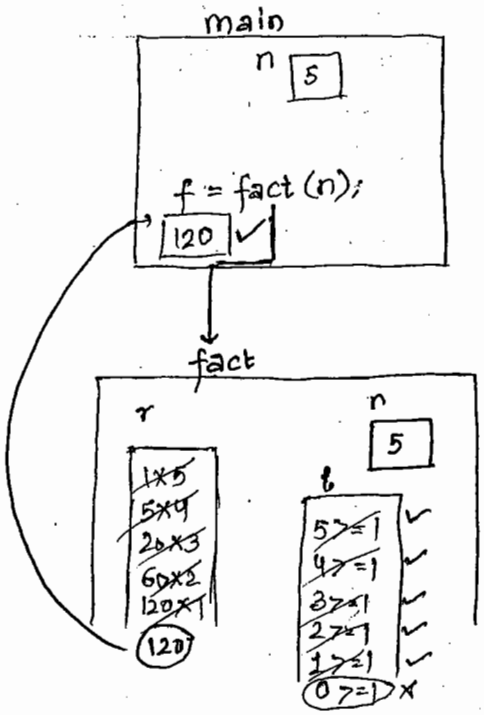




➤ By using POINTERS, we can collect any no. of values as per the requirement of a function

→ FACTORIAL

```
void main()
{  int n,f;
   //int fact(int);
   clrscr();
   printf("Enter a value");
   scanf("%d",&n);
   f= fact(n);
   printf("%d fact value is : %d", n,f);
   getch();
}
int fact (int n)
  {  int r=1, i;
     if (n<0)
      return 0;
     for (i=n; i>=1; i--)
      r = r*i;
      return r;
  }
```



- Acc to K and RC standard, when the return type is integer and parameter type is other than float then doesn't required to go for forward declaration
- As per ANSI-C standard when we are calling a function which is defined later for avoiding the compilation error, we required to go for forward declaration i.e prototype is required.

return by address.

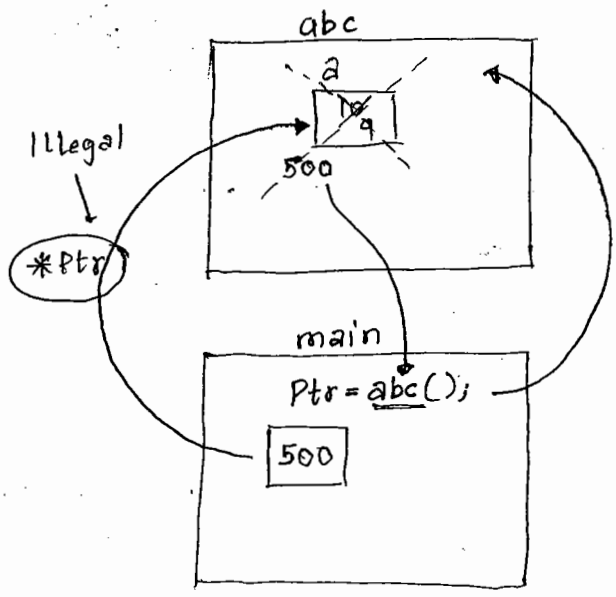
→ When the function is returning address type data then it is called return by address.
→ When the function is not returning any values then specify the return type as void
→ When the function is returning the int value then specify the return type as int i.e function returning value called return by value
→ When the func^n is returning an integer value address then specify the return type as an int * i.e function returning address called return by address.
→ The basic advantage of return by address is 1 func^n related local data can be accessed from outside of the function.

DANGLING POINTER

→ The pointer variable which is pointing to a inactive or dead location called Dangling Pointer

```
→ int *abc()
  {
    int a = 10;
    --a;
    return &a;
  }
  void main()
    {
      int *ptr; //dangling pointer
      ptr = abc();
      printf("value of a = %d", *ptr);
    }
```

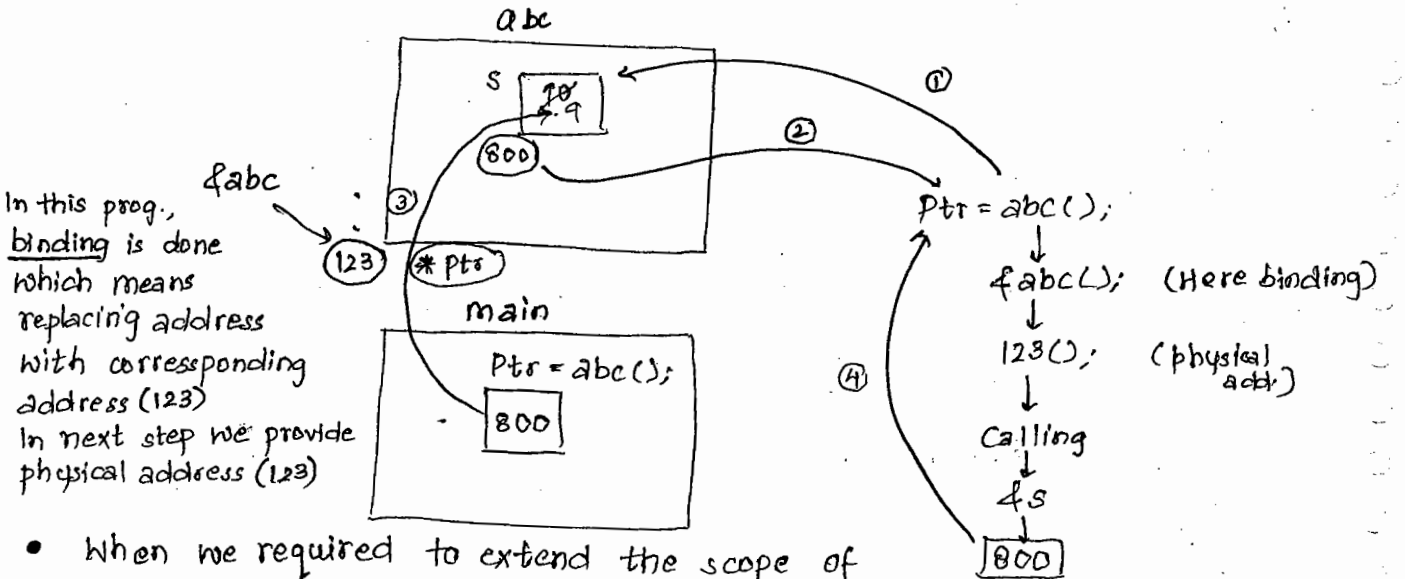
O/P: value of a = 9 (Illegal)



- In previous program, ptr is called Dangling Pointer because acc. to storage classes of C by default any type of variable storage class specifier is void and lifetime of the auto variable is within the body only. But in previous prog. control is passing back to main funcn it is destroyed but still pointer is pointing to that inactive variable only.

- Solution of dangling Pointer is In place of creating auto variable, recommended to create static variable because lifetime of the static variable is entire program..

Prog
```
int *abc() {
    static int s = 10;
    --s;
    return &s;
}
void main()
{
    int *ptr;
    ptr = abc();
    Printf ("static data : %d", *ptr);
}
```
O/P: static data : 9



- When we required to extend the scope of static variable then recommended to go for return by address

FUNCTION POINTER

- The pointer variable which holds the address of a function it is called function Pointer.
- The basic advantage of function pointer is 1 function ~~related~~ can be passed as a parameter to another function.
- Function pointer calls are faster than normal functions.
- Function pointers are similar to deligates in c#.

Syntax :- `Datatype (*ptr)(parameters);`

Ex :-

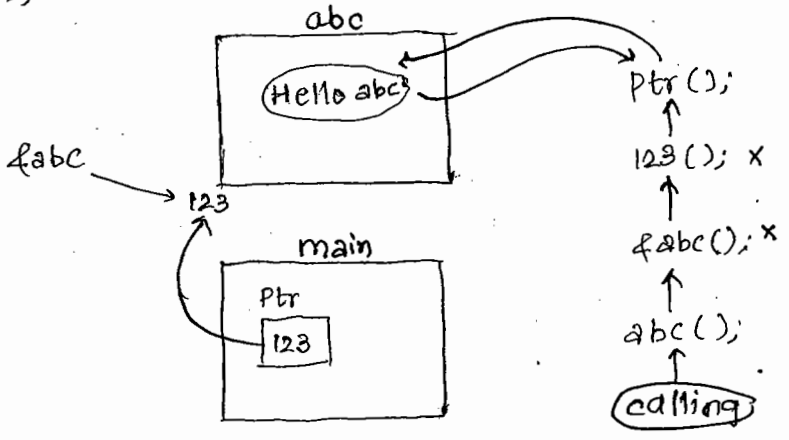
```
void (*ptr)(int);
int (*ptr)(int,int);
int *(*ptr)(int);
```

`Datatype (*ptr)();`

```
void (*ptr)();
int (*ptr)();
int*(*ptr)();
```

- If function takes parameters then go for Parameterized function Pointer.
- If function doesn't takes any parameters than go for non-parameterized function pointer.
- Acc to syntax, function pointer datatype must be required to match with return type of the function.

```
void abc()
{
 printf("Hello abc\n");
}
void main()
{
  void (*ptr)(void)      // Pointer Function
   ptr = &abc;
   ptr();      //abc();
}
```



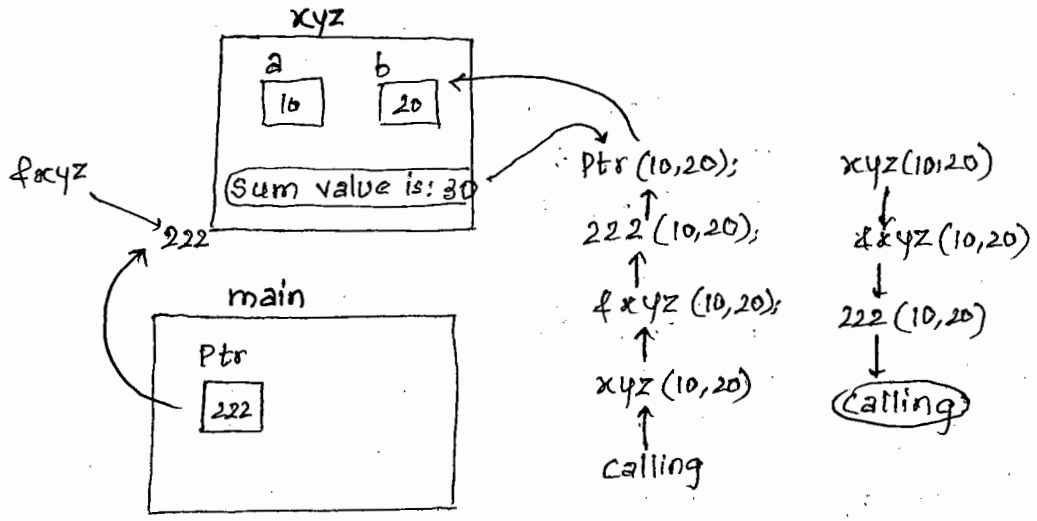
Here in function Pointer binding is not required because Ptr is already pointing to 123.

→ void xyz (int a, int b)

To create function pointer

```
{ printf ("sum value is: %d", a+b);
}
void main()
{ void (*ptr)(int, int);
  ptr = &xyz;
   ptr(10, 20);   // xyz (10, 20)
}
```

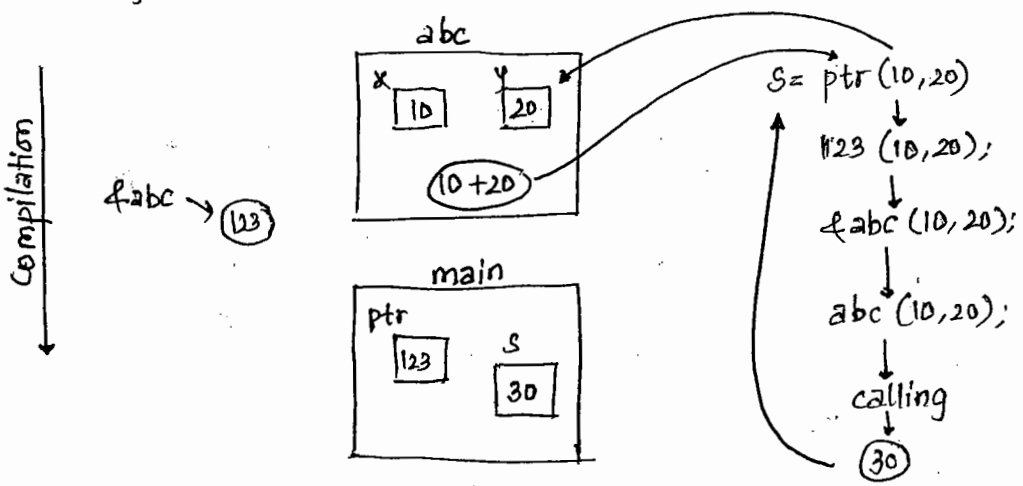
O/P : sum value is : 30



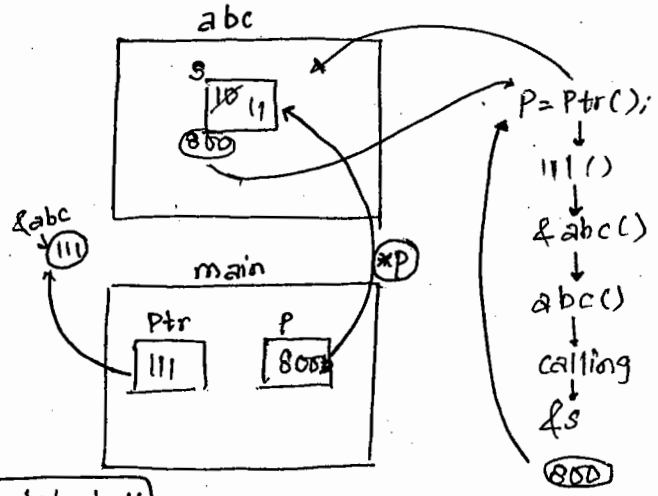
16th July 15

```
int abc(int x, int y)
{
  return (x+y);
}
void main()
{
  int (*ptr)(int, int);
   int s;
   ptr = &abc;
   s = ptr(10,20);    // s = abc(10,20);
   printf("sum value is: %d", s);
}
```

O/P: Sum value is 30

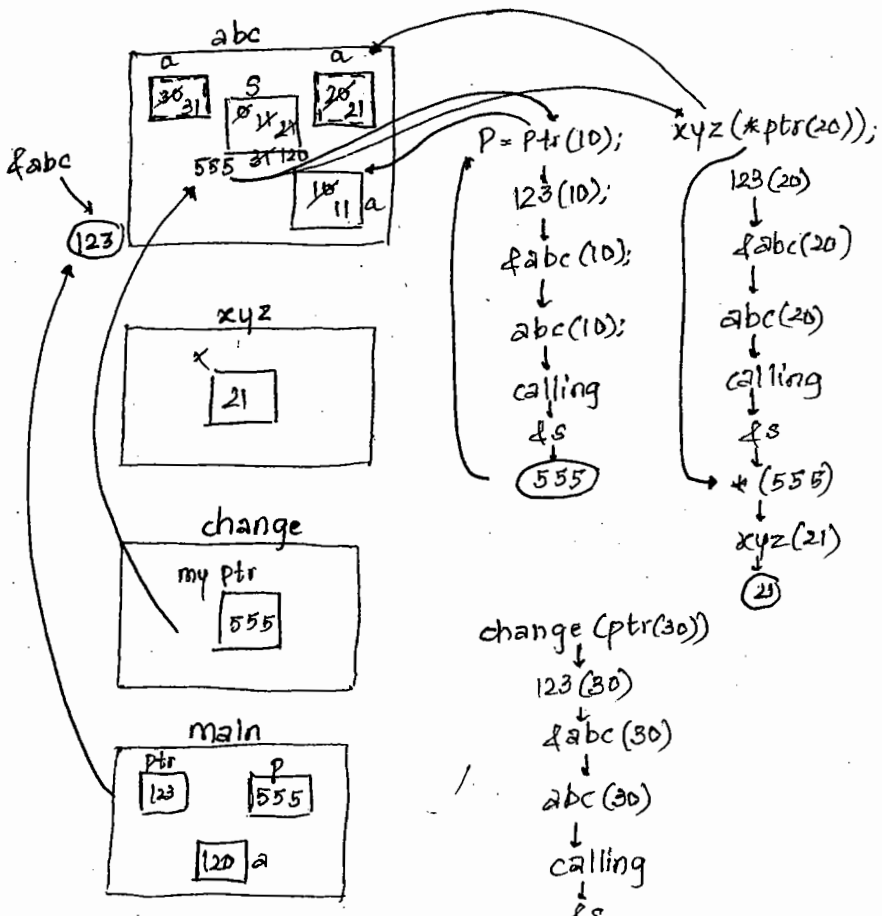


→
```
int *abc()
{
  static int s = 10;
   ++s;
   return &s;
}
void main()
{
  int *(*ptr)(); // pointer to fn
   int *p;      // pointer to integer
   ptr = &abc;
   p = ptr();
  printf("static data is : %d", *p);
}
```



O/P: static data is 11

→
```
int *abc(int a)
{
 static int s;
   s = ++a;
   return &s;
}
void xyz(int x)
 {
  printf("\n static data
          in xyz : %d", x);
 }
void change (int *myptr)
 {
  *myptr = 120;
 }
void main()
  {
   int *(*ptr)(int);
   int *p;
   int a;
   ptr = &abc;
   p = ptr(10);
   xyz(*ptr(20));
   change (ptr(30));
   a = *p;
   printf("static data in
          main m/m : %d", a);
  }
```



a = *p
= *555
a = 120

change (ptr(30))
123(30)
&abc(30)
abc(30)
calling
&s
change [555]
*myptr = 120

∴ a = 120

Recursion :- Function calling itself is called recursion.

→ The function in which control is present, if it calls itself again then it is called recursion process.

Advantages :-

1. By using recursion process only, function calling info. will be maintained in program.
2. By using recursion process stack evaluation takes places.
3. With the help of recursion only, infix, postfix, prefix notation are evaluated.
4. By using recursion process only trees and graphs are implemented.
5. By using recursion process we can develop tree traversing approach i.e inorder, preorder & postorder traversing process.
6. By using recursion only BFS, DFS are developed (Graphs traversing process)

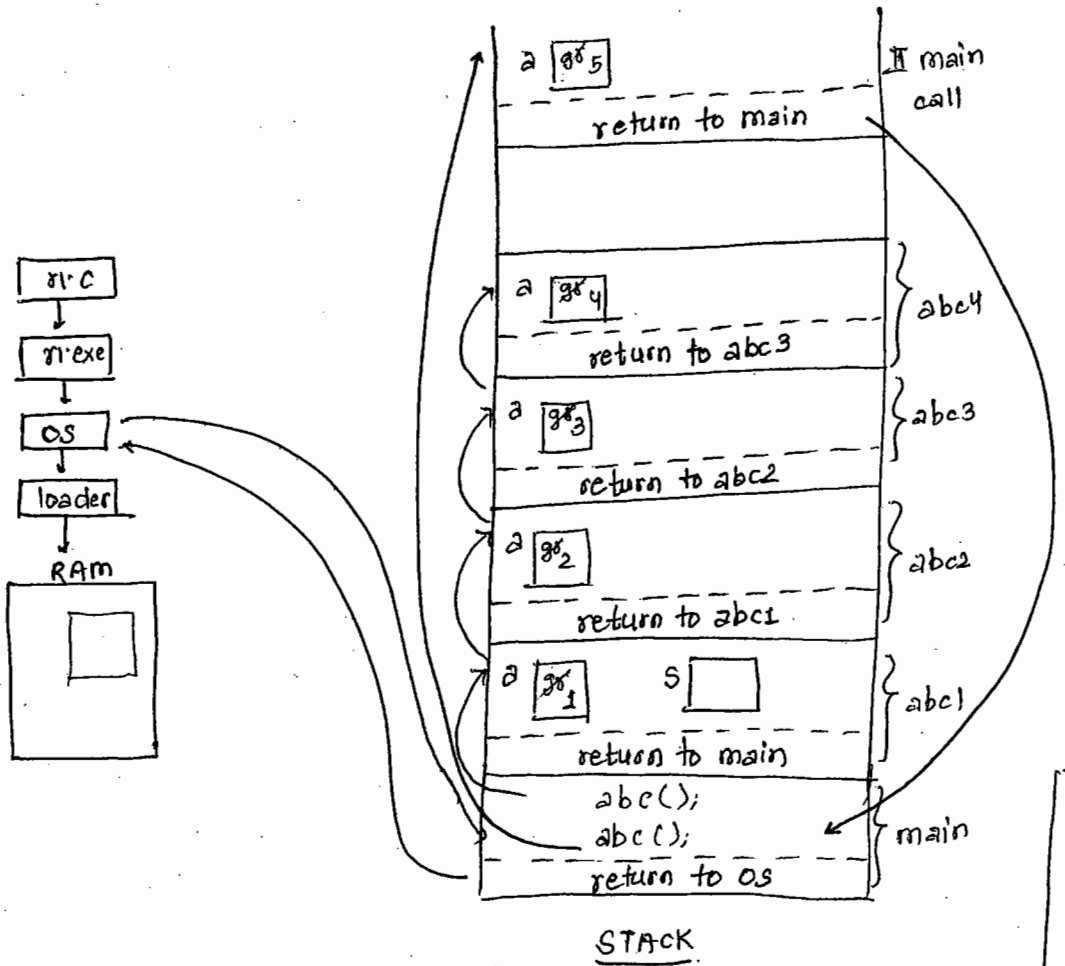
Drawbacks :-

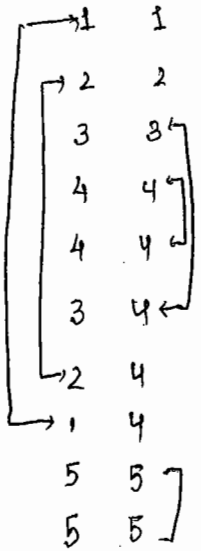
1. It is a very slow process due to stack overlapping
2. Recursion based programs can create stack overflow
3. Recursion based functions can create ∞ loop.

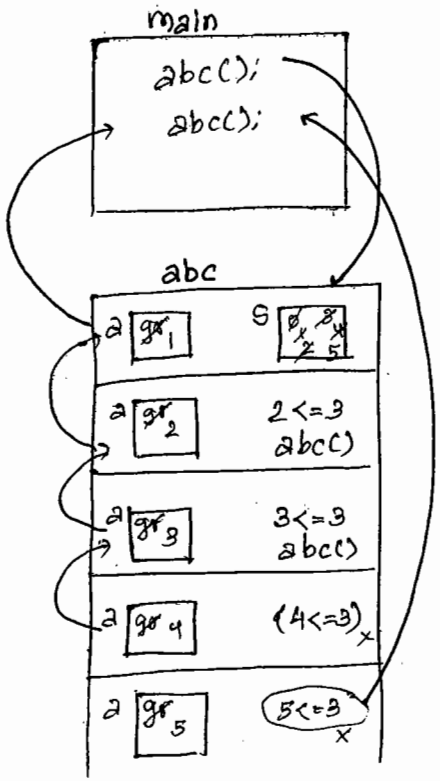
⇒
```
void abc()
{
  int a;
  static int s;
  a = ++s;
  printf("\n%d %d", a, s);
  if (a<=3)
    abc();
  printf("\n %d %d", a, s);
}
void main()
{
  abc();
  abc();
}
```

O/P:

1	1
2	2
3	3
4	4
4	4
3	4
2	4
1	4
5	5
5	5


a
return to main
main call
abc4
return to abc3
abc3
return to abc2
abc2
return to abc1
abc1
s
return to main
abc();
abc();
return to os
main
STACK
a.c
a.exe
OS
loader
RAM


1 1
2 2
3 3
4 4
4 4
3 4
2 4
1 4
5 5
5 5


main
abc();
abc();
abc
s
2<=3
abc()
3<=3
abc()
(4<=3)
5<=3

→ void xyz()

```
{
 int a;
 Static int s = 5;
 a = s++;
 printf ("\n%d %d ", a, s);
 if (a <= 7)
    xyz();
 printf ("\n %d %d ", a, s);
}
Void main()
{
   xyz();
   xyz();
}
```

O/P:

5	6
6	7
7	8
8	9
7	9
6	9
5	9
9	10
9	10

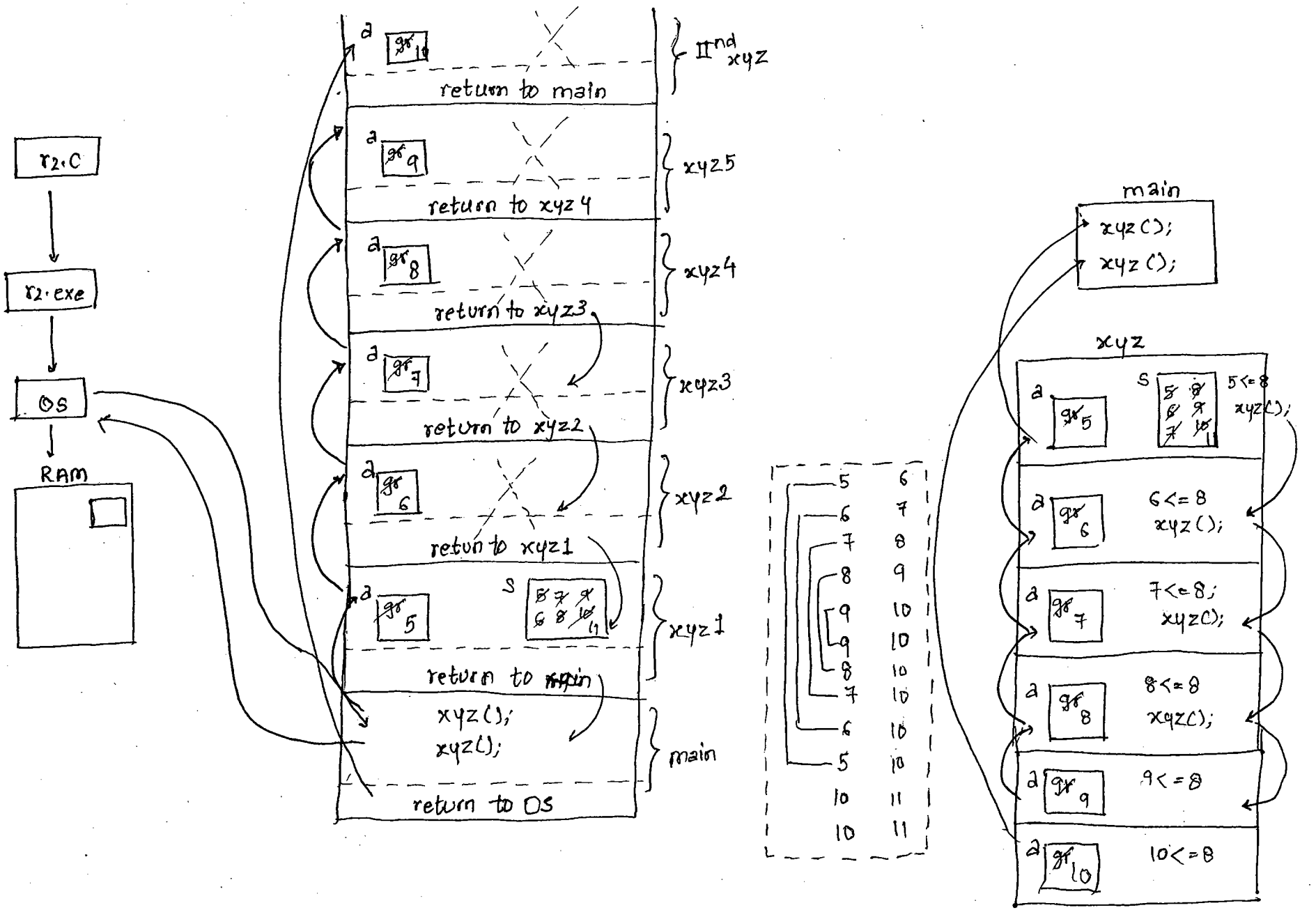
17|7|2015

→ void xyz()

```
{
 auto int a;
 static int s = 5;
 a = s++;
 printf ("\n %d", a, s);
 if (a <= 8)
 xyz();
 printf ("\n %d ", a, s);
}
 void main()
 {
   xyz();
   xyz();
 }
```

O/P:

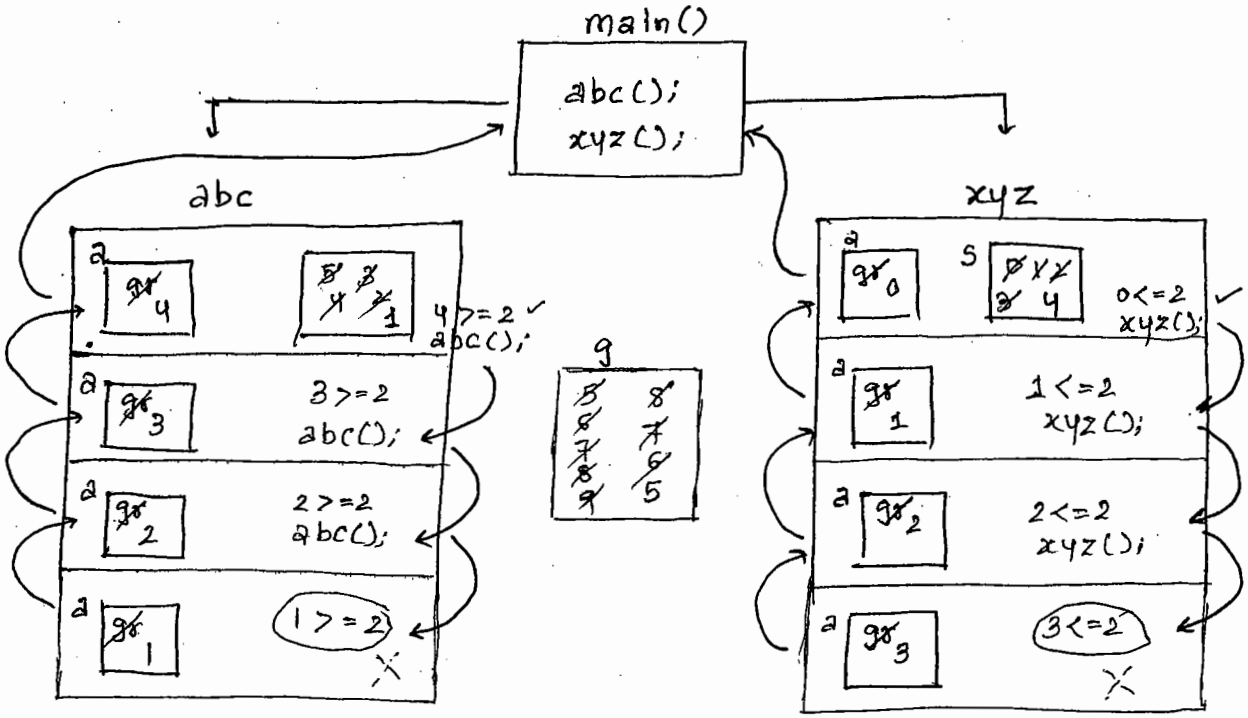
5	6
6	7
7	8
8	9
9	10
9	10
8	10
7	10
6	10
5	10
10	11
10	11


r2.c
r2.exe
OS
RAM
return to main
IInd xyz
xyz5
return to xyz4
xyz4
return to xyz3
xyz3
return to xyz2
xyz2
return to xyz1
xyz1
return to main
xyz();
xyz();
main
return to OS
5 6
6 7
7 8
8 9
9 10
9 10
8 10
7 10
6 10
5 10
10 11
10 11
main
xyz();
xyz();
xyz
5<=8
xyz();
6<=8
xyz();
7<=8;
xyz();
8<=8
xyz();
9<=8
10<=8

```
extern int g;
void abc()
{
 int a;
 static int s = 5
 a = --s;
 ++g;
 printf("\n%d %d %d", a, s, g);
 if (a >= 2)
  abc();
 printf("\n%d %d %d", a, s, g);
}
void main()
{
  void xyz (void);
  abc();
  xyz();
}
Void xyz()
{
  auto int a;
  static int s;
  a = s++;
  --g;
  printf("\n %d %d %d", a, s, g);
  if (a <= 2)
   xyz();
  printf("\n %d %d %d", a, s, g);
 }
 int g=5;
```

O/P:

4	4	6
3	3	7
2	2	8
1	1	9
1	1	9
2	1	9
3	1	9
4	1	9
0	1	8
1	2	7
2	3	6
3	4	5
3	4	5
2	4	5
1	4	5
0	4	5



➤ Recursions are classified into two types -

1) Internal Recursive process
2) External Recursive process

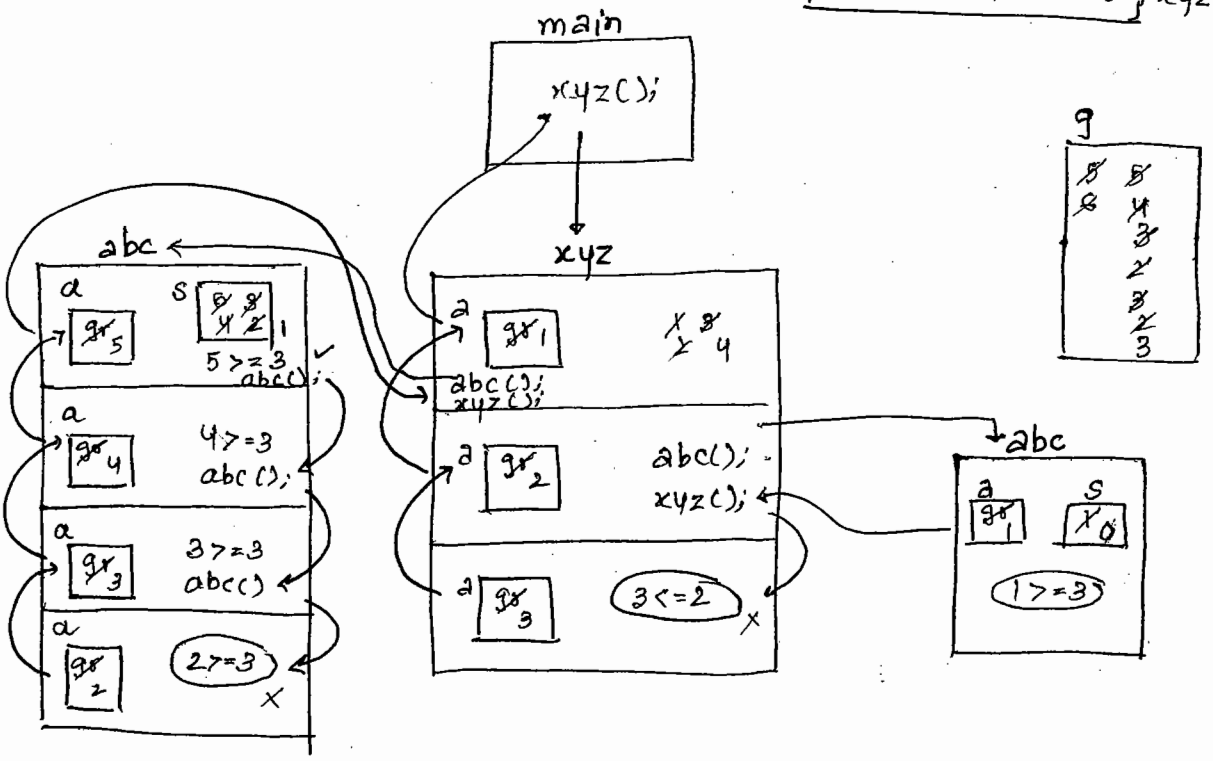
➤ When the function is calling itself then it is called Internal Recursive process. If Recursive function is calling another recursive function then it is called External Recursive process.

→
```
extern int g = 5;
void abc()
{ auto int a;
  static int s = 5;
  a = s--;
  --g;
  printf("\n %d %d", a, s, g);
  if (a >= 3)
    abc();
  printf("\n %d %d %d", a, s, g);
}
void main()
{
  void xyz(void);
  xyz();
}
```

```
void xyz()
{
  int a;
  static int s = 1;
  a = s++;
  ++g;
  printf("\n%d %d %d", a, s, g);
  if (a<=2)
  {
    abc();
    xyz();
  }
  printf("\n%d %d %d", a, s, g);
}
```

O/P:

1	2	6	xyz1
5	4	5	abc1
4	3	4	abc2
3	2	3	abc3
2	1	2	abc4
2	1	2	abc4
3	1	2	abc3
4	1	2	abc2
5	1	2	abc1
2	3	3	xyz2
1	0	2	2nd abc
1	0	2	2nd abc
3	4	3	xyz3
3	4	3	xyz3
2	4	3	xyz2
1	4	3	xyz2



- It is possible to perform recursion of main function also
- By using autovariable if we are performing the recursion of main function then it became stack overflow but for every call, autovariable is reconstructed

→ void main()

```
{
  int a=5;
  --a;
  printf("%d",a);
  if(a>=3)
  main();
  printf("%d",a);
}
```

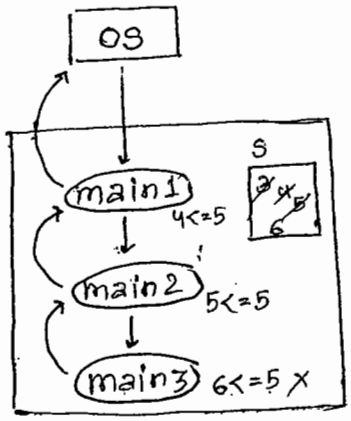
O/P: 4 4 4 4 - - - - stack overflow

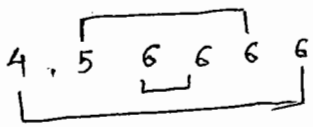
∴ loop will terminate automatically. Stack overflow because auto variable is there.

→ void main()

```
{ static int s=3;
  ++s;
  printf("%d",s);
  if(s<=5)
   main();
   printf("%d",s);
}
```

O/P: 4 5 6 6 6 6





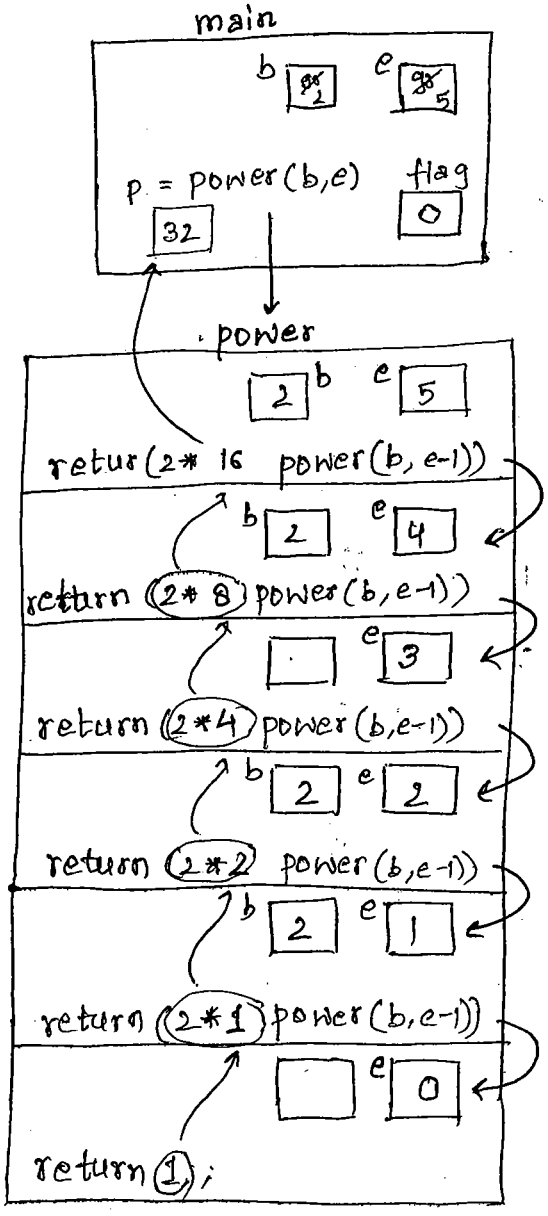
POWER PROGRAM (Recursion Approach)

```
float power(intb, inte)
{ if(e==0)
   return 1;
 else
   return (b*power(b,e-1));
}
void main()
{ int b,e,flag=0;
  float p;
  clrscr();
  printf("Enter value of b:");
```

```
scanf ("%d", &b);
printf ("Enter value of e : ");
scanf ("%d", &e);
if (e<0)
{
 flag = 1
 e = e*-1;
}
p = power (b,e);
 if (flag==0)
   printf ("\n %d ^ %d value is : %g", b, e, p);
 else
   printf ("\n %d ^ %d value is : %g", b, -e, 1/p);
 getch();
}
```

O/P:

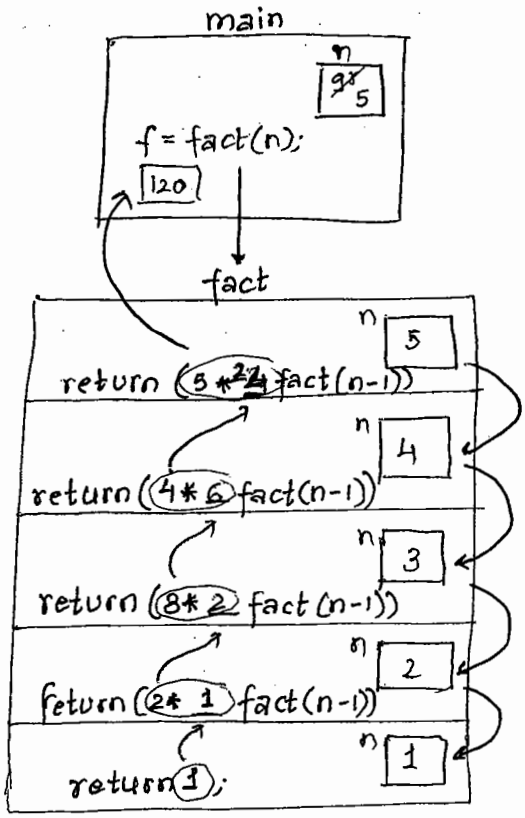
```
Enter value of b : 2
Enter value of e : 5
 32
```



FACTORIAL PROGRAM (Recursive Approach)

```
void main()
{ int n,f;
  int fact(int);
  clrscr();
  printf("Enter a value:");
  scanf("%d",&n);
  f=fact(n);
  printf("%d fact value is : %d", n,f);
  getch();
}
int fact(int n)
{
  if(n<0)
    return 0;
  else if(n<=1)
    return 1;
  else
    return (n*fact(n-1));
}
```

O/P:
Enter a value : 5
fact value : 120



- When the function is not returning any value then specify the return type as void
- Void means nothing i.e no return value

→
```
void abc()
{ printf("Hello abc\n");
}
void main()
{
  abc();
}
```

O/P: Hello abc

- If function return type is void then it is possible to place return statement also.
- From void function when we are placing empty return statement, then control will pass back to the calling place automatically without any values.

→ void abc()
```
{
  printf("Welcome abc\n");
  return ;
}
void main()
{
  abc();
}
```
O/P: Welcome abc

- If the function return type is void, then it is possible to place return statement with value also
- from void function, when we are returning the value, then compiler will gives a warning message i.e void function may not return any values

---

→ void abc()
```
{
  int a = 10;
  ++a;
  printf("Hello abc : %d", ++a);
  return a;
}
void main()
{
  abc();
}
```
O/P: Hello abc : 12

➤ From void function, when we are trying to collect the value it gives an error i.e not an allowed type.

→ void abc()
```
{
  int a = 10;
  ++a;
  return a;
}
void main()
{
  int x;
  x = abc();
  printf("x = %d", x);
}
```
O/P: Error

→ void sum(int x, int y)
```
{
  printf("Sum value is : %d", x+y);
}
void main()
{
  Sum(10,20);
}
```
O/P :- Sum value is : 30

→ int sum (int x, int y);
```
{ printf ("sum value is : %d", x+y);
}
void main ()
{
  sum (10,20);
}
```
O/P: sum value is : 30

→ When the function is returning the value then specifying the return statement is always optional.
In this case, compiler will give a warning message :- function should return a value.

→ int sum (int x, int y)
```
{ printf ("sum value = %d", x+y);
  return x+y;
}
void main ()
{
  sum (10,20);
}
```
O/P: sum value = 30

- When the function is returning the value then collecting the value is always optional.
In this case, compiler doesn't give any warning or error messages.

→ int sum (int x, int y)
```
{
  return (x+y);
}
void main ()
{ int s;
  s = sum (10,20);
  printf ("sum value is : %d", s);
}
```
Sum value is : 30

- When we are returning simple format data, then it doesn't require to specify return statement within the parathesis, when we are returning expression format data, then it is recommended to specify the return statement within paranthesis.
bcz if paranthesis is not there then it inc. the burden on compiler

return x ; } 100% same
return (x);

return x+y; → y, +, x } 6B

return (x+y);
↳ Sum → 2B

→
```
Float sum (int x, int y)
{
  return (x+y);
}
void main()
{
  int s;
  s = sum (12.8, 30);
  printf ("sum value is : %d", s);
}
```

O/P: sum value is : 42

int x, int y
12 30 12+30 = 42

- When the return type and parameter types are not matching at the time of execution then automatically type conversion will happen at the time of execution.

→
```
auto int g = 10;
void main ()
{
  ++g ;
  printf ("g= %d", g);
}
```

O/P: Error
Storage class 'auto' is not allowed here.

- By default any type of variable, storage class specifier is auto within the body, extern outside of the body

  In Global scope, it is not possible to place auto storage class specifier.

→
```
void abc (auto int a)
{
  ++a;
  printf ("a= %d", a);
}
void main ()
{
  abc (10);
}
```

O/P: Error
storage class 'auto' is not allowed here.

- For any type of parameters of a function, we can't specify storage class specifier because function m/m is temporary.
- for parameters, it is possible to specify register storage class specifier only.

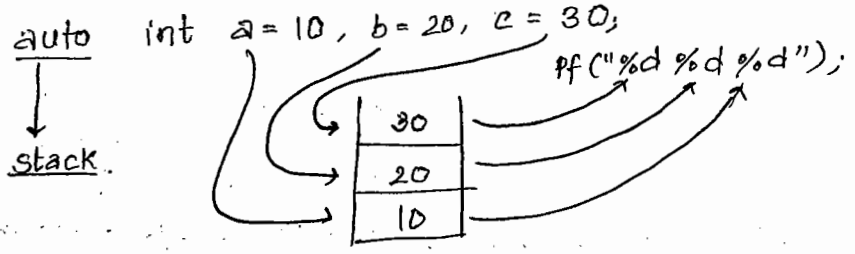
→
```
auto void abc
{
  printf ("Hello abc\n")
}
void main ()
{
  abc ();
}
```

O/P: Error
storage class 'auto' is not allowed here.

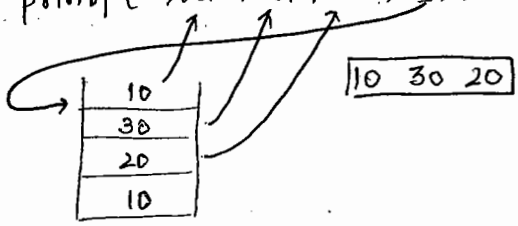
- By default scope of any function is extern i.e from anywhere in project we can call any function.
- By using auto storage class specifier, we can't restrict the scope of a function because it is self contained, independent block.
- for a function we can't apply auto, register storage class specifier.

→ void main()
{ int a=10, b=20, c=30
printf("%d %d %d");
}

O/P: 30 20 10

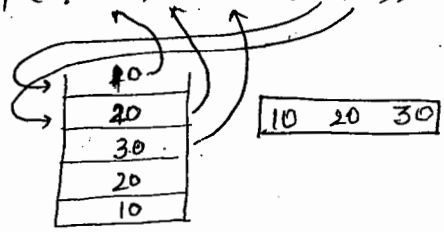


1. printf("%d %d %d", a);



10 30 20

2. printf("%d %d %d", a, b);



10 20 30

→ void main()
{ auto static int a=10;
++a;
printf("a=%d", a);
}

Error:
Too many storage classes in declaration.

- for any type of variable, only 1 storage class specifier is possible to be placed.

→ void abc()
{ auto int a=10;
static int s=a;
++a;
++s;
Printf("a=%d", a);
}
void main()
{ abc();
}

O/P: Error Illegal initialization

- When we are working with static variables, always required to initialised with constant value only.

→
```
static int g = 5;
void abc()
{
 static int s = 10;
++g;
 ++s;
 printf ("\n s = %d  g = %d", s, g);
}
void main()
{
 ++g;
 abc();
}
```

O/P: s = 11 g = 7

- When we are declaring a static variable within the function then it is called internal static declaration.
- When we are declaring a static variable outside the function then it is called external static declaration.

➤ Basic difference b/w internal & external static declaration :

Scope.

The scope of internal static declaration is restricted within the func^n.

The scope of external static declaration is restricted within the file.

CASE 1:

project global variable

1.c

```
extern int g = 10;
void abc();
{
 ++g; ✓
}
void xyz();
 {
 ++g; ✓
 }
```

main.c

```
void main()
{
  ++g; ✓
}
```

CASE 2:

file scope global variable

2.c

```
static int g = 10;
void abc()
{
 ++g; ✓
}
void xyz()
{
  ++g; ✓
}
```

main.c

```
void main()
{
  ++g;
}
```

Error

- "C" programming lang. doesn't supports static functions
- Static function is a OOPL concept which is used to access static member of a class.
- In "C" programming lang., static functions are not possible but we can place static storage class specifier to a func^n.
- When we are placing static storage class specifier to a function then scope of the function is limited to specific file only

CASE1:

default extern scope

1.c

```
Void abc()
{
}
Void xyz()
{
}
```

main.c

```
void main()
{
  abc(); ✓
  xyz(); ✓
}
```

CASE2:

1.c

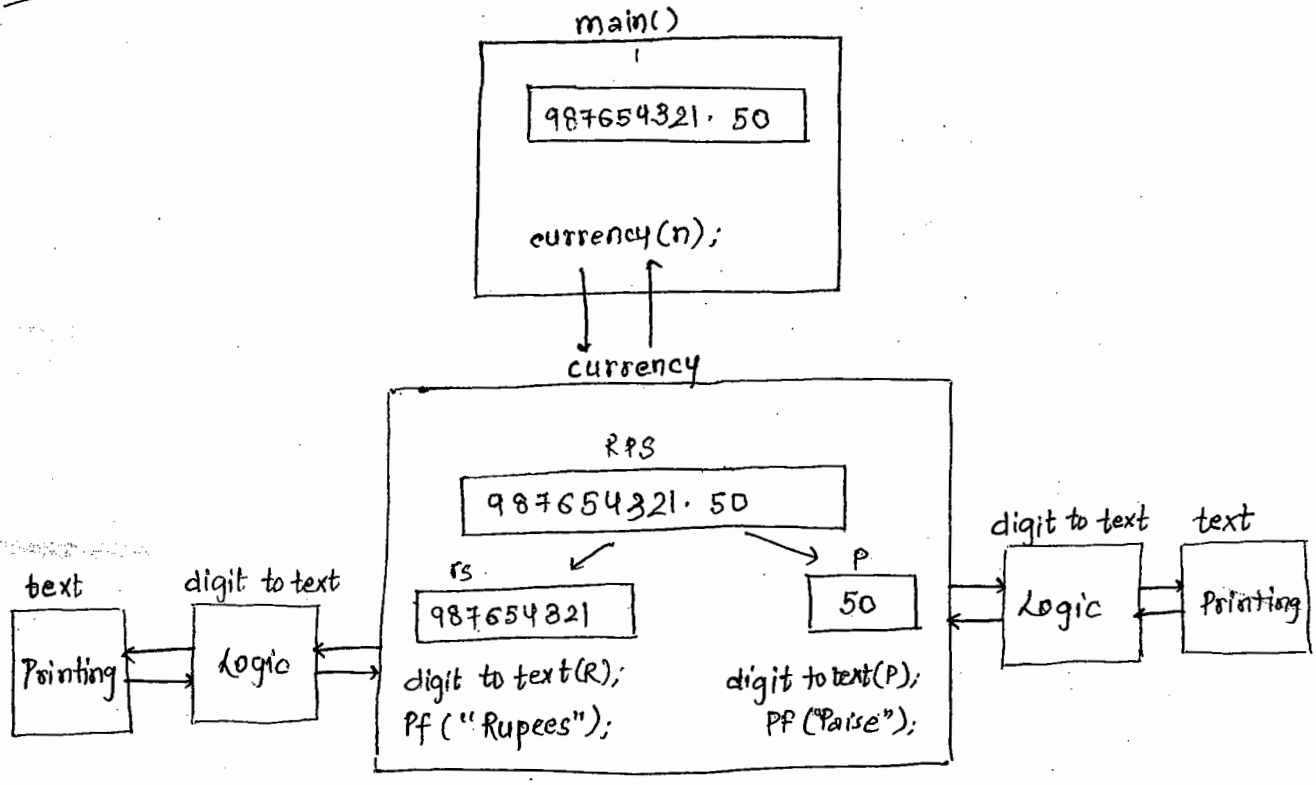
```
static void abc()
{
}
void xyz()
{
  abc(); ✓
}
```

main.c

```
void main()
{
  xyz(); ✓
  abc();
}
```

Error

21/7/2015.




```
void text (long int no)
{
 switch (no)
   {
    case 1: printf ("ONE");
            break;
    case 2: printf ("TWO");
            break;
    case 3: printf ("THREE");
            break;
    case 4: printf ("FOUR");
            break;
    case 5: printf ("FIVE");
            break;
```

```
case 6 : printf("SIX");
         break;
case 7 : printf("SEVEN");
         break;
case 8 : printf("EIGHT");
         break;
case 9 : printf("NINE");
         break;
case 10 : printf("TEN");
          break;
case 11 : printf("ELEVEN");
          break;
case 12 : printf("TWELVE");
          break;
case 13 : printf("THIRTEEN");
          break;
case 15 : printf("FOURTEEN");
          break;
case 16 : printf("SIXTEEN");
          break;
case 17 : printf("SEVENTEEN");
          break;
case 18 : printf("EIGHTEEN");
          break;
case 19 : printf("NINETEEN");
          break;
case 20 : printf("TWENTY");
          break;
case 30 : printf("FOURTEEN"); ("THIRTY");
          break;
case 40 : printf("FORTY");
          break;
case 50 : printf("FIFTY");
          break;
case 60 : printf("SIXTY");
          break;
case 70 : printf("SEVENTY");
          break;
case 80 : printf("EIGHTY");
          break;
```

```
        case 90 : printf ("NINETY");
                  break;
        case 100: printf ("HUNDRED");
                  break;
        case 1000: printf ("THOUSAND");
                   break;
        case 100000: printf ("LAKH");
                     break;
        case 10000000: printf ("CRORE");
                       break;
    }  case 32: // End of switch
} // End of text
Void digittotext ( long int n)
{
  long int t ;
    if (n == 0)
      printf ("ZERO");
    if (n >= 10000000)
     {
       t = n/10000000;
        if (t<= 20)
          text(t);
        else
          { text (t/10*10);
            text (t%10);
          }
        text (10000000);
         n = n % 10000000;
     }
     if (n>= 100000)
       {
         t = n/100000;
          if ( t<= 20)
              text (t);
     else
     {
        text ( t/10*10);
        text (t %10);
     }
      text (100000);
       n = n % 100000;
}
```

```
if (n >= 1000)
{
  t = n/1000;
   if (t <= 20)
     text (t);
  else
    {
     text (t/10* 10);
     text (t%10);
    }
   text (1000);
    n = n % 1000;
 }
if (n>= 100)
 {
   t= n/100;
    text(t);
    text (100);
    n = n%100;
  }
  if (n<= 20)
    text (n);
  else
    {
     text (n/10 *10);
     text ( t %10);
    }
 }
 Void currency (double rps)
{
   long int r, p;
   r = (long int) rps;
   digittotxt (r);
   printf ("RUPEES");
   p (rps-r)* 100;
   digittotext (p);
   printf ("PAISE");
 }
 void main ()
 {
    double n ;
```

```
    clrscr();
    printf("Enter Amount:");
    scanf("%lf",&n);
    currency(n);
    getch();
}
```

Calling & Declaration

```
int i1, i2;
int *ip1, *ip2;
int **ipp;
```

Calling

```
abc();
abc(i1, i2);
i1 = abc();
xyz(&i1, &i2);
abc(&i1, i2);
ip1 = abc(&i1, &i2);
swap(&i1, &ip1);
ipp = swap(&ip1, &ip2);
```

Declaration

```
void abc(void);
void abc(int, int);
int abc(void);
void xyz(int*, int*);
void abc(int*, int);
int *abc(int*, int*);
void swap(int*, int**);
int **swap(int**, int**);
```

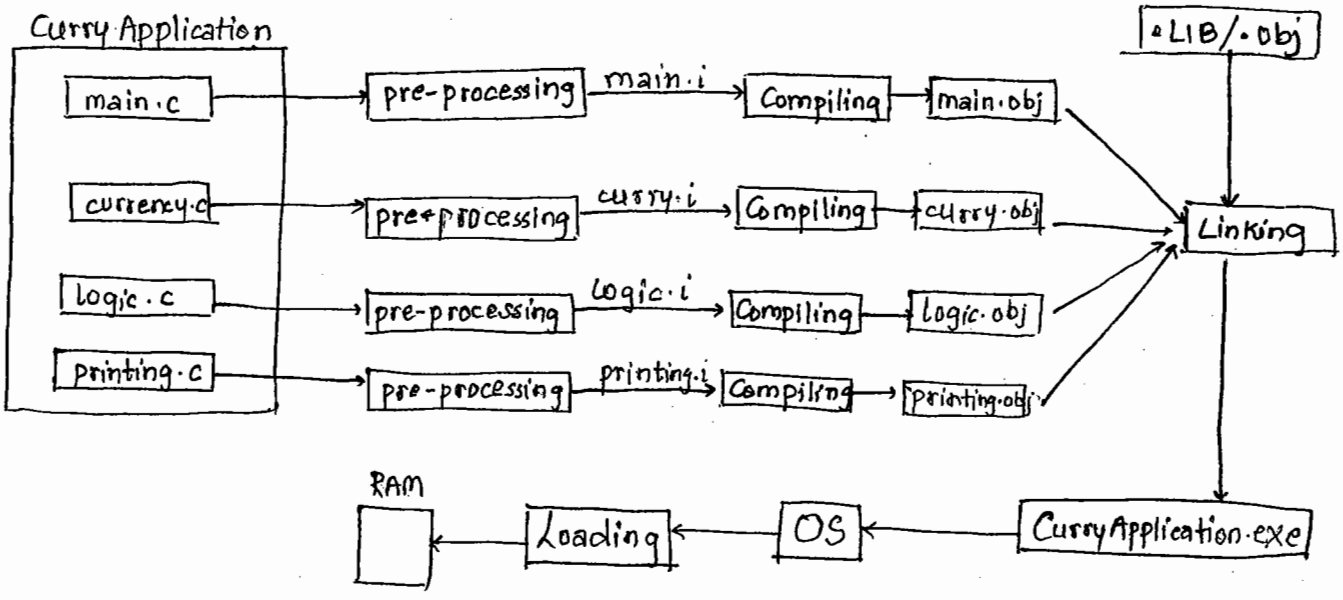
22/7/2015.

PRE-PROCESSING

Pre-Processing

- Pre-Processing is a program which will be executed automatically b4 passing the source program to compiler.
- Pre-Processing is under control of Pre Processor directives.
- All pre-processor directives start with pound (#) symbol and should be not ended with semicolon (;)
- When we are working with Pre-Processor directives, it can be placed anywhere within the program but recommend to place on top of the prog. before defining first function
- In 'C' Programming lang. Preprocessor directives are classified into 4 types -

1. Macro substitution Directives
   Ex: # define
2. File Inclusion Directives
   Ex: # include
3. Conditional compilation Directives
   Ex: # if, # else, # endif, # elif
   # ifdef, # ifndef, # undef
4. Miscellaneous Directives
   Ex: # pragma, # error, # line



➤ When we are working with any kind of 'C' Application we are required to perform 4 steps

1) Editing
2) Compiling
3) Linking
4) Loading

1) Editing is a process of constructing the source program and saving with .c extension

→ To perform the editing process, we are required any kind of text editors like notepad, wordpad or any other C language related IDE.

2) Compiling is a process of converting high level programming lang. data in the machine readable data i.e object code or compiled code.

→ To perform compilation process, must be required, 'C' prog. language related IDE's only

3) Linking is a process of combining all obj files of current project along with standard Lib or obj files to construct an executable file

~~To construct an ex~~

→ When the linking process is successful then automatically executable file is generated with .exe extension

4) Loading is a process of carrying the application file from secondary storage area to primary memory.

→ Editing, Compiling and Linking is under control of IDE and loading is under control of OS

When we are working with any C application, it creates 5 types of files i.e

.C	.BaK	.exe
.I	.obj	

.c, .I, and .BaK contains user readable format data i.e source format

Generally .I file contains extended source code which is constructed after pre-processing.

.obj file contains compiled code which can be understandable to system only.

.exe file contains native code of OS.

1) MACRO SUBSTITUTION (#define)

- When we are working with # define at the time of pre-processing where an identifier is occurred, which is replaced with replacement text.
- Replacement text can be constructed using single or multiple tokens
- A Token can be Keyword, operator, seperator, constant or any other identifier

Syntax: `#define identifier replacement text`

- Acc. to syntax, atleast single space must be required between #define, identifier and identifier, replacement text.
- When we are working with #define, it can be placed anywhere in the program but recommended to place on top of the program before defining first function.
- By using #define, we can create symbolic constants which decreases the burden on programmer when we are working with array.

DESIGNING A C PROGRAM WITH DOS COMMANDS

→ For editing the program, we required to use edit command
edit is a internal command which is available along with OS.

Syntax: `edit filename.c`

Ex: D:\C1100AM>edit p1.C

`code in p1.c`

```
# define A 15
 void main()
  {
    int x;
    x=A;
    printf("%d %d", x, A);
  }
// save p1.c (File ⟶ save)
// close p1.c (File ⟶ exit)
```

→ To perform the preprocessing we required to use cpp command.
cpp is an external command which is available in C:\TC\BIN directory

Syntax: `CPP filename.c`

D:\C1100AM>CPP P1.C

NOTE: Preprocessing is a automated program which will be executed automatically before passing the source code to compiler.

→ If we required to create .i file explicitly then mandatory to perform.

code in p1.i

```
p1.c  1 :
p1.c  2 : void main()
p1.c  3 : {
p1.c  4 : int x;
p1.c  5 : x=15;
p1.c  6 : printf ("%d %d", x,15);
p1.c  7 : }
p1.c  8 :
```

→ As per above observation, at the time of preprocessing where an identifier A is occurred, it is replaced with replacement text.

→ No any preprocessor related directives can be understandable to compiler, that's why all preprocessor related directives are removed from source code.

➤ .i file is called extended source code which is having actual source code which is passing to compiler.

➤ For compilation & linking process, we are required to use TCC command.

➤ TCC is a external command which is available in C:\tc\Bin directory.

Syntax: TCC filename.c

ex:- D:\C1100AM>TCC P1.C

➤ When we are working with TCC command compilation & linking both will be performed at a time.

➤ If compilation is success then we will get obj file, if linking is success then we will get .exe file.

➤ For loading or execution of program, we required to use application name or program name.exe

Syntax:

ex: D:\C1100AM>p1.exe
O/P: 15 15

ex: D:\C1100AM>P1
O/P: 15 15

```
# define size 120
  void main ()
{
  int x;
  x = ++size;
  printf ("x = %d", x);
}
```

O/P: Error L value req

- By using # define we can create symbolic constant value which is not possible to change at the time of execution.

```
# define A 2+3
# define B 4+5
 void main()
{ int c;
  C = A * B;
  printf ("c = %d", c);
}
```

O/P: C = 19

C = A * B
= 2+3 * 4+5
= 2 + 12 + 5
= 19

C = (A) * (B),
= (2+3) * (4+5);
= 5 * 9
= 45

- When we are creating the replacement text with multiple tokens then recommended to place replacement text in Parameters (c)

```
# define A (2+3)
# define B (4+5)
```

C = A * B
= (2+3) * (4+5)
= 5 * 9
C = 45

2) MACRO :-

- Simplified function is called macro
- When the function body contains 1 or 2 statements then it is called simplified function.
- When the simplified functions are occurred always recommended to go for macro

Advantages :-

- Macros are faster than normal functions.
- No any physical memory will be occupied when we are working with macros

- When we are working with macros, code substitution will happen in place of bouncing process

Drawbacks :-

- No any syntactical problems can be considered at the time of pre-processing.
- Macros required to construct in single line only.
- No any type checking process is occurred, when we are working with macros (parameter checking process)
- No any control flow statements are allowed.

Prog1
```
int sum(int x, int y)
{
  return (x+y);
}
void main()
{
  int s;
  s = sum(10,20);
  printf("sum value is %d", s);
}
```

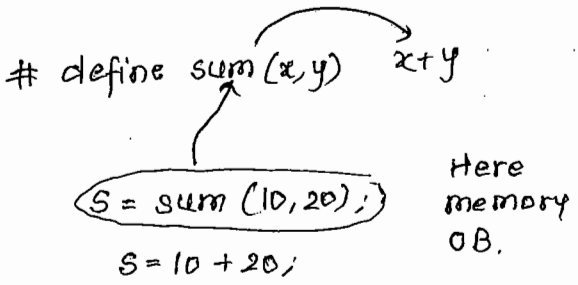
sum value is : 30

1) Parameters → 4B
2) Arguments → 4B
3) return value → 2B
4) func address → 2B/4B

12B/14B

By using macros

```
# define sum(x, y) x+y
  void main()
  {
    int s;
    s = sum(10,20);
  printf("sum value is %d", s)
  }
```



In previous program at the time of preprocessing when we are calling sum macro, automatically it is replaced with replacement text

Prog2.
```
int max(int x, int y)
{
  if (x>y)
    return x;
  else return y;
}
void main()
{
  int m;
  m = max(10,20);
  printf("Max value is : %d", m);
}
```

Using macro

```
# define max(x,y) x>y ? x:y

void main() {
 int m ;
  m = max (10,20);
  printf (" Max value is : %d", m);
}
```

O/P: Max value is : 20

Prog3.

```
# define SQR(a) a*a
void main()
 {
  int i, j;
  i = SQR(2);
  j = SQR (2+3);
  printf (" i = %d  j = %d ", i, j);
 }
```

O/P: i = 4 j = 11

i = SQR (2);
= a * a;
= 2 * 2;
= 4

j = SQR(2+3);
= a * a;
= 2+3 * 2+3;
= 2+6+3;
= 11

```
# define SQR(a) (a)*(a)
```

i = SQR(2);
= (a) * (a);
= (2) * (2);
= 4

j = SQR (2+3);
= (a) * (a);
= (2+3) * (2+3);
= 5 * 5;
= 25;

Prog4:

```
# define CUBE (a) (a)*(a)*(a)
void main()
 { int i, j;
   i = CUBE (2);
   j = CUBE (2+3);
   printf (" i = %d  j = %d", i, j);
 }
```

O/P: i = 8 . j = 125

i = CUBE(a);
= (a) * (a) * (a);
= (2) * (2) * (2)
= 8

j = CUBE (2+3)
= (a) * (a) * (a);
= (2+3) * (2+3) * (2+3)
= (5) * (5) * (5)
= 125

Prog5: NESTED MACRO

```
# define SQR(a) (a)*(a)
# define CUBE(a) SQR (a)*(a)
void main()
{ int i ;
  i = CUBE (2+3);
  printf ("i = %d", i);
}
```

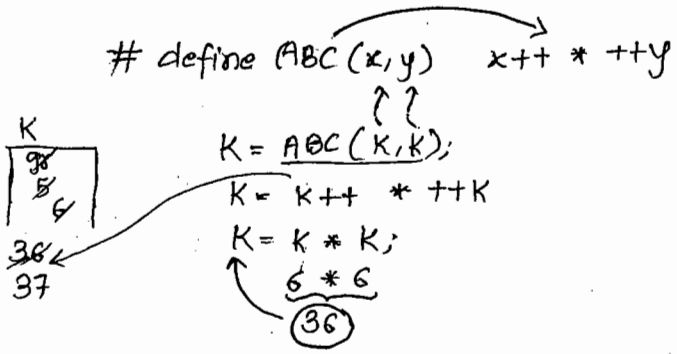
O/P: i = 125

i = CUBE (2+3);
= SQR (a) * (a)
= SQR (2+3) * (2+3);
= (a) * (a) * (5);
= (2+3) * (2+3) * 5;
= 5 * 5 * 5 = 125

Prog6:

```
#define ABC (x,y)   x++*++y
void main()
{
    int K;
    K = 5;
    K = ABC(K,K);
    printf("K = %d", K);
}
```

O/P: K = 37



2) FILE INCLUSION PRE PROCESSOR (# include)

- By using this preprocessor we can include a file in another file. Generally by using this preprocessor, we are including Headerfiles.
- A Header file is a source file which contains forward declaration of predefined functions, global variables, constant values, predefined datatypes, predefined structures, predefined macros, inline functions.
- .h file doesn't provides any implementation part of predefined functions, it provides only forward declaration (Prototype).
- A 'C' program is a combination of predefined and userdefined functions.
- .C file contains implementation part of user defined functions and calling statement of predefined functions.
- If the function is userdefined or predefined, logic part must be required.
- project related .obj files provides implementation of user defined functions, .lib files provides implementation part of pre-defined functions which is loaded at the time of linking.
- As per function approach, when we are calling a function which is defined later for avoiding the compilation error, we are required to go for forward declaration i.e prototype is required.
- If the function is user-defined, we can provide forward declaration explicitly but if it is pre-defined func^n, we required to use header-file.
- In 'C' programming language, .h files provides prototype of predefined function.
- As a programmer, it is possible to provide forward declaration of pre-defined func^n explicitly but when we are providing forward declaration then compiler thinks it is user-defined func^n so not recommended

➤ .h file doesn't pass for compilation process but .h file code is compiled. When we are including any header file at the time of pre-processing, that header file code will be substituted into current source code and along with current source code header file code also compile.

Syntax :

include <filename.h>
or
include "filename.h"

using angular body syntax

double quotation syntax

- #include <filename.h> when we are including header file.

→ By using this syntax, then it will loaded from default directory i.e C:\TC\INCLUDE.

→ Generally by using this syntax we are including pre-defined header files.

→ When we are including user-defined header files by using this syntax then we need to place user-defined header file in predefined header directory i.e C:\TC\INCLUDE.

- #include "filename.h"

→ By using this syntax, when we are including header, then it is loaded from current working directory.

→ Generally by using this syntax we are including user-defined header files.

→ By using this syntax, when we are including pre-defined header files then first it will search in current project directory, if it is not available then loaded from default directory. So it is time-taking process.

#include <stdio.h>
#include <conio.h> console related
#include <stdlib.h>
#include <stdarg.h>
#include <ctype.h>
#include <limit.h>
#include <math.h>
#include <string.h>
#include <dos.h>
#include <dir.h>
#include <graphic.h>
#include <process.h>

3) CONDITIONAL COMPILATION PREPROCESSOR

→ When we are working with this preprocessor, then depends on the condition status, code will pass for compilation process

- If condition is true code is pass for compilation
- If condition is false corresponding code is removed from source.

→ the basic advantage of this preprocessor is reducing .exe file size because when source code is reduced then automatically object code is reduced, so exe file size also will be reduced.

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

O/P: AB

→ In this program, when the code is passing for preprocessing, condition becomes false.

→ That's why corresponding block of the code is removed from source at the time of pre-processing

*
```
void main()
{ printf ("NIT");
# if 2>5 != 2<5
   printf ("A");
   Printf ("B");
# else
   printf ("C");
   printf ("D");
# endif
   printf ("Welcome");
}
```

O/P: NITABWelcome

*
```
void main()
{
 printf ("Hello");
# if 2>5 != 0
   Printf ("A");
   Printf ("B");
# elif 5<8
   printf ("NIT");
   printf ("C");
# else
   printf ("Hi");
   printf ("Bye");
# endif
}
```

not valid

valid

O/P: HelloNITC

- #ifdef and #ifndef are called MACRO TESTING CONDITIONAL COMPILATION PRERROCESSOR
- When we are working with this pre-processor, depends on the condition only, code will pass for compilation process (depends on macro status)
- By using this preprocessor, we can avoid multiple substitution of header file code.

```
#define NIT
void main()
{
 printf ("Welcome");
 ifdef NIT
   printf ("Hello");
   printf ("NIT");
# endif
}
```

O/P: WelcomeHelloNIT

- In previous prog. if NIT macro is not defined then corresponding block of the code is not passing for compilation process.
- In Previous prog., NIT is called null macro because it doesn't having any replacement text.

```
#ifndef TEST
  void main()
{ printf("NIT");
 #ifndef TEST
    printf("A");
    printf("B");
 #endif
    printf("Hello");
}
```

O/P: NITHello

#undef :-

- By using this preprocessor, we can close the scope of an existing macro.
- Generally this macro is required, when we are redefining an existing macro
- After closing the scope of a macro, it is not possible to access until it is redefined.

```
#define A 11
  void main()
{ printf("%d", A);                    // 11
Error{ // A = 22      Here A is constant, it is already replaced with 11
Error{ // #define A22  Here A is already defined with 11, we cannot do this
  #undef A              first undef, then def.

  #define A22
    printf("%d", A);                  //22

  #undef A
  #define A 33
    printf("%d", A);                  //33

  #undef A
    printf("%d", A);                  //Error
}
```

O/p: Error

pragma

- It is a compiler dependent preprocessor i.e all the compilers doesn't support this preprocessor.
- A preprocessor directive that is not specified by ISO Standard.
- Pragmas offers control actions of the compiler and Linker.
- #pragma is a miscellaneous directive which is used to turn on or off certain features.
- It varies from compiler to compiler if the compiler is not recognized then it ignores it.
- #pragma startup and #pragma exit used to specify which function should be called upon startup (before main()) or program exit (just before program terminates)
- startup and exit functions should not receive or return any values.
- #pragma warn used to suppress (ignore) specific warning msg from compiler.

→ #pragma warn -rvl
return value warnings

→ #pragma warn -par
parameter not used warnings

→ #pragma warn -rch
unreachable code warnings

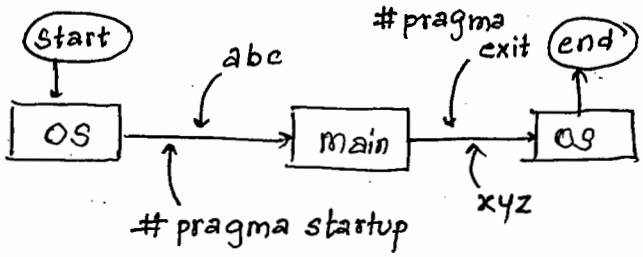
Prog-1:
```
#pragma warn -rvl
#pragma warn -par
#pragma warn -rch
int abc (int a)
  { print ("Hello abc");
  }
void main ()
  {
   abc (10);
   return;
   getch();
  }
```
O/P: Hello abc

→ When this code is passed for compilation then we are not getting any return value, parameter never used and unreachable code warning messages.

Prog-2
```
#p
void abc (void)
void xyz (void)
# pragma startup abc
# pragma exit xyz
void abc () {
    printf ("Hello abc\n");
}
```

O/P:
```
Hello abc
Hello main
Hello xyz
```

```
void xyz()
{
  printf("Hello xyz");
}
void main()
  {
   printf("Hello main");
  }
```



- In previous program, abc function is loaded first before loading the main function and xyz function is loaded after loading main function.
- Between startup and exit automatically main function is executed.
- In implementation when we are having more than 1 startup and exit function then according to the priority, we can execute those functions.

→ In #pragma startup, function which is having highest priority, it will be executed first and which is having least priority, it will be executed at last before main()

→ In #pragma startup, when equal priority occured then last specified funcⁿ will be executed first.

→ In #pragma exit, function which is having highest priority, it will be executed in end and which is having least priority, it will be executed first after main() only.

→ In #pragma exit, when equal priority occured, then last specified funcⁿ will be executed first.

```
void abc()
{ printf("from abc\n");
}
void xyz()
{
  printf("from xyz\n");
}
void close()
 {
   printf("from close\n");
 }
void end()
  {
   printf("from end\n");
  }
#pragma startup abc 2
#pragma startup xyz 1
#pragma exit close 1
#pragma exit end 2
 void main()
{
   printf("from main\n");
}
```

O/P:

```
from xyz
from abc
from main
from end
from close
```

#error

- By using this preprocessor, we can create user defined error messags at the time of compilation.

```
#define NIT
void main()
{
 #ifndef NIT
  #error NIT MACRO NEED TO BE DEFINE
 #endif
 #ifdef NIT
   printf("Welcome");
   printf("NIT");
 #endif
}
```

O/P: WelcomeNIT

In previous program, if NIT MACRO is not defined, it gives the error at the time of compiling.

#line

- By using this preprocessor, we can create user defined line sequences in intermediate file i.e .I

```
#void main()
{
    printf("A");
#if 5>2!=1
     printf("NIT");
     printf("B");
# endif
# line4
   printf("Welcome");
   printf("C");
}
```

O/P: AWelcomeC

When the previous code is preprocessing, line sequence is reset to 4

11/7/2015

ARRAYS

- An array is a derived data type in 'C' which is constructed from fundamental data type of 'C' Prog. Language.
- An array is a collection of similar types of data elements in a single entity.
- In implementation when we require 'n' no. of values of same data type, then recommended to create an array.
- When we are working with arrays always **static** memory allocation will happen i.e compile time memory management.
- When we are working with arrays always memory is constructed in continuous memory location that's why possible to access the data randomly.
- When we are working with arrays all values will share same name with unique identification value called 'index').
- Always array index must be required to start with '0' & ends with (size-1).
- When we are working with arrays we required to use array subscript operator i.e [].
- Always array subscript operator require 1 argument of type unsigned integer constant, whose value is always '>0' only.

Syntax `Data type arr [size];`

Properties of 1D Array

Size → no. of elements, Sizeof → no. of bytes

1. `int arr [5];`
   Size → 5
   size of (arr) → 10B (5 * 2 = 10B)

2. `int arr [4];`
   Size → 4
   Sizeof (arr) → 4B
   arr [0] → 1st element
   arr [1] → 2nd
   arr [2] → 3rd

3. int arr[]; error

4. int arr[0]; error

5. int arr[-5]; error

- In declaration of array size must be require to specify or else it gives an error i.e size is unknown.
- In declaration of array size must be unsigned integer constant, whose value is '> 0' only.

6. int arr[5] = {20, 10, 30, 40, 50};

   20 → arr[0];
   10 → arr[1];
   30 → arr[2];
   40 → arr[3];
   50 → arr[4];

7. int arr[5] = {10, 20, 30}

   arr[0] → 10
   arr[1] → 20
   arr[2] → 30
   arr[3] → 0
   arr[4] → 0

- In initialisation of array, if specified no. of elements are not initialised, the remaining all elements are automatically initialized with zero

8. int arr[2] = {10, 20, 30, 40, 50}; error
   In initialization of array we can't initially more than size of array elements, if we are initializing then it gives an error i.e too many initializations.

9. int arr[] = {10, 20, 30, 40, 50}; yes valid

   Size → 5;
   Size of (arr) → 10 B

→ • In initialisation of array specifying the size is optional, in this case hoN many elements are initialized that many variables are created automatically.

10. int arr[5];

    arr[0] = 10;
    arr[2] = 30;
    arr[4] = 50;

printf ("%d %d", arr[1], arr[3]);

O/P: gr → In declaration of the array by default all elements are having garbage values only bcz, by default it is autotype.

11. Static int arr[5];
arr[0] = 10;
arr[1]
arr[2]
printf ("%d %d", arr[3], arr[4]);

O/P: 0 0

12. int arr[2]
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
printf ("%d %d %d", arr[0], arr[1], arr[2]);

Valid

In C & CPP there is no any upper boundary checking process occurs, so when we are crossing the limit then depending upon OS, Security level anything can happen (Segmentation fault).

13. float arr[5.8]; (error)

14. float arr[5];
size → 5
sizeof (arr) → 5X4 = 20B.

→ On DOS based Compiler at compile time we can create maximum of 64 KB data i.e 65536B but in previous syntax we require 80,000B

16) long int arr[20000]; Error
20000 X 4 ⇒ 80,00B

17) char arr[4000] Valid

18) int size = 10;
int arr[size], error

19) Const int size = 10
int arr[size]; (error)

20) # define size 10
int arr[size]; (Valid)

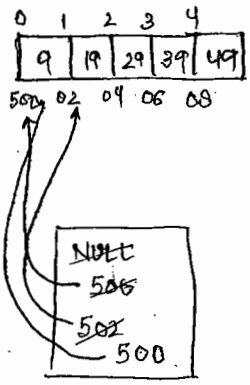
- In declaration of array size can't be variable or constant variable type.
- In declaration of array size can be symbolic constant value because at the time of preprocessing it is replaced with constant value.

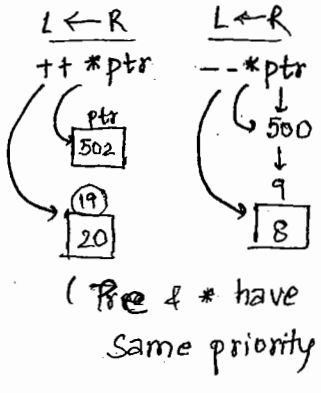
21) int arr [2+3]; Yes, valid
int arr [5];

22) int arr [2>5]; error // int arr [0];

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
int main()
{
  int arr[5] = {9,19,29,39,49}
  int near *ptr = (int near*) NULL ; // will take 2B
                  (equal priority for binary→ )
  ptr = &arr[0];
  ++ptr;
  ++*ptr;
  --ptr;
  --*ptr;
  printf("%d %d", arr[0], arr[1]);
  getch();
  return EXIT_SUCCESS
}
```

O/P: 8 20





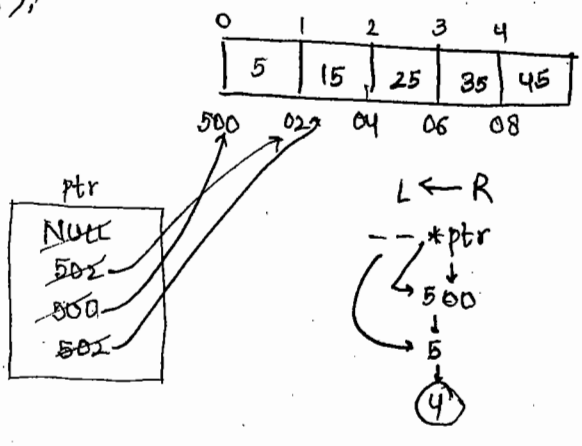
(Pre & * have same priority

* indirection operator & pre-operator both are having equal priority.
* When equal is occured, for unary operator it should be required to evaluate from Right to left only.
* Arithmatic operation of the pointers always data type dependent only.

```
# include <stdio.h>
# include <conio.h>
int main (void)
{
  int arr [5] = {5, 15, 25, 35, 45};
  int *ptr = (int *) NULL ; // ptr will take 4B (32 bit compiler)
  ptr = &arr[1];
  --ptr;
  --*ptr;
```

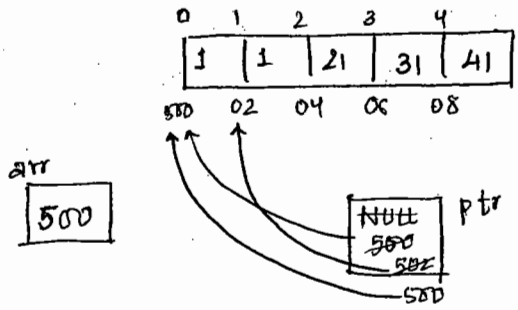
```
++ptr;
++*ptr;
printf("%d %d", arr[0], arr[0]);
getch();
return 0;
}
```

O/P: 4 16




```
#include <stdio.h>
#include <conio.h>
int main()
{
int arr[] = {1, 11, 21, 31, 41}
int *ptr = NULL;
ptr = arr;  // &arr[0]
++ptr;
--*ptr;
--ptr;
++*ptr;
Pf("%d %d %d", arr[0], arr[1], arr[2]);
getch();
return 0;
}
```

O/P: 2 10 21

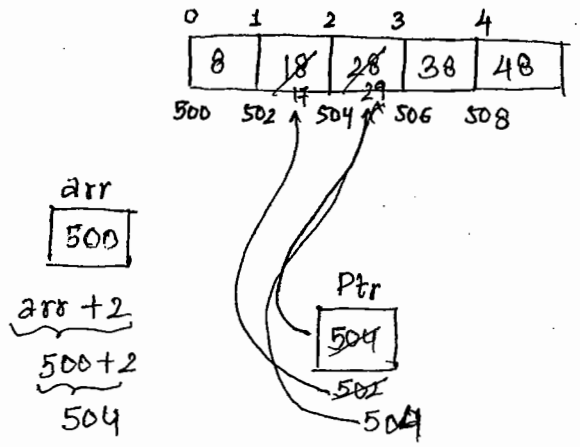


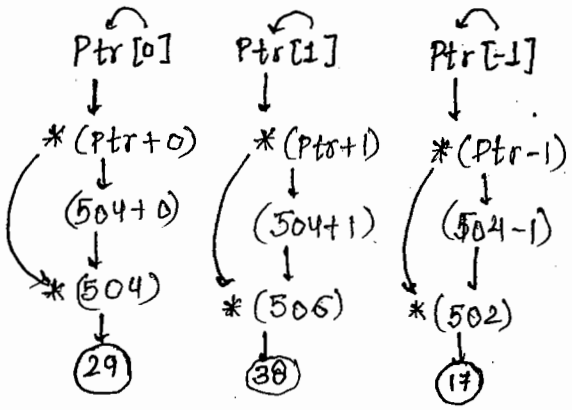
⇒ An array is an implicit pointer in C language which always maintain base address of an array.

⇒ Arr name always provides base address i.e &arr[0], arr+1 will provide next address of an array &arr[1]

```
# include <stdio.h>
# include <conio.h>
int main()
{
 int arr [] = {8, 18, 28, 38, 48}
 int *ptr = arr + 2;
 --ptr;
 --*ptr;
 ++ptr;
 ++*ptr;
 printf ("%d %d %d", ptr[0], ptr[1], ptr[-1]);
 getch();
 return 0;
}
```

O/P :- 29 38 17

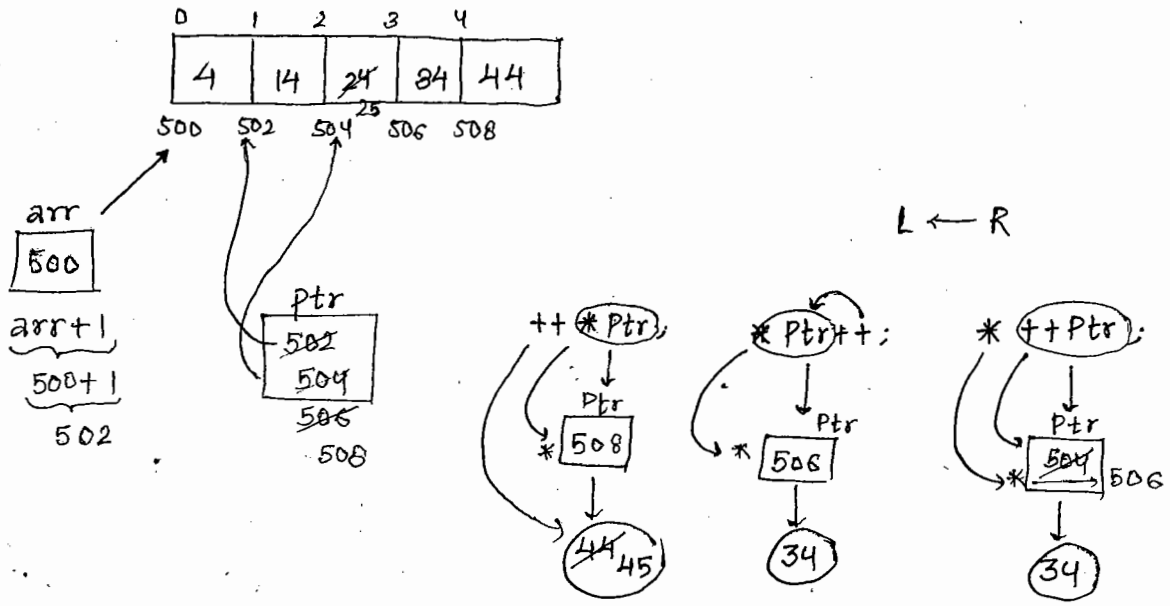




On pointer variable when we are applying subscript operator then index value will map with current value of the pointer, later it will access corresponding value.

```
# include <stdio.h>
# include <conio.h>
void int main()
{
 int arr [5] = {4, 14, 24, 34, 44};
 int * ptr = arr + 1;   // &arr [1]
 ++ptr;
 ++*ptr;
 printf ("%d %d %d", ++*ptr, *ptr++, *++ptr);
 getch();
 return 0;
}
```

O/P:- 45 34 34



Pointer Arithmetic Syntax :-

1. ++ptr;
2. ptr++;
3. --ptr;
4. ptr--;

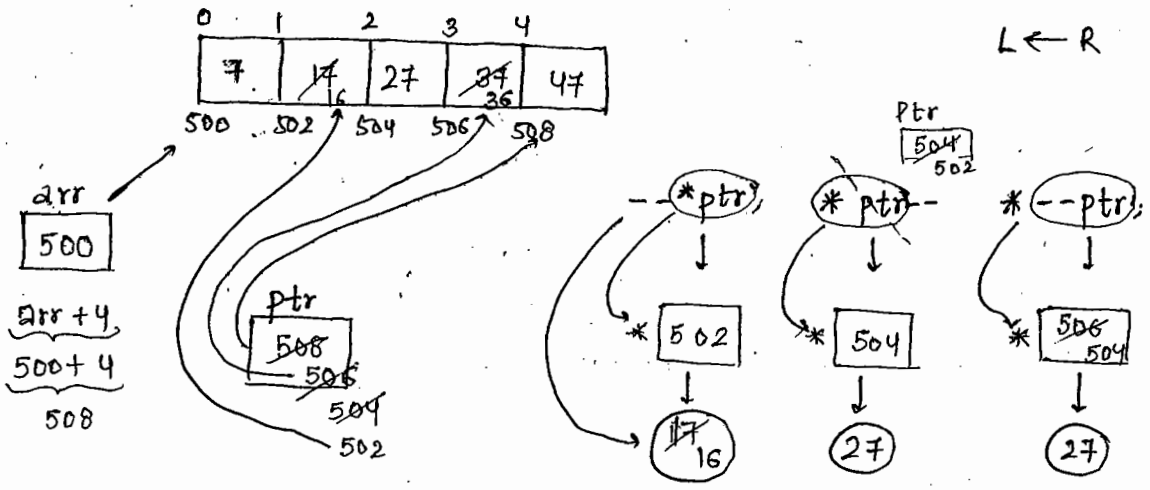
1. ++*ptr; (Pre incrementation of object)
2. (*ptr)++; (Post incrementation of object)
3. --*ptr; (Pre decrementation of object)
4. (*ptr)--; (Post decrementation of object)

1. *++ptr; (Pre incrementation of pointer & accessing the object
2. *ptr++; (Accessing the object & Post incrementation of pointer)
3. *--ptr; [Predecrementation of pointer & accessing the object]
4. *ptr--; [Accessing the object & post decrementation of pointer]

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main (void)
{
  int arr[] = {7, 17, 27, 37, 47};
  int *ptr = arr + 4;
  --ptr;
```

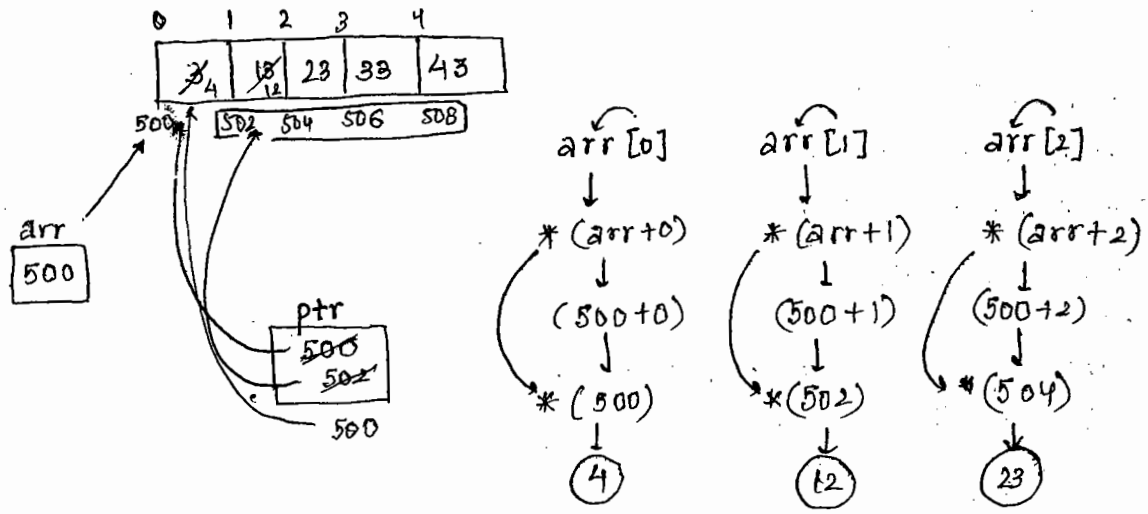
```
--*ptr;
printf("%d %d %d", --*ptr, *ptr--, *--ptr);
getch();
return EXIT_SUCCESS;
}
```

O/p :- | 16 | 27 | 27 |




```
# include <stdio.h>
# include <conio.h>
int main()
{
int arr[5] = {3, 13, 23, 33, 43};
int *ptr = arr;
++ptr;
--*ptr;
--ptr;
++*ptr;
printf("%d %d %d", arr[0], arr[1], arr[2]);
getch();
return 0;
}
```

O/p : 4 12 23

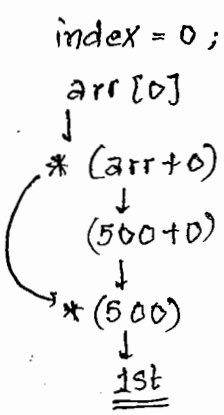


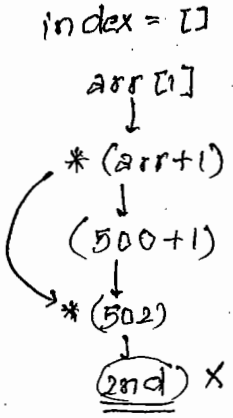
→ When we are working with arrays all element information doesn't maintain by program i.e compiler

→ When we are working with arrays, compiler will maintain only 1 element information that is called base address and remaining all elements are

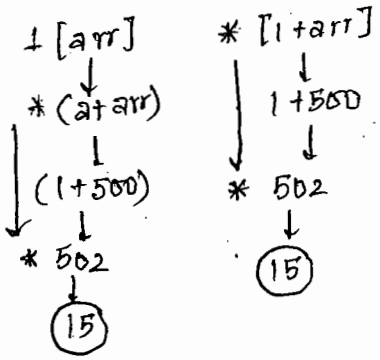
→ Acc. to index mapping mechanism, every element index value will map with base address of array to access corresponding element [Pointer Arithmetic Operation]

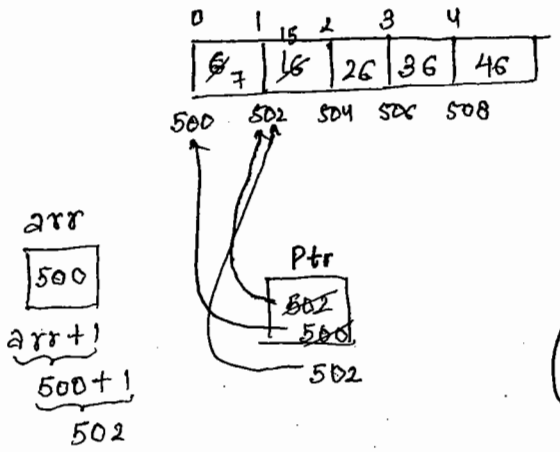
NOTE: Always Array index must be started with only 0, because according to mapping mechanism of index value is started from 1 then we can't access first element.

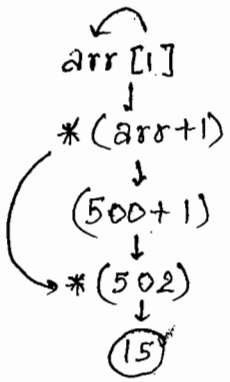


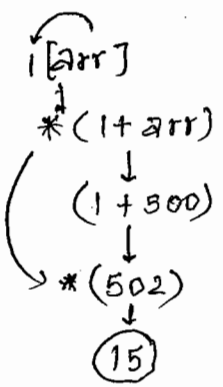



```
# include <stdio.h>
# include <conio.h>
int main()
{
  int arr[] = {6, 16, 26, 36, 46};
  int *ptr = arr+1;
  --ptr;
  ++*ptr;
  ++ptr;
  --*ptr;
  printf("%d %d %d %d", arr[1], *(arr+1), 1[arr], *(1+arr));
  getch();
  return 0;
}
```










```
# include <stdio.h>
# include <conio.h>
int main()
{
int ~~main()~~ arr[] = {12, 12, 22, 32, 42};
++arr;        //arr = arr+1;
++*arr;
--arr;        //arr = arr-1;
--*arr;
printf("%d %d, ~~%d~~ arr[0], arr[1]);
getch();
return 0;
}
```

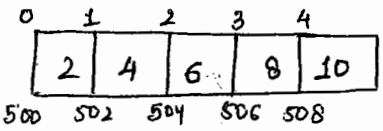
O/P: Error L value required

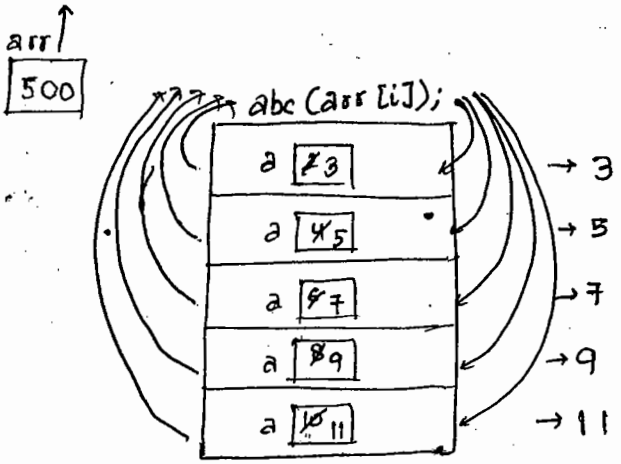
→ An array is a implicit constant pointer which always maintain base address and doesn't allows to modify it.

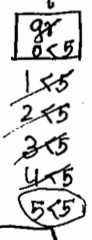
14th July 15

```
# include <stdio.h>
# include <conio.h>
void abc(int a)
{
  ++a;
  printf("%d", a);
}
int main()
{
  int arr[5] = {2,4,6,8,10};
  int i;
  printf("\n Data in abc: ");
```

```
for(i=0; i<5; i++)
    abc(arr[i]);
printf("\n Arr Data List: ");
for(i=0; i<5; i++)
    printf("%d", arr[i]);
    getch();
    return 0;
}
```





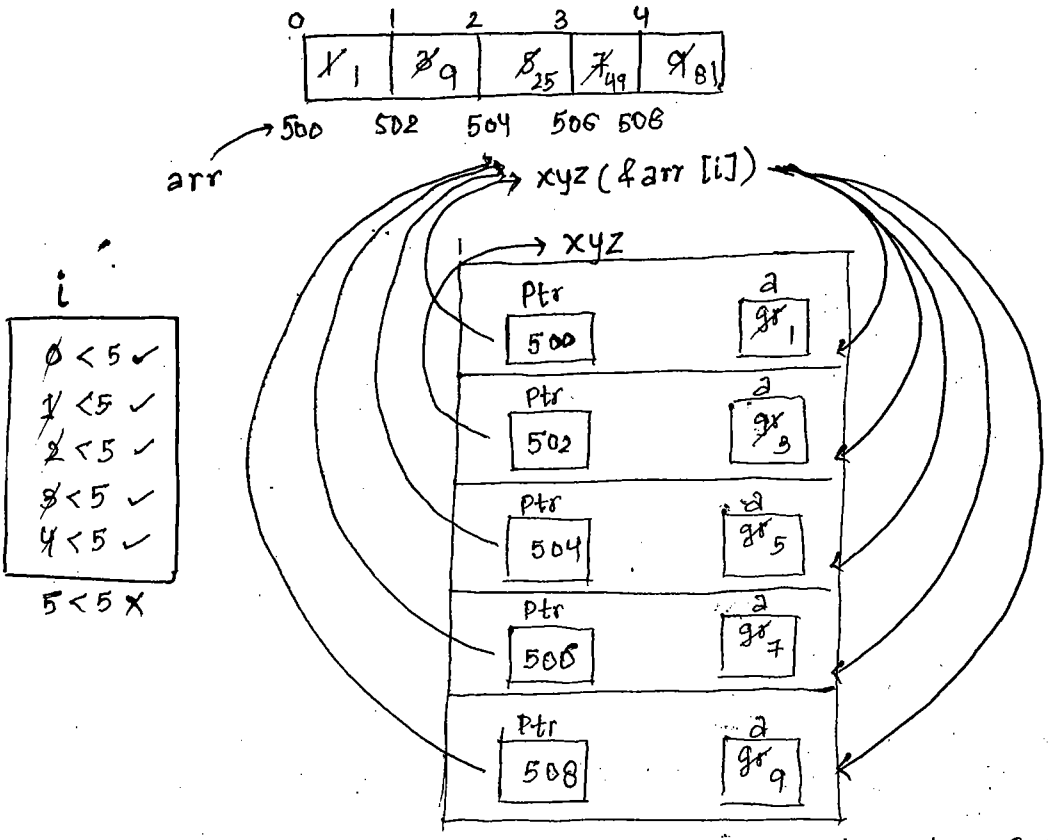


O/P:
```
Data in abc: 3 5 7 9 11
Arr Data list: 2 4 6 8 10
```

- In the above program, array elements are passing by using call by value mechanism thats why no modification of abc() function is passing back to main()
- In implementation when we are expecting the modifications then it is required to go call by address mechanism.

→
```
#include <stdio.h>
#include <conio.h>
#include <math.h>
  void xyz(int *ptr)
{ int a;
  a = *ptr;
  *ptr = (int) pow(a,2);
  printf("%d", *ptr);
}
int main() {
    int arr[5] = {1,3,5,7,9}
    int i; clrscr();
    printf("\n Data in xyz :");
    for (i=0; i<5; i++)
      xyz(&arr[i]);
      printf("\n Arr Data List : ");
      for(i=0; i<5; i++)
      printf("%d", arr[i]);
      return 0;
    }
```

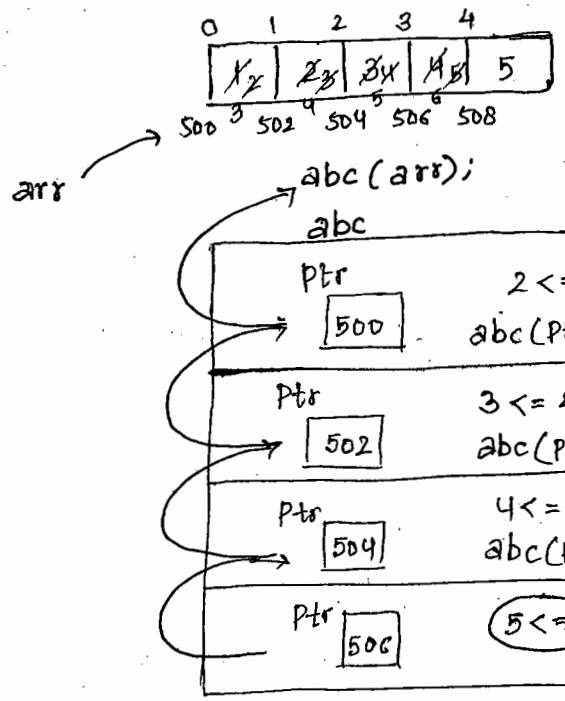
O/P:-
```
Data in xyz : 1 9 25 49 81
Arr Data List : 1 9 25 49 81
```



- In previous program, array elements are passing by using call by address mechanism that's why all the modifications of xyz() function is passing back to main function.

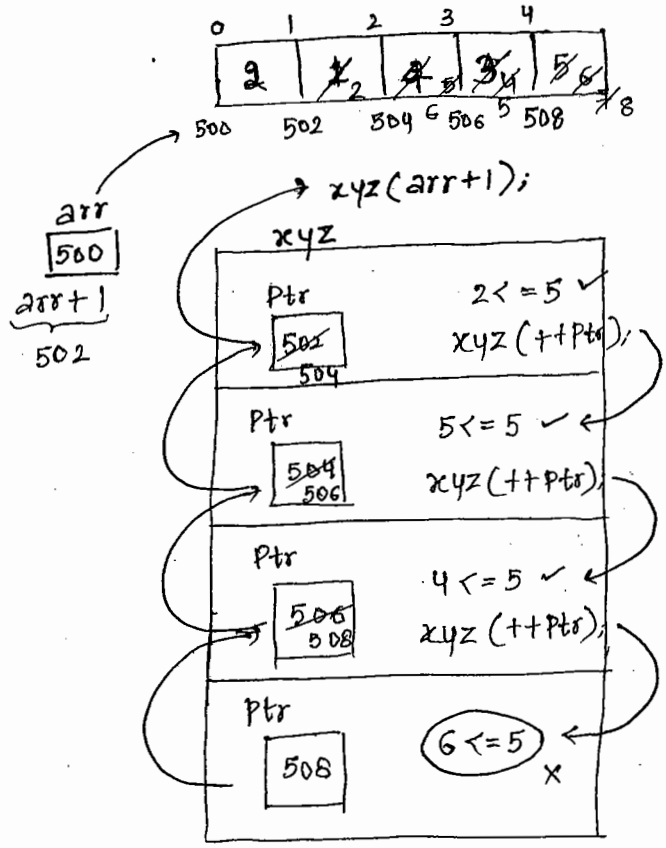
```
#include <stdio.h>
#include <conio.h>
void abc(int *ptr)
{
  ++*ptr;
  if(*ptr<=4)
    abc(ptr+1);
  ++*ptr;
}
int main()
{
  int arr[5]={1,2,3,4,5};
  int i;
  clrscr();
  abc(arr);
  printf("\n Arr Data list: ");
  for(i=0;i<5;i++)
    printf("%d",arr[i]);
  getch();
  return 0;
}
```

O/P: Arr data list : 3 4 5 6 5



Since it is a recursion when condition becomes false then second *ptr line is executed for each rollback & thus each index value is changed once again.

```
# include <stdio.h>
# include <conio.h>
  void xyz (int * ptr)
{
   ++ * ptr;
   if (* ptr <= 5)
     xyz (++ ptr);
   ++ * ptr;
}
int main()
 {
   int arr[5] = {2, 1, 4, 3, 5};
   int i;
   clrscr();
   xyz (arr+1);
   printf ("\n Arr Data List: ");
   for (i= 0; i< 5; i++)
   printf ("%d ", arr [i]);
   getch ();
   return 0;
}
```



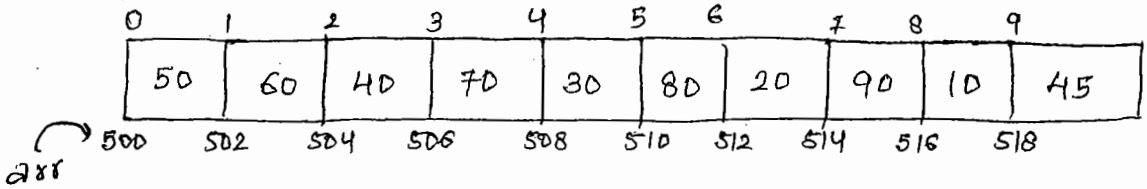
O/P:- Arr Data list : 2 2 6 5 8

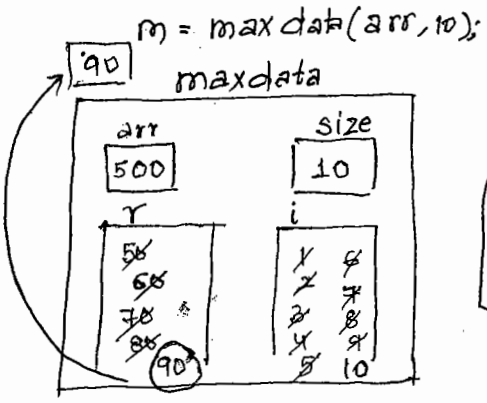
- It is not possible to pass complete array as a argument to the function. In implementation, when we required to pass complete array as a argument to the function but complete array we can access from outside of the function.
- In implementation when we required to access complete array from outside of the function then we required to pass base address of the array alongwith size.
- If we know the base address then along with the indexes, complete array we can access from outside of the function.
- In parameter location it is not possible to create an array.
- If array syntax is available, then it creates POINTER only.
- In parameter location, if `int arr[]` syntax is available then it creates a POINTER & it is indicating that hold 1D array address.

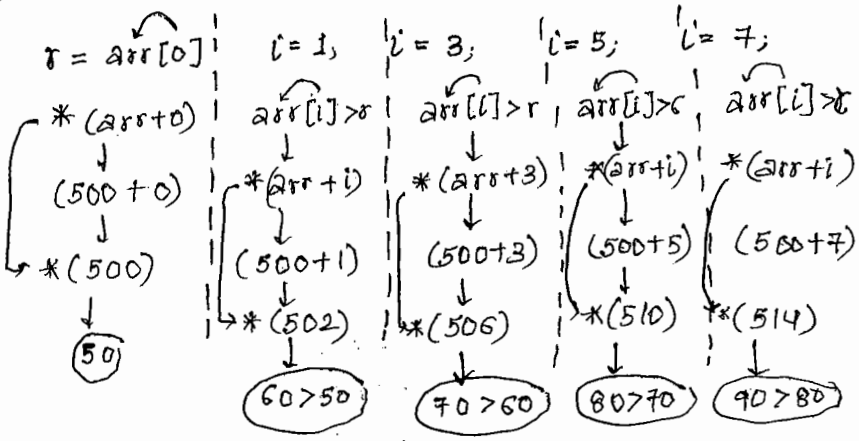
To find max no. from the data

```
# include <stdio.h>
# include <conio.h>
int maxdata (int arr[], int size)
 {
   int i, r;
   r = arr[0];   // r = *(arr+0);
   for(i=1; i<size; i++)
    {
      if (arr[i]>r)
        r = arr[i];
    }
  return r;
}
int main()
 {
   int arr[10];
   int i, m;
   clrscr();
   printf ("Enter 10 values :");
   for (i=0; i<size; i++)
    scanf("%d", &arr[i]);
   m= maxdata(arr, 10);
   printf ("Max value of list: %d", m);
   getch();
   return 0;
```

O/P: Enter 10 values:
50 60 40 70 30 80 20 90 10 45







15| July -15

```
#include <stdio.h>
#include <conio.h>
#define size 10
int main()
{
  int arr[size];
  int mindata(int *);    // declaration
  int i, m;
  clrscr();
  printf("Enter %d values: ", size);
  for(i=0; i< size ; i++)
    scanf("%d", &arr[i]);
  m = mindata(arr);      // int mindata(int arr[])
  printf("Min data of list : %d", m);
  getch();
  return 0;
}
int mindata(int *ptr)  // int mindata(int arr[])
{ int r, i;
  r = *(ptr+0);    // r = ptr[0];
  for(i=1; i<size; i++)
  {
    if(*(ptr+i) < r)    // if(ptr[i] <r)
      r = *(ptr+i);     // r = ptr[i]
  }
  return r;
}
```

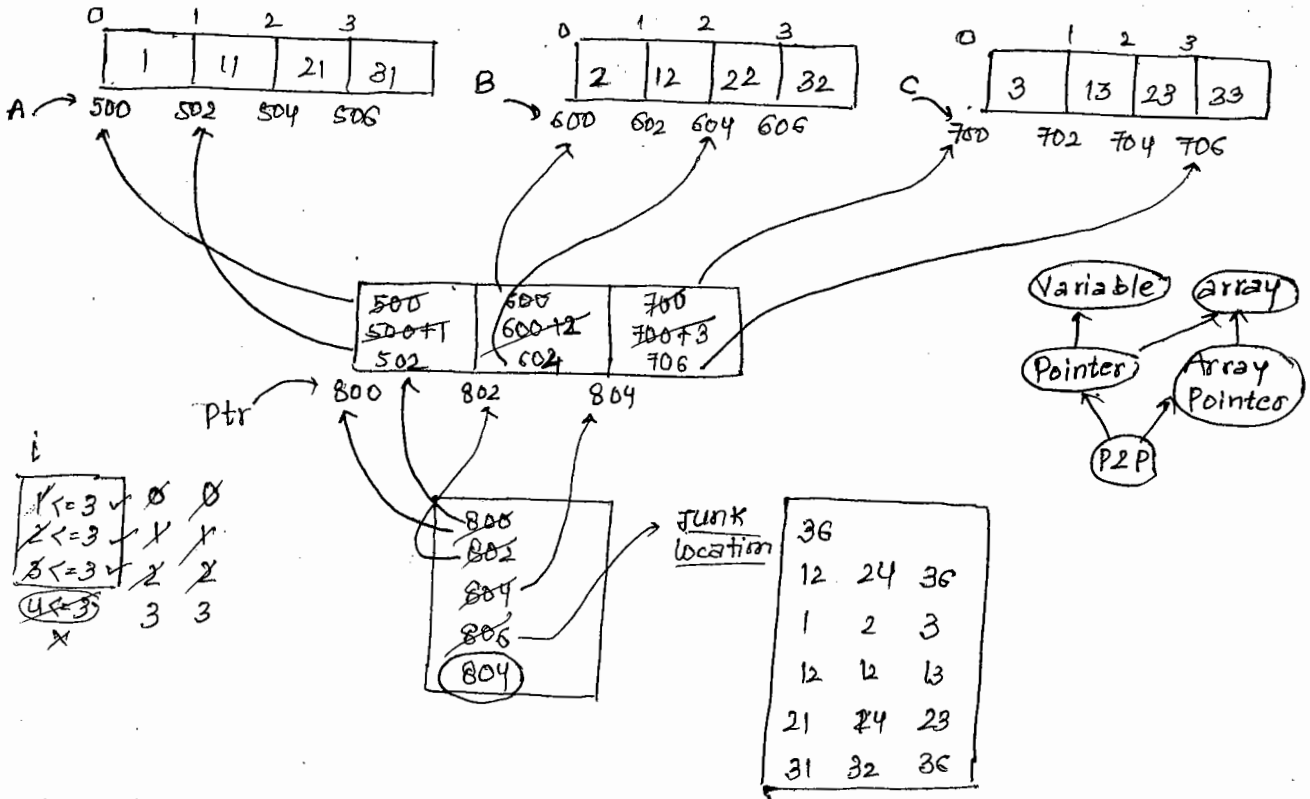
O/P:

```
Enter 10values : 50 60 70 40 30
                 80 20 80 10 45
Min data of list : 10
```

→
```
#include <stdio.h>
#include <conio.h>
int main()
{
    int A[] = {1, 11, 21, 31};
    int B[] = {2, 12, 22, 32};
    int C[] = {3, 13, 23, 33};
    int *ptr[3];        //Array Pointer
    int **pptr;         //Pointer to pointer
    int i;
    clrscr();
    ptr[0]=A;           //&A[0]
    ptr[1]=B;           //&B[0]
    ptr[2]=C;           //&C[0]
  pptr = ptr;           //&ptr[0]
  for(i=1; i<=3; i++)
    {
      *pptr++ = i;
    **pptr++ = i;
     ++ptr;
    }
  --pptr;
  printf("\n%d\n", **ptr);
  for(i=0; i<size; i++)
      printf("%d ", *ptr[i]);
  for(i=0; i<4; i++)
     printf("\n %d %d %d ", A[i], B[i], C[i]);
    getch();
    return 0;
}
```

O/P :-

```
36
12 24 36
1  2  3
11 12 13
21 24 23
31 32 36
```



- Collection of similar types of pointer in a single variable is called Array pointer.
- When we required n no. of pointer variables of same data type then go for array pointer
- Array pointer name always gives base address of array pointer.
- By using pointer to pointer variable, we can maintain pointer address / array pointer address.

SORTING

- It is a procedure of arranging the elements in a particular order i.e ascending or descending order
- In 'C' prog. lang., we are having 8 types
  1) Bubble Sort
  2) Selection sort
  3) Insertion sort
  4) Merge sort
  5) Quick Sort
  6) Heap Sort
  7) Radix or Bucket
  8) shell sort

1) **BUBBLE SORT** :-

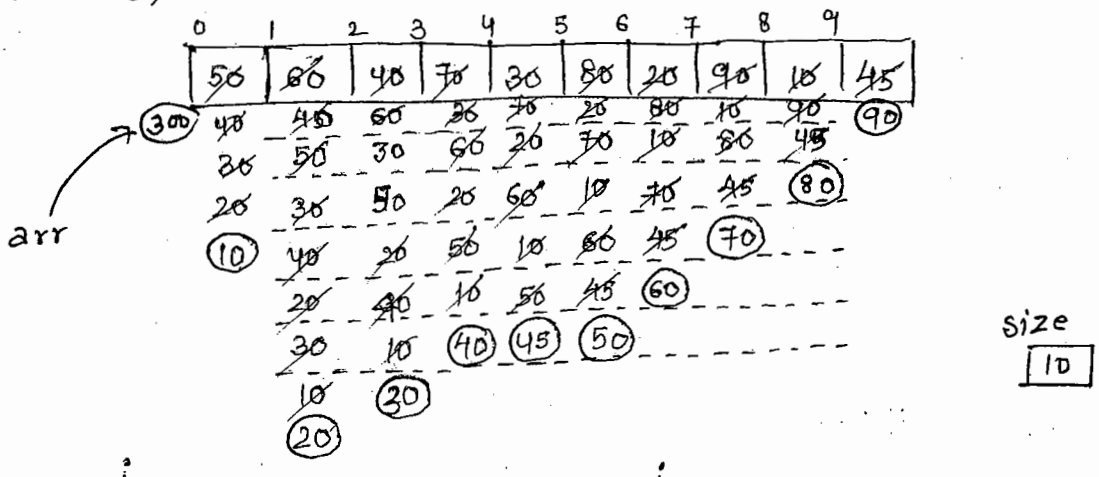
- When we are working with Bubble sort, adjacent elements are compared until last element will fix i.e max value of the list
- When we are working with Bubble Sort, if n no. of unsorted elements are available then n-1 comparision will take place.
- When we are working with Bubble sort, elements are arranged from max. to min i.e descending order but final result is as ascending order only.

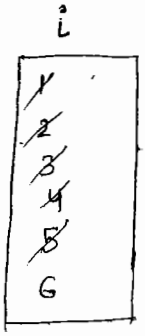
2) **SELECTION SORT** :-

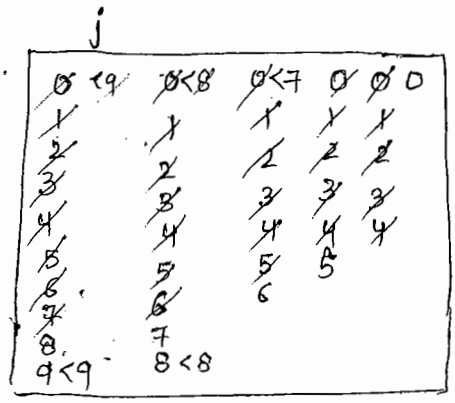
- When we are working with selection Sort, sequential comparision will take place until first value is fixed, i.e min. value of the list.
- If n no. of un-sorted elements are available then n no. of elements are available (n-1) comparison will take place.
- In this sort, elements are arranged in ascending order & final result also ascending order

```
# include <stdio.h>
# include <conio.h>
# define size 10
  int main()
{
  int arr[size];
  int t, i, j;
  clrscr();
  printf(" Enter %d values : ", size
  for(i=0; i<size ; i++)
    scanf("%d", &arr[i])
  for(i=1; i<size; i++)
    {
      for(j=0; j<size ; j++)
        {
          t = arr[j];
          arr[j] = arr[j+1];
          arr[j+1] = t;
        } //if
    } //inner
} //outer
printf("\n sorted arr data list: ");
```

```
for (i= 0 ; i< size; i++);
   printf ("%d", arr[i])
   getch ();
   return 0;
```

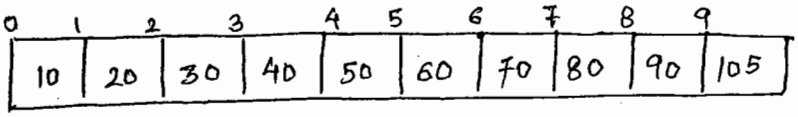







```
#include <stdio.h>
# include <conio.h>
# define size 10
int main()
{
  int arr [size]
  int sum =0, i;
  float avg;
  clrscr();
  printf (" Enter %d values : ", size);
  for (i = 0; i< size; i++)
    scanf ("%d", &arr [i]);
    for(i=0; i<size; i++)
    sum + = arr [i];
    avg = (float)sum/size;
    printf ("\n Sum of list : %d", Sum);
    printf ("\n~~Sum~~ Avg of List : %.2f", avg);
    getch();
    return 0;
}
```

```
O/P :- Enter 10 values :
  10 20 30 40 50 60 70 80 90 105
  Sum of list : 555
  Avg of list : 55.50
```



sum

~~0+10~~ ~~150+60~~
~~10+20~~ ~~210+70~~
~~30+30~~ ~~280+80~~
~~60+40~~ ~~360+90~~
~~100+50~~ ~~450+105~~

555

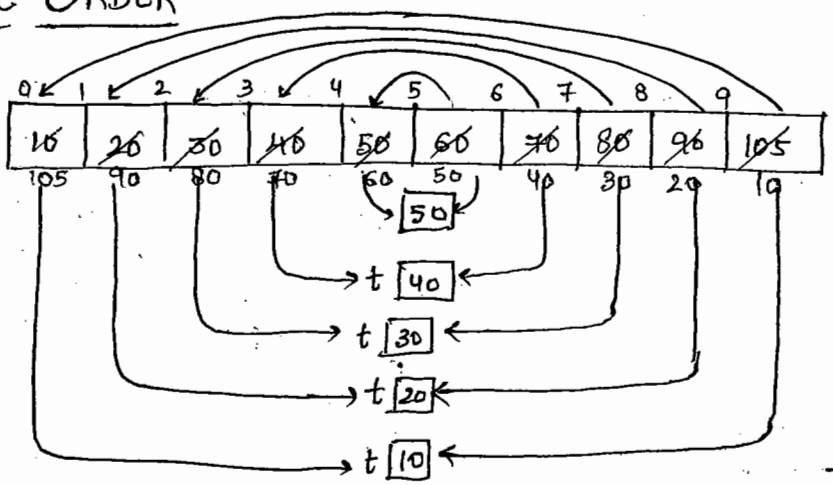
i

0<1 ✓
1<2 ✓
2<3 ✓
3<4 ✓
4<5 ✓
5<6 ✓
6<7 ✓
7<8 ✓
8<9 ✓
9<9 X → False so move outside of the loop.

Size
10

Avg
55.50

REVERSE ORDER



Size
10

i Size-i-1

0 ⟷ 9
1 ⟷ 8
2 ⟷ 7
3 ⟷ 6
4 ⟷ 5

```
# include <stdio.h>
# include <conio.h>
# define size 10
int main()
{
  int arr [size]
  int i,t;
  clrscr();
printf("Enter %d values:",size);
for (i=0; i<size; i++)
scanf("%d", &arr[i]);
for (i=0; i<size/2; i++)
{ t = arr [i]
  arr [i] = arr [size-i-1];
  arr [size-i-1] = t;
}
```

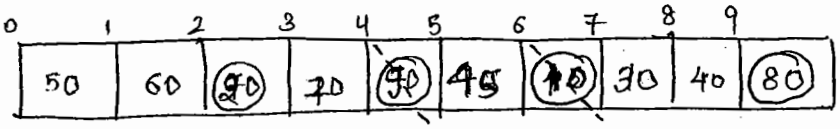
```
printf("\n Reverse arr data list: ");
for(i=0; i<size; i++)
printf("%d", arr[i]);
getch();
return 0;
}
```

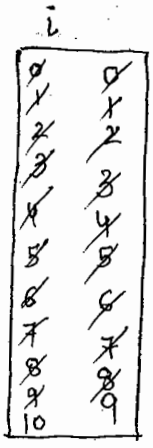
```
/*
  for(i=0; i<size/2; i++)
  { arr[i] = arr[i] + arr[size-i-1];  // a = a+b;
    arr[size-i-1] = arr[i] - arr[size-i-1];  // b = a-b;
    arr[i] = arr[i] - arr[size-i-1];  // a = a-b;
  }
*/
```

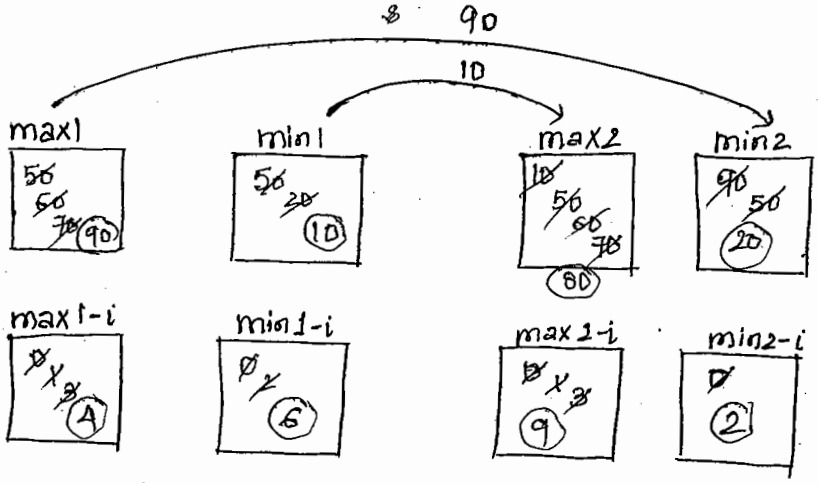
O/P:

```
Enter 10 values : 10 20 30 40 50 60 70 80 90 105
Reverse arr data list : 105 90 80 70 60 50 40 30 20 10
```

Program







For max 2, eliminate 90 we should not compare it because it is already value of max 1

Same for min 2, eliminate 10

```
#include <stdio.h>
#include <conio.h>
#define size 10
int main()
{
int arr[size];
int max1, min1, max2, min2;
int max1_i, min1_i;
int max2_i, min2_i;
int i;
clrscr();
Printf("Enter %d values : ", size);
for(i=0; i<size; i++)
 scanf("%d", &arr[i]);
max1 = min1 = arr[0];
max1_i = min1_i = 0;
for(i=1; i<size; i++)
 {
  if(arr[i] > max1)
   {
    max1 = arr[i];
    max1_i = i;
   }
  if(arr[i] < min1)
   {
    min = arr[i];
    min1_i = i;
   }
 }
 max2 = min1;
 min2 = max1;
 for(i=0; i<size; i++)
  {
   if(arr[i] > max2 && arr[i] != max1)
    {
     max2 = arr[i];
     max2_i = i;
    }
   if(arr[i] < min2 && arr[i] != min1)
    {
     min2 = arr[i];
     min2_i = i;
    }
```

O/P :-

```
Enter 10 values : 50 60 20 70 90 45 10 30 40 80
max1: 90   index: 4
min1: 10   index: 6
max2: 80   index: 9
min2: 20   index: 2
```

```
    }
    printf("\nMax1 : %d  Index : %d", max1, max1_i);
    printf("\nMin1 : %d  Index : %d", min1, min1_i);
    printf("\nMax2 : %d  Index : %d", max2, max2_i);
    printf("\nMin2 : %d  Index : %d", min2, min2_i);
    getch();
    return 0;
}
```

Two-dimensional Array

- In 2-d array, elements are arranged in rows, column format.
- When we are working with 2-d array, we required to use 2 subscript operators which indicates row size, column size
- The main m/m of 2d array is rows and elements are available in columns.
- On 2d array, when we are applying one subscript operator then it gives row name, raw name always provides corresponding row address
- From 2d array, when we required to access the element then 2 subscript operators required to use
- arr always provides main m/m, arr+1 will provides next memory of array

Syntax : `Datatype arr[RSIZE][CSIZE];`

Properties of 2D Array.

1. int arr[3][4]
   size --> 3*4
   sizeof(arr) --> 24B
2. int arr[3][3]
   size --> 3*3
   sizeof(arr) --> 18B (3*3 = 9*2B = 18B)
   3 rows
   each row 3 columns
   9 int variables
3. int arr[][]; Error
4. int arr[3][]; Error
5. int arr[][3]; Error

In declaration of 2D array, it is mandatory to specify row and column sizes or else it gives an error

6. int arr[2][3] = {10, 20, 30, 40, 50, 60};

10 --> arr[0][0]
20 --> arr[0][1]
30 --> arr[0][2]
40 --> arr[1][0]
50 --> arr[1][1]
60 --> arr[1][2]

#. By using above initialisation process, we required to initialise all elements in sequence only i.e selected no. of elements can't be initialised.

```
int arr[3][3] = {
                  {10},
                  {20,30},
                  {40,50,60}
                };
```

arr[0][0] --> 10 | arr[1][0] --> 20 | arr[2][0] --> 40
arr[0][1] --> 0 | arr[1][1] --> 30 | arr[2][1] --> 50
arr[0][2] --> 0 | arr[1][2] --> 0 | arr[2][2] --> 60

→ In initialisation of 2d array, if specific no. of elements are not initialised then remaining all elements are initialised with zero

8.
```
int arr[][] = {
                {10,20,30},
                {40,50},
                {60,70}
              };
```
Error

9.
```
int arr[3][] = {
                 {10,20},
                 {30},
                 {40,50,60}
               };
```
Error

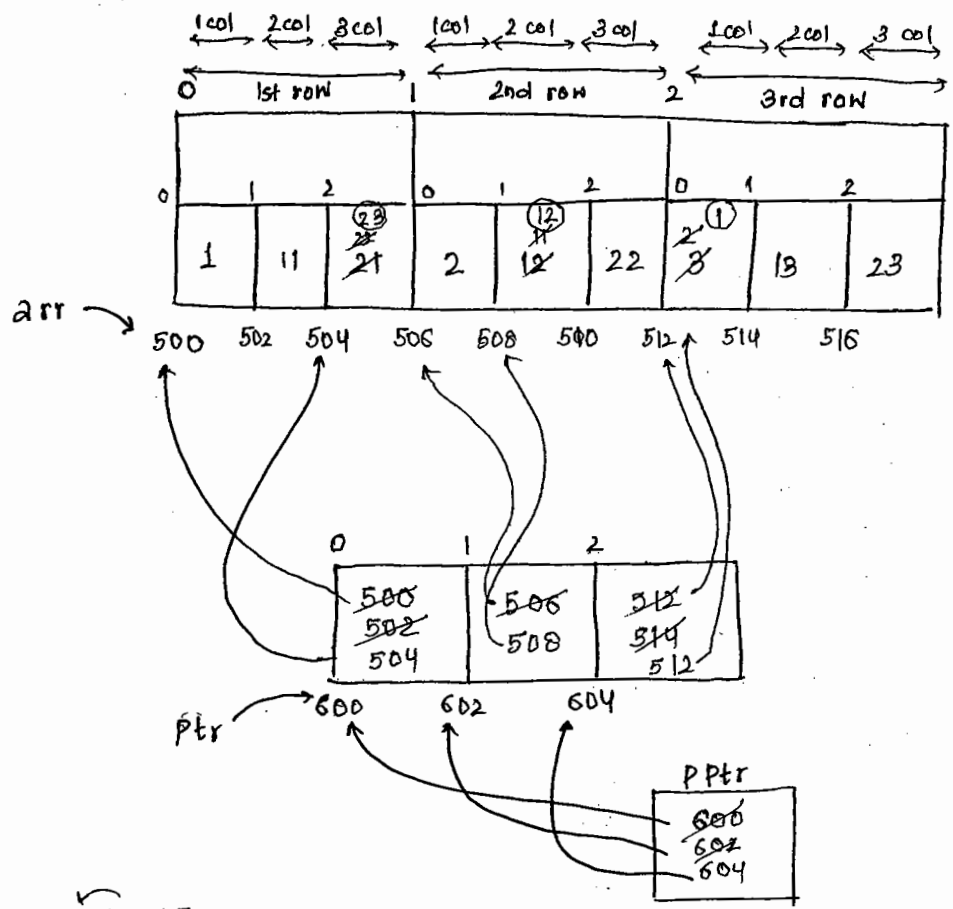
10.
```
int arr[][3] = {
                 {10},
                 {20,30},
                 {40,50,60}
               };
```
is it valid? Yes valid

→ In initialisation of 2d array, specifying the row size is optional but column size is mandatory.
But in declaration, row & column sizes are mandatory.

```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[3][3] = {
                    {1, 11, 21},
                    {2, 12, 22},
                    {3, 13, 23}
                  };
  int *ptr[3];  // Array pointer
  int **pptr;  // pointer to pointer
  clrscr();
  ptr[0] = arr;  //& arr[0][0]
  ptr[1] = arr+1;  //& arr[1][0]
  ptr[2] = arr+2;  //& arr[2][0]
  pptr = ptr;

  ++*pptr;
  ++ptr[0];
  ++**pptr;
  ++*ptr[0];

  ++pptr;
  ++*pptr;
  --**pptr;
  ++*ptr[1];

  ++pptr;
  ++*pptr;
  --ptr[2];
  --**pptr;
  --*ptr[2];
  printf("\n%d %d %d", arr[0][2], arr[1][1], arr[2][0]);
  printf("\n%d %d %d", *(*(arr+0)+2), *(*(arr+1)+1), *(*(arr+2)+0));
  printf("\n%d %d %d", *ptr[0], *ptr[1], *ptr[2]);
  printf("\n%d %d %d", **(ptr+0), **(ptr+1), **(ptr+2));
  getch();
  return 0;
}
```

O/P :-

23	12	1
23	12	1
23	12	1
23	12	1



18lock is having 18 bytes and 1 row contains 6 bytes.

arr[0][2]
↓
*(arr+0)[2]
↓
((arr+0)+2)
↓
(500+0) ← main m/m
↓
*(500) ← row address
↓
(500+2) ← column address
↓
*(504)
↓
23

arr[1][1]
↓
*(arr+1)[1]
↓
((arr+1)+1)
↓
(500+1) ← main m/m
↓
*(506) ← row address
↓
(506+1) ← column address
↓
*(508)
↓
12

arr[2][0]
↓
*(arr+2)[0]
↓
((arr+2)+0)
↓
(500+2) ← main m/m
↓
*(512) ← row address
↓
(512+0) ← column address
↓
*(512)
↓
1

---

*Ptr[0]
↓
**(Ptr+0)
↓
(600+0)
↓
*(600)
↓
*(504)
↓
23

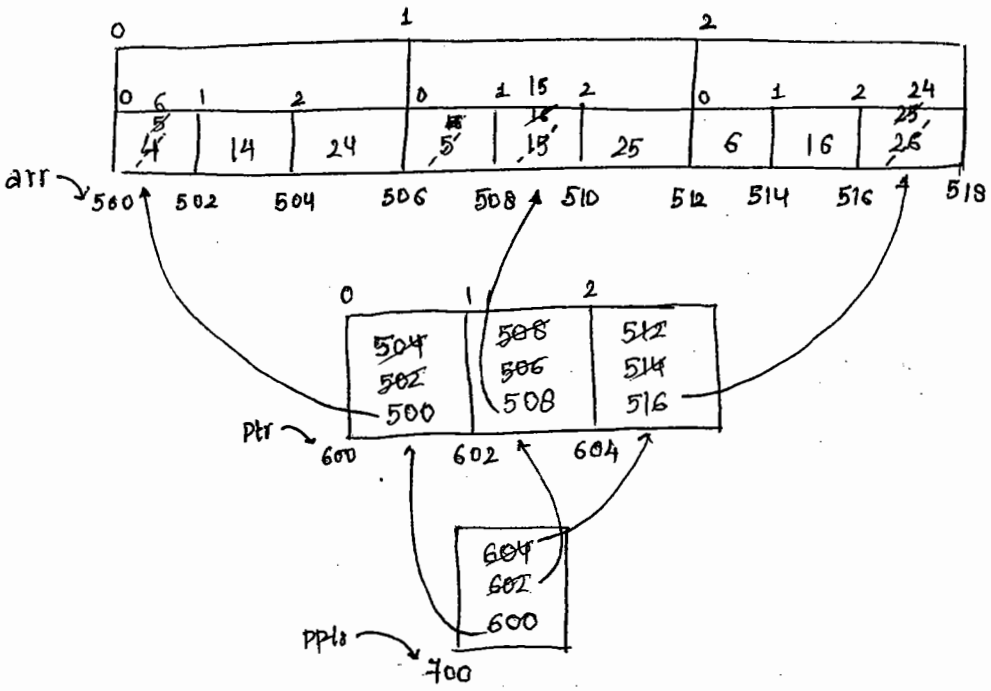
*Ptr[1]
↓
**(ptr+1)
↓
(600+1)
↓
*(602)
↓
*(508)
↓
12

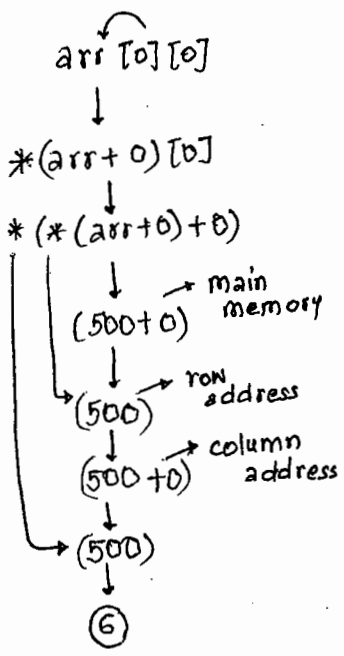
*Ptr[2]
↓
**(Ptr+2)
↓
(600+2)
↓
*(604)
↓
*(512)
↓
1

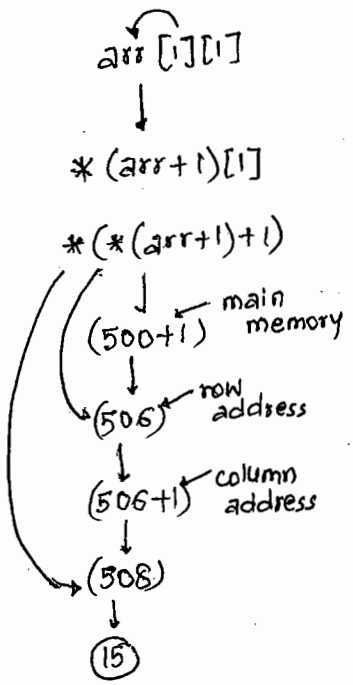
17/7/2015

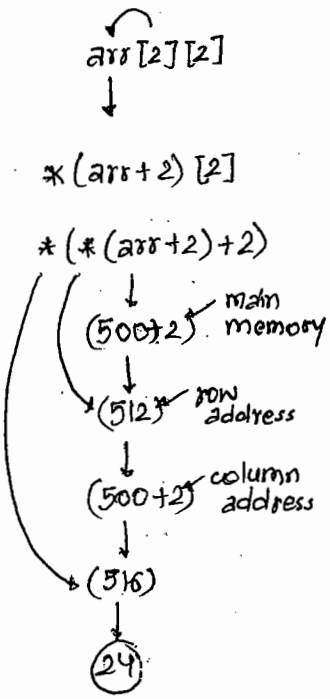
```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[][3] = {
                   {4, 14, 24},
                   {5, 15, 25},
                   {6, 16, 26}
                 };
        int *ptr[3];        // array pointer
        int **pptr;         // pointer to pointer
        clrscr();
        ptr[0] = arr[0] + 2;    // &arr[0][2]
        ptr[1] = arr[1] + 1;    // &arr[1][1]
        ptr[2] = arr[2] + 0;    // &arr[2][0]
        pptr = ptr + 2;         // &ptr[2]

        ++*pptr;
        ++ptr[2];
        --**pptr;
        --*ptr[2];

        --pptr;
        --*pptr;
        ++ptr[1];
        ++**pptr;
        --*ptr[1];

        --pptr;
        --*pptr;
        ++**pptr;
        ++**ptr[0];

        printf("\n%d %d %d", arr[0][0], arr[1][1], arr[2][2]);
```

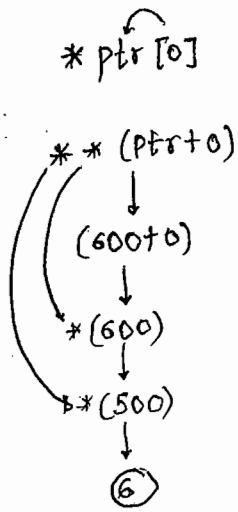
```
printf ("\n %d %d %d ", *ptr[0], *ptr[1], *ptr[2]);
printf ("\n %d %d %d ", **pptr, **pptr++, *pptr++);
printf ("\n %d %d %d", **pptr, **pptr--, **pptr--);

getch();
return 0;
}
```

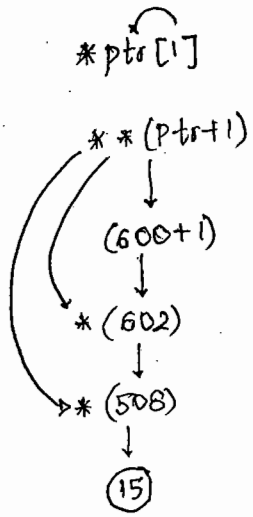


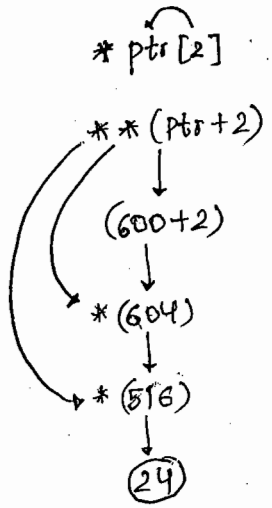


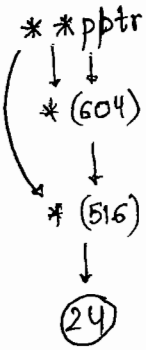


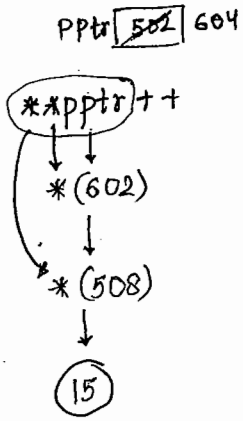


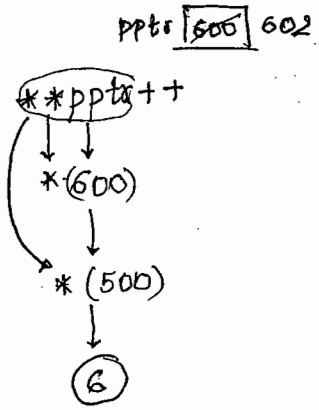












pptr
600
602
604

**pptr
*(600)
*(500)
6

pptr = 602
600

**pptr--
*(602)
*(508)
15

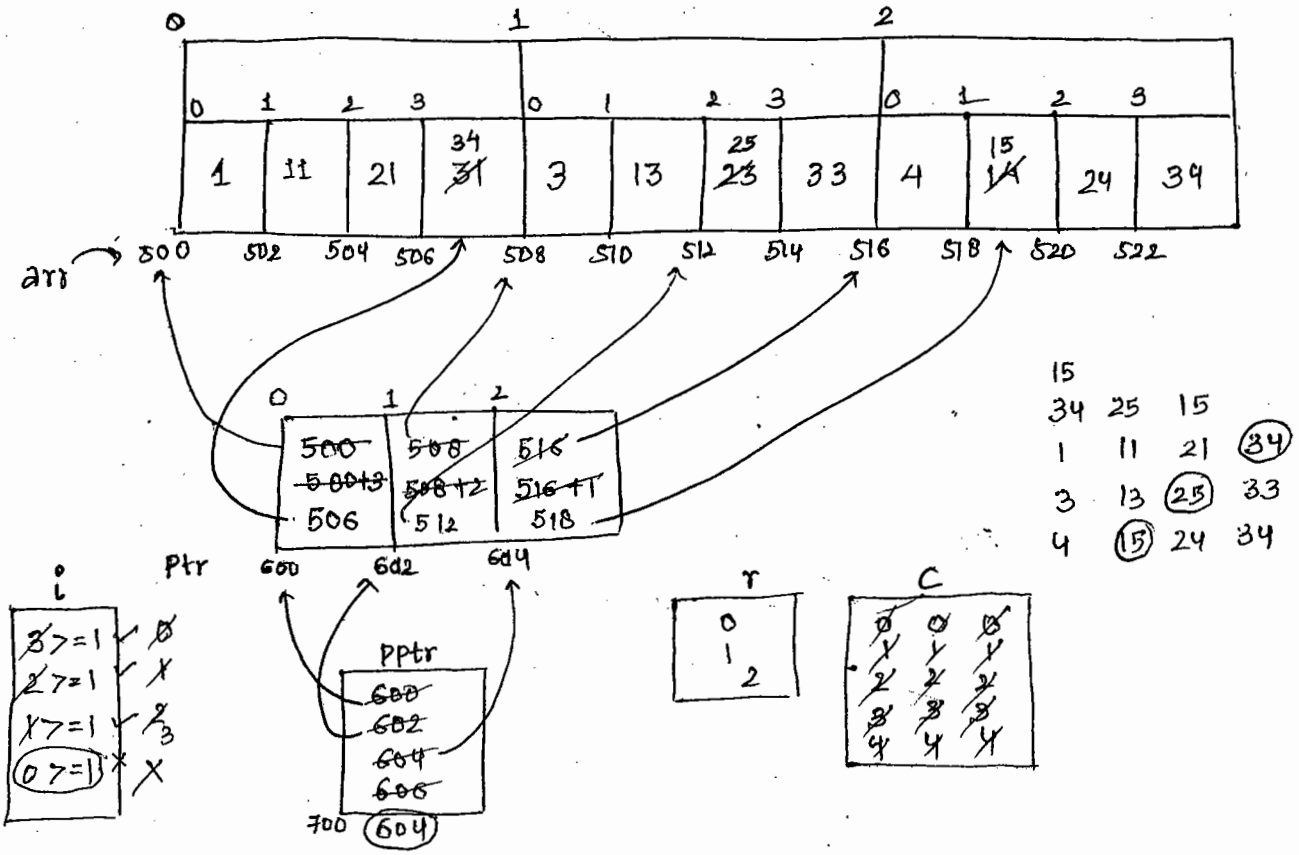
pptr ⇒ 604
602

**pptr--
*(604)
*(516)
24

Left ← Right

20/7/2015

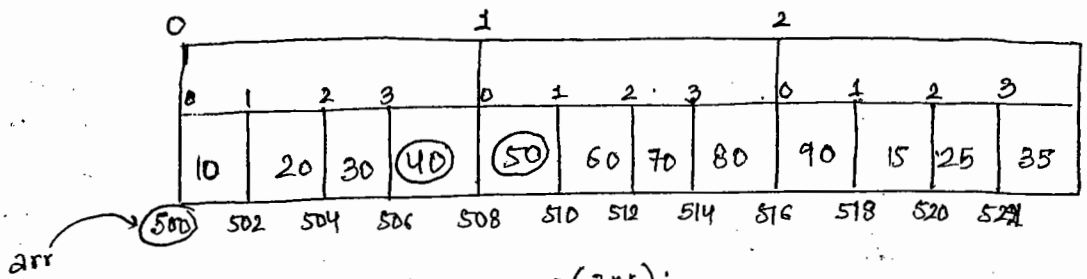
```
#include <stdio.h>
#include <conio.h>
int main()
{
  int arr[][4] = {
                   {1, 11, 21, 31},
                   {3, 13, 23, 33},
                   {4, 14, 24, 34}
                  };
  int *ptr[3];
  int **pptr;
  int i, r, c;
  clrscr();
  ptr[0] = &arr[0][0];   // arr;      //arr[0]+0;
  ptr[1] = &arr[1][0];   // arr+1;    // arr[1]+0;
  ptr[2] = &arr[2][0];   // arr+2;    // arr[2]+0;
  pptr = ptr;            // &ptr[0]
 for(i=3; i>=1; i--)
 {
   *pptr++ = i    //*pptr = *pptr+i;
  **pptr++ = i    //**pptr = **pptr+i;
  ++pptr;
 }
 --pptr;
  printf("%d\n", **pptr);
  for(i=0; i<3; i++)
   printf(" %d", *ptr[i]);
  //printf("%d", **(ptr+i));
  for(r=0; r<3; r++)
  {
   printf("\n");
   for(c=0; c<4; c++)
    printf("%3d", arr[r][c]);
  //printf("%3d", *(*(arr+r)+c));
}
getch();
 return 0;
```

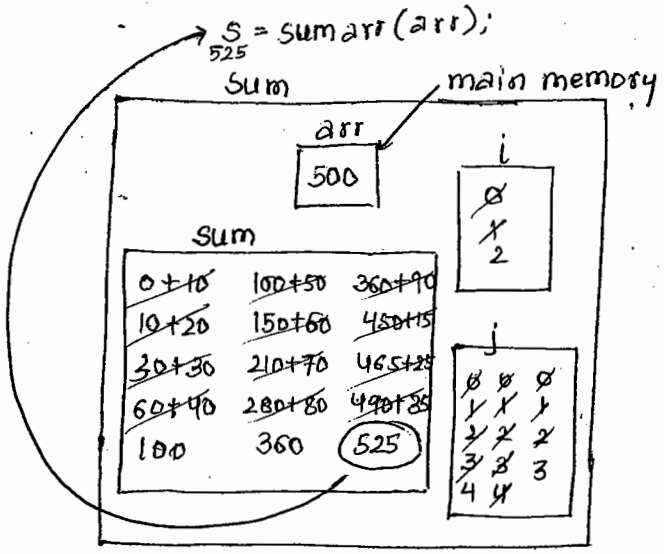


15
34 25 15
1 11 21 34
3 13 25 33
4 15 24 34

```
#include <stdio.h>
#include <conio.h>
#define rsize 3
#define csize 4
int sumarr(int arr[][csize])
{
 int i, j, sum=0;
 for(i=0; i<rsize; i++)
  {for(j=0; j<csize; j++)
    {sum += arr[i][j];    //sum = sum+*(*(arr+i)+j);
    }
  }
 return sum;
}
int main()
 {
   int arr[rsize][csize];
   int r, c, s;
   clrscr();
   printf("Enter %d*%d values:", rsize, csize);
```

```
for (r=0; r<rsize; r++)
for (c=0; c<csize; c++)
scanf ("%d", &arr[r][c]);
s = sumarr(arr);
printf ("Sum value of list: %d", s);
getch();
return 0;
}
```





i=0; j=0
arr[i][j]
((arr+i)+j)
(500+0)
*(500)
(500+0)
*(500)
10

i=0; j=3;
arr[i][j]
((arr+i)+j)
(500+0)
*(500)
(500+3)
*(506)
40

i=1; j=0;
arr[i][j]
((arr+i)+j)
(500+1)
*(508)
508+0
*(508)
50

i=1; j=3;
arr[i][j]
((arr+i)+j)
(500+1)
*(508)
(508+3)
*(514)
80

i=2; j=3;
arr[i][j]
((arr+i)+j
(500+2)
*(516)
(516+3)
*(522)
35

MULTIPLICATION OF TWO MATRICES

```
# include <stdio.h>
# include <conio.h>
# define rsize 10
# define csize 10
int main()
{
  int m1[rsize][csize];
  int m2[rsize][csize];
  int m3[rsize][csize];
  int r1, c1, r2, c2, r3, c3, c, r, t;
  clrscr();
  printf("\n ENTER MAT1 DETAILS: ");
  printf("\n ENTER NO OF ROWS: ");
  scanf("%d", &r1);
  printf("\n ENTER NO. OF COLUMNS:");
  scanf("%d", &c1);
  printf("\n ENTER MAT2 DETAILS: ");
  printf("\n ENTER NO. OF ROWS: ");
  scanf("%d", &r2);
  printf(" ENTER NO. OF COLUMNS: ");
  scanf("%d", &c2);
  if (c1 != r2)
  {
    printf("\n Error: Input data");
    getch();
    return 1;
  }
printf("\n ENTER MAT1 %d*%d ELEMENTS: ", r1, c1);
 for(r=0; r<r1; r++)
 for(c=0; c<c1; c++)
  scanf("%d", &m1[r][c]);
 printf("\n ENTER MAT2 %d*%d ELEMENTS: ", r2, c2);
 for(r=0; r<r2; r++)
 for(c=0; c<c2; c++)
  scanf("%d", &m2[r][c]);
printf("\n THE RESULT\n");
r3 = r1;
c3 = c2;
```

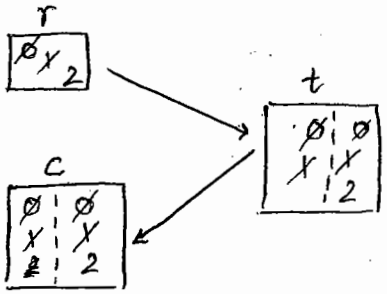
O/P:-

```
ENTER MAT1 DETAILS:
ENTER NO OF ROWS: 2

ENTER NO OF COLUMNS: 2
&
ENTER MAT2 DETAILS:
ENTER NO. OF ROWS: 2
ENTER NO. OF COLUMNS: 2

ENTER MAT1 2*2 ELEMENTS:
 2  3
 4  1

ENTER MAT 2*2 ELEMENTS:
 4  1
 2  3

THE RESULT
 14  11
 18  7
```

```
for(r=0; r<r3; r++)
{
 printf("\n");
 for(c=0; c<c3; c++)
  {
   m3[r][c]=0;
   for(t=0; t<c1; t++)
    m3[r][c]+=m1[r][t]* m2[t][c];
    printf("%3d", m3[r][c]);
  }
}
getch();
return 0;
}
```

r1 = 2 ✓ | c1 = 2 | r2 = 2 | c2 = 2 ✓ | r3 = 2 | c3 = 2

$$m1\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} * m2\begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} = m3\begin{bmatrix} 0+8+6 \ (14) & 0+2+9 \ (11) \\ 0+16+2 \ (18) & 0+4+3 \ (7) \end{bmatrix}$$



21st July 15

3D ARRAY

- In 3D array elements are arranged in blocks, rows & column format.
- When we are working with 3D array we required to use three subscript operator which indicates block size, row size & column sizes.
- The main memory of 3D array is blocks, sub main memory is rows & elements are available in columns.

- On 3d array when we are applying 2 subscript operators then it gives rowname, rowname always provides corresponding row address.
- On 3d array when we are using one subscript operator then it gives blockname, blockname always provides corresponding block address.

- from 3d array, when we required to access the data then 3 subscript operators required to use
- The Main memory of 3d array are blocks, that's why arrname always provides main memory address, arr+1 will provide next main m/m of array.

Syntax :- `Datatype arrName [BSize][Rsize][Csize]`

Properties of 3d array

1. int arr[2][3][4];
   size → 2*3*4
   sizeof(arr) → 48B
2. int arr[2][2][2];
   size → 2*2*2
   sizeof(arr) → 16B
   2 blocks
   each block → 2 rows
   each row → 2 columns
   8 int variable

arr[0][0][0]	→ 1	arr[1][0][0]	→ 5
arr[0][0][1]	→ 2	arr[1][0][1]	→ 6
arr[0][1][0]	→ 3	arr[1][1][0]	→ 7
arr[0][1][1]	→ 4	arr[1][1][1]	→ 8

3. int arr[][][]; Error
4. int arr[][][3]; Error
5. int arr[][2][3]; Error

- In declaration of 3d array, mandatory to specify block size, row size and column sizes.

6. int arr[2][2][2] = {10, 20, 30, 40, 50, 60, 70, 80};

arr[0][0][0]	→ 10	arr[1][0][0]	→ 50
arr[0][0][1]	→ 20	arr[1][0][1]	→ 60
arr[0][1][0]	→ 30	arr[1][1][0]	→ 70
arr[0][1][1]	→ 40	arr[1][1][1]	→ 80

- By using above initialization process, we required to initialize all elements in sequence only i.e selected no. of elements we can't initialize.

7. int arr[2][3][4] = {

```
{
    {10,20},
    {30},
    {40,50,60}
},
{
    {70,80,90,15},
    {25,35},
    {10}
}
};
```

- In initialization of 3d array, if specific no. of elements are not initialized then remaining all elements are automatically initialised with zero.

8. int arr[][][] = {

```
{{
  {10,20},
  {30},
  {40,50,60}
},
{
  {70},
  {80,90}
}
};
```

Error

9. int arr[][][3] = {

```
{
  {10},
  {20,30},
},
{
  {40,50,60},
  {70},
}
};
```

Error

10. int arr[][2][3] = {

```
{
  {10,20,30}
  {40,50}
},
{60},
{70,80}
}
};
```

Valid

- In initialisation of 3d array, specifying the block size is optional. but row and column sizes are mandatory.

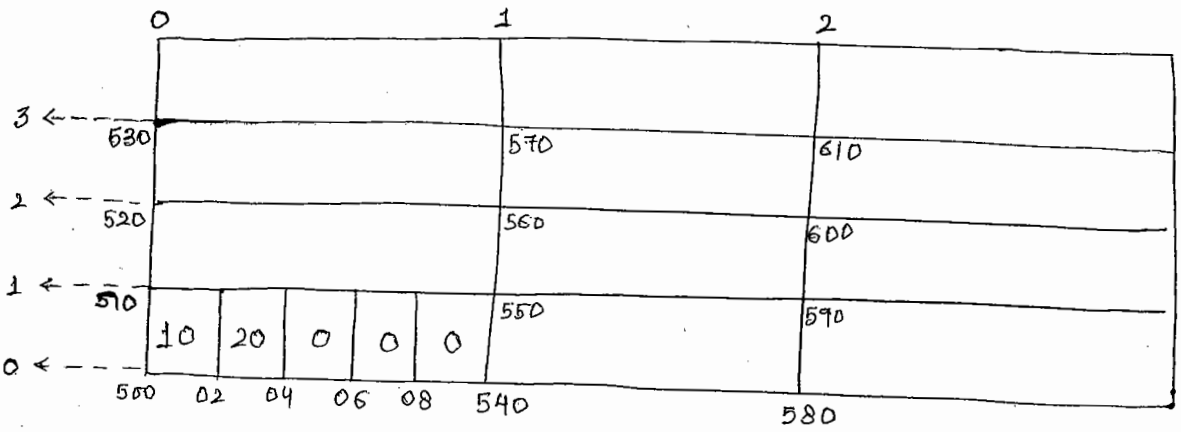
4D ARRAY

- In 4D Array, elements are arranged in sets, blocks, rows and column format.
- When we are working with 4D array, we required to use 4 subscript operators which indicates set size, block size, row size & column size.
- The Main memory of 4d array is sets, next main memory is blocks, sub-main memory is rows & elements are available in columns.
- On 4D array when we are using 1 subscript operator, then it gives set name, set name always provides corresponding set address
- On 4D array, when we are using 2 subscript operators, then it gives block name, block name always provides corresponding block address.
- On 4D array, when we are using 3 subscript operators, then it gives row address, 4 subscript operator provides elements.
- In declaration of 4D array, mandatory to specify all information, in initialization, specifying the block address is optional

syntax: `Datatype arr[SSIZE][BSIZE][RSIZE][CSIZE];`

Memory Management in 3D Array or Multidimensional Array

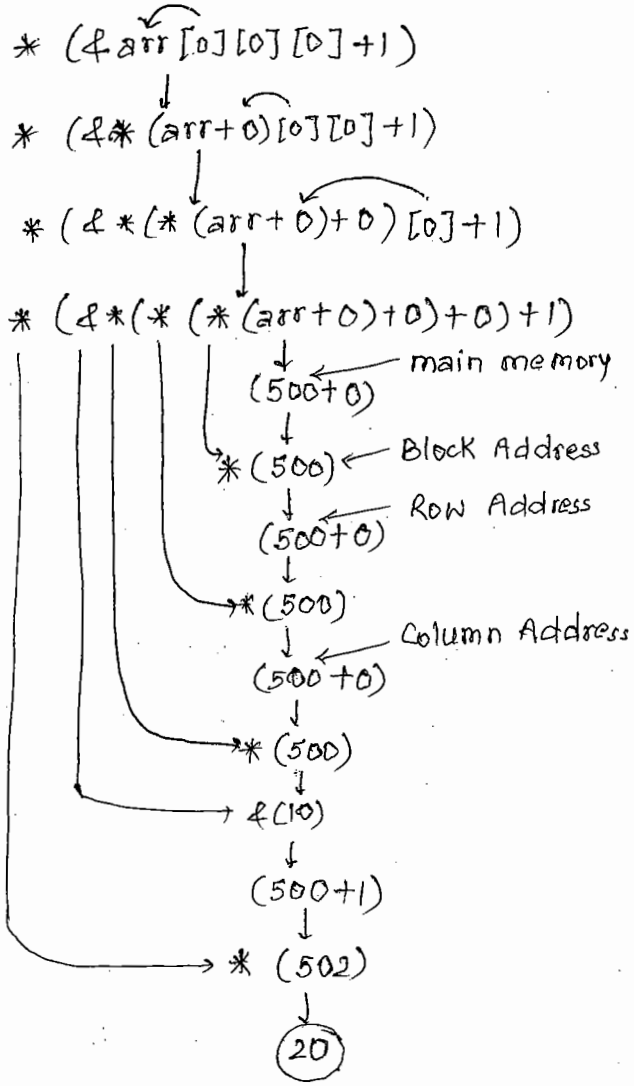
* `int arr[3][4][5] = {10, 20};`



1) arr → 500
2) arr[0] → 500
3) arr[0]+1 → 510
4) arr[0][0] → 500
5) arr[0][0]+1 → 502
6) arr[0][0][0] → 10

1) arr+1 → 540
2) arr[1] → 540
3) arr[1]+1 → 550
4) arr[1][1] → 550
5) arr[1][1]+1 → 552
6) arr[0][0][0]+1 → 11

1) arr+2 → 580
2) arr[2] → 580
3) arr[2]+1 → 590
4) arr[2][2] → 600
5) arr[2][2]+2 → 604
6) &arr[0][0][0]+1 → 502



CHARACTER OPERATIONS IN 'C'

- In 'C' programming lang, we are having 256 characters
- This all 256 characters are represented with the help of an integer value called Ascii value.
- The range of Ascii value is −128 to +127
- When we are working with characters, the representation of character must be within the single quotes only
- Within ' ' any content is called character constant
- The size of character constant is 2 bytes because by default it returns an integer value i.e Ascii value of a character constant

SPECIAL CHARACTERS IN C

	Ascii value
'A'	A 65
'a'	a 97

'Z'	90	Z
'z'	122	z
'0'	48	0
'9'	57	9
'\b'	8	backspace
'\r'	13	carriage return
' '	space 32	
'\t'	9	tab
'\n'	10	new line
'\\'	\ (92)	
'\''	' (39)	
'\"'	" (34)	
'\a'	beep 7	
"%%"	% 37	
'\%'	% 37	
';'	; 59	
':'	: 58	

1. printf ("Welcome"); Welcome
2. printf ("\"Welcome\""); "Welcome"
3. printf ("%d %d", 'A', 'a'); 65 97
4. printf ("%d %d", 'Z', 'z'); 90 122
5. printf ("%c %c", 68, 100); D d
6. printf ("%c %c", 68, 100); D d
7. printf ("%d %c", 'd', 100); 100 d
8. printf ("%d %d", 'A'+32, 'a'-32); 97 65
9. printf ("%c %c", 'A'+32, 'a'-32); a A
10. printf ("\'Welcome\'"); 'Welcome'
11. printf ("abc\nxyz"); abc
xyz
12. printf ("abc\\xyz"); abc\xyz

13. printf ("abc\\\\xyz"); abc\\
xyz

14. printf ("abc %xyz", 10); abc
xyz

15. printf ("abc%dxyz.", \n); abc10xyz

16. printf ("abc\t xyz"); abc xyz

17. printf ("abc %xyz", '\t'); abc 9xyz

18. printf ("abc %cxyz", a); abc xyz

19. printf ("Hello\b"); Hello

20. printf ("Hello\b\babc"); Helabc

21. printf ("Hello\b\b\b\b\b abc"); abcLo

22. printf ("Welcome\r"); Welcome

23. printf ("Welcome\rHello"); Hellome

24. printf ("Hello\r Welcome"); Welcome

25. printf ("%d %d %d" 10); %d

26. printf ("% %d =%d", 10); %d=10

27. printf ("Hello \a"); Hello beep(sound)

Buffer Concept :-

- Temporary storage area is called buffer.
- All standard I/O devices contain temporary memory those are called standard I/O buffer.
- In Implementation when we are passing more than required number of elements then automatically remaining all values are stored in standard input buffer and these values automatically will pass to next input functionality.

```
# include <stdio.h>
# include <conio.h>
int main()
{
  int v1, v2;
```

```
clrscr();
printf ("\n Enter value of V1 : ");
scanf (" %d ", &v1);
printf ("\n Enter value of V2: ");
scanf ("%d ", &v2);
printf ("\n value 1 : %d  Value 2 : %d", V1, V2);
getch ();
return 0;
}
```

O/P:

```
Enter Value of V1 : 10
Enter Value of V2 : 20
Value 1 : 10   value 2 : 20
```

O/P:

```
Enter value of V1 : 10 20
Enter value of V2 :
Value1 : 10    Value 2 : 20
```

➤ In implementation when we required to remove the data from standard input buffer then recommended to go for flushall() or flush() function.

flushall():- It is a predefined function which is declared in stdio.h, By using this function, we can delete the data from standard input buffer.

→ flushall() function doesn't required any parameters and doesn't return any value also.

Syntax: `void flushall (void);`

Fflush():- It is a predefined function which is declared in stdio.h, by using this function we can clear the data from standard input buffer.

→ Fflush() function requires 1 argument of type FILE* and returns void type data

Syntax: `void fflush (FILE *stream);`

```
# include <stdio.h>
# include <conio.h>
int main ()
{
    int V1, V2 ;
     clrscr ();
     printf ("\nEnter the value of V1 : ");
     scanf ("%d", &V1);
     printf ("\nEnter the value of V2 : ");
     flushall ();          // fflush (stdin);
     printf scanf ("%d ", &V2);
```

O/P:

```
Enter the Value of V1 : 10 20 30
Enter the value of V2 : 40
Value 1 :   Value 2 : 40
```

```
    printf ("\n value1: %d   value2 : %d", v1, v2);
    getch();
    return 0;
}
```

Prog-
```
#include <stdio.h>
#include <conio.h>
int main()
{
    char ch1, ch2;
    clrscr();
    printf ("\n Enter char1: ");
    //scanf("%c", &ch1);
    printf ("\n Enter char2: ");
    //fflush (stdin)
    //scanf ("%c", &ch2);
    // ch2 = getchar();
    ch2 = getch();
    printf ("\n char1 : %c   char2: %c", ch1, ch2);
    getch();
    return 0;
}
```

OUTPUT:

```
Enter char1: A
Enter char2:
char1: A     char2: B
```

→ by using scanf function we can't read multiple characters properly bcoz it reads the data from standard input buffer.

→ In implementation when we are working with multiple characters then always recommended go for getche(), or getch() function.

getche():-

- It is a predefined unformatted function which is declared in conio.h, by using this function we can read a character directly from keyboard. getche() function returns an integer value i.e Ascii value of input character.

getch():-

- It is a predefined unformatted function which is declared in conio.h, by using this function we can read a character directly from Keyboard

getch() function returns an integer value i.e Ascii value of input character.

Syntax:-

```
int getch (void);
```

NOTE :- The basic difference between getche() and getch() function is

When we are reading the character by using getche() function then it displays what character will pass but if we are using getch() function then it will not display

getchar(), putchar() :-

→ getchar() is a predefined macro which is defined in stdio.h.

→ By using getchar() macro we can read a character from standard input buffer.

→ getchar() macro returns an integer value i.e ASCII value of a input character

Syntax :

```
int getchar(void);
```

- putchar() is a predefined macro which is defined in stdio.h by using this macro we can print a character on console.
- putchar() macro required 1 argument of type integer i.e ASCII value of a printing character.

Syntax :

```
int putchar(int ch);
```

- When we are working with putchar() macro what character will print same character ASCII value is return back.

```
#include <stdio.h>
#include <conio.h>
int main()
{
char ch1, ch2;
clrscr();
printf("Enter a char : ");
```

```
    ch1 = getchar();
    ch2 = putchar(ch1);
    printf("\nchar1 : %c    char2 : %c", ch1, ch2);
    getch();
    return 0;
}
```

Output:

```
Enter a char : A
A <------- ch2 = putchar(ch1);
char1 : A   char2 : A
```

Numeric Value to Integer :-

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char ch  int value;
    clrscr();
    printf("Enter a value (0-9)");
    ch= getchar();
    if(ch>='0' && ch<='9')
      {
        value = ch - '0';
        printf("input value is : %d", value);
      }
    else
        printf("invalid input data");
        getch();
        return 0;
}
```

O/P:-
```
Enter a value (0-9): 5
input value is : 5
```

value = ch - '0'
= '5' - '0'
= 53 - 48
= 5

Read Only Variables -

- Constant variables are called Read-only variables
- When we are applying const modifier to a variable which enables read only property.
- When we are applying read only property on a variable then it doesn't allows to modify the value by using variable name.

Ex:-

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

O/P: a = 11

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

O/P: a = 11

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

O/P: Error

→ `const int a;`

'a' is a variable of type constant integer.

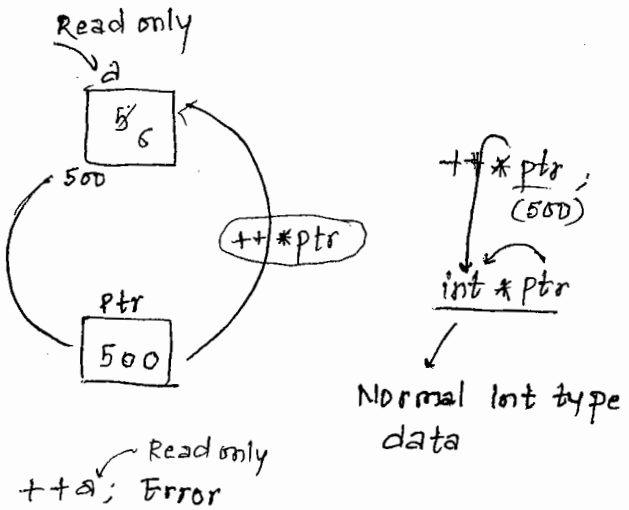
→ `int const a;`

'a' is a variable of type integer constant.

- If the variable is constant integer or integer constant both are having read only properties so doesn't allows to modify the data by using Variable name.

```c
#include <stdio.h>
int main()
{
    int const a = 5;
    int *ptr;
    ptr = &a;
    //++a; Error
    ++*ptr;
    printf("a = %d", a);
    return 0;
}
```

O/P: a = 6



* `int *ptr;`

- Ptr is called pointer to integer
- When we are working with pointer to Integer then it allows to modifies normal and read only variable data also.
- In implementation, when we are working with read only variables then recommend to go for pointer to constant integer or pointer to integer constant variables.

* `const int *ptr;`

  ptr is called pointer to constant integer

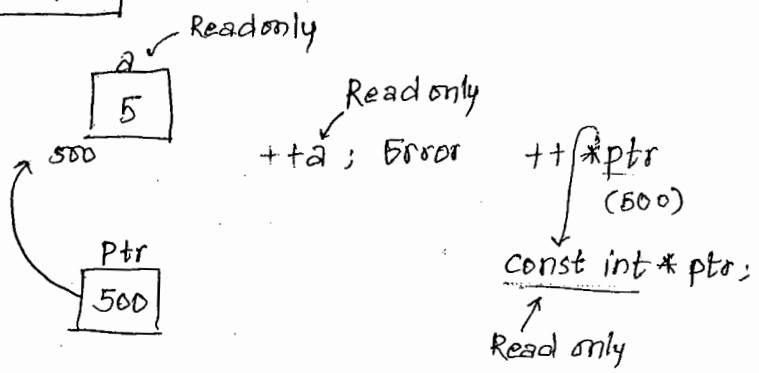
* `int const *ptr;`

  ptr is called pointer to integer constant.

> When we are working with pointer to constant integer or pointer to integer constant then it doesn't allows to modify the data even though it is normal or read only.

```
#include <stdio.h>
int main()
{
  int const a = 5;
  const int *ptr;
  ptr = &a;
  ++a;     Error
  ++*ptr;  Error
  printf("a = %d", a);
  return 0;
}
```

O/P: Error



```
#include <stdio.h>
int main()
{
  int a = 5;
  int const *ptr;
  ptr = &a;
  //++a; yes valid
  ++*ptr
  printf("%d", a);
  return 0;
}
```

O/P: Error

```
#include <stdio.h>
#include <conio.h>
int main()
{
  const int arr[2] = {5,15};
  int *ptr;
  ptr = &arr[0];
++ptr;
  ++*ptr;
  --ptr;
  --*ptr;
  printf("%d %d", arr[0], arr[1]);
  getch()
  return 0;
}
```

O/P: 4 16

When we are working with pointer to integer variable then it doesn't having any limitation i.e it is possible to modify pointer value and object value also.

```
→ #include <stdio.h>
  #include <conio.h>
    int main()
  {
    int const arr[2] = {10, 20}
    const int *ptr;   → here object is constant
    ptr = &arr[1];        pointer to integer constant
    --ptr;          yes
    --*ptr;         Error
    ++ptr;          yes
    ++*ptr          Error
    printf("%d %d", arr[0], arr[1]);
    getch();
    return 0;
  }
```

- When we are working with pointer to constant integer or pointer to integer constant then object modification is restricted but pointer modification is allowed.
- In implementation when pointer modification is required to be restricted then go for constant pointer type.
- When we are working with constant pointer type, then always recommended to initialise pointer variable.

`int *const ptr;`

→ ptr is called constant pointer to integer.

```
# include <stdio.h>
# include <conio.h>
 int main()
{
 const int arr[2] = {2, 12};
 int *const  ptr = &arr[0]
 ++ptr;      Error
 ++*ptr;     yes
 --ptr;      Error
 --*ptr      yes
 printf ("%d %d", arr[0] ,arr[1]);
 getch()
 return 0;
}
```

- When we are working with constant pointer to integer, then it doesn't allows to modify pointer value but it is possible to change object value
- In implementation, when we are required to be restricted, pointer modification and object modification then go for constant pointer to constant integer or constant pointer to integer constant.

* `const int *const ptr;`

  ptr is called constant pointer to constant integer

* `int const *const ptr;`

  ptr is called constant pointer to integer constant.

- By using this type of pointers, we can't perform any kind of operations xcept accessing the data

```
# include <stdio.h>
# include <conio.h>
 int main()
 {
  const int arr[2] = {10, 20};
  const int* const  ptr = &arr[0];
  ++ptr;      error
  ++*ptr      error
  --ptr;      error
  --*ptr      error
  printf ("%d %d", arr[0] , arr[1]);
  getch();
 }
```

STRINGS

- Character array or group of characters or collection of characters are called Strings.
- In implementation when we are manipulating multiple characters, then recommended to go for strings.
- Within the ' ' any content is called character constant, within the " " any content is called string constant.
- character constant always returns an integer value i.e ASCII value of a character
- String constant always returns base address of a string
  When we are working with string constant, always ends with nul('\0')
- The representation of null character is nul('\0') and Ascii value is 0

Syntax:

```
char str [SIZE];
```

NOTE: NULL is a global constant value which is defined in <stdio.h>

- NULL is a macro which is having the replacement data as 0 or (void *)0

  Ex: int x = NULL ; int *ptr = NULL

- nul(\0) is a ASCII character data which is having Ascii value as 0

PROPERTIES OF STRINGS

1. char str [5] ;
   size ---> 5
   sizeof (str) ---> 5B
2. char str[4];
   size ---> 4
   sizeof (str) ---> 4B

   4 char variables
   str [0] → 1
   str [1] → 2
   str [2] → 3
   str [3] → 4
3. char str [0] ; Error
4. char str [-5]; Error
5. char str [] ; Error

- In declaration of string size must be unsigned integer constant whose value is greater than zero only.

6. char Str [5] = {a,b,c,d,e}; Error

7. char str [5] = {'A', 'B', 'C', 'D', 'E'};

A ---> str [0]
B ---> str [1]
C ---> str [2]
D ---> str [3]
E ---> str [4]

8. char str [5] = {'A', 'P', 'P'};

str [0] ---> A
str [1] ---> P
str [2] ---> P
str [3] ~~str~~ [4] ---> \0(nul)

- In initialization of the string, specific characters are not initialized then remaining all elements are automatically initialised with \0(nul).

9. char str [3] = {'A', 'P', 'P', 'L', 'E'}; Error

In initialization of the string, it is not possible to initialize more than size of string elements

10. char str [5] = {100, 65, 97, 48, 57};

str [0] ---> d (100)
str [1] ---> A (65)
str [2] ---> a (97)
str [3] ---> 0 (48)
str [4] ---> 9 (57)

- In initialisation of the string, if we are assigning numeric value, then according to Ascii value, corresponding data will be stored.

11. char str [] = {'H', 'E', 'L', 'L', 'O'}; valid

size ---> 5
sizeof (str) ---> 5B

- In initialisation of the string, specifying the size is optional, in this case, how many characters are initialised that many variables are created.
- When we are working with strings, always recommended to initialise the data in double quotes only.

12. char str[10] = "Hello";

13. char str[] = "Hello";

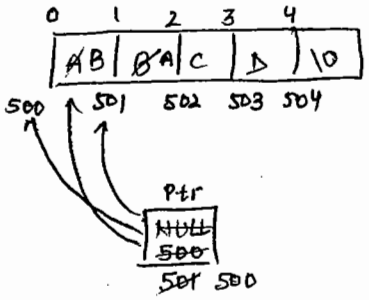
size --> 5

sizeof(str) ---> 6B

- When we are working with string constant, always it ends with \0 (nul) character thats why one extra byte m/m is required but if we are working with character array then it doesn't require one extra byte memory.

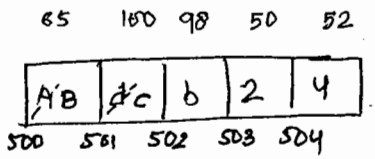
```
# include<stdio.h>
# include<conio.h>
 int main()
 {
  char str[5] = {'A', 'B', 'C', 'D', '10'};
  char near *ptr = (char near*)NULL;
  ptr = &str[0];
  ++ptr;
  --*ptr;
  ++*ptr;
  printf("%c %c%c", str[0], str[2], str[1]);
  getch();
  return 0;
 }
```

O/P: BCA



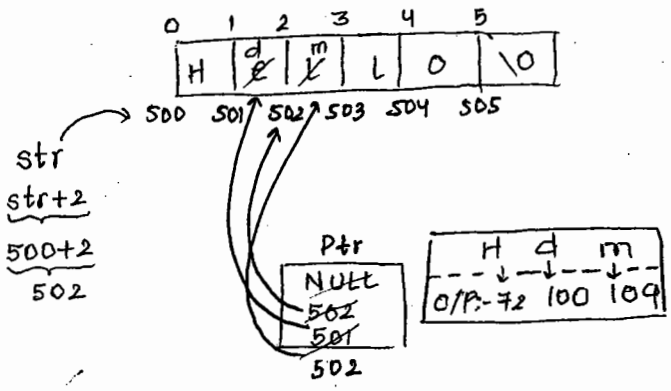
O/P: B C A

```
→ #include<stdio.h>
  # include<conio.h>
   int main()
  {
   char str[5] = {'A', 100, 'b', 50, '4'}
   char *ptr = (char*)NULL;
   ptr = &str[1];
   --ptr;
   ++*ptr;
   ++ptr;
   --*ptr;
   printf("%c %c %d", str[0], str[1], str[2]);
   getch();
  return 0;
 }
```



Prog3

```
# include <stdio.h>
# include <conio.h>
int main()
{ char str[] = "Hello";
  char *ptr = NULL;
  ptr = str + 2;
  --ptr;
  --*ptr;
  ++ptr;
  ++*ptr;
  printf ("%d %d %d", str[0], str[1], str[2]);
  getch();
  return 0;
}
```



Prog4

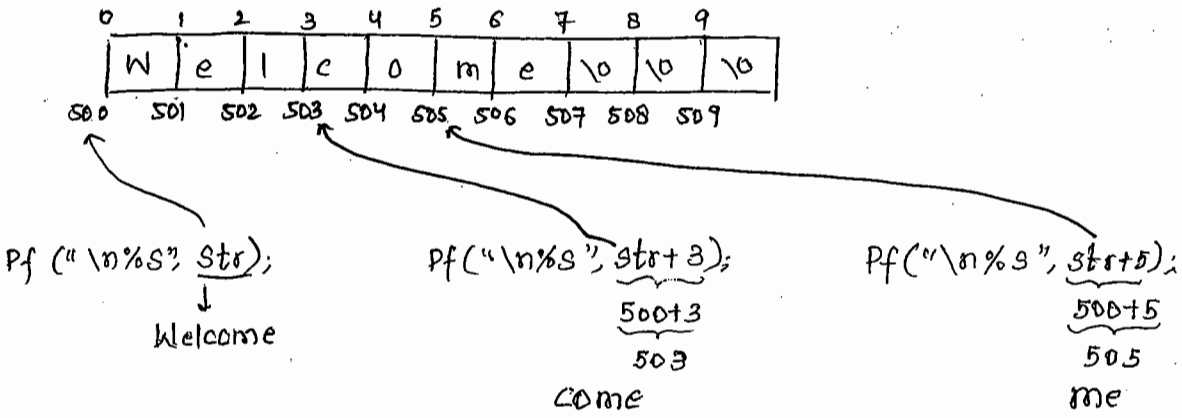
```
# include <stdio.h>
# Include <conio.h>
int main ()
{
  char str [] = "Hello"
  printf ("%c %c %c %c %c", str[0], str[1], str[2], str[3], str[4]);
  getch();
  return 0;
}
```

O/P : Hello

- When we are working with character operations recommended to go for %c format specifier.
- When we are working with string operations recommended to go for %s format specifier.
- When we are working with %s format specifier then we required to pass an address of a string, from given address upto null, entire content will print on console

```
# Include <stdio.h>
# Include <conio.h>
int main()
{
  char str [10] = "Welcome";
  printf ("\n %s", str);
  printf ("\n %s", str+3);
  printf ("\n %s", str+5);
  getch();
  return 0;
}
```

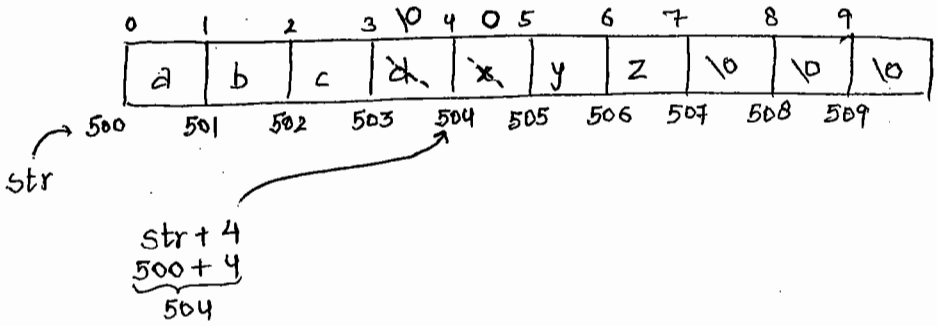
O/P : Welcome
come
me




```
# include <stdio.h>
# include <conio.h>
int main()
{ char str[10] = "abcdxyz";
  printf("\n%s", str);
  printf("\n%s", str+4);

  str[3] = 0;   → character value is \0
                  str[3] = \0
  printf("\n%s", str);
  printf("\n%s", str+4);

  str[3] = '0';     //str[3] = 48 (Ascii value)
  str[4] = '\0';
  printf("\n%s", str);
  printf("\n%s", str+5);
  getch();
  return 0;
}
```

O/P:-

```
abcdxyz
xyz
abc
xyz
abc0
yz
```



- When null character is occurred in the middle of the string, then we are not able to print complete data bcz null character indicates termination of string.

```
# include <stdio.h>
# include <conio.h>
int main()
{
 char str[] = "Welcome";
 puts(str);
 printf("%s", str);
 return 0;
}
```

O/P: Welcome
Welcome

puts() method have inbuilt new line character JAVA

puts() – It is a predefined unformatted function, which is declared in stdio.h

- By using this function, we can print string data on console.
- puts() function required 1 argument of type char* and returns an integer value.
- When we are working with puts function, automatically its prints new line character after printing string data.

Syntax:

```
int puts (char *str);
```

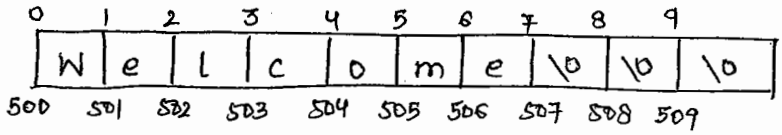
NOTE: Functions that works with the help of format specifier and which can be applied for any datatype, these are called formatted function Eg :- printf(), scanf(), fprintf(), fscanf(), cprintf(), cscanf();

→ Functions that doesn't require any format specifier and which is required for specific datatype are called Unformatted function
Eg :- puts(), gets(), getche(), cputs(), cgets()

```
# include <stdio.h>
# include <conio.h>
int main()
{
 char str[10] = "Welcome";
 int i;
 clrscr();
 for(i=0; str[i] != '\0'; i++)
  puts(str+i);
 getch();
 return 0;
}
```

O/P:
Welcome
elcome
lcome
come
ome
me
e
----
1 Blank line

at i = 7 condition becomes false

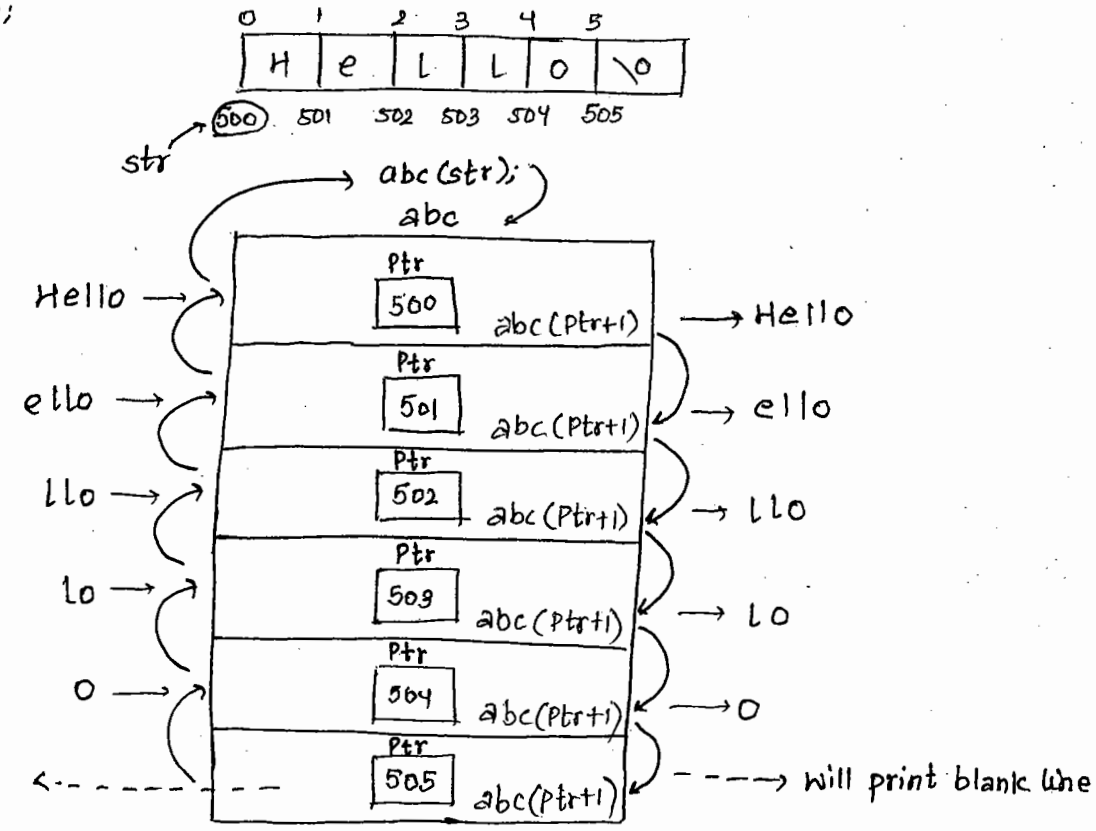


i=0;	i=1;	i=2;	i=3;	i=4;	i=5	i=6
puts(str+i)	puts(str+1)	puts(str+1)	puts(str+i)	puts(str+i)	puts(str+1)	puts(str+i)
(500+0)	(500+1)	(500+2)	(500+3)	(500+4)	(500+5)	(500+6)
puts(500)	puts(501)	puts(502)	puts(503)	puts(504)	puts(505)	put(506)
Welcome	elcome	lcome	come	eme	me	e

```
# include <stdio.h>
# include <conio.h>
 void abc (char*ptr)
{ puts (ptr);
  if (*ptr)
   abc (ptr+1);
   puts (ptr);
}
int main ()
 {
   char str[] = "Hello";
   clrscr ();
   abc (str);
   getch ();
   return 0;
}
```

O/p:

```
Hello
ello
llo
lo
o
-----
o
lo
llo
ello
Hello
```



```c
#include <stdio.h>
int main()
{
char str[] = "Rama Rao";
puts(str);
printf("%s", str);
return 0;
}
```

O/P:
```
Rama Rao
Rama Rao
```

```c
#include <stdio.h>
#include <conio.h>
int main()
{
char str[10];
clrscr();
printf("Enter a string: ");
scanf("%s", str);
//gets(str);
printf("Input string:%s", str);
getch();
return 0;
}
```

O/P:
```
Enter a string : Naresh IT
Input string : Naresh
```

- By using scanf function, we can't read the string data properly when we have multiple works because in scanf function space, tab and newline characters are treated like seperators so when the seperator is present, it is replaced with \0 character
- In scanf function, when we are using %[^\n]s format specifier, then it indicates that read the string data upto newline character occurrence.

gets():-

- It is a predefined unformatted function which is declared in stdio.h.
- By using this function we can use read the string data properly, when we are having get even multiple words.

gets() function requires one argument of type (char*) & returns (char*) only
In gets() function only newline character is treated as seperator.

Syntax: `char* gets(char *str);`

```
#include <stdio.h>
#include <conio.h>
int main()
{
 char s1 [10] = "Hello";
 char s2 [10] = "Welcome";
 clrscr();
 puts (s1);
 puts (s2);
 s2 = s1;        // Address = Address;
 puts (s1);
 puts (s2);
 getch();
 return 0;
}
```

O/P: Error L value required

- Any kind of string manipulations, we can't perform directly by using operators.
- In implementation when we required to perform any kind of string operations then recommended to go for any string handling functions or go for user defined function logic.
- String related pre-defined functions are declared in string.h

1. strcpy()
2. strlen()
3. strrev()
4. strcat()
5. strupr()
6. strlwr()
7. strcmp()
8. stricmp()
9. strstr()

i) strcpy()- By using this predefined function, we can copy a string to another string.

→ It requires 2 arguments of type (char*) and returns (char*) only.

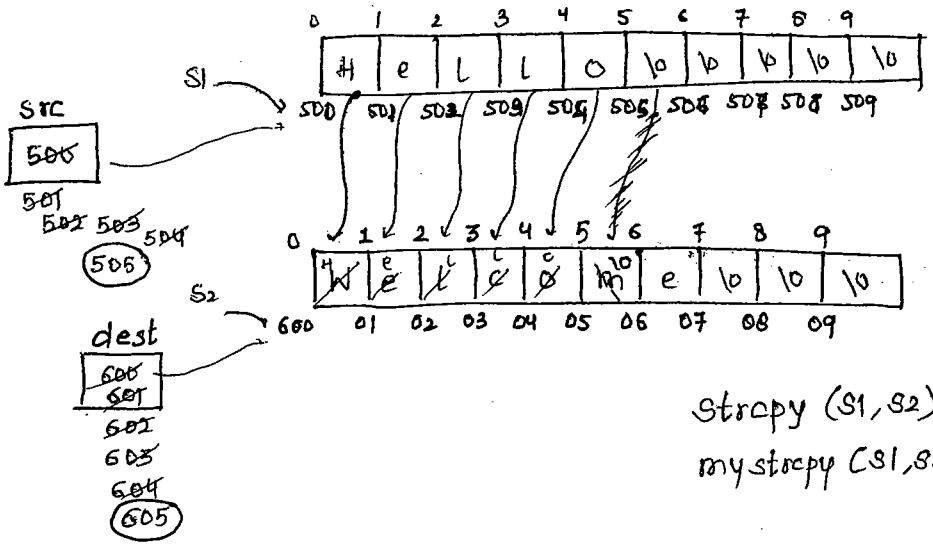
→ When we are working with strcpy() from given source address upto \0 entire content will be copied to destination string

Syntax: `char * strcpy (char * dest, const char * src);`

Prog-
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
  char s1[10] = "Hello";
  char s2[10] = "Welcome";
  clrscr();
  puts(s1);
  puts(s2);
  strcpy(s2, s1);
  puts(s1);
  puts(s2);
  getch();
  return 0;
}
```

O/p: 1. strcpy(s2, s1);
```
Hello
Welcome
Hello
Hello
```
2. strcpy(s2, s1+2);
```
Hello
Welcome
Hello
WeHello
```
3. strcpy(s2+2, s1);
```
Hello
Welcome
Hello
WeHello
```
4. strcpy(s2+2, s1+2);
```
Hello
Welcome
Hello
Wello
```

Copying a string to another string without using strcpy



strcpy(S1, S2);
mystrcpy(S1, S2);

```
#include <stdio.h>
#include <conio.h>
void mystrcpy(char *dest, const char *src)
{
  while(*src != '\0')
  {
    *dest = *src;
    ++src;
    ++dest;
  }
  *dest = '\0';
}
```

```
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "Welcome";
    clrscr();
    puts(s1);
    puts(s2);
    mystrcpy(s2, s1);    // strcpy(s2, s1);
    puts(s1);
    puts(s2);
    getch();
    return 0;
}
```

2) strlen() – By using this predefined function, we can find length of string

→ strlen function requires 1 argument of type (const char*) and returns an int type.

→ When we are working with strlen() from given address upto ~~nul~~ \0, Entire character count value will return.

→ Length of the string means total no. of characters excluding \0 character

→ Size of the string means total no. of characters including \0 character

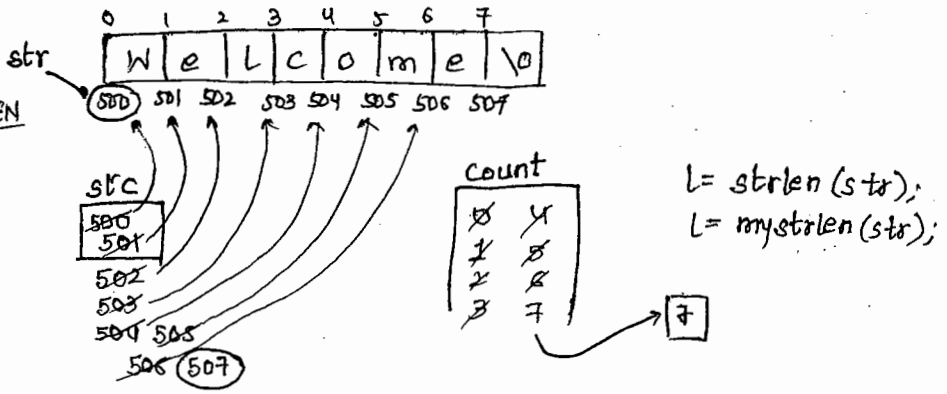
Syntax: `int strlen(const char* str);`

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
 char str[] = "Welcome";
 int l, s;
 clrscr();
 l = strlen(str);
 s = sizeof(str);
 printf("\n Length of string: %d", l);
 printf("\n size of string: %d", s);
 getch();
 return 0;
}
```

O/P:
```
Length of string : 7
size of string : 8
```

1. strlen(str); O/P: 7
2. strlen(str+3); O/P: 4
3. strlen(str+5); O/P: 2
4. strlen(str+7); O/P: 0

WITHOUT USING STRLEN FINDING LENGTH



USER-DEFINED CODE

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int mystrlen(const char *src)
{
 int count=0
 while(*src!='\0')
 {
   ++count;
   ++src;
 }
 return count;
}
int main()
 {
  char str[10];
  int s,l;
  clrscr();
  printf("\n Enter a string:");
  gets(str);
  s=sizeof(str);
  l=mystrlen(str);
  printf("\n size of string:%d",s);
  printf("\n length of string:%d",l);
  getch();
  return 0;
 }
```

O/P:
```
Enter a string : Welcome
size of string : 10
length of string : 7
```

3) strrev():- By using this predefined function, we can make string
→ strrev() requires 1 argument of type (char*) and returns (char*)
→ When we are working with strrev() from given address upto null entire string data will make reverse except null characters

Syntax: `char *strrev(char* str);`

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
 int main()
 { char str[10] = "Welcome";
   clrscr();
```

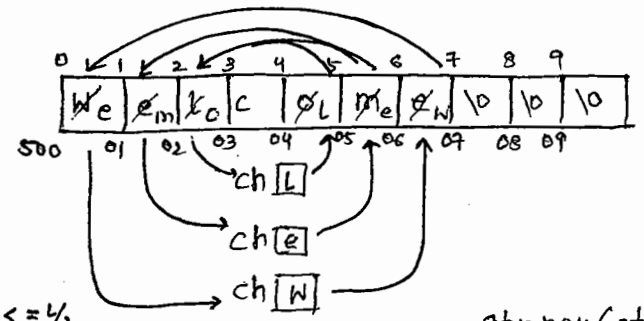
```
puts (str);
strrev(str);
printf ("Reverse string : %s", str);
getch ();
return 0;
}
```

O/P:
```
Welcome
Reverse string : emocleW
```

1. strrev(str); emocleW
2. strrev(str+3); Welemoc
3. strrev(str+5); Welcoem
4. strrev(str+7); Welcome

FINDING REVERSE WITHOUT USING PRE-DEFINED FUNCTION strrev():-

strrev() is a dependent function.
1 more function is required i.e. strlen()



src
500

L
7

i<=l/2
0<3 ✓
1<3 ✓
2<3 ✓
3<3 ✗

```
strrev(str);
mystrrev (str);
```

USER DEFINED CODE

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
Void mystrrev (char * src)
 {
    int i, l;
    char ch;
    l = strlen(src);
    for (i=0; i< l/2; i++)
    {  ch = src [i];
       src [i] = src [l-i-1];
       src [l-i-1] =ch;
    }
 }
  int main (void)
  {
    char str[10];
    clrscr();
printf ("\n Enter a string : ");
 gets (str);
 mystrrev (str);   // strrev (str)
 printf ("\n Reverse String : %s", str);
 getch ();
 return 0;
 }
```

O/P:-
```
Enter a string : Welcome
Reverse string : emoclew
```

4) strcat() :- By using this predefined function, we can concatenate a string to another string

→ concatenation means copying data from end of the string i.e appending process.

→ strcat() requires 2 argument of type(char*) and returns(char*) only

Syntax : `char* strcat (char* dest, const char *src);`

→ When we are working with strcat() function, always appending will take place at end of the destination only.

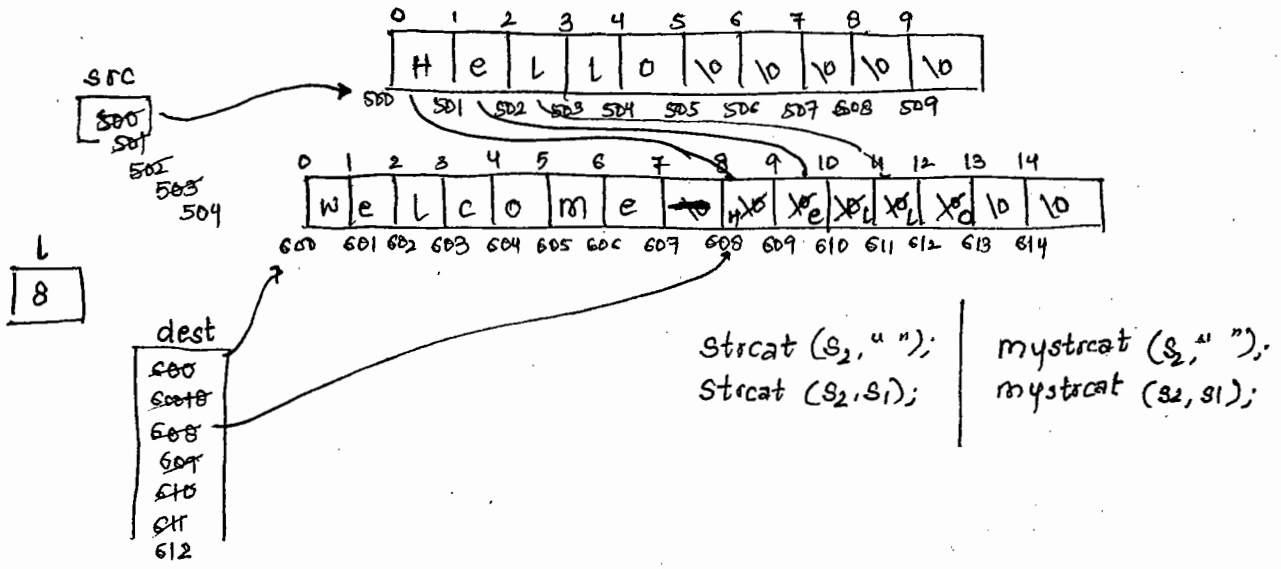
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
 char s1 [10]= "Hello";
 char s2 [15] = "Welcome";
 clrscr();
 puts (s1);
 puts (s2);
 strcat (s2, " ");
 strcat (s2, s1 );
 puts (s1);
 puts (s2);
 getch();
 return 0;
}
```

1. `strcat (s2, " ");` `strcat (s2, s1);` — `strcat(s2, " ");` `strcat (s2+3, s1);`

```
Hello
Welcome
Hello
Welcome Hello
```

2. `strcat ( s2, " ");` `strcat (s2, s1+2);` — `strcat (s2, " ");` `strcat (s2+3, s1+2);`

```
Hello
Welcome
Hello
Welcome llo
```



USER DEFINED CODE

```
# include < stdio.h>
# include <conio.h>
# include < string.h>
 void mystrcat(char *dest, const char* src)
{
  int l;
  L = strlen(dest);
  dest = dest + l;
  while (* src! = '\0')
  {
    *dest = *src;
    dest++;
     src++;
  }
   *dest = '\0';      // while (* dest ++ = * src ++) { }
}
  int main()
  {
    char s1 [15];
    char s2 [10];
    clrscr();
    printf ("\n Enter str 1 : ");
    gets (s1);
    printf ("\n Enter str 2: ");
    gets (s2);
    mystrcat (s1," ");
    mystrcat (s1,s2);
    puts (s1);
    puts (s2),
    getch();
   return 0;
```

5) strupr()

By using this predefined function, we can convert a string into upper case.
strupr() function requires 1 argument of type (char*) and returns (char*)
When we are working with strupr() function from given address upto null all lower case characters are converted into uppercase.

Syntax: `char* strupr (char* str);`

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main()
{
 char str[10];
 clrscr();
 printf("Enter a string: ");
 gets(str);
 strupr(str);
 printf("Upper case String: %s", str);
 getch();
 return 0;
}
```

O/P: Enter a string: welcome
Upper case String: WELCOME

```
1. strupr(str);      WELCOME
2. strupr(str+3);    welCOME
3. strupr(str+5);    welcoME
4. strupr(str+7);    welcome
```

USER-DEFINED CODE

```
#include <stdio.h>
#include <conio.h>
void mystrupr(char* str)
{
  int i;
  for(i=0; str[i]!='\0'; i++)
  {
    if(str[i]>='a' && str[i]<='z')
      str[i]=str[i]-32;
  }
}
int main(void)
{
  char str[10]="Welcome";
  clrscr();
  puts(str);
  mystrupr(str);
```

O/P: Welcome
Uppercase String: WELCOME

```
printf("Upper Case String : %s", str);
getch();
return 0;
}
```

6) strlwr() :- By using this predefined function, we can convert a string to lowercase.

- strlwr() function requires 1 argument of type (char*) & returns (char*)
- When we are working with strlwr(), from given address upto null all upper case characters are converted into lower case.

Syntax: `char* strlwr(char* str);`

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
void mystrlwr(char* str)
{ while(*str != '\0')
  { if(*str >= 'A' && *str <= 'Z')
      *str = *str + 32;
      ++str;
  }
}
int main()
{
 char str[15];
 clrscr();
 printf("Enter a string: ");
 gets(str);
 mystrlwr(str);          //strlwr(str);
 printf("Lower Case String : %s", str);
 getch();
 return 0;
}
```

O/P:

```
Enter a string : WELCOME
Lower case string : welcome
```

7) strcmp() - By using this predefined function, we can compare

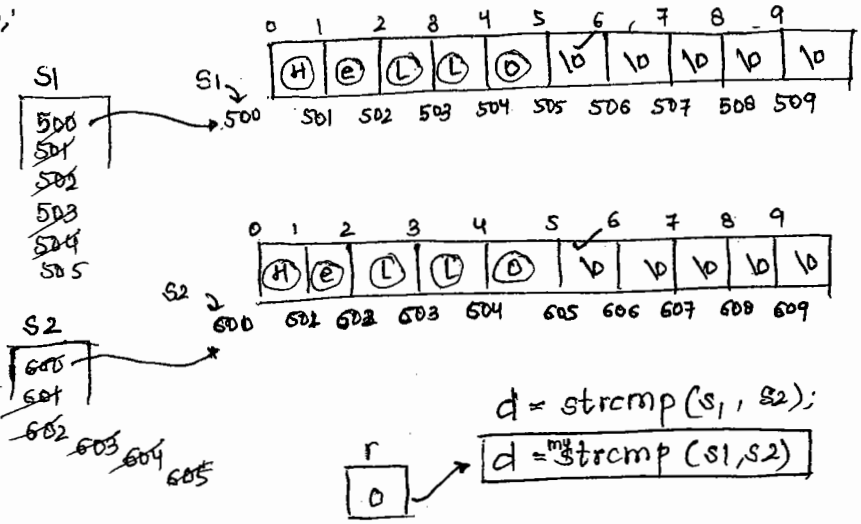
- strcmp() requires 2 arguments of type (const char*) & returns an integer value.
- When we are working with strcmp(), then character by character comparision takeplace until 1st unpaired character set is occurred.
- When 1st unpaired character set is occured then it returns ASCII value difference.

➤ At the time of comparision if there is no any diff., then by default it returns 0.

Syntax: `int strcmp (const char* s1, const char *s2);`

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
char s1[10] = "Hello";
char s2[10] = "Hello";
int d;
clrscr();
puts(s1);
puts(s2);
d = strcmp(s1,s2);
printf("ASCII value DIFF: %d",d);
getch();
return 0;
}
```

O/P:
```
Hello
Hello
ASCII value DIFF: 0
```



d = strcmp(s1, s2);

r → 0

d = mystrcmp(s1, s2)

USER-DEFINED CODE

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
    int mystrcmp (const char *s1, const char *s2)
    { int r=0;
      while (*s1 != '\0' || *s2 != '\0')
      {
        if (*s1 != *s2)
        {
          r= *s1 - *s2
          return r;  // return back to main function
        }
        ++s1;
        ++s2;
      }
```

```
int main()
{
 char s1[10];
 char s2[10];
 int d;
 clrscr();
 printf("\nEnter str1: ");
 gets(s1);
 printf("\n Enter str2: ");
 gets(s2);
 d = mystrcmp(s1,s2);        // d = strcmp(s1,s2);
 printf("\n ASCII value diff: %d", d);
 getch();
 return 0;
 }
```

8) stricmp() :- By using this predefined function, we can compare the strings without any case i.e uppercase and lowercase contents both are treated like same.

→ When we are working with strcmp() function it works with the help of case i-e uppercase and lowercase content both are different.

→ stricmp() requires 2 arguments of type (const char*) & returns an int value

syntax: `int stricmp(const char *s1, const char *s2)`

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
 char s2[10] = "hello";
 char s2[10] = "HELLO";
 int d;
 clrscr();
 puts(s1);
 puts(s2);
 d = striomp(s1, s2);
 printf("ASCII VALUE DIFF : %d", d);
 getch();
 return 0;
}
```

O/P:
```
hello
HELLO
ASCII VALUE DIFF : 0
```

1. S1 → ABC, S2 → ABC } 0
S1 → abc, S2 → abc } 0 } 0
S1 → ABC, S2 → abc } -32
S1 → abc, S2 → ABC } 32

2) S1 → ABC A, S2 → bca B (-32) } -1

3) S1 → bca B (-32), S2 → ABC A } +1

4) a) BCA → B, ABC → A } +1
b) bca → b, abc → a } +1

5) S1 → abc A (65), S2 → nul 0 65

6) S1 → nul, S2 → abc } -97

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int mystricmp(const char *s1, const char *s2)
{
 int d1, t1, t2, d2;
 while(*s1 != '\0' || *s2 != '\0')
  {
   if(*s1 - *s2 == 32 || *s1 - *s2 == -32 || *s1 - *s2 == 0)
     {
      s1++;
      s2++;  // case 1
     }
    else
      {
       t1 = *s1;
       t2 = *s2;
      if(t1 >= 'a' && t1 <= 'z' || t2 >= 'a' && t2 <= 'z')
        {
          t1 -= 32;    // case 3,5
          return (t1 - t2);
        }
      else
          if(t2 >= 'a' && t2 <= 'z' && !(t1 >= 'a' && t1 <= 'z') && t1 != 0)
          {
           t2 -= 32;   // case 2
           return (t1 - t2);
          }
       else return (t1 - t2); // case 6, 4(b)
      }
      else return (t1 - t2); // case 4(a)
    }
```

```
    return 0;
}
int main()
{
    char s1[10];
    char s2[10];
    int d1, d2;
    printf("Enter str1 : ");
    gets(s1);
    printf("Enter str2 : ");
    gets(s2);
    d1 = stricmp(s1, s2);
    d2 = mystricmp(s1, s2);
    printf("\n%d %d", d1, d2);
    getch();
    return 0;
}
```

9) strstr() :- By using this predefined function, we can find substring of a string.

- strstr() func^{n} requires 2 arguments of type const char* and returns char*

Syntax : `char *strstr(const char *str, const char* sub);`

- If searching substring is available, then strstr() returns base address of substring else returns null.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
    char str[50];
    char sub[10];
    char *ptr;
    clrscr();
    printf("Enter a string.");
    gets(str);
    printf("Enter a substring : ");
    flushall();
    gets(sub);
    ptr = strstr(str, sub);
```

O/P:

```
Enter a string : Hello Naresh IT
Enter a substring : Naresh
Naresh IT
```

```
if (ptr! = NULL)
   printf ("\n %s", ptr);
else
   printf ("\n substring not found");
  getch ();
  return 0;
}
```

28/7/15.

Prog :-
```
# include <stdio.h>
# include <conio.h>
# include <string.h>
  int alphacount (char str[])
 {
   int count = 0, i ;
    for (i=0; str [i] ! = '\0'; i++)
  {
    if (str [i] > = 'A' && str [i]<= 'Z' || str[i]>='a' && str [i]<='z')
     ++ count;
   }
    return count;
    }
   int main ()
    {  char str [50];
       int c ;
       clrscr();
       printf (" Enter a string : ");
       gets (str);
       c = alphacount (str);
       printf (" Total count value = %d", c);
       getch();
       return 0;
    }
```

OUTPUT :

```
Enter a string : Welcome!@#*%Hello123456
Total count value : 12
```

```
#include <stdio.h>
#include <conio.h>
void changecase (char *str)
{
 int i;
 for(i=0; str[i]!='\0'; i++)
  {
   if(str[i]>='A' && str[i]<='Z')
      str[i]=str[i]+32;
    else if (str[i]>='a' && str[i]<='z')
      str[i] = str[i]-32;
    else;
  }
}
int main()
{
  char str[50];
  clrscr();
  printf("Enter a string:");
  gets(str);
  changecase(str);
  printf("change case string: %s", str);
  getch();
  return 0;
}
```

O/P: Enter a string: Hello Naresh IT
change case string: hELLO nARESH it

Program: COUNTING NO. OF VOWELS

```
#include <stdio.h>
#include <conio.h>
 int main()
{
 char str[50];
 int i, count = 0;
 clrscr();
 printf("Enter a string:");
 gets(str);
 for(i=0; str[i]!='\0'; i++)
  {
    switch(str[i])
      {
        case 'A':
        case 'a': ++count;
                  break;
```

```
case 'E':
case 'e' : ++count;
           break;
case 'I':
case 'i' : ++count;
           break;
case 'O':
case 'o' : ++count;
           break;
case 'U':
case 'u' : ++count;
           break;
}
}
printf ("Total count value = %d", count);
getch();
return 0;
}
```

O/P : Enter a string : Welcome Naresh IT Hello
Total Count value : 8

Here it is taking duplicate values too.

for unique occurrence

```
#include <stdio.h>
#include <conio.h>
int main()
{
char str[50];
char temp[] = "AEIOU";
int i, k, count = 0;
clrscr();
printf ("Enter a string : ");
gets (str);
for (k=0; temp[k] != '\0'; k++)
{
  for (i=0; str[i] != '\0'; i++)
    { if (temp[k] == str[i] || temp[k] == str[i] - 32)
      {
        ++count;
        break;
      }
    }
}
printf ("Total count value = %d", count);
getch();
return 0;
}
```

O/P: Enter a string : Welcome Naresh IT Hello
Total Count value : 4

PALINDROME PROGRAM :-

```
#include <stdio.h>
# include <conio.h>
# include<string.h>
int main()
{
   char str [15];
   char temp [15];
   clrscr ();
   printf ("Enter a string: ");
   gets (str);
   strcpy (temp, str);
   strev (temp);
   if (strcmp (str, temp)==0)
      printf ("\n %s PALINDROME", str);
  else printf ("\n %s NOT PALINDROME", str);
   getch ();
   return 0;
}
```

O/P: Enter a string: MadAm
MadAm PALLINDROME

To print total no. of occurrence of a digit into '*'

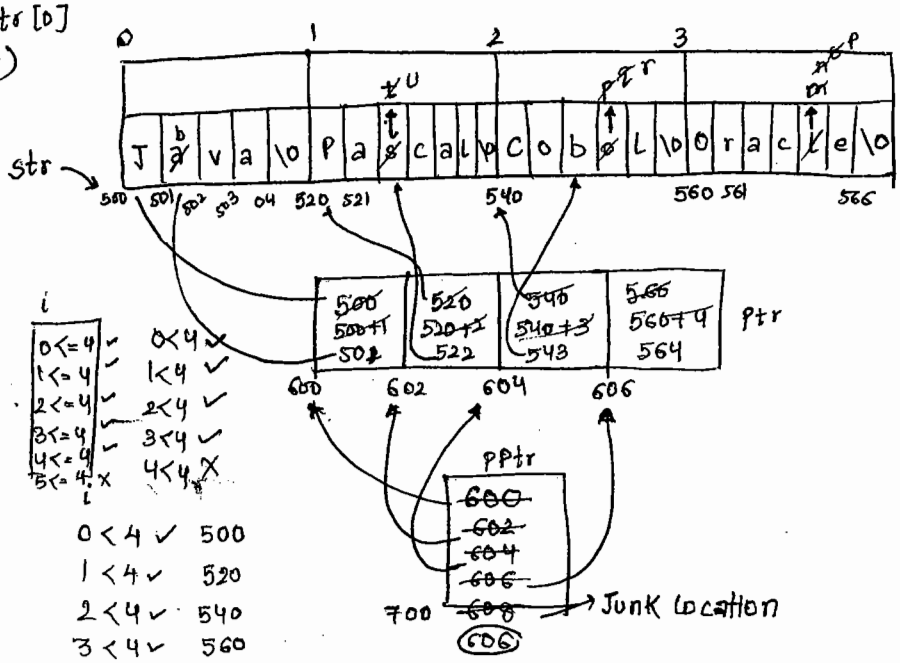
```
# include <stdio.h>
# include <conio.h>
# include <string.h>
int main()
{
   char str [50];
   char ch ;
   int i, count;
   clrscr();
   printf ("Enter a string: ");
   gets (str);
   for (ch='0' ; ch<='9'; ch++)    //for (ch = 'A'; ch<='Z'; ch++)
   {
      count = 0;
      for (i = 0; str[i] != '\0'; i++)
      {
         if (ch == str [i])   // if (ch == str [i] || ch == str [i] - 32)
            ++ count ;
      }
    if (count > 0)
    {
```

```
            printf("*");
            --count;
        }
    }
}
getch();
return 0;
}
```

O/P:

```
Enter a string: 345695425
2: *
3: *
4: **
5: * * *
6: *
9: *
```

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{ char str[4][20] = {
                      "Java",
                      "Pascal",
                      "Cobol",
                      "Oracle"
                    };

char *ptr[4]
char **pptr
int i;
clrscr();
ptr[0] = str;      //str[0];      //&str[0][0];
ptr[1] = str+1;    //str[1];      //&str[1][0];
ptr[2] = str+2;    //str[2];      //&str[2][0];
ptr[3] = str+3;    //str[3];      //&str[3][0];
pptr = ptr;        //&ptr[0]
for(i=1; i<=4; i++)
{
   *pptr++ = i;
   **pptr++ = i;
   ++pptr;
}
--pptr;
puts(*pptr);
[for(i=0; i<4; i++)
  puts(ptr[i]);]
[for(i=0; i<4; i++)
  puts(str+i);]
getch();
return 0;
}
```



i	
0<=4 ✓	0<4 ✓
1<=4 ✓	1<4 ✓
2<=4 ✓	2<4 ✓
3<=4 ✓	3<4 ✓
4<=4 X	4<4 X

i

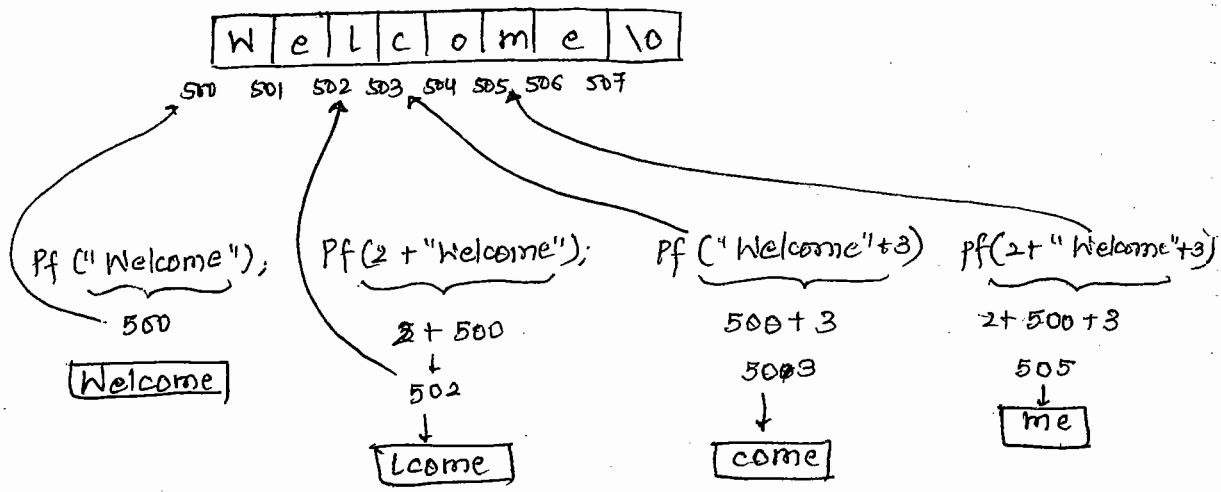
0 < 4 ✓ 500
1 < 4 ✓ 520
2 < 4 ✓ 540
3 < 4 ✓ 560
4 < 4 X false

INTERVIEW QUESTIONS

```
#include <stdio.h>
#include <conio.h>
int main()
{ printf ("Welcome");
  return 0;
}
```

O/P: Welcome

1. printf (" Welcome"); Welcome
2. printf (2+"Welcome"); Lcome
3. printf ("Welcome"+3); come
4. printf (2+ Welcome"+3); me




```
#include <stdio.h>
int main()
{ char str [] = "Welcome";
  printf ("str");
  return 0;
}
```

1. printf (str); Welcome
2. printf (2+str); Lcome
3. printf (str+3); come
4. printf (2+str+3); me

```
# include <stdio.h>
  int main()
{
  puts ("Welcome");
  return 0;
}
```

1. puts ("Welcome"); — Welcome
2. puts (2 + "Welcome"); — lcome
3. puts ("Welcome" + 3); — come
4. puts (2 + "Welcome" + 3); — me

```
# include <stdio.h>
  int main()
{  char str[] = "%d Hello %d";
   printf (str);
   return 0;
}
```

O/P: gr Hello gr

1. printf (str); — gr Hello gr → printf is formated
2. printf (str, 10, 20) — 10 Hello 20
3. puts (str); — %d Hello %d → puts is unformatted :. doesn't understand format specifiers
4. puts (str, 10, 20); — Error → puts takes only one argument

```
# include <stdio.h>
  int main()
{  printf ("Hai" "Bye");
   return 0;
}
```

O/P: Hai Bye — Printf internally concates 2 strings seperated with space

1. printf ("Hai" "Bye"); — HaiBye
2. printf ("Hai", "Bye"); — Hai — Here also we have 2 strings but ',' ∴ will treat as 1o ∴ only 1 string is printed
3. printf ("Hai %s", "Bye"); — Hai bye — 1st %s (in 1st argument) treated as format specifier
4. printf ("Hai %s", "Bye %s", "Hello"); — Hai Bye %s — In 2nd argument it is not treated as format specifier.
5. printf ("Hai %s %s", "Bye", "Hello"); — HaiByeHello; → The correct way

```
#include <stdio.h>
 intmain()
{ printf ("\n%s", "Hello");          // Hello
  printf ("\n%-3s","Hello");         // Hel
  printf ("\n%.3s", 2+"Hello");      // llo
}
```

```
#include <stdio.h>
  int main()
{ char str[]="Hello";
  printf ("\n%-3s", str);            // Hello
  printf ("\n%.3s", str);            // Hel
  printf ("\n%.3s",2+str);           // llo
  return 0;
}
```

```
#include <stdio.h>
  int main()
  {
   char str1[]
   char str2[]
   char temp[10];
   sprintf (temp, "%s%.3s", str1+3, str2);
   printf ("\n%s", temp);
   return 0;
  }
```

O/P: comeHel

STRINGZING OPERATOR (#)

This operator is introduced in NCC version. By using this operator, we can convert the text in the form of string i.e replacement in " ".

```
#include <stdio.h>
#define ABC (xy) printf(#xy "=%d", xy);
 int main()
{ int a, b;
  a = 10; b= 20;
  ABC (a+b)
  return 0;
}
```

O/P: a+b = 30

```
# define ABC(xy)    printf (#xy " = %d", xy);
                    ↑
ABC(a+b)
```

```
printf (# a+b  " = %d", a+b);
printf ("a+b" " = %d ", a+b);
printf ("a+b = %d", a+b);  → concatenation
```

o/p :- a+b = 30

TOKEN PASTE OPERATOR (##)

- Ncc Programming language supports this operator.
- By using this operator, we can concatenate multiple tokens.

```
# include <stdio.h>
# define ABC(x,y) printf ("%d", x##y) o isliye yahan nahi hai
void main()
{
   int var12 = 120;
   ABC (var, 12);
   return 0;
}
```

```
# define ABC (x,y)    printf ("%d ", x##y)
                      ↑
ABC (var, 12);
```

```
printf ("%d", var##12);
printf ("%d", var12);
```

o/p : 120

```
# include <stdio.h>
# define START  m##a##i##n
void START()
{  printf (" With out main");
}
```

o/p :- Without main

start is replaced with

```
void START()
   m##a## i##n
   ma##i##n
   mai##n
   main
```

29/7/2015.

MEMORY MANAGEMENT IN 'C'

In C Programming lang., we are having 2 types of m/m management

1. Static Memory management
2. Dynamic Memory Management

1) Static Memory Management

- When we are creating the memory at the time of compilation then it is called Static memory allocation or Compile-time m/m.
- Static memory Allocation is under control of compiler.
- When we are working with static memory Allocation, it is not possible to extend the memory at the time of execution, if it is not sufficient
- When we are working with static memory Allocation, we need to go for preallocation of memory i.e how many bytes of data ~~need~~ required to be created is needed to be decided using coding only

```
Ex: int a;          // 2B
    float f;        // 4B
    int arr[10];    // 20B
    char str[50];   // 50B
```

2) Dynamic Memory Management

- It is a procedure of allocating or deallocating the memory at run time i.e dynamically
- By using DMA, we can utilize the memory more efficienty according to the requirement
- By using DMA, whenever we want, which type we want & how much we want, that time, type and that much we can create dynamically.
- DMA related, all predefined functions are declared in <malloc.h>
  <alloc.h>
  <stdlib.h>

DMA related predefined functions are :-

1. malloc()
2. calloc()
3. realloc()
4. free()
5. farmalloc()
6. faralloc()
7. farrealloc()
8. farfree()

1) malloc():- By using this predefined function, we can create the memory dynamically at initial stage.

- malloc function require 1 argument of type size_type i-e, datatype size
- malloc() creates memory in bytes format and initial value is garbage.

Syntax: `void *malloc (size_type);`

NOTE: DMA related functions can be applied for any datatype thats why functions returns void* i.e. generic type.

- When we are working with DMA related functions, we are required to perform Type casting because functions returns void*.

```
Ex:- int *ptr;
     ptr = (int*) malloc (sizeof (int));          //2B

     float *ptr;
     ptr = (float*) malloc (sizeof (float));      //4B

     int *arr;
     arr = (int*) malloc (sizeof (int)*10);       //20B

     char *str;
     str = (char*) malloc (sizeof (char)*50);     //50B
```

2) free():- By using this predefined function, we can deallocate dynamically allocated memory.

- When we are working with DMA related memory, it stores in heap area of data segment and it is permanent memory, if we are not deallocating
- When we are working with DMA related programs, at end of the program, recommended to deallocate memory by using free() function.
- free() function requires 1 argument of type (void*) & returns void type

Syntax: `void free (void *ptr);`

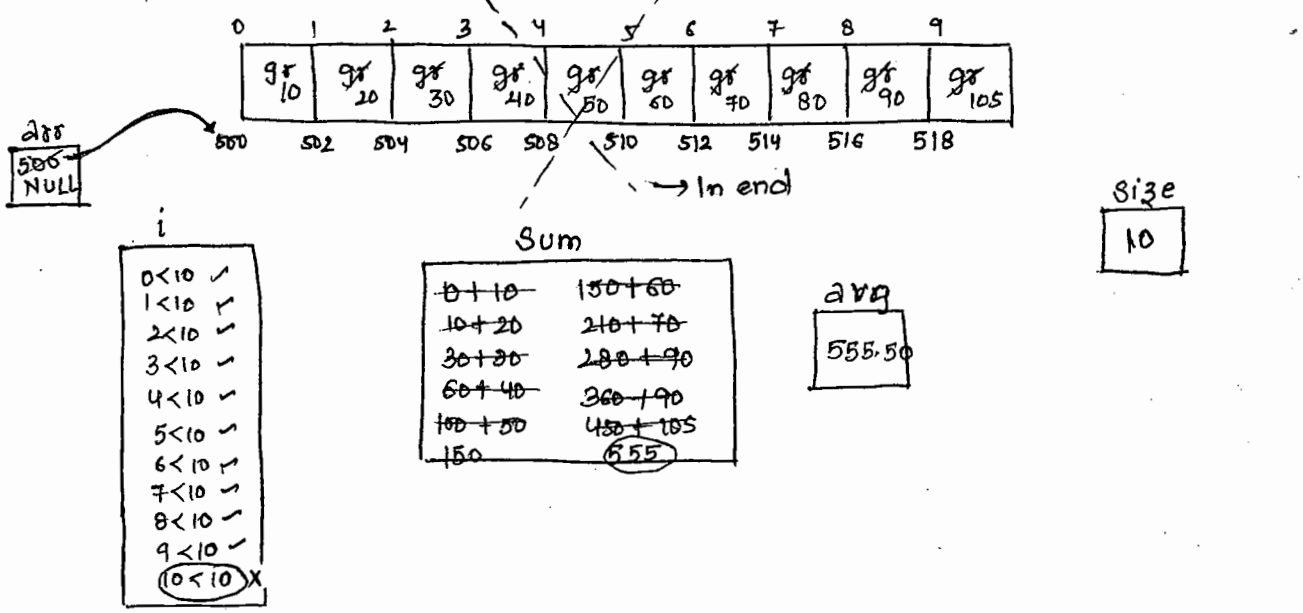
```
#include<stdio.h>
#include <conio.h>
#include <malloc.h>
int main()
{
    int *arr;
    int sum =0, i, size;
    float avg;
    clrscr();
    printf("\n Enter array size");
```

```
scanf("%d", &size);
print arr = (int *) malloc (sizeof(int) * size);
printf("\n Default values: ");
for (i=0; i<size, i++)
printf ("%d", arr[i]);

print("\n Enter %d values : ", size);
for (i=0 ; i<size ; i++)
{
  scanf ("%d", &arr[i]
  sum += arr[i]      //Sum = sum + * (arr + i);
}
  avg = (float) sum/size ;
  printf ("\n Sum of list : %d", sum);
  printf ("\n Avg of list : %d %.2f", avg);
  getch();
  return 0;
}
```

```
free(arr);
arr = NULL;
```

O/P:

```
Enter array size : 10
Default values : gr gr gr gr gr....
Enter 10 values : 10 20 30 40 50 60 70 80 90 105
Sum of list :  555
Avg of list :  555.50
```



3) calloc() By using this predefined function, we can create the memory dynamically at initial stage.

→ calloc() requires 2 argument of type (count, size_type)

→ count will provide no. of elements, size_type is datatype size

→ When we are working will calloc() function, it creates the memory in block format & initial value is zero

Syntax: `void* calloc(count, size_type)`

Eg:

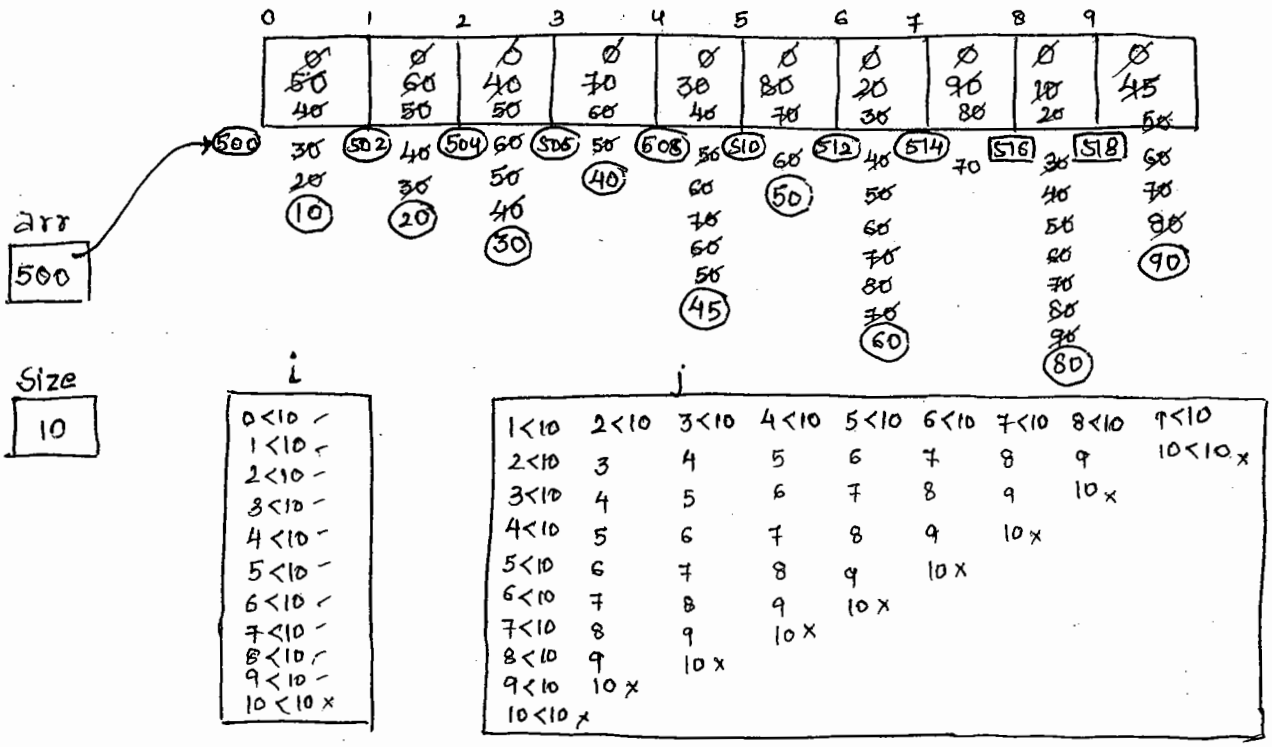
```
int *arr;
arr = (int*) calloc (10, sizeof (int));     //20B

char * str;
str = (char*) calloc (50, size of char));   //50B
```

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>
int main()
{
  int * arr;
  int size, i, j, t;
  clrscr();
  printf("Enter array size: ");
  scanf("%d", size);
  arr = (int*)calloc (size, sizeof (int));

  printf("\n Default values : ");
  for (i=0; i<size ; i++)
  printf ("%d", arr [i]);

  printf("\n Enter %d values: ", size);
  for (i=0; i< size ; i++)
  printf scanf (" %d", &arr [i]);

  for (i=0; i<size; i++)
   {
    for (j= i+1; j<size ; j++)
     {
      if (arr [j] < arr [i])
       {
        t = arr [i];
        arr [i] = arr [j];
       } arr [j] = t;
     }
}
```

```
printf("\n Sorted arr List:");
for(i=0; i<size, i++);
 { printf("%d", arr[i]);
  free(arr);
  arr = NULL;
  getch();
  return 0;
}
```

O/P:

```
Enter array size: 10
Default values: 0 0 0 0 0 0 0 0 0 0
Enter 10 values: 50 60 40 70 30 80 20 90 10 45
Sorted arr List:
10 20 30 40 45 50 60 70 80 90
```



4) realloc() :- By using this predefined function, we can create the memory dynamically at middle stage of the program.

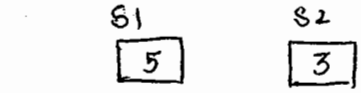
- Generally this function is required to use when we are reallocating memory.
- realloc() requires 2 argument of type void*, size_type
- void* indicates previous block base address, size_type is datatype size
- When we are working with realloc() function, it creates the memory in bytes format and initial value is garbage.

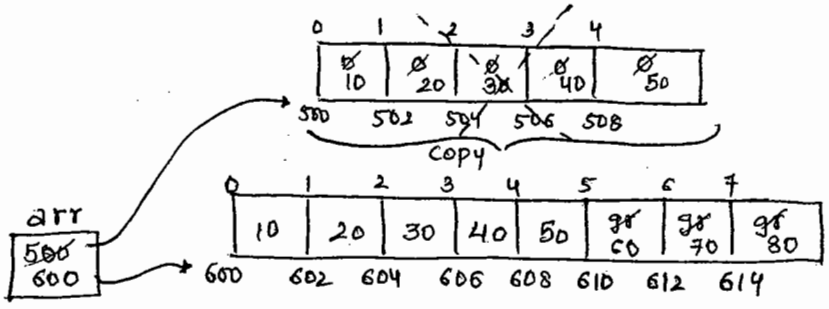
Syntax: `void * realloc(void*, size_type)`

Eg:-

```
int *arr;
arr = (int *) calloc (5, sizeof(int));      //10B
----
----
arr = (int*) realloc (arr, sizeof(int)* 10);   //20B
```

```c
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
int main()
{
  int *arr;
  int s1, s2, i;
  clrscr();
  printf("Enter array size1: ");
  scanf("%d", &s1);
  arr = (int *) calloc (s1, sizeof(int));
  printf("\n Enter %d values : ", s1);
  for(i=0; i<s1; i++),
  scanf("%d ", &arr[i]);
  printf("\n Enter array size2: ");
  scanf("%d", &s2);
  arr = (int *) realloc (arr, sizeof(int)*(s1+s2));
  printf("\n Enter %d values", s2);
  for(i=s1; i<s1+s2; i++)
  scanf("%d", &arr[i]);
  printf("\n Arr Data list: ");
  for(i=0; i<s1+s2; i++)
  printf("%d", arr[i]);
  free(arr);
  arr = NULL;
  getch();
  return 0;
}
```





O/P:

```
Enter array size1: 5
Enter 5 values: 10 20 30 40 50
Enter array size2: 3
Enter 3 values: 60 70 80
Arr Data list: 10 20 30 40 50 60 70 80
```

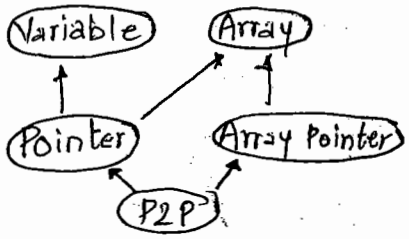
2D ARRAY DYNAMIC CREATION

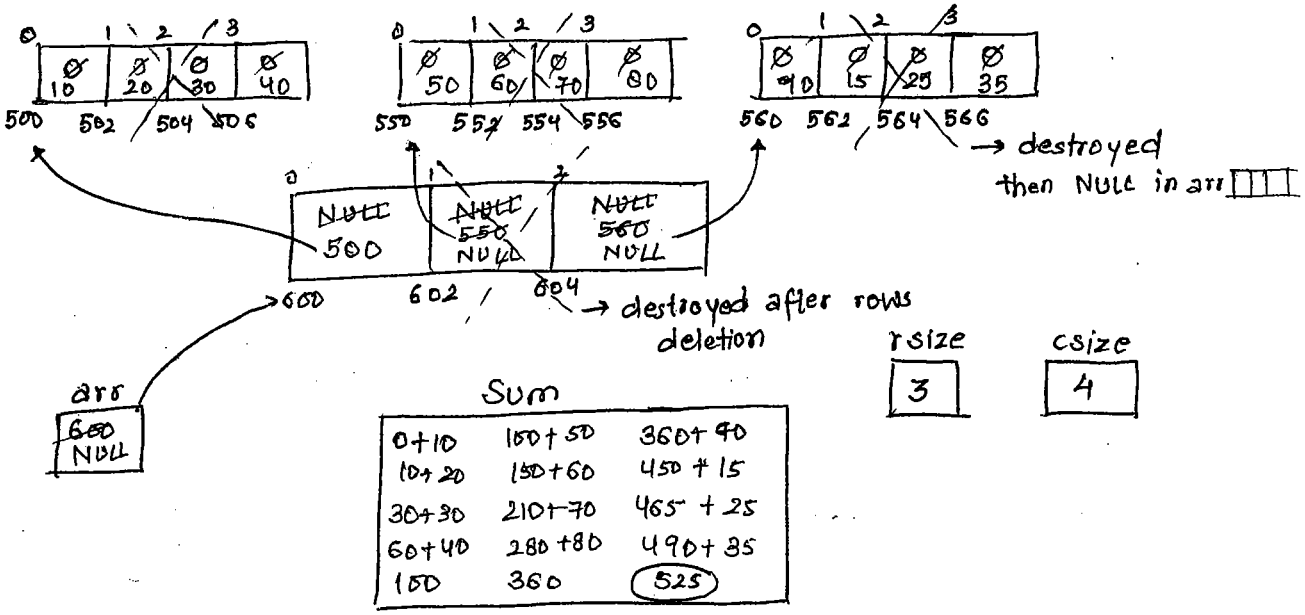
- To develop 2D Array dynamically, we are required to take pointer to pointer variable then 1 array is required to create, to manage multiple rows.

```
#include <stdio.h>
# include <conio.h>
# include <malloc.h>
int main()
{
    int **arr;
    int rsize, csize;
    int sum = 0, r,c;
    clrscr();
    printf("Enter row size: ");
    scanf("%d", &rsize);
    arr = (int**) calloc (rsize, sizeof (int*));
    printf("Enter column size: ");
    scanf("%d", &csize);
    for (r=0; r < rsize; r++)
    arr[r] = (int*) calloc (csize, sizeof (int));
    printf("Enter %d * %d values: ", rsize, csize);
    for (r=0; r< rsize; r++)
    for (c=0; c < csize; c++)
    scanf("%d", &arr[r][c]);

    for (r=0; r <rsize ; r++)
    {
      for (c= 0; c < csize; c++)
        {
          sum += arr[r][c];      // sum = sum+*(*(arr+r)+c),
        }
    }
    printf("\nsum value is : %d", sum),
    for (r=0; r< rsize; r++)
    {
      free (arr[r];
      arr[r] = NULL;
    }
      free (arr);
      arr = NULL;
      getch();
      return 0;
}
```

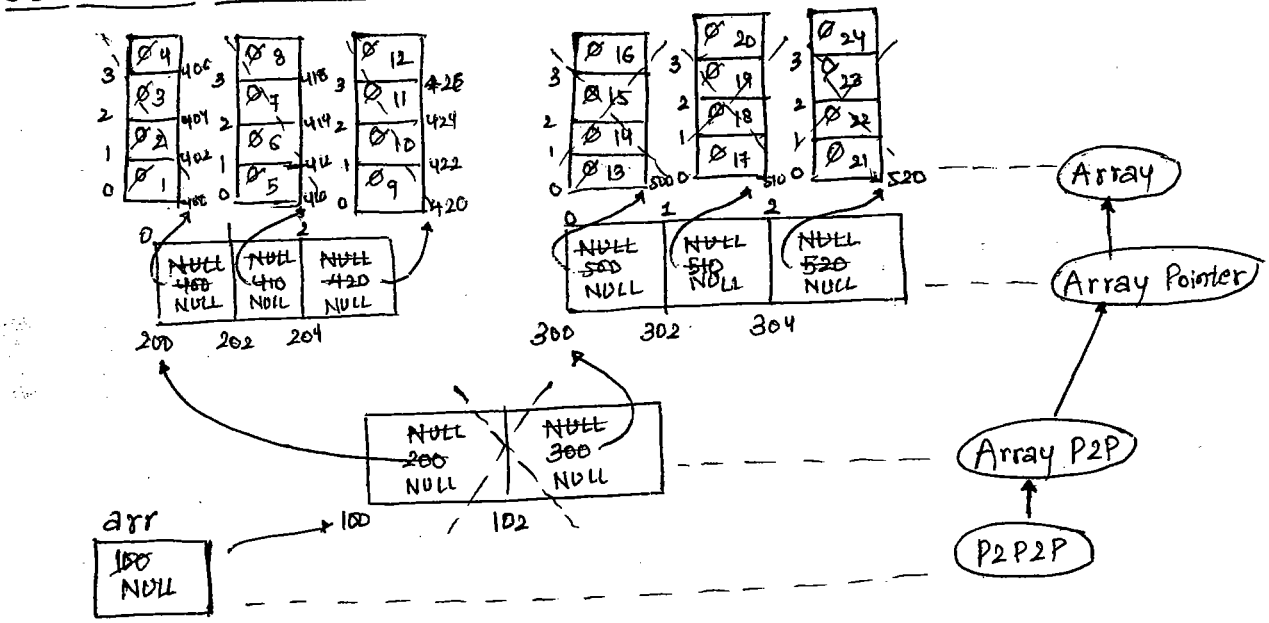
O/P :-

```
Enter row size: 3
Enter column size: 4
Enter 3 * 4 values :
   10  20  30  40
   50  60  70  80
   90  12  25  35
Sum value is : 522
```



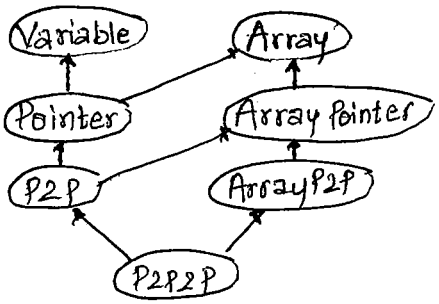


3D ARRAY DYNAMIC CREATION



Destroyed in reverse order

Array → Array Pointer → Array P2P → P2P2P



bsize
2
rsize
3
csize
4

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
int main()
{
  int ***arr;
  int bsize, rsize, csize;
  int b,r,c;
  clrscr();
  printf("Enter block size");
  scanf("%d", bsize);
  arr = (int ***)calloc(bsize, sizeof(int**));
  printf("Enter row size:");
  scanf("%d", &rsize);
  for(b=0; b< bsize; b++)
  arr[b] = (int **) calloc (rsize, sizeof(int*));
  printf("Enter column size:");
  scanf("%d", &csize);
  for(b=0; b< bsize, b++)
  for(r= 0; r<rsize; r++)
  arr[b][r]= (int *) calloc(csize, sizeof(int));
  printf("Enter %d*%d*%d values:", bsize, rsize, csize);
  for(b=0; b<bsize; b++)
  for(r=0; r<rsize; r++)
  for(c=0; c<csize; c++)
  scanf("%d", &arr[b][r][c]);
  printf("\n Arr Data List:");
  for(b=0; b<bsize; b++)
  {
    printf("\n Block: %d", b+1);
    for(r=0; r<rsize; r++)
    {
      printf("\n");
      for(c=0; c<csize; c++)
      printf("%3d", arr[b][r][c]);
                  // *(*(*(arr+b)+r)+c);
    }
}
```

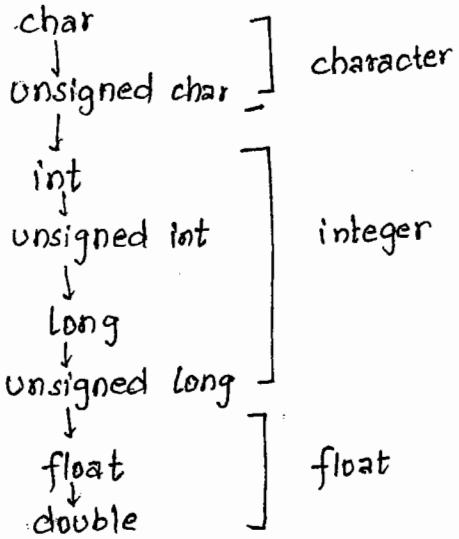
```
    for (b= 0; b<bsize; b++)
     {
      for (r=0; r< rsize; r++)
        {
          free (arr [b][r]);
          arr[b] [r] = NULL;
        }
          free (arr [b])
          arr [b] = NULL;
      }
    free(arr);
    arr = NULL;
    getch C);
    return 0;
}
```

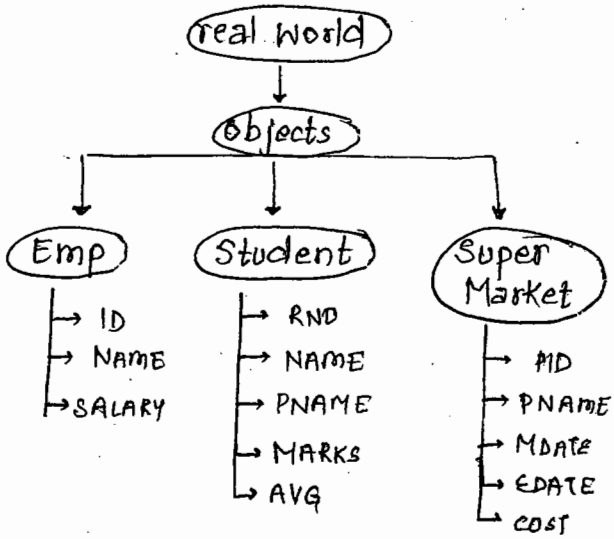
O/P:

```
Enter block size: 2
Enter row size : 3
Enter column size : 4
Enter 2*3*4 values:
1 2 3 4 5 6 7 8 9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24

Arr Data List :
Block: 1
      1  2  3  4
      5  6  7  8
      9  10 11 12
Block: 2
      13 14 15 16
      17 18 19 20
      21 22 23 24
```

STRUCTURES

- In 'C' programming language, we are having 3 types of datatypes
  1. Primitive Datatype
  2. Derived Datatype
  3. User-defined Datatype
- All primitive datatypes are used to manipulate basic data types i.e char, int, float
- All derived datatypes works for primitive datatypes.
- In real world, every information will be there in the form of objects.
- Every object having their own properties and behaviour.
- No any primitive or derived datatypes support real time object information.
- When the primitive or derived datatypes are not supporting user requirement then go for user-defined datatypes.





- A structure is a collection of different types of data elements in a single entity.
- A structure is a combination of primitive and derived datatype variables.
- By using structures we can create user defined datatypes
- Sizeof the structure is sum of all member variable sizes.
- Least size of structure is 1B
- In 'C' programming language, it is not possible to create empty structure.

- C language structure contain data members only but in C++, data members and member functions.
- Syntax:

```
struct tagname
{
 Datatype1  mem1;
 Datatype 2  mem2;
 Datatype 3  mem3;
 ----
 ----
};
```

- According to syntax of the structure, semicolon must be required at end of the structure body.

Ex1:
```
struct emp
{
 int id;
 char name[36];
 int sal;
};
```
sizeof(struct emp) ----> 40B (2+36+2)

Ex2:
```
struct student
{
 int rno;
 char sname[20];
 char fname[20];
 char sname[20];
 int marks[6];
 int tmarks;
 float avg;
};
```
sizeof(struct student) ----> 80B (2+20+20+20+12+6)

Syntax to create Structure variable.

`struct tagname variable;`

Ex1:
```
struct emp
{
 int id;
 char name[36];
 int sal;
};
```

```
void main()
{
  emp e1;
 struct emp e1;
}
```

⇒
```
struct emp
 {
  int id;
  char name [36];
  int sal;
 }
  e1, e2;
 void main()
  {
   struct emp e3, e4;
  }
```

sizeof (struct emp) ---→ 40B
sizeof (e1) ⟶ 40B
sizeof (e2) ⟶ 40B

- When we are creating the structure variable at end of the structure body, then it becomes global variable i.e. e1, e2
- When we are creating the structure variable within the body of the function, then it is auto variable which is local to specific function i.e e3, e4.

Syntax to create structure type pointer :-

```
Struct tagname *ptr;
```

Eg :- on next page

Ex:
```
struct emp
{
  int id;
  char name[36];
  int sal;
} e1, *ptr1;
void main()
{
 struct emp e2;
 struct emp *ptr2;
  ptr1 = &e1;
  ptr2 = &e2;
}
 // size of(e1) ---> 40B
 // size of (ptr1) --> 2B
```

Size of user defined pointer is 2B only because it holds address.

Syntax to create structure type array :-

```
struct tagname arr[SIZE];
```

Ex-
```
struct emp
  {
    int id;
    char name[36];
    int sal;
  };
 void main()
   { struct emp arr[10];
  // size ---> 10
  // sizeof (arr) ---> 400B(10 * 8B = 400B)
 }
```

CREATING THE ARRAY DYNAMICALLY

```
struct emp
{
  int id;
  char name[36];
  int sal;
};
 void main()
{
 struct emp * arr;
  arr = (struct emp *) calloc(10, sizeof(struct emp));
```

```
    // size 10
    // size of (arr) ⟶ 28
    free(arr);
}
```

Syntax to Initialize Structure Variable

```
struct tagname variable = {value1, value2, value3, ...};
```

Ex-
```
struct emp
{
 int id;
 char name[36];
 int sal;
} e1 = {101, "Raj", 125000}, ~~125000~~ e2 = {102, "Teja"};
void main()
 {
   struct emp e3 = {103};
 }
```

- In initialisation of structure variable, if specific no of members are not initialized, then remaining all members are initialised with 0 or nul.
- If value type member is not initialised, then it becomes 0, if string type data is not initialised, then it becomes null.

Syntax to access Structure Members :-

By using following operaters we can access structure members :-

1. struct to member
2. pointer to member

→ If variable is normal operator struct to member operator.
→ If variable is pointer type, then go for pointer to member operator.

Syntax :-

```
struct tagname variable;
Variable.member = value;
        (or)
variable.member;

struct tagname variable;
Struct tagname * ptr;
ptr = &variable;
ptr → member = value;
        or
ptr → member;
```

```
struct emp.
{
  int id;
  char name [36];
  int sal;
};
 e1 = {101, "Raj", 12500}, e2 = {102, "Teja"};
void main()
 { struct emp e3, e4, e5, e6;
   e3.id = 103;
   e3.name = Rajesh;      Error
   e3.name = "Rajesh";    Error  ∴ left side string, right side string
                                 it is not possible to assign 1 string
correct                          value to other string
way  strcpy (e3.name, "Rajesh");
   e3.sal = 14000;
   e4 = e3 + 1;          Error
   e4 = e3.id+1;         Error
   e4.id = e3.id+1;      yes

   e4.name = e3.name;    Error
   strcpy(e4.name, e3.name);

   e4.sal = e1+e2;       Error
   e4.sal = e1.sal + e2.sal;  yes

   e5 = e4;  ✓
   We can assign 1 structure variable to another structure
   variable of same type
   e4 == e5;     Error
   e4.id = e5.id;    yes
   e3.name > e4.name   Error
   strcmp(e3.name, e4.name);
   e3.sal < e2.sal     // yes
}
```

→ Any kind of manipulations can be performed on structure members.
→ Except the assignment, no any other operations can be performed on structure variables.
→ When 2 variables are same structure type then it is is possible to assign 1 variable data to another variable.

```
struct emp;
{
  int id;
  char name[36];
  int sal;
};
  void main()
  {
   struct emp e1;
   struct emp* ptr;
   ptr = &e1;

   // e1.id = 101;
    ptr → id = 101;

    strcpy (ptr → name, "Rajesh");
   // strcpy (e1.name, "Rajesh");

    // e1.sal = 12500;
    ptr → sal = 12500;
  }
```

typedef :-

→ It is a keyword, by using this keyword, we can create user-defined name for existing data type.

→ Generally typedef keyword used to create an alias name for existing datatype.

Syntax: `typedef Datatype user_defined_name`

```
# include <stdio.h>
# include <conio.h>
type int myint;  // myint → user defined name . ∴ alias name of integer
int main()                                      data type
{
    int x;
    myint y;
    typedef myint smallint;  // smallint is an alias name to myint
    smallint z;
    clrscr();
    printf (" Enter 2 values:");
    scanf (%d%d, &x, &y);
    z = x+y;
    printf ("Sum value is : %d", z);
    getch();
    return 0;
}
```

O/P:

```
Enter 2 values : 10 20
Sum value is : 30
```

```
#include<stdio.h>
#include<conio.h>
#define MYCHAR char    // MYCHAR is identifier which is replaced with char
typedef char BYTE;
int main()
{
  char ch1='A';
  MYCHAR ch2='b';    // char ch2='b';
  BYTE ch3;
  ch3=ch2-ch1+20;    // ch3=98-65+20; 53 (it corressponding data is 5)
  printf("char1: %c char2: %c char3:%c",ch1,ch2,ch3);
  getch();
  return 0;
}
```

O/P: char1: A char2: b char3: 5

- By using #define, we can't create alias name because at the time of preprocessing, identifier is replaced with replacement text
- #define is under control of preprocessor, typedef is under control of compiler

INTERVIEW QUESTIONS

1)
```
struct
{
  int id;
  char name[36];
  int sal;
};
```
Valid

- When we are working with structures, mentioning the tagname is optional, if tagname is not given, then compiler creates nameless structure.
- When we are working with nameless structures, it is not possible to create structures variable whether the body of the function, i.e global variables are possible to create.

2)
```
struct
{ int id;
  char name[36];
  int sal;
} e1, e2, e3;
Void main()
{ struct tagname Yes valid
}
```

* type def struct

```
{
 int id;
 char name [10]
  int sal;
} EMP;
 void main()
{
 Emp. e1;          Valid
}
```

In previous syntax, 1st compiler creates a nameless structure, then it gives an alias name called EMP

* typedef struct emp

```
{
  int id;
  char name [36];
  int sal;
} EMP;
 Void main()
 {
   struct emp e1;
     Emp e2;          Valid
 }
```

In previous syntax, 1st compiler creates a user-defined datatype called struct emp, then it gives an alias name called EMP

→ struct emp

```
{
  int id;
  char name[36];
  int sal;
} e1, e2, e3;
```

What is e1, e2, e3 ? Global variables

→ typedef struct

```
{
  int id;
  char name [36];
  int sal;
} EMP e1, e2, e3;
```

Error

typedef struct emp

```
{
  int id;
  char name [36];
  int sal;
} EMP e1, e2, e3;
```

Error

- When the structure body is started with typedef keyword, then it is not possible to create structure variable at end of the body i.e global variables are not possible to create.

```
typedef struct
{ int id;
  char name[36];
  int sal;
} Emp, e1, e2, e3;
```

Yes valid, EMP e1, e2, e3, alias name.

→
```
typedef struct emp
{ int id;
  char name[36];
  int sal;
} EMP, e1, e2, e3;
```

What is EMP, e1, e2, e3 ? Alias names

→
```
struct emp
{
};   Not valid, Error
```

→ In 'C' programming language it is not possible to create empty structure because least size of structure is 1 byte.

→
```
struct emp
{
  int id = 101;
  char name[36] = "Payal";
  int sal = 12500;
};   Not Valid, Error
```

- For creation of structure, it doesn't occupy any physical memory.
- When we are working with the structure, physical m/m will occupy.
- When we are creating variable but for initialization of member we required physical m/m.

---

* Self referential structure - placing a structure type pointer has a member to same structure is called self-referential structure. By using left referential structure, we can handle any type of data structure.

Ex-
```
struct emp
{ int id
  char name[36];
  int sal;
  struct emp* ptr;
};
  // Size of (struct emp) → 42B
```

→
```
struct emp
{
  int id;
  char name[36];
  int sal;
};
void abc()
{
  struct emp e1, e2;
}
void main()
{
  struct emp e1, e2;
}
```
Valid

When we are creating the structure in global scope, then it is possible to access within the program in any function.

→
```
void main()
{
  struct emp
  {
    int id;
    char name[36];
    int sal;
  };
  struct emp e1, e2;
}
void abc()
{
  struct emp e1, e2;
}
```
Is it valid?
No, Error

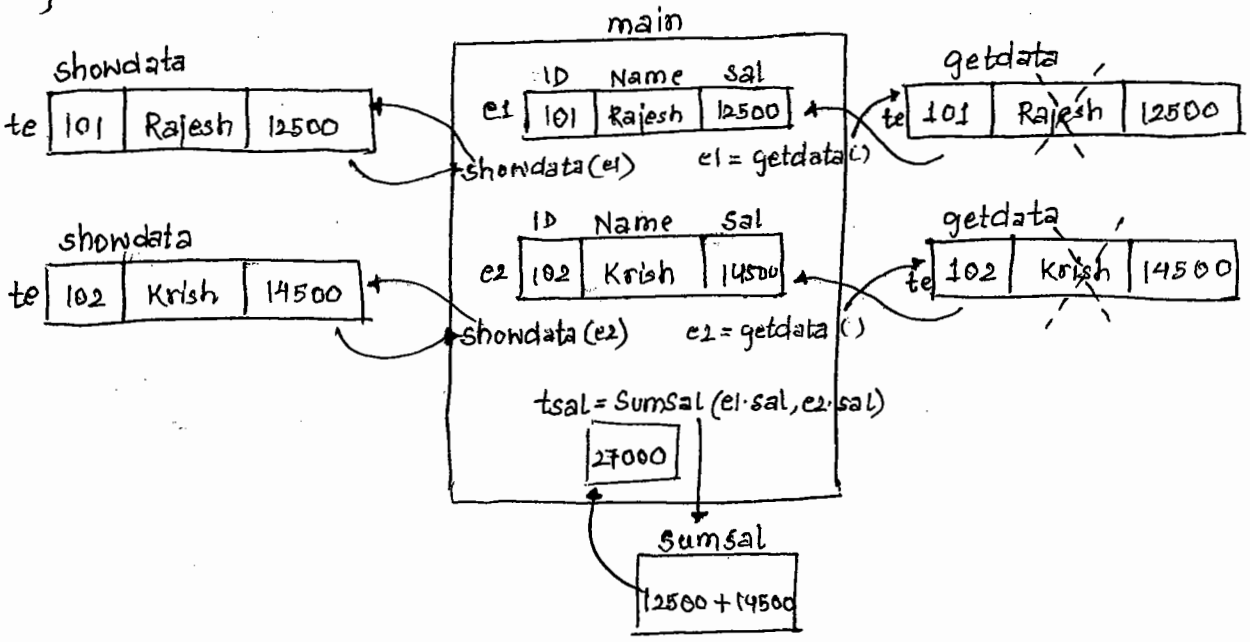
When the structure is created within the body of the function, then that structure need to be accessed in same function only.

PROG :-
```
#include <stdio.h>
#include <conio.h>
typedef struct
{
  int id;
  char name[36];
  int sal;
} EMP;
EMP getdata()
{
```

```
    EMP te;
    printf("\n ENTER EMP ID:");
    scanf("%d, &te.id);
    printf("\n ENTER EMP NAME:");
    fflush(stdin);
    gets(te.name);
    printf("\nENTER EMP SALARY:");
    scanf("%d", &te.sal);
    return te;
}
void showdata(EMP te)
{
    printf("\n ID: %d  NAME: %s  SALARY= %d", te.id, te.name, te.sal);
}
int sumsal(int s1, int s2)
{
    return(s1+s2);
}
int main()
{
    EMP e1, e2;
    int tsal;
    e1 = getdata();
    e2 = getdata();
    showdata(e1);
    showdata(e2);
    tsal= sumsal(e1.sal, e2.sal);
    printf("\n Sum Salary = %d", tsal);
    return EXIT_SUCCESS;
}
```

O/P:

```
ENTER EMP ID : 101
ENTER EMP NAME: Rajesh
ENTER EMP SALARY : 12500
ENTER EMP ID: 101
ENTER EMP NAME: Krish
ENTER EMP SALARY: 14500
ID: 101 Name: Rajesh sal: 12500
ID: 102 Name: Krish  sal: 14500
Sum Salary : 27000
```



PROG:- DISPLAYING DATA IN ALPHABETICAL ORDER

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 5
struct emp
{
  int id;
  char name[36];
  int sal;
};
struct emp getdata()
{ struct emp te;
  printf("\n ENTER ID:");
  scanf("%d", &te.id);
  printf("\n ENTER NAME:");
  fflush(stdin);
  gets(te.name);
  printf("\n ENTER SAL:");
  scanf("%d", &te.sal);
  return te;
}
void showdata(struct emp te)
{
  printf("\n id: %3d  Name: %5s  SAL: %3d", te.id, te.name, te.sal);
}
int main()
{
  struct emp arr[SIZE];
  struct emp te;
  int i, j;
  printf("\nEnter %d emp's data:", SIZE);
  for(i=0; i<SIZE; i++)
    arr[i] = getdata();
  for(i=0; i<SIZE; i++)
  {
    for(j=i+1; j<SIZE; j++)
    {
      if(stricmp(arr[i].name, arr[j].name)>0)
        // if(arr[i].sal > arr[j].sal) Ascending
        // if(arr[j].id > arr[j].id) Descending
```

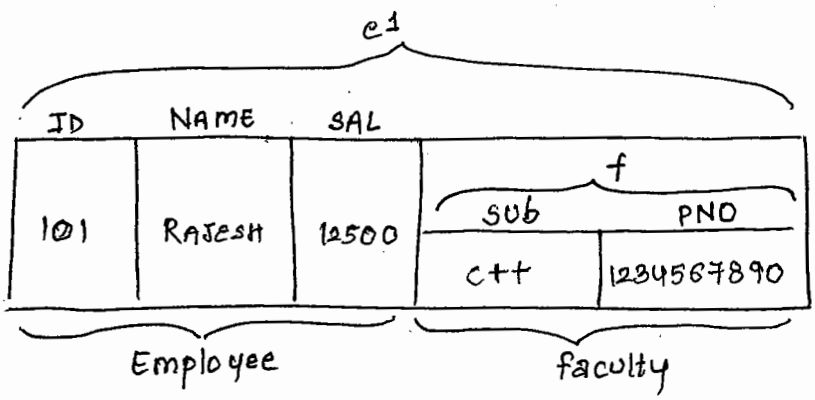
```
        {
          te = arr[i];
          arr[i] = arr[j];
          arr[j] = te;
        }
      }
    }
        printf("\n Sorted emp data:");
        for (i=0; i< SIZE ; i++).
            showdata (arr [i]);
          return EXIT_SUCCESS;
    }
```

NESTED STRUCTURE

- It is a procedure of placing a structure within an existing structure body.
- When we are working with nested structure, size of the structure is sum of inner structure properties and outer structure properties is to be calculated.
- When we are working with nested structure, it is not possible to access inner structure members directly by using outer structure variable.
- In order to access inner structure ~~variable~~ members by using outer structure variable we are required to create inner structure variable within the body only.
- Inner structure variable doesnot have to access outer structure members directly or indirectly

Ex-
```
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct emp
{ int id;
  char name[30];
  int sal;
    struct faculty
    {
      char sub [20];
      char pno [10];
    }
};
void main ()
{
  struct emp e1;        //sizeof(e1) ----> 108 (40+30)
```

```
struct faculty f1;        // sizeof(f1) ----> 30B
clrscr();
e1.id = 101;
strcpy(e1.name, "Rajesh");
e1.sal = 12500;
//strcpy(e1.sub, "c++");  Error
strcpy(e1.f.sub, "c++");
strcpy(e1.f.pno, "1234567890");
printf("\n ID: %d NAME: %s SAL: %d SUB: %s PNO: %s", e1.id, e1.name,
        e1.sal, e1.f.sub, e1.f.pno);
getch();
return 0;
}
```



CONTAINERSHIP OR COMPOSITION :-

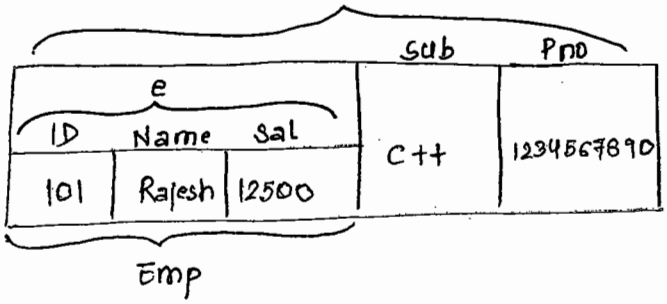
→ It is a procedure of placing a structure variable as a member to another structure.

→ Containership will occupy less memory when we are comparing it with nested structure.

→ Containership looks like inheritance in OOP language.

Eg:-
```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct emp
 {
   int id;
   char name[30];
   int sal;
 };
struct faculty
 {
   struct emp e
   char sub[20];
   char pno[10];
 };
```

```
void main()
{
  struct emp e1;        //size of (e1) ----> 40B
  struct faculty f1;    //size of (f1) ----> 70B (40+30)
  clrscr();
  f1.e.id = 101;
  strcpy (f1.e.name, "Rajesh");
  f1.e.sal = 12500;

  strcpy (f1.sub, "C++");
  Strcpy (f1.pno, "1234567890");
  printf ("\n ID: %d NAME: %s SAL: %d SUB: %s PNO: %s", f
          f1.e.id, f1.e.name, f1.e.sal, f1.sub, f1.pno);

  getch();
  return 0;
}
```



2/8/2015.

UNION

→ A Union is a collection of different types of dataelements in a single entity.

→ It is a combination of primitive and derived datatype variables.

→ By using union, we can create user-defined data type elements.

→ Size of a union is max. size of a member variable.

→ In implementation, for manipulation of the data, if we are using only one member then it is recommended to go for union.

→ When we are working with unions, all member variables will share same m/m location.

→ By using union, when we are manipulating multiple members then actual data is lost.

Syntax:
```
union tagname
{
 Datatype1 mem1;
 Datatype2 mem2;
 Datatype3 mem3;
 ----
 ----
};
```

Ex-1
```
union abc
{
 int i;
 float f;
 char cj;
};
// sizeof (Union abc)
      ↳ 4B
```

Ex2:
```
union emp
{
 int id;
 char name [36],
 int sal;
};
sizeof (union emp)
      ↳ 36B
```

- All the properties of structures are applicable to union also like variable creation, pointer creation, array creation, typedef approach,
- The basic difference b/w structure and union is

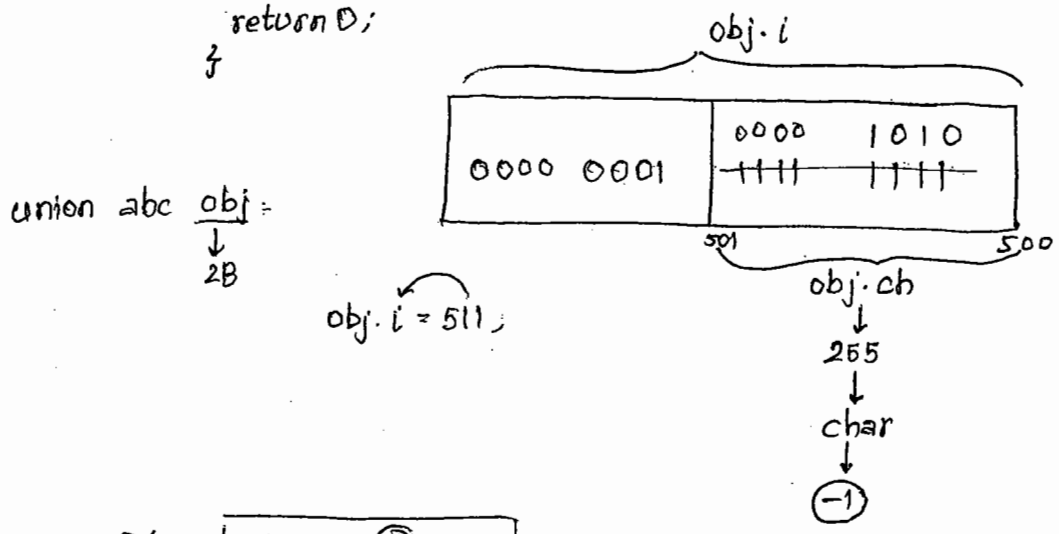
1. Size - structure size is sum of all member variable size
   Union size is max. size of a member variable

2. Memory Allocation - In structure, memory will be allocated for all the members but
   In Union, memory will be allocated for one member which occupies max. size.

3. Data Manipulation - In structure, data can be manipulated on multiple members properly without losing the content but
   - In union if we are manipulating multiple members then actual data is lost.

4. Initialisation - When we are working with structures, multiple members can be initialised but
   - In union only 1 member required to be initialised.

Ex-
```
#include <stdio.h>
#include <conio.h>
union abc
{
 int i;
 char ch;
};
int main()
{
 union abc obj;
 clrscr();
 obj.i = 511;
 printf("\n%d %d", obj.i, obj.ch);
 obj.ch = 10;
 printf("\n%d %d", obj.i, obj.ch);
 getch();
 return 0;
}
```



O/P:-

511	-1 → Illegal
266 → Illegal	10

ENUM

- enum is a Keyword, by using this Keyword we can create sequence of integer constant value.
- Generally by using enum, we can create userdefined datatype of integer
- Size of enumerator datatype is 2B & the range from -32768 to 32767
- It is possible to change enum related variable value but it is not possible to change enum constant data

Syntax :- `enum tagname { const1 = value, const2 = value, const3 = value };`

- Acc. to syntax, if const1 value is not initialised, then by default sequence will start from 0 and next generated value is prev. const value + 1.

→
```
#include<stdio.h>
enum ABC {A,B,C};
int main()
{
  enum ABC x;
    x = A+B+C;   // x = 0+1+2;
  printf("\n x = %d",x);
  printf("\n %d %d %d", A,B,C);
  return 0;
}
```

O/P :
```
x = 3
0 1 2
```

→
```
#include <stdio.h>
#include <conio.h>
enum month {Jan=1, feb, Mar};
int main()
{
  enum month Aprl;
  Aprl = Mar+1;
  printf("\n%d %d", feb, Aprl);
  getch();
  return 0;
}
```

O/P: 2 4

`++ Feb;` Error

`++ Aprl;` Yes

→ In previous program, enum month is an user defined data type of int, Aprl is variable of type enum month

→ enum related constant values are not allows to modify, enum related variables are possible to modify.

```
#include <stdio.h>
#include <conio.h>
typedef enum ABC {A=40, B, X=50, Y} XYZ;
int main()                                  ↳ alias name of ABC
{
 enum ABC c;
  XYZ z;
 c = B+1;  // c = 41+1;
 z = Y+1;  // z = 51+1
 printf("\nc=%d  z=%d", c, z);
 printf("\n B=%d  Y=%d", B, Y);
 getch();
 return 0;
}
```

O/P:

C = 42	Z = 52
B = 41	Y = 51

In previous program enum ABC is an user-defined data type of integer, XYZ is an alias name to enum ABC

FILE OPERATIONS

- A file is a name of physical memory location in secondary storage area.
- File contains sequence of bytes of data in secondary storage area in the form of unstructured manner.
- In implementation, When we required to interact with secondary storage area, then recommended to go for file Operations.
- By using files, primary m/m related data can be send to secondary storage area & secondary storage area info can be loaded to primary memory.
- In 'C' programming lang., IO Operations are classified into 2 types -
  1. standard I/O operations
  2. secondary I/O operations.
- When we are interacting with primary I/O devices, then it is called standard I/O operations.
- When we are interacting with secondary I/O devices, then it is called secondary I/O operations
- Standard I/O related and Secondary I/O related, all predefined functions are declared in stdio.h only.
- file operations related specific functions are -

<STDIO.H>

Function			
fopen	fclose	fprintf	fputs
fscanf	fgetc	fgets	fgetchar
fread	fwrite	flushall	ftell
feof	fseek	remove	rename
rewind	fflush		

Constant	datatypes	global variables
EOF	FILE	stdin
NULL		stdout
SEEK_CUR		stderr
SEEK_END		stdprn
SEEK_SET		

→
```
#include <stdio.h>
# include<dos.h>
# include <stdlib.h>
int main()
{
  FILE *fp;
  //system("CLS"); //clrscr();
fp=fopen("E:\\1.txt","W");   → Exception rised
  if (fp==NULL)
   {
     printf("\n UNABLE TO CREATE FILE");
     // system("PAUSE"); // getch();
     return EXIT_FAILURE;
   }
  fprintf(fp,"Welcome");
  fprintf(fp,"\n%d NARESH IT%f",100,12.50);
  fclose(fp);
  return EXIT_SUCCESS;
}
```

When we are opening file & parameters must be passed 1. path 2. mode

W → Write

file may open or may not open
if open then it returns ADDRESS
if not then it returns NULL.

O/P:
```
Welcome
100 Naresh 12.50
```

Prog2:- For creating ASCII text file

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
   FILE *fp;
   char path[50];
   int i;
   printf("Enter a file path:");
   gets(path);
   fp = fopen(path,"W");
   if (fp == NULL)
  {
     printf("\n UNABLE TO CREATE FILE");
     return EXIT_FAILURE;
  }
  for(i=-128; i<=127; i++)   ⟶ Range of ASCII -128 to 127
  fprintf(fp,"%d=%c\t",i,i);
  fclose(fp);
  return EXIT_SUCCESS;
}
```

Static path E:\\1.txt
dynamic path E:\1.txt

- stdio.h provides standard I/O related predefined function prototype.
- conio.h provide console related predefined function prototype.
- FILE is a predefined structure which is available in stdio.h
  By using FILE structure, we can handle file properties.
  Size of FILE structure is 16 bytes.
- fp is a variable of type FILE*, which maintain address of FILE.
  Size of fp is 2 bytes because it holds address.

fopen() :- It is a predefined function, which is declared in stdio.h,
- → by using this function we can open a file in specific path with specific mode.
- → It requires 2 arguments of type const char*.
- → On success, fopen() returns FILE*, On failure returns NULL
- → Generally fopen() is failed to open FILE in following cases
  1. path is incorrect
  2. mode is incorrect
  3. Permissions are not available
  4. Memory is not available.

Syntax:

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

fprintf() :-
- ▸ By using this predefined function, we can write the content in file.
- ▸ fprintf() can take any no of arguments but 1st argument must be FILE* and remaining arguments are of

Syntax :-

```
int fprintf (FILE * stream, const char*path, ---);
```

fclose():-
- ▸ By using this predefined function, we can close the file after saving data.
- ▸ fclose() requires 1 argument of type FILE* and returns an int value

Syntax :-

```
int fclose (FILE * stream)
```

FILE MODES :-

→ Always FILE modes will indicate that for what purpose file need to be open or create.

FILE modes are classified into 3 types -

1. write
2. read
3. append

Depending on operations, file modes are classified into 6 types -

1) write (w) - Create a file for writing, if file already exists, then it will override (old file is deleted, new file created)

- In w mode, file exists or not, always new file is constructed.

2) read (r) - Open an existing file for reading, if file not exists then fopen() returns NULL

- When we are working with r mode, if file doesn't exist, new file is not constructed.

3) append (a) :- Open an existing file for appending (write the data at end of file) or create a new file for writing if it doesn't exist

- When we are working with "a", if file doesn't exist, then only new file is constructed

4) w+ (write and read) - Create a file for update i.e write & read, if file already exists then it will override.

- In w+ mode, if file is available or not, always new file is constructed.

5) r+ (read and write) - Open an existing file for update i.e read & write.

- Generally r+ mode is required, when we need to update existing information.
- In r+ mode, if file doesn't exist, new file is not constructed

6) a+ (w+ and r+) - Open an existing file for update or create a new file for update.

- By using a+ mode, we can perform random operations.

Files are classified into two types -

1. text files
2. Binary files

→ In Text Files, data is represented with the help of Ascii values
  • .txt , .c , .cpp

→ In binary files, data is represented with the help of byte.
  • .exe, .mp3, .mp4, .jpeg

1) To specify that a given file is being opened or create in "text mode" then append "t" to the string mode.
   Eg :- rt, wt, at, rt+, wt+, at+

2) To specify binary mode then append "b" to end of the string mode.
   Eg :- rb, wb, ab, r+b, w+b, a+b.

3) "fopen" and "fsopen" also allow that "t" or "b" to be inserted between the letter and the '+' character in the string
   Eg :- rt+ is equivalent to r+t.

4) If "t" or "b" is not giving in the string the mode is governed by "f" mode, if "f" mode is set to O_BINARY, files are opened in BINARY Mode

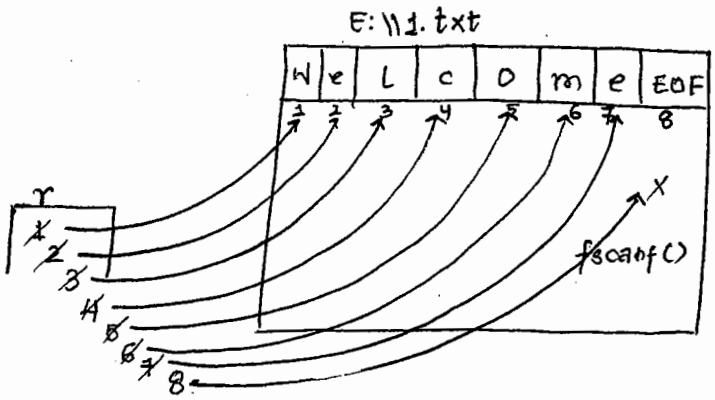
5) If "f" mode is set to O_TEXT, they are opened in text mode. These constants are defined in FCNTL.H

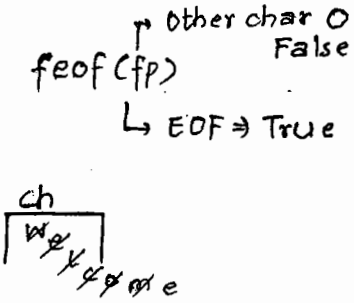
for reading data from files :-

```
# include<stdio.h>
# include <stdlib.h>
 int main()
{
  FILE *fp;
  char ch;
 fp = fopen ("E:\\1.txt", "r");
 if (fp == NULL)
 { printf ("\n FILE not FOUND");
   return EXIT_FAILURE;
 }while (1)
 {
   fscanf(fp, "%c", &ch);
   if (feof (fp))    //if (ch == EOF)
    break;
    printf ("%c", ch);
 }
```

```
fclose(fp);
return EXIT_SUCCESS;
}
```

O/p: Welcome





fscanf():- It is a predefined function which is declared in stdio.h, by using this function, we can read the data from file.

- fscanf() can takes any no. of arguments but 1st argument must be and remainining arguments should be scanf() function format.
- When we are working with fscanf() function, it can read entire content of file except

Syntax :- `int fscanf(FILE * stream, const char* format, ...);`

feof():- By using this function, we can find eof character position.

- It requires 1 argument of type FILE* and returns an int value.
- When file pointer is pointing to EOF character then it returns non-zero value, if it is pointing to other than EOF character then it returns zero.

Syntax :- `int feof (FILE * Stream);`

fgetc():- It is a predefined unformatted function which is declared in stdio.h, by using this function we can read the data from file

* including EOF character also

➤ It returns an int value i.e Ascii value of a character

Syntax: `int getc (FILE * stream);`

Reading 3 lines at a time & displaying from a file :-

Turbo C

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>  → For using delay function
int main()
{
 FILE *fp;
 char path[30];
 char ch;
 int count=0;
 clrscr();
 printf("Enter a file path: ");
 gets(path);
 fp=fopen(path,"rt");
 if(fp==NULL)
 {
  printf("\n FILE NOT FOUND");
  getch();
  return1;
 }
  while(1)
  {
   ch=fgetc(fp);      //fscanf(fp,"%c",ch);
   if(ch==EOF)          //if(feof(fp))
   break;
   printf("%c",ch);
   if(ch=='\n')
    ++count;
    if(count==3)
     {
      delay(1000);
      count=0;
     }
   }
   fclose(fp);
   getch();
   return 0;
}
```

3/8/2015.

```
#include <stdio.h>
#include <conio.h>
int main()
{
  FILE *fp;
  char path[50];
  char str[50];
  clrscr();
  //printf("Enter a file path:");
  fprintf(stdout, "Enter a file path: ");   stdout → it is the standard o/p buffer
  gets(path);
  fp = fopen(path, "a");                    → append
  if(fp == NULL)    ⇒ When because of some reason file is unable to open
  {
   fprintf(stderr, "UNABLE TO CREATE FILE");
   //printf("UNABLE TO CREATE FILE");
   getch();
   return 1
  }
 printf("Enter a string:");
 fflush(stdin);
 gets(str);
 fputs(str, fp);
 //fprintf(fp, "%s", str);
 fclose(fp);
 return 0;
}
```

delay :- It is a predefined function which is declared in dos.h

- By using this function, we can suspend the program from execution.
- delay() function requires 1 argument of type unsigned integer i.e milliseconds value

Syntax :- `void delay(unsigned milliseconds);`

- By using delay(), we can suspend the program for max. of 1 min 5 sec

Sleep() :- It is a predefind function which is declared in dos.h

By using this function, we can suspend the program from execution

Sleep() function requires 1 argument of type unsigned integer seconds format data

Syntax :- `void sleep(unsigned seconds);`

- stdout :- It is a global pointer variable which is defined in stdio.h
  → By using this global pointer, we can handle standard output buffer.

- stdin :- By using this global pointer, we can handle standard input buffer.

- stderr :- By using this global pointer, we can handle standard I/O related error.
  When we are working with stderr, it will redirect the data back to stdout.

- stdprn :- By using this global pointer, we can handle printer

- fseek() :- By using this predefined function, we can create the movement in file pointer.
  → fseek() requires 3 arguments of type FILE*, long integer & integer type.

  Syntax :- `int fseek(FILE* stream, long offset, int whence);`

  stream will provides file information
  offset is no. of bytes
  whence value is file pointer location

whence value can be recognized by using following constant values.

1. SEEK_SET :- This constant will pass the file pointer to beginning of the file.
2. SEEK_CUR :- This constant will provide constant position of file pointer
3. SEEK_END :- This constant value will send the file pointer to end of the file.

These all constant values can be recognised using INTEGER VALUES also.

SEEK_SET value is 0

SEEK_CUR value is 1

SEEK_END value is 2

rewind() :- By using this predefined function we can send the control to the beginning of the file

rewind() requires 1 argument of type FILE*

Syntax : `void rewind(FILE* STREAM)`

The behaviour of rewind() is similar to -

```
fseek(FILE*, 0, SEEK_SET);
```

ftell():- By using this predefined function, we can find the size of the file.

- ftell() requires 1 argument of type FILE* and returns long integer value.
- Generally ftell() returns file pointer posn, so if file pointer is pointing to eof character then it is equal to size of the file.

Syntax:- `long ftell(FILE* stream);`

remove():- By using this predefined function, we can delete a file permanently from Harddisk.

- remove() requires 1 argument of type const char* & returns int value

Syntax: `int remove(const char *filename);`

rename():- By using this predefined function, we can change the name of an existing file.

- rename() funcn requires 2 arguments of type const char* & returns int value.

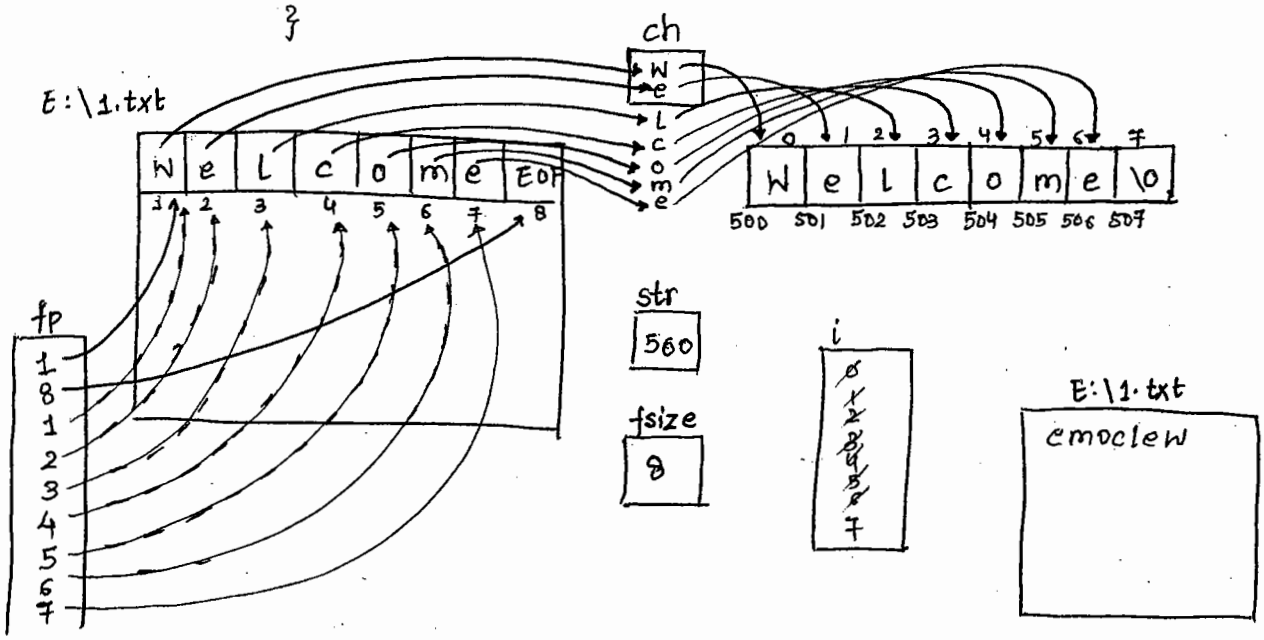
Syntax: `int rename(const char * oldname, const char* newname);`

---

prog: To reverse the string data present in a File

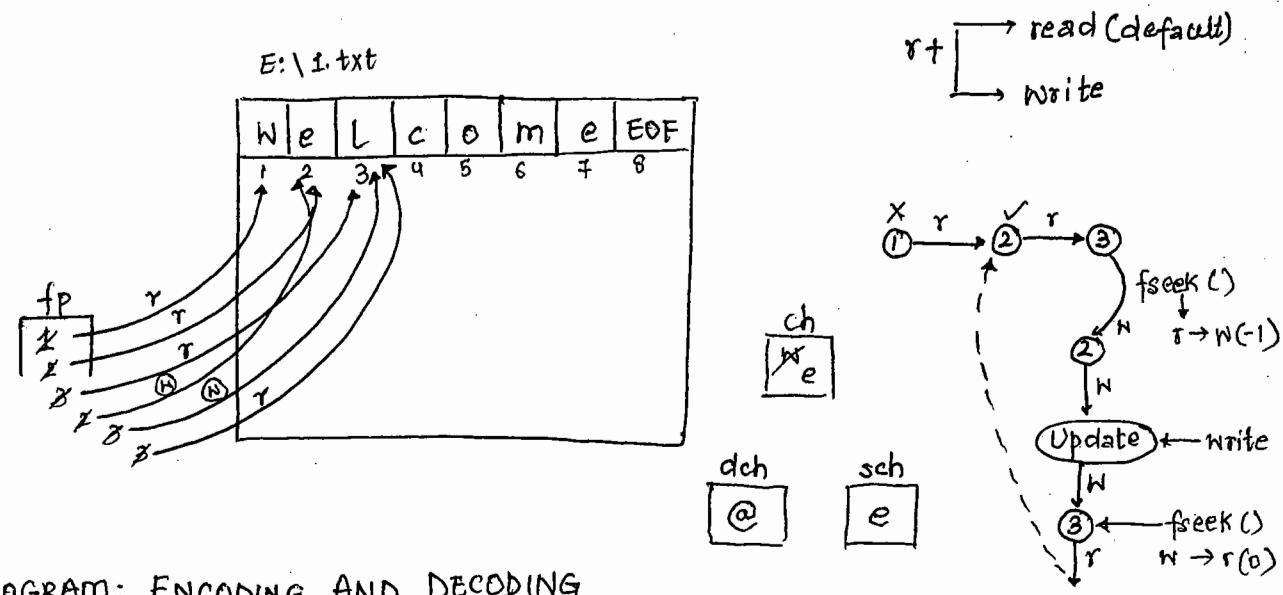
```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
int main()
{
  FILE *fp;
  char path[30];
  long int fsize, i=0;
  char *str;
  char ch;
  printf("Enter a file path:");
  gets(path);

  fp = fopen(path, "rt");
  if (fp == NULL)
   {
     printf("\n FILE NOT FOUND:");
     return 1;
   }
  fseek(fp, 0, SEEK_END); // file pointer pointing to end of the file posn.
```

```
fsize = ftell (fp);  // ftell finds the no. of bytes holded by fp and that
                        is assigned to fsize.
fseek (fp, 0, SEEK_SET); // rewind (fp) fp points to the starting position
                            of the file
str = (char*) calloc (fsize, sizeof (char));
                        // dynamically creating a string of
while(1)    // entering ∞ loop     size fsize for all the data of file
{
   ch = fgetc (fp);  → getting each character from existing file.
   if (feof (fp))    → if gets end of file then break
     break;
   str [i++] = ch;   → Add each character to str[] & increment i
  }
   str[i] = '\0';    → In string, we don't have EOF, we have \0 only
   fclose (fp);      → Save the file & close.
   remove (path);    → delete the existing file.
   strev (str);      → reversing the string characters
   fp = fopen (path, "wt");  → then opening the same file in write mode
   fputs (str, fp);  → writing character from str into fp pointing file
   // fprintf (fp, "%s", str);
   fclose(fp);
   free (str);       → freeing the m/m hold by str. string
    str = NULL;
    return 0;
  }
```



PROGRAM:- FIND AND UPDATE (Replace) A CHARACTER

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
   FILE *fp;
   char path[50];
   char ch, sch, dch;
   printf("Enter a file path:");
   gets(path);
   fp = fopen(path, "rt+");
   if (fp == NULL)
    {
     printf("\n FILE NOT FOUND:");
     return 1;
    }
    printf("\n ENTER a char (s): ");  // Enter source character to be replaced
    fflush(stdin);
    sch = getchar();          // Get it from user
    printf("\n Enter a char (D): "); // Enter dest character to be replaced with
    fflush(stdin);
    dch = getchar();
    while(1)
    {  ch=fgetc(fp);  // The character read by the file pointer
       if(ch == EOF)
        break;
       if(ch==sch)
       {
          fseek(fp, -1, SEEK_CUR); //r---> w(-1)   The current position pointed by the file pointer is shifted 1 Byte back & mode changed from read to write automatically.

          fprintf(fp, "%c", dch); #

          fseek(fp, 0, SEEK_CUR); //w -----> r(0)
       }
    }
    fclose(fp);
    return 0;
}
```

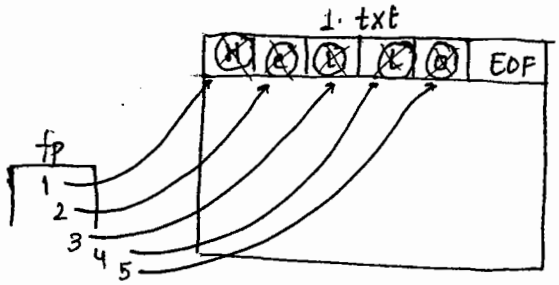


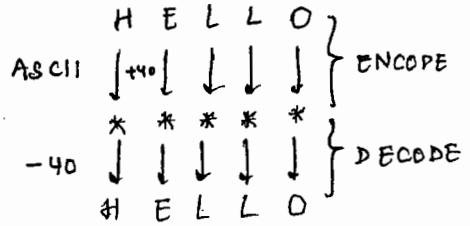
PROGRAM: ENCODING AND DECODING

```
#include <stdio.h>
#include <conio.h>
int main()
{
   FILE * fp
   int flag, code = 0;
   char path [30];
   char ch;
   clrscr();
   printf ("Enter a file path: ");
   gets(path);
   fp = fopen (path, "r+");
   if (fp == NULL)
  {
     printf ("\n FILE NOT FOUND");
     getch ();
     return 1;
   }
  do
    {
      printf ("\n 1 FOR ENCODE:");
      printf ("\n 2 FOR DECODE:");
      scanf ("%d", &flag);
    }
    while (flag != 1 && flag != 2);   // When we enters i/p other then 1 & 2
    if (flag == 1)                    then do while repeats to take value
       code = 40;                     again bz user can enter value other
    else                              than 1 & 2
       code = -40;
    while (1)
       {
```

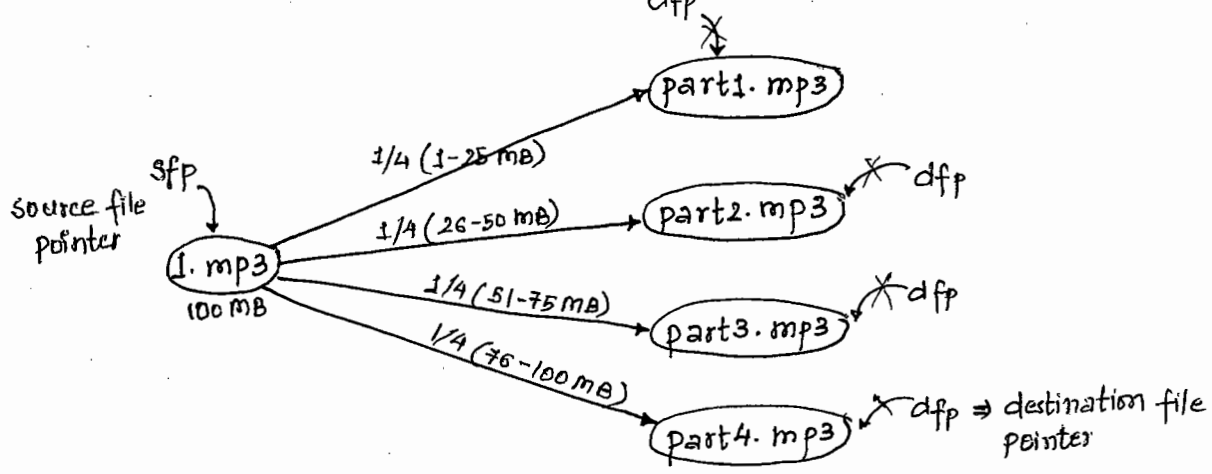
```
        ch = fgetc(fp);
        if (ch == EOF)
            break;
        if (ch != '\n' && ch != '\r')   → For encoding & decoding, we have to perform all other operations instead of these 2.
        {
            fseek(fp, -1, SEEK_CUR);        //r ----> w(-1)
            fprintf(fp, "%c", ch+code);     //update
            fseek(fp, 0, SEEK_CUR);         //w ----> r(0)
        }
    }
    fclose(fp);
    return 0;
}
```

Enter a file path: E:\1.txt





PROGRAM: OPERATIONS ON AUDIO FILE (Binary File)




```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define f1 "E:\\part1.mp3"
#define f2 "E:\\part2.mp3"
#define f3 "E:\\part3.mp3"
#define f4 "E:\\part4.mp3"
int main(void)
{
```

```
FILE * sfp;
FILE * dfp;
char spath [30];
char dpath [4][30] = {f1, f2, f3, f4};
long int fsize = 0, nb = 0;        nb → no. of bytes
int i = 0;
char ch
printf ("\n Enter sfp path:");
gets (spath);
sfp = fopen(spath, "rb");        rb → read binary file
if sfp == NULL)
{
  printf ("\n %s NOT FOUND", spath);      // This means file is already opened,
  return EXIT_FAILURE;                       now we need to find path of file
}
fseek (sfp 0, SEEK_END);  // If the file is open then find the file size
 fsize = ftell (sfp);                  // gives files path
  rewind (sfp);      //fseek(sfp, 0, SEEK_SET);
  dfp = fopen (dpath [i], "wb");  // To open 1st part
  if (dfp == NULL)  // If this condition is false then it means 1st part is created
{
   fclose (sfp);
    printf ("\n%s UNABLE TO CREATE");
    return EXIT_FAILURE;
  }
   while (1)
   {
     ch = fgetc (sfp);
     if (feof (sfp))
     break;                                  } reading 1st part data
     fprintf (dfp, "%c", ch);
     ++nb;      ⇒ increments no. of bytes
      if (nb = fsize/4 && i != 3)  ⇒ we have to close only 3 initial parts within
                                     the loop [0,1,2] & not the 4th part, otherwise source
     {                               file will be terminated
       fclose (dfp);
        nb = 0;
        ++i;
        dfp = fopen (dpath [i], "wb");
        if (dfp == NULL)
     {
        printf ("\n %s UNABLE TO CREATE ", dpath [i]);
        fclose (sfp);                  \\part1 closed    // part2 closed
        return EXIT_FAILURE;
     }
```

```
        }
    }
    fclose(sfp);
    fclose(dfp);
    return EXIT_SUCCESS;
}
```

PROGRAM: TO COMBINE 2 OR MORE AUDIO FILES TOGETHER :

```
#include <stdio.h>
#include <stdlib.h>
#define f1 "E:\\part1.mp3"
#define f2 "E:\\part2.mp3"
#define f3 "E:\\part3.mp3"
#define f4 "E:\\part4.mp3"
int main(void)
{
    FILE *sfp;
    FILE *dfp;
    char dpath[30];
    char spath[4][30] = {f1, f2, f3, f4};
    int i = 0;
    char ch;
    printf("\nEnter dfp path: ");
    gets(dpath);
    dfp = fopen(dpath, "wb");
    if (dfp == NULL)
    {
        printf("\n%s UNABLE TO CREATE", dpath);
        return EXIT_SUCCESS
    }
    for(i=0; i<4; i++)
    {
        sfp = fopen(spath[i], "rb");
        if (sfp == NULL)
        {
            fclose(dfp);
            printf("\n%s NOT FOUND", spath[i]);
            return EXIT_FAILURE;
        }
        while(1)
        {
```

```
    {
      ch = fgetc (sfp);
          if (feof (sfp))
            break;
       fprintf (dfp, "%c", ch);
    }
    fclose (sfp);
  }
  fclose (dfp);
  return EXIT_SUCCESS;
}
```

4/8/2015

COMMAND-LINE ARGUMENTS

- It is a procedure of passing the arguments to the main function from command prompt.
- By using command line arguments, we can create user-defined commands.
- In implementation, when we required to develop an application for dos operating system then recommended to go for command line arguments.
- DOS is a character based or command based OS i.e if we require to perform any task, we need to use specific command only.
- When we are developing the program in command line arguments, then main function will take 2 arguments i.e argc & argv
- argc is a variable of type an integer, it maintains total no. of arguments count value.
- argv is a variable of type char*, which maintains actual argument values which is passed to main().
- In C and C++, by default total no. of arguments are 1, i.e programname.exe

Ex-
```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("\n Total No. of Arguments : %d", argc);
    for(i=0; i<argc; i++)
    printf("\n %d. Argument : %s", i+1, argv[i]);
    return 0;
}
//save the file as mytest.c
// compile mytest.c
// Build or rebuild mytest.c
```

Ex

- Load the command prompt & change the directory to specific location where application file is available.
- To execute the program, just we required to use program name or program name.exe
- CommandLine Arguments related programs not recommended to execute by IDE because we can't pass the arguments to main-function.

```
O/P: D:\C400PM\4>mytest 10 Hello 20
     Total No of Arguments : 4
  1. Argument: mytest
  2. Argument: 10
  3. Argument: Hello
  4. Argument: 20
```

→ argc & argv are local variables to main() so command Prompt related data we can't access outside of the main()

→ In implementation, when we required to access command prompt related data outside of main(), then recommended to go for _argc & _argv variables, which is defined in dos.h.

```
#include <stdio.h>
#include <dos.h>
   void abc()
   {
      int i;
      printf("\n Data in abc:");
      printf("\n Total No of arguments : %d", _argc);
      for(i=0; i<_argc; i++)
      printf("\n %d. Argument : %s", i+1, _argv[i]);
   }
   int main(int argc, const char*argv[])
     {
        int i;
        printf("\n Data in main :");
        printf(
        for(i=0; i<argc; i++)
            printf("\n %d Arguments: %s", i+1, argv[i]);
           abc();
          return 0;
        }
          // Save as mycmd.c
          // Compile mycmd.c
          // Build are rebuild mycmd.e
```

```
O/P : D:\C400PM> mycmd Hello 10 Welcome

Data in main :
  Total No of Arguments : 4
  1. Argument : D:\C400PM\MYCMD.EXE
  2. Argument : Hello
  3. Argument : 10
  4. Argument : Welcome

Data in abc :
  Total No of Arguments : 4
  1. Argument : D:\C400PM\MYCMD.EXE
  2. Argument : Hello
  3. Argument : 10
  4. Argument : Welcome
```

Due to local variables

Due to global variables - _argc & _argv

- By using command Line Arguments, it is possible to pass any type of data but all are treated like string only.
- When we are working with Command line Arguments, always recommended to convert string type to numeric type properly before execution of the logic
- Conversion related all predefined functions are declared in stdlib
- stdlib related predefined functions are -

<stdlib.h>

atoi()	atol()	atof()	_atold()	itoa()
ltoa()	abort()	exit()	_exit()	_c_exit()
atexit()	raise()	signal()		
min()	max()	random()	randomize()	
rand()	srand()	abs()	labs()	
strtod()	strtol	strtold()	strtoul()	
wctomb()	wcstombs()	fcvt()	gcvt()	
lfind()	lsearch()	_lrotl()	_lrotr()	div() ldiv()

Constants — EXIT_SUCCESS — EXIT_FAILURE

1) atoi() - By using this, we can convert a string into integer type
   - atoi() function requires 1 argument of type char* & returns int type.

Syntax :- `int atoi(const char *str);`

2) atol() :- By using this, we can convert a string into long integer type.

Syntax: `long atol (const char* str);`

3) atof() :- By using this predefined function, we can convert a string into double datatype.

Syntax: `double atof (const char* str);`

4) _atold() :- By using this predefined funcn, we can convert a string into long double type.

Syntax: `long double _atold (const char*str);`

By using fcvt(), gcvt(), ecvt(), we can convert float to string type.

USER-DEFINED FUNCTION for atoi()

```
#include<stdio.h>
#include<stdlib.h>
int myatoi (const char*str)
{
  int r=0; i=0; flag=0;
  if (str[0]=='+' || str[0]=='-')
    {
      i=1;
      if (str[0]=='-')
        flag=1;
    }
  for (; str[i]!='\0'; i++)
    {
      if (str[i]>='0' && str[i]<='9')
        r = r*10+ str[i] - '0';
      else
        {
          if (flag==0)
            return r;
          else
            return (-r);
        }
    }
  if (flag==0)
    return r;
  else
    return (-r);
}
```

```
int main(int argc, const char *argv[])
{
  int i, n;
   if (argc != 2)
   {
    printf ("\n invalid command syntax");
     return EXIT_FAILURE;
   }
   // n = atoi(argv[1]);
     n = myatoi(argv[1]);
     for(i=1; i<=10; i++)
         printf ("\n%d* %2d = %2d", n, i, n*i);
      return EXIT_SUCCESS;
   }
     //Save as mytab.c
     // compile mytab.c
     // Link or Build mytab.c
     // Run mycmd.exe using command prompt
```

Explaination

1) my tab 25
   mytab +25
   mytab 25ABC
   mytab +25ABC
   } 25

2) mytab -25
   mytab -25ABC
   } -25

3) mytab abc
   mytab
   mytab
   mytab
   mytab
   } 0

<ctype.h>

- character type related specific functions are available in ctype.h
- ctype.h related predefined functions or MACROs are-
  1. isalpha(); - checking alphabet or not
  2. isascii(); - checking I/P data within ascii range or not.
  3. iscntrl(); - If control button from keyboard
  4. isspace(); - If space button from keyboard
  5. isupper(); - required to use from A to Z
  6. isdigit(); - checking whether numeric type or not
  7. islower(); - To check if it is a-z
  8. tolower(); - convert upper to lower
  9. toupper(); - convert lower to upper
  10. toascii(); - converts to ascii value

→ When the function prefix 'is' available, then it returns boolean type i.e true or false

→ If 'to' is available, then it returns integer value

```
# include <stdio.h>
# include <ctype.h>
  int main ()
{
   char ch;
   int flag;
   printf ("Enter a char: ");
   ch = getchar();
   flag = isdigit (ch);
   if (flag != 0)
      printf ("\n %c IS DIGIT DATA", ch);
   else
      printf ("\n %c IS NOT DIGIT DATA", ch);
      return 0;
   }
```

O/P:

```
Enter a char: 9
9 IS DIGIT DATA
```

```
# include <stdio.h>
# include <ctype.h>
 int main()
 {
    char ch1, ch2;
    printf ("Enter a char: ");
    ch1 = getchar();
    ch2 = tolower (ch1);
    printf("\n% Lower case char is = %c", ch1, ch2);
     return 0;
   }
```

O/P:

```
Enter a char: A
A Lower case char is : a
```

PROJECT-1

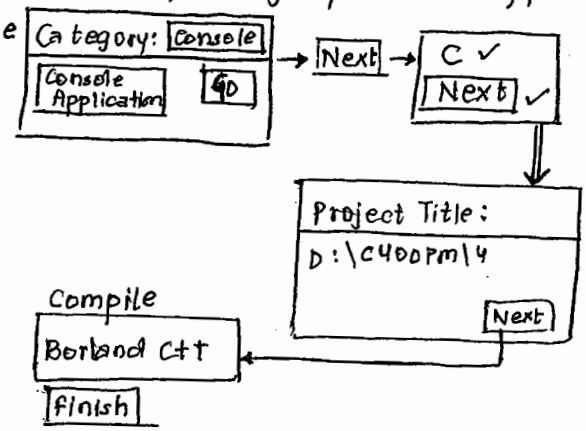
Q Create a console based applicated using code Block IDE with the name of Employee Management System with following modules.

1) Login Module
2) Add new record
3) Display records
4) Search specific record
5) Update existing records
6) Delete existing records
7) Display backup data

CONFIGURATION OF PROJECT :-

Open CODE BLODE ; create new project of category console Applicatin

File → New → Project → Console App.

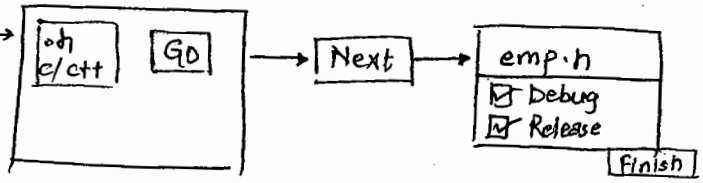


⇒ To maintain employee information, we require a structure that's why it is recommended to create user defined datatype by using structures.

⇒ Always recommended to create a structure in header files only. [user-defined header files]

⇒ To create user defined header files, it is recommended to following steps :-

File → New → File →

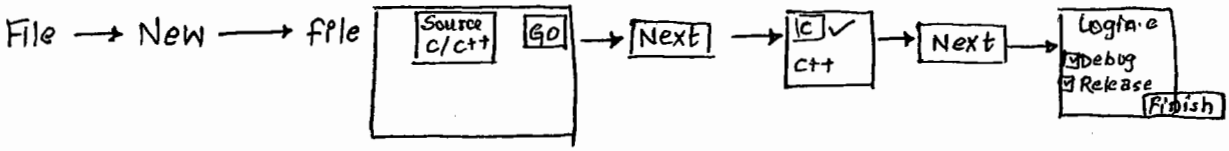


Code in emp.h

```
# ifndef EMP_H_INCLUDED
# define EMP_H_INCLUDED
  typedef struct
    {
      char ID [10];
      char NAME [36];
```

```
    char EMAIL [20];
    unsigned int SALARY;
    float PF;
} EMP;
# endif  // EMP_H_INCLUDED
```

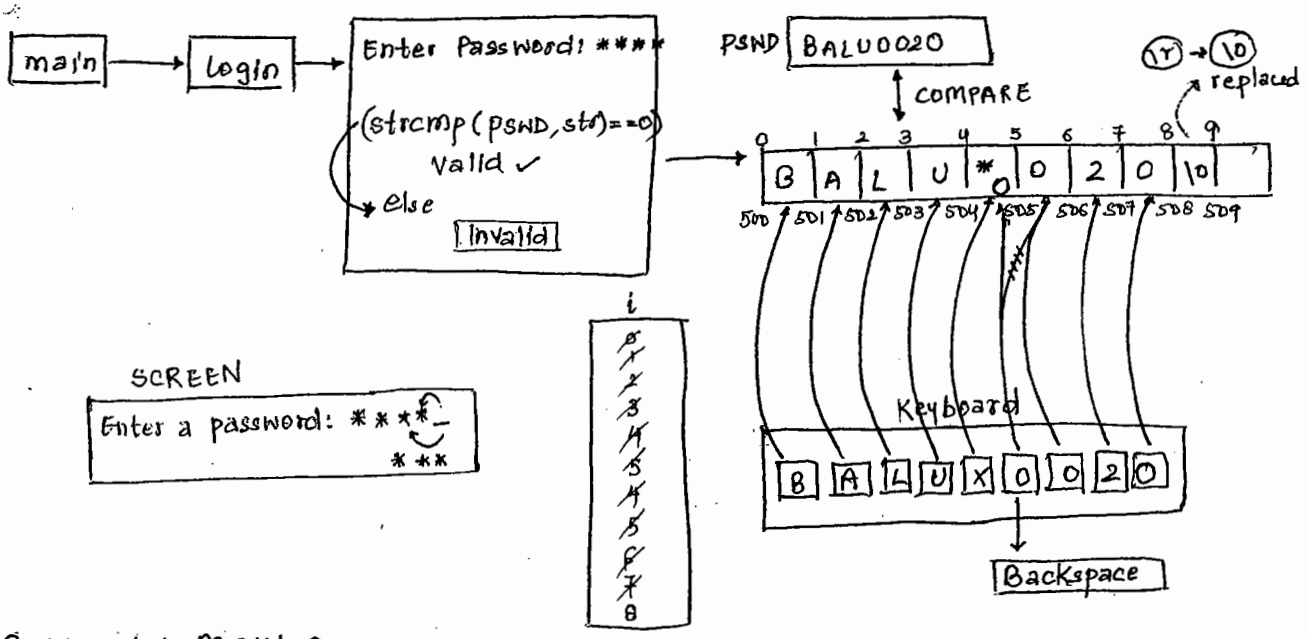
- Create a new module file with a name login.c



CODE IN LOGIN.C

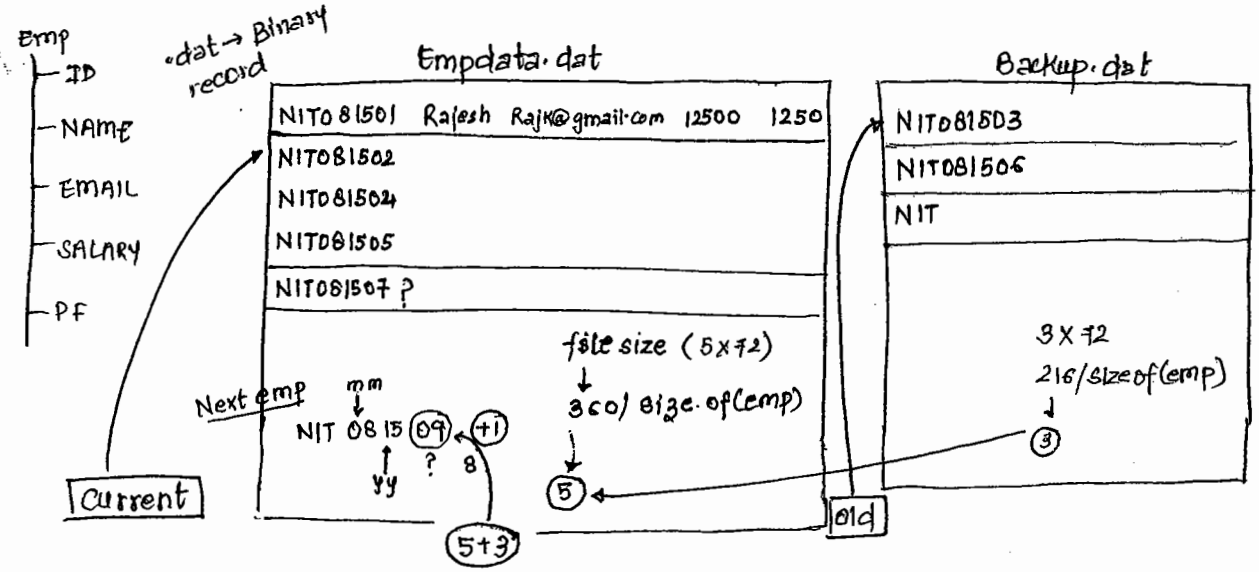
```
# Include <stdio.h>
# include <conio.h>
# include <string.h>
# include <process.h>
 Void password()
 {
   char pswd[10] = "BALU0020";
   char str[10];
   int i = 0;
   void mainmenu(void);
   printf(" Enter password");
  While(1)
  {
     str[i]= getch();
     if (str[i]=='\r')  // Enter button action
      {
         str[i] = '\0';
         break;
       }
     else if (str[i]==0)  // backspace action
     {
          if (i>0)        // Min limit of backspace
        {
          printf ("\b");  // backspace
          printf(" ");  // remove * replace with 'space'
          printf("\b");  // backspace
          --i;
          str[i]='0';
        }
       }
```

```
else
  {
    printf ("* ");
     ++i;
  }
}
if (strcmp (pswd, str) == 0)
  {
    mainmenu();     // for next view of project
  }
else
   {
    printf ("\n invalid password: ");
    exit (1);
   }
}
```



CODE IN MAIN.C

```
# include <stdio.h>
# include <stdlib.h>
int main()
{  void password(void)
   password();
   return 0;
}
```



CREATE A NEW MODULE FILE WITH THE NAME empcount.c

CODE IN empcount.c

```
#include <stdio.h>
#include <emp.h>
/* Application global variables */
FILE *cur;
FILE *old;
EMP e;
int ne;
int curcount()
{
   int n;
   cur = fopen ("EMPDATA.dat", "rb");
   if(cur == NULL)   // When the database file doesnot exist (EMPDATA)
        return 0;    // No Records
   fseek(cur, 0, SEEK_END); // Will reach end of the file & will give total no. of bytes
   n = ftell(cur) / sizeof(EMP); // n will give total no. of current records
   fclose(cur);
   cur = NULL;
   return n;        // n returns total no. of records
}
int oldcount()
{
    int n;
    old = fopen ("BACKUP.dat", "rb");
    if (old == NULL)
        return 0;
    fseek (old, 0, SEEK_END);
     n = ftell (old)/ sizeof (EMP);
     fclose (old);
```

```
    old = NULL;
    return n;
}
```

SQLite Database

→ It is a light weighted server less open source database engine.

→ When we are working with this database it doesnot require any pre configuration and it doesnot require server also.

→ Generally this database is used in mobile application.

→ When we are developing stand alone applications then recommended to go for SQLite database.

→ Standalone application means, the system in which application is running in same system database is available.

→ The complete reference of SQLite database is available in www.sqlite.org.

→ Document reference is available in www.sqlite.org/docs.html

→ In 'C' language when we are developing standalone application with database they recommended to go for SQLite in place of using FILE system.

→ In 'C' language we doesnot have any kind of predefined functions to interact with SQLite database.

→ In implementation when we are interacting with SQLite database engine then recommended to go for

API'S which is provided by SQLite database.

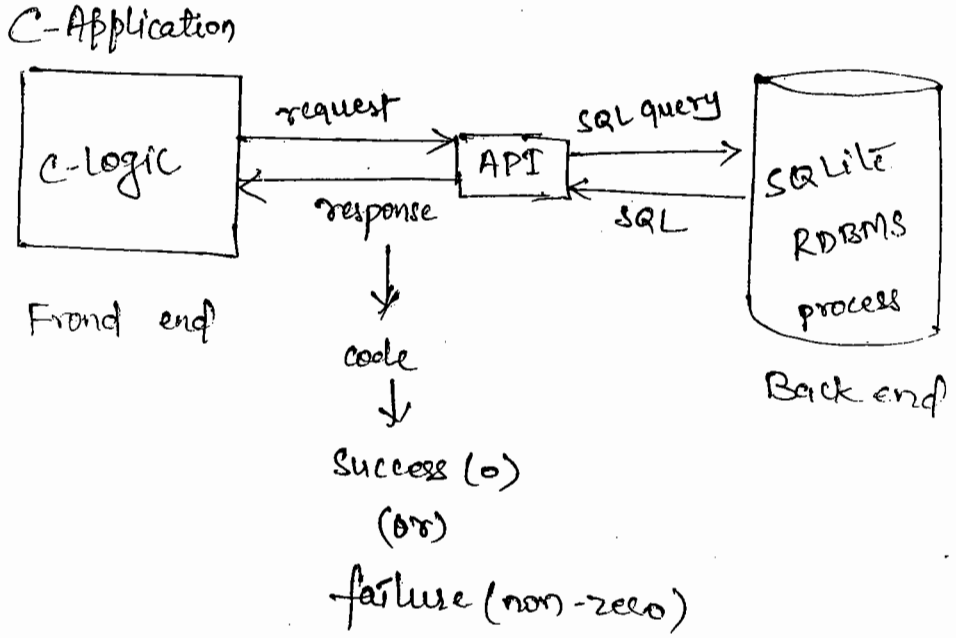
→ SQLite related all API's are developed in native 'C' language.

→ Complete API reference we can get from www.sqlite.org/cintro.html

→ Function means when we are performing the task within the program.

→ API means b/w the application when we need to perform.

→ In 'C' application when we are performing the subtask they we'll go for the function, if 'C' application is interacting with any other application they go for API. (Application programming Interface).



Configuration of SQLite database with Dev C++ IDE.

→ When we are working with Dev C++ IDE it provides multiple packages for different type of application development like 2D or 3D Graphics, Cryptography, Image manipulation, Networking, Database, multithreading and many more.

→ When we are working with any kind of packages first we require to config that specific package.

→ Goto devpaks.org website then select database category.

→ From list of libraries recommended to select sqlite3 ~~vers~~ library version: 3.7.4 (up to now it is latest)

→ After selecting specific database click on download url and wait until download is finished.

→ Once download is finished run package installer file. Then automatically configuration is completed.

Configuration of SQLite database with current project

→ Create a new project directory with the name SQLite_emp.

→ Go to www.sqlite.org website then click on download tab.

→ Under source code section download sqlite-amalgamation-xxxxxx.zip file.

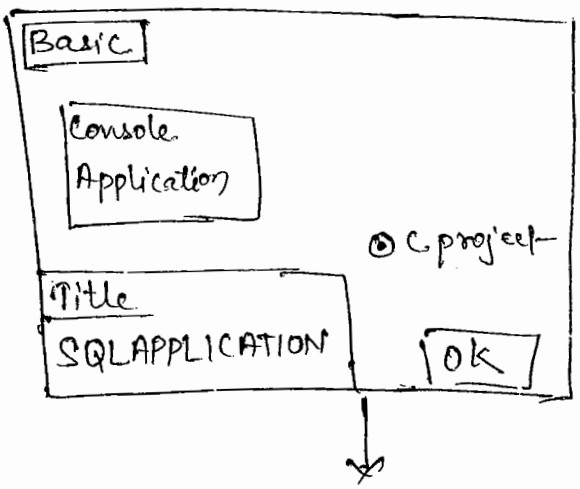
→ Extract the downloaded zip file which is having following files

- shell.c
- sqlite3.c
- sqlite3.h
- sqlite3ext.h

→ Copy sqlite.c and sqlite3.h into newly constructed folder i.e SQLiteemp.

→ Open Dev C++ IDE and create new console application type project with the name SQL APPLICATION and save this project template into SQLiteemp folder which was created earlier.

File → New → Project



Save project template & main.c in SQLiteemp directory only.

→ Go to project menu and select Add to project.

→ Select Sqlite3.c & sqlite.h files. then click on open button.

25/08/2015

Syntax to open and close the database

→ For openning the database we have an API called sqlite3_open(), for closing the database we have an API called sqlite3_close().

→ For handling the complete application we are creating three application variables' i.e!-

→ sqlite3 *db; // database pointer

→ char* errmsg; // char type pointer

→ int * errcode; // global int variable

→ sqlite3 is a predefined structure which is available in "sqlite3.h", Using this structure we can handle sqlite database.

2015

→ sqlite3_open():-

→ Using this API we can open SQLite database

→ This API requires two arguments of type const char* and sqlite3**.

→ On success this API returns zero, On failure it returns non-zero value.

```
ercode = sqlite3_open("EMPDATABASE.db",&db);
if (ercode==0)  // success
{
   // go for next step
}
else
printf("\n Error in opening database : %s",
                sqlite3_errmsg(db));
```

sqlite3_errmsg() :-

→ Using this API we can find any kind of error which is occurred at the time of working with SQLite data base.

→ This API requires one argument of type sqlite3** and returns char*.

⇒ sqlite3_close() :-

→ Using this API we can close a database which is opened by sqlite3_open() API.

→ sqlite3_close() requires one argument of type sqlite3* .

Syntax to create table

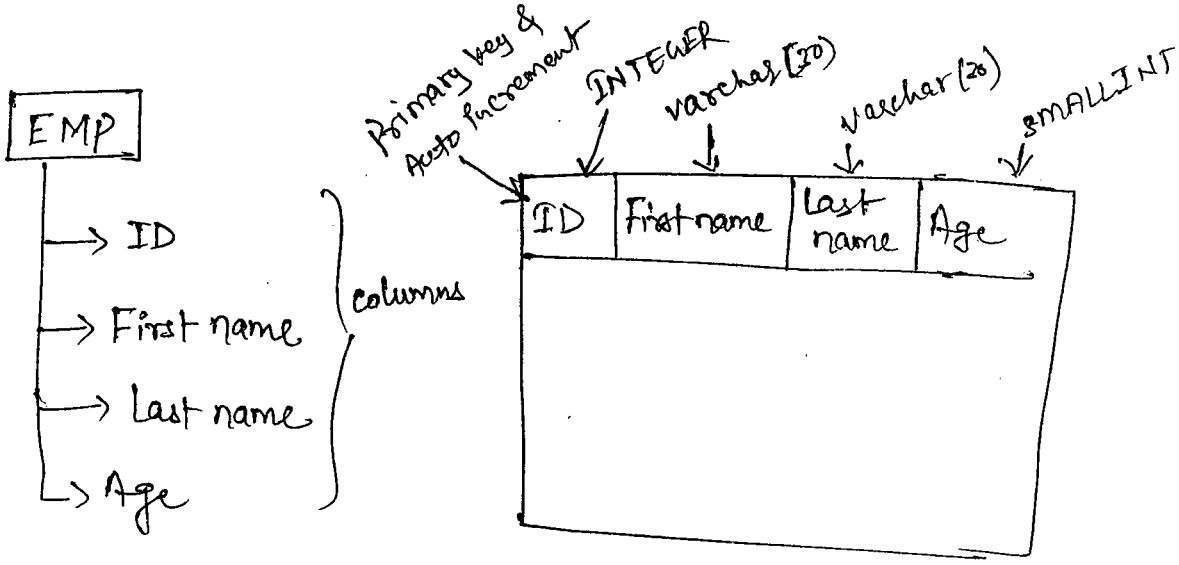
→ A table is a basic element of any kind of database.

→ Table maintains the information in the form of rows and columns with the help of RDBMS rules.

→ When we require to create any kind of table then we need

to use create table statement.

→ When we are creating table initially we need to decide ~~skeleton~~ skeleton or structure of table.



Syntax:-

Create table <tableName> (column1 Datatype, column2 Datatype, column3 Datatype, ...)

Ex:- create table if not exists EMP (
ID INTEGER PRIMARY KEY AUTOINCREMENT,
FirstName varchar(30), Last-Name varchar(30),
Age smallint)

→ The last query is available in front end only and it is not understandable to 'c' compiler that's why this query is needed to be passed to back end using API.

sqlite3_exec()

→ Using this API we can execute any type of SQL query.

→ When we are using this API we need to pass database pointer, Query string, attributes and character pointer

→ If the query execution is success then we will get return value SQLITE_OK i.e. 0 else it returns positive value.

⇒ char* sql_stmt = " create table if not..... ";

```
if (sqlite3_exec(db, sql_stmt, NULL, NULL, &errmsg)
        != SQLITE_OK)
{
 //failure stop
}
else    // success
fprintf (stdout, "\n Table is created ");
```

sqlite3_free():-

→ Using this API we can release dynamically created memory ~~for~~ by SQLite database.

Adding rows in table

→ When we are adding the data in table we need to use insert statement.

→ Using insert statement we can add all columns data or we can pass specific column data also

Syntax 1:-

```
insert into tableName values (value1, value2, value3, ...);
```

Ex:-

```
Insert into EMP values (1, 'Rajesh', 'Kumar', 26);
```

✗ Not suitable because ID has auto incrementation.

Syntax 2:-

```
insert into tableName (column1, column2, ...) values(
value1, value2, value3, ...);
```

✓

Ex:-

```
insert into EMP (FirstName, LastName, Age) values(
'Rajesh', 'kumar', 26);
```

✓

EMP

ID	First name	Last name	Age
1	Rajesh	Kumar	26

Back end

C Application

Enter First Name: Rajesh
Enter last name: Kumar
Enter Age: 26

Front end

sqlite3_exec();

fname: Rajesh

lname: Kumar

Age: 26

sprintf (sql_strnt, " insert into EMP (Firstname, Lastname, age) values ('%s', '%s', %d", fname, lname, age);

copy

insert into EMP (Firstname, Lastname, Age) values ('Rajesh', 'Kumar', 26).

ID
1
2
3

* sqlite3_prepare_v2()'-

→ Using this API we can prepare the database which can take huge memory from front end application.

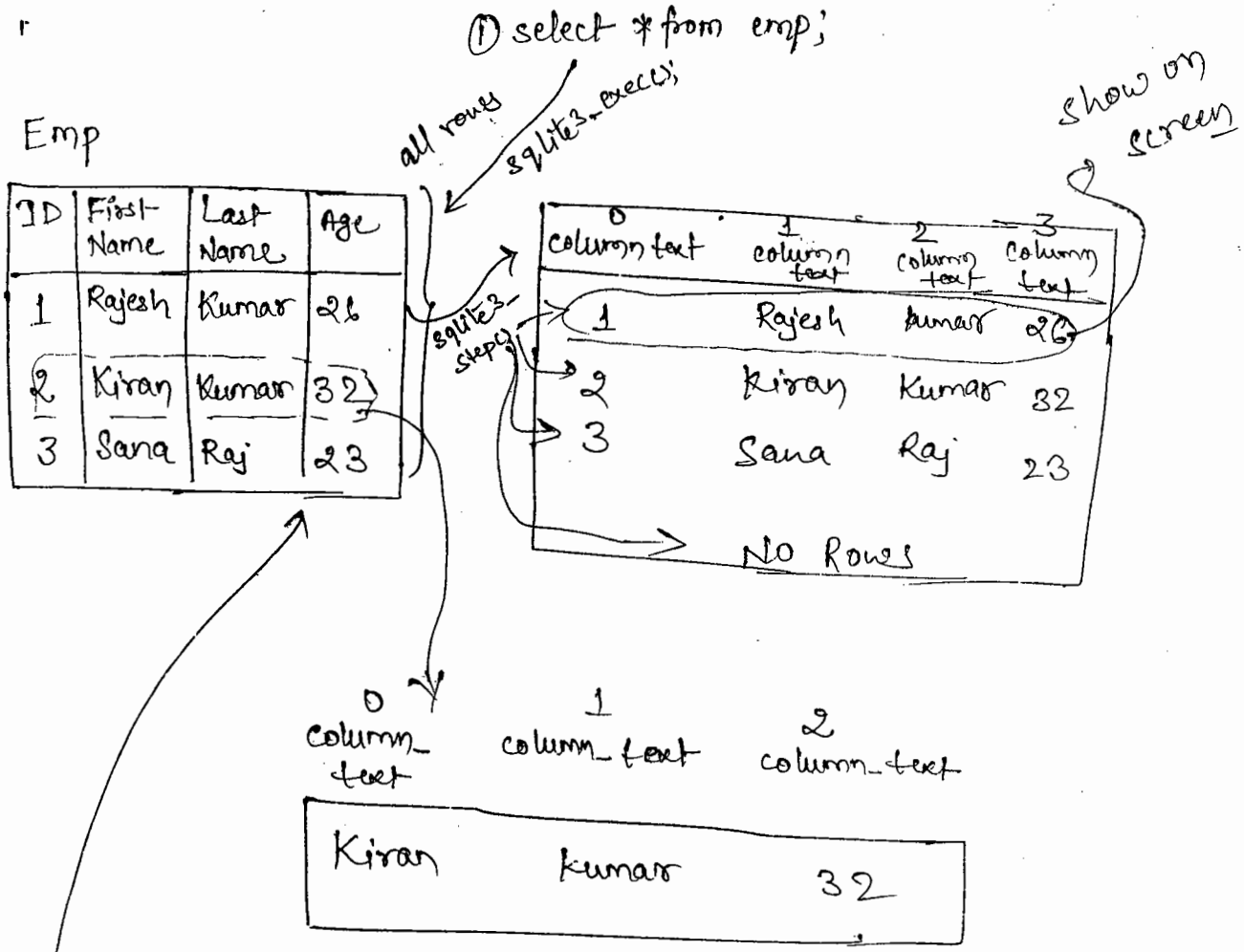
Procedure to retrieve the data from table

→ When we require to retrieve the data from table then we need to use select statement.

→ By using select statement we can retrieve all rows or any specific row also possible.

Emp

ID	First Name	Last name	Age
1	Rajesh	Kumar	26
2	Kiran	Kumar	32
3	Sana	Raj	23



② Select (FirstName, LastName, Age) from Emp where ID == 2;

sprintf (sql_stmt, "select (Firstname, Lastname, Age) from Emp where ID == %d", id);

sqlite_exec()

C Application

Enter Emp ID: 2

* sqlite3_step() :-

→ Using this API we can retrieve a single row at a time from memory.

→ When row is available then this API returns ~~sqlite~~ SQLITE_ROW, when all rows are finished then it returns SQLITE_DONE.

* sqlite3_column_text() :-

→ Using this API we can extract the content based on ~~ID~~ Index.

Procedure to update the Record.

→ Using update statement we can update the data in table.

→ When we are working with update query we require to use set keyword along with where condition.

Emp

ID	First Name	Last Name	Age
1.	Rajesh	Kumar	26
2.	Kiran	Kumar	32
3.	~~Saha~~ Raj	~~Raj~~ Sana	~~25~~ 32

sqlite_exec();

C-Application

Enter ID : 3

Enter new First name: Raj

Enter new Last name : Sana

Enter new age : 32

fname [Raj]

lname [Sana]

age [32]

id [3]

update emp set Firstname = 'Raj',

Lastname = 'Sana', age = 32

where ID == 3

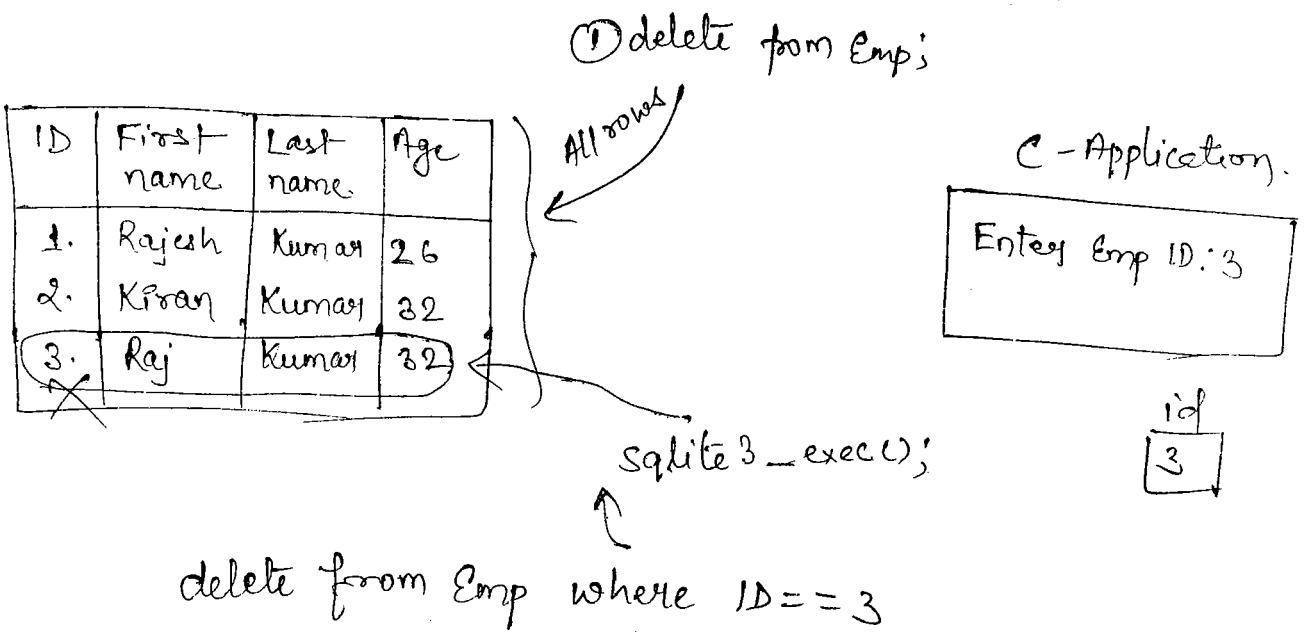
ID
1.
2.
3. ~~X~~

```
Sprintf(sql_stmt, "update Emp set FirstName = \'%s\',
        LastName = \'%s\', Age = %u" where ID == %u",
                fname, lname, age, id);
```

Deleting the data from table

→ Using delete statement we can delete the data from table.

→ Using delete statement we can delete all the row or any specific row also can be deleted.



sprintf (sql_stmt, "delete from Emp where ID == %d", id);

Deleting the table

→ Using drop statement we can delete table

→ `drop table emp;`

Code in main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include "sqlite3.h"

/* run this program using the console pauser or add your own getch,
system("pause") or input loop */

sqlite3*db;

int main(int argc, char *argv[])
{
        int option,ercode;
        ercode=sqlite3_open("test.db", &db);
        if(ercode)
        {
         //failed
         fprintf(stderr,"\nCan't open database: %s\n", sqlite3_errmsg(db));
         return 0;
   }
   else
   {
                // success
                while(1)
                {
                        system("CLS");
                        printf("\nFor Create Table---------->1: ");
                        printf("\nFor Insert Data----------->2: ");
                        printf("\nFOr Display Data---------->3: ");
                        printf("\nFor Update Table Data----->4: ");
                        printf("\nFor Delete Data----------->5: ");
                        printf("\nFor Delete Table---------->6: ");
                        printf("\nFor Exit Application------>7: ");
                        scanf("%d",&option);
                        switch(option)
                        {
```

```
                case 1:create_table();
                        break;
                case 2:insertdata();
                        break;
                case 3:display();
                        break;
                case 4:update_table();
                        break;
                case 5:deletedata();
                        break;
                case 6:droptable();
                        break;
                case 7:
                        sqlite3_close(db);
                        return 0;
            default: printf("Invalid Option: \n");
                system("PAUSE");

                                break;
                }
            }
        }
}
```

Code in createtable.c

```
#include <stdio.h>
#include <process.h>
#include "sqlite3.h"
void create_table()
{
  char*zErrMsg;
  extern sqlite3*db;
  char *sql_stmt="create table if not exists myTable (ID INTEGER PRIMARY
KEY AUTOINCREMENT,FirstName varchar(30), LastName varchar(30), Age
smallint)";
  if(sqlite3_exec(db,sql_stmt, NULL, NULL, &zErrMsg) != SQLITE_OK)
```

```
  {
   fprintf(stderr,"Failed to create table:%s\n",sqlite3_errmsg(db));
   sqlite3_close(db);
   sqlite3_free(zErrMsg);
        system("PAUSE");
        exit(1);
  }
  else
        fprintf(stdout,"Table is created\n");
  system("PAUSE");
  return;
}
```

Code in display.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
void displayallrow()
{
        char*query_stmt="select * from myTable";
        extern sqlite3*db;
        char*zErrMsg;
        sqlite3_stmt *statement;
        sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
   if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
     printf("\n==============================\n");
          while(sqlite3_step(statement)== SQLITE_ROW)
     {
      printf("\n%5s %10s %3s %5s",sqlite3_column_text(statement,
0),sqlite3_column_text(statement, 1),sqlite3_column_text(statement,
2),sqlite3_column_text(statement, 3));
     }

     printf("\n==============================\n");
     sqlite3_finalize(statement);
        }
```

```
        else
        {
         fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
         sqlite3_free(zErrMsg);
        }
   getch();
        return;
}
void  displaysinglerow()
{
        unsigned int id;
        sqlite3_stmt*statement;
        extern sqlite3*db;
        char*zErrMsg;

   char *query_stmt=(char*)calloc(200,sizeof(char));
        printf("\nEnter ID: ");
        scanf("%u",&id);
        sprintf(query_stmt,"SELECT FirstName,LastName,Age FROM myTable
WHERE ID ==%u",id);;
   sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
   if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
         if(sqlite3_step(statement) == SQLITE_ROW)
    {
      printf("\n==============================\n");
          //display here selected row data
      printf("\n%5s %3s %2s",sqlite3_column_text(statement,
0),sqlite3_column_text(statement, 1),sqlite3_column_text(statement, 2));
      fprintf(stdout,"\n1 Row is selected");
      printf("\n==============================\n");

    }
    else
    {
     fprintf(stderr,"\nNo Rows are selected");
    }
    sqlite3_finalize(statement);
```

```
    }
        else
        {
         fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
         sqlite3_free(zErrMsg);
        }
    free(query_stmt);
        getch();
        return;
}
void display()
{
        int option;
        printf("\nDisplay All Rows.....1: ");
        printf("\nDisplay single Row...2: ");
        scanf("%d",&option);
        if(option==1)
     displayallrow();
        else if(option==2)
          displaysinglerow();
        else
                fprintf(stderr,"\nInvalid Option: ");
    getch();
        return;
}
```

Code in updatedata.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
#define TRUE 1
#define FALSE 0
int isValidid(int tempid)
{
        sqlite3_stmt*statement;
        extern sqlite3*db;
```

```
        char*zErrMsg;
        char *query_stmt=(char*)calloc(200,sizeof(char));
        sprintf(query_stmt,"SELECT FirstName,LastName,Age FROM myTable
WHERE ID ==%u",tempid);;
   sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
   if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
         if(sqlite3_step(statement) == SQLITE_ROW)
         {
          sqlite3_finalize(statement);
     return TRUE;   //tempid id valid
     }
     else
     {
       return FALSE;
       sqlite3_finalize(statement);
     }

   }
        else
        {
         fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
         sqlite3_free(zErrMsg);
         return FALSE;
        }
}
void update_table()
{
        char *fname;
        char *lname;
        unsigned int age;
        unsigned int tempid;
        sqlite3_stmt*statement;
   char *query_stmt;
   extern sqlite3*db;
   char*zErrMsg;
        printf("\nEnter ID: ");
        scanf("%u",&tempid);
```

```
        if(isValidid(tempid)==TRUE)
        {
                query_stmt=(char*)calloc(200,sizeof(char));
           fname=(char*)calloc(30,sizeof(char));
           lname=(char*)calloc(30,sizeof(char));
           fprintf(stdout,"Enter New First Name: ");
           fflush(stdin);
           gets(fname);
           fprintf(stdout,"Enter New Last Name: ");
           fflush(stdin);
           gets(lname);
           fprintf(stdout,"Enter New Age: ");
           fscanf(stdin,"%u",&age);
           sprintf(query_stmt,"update myTable set
FirstName=\'%s\',LastName=\'%s\',Age=%u WHERE ID
==%u",fname,lname,age,tempid);
           sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
      if (sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        fprintf(stdout,"\nRow is Updated");
        }
        else
        {
                fprintf(stderr,"RECORD NOT FOUND");
        }
        free(query_stmt);
        free(fname);
        free(lname);
        getch();
        return;
}
```

Code in insertdata.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
void insertdata()
{
```

```
        char *fname;
        char *lname;
        unsigned int age;
        char ch;
        extern sqlite3*db;
        char*zErrMsg;
        sqlite3_stmt *statement;
        fname=(char*)calloc(30,sizeof(char));
        lname=(char*)calloc(30,sizeof(char));
        char *sql_stmt=(char*)calloc(200,sizeof(char));
        do
        {
         fprintf(stdout,"Enter First Name: ");
         fflush(stdin);
         gets(fname);
         fprintf(stdout,"Enter Last Name: ");
         fflush(stdin);
         gets(lname);
         fprintf(stdout,"Enter Age: ");
         fscanf(stdin,"%u",&age);

        sprintf(sql_stmt,"insert into myTable (FirstName, LastName, Age) values
(\'%s\', \'%s\',%u)",fname,lname,age);
        sqlite3_prepare_v2(db,sql_stmt, -1, &statement, NULL);
    if (sqlite3_exec(db, sql_stmt,0, 0, &zErrMsg) == SQLITE_OK)
    {
                fprintf(stdout,"\nNew Record inserted");
                fprintf(stdout,"\nDo you want to continue Y?: ");
                fflush(stdin);
                ch=getchar();
        }
        else
        {
          fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
          sqlite3_free(zErrMsg);
          ch=getchar();
    }
        }while(ch=='y'||ch=='Y');
```

```
        free(fname);
        free(lname);
        free(sql_stmt);
        return;
}
```

Code in deletedata.c

```
#include <stdio.h>
#include <malloc.h>
#include "sqlite3.h"
#define TRUE 1
#define FALSE 0
void deleteallrow()
{
        //delete all record from table
   extern sqlite3*db;
        char*zErrMsg;
        sqlite3_stmt*statement;
        const char *query_stmt="delete from myTable";
        char *sql_stmt=(char*)calloc(200,sizeof(char));
        sqlite3_prepare_v2(db,query_stmt, -1, &statement,NULL);
        if(sqlite3_exec(db, query_stmt,0,0,&zErrMsg)== SQLITE_OK)
        {
          if (sqlite3_step(statement) == SQLITE_ROW)
            fprintf(stdout,"\nrecords are deleted");
     else
          fprintf(stderr,"\nNO Rows are selected");
    sqlite3_finalize(statement);
 }
 else
        fprintf(stdout,"\nError:%s",sqlite3_errmsg(db));
   sqlite3_free(zErrMsg);
 getch();
 return;
}
void deletesinglerow()
{
```

```
 int isValidid(int);
 unsigned int tempid;
 extern sqlite3*db;
 sqlite3_stmt*statement;
 char*zErrMsg;
 char *query_stmt=(char*)calloc(200,sizeof(char));
 printf("\nEnter ID: ");
 scanf("%u",&tempid);
 if(isValidid(tempid)==TRUE)
 {
        sprintf(query_stmt,"DELETE FROM myTable WHERE ID ==
%u",tempid);
        sqlite3_prepare_v2(db,query_stmt, -1, &statement, NULL);
        if(sqlite3_exec(db, query_stmt,0, 0, &zErrMsg) == SQLITE_OK)
        {
                printf("\nRecord is Deleted");
                getch();
                return;
   }
 }
 fprintf(stdout,"\nRecord not Found");
        getch();
        return;
}
void deletedata()
{
        int option;
        printf("\nDelete All Rows.....1: ");
        printf("\nDelete single Row...2: ");
        scanf("%d",&option);
        if(option==1)
    deleteallrow();
        else if(option==2)
          deletesinglerow();
        else
                fprintf(stderr,"\nInvalid Option: ");
    getch();
        return;
```

}

Code in droptable.c.c

```
#include <stdio.h>
#include "sqlite3.h"
void droptable()
{
   extern sqlite3*db;
   char*zErrMsg;
        char *sql_stmt="drop table myTable";
        if(sqlite3_exec(db,sql_stmt, NULL, NULL, &zErrMsg)==SQLITE_OK)
        {
                fprintf(stdout,"\ndeleted table");
               sqlite3_free(zErrMsg);
                getchar();
                return;
        }
        else
        {
                fprintf(stderr,"\nError:%s",sqlite3_errmsg(db));
                sqlite3_free(zErrMsg);
                getchar();
                return;
        }
}
```

26/08/2015

Multithreading

→ Multithreading means simultaneously running multiple processes at a time.

→ Multithreading is an ability of an operating system which can allow multiple processes simultaneously.

→ Multithreading is possible in multiprocess ~~based~~ architecture based Operating System only.

→ Using multithreading we can develop parallel applications.

→ When we are working with DOS O.S. it doesnot support multithreading.

→ DOS is a single user, single task based Operating System. So doesnot allow to run multiple processes.

Process :-

→ When a compuley pruegram is loaded into the memory for execution then it is called process.

Thread :-

→ A thread is a sequence of instructions within a process which can be executed independently from other code.

→ In C & C++ when we require to develop multiprocessor based application then go for pThread (POSIX)

library.

→ POSIX means Portable Interface of Unix operating system.

→ For Unix and Linux based compilers this library is built In.

→ When we are working with Windows based compiler then only explicitly we require, to config.

Integration of pThread library with DevC++ IDE.

→ Open devpaks.org website and select POSIX category.

→ Click on pthread_w32 library version.

→ Wait until download is finished
-then run package installer file. They automatically pthread library is installed.

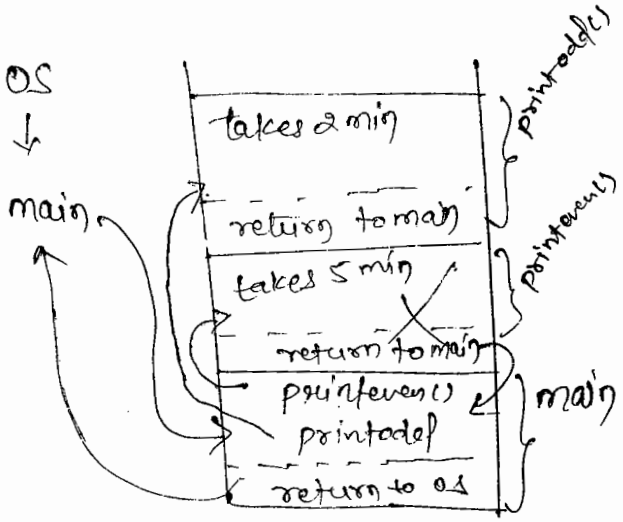
Single process based C application

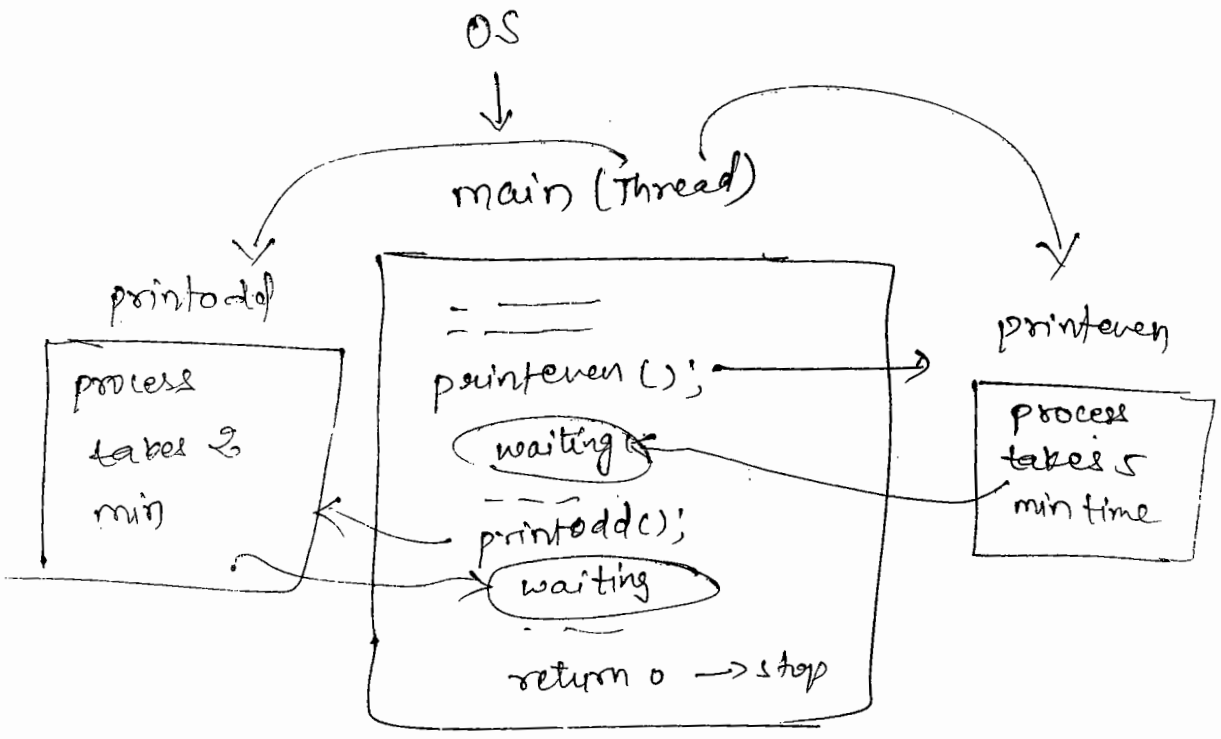
→ In normal C applications at a time we can execute one process only. or one thread can only be evaluated.

→ By default application is started from main() and as per the requirement- main() can be call any other functions.

⇒
```
# include <stdio.h>
void printeven()
{ // printing all even no from 2 to 123456789
}
void printodd()
{ // printing all odd numbers from 1 to 123456
```

```
}
int main()
{
    ----
    printeven();
    ----
    printfodd();
    ----
    return 0;
}
```





total takes 7 min time

→ As per above observation application is started from main() and it is handled by thread.

→ As per the requirement main() is calling printeven() so thread is replaced with printeven() code so automatically main() goes in waiting status (Idle state) for 5 min. of time

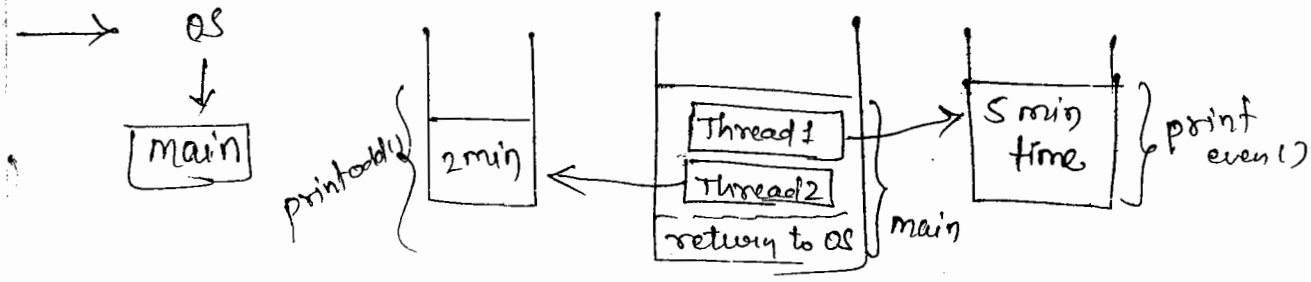
→ When the main() is in waiting status no any instructions of main() can be executed.

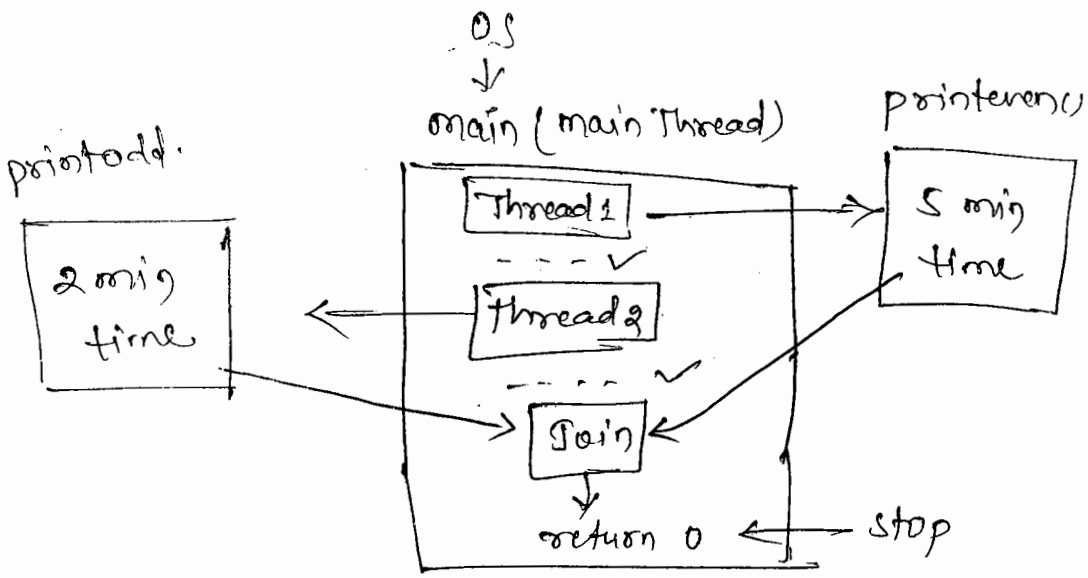
→ After 5 minutes of time thread is shifted back to main() and again it is replaced with printodd().

→ Where print-odd() is in the execution then main() will be in waiting status for 2 minutes of time.

→ To complete the process completely it takes 7 minutes ~~to~~ which is possible to finish in 5 minutes with the help of multiprocessing.

Multiprocess based C Application





→ Open Dev C++ IDE and create a new console based C project with the name pthreadEx.

→ Save the project template into any specific drive.

→ Go to project menu and select project option, click on parameters tab then select Add library or object button.

→ Select libpthreadGC2.a file from C:\Program Files\Dev-Cpp\Lib

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

void * printeven (void * msg)
{
  int i;
  char* data = (char*) msg;
  for (i = 2; i<=1234; i+=2)
  printf ("\n%s=%d", data, i);
}

void * printodd (void *msg)
{
  int i;
  char *data = (char*) msg;
  for (i = 1; i<=1234; i+=2)
  printf ("\n%s:%d", data, i);
}

int main (int argc, char *argv[])
{
  pthread_t thread1, thread2;
  int thread1_ercode, thread2_ercode;
```

```
thread1_ercode = pthread_ create (& thread1, NULL,
printeven, (void *) " from even");
if (thread1_ercode != 0)
{
  printf ("\n Error in thread 1");
  return 1;
}
thread2_ercode = pthread_ create (& thread2,
NULL, printodd, (void *) " from odd");
if (thread2_ercode != 0)
{
  pthread_cancel (thread1);
  printf ("\n Error in thread 2");
  return 1;
}
  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);
  return 0;
}
```

Graphics

→ When we are applying visual affects in DOS based application then it is called Graphics.

→ Graphics related C applications doesnot works on High End processors because of resolution problem. (XP is supported)

→ When we are working with Graphics related applications then recommended to go for <graphics.h> header file.

→ <graphics.h> provides all the predefined functions prototypes which are related to graphics.

→ When we are working with graphic related functions mandory to find out EGAVGA.BGI file location.

→ Generally this file is available in C:\TC\BGI directory.

→ EGAVGA.BGI file provides graphics related all resources.

→ When we are working with graphics related applications then initially we require to convert Dos mode into graphics mode.

→ After completion of the program at end of the application we require to convert graphics mode into Dos mode.

initgraph() :-

→ Using this predefine function we can initialize the graphics.

→ At the time of initialization we will get initialization related errors also.

→ If initialization is done properly then we will get the return value grOk. or else negative return value is returned.

→ graphresult() :-

→ Using this predefine function we can find initialization code i.e. success or failure.

⇒

⇒ grapherrormsg():-

→ Using this predefined function we can find graphics initialization related error messages.

→ This function requires one argument of type Integer i.e. error code value.

⇒ cleardevice():-

→ Using this predefined function we can clear the data from console or graphics mode.

⇒ closegraph():-

→ Using this predefined function we can close the graphics i.e converts graphics mode to DOS mode.

setcolor()

→ Using this predefined function we can change the color in graphics mode.

— setbkcolor()

⇒ Using this predefined function we can change background color.

⇒
```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <stdlib.h>
#include <dos.h>
#include <process.h>

int main()
{
 int gdriver = DETECT, gmode, errcode;
 int radius = 200, i=0, flag=0;
 int midx, midy;
 clrscr();
```

```
initgraph(&gdriver, &gmode, "C:\\TC\\BGI");
ercode = graphresult();
if (ercode != grOk)
 {
  printf("Error : %s", grapherrormsg(ercode));
  getch();
  return 1;
 }
 midx = getmaxx()/2;
 midy = getmaxy()/2;
 while (!kbhit())
  {
    setcolor(i);
    setbkcolor(15);
   if (flag==0)
      circle (midx, midy, radius--);
   if (flag==1)
      circle (midx, midy, radius++);
   delay (100);
   if (radius==0)
    {
      flag = 1;
```

```
        cleardevice();
    }
    if (radius == 200)
    {
        flag = 0;
        cleardevice();
    }
    ++i;
    if(i>15)
    i=0;
}
closegraph();
return 0;
}
```

printing without any printing function.

=> int main()

{

~~char far *ptr = (char far *) 0XB8000000;~~

0XB8000000;

char far *ptr = (char far *) 0XB8000000;

*(ptr + 0) = 'B';

*(ptr + 1) = 1;

*(ptr + 2) = 'A';

*(ptr + 3) = 2;

*(ptr + 4) = 'L';

*(ptr + 5) = 3;

*(ptr + 6) = 'U';

return 0;

}