RS = 150/=

PROGRAMMING IN C'

3/9/12

- C Programming Language -

Software :-

- As per the industrial standard, a digitalized automated process is called as Software. When the graphics interface is available, then it is called digitalized; without human interaction. If the process is completed, then it is called automated process.
- A software is a collection of programs. A program is a set of instructions, which is designed for a particular task.
- 'n' no of programs combined together like a single unit, that is called Software tool or software component.
- Softwares are classified into 2 types.
  1. System s/w
  2. application s/w.

① **System s/w :-**

- The s/w is designed for general purpose & doesnot contain any limitations, it is called System Software.
- O.S is an interface between component and software.
- Always O.S will utilize the features properly.

② **Application s/w :-**

- Which s/w is designed for a specific task, it is called application s/w.
- Always application software needs to be used for particular work only that is for which purpose it is designed, for same purpose it should be used.
- ex :- oracle, ms office, tally.
- oracle is an application software, which maintains data in tabulate format.
- MS-office is a microsoft product which maintains information in document format.
- tally is an application s/w which maintains account related information.

- A computer is an electronic device which accepts the data

Programming Language :-

- It is a special kind of instructions, which is used to communicate with computer.
- Programming languages are classified into two types, i.e.
  1) High level programming language
  2) Low level programming language.

1) HL Prog. Language :-

- which programming lang. is syntactically similar to english and easy to understand, is called Highlevel prog. language.
- By using HL prog. languages, we can develop user interface applications.
- ex:- C, C++, C#, VC++, JAVA, COBOL, PASCAL, OBJECTIVE-C

2) Low level prog. lang. :-

- In this programming language, instructions will be there in symbolic format and that is not to easy remember all instructions along with syntax.
- By using Low level prog. languages, we are developing device drivers
- A device driver is a system software component, which is used to communicate an application software with hardware component.
- Low level prog. lang. is also called Assembly language.
- ex:- microcontroller prog. language (8085, 8086, 8051)

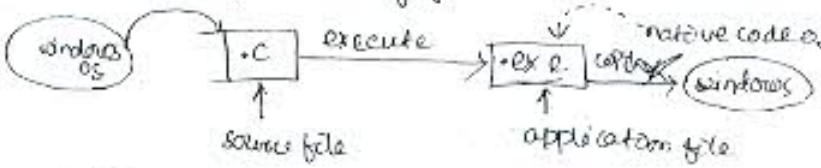
C' PROGRAMMING LANGUAGE :-

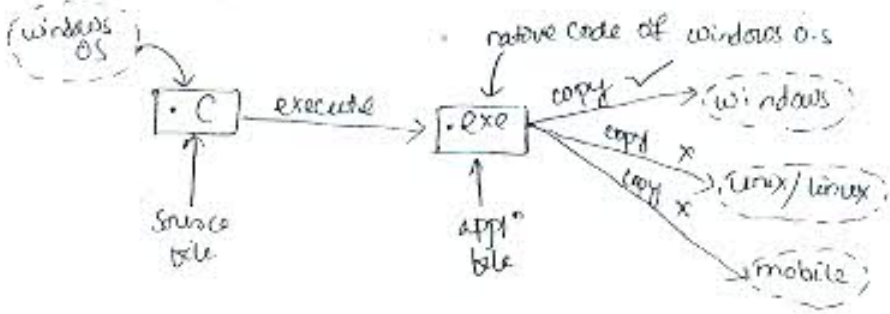
- It is a high level, procedure oriented, structured programming language.
- Procedure oriented is a concept of designing an application in module format.
- Top-down approach with blocks format, is called Structured programming language

Advantages of C'

① portability :-

It is a concept of copying the instructions from one system to another system





← Platform dependency →

- As per above observation, when we are copying .exe file to any other system, which contains windows O.S., then it works properly, because native code of the exe

and operating system is same.

- Same .exe file, when we are copying to any other operating systems like unix/Linux, then it does not work because native code is different.

- C' prog. language is platform dependent and machine independent programming language (doesnot depend on hardware components of the system).

- When we are carrying the application file from one operating system to another O.S, if it works properly, then it is called platform independent if it doesn't work then it is called platform dependant.

② Modularity ⇒

- It is a concept of dividing an application into sub modules. i.e. procedure oriented approach.

③ Mid level ⇒

- C' Programming language can support high level instructions with the combination of low level programming also, that's why it is called middle level prog. language.

④ Simple ⇒

- This prog. language syntactically similar to english and limited concepts are available.

Applications of C' :-

In software field, 90% applications are working with the help of C'-programming language only, like :-

① System software designing, i.e. O.S. and compilers
② Application s/w designing, i.e. databases and excel sheets
③ Any complex mathematical equations can be evaluated.
④ Graphics related applications, i.e. PC and mobile games.

History of C' :-

- The programming language term is started in the year of 1950's with language called fortan.

- From fortan language, the next prog. lang. evaluated called Algol, i.e. an acronomy for algorithmic language.

- The begining of C' is started in the year of 1968 with the language called BCPL (Basic Combine Programming Language), which is evaluated by Martin Richard.

- In the year of 1970, from BCPL one more prog. language evaluated by "Ken Thomson", i.e called B-language (Basic language).
- In the years of 1972, C' prog. language evaluated by "Dennis Ritchie" at "AT & T Bell laboratories" for developing
- In the year of 1988, C' prog. language standardized by ANSI, that version is called ANSI-C version
- In the year of 2000, C' prog. language standardized by ISO, that version is called C99.

6/9/12

Characteristics of C':-

① Keywords.
- It is a reserved word, some meaning is already allocated to that word and that meaning can recognized by the compiler.
- In C' Prog. language, total no. of keywords are 32.
- ex:- while, for, if, else, break, const, short

② Operators :-
- It is a special kind of symbol, which performs a particular task.
- In C-prog. language, total no. of operators are 44.
- ex:- +, -, *, /, %, <<, &

③ Separators :-
- By using separators, we can separate an individual unit called token.
- In C- prog. language, total no of separators are 14.
- ex:- ' ', ;, :, ` `, " ", {, }, space

④ Constants :-
- It is a fixed one, never be changed during the execution of the program.
- constants are classified into 2 types, i.e. :-
  - i) alphanumeric constants.
  - ii) numeric constants.
- (alphanumeric constant) :-
  - By using alphanumeric constant, we can represent 52 alphabets, 44 operators, 14 separators, 0-9 numeric characters constants & some special kind of symbols.
  - Under alphanumeric constants, we are having only one type of data value, i.e. char.
  - In C' prog. language, total no. of characters are 256.
  - When we are working with the character, character representation must be with single quotation mark (' ') only.

- Within the single quotation mark, any content is called character constant.
  - ex:- 'A', 'a', '+', '7', '@';

↳ (numeric constants)

- By using numeric constant, we can represent value type data.
- Numeric constants are classified into two types. such as:-
  - ✱ int, float.
- When we need to represent the values without any fractional parts, then go for an integer type.
  ex:- 10, -12, 100, 150, 147 ...
- When we need to represent the value data with fractional part, then go for float type.
  ex:- 17.5, 187.25, -17.618 ..

note ↳ int, float and char are called basic datatypes or basic data elements, because any data is a combination of these three types of constant types only.

⑤ predefined function :-

↳ The functionality of the function is already implemented, which is available along with compilers.

↳ As a programmer, whenever we need to perform the task, then simply we require to call predefined function

↳ ex:- printf(), scanf(), clrscr(), getch(), atoi(), memset(), evct(), gettime(), getdate(), delay(), rename(), ftell(), strncat()

⑥ Syntax :-

↳ Grammar of specific programming language is called syntax.

↳ The basic syntax of the C language is every statement should ends with semicolon (;).

OPERATORS :-

- In C - Prog. language, total no. of operators are 44 and depending on the operands, these complete operators are classified into three types, such as:-

i) unary,
ii) binary, &
iii) teritary.

- When we are working with unary operator, this requires only one argument, binary operator will take two arguments and teritary operator will require three arguments.

① assignment Operator :-

- It is a binary catagory operator, which requires two arguments i.e. left, right side argument.

- When we are working with binary operator, if any one of the argument is missing, then it gives an error.

- when we are working with assignment operator, then left side argument must be variable type only, and rightside argument is optional.

syntax: `L = R ;`

ex:

`a = 10`	error: Statement missing
`a = 10;`	10
`a = ;`	error: R value required
`= 15;`	error: L value required.
`10 = 20;`	error: L value required.

(10 = 20; — variable type)

```
{ a = 10;
{ b = a;
```

② Arithmatic Operator :-

binary operators:

*	multiplication
/	division
%	Reminder
+	Add
-	Subtraction

✓ All arithmatic operators are binary operators only.

✓ In implementation, whenever an expression contains more than one operator then inorder to evaluate the expression, we require to follow Priority of operators.

✓ As per the priority, which operator contains heighest priority, that one should be evaluated first, whichone contains least priority, that one should be evaluated at last.

note → whenever the equal priority is occured, if it is binary operator, then need to be evaluated towards from left to right,
✓ if it is ternary operator, then it is evaluated towards from right to left.

1.	*	/	%
2.	+	−	
3.	=		

↓

① a = 2 + 3 ; ⑤ (return value)

② a = 5 − 8 + 2 ; ④
→ 2 + 2

③ a = 8 + 2 − 5 ; ⑤
→ 10 − 5

④ a = 5+ ; error

⑤ a = 5 − ; error

⑥ a = +5 ; ⑤

⑦ a = −8 ; −8
sign

+, −
↳ unary - sign opr
↳ binary - arithmetic opr.

sign

- When we are working +, − symbol, then it works like a binary, unary operator also.
- When the symbol is available before the operand, then it indicates sign of value that is unary operator.
- When the symbol is available after the operand, then it indicates the arithmatic operation, i.e. binary operator.
- Ex:- a = 2 + 3 * 2 + 3
= 2 + 6 + 3
= 8 + 3 = 11

Ex:- a = $\frac{2*3}{6} + \frac{2*3}{6}$?

value	return value
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
-5/-2	2

- Operator behaviour is always variable independent only that is dependent of variable of type, the behaviour of operator is not changed.
- operator behaviour is always operand dependent. only that is what type of i/p values we are providing, according to the type, the behaviour of the operator will change.
- When we are working with binary operator, if both arguments are integer, then return value is integer, any one of the arguments is float or both are float, return value will be float type.

  int, int → int
  int, float → float
  float, int → float
  float, float → float

- On division operator, output sign will depend on the numerator value sign and denominator value sign also.
- On any one of the argument is -ve, the o/p is -ve, if both are -ve, then return value will be +ve.
- On division operator, when the numerator value is less than of denominator value, then return value is zero.

⇒ modulus operator (%) :-

- modulus operator returns remainder value.
- On modulus operator, output sign will depend on numerator value sign only.
- On modulus operator, when the numerator value is less than denominator value, then it returns numerator value as the return value

(i) 12 % 6 = 0
(ii) a = 15 % 2 = 1
(iii) a = 23 % 12 = 11
(iv) a = 18 % 9 = 0
(v) a = 27 % 7 = 6
(vi) a = 10 % 3 = 1

value	return value
47 % 5	2
47 % -5	2
-47 % 5	-2
-47 % -5	-2
5 % 2	1
2 % 5	2

(i) a = 15 * 30 / 5 % 2 ;
→ = 45.0 / 5 % 2
= 9.0 % 2 (error : float, int)

→ modulus operator is only applied on integer type.

note → When we are working with modulus operator, it requires two arguments and both arguments must be only integer type.
→ we can't apply for float data type.

- In implementation, when we require to find remainder value of float datatype, then go for fmod() or fmodl().
- fmod() or fmodl() is declared in math.h. By using this function, we can find the remainder value of float datatype.

```
value = fmod(n,d);
value = fmodl(n,d);
```

1. a = 158 % 10 ; 8	a = 158/10; 15
2. a = 124 % 10; 4	a = 124/10 ; 12
3. a = 87 % 10 ; 7	a = 87/10 ; 8
4. a = 35 % 10 ; 5	a = 35/10 ; 3
5. a = 10 % 10 ; 0	a = 10/10 ; 1
6. a = 5 % 10 ; 5	a = 5/10 ; 0

- In implementation, when we need to extract last digit from given number, then go for mod of 10 (% 10), if we need to remove the last digit, then go for divided by 10 (/10).

⑧ Relational and Logical operators :-

- In C and C++, all relational and logical operation returns 1 or 0.
- If expression is true, then returns with '1', if expression is false, then returns with zero.
- Every non-zero is called true, when the value becomes zero, it is false.

T	f
1	0
-5	
0.8	
-0.01	

- Relational operators are :- <, >, <=, >=, ==, !=
- Logical operators are :- &&, ||, !

Priority :-

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

ex:

① a = 275 ; (0)

② a = 5<8 ; (1)

③ a = 37271 ; (0)

171

④ a = 5<870 ; (1)

⑤ a = 37270 ; (1)

⑥ a = 3727<1 ; (1)

1>1<1

0<1 ; 1

⑦ a = 372717071 ; (1)

1>1>07-1

0>07-1

07-1 → 1

8/9/12

ex: ① a = 2<=2<2 ;

= 1<2

= (1)

② a = 57=8<=0 ;

= 0<=0

= (1)

③ a = 2>5<=0>=1

= 0<=0>=1

= 1>=1 = (1)

==

④ a = 2==5 ; ⑤ a = 8==8 ; (1)

a = 0

⑥ a = 1==2<5 ;

= 1==1

= (1)

⑦ a = 275==2<5 ;

= 0==1 ;

= (0)

⑧ a = 5<8==275 ;

= 1==0

= (0)

⑨ a = 2!=5 ; (1)

⑩ a = 5!=5 ; (0)

⑪ a = 275!=1 ;

= 0!=1 ;

= (1)

⑫ a = 5<8!=2<5

= 1!=1

= (0)

⑬ a = 1!=175

= 1!=0

= (1)

⑭ a = 275!=5

= 0!=5

= (1)

ex:- ① a = 275 == 1 != 0

= 0 == 1 != 0

= 0 != 0

= (0)

② a = 5 == 5 > 2 != 1 ;

= 5 == 1 != 1

→

= 0 != 1

= (1)

③ a = 8 > 8 == 2 != 5 > 8 ;

= 1 == 2 != 0 ;

→

= 0 != 0 = (0)

Logical operator :-

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

&&

TT → (T)

TF, FT, FF → (F)

||

TT, TF, FT → T

FF → (F)

!

T → F

F → T

(&&)

ex:- ① a = 275 && 2 < 5

= 0 && 1

= (0) (F && T)

② a = 5 < 8 && 8 > 9

= 1 && 0

= (0) (T && F)

③ a = 275 && 5 > 8

= 0 && 0

= (0) (F && F)

④ a = 5 > 2 && 2 < 5

= 1 && 1 = (1)

⑤ a = 2 > 2 && 1 <= 5

= 1 && 1 = (1)

⑥ a = 1 != 2 < 5 && 0 != 2 > 5 ;

= 1 != 1 && 0 != 0 ;

= 0 && 0 = (0)

⑦ a = 275 != 1 && 8 > 5 != 0

= 0 != 1 && 1 != 0

= 1 && 1 = (1)

⑧

OR(||)

① a = 2>5 || 2<5
= 0 || 1
= ① F|T

② a = 8>5 || 5<2
= 1 || 0
= ① T||F

③ a = 2<5 || 8>5
= 1 || 1
= ① T||T

④ a = 1 != 2<5 || 0 != 2>5
= 1 != 1 || 0 != 0
= 0 || 0 = ⓪

⑤ a = 2<5 != 0 || 1 != 2>5
= 1 != 0 || 1 != 0
= 1 || 1 = ①

✓ When we are working with logical AND operator (&&), it requires two arguments, i.e. left and right side argument
✓ On Logical AND, both expressions are true, then the return value will true, any one of the expression is false, then return value is false.
✓ On Logical OR operator (||), any one of the expression is true, then return value will be true, if both are false, then return value is false

not(!)

① a = !5; ⓪

② a = !0; ①

③ a = !1 != 1
= 0 != 1 = ①

④ a = !(2>5)
= !0 = ①

⑤ a = !(2>5 && 5<2)
= !(0 && 0) = !0 = 1
= ~~①~~

- C programming language is a case-sensitive language, i.e. upper case & lower case content, both are different.
- All existing keywords & predefined functions are available in lower case only.

(ex1)
```
void main ()
{
    Printf ("welcome");
}
```
o/p: welcome

extension → .c ✓
.cpp ✗

✓ when we are working with any C program, by default standard & console-related I/O functions are including automatically, that's why doesnot require to go for stdio.h & conio.h using .c extension only.

✓ void is a keyword, which indicates, return type of the function.

✓ main() is an identifier, which indicates start up point of the application.

✓ '{' indicates that instruction block is started, '}' indicates that instruction block is ended.

✓ All the statements of the program should be within curly braces ("{ }") only.

Printf ()

- It is a predefined function, by using this function we can print the data on console.
- Scientific name of the monitor is called Console.
- When we are working with the printf(), it can take any no. of arguments, but first argument must be within the double quotes (" ") only and every argument should be separated with comma (,).
- Within the double quotes, whatever we pass, it brings as it is, if any format specifiers are there, then copying the type of values.

Format Specifiers:-

- Format specifiers will decide, what kind of data is printed on console.

%d --> int
%f --> float
%c --> char

TOKEN :-

- Smallest unit in program or an individual unit in program is called token.
- A C program is a collection of tokens.
- Token can be keyword, operator, separator, constant and any other identifiers also.
- When we are working with the tokens, we can't split or break the token, but between the tokens, we can give 'n' no. of spaces, tabs and newline characters.

ex:-
```
vo id main ()
{
    pr int f ("welcome");
}
```
o/p :- Error 6 tokens can't be broken.

Ex:-
```
void
main
(
)   {
printf
("welcome")
;
}
```
O/P :- welcome

1. printf (" HELLO "); HELLO
2. Printf ("... Hello"); ... Hello
3. printf (" ... Hello ... "); ... Hello ...
4. printf (" %d welcome %d ", 10, 20); Error : function call missing. (no space)
5. printf (" %d welcome %d", 10, 20,); Error : function call missing.
6. printf (" %d welcome %d", 10, 20) Error : Statement missing.

10/9/22

7. Printf (" %d welcome %d", 10, 20); // 10 welcome 20
8. printf (" %d %d %d ", 10, 20, 30); // 102030
9. printf (" %d %d %d ", 10, 20, 30); // 10 20 30
10. printf (" %d, %d, %d", 10, 20, 30); // 10, 20, 30
11. Printf (" 2+3 = %d", 2+3); // 2+3 = 5
12. printf (" %d * %d = %d", 2, 3, 2*3); // 2 * 3 = 6

✓ In printf() statement, except format specifiers & special characters, rest of all contents will be printed like that only.

13. printf (" %d %d %d ", 10, 20); // Garbage/Junk (10 20)

✓ In printf() statement, when we are passing an extra format specifier which doesnot contain corresponding value, then it prints some unknown or undefined value called Garbage/Junk value.

14. Printf (" %d %d ", 10, 20, 60); 10 20

✓ In printf() strmt, when we are passing an extra value which doesnot contain correspondent format specifier, then it leaves that value.

15. Printf (" Total Sal = %d ", 15,000); // Total Sal : 15
16. printf (" Total sal = %d ", 25000); Total Sal = 25000

If. printf ("%d, %d", 2>5, 2<8) ; // 0, 1

(ex) void main ()
{
    a = 10 ;
    printf ("a = %d", a) ;
}

op :- error : undefined symbol 'a' .

(ex) void main ()
{
    int a ; // declaration
    a = 10 ; // assigning
    printf ("a = %d", a) ;
}

op: a = 10

Variable Declaration :-

- A name of the memory location is called variable.
- Before using any variable in the program, it must be declared first.
- Declaration of variable means, needs to mention datatype and name of identifier.
- In C-prog. language, variable declarations must be existed on top of the programs after opening the curly brace ("{") before writing the first statement.
- In variable declarations, name of the variable must start with alphabet or underscore only.
- In variable declaration, maximum length of a variable name is 32 characters, after 32 characters, compiler will not consider the remaining characters.
- In variable declaration ; existing keywords, operators, separators and constant values are not allowed.
- In variable declaration, atleast single space should be required between datatype and variable name.
- when we are declaring multiple variables of same datatype, then it is required to go for comma (,) as a separator.
- Syntax :- Datatype variable ;

1. int a ; error
2. int a error
3. int a ; Yes
4. int a b c d ; error
5. int a, b, c, d ; Yes
6. int abc, d ; Yes
7. int 1a, 1b, 1c ; no
8. int a1, b1, c1 ; Yes
9. int if ; error
10. int If ; Yes
11. int iF ; Yes
12. int _if ; Yes
13. Int total-sal ; error
14. int total_sal ; Yes
15. int _1, _2, _3 ; Yes

Hungarian notation

16. int i_EMPID ; // 'i' indicates int type
17. float f_EMPSAL ;
18. char c_gen ;

(ex) void main()
{
int i ;
float f ;
i = 5/2 ;
f = 5/2 ;
Printf ("%d %f", i, f) ;
}

o/p : 2. 2.000000

int

i = 5/2 = 2
i = 5.0/2 = 2.5

float

f = 5/2 = 2.0
f = 5.0/2 = 2.5

	value	int x ;	float f ;
①	5/2	2	2
②	5.0/2	2	2.5
③	5/2.0	2	2.5
④	5.0/2.0	2	2.5
⑤	2/5	0	0.0
⑥	-	Gr	Gr
⑦	2.0/5	0	0.4

{ decimal part contains 6 digits when displaying the output.
So: f = 5/2 = 2.000000

✓ operator behaviour is always operand dependent only. i.e. depending on the o/p value, the behaviour of the operator will be changed

✓ The return value behaviour is always variable dependent only, i.e. depending on the datatype, the return value will be converted automatically to corresponding datatype.

✓ In implementation, whenever an operator returns an integer value and we need to assign it to float variable, then data will be converted into float format automatically by adding ".0"

✓ In implementation, whenever an operator returns float data and we need to assign it to integer variable, then only decimal part will be converted into corresponding type.

✓ By default, whenever we are declaring a variable, it holds garbage value only.

CONTROL FLOW STATEMENTS :-

✓ The execution process of the program is under control of the control flow statement.

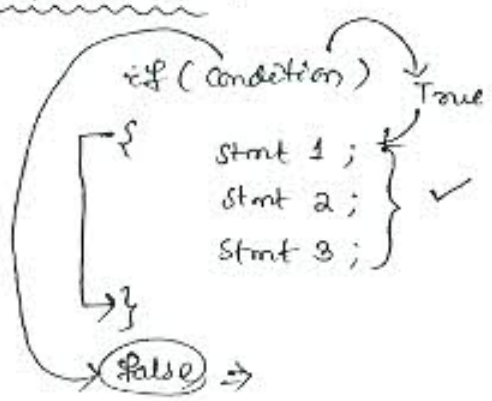
✓ In 'C' prog. language, control flow statements are classified into three types :-

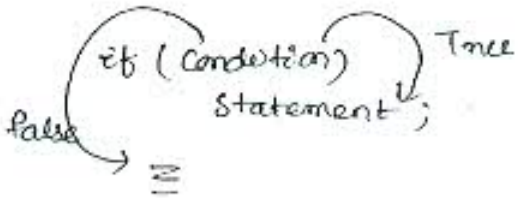
i) Selection statements
   ↳ if, else, else-if, switch
ii) Iteration statements
   ↳ while, for, do-while.
iii) Jumping statements
   ↳ break, continue, go to.

① SELECTION STATEMENT :-

- Selection statements are also called decission making statements.
- when we are working with selection statement, then if condition is true, then corresponding block will be executed, if condition false, then the block will not be executed.

- Syntax to "if" :-





- when we are having multiple statements within the condition, then it is recommended to go for body.
- For a single statement, it is not required to specify the body.
- If the body is not specified, then automatically condition part will be terminated with next semicolon.

else :-

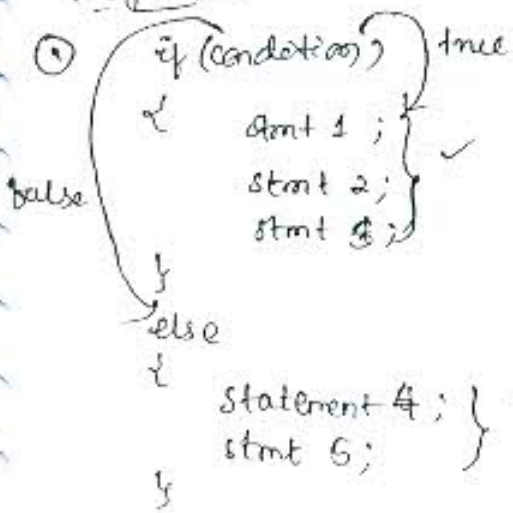
✓ 'else' is a keyword, by using else-part, we can create alternate block of if-condition.

✓ Using 'else' is always optional, it is recommendent to use, when we are having alternate block of 'if'.

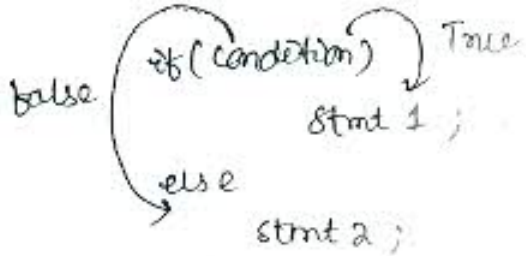
✓ when we are working with 'if and else', at given any point of time, only one block will be executed, ie. when if-cond" is false, else part is executed, if cond" is not true, then else-part will be ignored.

✓ Syntax :-

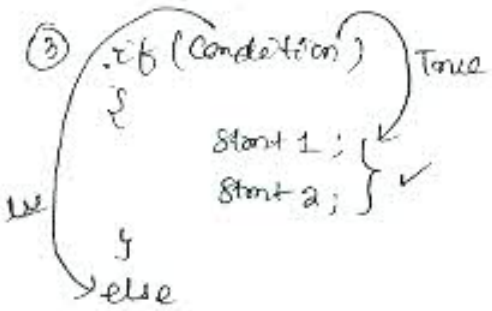
①



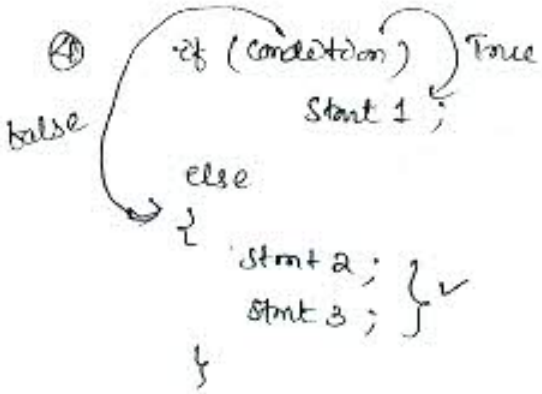
②



③



④



Note

Ex1)

```
void main()
{
    Printf("welcome");
    if(2>5)  // if(0)
    {
        printf("A");
        Printf("B");
    }
    Printf("Hello");
}
```

o/p: welcome Hello

Ex2)

```
void main()
{
    printf("NIT");
    if(1!=2>5 && 0!=2<5)
    {
        Printf("B");
        printf("A");
    }
    printf("welcome");
}
```

⇒ 1 != 2>5 && 0 != 2<5
⇒ 1 != 0 && 0 != 1
⇒ 1 && 1 ⇒ (1)
if(1) → true.

o/p:- NIT BA welcome

Ex3)

```
void main()
{
    Printf("Hello");
    if(!5)
        Printf("welcome");
    printf("B");
    Printf("A");
}
```

o/p:- Hello BA

- When we are placing multiple statements without using body, then the scope will be terminated with next semicolon ; i.e under the condition, only one statement can be placed

(ex4)
```
void main()
{
    printf("A");
    if(5<2 && 2>5)
    {
        printf("NIT");
        printf("Welcome");
    }
    else
    {
        printf("Hello");
        printf("Naresh");
    }
    printf("B");
}
```

0 && 0 ⇒ 0

O/P :- AHELLONareshB

(ex5)
```
void main()
{
    Pf("NIT");
    if(!1!=1)
    {
        Pf("A");
        Pf("B");
    }
    else
    {
        printf("Java");
        printf("C");
    }
}
```

!1! = 1
⇒ 0! = 1 if(1) → true
⇒ (1)

O/P :- NIT A B

(ex6)
```
void main()
{
    printf("Wellcome");
    if(5>2!=0 && 1!=2>5)
    {
        Printf("B");
        Printf("A");
    }
    else
        printf("Java");
    Printf("C");
    Printf("Hello");
}
```

O/P :- WellcomeBAC Hello

- when the body is not specified for else part, then automatically scope of the else will terminate with the next semicolon, i.e. first statement only placed under the else part.

(ex7)
```
void main()
{
    printf("Java");
    if(578!=1)
        printf("B");
    printf("A");  // error
    else
    {
      printf("welcome");
      printf("Hello");
    }
}
```

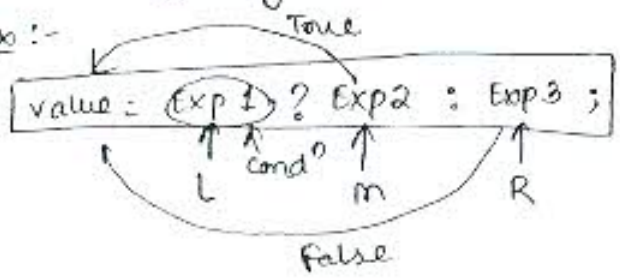
O/P:- error : misplaced else

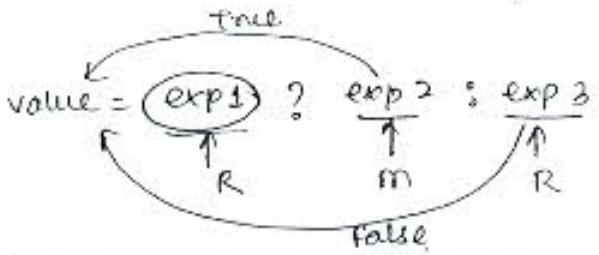
- Placing 'else' is always optional, but if we are constructing the 'else' part, it must start immediately after 'if' scope only; i.e. where the scope of the 'if' is closed, there itself 'else' part must be started.

Conditional Operators :-

- Conditional operators are ternary category operators.
- Ternary category means, it requires three arguments, i.e left, middle & right side arguments.
- When we are working with conditional operator, if expression is true, then returns with middle argument; if expression is false, then returns with right side argument.
- When we are working with conditional operator, need to satisfy following condition :- i.e
  1. no. of '?' and ':' should be equal
  2. every colon (:) should match with just before the '?'.
  3. every colon (:) should be followed by '?' only.

- Syntax :-





- If 'exp1' is true or left side data is non zero, then 'exp2' or 'middle value' will return
- If 'exp1' is false or left side data is zero, then 'exp3' or 'right side data' is the return value.

① a = 10 ? 20 : 30 ; 20

② a = (5<2) ? 20 : 30 ; 30

⇒ a = 0 ? 20 : 30

③ a = 2<5 && !0 ? 10 : 20 ; 10

1 && 1

1

④ a = 10 ? 20 ; error

⑤ a = 10 : 2 ? 5 ; error

⑥ a = 2<5 ? 10 : 20 : 30 ; error

⑦ a = 5?2 ? 10 : 20 : 5<8 ? 30 ; error

⑧ a = 275 ? 10 : !5! = 275 ? 20 : 30 ; (30)

275 ? 10 : 0 ? 20 : 30

275 ? 10 : 30

0 ? 10 : (30)

upto 1st ? → L

after 1st ? before 1st : → m

after first :, before ; → R

a = 275 ? 10 : !5! = 275 ? 20 : 30

L 1 m 1 2 2 R

!5! = 275 ? 20 : 30

L m R

0 ? 20 : (30)

a = 275 ? 10 : 30 ; 30

⑨ a = (2<5) ? 1 !! = 275 ? 10 : 20 : 30 ;

1 (1) L 1 2 2 m 1 R

1 !! = 275 ? 10 : 20

1 !! = 0 ? 10 : 20

1 ? 10 : 20

(10)

Solve

(10) `a = 2>5 ? 10 : 5<8!=1 ? 20 : !5 ? 30 : 40;`

`2>5 ? 10 : 5<8!=1 ? 20 : 40;`

`2>5 ? 10 : 40;`

40

`5<8!=1 ? 20 : !5 ? 30 : 40;`

`!5 ? 30 : 40`

(11) `a = 5>2!=1 ? 10 : 2<5 ? 1!=2>5 ? 20 : 30 : 40;`

`2<5 ? 1!=2>5 ? 20 : 30 : 40;`

`1!=2>5 ? 20 : 30;`

20

(12) `a = 1!=2>5 ? !5!=5 ? 8>8!=0 ? 10 : 20 : 30 : 40;`

`!5!=5 ? 8>8!=0 ? 10 : 20 : 30`

`8>8!=0 ? 10 : 20`

20

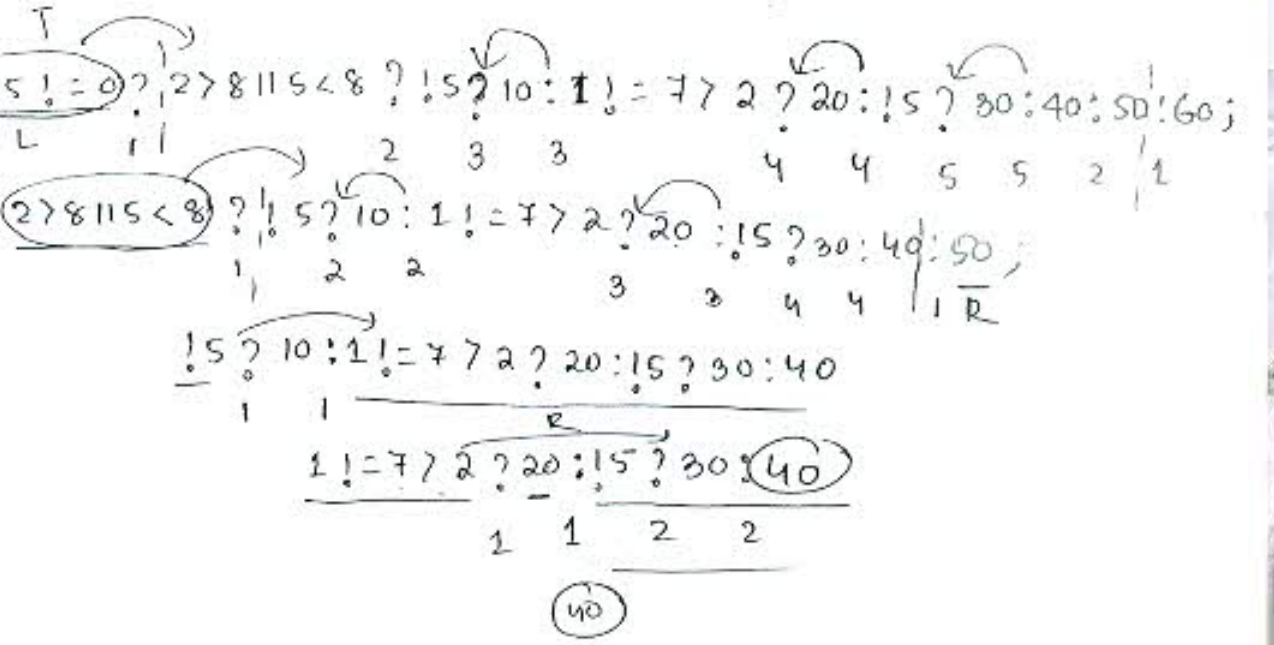
(13) `a = 5<2 ? 8>5 ? 10 : 20 : 2!=5<8 ? !8 ? 30 : 40 : 50;`

`2!=5<8 ? !8 ? 30 : 40 : 50;`

`!8 ? 30 : 40`

40

(14) a = (2<5!=0) ? 1 : (2>8||5<8) ? 2 : !5 ? 3 : 10 : 1 != 7 > 2 ? 4 : 20 : !5 ? 5 : 30 : 40 : 50 : 60;



- By using conditional operators, we can increase the execution speed of the program.
- Generally by using conditional operators, we are reducing coding part; if we are reducing coding part, then it occupies less memory, so automatically performance of the application will be increased.

(15)
```
void main()
{
    int x, y, z, min;
    x=10; y=5; z=20;
    if (x<y && x<z)
        min = x;
    if (y<x && y<z)
        min = y;
    if (z<x && z<y)
        min = z;
    printf("min value = %d", min);
}
```

o/p :- min value : 5

- In implementation, when we are having interrelated conditions, then if we are constructing the program independently, then after completion of requirement also, the compiler checks rest of all conditions.
- When we are having interrelated conditions, then always it is recommend to create in optional blocks only by using 'else' part.
- By using 'else' part, we can create only one alternate block, if we are having more than one alternate blocks, then it is recommendent to go for 'nested if-else' (or nested else-if).

- In the previous program, in place of constructing multiple conditions, we can create only single statement, which can perform same task.

e.g.: /* finding the min value using conditional operator */

```
void main()
{
    int x, y, min;
    x=10; y=5; z=20;
    min = x<y && x<z ? x : y<z ? y : z;
    printf("min value = %d", min);
}
```

Nested 'if-else' :-

- It is a concept of placing a condition part within an existing condition.
- Nested concept can be applied upto 255 blocks, but generally there is limitations.
- Syntax:-

```
if (condition 1)
{
    block 1;
}
else
{
    if (condition 2)
    {
        block 2;
    }
    else
    {
        if (condition 3)
        {
            block 3;
        }
        else
            block4;
    }
}
```

13/09/12 - According to the above syntax, if condition 1 is true, then block1 will be executed, if condetion1 is false, then control will pass to else part.

- Within the 'else', if condetion2 is true, then block2 will be executed, if it is false, then control will be passed to within nested else.
- Within the nested else, if condetion3 is true, then block3 will be executed, if it is false, then block4 will be executed.
- When we are working with nested if else at given any point of time, only one block will be executed.

(ex2) // max value using nested if else

```
void main ()
{
    int a, b, c, d, max ;
    // a=10; b=30; c=20; d=14;
    clrscr() ;
    printf ("Ents four values :") ;
    scanf ("%d %d %d %d", &a, &b, &c, &d ) ;
    if ( a>b && a>c && a>d)
    {
        max = a ;
    }
    else
    {
        if (b>c && b>d )
        {
            max = b ;
        }
        else
        {
            if (c>d )
                max = c ;
            else
                max = d ;
        }
    }
    printf (" maxium value = %d", max ) ;
  getch () ;
}
```

output :-

Enter four values : 10 20 50 30
maximum value = 50

scanf() :-

- By using this predefined function, we can read the data from user.
- "scanf()" function can take any no. of arguments, but first argument must be within the double quotes (" ") & every argument should be separated with comma(,).
- Within the " ", we require to pass proper format specifiers only; i.e. what kind of data we are reading, same type of format specifier required to use.
- When we are working with scanf(), argument list should be provided by using '&' symbol.

clrscr() :-

- By using this predefined function, we can clear entire data from console.
- Using clrscr() is always optional, but it should be placed after variable declarations only.

getch() :-

- By using this predefined function, we can hold the output screen until, we are passing any key stroke from keyboard.

comments :-

- Comments are used to provide the description about logic.
- When we are using comments, that specific part of the program will be ignored by compiler.
- On C programming language, there are 2 types of comments are possible. i.e
  1. single line comments.
  2. multiple line comments.
- By using two forward slashes i.e. "//", we can create single line comments.
- By using /* ..... */, we can provide multiple line comments.
- When we are working with multiple line comments, then nested comments are not allowed.

(Ex3) Generate an electricity bill as per the following cond^n :-

① 1 to 50 units → 1.5
② 51 to 100 units → 2.75
③ 101 to 150 units → 4.00
④ >= 151 → 5.00

Ans:

```
void main()
{
    int sno, cread, pread=0, nunits, t;
    float rps;
    clrscr();
    printf("Enter SNO:");
    scanf("%d",&sno);
    if(sno != 4235)
    {
      printf("\n Invalid SNO");
      getch();
    }
    else
    {
      printf("\n ENTER CURRENT READING :");
      scanf("%d",&cread);
      nunits = cread - pread;
      if(nunits >= 151)
      {
          t = nunits - 150;
          rps = t * 5.0;                    // 4th
          rps = rps + 50 * 4.00;            // 4th + 3rd
          rps = rps + 50 * 2.75;            // 4th + 3rd + 2nd
          rps = rps + 50 * 1.50;            // 4th + 3rd + 2nd + 1st
      }
      else if(nunits >= 101 && nunits <= 150)
      {
          t = nunits - 100;
          rps = t * 4.00;                   // 3rd
          rps = rps + 50 * 2.75;            // 3rd + 2nd
          rps = rps + 50 * 1.5;             // 3rd + 2nd + 1st
      }
      else if(nunits >= 51 && nunits <= 100)
      {
          t = nunits - 50;
          rps = t * 2.75;                   // 2nd
          rps = rps + 50 * 1.5;             // 2nd + 1st
      }
```

```
else
    ops = nunits * 1.50 ;
if (ops < 64)  // minimum amount
    ops = 64 ;
printf ("\nTotal amount = %.2f", ops ) ;
getch() ;
}
}
```

output :

```
Enter Sno : 4235
Enter current reading : 180
Total amount = 562.50
```

09/12

(Ex4) /* maximum value using conditional operator */

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

LOOPS :-

- Set of instructions given to the compiler to execute set of statements, until the condition becomes false, it is called as **Loop**.

- In implementation, whenever the repeatations are required, then in place of writing the statements again and again, recommendent to go for **Loops**.

- Generally, iteration statements are called **loops**, because the way of the repeatation will form a circle. That's why iteration statements are called **Loops**.

- In C prog. language, loops are classified into three types :-
  1. while
  2. for
  3. do while

1. **while**

Syntax to while :-

```
while (condition)
{
    stmt 1 ;
    stmt 2 ;
    stmt 3 ;
    ---------
    ---------
    ---------
    inc/dec ;
}
```

- When we are working with 'while', first compiler will check condition, if the condition is true, then control will pass within the body of loop.
- After execution of the body once again, control will pass back to the condition and until the condition becomes false, complete body will be repeated. If condition is not false, then we will get infinite loop.

Increment order	Decrement order
① Asign → min	① Asign → max
② condition → max	② condition → min
③ relation → < / <=	③ relation → > / >=
④ Add (+)	④ subtract (-)

(ex1) output :- 12345

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

(ex2) output : 10 9 8 7 6 5 4 3 2 1

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

(x3) /* Print even numbers upto a given range given by user
ex:- Enter a value : 20
2 4 6 8 ---- 20 */

```
void main ()
{
    int i, n ;
    clrscr() ;
    printf ("Enter a value : ") ;
    scanf ("%d", &n) ;
    i=2;
    while(i<=n)
    {
        printf ("%d ",i) ;
        i=i+2 ;
    }
    getch() ;
}
```

(ex1) /* Print even numbers between 2 given numbers */
/* Actor may be increment or decrement :

ex:- Enter two values : 10 20 — 10 12 14 ---- 20 } Increment
Enter two values : 9 20 — 10 12 14 ---- 20 } Increment

Enter two values : 20 10 — 20 18 16 14 ---- 10 } decrement
Enter two values : 21 10 — 20 18 16 ---- 10 } decrement

*/

```
void main ()
{
    int n1, n2
    clrscr() ;
    Printf (" Enter two values : ")
    Scanf ("%d %d", &n1, &n2) ;
    if (n1<=n2)
    {
        if (n1%2 !=0) // if (n1%2==1)
            n1 = n1 + 1 ;
        while (n1<=n2)
        {   printf ("%d", n1) ;
            n1 = n1 + 2 ;
        }
}
```

```
        else
        {
            if (n1 % 2 == 1) // if (n1 % 2 != 0)
                n1 = n1 - 1
            while (n1 >= n2)
            {
                printf("%d", n1);
                n1 = n1 - 2;
            }
        }
        getch();
    }
```

15/09/12 ex5

```
/* Fibonacci series ---
   0 1 1 2 3 5 8 13 21 34 ---- */
void main()
{
    int n, i, j, k;
    clrscr();
    printf("Enter a value : ");
    scanf("%d", &n);
    if (n > 0)
    {
        i = 0;
        j = 1;
        printf("%d %d", i, j);
        k = i + j;
        while (k <= n)
        {
            printf(" %d", k);
            i = j;
            j = k;
            k = i + j;
        }
    }
    else
        printf("Invalue should be greater than 0");
    getch();
}
```

output

Enter a value : 20

0 1 1 2 3 5 8 13

n = 20

i = 0, j = 1, k = i + j = 1

i	j	k
0	1	1
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13
8	13	

(2x-6) Without any condition & providing any space, print the output as follows:-

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 n, i;
    clrscr ();
    printf ("Enter a value : ");
    scanf ("%d", &n);
    i=1;
    while (i<=10)
    {
        printf ("\n %d * %2d = %d", n, i, n*i);
        i = i+1;
    }
    getch ();
}
```

%2d → right align, space at left

```
5 *  1 = 5
5 *  2 = 10
  :
5 *  8 = 40
5 *  9 = 45
5 * 10 = 50
```

%3d

```
5 *  1 = 5
5 *  2 = 10
  :
5 *  8 = 40
5 *  9 = 45
5 * 10 = 50
```

%-2d → left align, space at right

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
  :
5 * 9 = 45
5 * 10 = 50
```

✓ In printf statement, when we are passing '%2d' format specifier, then it indicates that, 2 digit decimal data needs to be printed, if two digits are not occured, then go for right alignment; i.e. digits will be printed towards right side & space will be printed towards left side.

✓ In printf() statement, when we are passing "%-2d", then it indicates that, 2 digit decimal value needs to be printed, if 2 digits are not occured, then applied left alignment, i.e. digit will be printed towards left side & space will be printed towards right side.

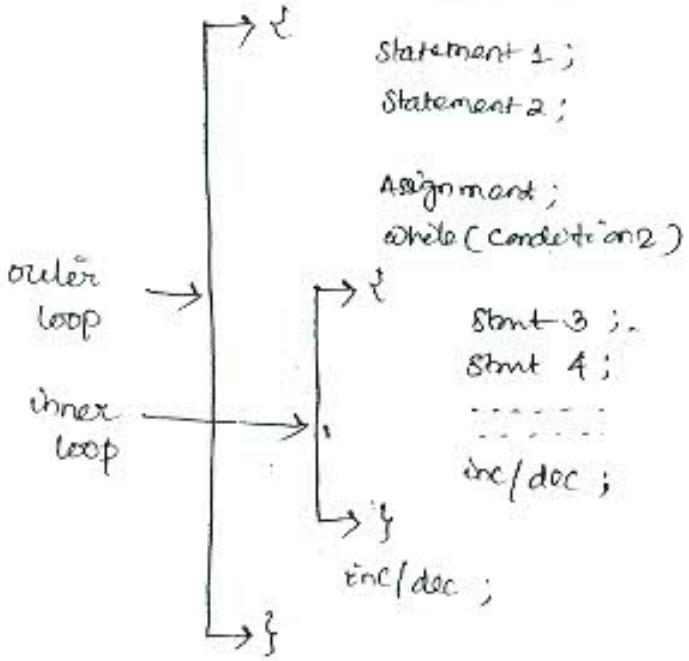
✓ In printf, when we are passing "%5.3d", then it indicates that, 5 digit decimal value need to be printed, if 5 digits are not occured, then applied right alignment, but mandatory to print 3 digits; if 3 digits are not occured, then fill with zeros.

✓ When we are passing "%-5.3d", then it indicates that 5 digit decimal value need to be printed, if 5 digits are not occured, then apply left alignment, but mandatory to print 3 digits; if 3 digits are not occured, then apply ball with zeroes.

Nested Loops:-

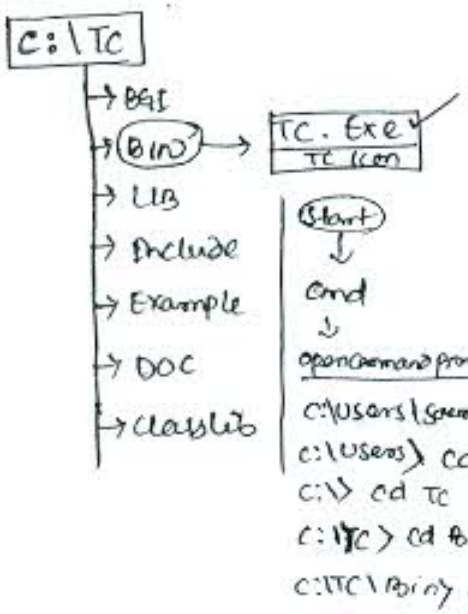
- Constructing a loop body, within another loop is called nested loop.
- On implementation, when we need to repeat a loop body itself 'n' no. of times, than go for nested loops.
- Nested loop concept can be applied upto 255 blocks.
- Syntax:-

```
Assignment;
while(condition1)
{
    Statement 1;
    Statement 2;

    Assignment;
    while(condition2)
    {
        Stmt 3;
        Stmt 4;
        .....
        .....
        inc/dec;
    }
    inc/dec;
}
```

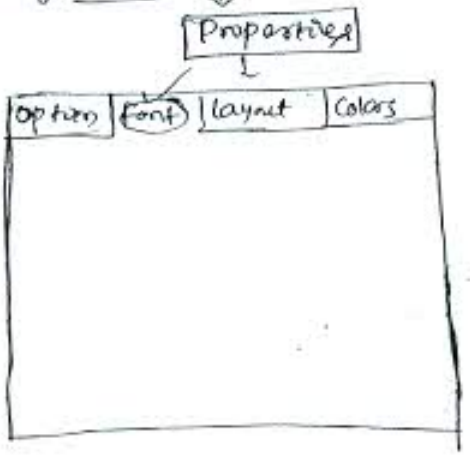


- In order to execute the nested loop, always execution process will start from outer loop body. i.e. condition 1.
- When outer loop condition is true, then control will pass within the outer loop body.
- In order to execute the outer loop body, if any while statements are occured, then it is called inner loop.
- When the inner loop is occured, then we need to evaluate inner loop condition; ie. condition 2. If the inner loop condition is true, then control will pass within the inner loop body.
- After execution of the inner loop body, control will pass back to inner loop condition, until the inner loop condition becomes false, the inner loop body will be executed, when the condition becomes false, control will pass outside the inner loop

- When control is passed to outer loop body, once again control will pass back to the outer loop condition and until the outer loop condition becomes false, complete body will be executed.



Right click on title bar of window



Pathsetting :-

16/09/22

ex7

output:

```
Enter a value : 4286

Reverse value : 6824
Sum of digits : 20
No. of digits : 4
```

```
void main ()
{
    int n, rn, sd, nd ;
    clrscr ();
    printf (" Enter a value : " );
    scanf (" %d", &n) ; rn = sd = nd = 0 ;
    while (n > 0) // +ve .. while (n < 0) -ve
    {                  // ... while (n != 0) for both value
        rn = rn * 10 + n % 10 ;
        sd = sd + n % 10 ;
        nd = nd + 1 ;
        n = n / 10 ;
    }
    printf ("Reverse no : %d", rn ) ;
    Printf (" \n Sum of digits : %d", sd) ;
    printf ("\n no. of digits : %d", nd) ;
    getche ) ;
}
```

n	rn	sd	nd
4286	0	0	0
428	0+6=6	0+6=6	0+1=1
42	60+8=68	6+8=14	1+1=2
4	680+2 = 682	14+2 = 16	2+1 = 3
0	6820+4 = 6824	16+4 = 20	3+1 = 4

note

① while (n > 0) → for +ve value only
while (n < 0) → for -ve value only.
while (n != 0) → for both values only.

② maximum value of integer = 32767

Some exceptions :-

① Enter a value = 00123
rn = 321
sd = 6
nd = 3

② n = 12300
rn = 321
sd = 6
nd = 5

③ n = 12345
rn = -11245
sd = [illegible]
nd = 5

X-8. Enter Two values : 2 4 → using nested loop.

2 * 1 = 2	3 * 1 = 3	4 * 1 = 4
2 * 2 = 4	3 * 2 = 6	4 * 2 = 8
2 * 3 = 6	3 * 3 = 9	4 * 3 = 12
. . . . .	. . . . .	. . . . .
. . . . .	. . . . .	. . . . .
2 * 8 = 16	3 * 8 = 24	4 * 8 = 32
2 * 9 = 18	3 * 9 = 27	4 * 9 = 36
2 * 10 = 20	3 * 10 = 30	4 * 10 = 40

```
void main()
{
    int n, n1, n2, i;
    clrscr();
    printf("Enter two 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();
}
```

execution :

n1 = 2

n2 = 4

n = n1 = 2

i	n	inner
1	2 3 4 (5)	2*1=2 3*1=3 4*1=4
2	2 3 4 (5)	2*2=4 3*2=6 4*2=8
	⋮	
10 (11)	2 3 4 (5)	2*10=20 3*10=30 4*10=40

outer → outer

Ex-9

output

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

i
1
2
3
4
5
6

```
void main()
{
    int i, n, 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();
}
```

n	i	in (inner)	Printf
6	1	1 2	1
	2	1 2 3	1 2
	3	1 2 3 4	1 2 3
	4	1 2 3 4 5	1 2 3 4
	5	1 2 3 4 5 6	1 2 3 4 5
	6	1 2 3 4 5 6 7	1 2 3 4 5 6

out

Ex 10

output

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

Steps

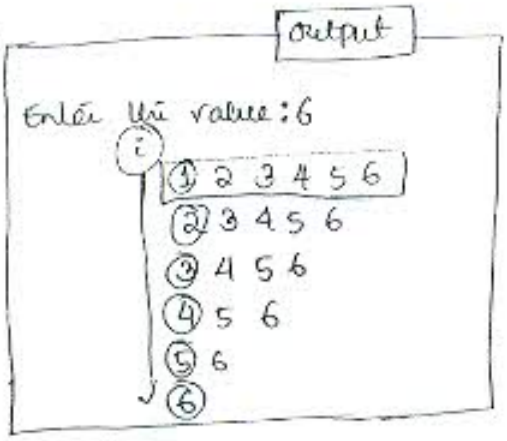
① need containing many values → inner loop write inner loop body.

② find i, then design outer loop according to the cond^n based on 'i'.

③ adjustment of the inner loop assignment for desired output.

```
void main()
{
    int i, n, 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();
}
```

ex-11



n	i	in	Printf
6	1	123456	1 2 3 4 5 6
	2	23456	2 3 4 5 6
	3	3456	3 4 5 6
	4	456	4 5 6
	5	56	5 6
	6	6	6
	7		

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

```
while (in <= n)
{
    printf (" %d ", in);
    in = in + 1;
}
```

output

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

EX12)

output

```
Enter a value = 6
65432 (1)
6543 (2)
654 (3)
65 (4)
6 (5)
(6)
```

n	i	dn	Printf
6	1	6 5 4 3 2 1 0	6 5 4 3 2 1
	2	6 5 4 3 2 1	6 5 4 3 2
	3	6 5 4 3 2	6 5 4 3
	4	6 5 4 3	6 5 4
	5	6 5 4	6 5
	6	6 5	6

out (7)

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

ex13

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

n	i	in	printf
6	1	1 2	1
	2	1 2 3	1 *
	3	1 2 3 4	1 * 3
	4	1 2 3 4 5	1 * 3 *
	5	1 2 3 4 5 6	1 * 3 * 5
	6	1 2 3 4 5 6 7	1 * 3 * 5 *
out	6		

```
void main ()
{
    int i, in, n ;
    clrscr ();
    printf (" Enter a number : ");
    scanf ("%d", &n);
    i = 1 ;
    while ( i <= 6)
    {
        printf ("\n");
        in = 1 ;
        while ( in < i )
        {
            printf (" %d", in) ;
            in = in + 1 ;
        }
        printf ("%d", in ) ;
        i = i + 1 ;
    }
    getch ();
}
```

```
while ( in < i )
{
    if (in % 2 == 0)
        printf (" * " );
    else
        printf ("%d", in);
    in = in + 1 ;
}
i = i + 1 ;
}
getch ();
}
```

Ex-14

```
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
```

n	i	in	Printf
6	1	1 2	1
	2	1 2 3	2 2
	3	1 2 3 4	3 3 3
	4	1 2 3 4 5	4 4 4 4
	5	1 2 3 4 5 6	5 5 5 5 5
	6	1 2 3 4 5 6 7	6 6 6 6 6 6

out (7)

```
void main()
{
    int i, in, n;
    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();
}
```

Ex-15

```
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
```

```
void main()
{
    int i,in,n,j;clrscr();
    printf("Enter a number:");
    scanf("%d",&n);
    i=j=1;
    while(i<=n)
    {
        printf("\n");
        in=1;
        while(in<=i)
        {
            printf("%2d",j);
            in=in+1;
            j=j+1;
        }
        i=i+1;
    }
    getch();
}
```

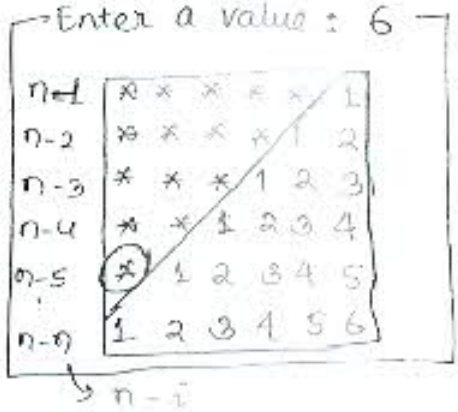
n = 6

i	in	j	Printf
1	1 2	1	1
2	1 2 3	2 3	2 3
3	1 2 3 4	4 5 6	4 5 6
4	1 2 3 4 5	7 8 9 10	7 8 9 10
5	1 2 3 4 5 6	11 12 13 14 15	11 12 13 14 15
6	1 2 3 4 5 6 7	16 17 18 19 20 21	16 17 18 19 20 21
out (7)			

Ex-16

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

n = 6
i = 1 to 6
s 1 to n



```
        in=1;
        while(in<=i)
        {
            Pf("%d", in);
            in = in+1;
        }
        i=i+1;
    }
    getch();
}
```

n = 6

i	s	n-i	in	Printf
1	12345 6	5	1 ② ↓	* * * * * 1
2	1234 ⑤	4	1 2 ③ ↓	* * * * 1 2
3	123 ④	3	1 2 3 ④ ↓	* * * 1 2 3
4	12 ③	2	1 2 3 4 ⑤ ↓	* * 1 2 3 4
5	1 ②	1	1 2 3 4 5 ⑥	* 1 2 3 4 5
6	① ↓	0	1 2 3 4 5 6 ⑦ ↓	1 2 3 4 5 6
out	⑦			

Ex1:

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

i = 6 to 1

n = 6

S = (n-i) to (n-i)

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

```
i=n;
while(i>=1)
{
    Printf("\n");
    s=n-i;
    while(s>=1)
    {
        Printf("*");
        s=s-1;
    }
    dn=i;
    while(dn>=1)
    {
        Printf("%d", dn);
        dn=dn-1;
    }
    i=i-1;
}
getch();
}
```

```
s=1;
while(s<=n-i)
{
    Pf("*");
    s=s+1;
}
```

OR

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

Ex-18 Enter a value : 6

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

Ex-19 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
```

Ex-20 Enter a value : 6

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

6 inner loops
2 outer loops

Ex-21

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

21/09/12

Answers

Ex 18

```
void main()
{
    int i, n, in;
    clrscr();
    printf("Enter a value:");
    scanf("%d", &n);
    i = 1;
    while(i<=n)
    {
        printf("\n");
        in = 1;
        while(in<=i)
        {
            if(i%==0)
                Pf("*");
            else
                Pf("%d", in);
            in = in+1;
        }
    i = i+1;
    }
    getch();
}
```

output

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

Ex 20

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

(Pr 19)

```
void main()
{
  int i, n, in, dn, s;
  clrscr();
  Pf("Enter a value:");
  Sf("%d", &n);
  i=1;
  while(i<=n)
  {
    Pf("\n");
    s=1;
    while(s<=(n-i)*2)
    {
    in=1;
    while(in<=i)
    {
      Pf("%d", in);
      in=in+1;
    }
    s=1;
    while(s<=(n-i)*2)
    {
      Pf("*");
      s=s+1;
    }
    dn=i;
    while(dn>=1)
    {
      Pf("%d", dn);
      dn=dn-1;
    }
    i=i+1;
  }
  getch();
}
```

Output

```
Enter a value: 6
1**********1
12********21
123******321
1234****4321
12345**54321
123456654321
```

Increment & Decrement operators :-

- On implementation when we need to modify the initial value of a variable by '1', then go for increment or decrement operators that is $\underline{+1}$ or $\underline{-1}$.
- when we are working with increment or decrement operator, then difference between existing value & new value is $\underline{+1}$ or $\underline{-1}$.
- Depending on the position, these operators are classified into two types, i.e.

  (i) pre operator (ii) post operators
- If the symbol is available before the operand, then it is called pre operator, & if the symbol is available after the operand, then it is called post-operator.
- when we are working with pre operator before evaluating the expression, value need to be changed, when we are working with post-operator after evaluating the expression value needs to be changed.

- Eg :-

Syntax:
```
int a, b;
a = 1;
b = ++a ; // pre-increment
```

→ first increment the value of 'a' by '1', then evaluate the expression that is b = a;

here output :- a = 2, b = 2

Syntax - 2 → `b = a++ ; // Post increment`

First evaluate the exp^n, then increment the value of 'a' by '1'.

O/P → a = 2, b = 1.

Syntax - 3 → `b = --a ; // predecrement`

- First decrement, then assignment
- O/P → a = 0, b = 0

Syn-4 → `b = a-- ; // Post decrement`

- First evaluate the exp^n & then decrement 'a'.
- O/P → a = 0; b = 1.

Priority :-

① ( )

② +, −, ! ; ++, -- (Pre) ← sign operator (unary)

③ *, /, %

④ +, − (binary)

⑤ ==, !=

⑥ &&

⑦ ||

⑧ ? :

⑨ =

⑩ ++ -- (Post operators)

(ex) void main()
{
int a = 10;
++a;
Printf("a = %d", a);
}
o/p ⇒ a = 11

(ex) void main()
{
int a;
a = 10;
a++;
Printf("a = %d", a);
}
o/p: a = 10

note: Until we are not assigning the data to any other variable, there is no difference between pre & post operator.

(ex) void main()
{
int a;
a = 1;
a = ++a + ++a + ++a;
Pf("a = %d", a);
}
o/p: a = 12

a: 1 → 2 → 3 → 4

a = ++a + ++a + ++a
a = a + a + a
= 4 + 4 + 4 = 12

(ex) void main()
{
int a;
a = 1;
a = ++a + a++ + ++a;
Printf("a = %d", a);
}

a: 1 → 2 → 3 → 9 → 10

a = ++a + a++ + ++a
a = a + a + a
= 3 + 3 + 3
= 9

a = 10 ⇒ o/p

(ex) void main()
{
int a;
a = 1;
a = a++ + ++a + a++;
Pf("a = %d", a)
}
o/p a = 8

a: 1 → 2 → 6 → 7 → 8

a = a++ + ++a + a++
= a + a + a
= 2 + 2 + 2
= 6

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

a	b
90	50
101	51
102	154
103	155

o/p → a = 103, b = 155

a = a++ + ++b
= 50 + 51 = 101 102 103

b = ++a + b++
= 103 + 52 = 154 155

22/8/12

(ex) void main()
{
int a, b;
a = b = 5;
a = a-- + --b;
b = --a + b--;
printf("a = %d b = %d", a, b);
getch();
}

a	b
5	5
4	4
3	3

a = a-- + --b
= a + b = 5 + 4 = 9

b = --a + b--;
= a + b = 7 + 4 = 11

o/p :- a = 7, b = 10

(ex) void main()
{
int a, b, c;
a = 1; b = 2; c = 3;
a = a++ + ++b + c--;
b = ++a - b++ + c++;
c = a-- + --b - c++;
printf("a = %d b = %d c = %d", a, b, c);
}

a	b	c
1	2	3
2	2	3
2	2	2

a = a++ + ++b + c--
= a + b + c = 1 + 2 + 3 = 6

b = ++a - b++ + c++
= 3 - 2 + 3 = 4

c = a-- + --b - c++
= 3 + 2 - 3 = 2

a = 2, b = 4, c = 2

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

a	b	c
2	4	6
1	3	5
10	11	19

a = (---a) + (--b) + c--

= a + b + c = 1 + 3 + 6 = (10)

b = a-- - b++ + (--c);

= 10 - 3 + 4 = (11)

c = (++a) + b-- - c++;

= 10 + 12 - 4 = 18

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

o/p: 8 6 6

a: 7 → 8

Printf("%d %d %d", ++a, a++, ++a); (R → L)

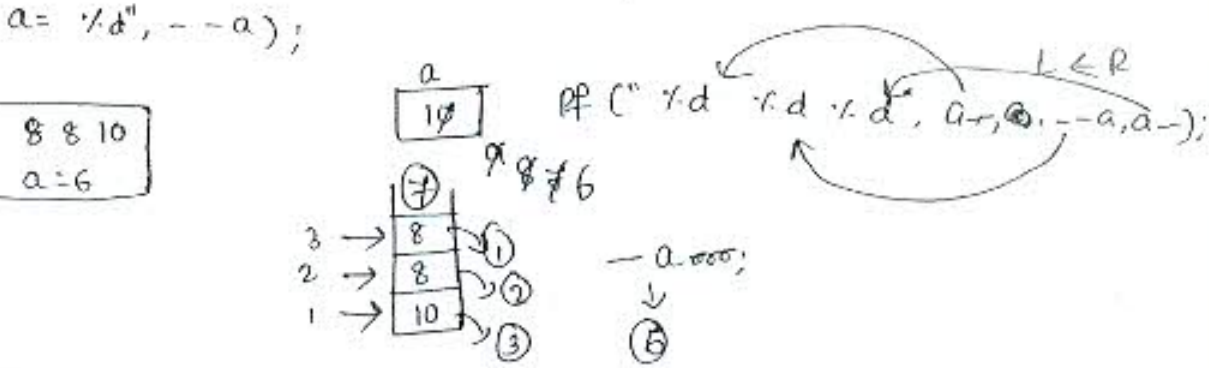
3 →	8 → (1)
2 →	6 → (2)
1 →	6 → (3)

o/p :- 8 6 6

- When we are working with "Printf()", then it works with the help of stack, i.e. LIFO concept.
- In printf(), always data will be pass towards right to left and data will be printed towards left to right

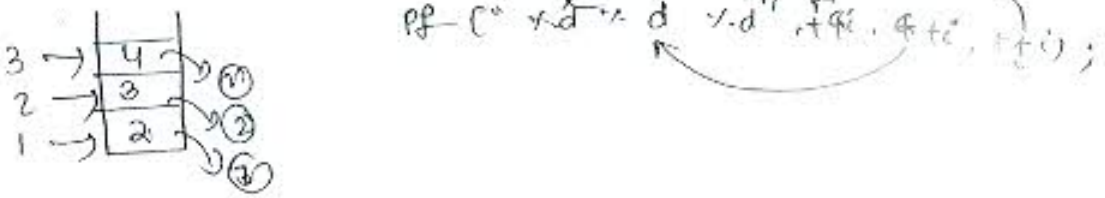
(Que) void main()
{
int a;
a = 10;
pf("%d %d %d", a--, --a, a--);
pf("\n a = %d", --a);
}

o/p:- 8 8 10
a = 6



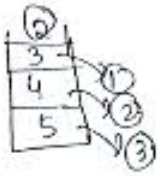
(Ex) void main()
{
int i;
i = 1;
pf("%d %d %d", ++i, ++i, ++i);
}

o/p:- 4 3 2



(Ex) void main()
{
int a = 5;
pf("%d %d %d", a--, a--, a--);
}

o/p:- 3 4 5



(Ex) void main()
{
int a = 100;
pf("%d %d %d", ++a, ++a);
}

o/p:- 102 101 Gr/Jeek

(ex) void main()
{
int a, b;
a=1;
b = ++a * ++a * ++a;
pf("a = %d, b = %d", a, b);
}

a = 4, b = 64

b = ++a * ++a * ++a
b = a * a * a
= 4 * 4 * 4 = (64)

a
4
2
3
(4)

(ex) void main()
{
int a, b;
a=1;
b = ++a * a++ * ++a;
pf("a = %d, b = %d", a, b);
}

o/p :- a = 4, b = 27

b = (++a) * a++ * (++a);
= 3 * 3 * 3 = (27)

a
4
2
3
(4)

(ex) void main()
{
int a, b;
a=1;
b = a++ * ++a * a++;
pf("a = %d b = %d", a, b);
}

a
4
2
3
(4)

b = a++ * ++a * a++
= 2 * 2 * 2 = (8)

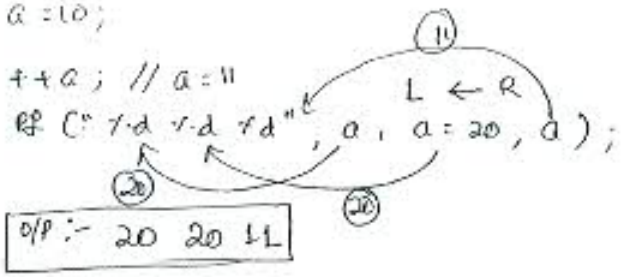
(ex) void main()
{
int a, b;
a=1;
b = a++ * a++ * a++;
pf("a = %d b = %d", a, b);
}

op :- a = 4 b = 1

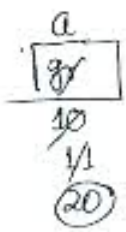
b = a++ * a++ * a++
= a * a * a = (1)

a
4
2
3
(4)

(Ex2)
```
void main()
{
    int a;
    a = 10;
    ++a; // a = 11
    pf("%d %d %d", a, a = 20, a);
}
```



o/p :- 20 20 11



- When we are working with 'Printf()', always arguments will be passed towards from right to left, because it works with the help of stack.

(Ex3)
```
void main()
{
    int a;
    a = 10;
    pf("%d %d %d", a = 24, a = 26, a = 42);
}
```

L ← R

o/p :- 24 26 42

- On printf(), data will be passed towards from right to left, and data will be printed towards from left to right.

(Ex4)
```
void main()
{
    pf("A");
    if(5<2)
    {
        pf("Hello");
        pf("welcome");
    }
    pf("B");
}
```

o/p :- AB

Ex5)
```
void main ()
{
    Printf (" Hello");
    if ( 5<2)
        Pf (" A");
        Pf (" B");
    Pf (" Welcome");
}
```

o/p :- Hello B Welcome

Ex6)
```
void main ()
{
    Printf (" A");
    if (5<2) ;
    {
        printf (" Hello");
        printf (" Welcome");
    }
    printf (" B");
}
```

o/p :- A Hello Welcome B

dummy condition

```
if (5<2)
{
}
{
    Pf (" Hello");
    Pf (" Welcome");
}
```

- When we are placing the semicolon (;) at end of the 'if', then it becomes dummy condition.
- When the dummy condition is created, then compiler will create a new body without any statements and current body becomes outside the condition.
- When we are working with dummy condition then if condition is true or false, always current body will be executed.

Ex7)
```
void main ()
{
    int i;
    i = 1;
    while (i <= 10)
    {
        printf ("%d", i);
        ++i;
    }
}
```

o/p :- 12345678910

ex 8)
```
void main ()
{
    int i;
    i = 1;
    while (i<=10)
    printf ("%d ", i);
    ++i;
}
```

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

op :- 1 1 1 1 . . . - infinite loop
(Ctrl + break)

- If the body is not specified for while, then first statement only will be placed inside the loop and untill the condition becomes false, it will repeats, if condition is not false, then we'll get infinite loop.

ex 9)
```
void main ()
{
    int i;
    i = 1;
    while (i<=10);
    {
        printf ("%d ", i);
        ++i;
    }
}
```

dummy loop

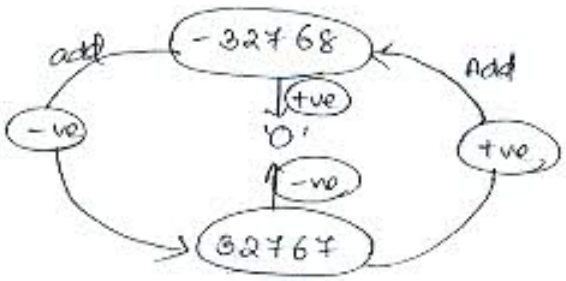
```
i = 1;
while (i<=10)
{
}
{
    printf ("%d", i);
    ++i;
}
```

op :- No output with infinite loop

- When we are placing the semicolon (;) at end of the "while", then it becomes dummy loop.
- When the dummy loop is constructed, then compiler will create a null body without any statements and current body becomes outside of the condition loop. So empty body will be executed untill the condition becomes false, if condition is not false, then we will get infinite loop.

Integer :-

- The size of the 'integer' on dos based compiler is 2 bytes and the range is from "-32768 to +32767"
- When the values exit the maximum range, then automatically controll will pass to opposite direction.



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)

(ex)

```
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

L → R

a = 200 * 200/200

= (40000)

↓

32767 + 7233 → (a)

= -32768 + 7232 → (a-1)

= -25536

∴ a = -25536/200

= $\frac{-25536}{2} * 10^2$

= -12768 * 10^{-2}

= -127.68

= (-127)

→

b = 200/200 * 200

(ex 2) void main()

```
{ int a, b;
  a = 300 * 200 / 300 ;
  b = 300 / 200 * 300 ;
  printf ("a = %d b = %d ", a, b);
}
```

o/p :- a = -18 b = 300

$a = 300 * 200 / 300$;

$= 160,000$)

60000
32767
27233

$-32768 + 27232$

$= -5536$

$a = -5536 / 300$

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

$= -18.453 * 10^2$

$= -18.453$

$= -18$

$b = 300 / 200 * 300$;

$= 1 * 300$

$= 300$

25/09/12

(ex 3) 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

(ex 4) void main()

```
{
  int i;
  i = -32768 ;
  if (i-- > -32768) // -32768 > -32768
  {
     printf (" welcome %d", i);
  else
    printf (" Hello %d", i);
}
```

o/p :- Hello 32767

Ex5)
```
void main()
{
    int i;
    i = 32767;
    if (++i < 32767)  // if (-32768 < 32767)
        Printf (" welcome %d", i);
    else
        printf (" Hello %d", i);
}
```

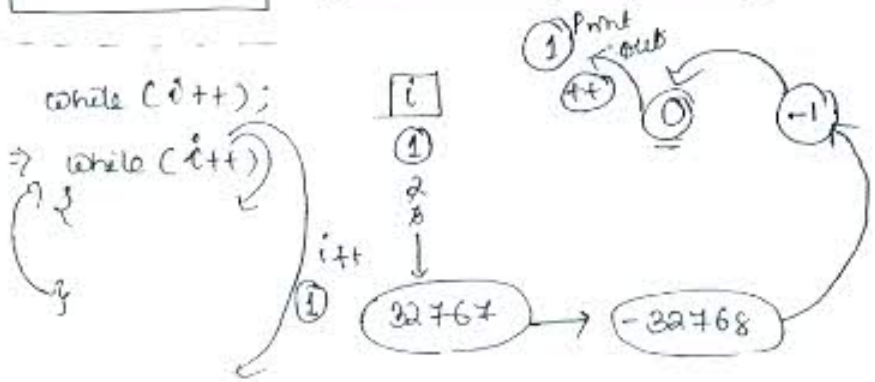
o/p :- welcome -32768

Ex6)
```
void main()
{
    int i;
    i = -32768;
    if (--i > -32768)  // if (-32767 > -32768)
        Printf (" welcome %d", i);
    else
        Printf (" Hello %d", i);
}
```

welcome 32767

Ex7)
```
void main()
{
    int i;
    i = 1;
    while (i++);
      Printf (" i = %d", i);
}
```

o/p :- i = 1 (65536 → no of repetation)



(ex8) void main()
{
int i;
i = 1;
while (i++ < 32767);
Printf("i = %d", i);
}

o/p :- i = -32768

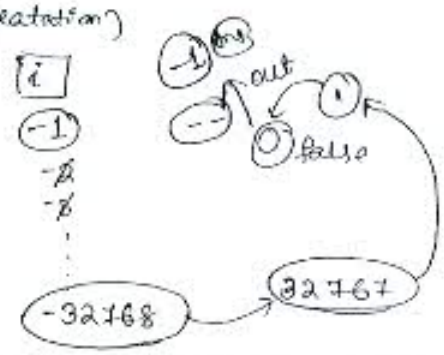
while (i++ < 32767);
⇒ while (i++ < 32767)
{
}

i = (1)
2
3
:
32767
out (i++)
-32768 (o/p)

(ex9) void main()
{
int i;
i = -1;
while (i--);
Printf("i = %d", i);
}

o/p :- i = -1 (65535 repeatations)

while (i--);
⇒ while (i--)
{
}



(ex10) void main()
{
int i;
i = -1;
while (i-- > -32768);
~~Printf("\n Welcome %d")~~
Printf("i = %d", i);
}

o/p :- i = 32767

Ex11)

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

o/p :-

```
1 2
1 3
1 4
0 5
a = 10  b = 15
```

a	a = b++ <= 3	
1		
2	① < 3 → ①	1 2
1	→ ② < 3 → ①	1 3
	→ ③ < 3 → ⓪	1 4
	④	0 5

a	b	while(a)
→ 1	1 <= 3	{
→ 1	2 <= 3	a = b++ <= 3
→ 1	3 <= 3	Pf("\n %d %d", a, b);
→ 0	4 <= 3	} → false
⓪	⑤	

Ex12)

```
void main()
{
    int a, b;
    a = b = 5;
    while(a)
    {
        a = ++b <= 8;
        Pf("\n %d %d", a, b);
    }
    Pf("\n a = %d  b = %d", a+10, b+10);
}
```

a	b	
5	5	
1	6 <= 8	6
1	7 <= 8	7
1	8 <= 8	8
⓪	⑨ <= 8	9
out		

output :-

```
1 6
1 7
1 8
0 9
a = 10, b = 19
```

Ex13
```
void main()
{
    int a;
    a = printf("welcome");
    printf("\n a = %d", a);
}
```

output
welcome
a = 7;

- By using "printf()" function, we can print the data on console.
- whenever we are working with "printf()", it returns an integer value, i.e. total no of characters, which is printed on console.

Ex14
```
void main()
{
    int a;
    a = printf("Naresh IT\n");
    printf("a = %d", a);
}
```

o/p
Naresh IT
a = 10

a = 6 (Naresh) + 1 (space) + 2 (IT) + 1 (\n) = 10

Ex15
```
void main()
{
    int a;
    a = printf("%d %d %d", 10, 20, 30);
    printf("\n a = %d", a);
}
```

o/p
10 20 30
a = 8

a = 2(10) + 1(sp) + 2(20) + 1(sp) + 2(30)
= 8

ex16)
```
void main()
{
    int a;
    a = printf("welcome %d", printf("Naresh IT\n"));
    printf("\n a = %d", a);
}
```

L ← R

a = printf("welcome %d", printf("Naresh IT\n"));
(1) ← (10) ← (10)

o/p:
```
Naresh IT
welcome 10
a = 10
```

- when we are placing the 'printf()' within the 'printf()', then towards from right side, first "printf()" always executes first; because to "printf()", parameter will be passed towards from right to left.

ex17)
```
void main()
{
    int a;
    a = printf("\n Three %d", printf("\n Two %d", printf("\n One")));
    printf("\n a = %d", a);
}
```

o/p:
```
One
Two 4
Three 5
a = 7
```

L ← R

a = Printf("\n Three %d", Printf("\n Two %d", Printf("\n One")));
(7) ← (5) ← (3)

ex18)
```
void main()
{
    int a;
    a = printf("One") + printf("\n Two") + printf("\n Three");
    printf("a = %d", a);
}
```

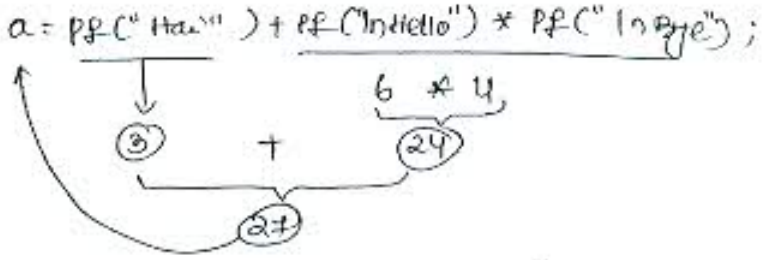
o/p:
```
One
Two
Three a = 13
```

L → R (operators having equal priority)

a = Pf("One") + Pf("\n Two") + Pf("\n Three");
3 + 4 + 6
= (13)

Ex19 void main()
{
int a;
a = printf("Hai") + Pf("In Hello") * printf("In Bye");
printf("\na = %d", a);
}

o/p
Hello
Bye Hai
a = 27



26/09/12

Ex20 void main()
{
int a, b, c;
c = scanf("%d %d", &a, &b);
Printf("In a = %d b = %d c = %d", a, b, c);
}
// input values are 100 200.

o/p :- a = 100 b = 200 c = 2

- when we are working with "scanf()" function, it returns an integer value, i.e. total no. of input values provided by user.

Ex21 void main()
{
int a, b, c;
a = b = 50;
c = scanf("%d %d", &a, b);
printf("In a = %d b = %d c = %d", a, b, c);
}
// input values are 500 600

o/p :- a = 500 b = 50 c = 2

- The complete behaviour of "scanf()" depends on format specifiers only, not on argument list.
- As a programmer, it's our responsibility to store the data properly by using '&' symbol.
- when '&' symbol is not provided by programmer, then existing value will not be updated with newly entered value.

(22)
```
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);
} // input values are 100 200 300
```

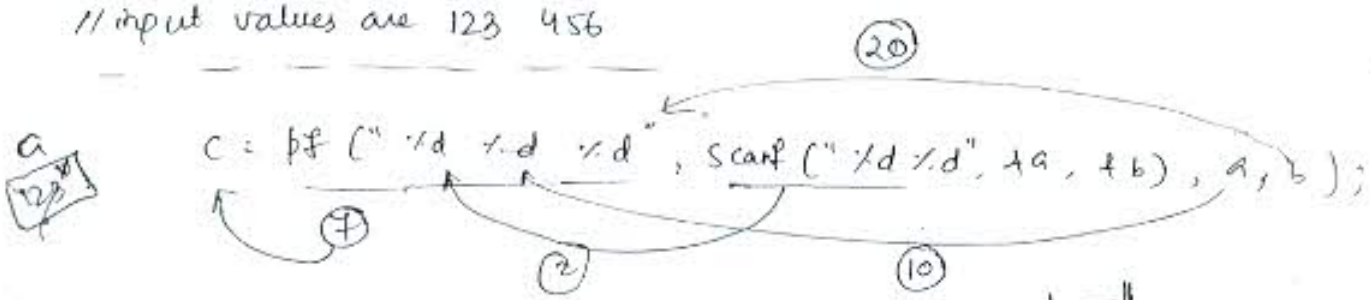
o/p :- a=10 b=20 c=3

(23)
```
void main()
{
    int a,b,c;
    c=printf("welcome%d", scanf("%d%d", &a, &b));
    printf("\n a=%d b=%d c=%d", a,b,c);
}
// input data are 111 222
```

o/p :-

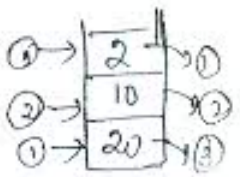
welcome2
a=111 b=222 c=8

(24)
```
void main()
{
    int a,b,c;
    a=10; b=20;
    c=printf("%d %d %d", scanf("%d%d", &a, &b), a, b);
    printf("\n a=%d b=%d c=%d", a,b,c);
}
// input values are 123 456
```



o/p

```
2  10  20
a=123  b=456  c=7
```



- Always 'printf()' function will execute towards from right to left, because it works with the help of stack.

- After passing the data into 'printf()' related stack, then data will not be updated by using "scanf()" function, because 'printf()' related stack is temporary memory, which will distroy automatically after execution.

Ex:
```
void main()
{
    int a, b, c;
    a=10; b=20;
    c = printf("%d %d %d", a, b, scanf("%d %d", &a, &b));
    Printf("\n a= %d  b= %d  c= %d", a, b, c);
}
// input values are 100 200
```

o/p:-

100 200 2
a=100 b=200 c=9

'break' and 'continue' :-

- 'break' is a keyword, by using break we can terminate loop body or switch body.
- Using 'break' is always optional, but it should be placed within the loop body or switch body only.
- In implementation, where we know the maximum no. of repeatations, but some condition is there, where we require to terminate the repeatation process, then go for 'break'.

continue

↳ 'continue' is a keyword, by using 'continue' we can skip the statements from loop body.

↳ Using 'continue' is always optional, but it should be placed within the loop body only.

↳ In implementation, where we know the maximum no. of repeatations, but some condition is there, where we require to ~~stop~~ skip the repeatation process, then go for 'continue'.

Ex1
```
void main()
{
    int i;
    i = 1;
    while (i<=10)
    {
        printf ("%d", i);
        if (i>3)
            break;
        ++i;
    }
}
```

o/p :- 1 2 3 4

- In above program, when the value of 'i' became '4', then condition became true, so break statement is executed.
- Within the loop body, when the 'break' statement is executed, then control will pass outside of the loop body.

Ex2
```
void main()
{
    int i;
    i = 20;
    while (i>=2)
    {
        printf ("%d", i);
        i -= 2; // i = i-2
        if (i<=14)
            break;
    }
}
```

o/p
20 ~~18~~ ~~16~~ ~~14~~

o/p
20 18 16

note

+ → +=	combination operator ↓ having the least priority than assignment operator.
- → -=	
* → *=	
/ → /=	
% → %=	

ex:- i += 2
↳ i = i + 2

i *= 5;
↳ i = i * 5;

(ex3)

```
void main()
{
    int a;
    a = 1;
    while (a <= 30)
    {
        a += 2; // a = a+2
        if (a >= 15)
            break;
        Printf ("%d ", a);
    }
}
```

o/p: 3 5 7 9 11 13 15

(ex4)

```
void main()
{
    int a;
    a = 2;
    while (a <= 20)
    {
        printf ("%d ", a);
        if (a) // if (2) → true
            break;
        a += 2;
    }
}
```

o/p : 2

(ex5)

```
void main()
{
    int i;
    i = 5;
    while (i <= 25)
    {
        Printf ("%d ", i);
        if (!i)
            break;
        i += 5;
    }
}
```

o/p :- 5 10 15 20 25

(26)
```
void main()
{
    int i;
    i=1;
    while(i<=10)
    {
        printf("%d", i);
        if(i>=5); //
            break;
        ++i;
    }
```

```
if (i>=5)
{ }
```

O/P: 1

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

out

- When we are placing the semicolon (;) after 'if', then it becomes dummy condition.
- When the dummy condition is created, then compiler will create a new body without any statements and current body becomes outside the condition.
- In above program, in order to execute the loop body, first time automatically 'break' statement will be executed, because it is outside the condition.

(27)
```
void main()
{
    int i;
    i=1;
    while(i<=20);
    {
        printf("%d", i);
        if (i<=5)
            break;
        ++i;
    }
}
```

O/P: error → misplaced break.

```
while (i<=20)
{ }
{
    Pf("%d", i);
    if (i<=5)
        Break;
}
```

- when we are placing the semicolon at end of the 'while', then it becomes dummy loop,
- when the dummy loop is constructed, then compiler creates a new body without any statements, then automatically control'll pass outside of the loop body.
- In the above program, due to dummy loop, 'break' statement is placed outside the loop body, so it is not allowed.

27/10

(ex8)

```
void main()
{
    int i;
    i=2;
    while(i<=100)
    {
        i+=2; // i=i+2
        if(i>=40 && i<=80)
            continue;
        Printf("%d",i);
    }
}
```

o/p :- 4 6 8 10 ---- 38 82 84 86 ---- 98 102 104

- when we are working with continue statement, then it passes the control back to the condition without executing rest of statements within the body.

(ex9)

```
void main()
{
    int i;
    i=1;
    while(i<=80)
    {
        i+=2;
        if(i>15 && i<65)
            continue;
        Printf("%d",i);
    }
}
```

output :-

3 5 7 9 ---- 13 15 65 67 ---- 75 77 79 ...

ex 10

```
void main()
{
    int a;
    a=0;
    while (a<=50)
    {
        printf("%d", a);
        if (a>=20 && a<=40)
            continue;
        a+=5;
    }
}
```

o/p:-

0 5 10 15 20 20 20 - - - - → ctrl + break

↳ infinite loop

ex 11

```
void main()
{
    int i;
    i=2;
    while (i<=60)
    {
        if (i>=10 && i<=30)
            continue;
        printf("%d", i);
        i+=2; // i=i+2;
    }
}
```

o/p:-

2 4 6 8 .. no output with inf. loop

ex 12

```
void main()
{
    int i=1;
    while (i<=30);
    {
        i+=2;
        if (i>=15 && i<25)
            continue;
        printf("%d", i);
    }
}
```

→

```
i=1;
while (i<=30)
{
}
{ i+=2;
    if (i>=15 && i<25)
        continue;
    pf("%d", i);
}
```

o/p:-

error : misplaced continue

Diff. Types of Loops in C' :-

- In C' prog. language, there are three types of loops are available.
  i.e. (i) while
  (ii) for
  (iii) do while.

① while loop :-

↳ When we are working with 'while' loop, always prochecking process is occurred, i.e. before execution of the statement body condition part of 'while' will be evaluated.

↳ Always 'while' loop repeats in clock-direction.

↳ Syntax :-

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

(ex1)

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

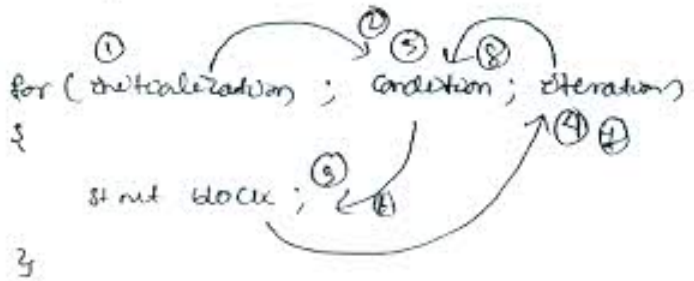
o/p :- 1 2 3 4 5 ... 910

② For loop :-

- When we are working with 'for' loop, it contains 3 blocks ; i.e.
  1. Initialization.
  2. condition
  3. iteration.

- Syntax :-

```
for ( initialization; condition ; iteration )
{
    statement block ;
}
```



- When we are working with for loop, always execution process will start with initialization block. Initialization block will be executed only once, when we are passing the control either the loop body first time.
- After execution of initialization block, control will pass to condition block, if condition is evaluated as 'true', then control will pass to statement block.
- After execution of the statement block, control will pass to iteration, from iteration block back to the condition.
- Always repeatations will happen between condition, statement block, & iterations only.
- When we are working with 'for loop' every thing is optional. but mandatory to place two semicolons (;).

```
while ( ) → error
for ( ; ; ) → yes valid
```

- In 'for' loop when the condition part is not available it repeats infinite times, because condition part is replaced with non-zero.
- When we are working with 'for' loop, it repeats in anti-clock direction.

1. while (0) → no repeatation
2. for (; 0 ;) → repeat once.

- when we are placing constant '0' as a condition in for loop, then it will repeat once

```
int i;
i=0;
while (i) → no repeatation
for (; i ; ) → no repeatation.
```

(ex)
```
void main()
{
    int i;
    for (i=1; i<=10; i++)
        Printf ("%d ", i);
}
```
(op):- 1 2 3 ---- 10

(3) do-while loop :-

- In implementation when we need to repeat the statement block atleast once, then go for 'do-while'.
- In 'do-while' post-checking process occurs; i.e. after execution of the statement block, condition is evaluated.
- Syntax:-

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

- When we are working with 'do-while' loop, semicolon must be required at end of the 'while'.

(ex)
```
void main()
{
    int i=1;
    do
    {
        printf ("%d", i);
        ++i;
    } while (i<=10);
}
```
op:- 1 2 3 4 -- 10

(02)
```
void main()
{
    int a;
    a=1;
    for(; a<=10; )
    {
      printf("%d ", a);
      a+=2; //a=a+2;
    }
}
```

o/p :- 1 3 5 7 9

- In implementation, when we having more than one initialization & iteration blocks, then go for comma (,) operator.

e.g.

(Ex)
```
void main()
{
    int a, b;
    for(a=10; b=1; a>b; a--, b++)
    {
        Printf("\n %d %d", a, b);
    Printf("\n a=%d b=%d", a+10, b+10);
}
```

o/p

10	1
9	2
8	3
7	4
6	5
5	6
4	7

a=15 b=16

a	b
10	1
9	2
8	3
7	4
6	5
(5)	(6) → out

a+10 = 15, b+10 = 16

(ex4)

```
void main()
{
  int a, b ;
for (a=b=10;  a  ; )
{
     a= b-- >=8 ;
     Printf ("\n %d  %d", a, b) ;
}
 Printf (" \n a= %d  b= %d", a+10 , b+10) ;
}
```

o/p :-

1	9
1	8
1	7
0	6

a=10 b=16

a	b
10	10
9	9
8	8
7	7
0	6

a = b-- >=8 10 >= 8 ⇒ 1

9 >= 8 ⇒ (1)

8 >= 8 ⇒ (1)

7 >= 8 ⇒ (0)

a = a+10 = 10 , b = b+10 = 16

(ex5)

```
void main()
{
   int a, b ;
   for ( a=b=8 ;  a ; Printf ("\n%d  %d", a, b))
            a= --b >=4 ;
   Printf (" \n a= %d  b= %d", a+10, b+10 ) ;
}
```

o/p :-

1	7
1	6
1	5
1	4
1	3

a=10 b=13

a	b
8	8
7	7
6	6
5	5
4	4
3	3

a+10 = 10 b+10 = 13

7 >= 4 ⇒ (1)

6 >= 4 ⇒ (1)

5 >= 4 ⇒ (1)

4 >= 4 ⇒ (1)

3 >= 4 ⇒ (0)

28/9/12

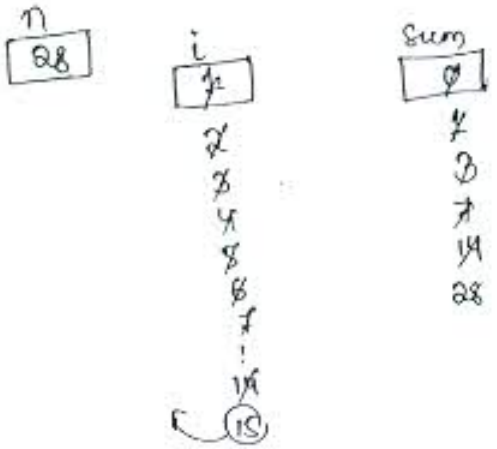
Ex1) Enter a value : 28
28 is PERFECT NUMBER.

Perfect number

When sum of all factor = Number

28 → 1 + 2 + 4 + 7 + 14 = 28

```
void main()
{
    int i, n, sum=0;
    clrscr();
    printf("\n 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 a PERFECT NUMBER", n);
    else
      Printf("\n%d is not a PERFECT number", n);
     getch();
}
```



(K2) Armstrong Number :- $153 \Rightarrow 1^3 + 5^3 + 3^3$

$= 1 + 125 + 27 = 153$

$370 \Rightarrow 3^3 + 7^3 + 0^3 = 27 + 343 = 370$

$371 \Rightarrow 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$

$407 \Rightarrow 4^3 + 0^3 + 7^3 = 64 + 0 + 343 = 407$

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

```
for (temp=n ; temp!=0;)
{
    i = temp % 10;
    sum += (i*i*i);
    temp /= 10;
}
```

n	temp	i	sum
153	153	3	0
	15	5	0+27 = 27
	1	1	27+125 = 152
	0		152+1 = 153

(K3) Prime Number

When a number only contains factor 1 & itself.

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

```
for(t=2; t<=n/2; t++)
{
    if(n%t==0)
    {
        flag=1;
        break;
    }
}
if(flag==0 && n>1)
    printf("\n %d is Prime number", n);
else
    printf("\n %d is not prime number", n);
getch();
}
```

```
for(t=2; t<=n/2; t++)
{
    if(n%t==0)
    {
        flag=1;
        break;
    }
}
```

n: 12 | flag: 0 | t: 2, 3, 4, 5, 6, 7 →

n: 9 | flag: 0 | t: 2, 3 → out | 1

(ex4) Enter two numbers : 2 20

1. Prime = 2
2. Prime = 3
3. Prime = 5
4. Prime = 7
5. Prime = 11
6. Prime = 13
7. Prime = 17
8. Prime = 19

~~or Prime~~

```
void main()
{
    long int n, n1, n2, t, count=0;
    int flag;
    clrscr();
    printf("\n Enter two values :");
    scanf("%ld %ld", &n1, &n2);
    for(n=n1; n<=n2; n++)
    {
        flag=0;
        for(t=2; t<=n/2; t++)
        {
            if(n%t==0)
            {
                flag=1;
                break;
            }
        }
```

```
if (flag == 0 && n > 1)
    Printf ("\n %3d . Prime: %5d", ++count, n);
}
getch();
}
```

n1	n2	n	flag	t
2	20	7	0	7
		8 9 10 11		3
		7 8 9 10		

98 (n-1) t<n

49 (n/2) t<=n/2

9 (√n) t<=sqrt(n)

(ex)
```
void main()
{
    int n, n1, temp, sum, i, count = 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);
        }
        if (sum == n && n > 1)
            printf ("\n %d. Armstrong = %d", ++count, n);
    }
    getch();
}
```

(o/p) Enter a value : 1000

1. Armstrong = 153
2. Armstrong = 370
3. Armstrong = 371
4. Armstrong = 407

2/10/12

(Ex6) Pascal's triangle :

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

```
void main()
{
    int n=1, s, q=0, r, x;
    clrscr();
    printf(" Rows you want to input :");
    scanf(" %d", &r);
    printf(" In Pascal's Triangle : \n");
    while(q<r)
    {
        for (s=40-3*q; s>0; s--)
        {
            printf(" ");
            for (x=0; x<=q; ++x)
            {
                if ((x==0) || (q==0))
                    n=1;
                else
                    n= (n* (q-x+1))/x; printf(" %6d ", n);
                printf("\n");
            q++;
    }
    getch();
}
```

Q7) void main()

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

```
                1<1
while (a++ <= 1)
{       ↓2
   while (a++ < 2)   2<=2
   {
          ↓3
   }
}
```

output :- a=5

```
4<=1
while (a ++ <= 1)    1<=1
{  3<=2                ↓(2)
   while (a++ <= 2)   2<=2
   {
                       ↓
                      (3)
   }
  (4)
}
(5)
```

- In implementation, when two while statements are occurred; immediately without body, without statement and without semicolon (;), then first while is treated as outer loop. that's according to the nested loop, the program will execute.

8) void main()

```
{
    int a;
    a=1;
    while (a++ <=1);
    while (a++ <=2);
    Printf ("a=%d",a);
}
```

o/p :- a=4

```
2<=1          1<=1
while (a++ <=1)
{
                ↓(2)
}
3<=2
while (a++ <=2)
{
}
(4)
```

Q9) void main()

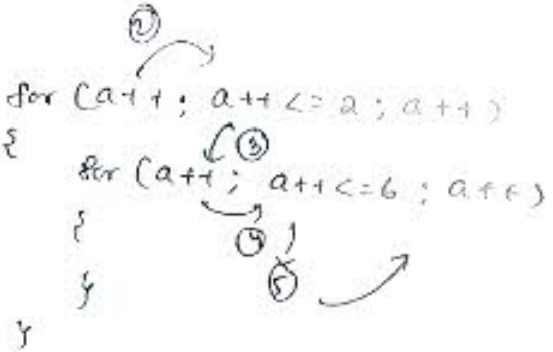
```
{
    int i;
    i=1;
    for (i++ ; i++ <=2; i++)
       i++ ;
    Printf (" i = %d", i);
}
```

o/p :- i=6

```
     1      2<=2   5<=2
for (i++ ; i++ <= 2; i++)
             i++ ;  (3)  (5)    (6)
             (4)
```

(Ex10)
```
void main()
{
    int a;
    a = 1;
    for(a++; a++<=2; a++)
    for(a++; a++<=6; a++);
        Printf("a = %d", a);
}
```

o/p :- a = 11



(Ex11)
```
void main()
{
    int a;
    a=1;
    for(a++; a++<=2; a++);
    for(a++; a++<=6; a++);
    Printf("a = %d", a);
}
```

o/p :- a = 9

(Ex12)
```
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

✓ In implementation, even we need to create dummy 'do while', then place the semicolon (;) after do, i.e. do ;

(Ex13)
```
void main()
{
    int a;
    a = 1;
    do;
    while(a++<=1);
    Printf("a = %d", a);
}
```

→

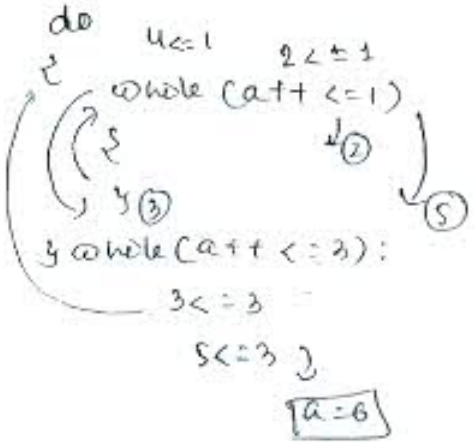
```
do
{
} while(a++<=1);
```

o/p : a = 3

(14)
```
void main()
{
  int a;
  a=1;
  do
   while(a++<=1);
   while(a++<=3);
   Printf("a=%d",a);
}
```

o/p:- a=6

```
do
{
   while(a++<=1)
   {
   }
}
while(a++<=3);
```



a
1

Switch :-

- Switch is a keyword, by using switch we can create a selection statement with multiple ~~case~~ choices.
- Multiple choices can be constructed by using 'case' keyword.
- When we are working with 'switch', it requires a condition or [illegible] of type integral (integer type).
- 'case' keyword requires constant expression or constant value of type integer only.
- syntax:

```
switch (condition / Expression)
{
   case const 1 : Block 1;
                  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 return value will be mapped with case constant value.
- At the time of execution, if the matching case is occurred, then from that matching case upto break, everything will be executed; if break is not there, then including default, all cases will be executed.
- At the time of executing, if matching case is not available, then automatically control will pass to default block.
- 'default' is a special kind of case which will be executed automatically when the matching case is not occurred.
- Using 'default' is always optional, it is recommended to use when we are not handling all cases of switch block.
- By using nested if-else, we can create multiple blocks also.
- In nested if-else, if we need to create 'n' no. of blocks, then (n-1) conditions are required, but in switch statement, multiple blocks are kept under single condition.
- In nested if-else, at given any point of time, only one block can be executed, but in switch, more than one block execution is possible to execute by removing 'break' statements between the blocks.

ex1
```
void main()
{
    int i;
    i=2;
    switch(i)
    {
        case 1: printf("A");
                break;
        case 2: printf("B");
                break;
        case 3: printf("C");
                break;
        default: printf("D");
    }
}
```
out:- B

ex2
```
void main()
{
    int i;
    i=1;
    switch(i);
    {
        case 1: printf("1");
        case 2: printf("2");
                break;
        case 3: printf("3");
                break;
        default: printf("D");
    }
}
```
o/p :- 12

xo) void main()
{
int i;
i = 5;
switch (i)
{
case 1 : printf ("A");
break;
case 2 : printf ("B");
break;
case 3: printf ("C");
break;
default : printf ("D");
}
}

o/p :- D

12
ery) void main()
{
int i;
i = 3;
switch (i)
{
case 1: printf ("A");
break;
case 3 : printf ("C");
case 2 : printf ("2");
break;
}
}

o/p :- C 2

- when we are working with switch statement, cases can be constructed in any sequence, that is randomly we can create cases.
- when we are creating the cases randomly, then from matching case upto break everything will be executed in any sequence.

ex) void main()
{
int i;
i = 5;
switch(i)
{
case 1 : pf ("1");
break;
default : pf ("D");
case 2: pf ("2");
break;
case 3 : printf ("C");

o/p :- D2

- Placing the default is always optional, i.e it can be placed at the top of the switch body or middle of the switch body or ending part of the switch body.
- But generally, it's recommended to place it at the end of the switch body.

ex6)
```
void main()
{
    int i;
    i = 3.8; // i = 3
    switch(i)
    {
        case 1: Pf("A");
                break;
        case 2: Printf("2");
                break;
        case 3: Printf("B");
                break;
        default: Printf("DY");
    }
}
```
o/p: C

ex7)
```
void main()
{
    float i;
    i = 5; // i = 5.0;
    switch(i)
    {
        case 1: Printf("1");
                break;
        case 2: Printf("2");
                break;
        case 3: Printf("3");
                break;
        default: Printf("0");
    }
}
```
o/p: Error
↳ switch selection expression must be of integral type

- When we are working with 'switch' statements, then it requires a condition or expression of type integral value only; i.e we can't pass float type data as a condition of 'switch' statement.

(5)
```
void main()
{
    int i;
    i=3;
    switch(i)
    {
        case 1.0 : printf(" H");
                   break;
        case 2.0 : printf("Y");
                   break;
        case 3.0 : printf("D");
                   break;
        default : printf(" NIT");
    }
}
```

o/p : error : constant expression required.

- 'case' keyword always requires a constant value or constant integral expression only.

eg)
```
void main()
{
    int a, b, c, d;
    a=1; b=2; c=3;
    d = c-a;
    switch(d)
    {
        case a : printf(" N");
                 break;
        case b : printf("I");
                 break;
        case c : printf("T");
                 break;
        default : printf("HYD");
    }
```

variable type not allowed.

o/p :- error : constant expression required.

- on 'switch' statement; 'case' can't be constructed by using variable type data because always it is required constant type value only.

(ex10)
```
void main()
{
    int i;
    i = 4/2;
    switch (i)
    {
        case (2*2) : printf (" A") ; // case 4 :
                     break;
        case 8/4 : printf(" B") ; // case 2 :
                   break ;
        case 2+1 : printf (" C") ; // case 3:
                   break;
        case 5-4 : printf (" D"); // case 1 :
                   break ;
        case 12%8 : printf (" E"); // case 0
    }
}
```

o/p :- B

- Inside the switch body, when we are writing constructing the expression, then it works according to return value of the expression.

(ex11)
```
void main ()
{
    int i;
    i = 5/2;
    switch(i)
    {
        case 2>5 : printf (" N"); // case 0 :
                   break;
        case 5.0/2 : printf (" I") ; // case (2.5) :
                     break;
        case 5<8 : printf (" T"); // case 0 :
                   break;
        case 1 != 2<5 : printf (" ##D"); // case 0:
    }
}
```

o/p :- error : Constant expression required
duplicate case

when we are working with 'switch' statement, duplicate cases are not allowed, i.e. more than one 'case' with same value are not possible.

goto :-

- 'goto' is a keyword, by using this keyword, we can pass the control anywhere in the program within the local scope.
- 'goto' keyword always prefers an identifier called 'label'.
- Any valid identifier followed by colon (:) is called label.
- 'goto' statement is called unstructured control flow statement, because it breaks the rule of structured programming language.

Syntax

```
statement 1;
statement 2;
goto LABEL;
    statement 3;
LABEL:
    Stmt 4;
    Stmt 5;
```

Ex1)

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

o/p :- ABCD

Ex2)

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

o/p :- NITABCXY

- In order to execute the program, if the label is occurred, it will executed automatically without calling also.
- Creating the label is always optional; while creating the label, calling the label is also optional but if we are calling the label, then it should be exist within the program.

- In implementation, when we require the repetation without using loops, then go for 'goto' statement.

ex3
```
void main()
{
    int i;
    i = 2;
    EVEN:
        Printf("%d", i);
        i = i+2;
        if (i<=20)
            goto EVEN;
}
```
O/P:- 2 4 6 8 10 ... 20

ex4
```
void main()
{
    Printf("A");
    Printf("B");
    goto ABC;
        printf("welcome");
XYZ:
    Pf("X");
    Pf("Y");
ABC:
    Printf("C");
    Printf("D");
    goto XYZ;
}
```
O/P:- A B C D X Y C D X Y C D X Y C D X Y C D ... ctrl+break

ex5
```
void main()
{
    Printf("welcome");
    Printf("NIT");
    goto ABC;
        Printf("HyD");
abc:
    Printf("A");
    Printf("B");
}
```
O/P:- error: undefined label 'ABC'

- labels are worked with the help of case sensitivity, i.e. uppercase & lowercase labels are different.

ex6
```
void main()
{
    int i;
    i=2;
    switch(1)
    {
    case 1: printf("A");
            break;
    case 2: printf("B");
            break;
    case 3: Printf("C");
            break;
    default: Printf("D");
}
```
O/P:- D

→ According to the syntax of switch case & constant value should be required space. If space is not provided, then it becomes label, that's why default will be executed.

ex7)
```
void main()
{
    int i;
    i= 2;
    switch(i)
    { case 1 : printf("1");
               break;
      case 2 : printf("2");
               break;
      case 3 : printf("3");
               break;
      case default : printf("D");
    }
}
```

→ label.

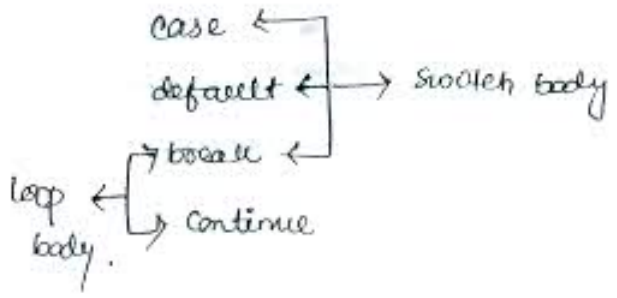
```
if casedefault : printf("D");
```

⇓

then output: 2

o/p:- error:

ex8)
```
void main()
{
    int i;
    i = 3;
    switch(i)
    {
     case 1 : Pf("A");
              break;
     case 2 : Pf("B");
              break;
     case 3 : Pf("C");
              continue;
     default : printf("D");
    }
}
```



o/p :- error: misplaced continue.

- When we are working with continue statement, it must be placed inside the loop body only.

ex9)
```
void main()
{
    int i;
    i=5;
    switch(i);
    {
        case 1 : printf("A1");
                 break;
        case 2 : printf("B2");
                 break;
```

dummy switch

98

```
        case 3 : printf ("C3");
                 break;
        default : printf("04");
    }
}
```

O/P:

- When we are placing semicolon after 'switch', then it becomes dummy switch body, when the dummy switch body is constructed, then compiler will create new body without any statement and current body becomes outside the switch, so automatically case, default & break will place outside.

4/10/12

(Ex10) WAP to show Day of a given date.

```
void main()
{
    int dd, mm, yy, nleap;
    long int dp;
    clrscr();
    printf("\n Enter a year : ");
    scanf("%d", &yy);
    if(yy <= 0)
    {
        printf("Invalid Year ....");
        goto END;
    }
    printf("\n Enter month : ");
    scanf("%d", &mm);
    if(mm < 1 || mm > 12)
    {
        printf("Invalid month ...");
        goto END;
    }
    printf("Enter date ..");
    scanf("%d", &dd);
    if(dd < 1 || dd > 31)
    {
        Printf("\n Invalid date ...");
        goto END;
    }
```

```
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 ("Invalid date ~ ");
    goto END;
  }
}
nleap = (yy-1)/4 - (yy-1)/100 + (yy-1)/400;
dp = (yy-1) * 365 L + nleap;
switch (mm)
{
    case 12 : dpt = 30;
    case 11 : dpt = 31;
    case 10 : dpt = 30;
    case 9 : dpt = 31;
    case 8 : dpt = 31;
    case 7 : dpt = 30;
    case 6 : dpt = 31;
    case 5 : dpt = 30;
    case 4 : dpt = 31;
    case 3 : dpt = 28;
    case 2 : dpt = 31;
    case 1 : dpt = dd;
}
if ((yy%4 == 0 && yy%100 != 0 || yy%400 == 0) && mm > 2)
    dp++;
printf ("%d %d %d %d week day 100", dd, mm, yy);
```

```
switch(dp%7)
{
    case 1 : printf("monday");
             break;
    case 2 : Pf("Tuesday");
             break;
    case 3 : Pf("wednesday");
             break;
    case 4 : Pf("Thursday");
             break;
    case 5 : Pf("Friday");
             break;
    case 6 : Pf("Saturday");
             break;
    case 0 : Pf("Sunday");
             break;
}
END:
getch();
}
```

O/P :-

```
Enter a year : 1947
Enter month : 8
Enter date : 15
15/8/1947 weekday : Friday
```

(Pkii) WAP to enter a year, if it is leap year, then give the day of 29 feb of the year.

```
void main()
{
    int yy, nleap;
    longint dp;
    clrscr();
    do
    {
        printf("\n Enter a year : ");
        scanf("%d", &yy);
    } while(yy<=0);
    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", yy);
```

```
switch (dp % 7)
{
    case 1 : printf ("Monday");
             break;
    case 2 : printf ("Tuesday");
             break;
      :
    case 6 : printf ("Saturday");
             break;
    case 0 : printf ("Sunday");
  }
}
else
    printf ("\n %d is not a leap year", yy);
getch();
}
```

op:-
```
Enter a year : 2012
29 - Feb - 2012 : Wednesday
```

1/10/12

n

7342
734
73
7
0

rn

012
20+4
240+3

```
while (n)
{
    rn = rn * 10 + n % 10;
    n = n/10;
}
```

Enter a value : 7342
SEVEN THREE FOUR TWO

Enter a value : 1000
one

Enter a value : 1500
ONE FIVE

```
void main()
{
  int n, rn=0, count=0;
  clrscr();
  printf ("Enter a value : ");
  scanf ("%d", &n);
  while (n)
  {
    rn = rn * 10 + n % 10;
    n = n/10;
    ++ count;
  }
```

```
    if (rn < 0)
        Printf (" -ve data");
    while (rn)
    {
      switch (rn % 10)
      {
          case 0: Printf (" ZERO");
                  break;
          case 1:
          case -1: Printf (" ONE");
                   break;
          case 2:
          case -2: Printf (" TWO");
                   break;
          case 9:
          case -9: Printf (" NINE");
                   break;
      }
      rn = rn/10;
      --count;
    }
    while (count > 0)
    {
        Printf (" ZERO");
        --count;
    }
    getch ()
}
```

Datatypes :-

- Datatypes will decide that what type of data can be holding by the variable.
- In C-prog. language, there are three types of basic datatypes are available i.e. char, int and float type.
- In implementation whenever the basic datatypes are not supporting user requirement, then go for primitive datatypes.
- Primitive data types are created by extending the range & size of basic datatypes
- In C-Prog. lang., there are 9 types of predefined or primitive datatypes are available.

Datatypes in C

Type	Size	Range	%	Ex
char	1 byte	-128 to 127	%c	'a', 'A', '#', '18'
unsigned char	1 byte	0 to 255		
int	2 bytes	-32768 to 32767	%d	-25, 5, 0, -5
unsigned int	2 bytes	0 to 65535	%u	25u, 32768u
) long int	4 bytes	-2,147,483,648 to 2147483647	%ld	45l, -5l, 4000l
) unsigned long	4 bytes	0 to 4294967295	%lu	100lu, 151lu.
) float	4 bytes	± 3.4 * 10^x = 38	%f	-3.5f, 7.5f
) double	8 bytes	± 1.7 * 10^x = 38	%lf	-3.5, 7.5
) longdouble	10 bytes	± 3.4 * 10^x = 9732	%Lf	-3.5L, 7.5L

The basic advantage of classifying these many types is nothing but utilizing the memory more efficiently & increasing the performance

On implementation, when we require character operations, then go for char or unsigned char.

Wherever we need an integer value from the range of -128 to +127, then go for char datatypes in place of creating an integer. In this case we required to use %d format specifier.

On implementation, when we require integer value from 0 to 255, then go for unsigned char type in place of creating unsigned int type. In this case, we require to use %u format specifier.

For normal integer operations, go for an int type, if there is no any negative representations like employee salary, then go for unsigned int type.

~~int~~ int a;

short int a;

signed short int a;

signed / unsigned — size: long — type: int — variable: a

- signed, unsigned, short & long are called qualifiers.

- signed, unsigned will indicate sign specification; short & long are indicating size specification. these four types of qualifiers should be applied for an integral type only, i.e. we can't apply for float, double, & long double type
- by default any integral type sign is signed type, size is short type

(or)

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

output

-32768 -32768
-32768.000000

Arithmetic operations on datatypes :-

① int, int, → int
② char, int → int
③ int, longint → longint
④ short, long → long
⑤ signed int, unsigned int → signed int
⑥ signed, unsigned → signed
⑦ int, float → float
⑧ long int, float → float
⑨ float, double → float
⑩ double, long double → long double

- if both data types are of same datatype, then return value will be same type
- if anyone of the argument is different, then among those all which will occupy maximum memory that will be returned.

Typecasting process :-

- It is a process of converting one datatype values into another datatype
- Type casting is under control of (type) operator.
- In C-prog. language, type casting are classified into two types :-
  ① Implicit type casting
  ② explicit " "

① Implicit type casting :-

- when we are converting lower datatype values into higher datatype values, then it is called implicit type casting
- Implicit type casting is under the control of compiler
- As a programmer, it is not required to handle implicit type conversion

② Explicit type casting :-

- When we are converting higher datatype values into lower datatype, then it is called explicit datatype.
- Explicit type casting process is under the control of programmer.
- As a programmer, when we explicit type casting is occurred, then it is mandatory to handle or else data overflow will occur.

syntax:-

```
datatype1 variable1 = value;
datatype2 variable2;
variable2 = (datatype2) variable1;
```

ex:1)
```
int i = 32452;
long int l;
l = (long int) i;
```

ex:2)
```
float f = 123.456;
int i;
i = (int) f;
```

ex:3)
```
void main()
{
    float f;
    f = 3.8;
    if(f==3.8) // 4B  8B
    if(float == double)
        printf("welcome");
    else
        printf("Hello");
}
```

o/p: Hello

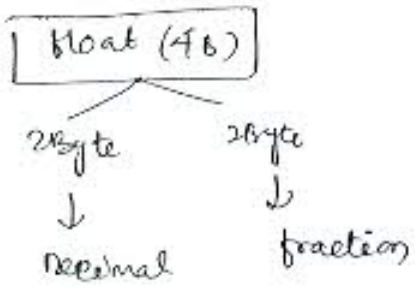
- By default, any type of real value is double type, that's why we are comparing the float data with the double condition, it becomes false.
- When we are applying the relational operators on float type data, then always comparison will be taken place in binary representation only.

(ex7)
```
void main()
{
    float f;
    f = 6.4;
    if (f == 6.4f)
      printf("welcome");
    else
      Printf("Hello");
}
```

(o/p) welcome

(ex8)
```
void main()
{
    int i;
    float f;
    i = 3;
    f = 3.0;
    if (i == f)
        printf("Welcome");
    else
      printf("Hello");
}
```

(o/p) welcome



- When we are working with float values, it occupies 4 bytes of physical memory, that's 2 bytes for decimal & rest of 2 bytes for the fractional part.
- In float value, when the fractional part doesn't contains any values, that's if it is zero, then it occupies 2 bytes only, that's why integer value is equals to float.

(16)
```
void main()
{
    float f;
    f = 8.0;
    if (f == 8.0)
        printf("welcome");
    else
        printf("Hello");
}
```

o/p :- welcome

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

.0
.5
} float = double

Number System :-

- Number system will decide that how the numeric values can be represented in physical memory.
- Number systems are classified into four types :-
  1. Decimal Number system
  2. Hexadecimal Number system.
  3. Octal Number system.
  4. Binary Number system.
- The base value of decimal number system is : 0 and range from 0 to 9
- when we are working with decimal number system, we required to use %d format specifier.
- by using decimal number system we can represent positive & negative data also.
- The base value of octal number system is '8' and the range is from 0-7.

- when we are working with octal number system, we required to use %o format specifier.
- If any numeric value is started with zero, it indicates octal number.
- In octal number system, there is no any negative representation available.
- The base value in hexadecimal number system is 16 and the range is from 0 to 9ABCDEF.
- when we are working with hexadecimal number system, %x, %X, %h & %p format specifiers are used.
- when we are using %X, then alphabets will be printed in upper case.
- If any numeric value is started with '0X', then it indicates a hexadecimal value.

Decimal (10)	Octal (8)	Hex (16)
0-9	0-7	0-F
%d	%o	%x %X %h %p
100	0144	0x64
127	0177	0x7F
255	0377	0xFF
32767	077777	0x7FFF
10	012	0xA
83	8^1 x 1 + 8^0 x 2 = 0123	0x53
error	0181	error
error	0912	error
17		0x11
161	0241	0xA1
538		0x21A

Binary :-

- The base value of binary number system is 2 and the range is 0, 1
- Binary representations can't be represented directly by using any format specifier.

Whenever we are representing the data in memory always it represents by using binary representation only.

- Whenever we are dealing with binary representation complete data will be represented in 1's and 0's only.

Decimal	Binary
1	01
2	10
3	11
4	0100
5	0101

33 → 0010 0001

62 → 0011 1110

127 → 0111 1111

255 → 1111 1111

32767 → 0111 1111 1111 1111

-32768 → 1000 000 000 000

bits

8 bits → 1 byte

1024 byte → 1 KB

1024 KB → 1 MB

1024 MB → 1 GB

1024 GB → 1 TB

No. of bits	No. of combinations	max value	max value
1	2 (2^1)	1 (2^1-1)	01
2	4 (2^2)	3 (2^2-1)	11
3	8 (2^3)	7 (2^3-1)	0111
4	16 (2^4)	15 (2^4-1)	1111
5	32 (2^5)	31 (2^5-1)	0001 1111
6	64 (2^6)	63 (2^6-1)	0011 1111
7	128 (2^7)	127 (2^7-1)	0111 1111
8	256 (2^8)	255 (2^8-1)	1111 1111
9	512		

8/10/12

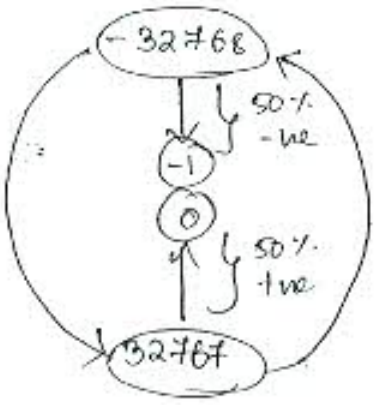
int i

→ On DOS based compiler, the size of integer is 2 bytes and the range from -32768 to 32767.

→ For representing any integer value, we require 16 bit combination i.e. 65536 combination (2^{16}).

→ Among these all representations 50% will represent +ve data and remaining 50% represents -ve data.

- Towards from leftside, first bit is called sign bit (Left side most significant bit is called sign bit).
- Always sign bit decides the return value of the binary representation i.e. positive or negative.
- If the sign bit is zero and remaining all bits are zero, then it gives minimum value of positive sign i.e. '0'.
- If the sign bit is zero & remaining all bits are '1's, then it gives maximum value of the positive sign, i.e. + 32767.
- If sign bit is '1' & remaining all bits are zeros, then it gives minimum value of -ve sign, i.e. -32768.
- If the sign bit is '1' & remaining all bits are 1's, then it gives maximum value of -ve sign i.e. -1.



sign bit

0000 0000 0000 0000 → 0

0111 1111 1111 1111 → 32767

1000 0000 0000 0000 → -32768

1111 1111 1111 1111 → -1

1111 1111 1111 1111

65535 - 32767 = 32768

-32768 + 32767 (n-1) = -1

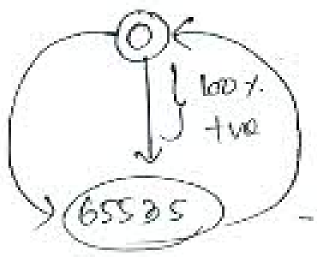
n

5) int i;
i = 32767 + 1;

32767 →	0111	1111	1111	1111	
+1 →	0000	0000	0000	0001	
	(1)000	0000	0000	0000	→ -32768

unsigned int u;

- The size of unsigned integer is 2 bytes & the range from 0 to 65535.
- For representing any unsigned integer value, we require 16 bit combination, i.e. 65536 representations.
- Among those all representations, all represent positive data only, because sign bit is not available to unsigned type.

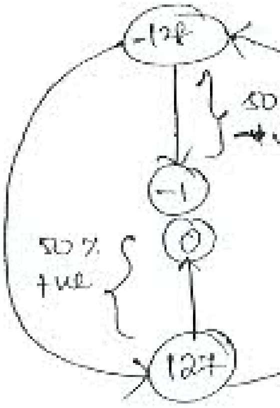


%d	Binary				%u
0	0000	0000	0000	0000	0
32767	0111	1111	1111	1111	32767
-32768	1000	0000	0000	0000	32768
-1	1111	1111	1111	1111	65535

- When we are working with integer & unsigned integer, always format specifier will decide the return value of binary representations.

Char ch;

The size of the character is 1 byte, and the range from -128 to +127.
- For representing any character, we require 8 bit combination; i.e. 256 representations.
- Among those all representations, 50% will represent +ve data & remaining 50% will represent negative data.



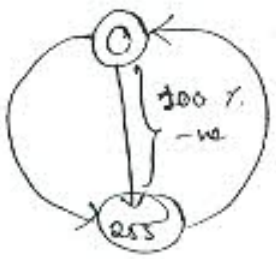
0000 0000 → 0
0111 1111 → 127
1000 0000 → -128
1111 1111 → -1

255	-128
-127	127 (n-1)
128	-1

n

unsigned char ch :-

- the size of unsigned character is 1 byte & the range from 0 to 255.
- For representing any unsigned character, we require 8-bit combinations, i.e. 256 combinations.
- All represent positive data only.



0000 0000 → 0
0111 1111 → 127
1000 000 → 128
1111 1111 → 255

- when we are working with character & unsigned character, datatype decides the return value of the binary representation.

(ex) printf (" %d %u ", 65535, -1) ; // -1 65535

printf (" %u %d ", -32768, 32768) ; // 32768 -32768

printf (" %d %o %x ", 65535, 65535, 65535); // -1 0177777 0xffff

65535 → 1 111 111 111 111 111 → f f f f

%o → 0177777
%d → (-1)
%x → 0xffff

for representing any octal digit, we require 3 binary bits only.
for representing any hexadecimal value, we require 4 binary bits

(ex) printf (" %d %o %x ", 32767, 32767, 32767);

(o/p) 32767 077777 0x7fff.

32767 → 0 111 111 111 111 111 → 7 f f f

%o → 077777
%d → 32967
%x → 07fff

Bit wise Operator :-

- On implementation, when we require to manipulate the data on binary representation then go for bit wise operators.
- when working with bitwise operator, manipulation will take place on memory itself.
- Bitwise operators need to be applied for an integer datatype only. It does not apply for float, double & long double type.

- Bitwise operators are :-

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

1's complement operator (~) :-

- This operator is a unary operator, which returns complement value of given i/p.
- complement value means all 1's will be converted into 0's format and all 0's will be converted into 1's format.

(ex) int a;

1. `a = ~5; // -6`

Processor:

5 → 0000 0000 0000 0101
~5 → 1111 1111 1111 1010 → (-6)

1010 → 2^2 + 2^0

-1 -1 -4 = -6

or -1 - 5 = -6

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

2^2 + 2^0
4 + 1 = 5

65535 − 5 = 65530

65530 − 32767 = 32763 → n

-32768 + 32762 (n-1) = -6

2. `a = ~48; // -49`

48 → 0000 0000 0011 0000
~48 → 1111 1111 1100 1111

2^4 + 2^5

-1 - 16 - 32 = (-49)

65535 → 1111 1111 1111 1111
~48 → 1111 1111 1100 1111

2^4 + 2^5 16 + 32 = 48

65535 − 48 = 65487

65487 − 32767 = 32720 → n

-32768 + 32719 (n-1) = (-49)

3. `a = ~0127 ; // -128`

127 → 0000 0000 0111 1111
~127 → 1111 1111 1000 0000 -1 - 64 - 32 - 16 - 8 - 4 - 2 - 1
= -1 - 127 = (-128)

4. `a = ~0; -1`

0 → 0000 0000 0000 0000
~0 → 1111 1111 1111 1111 → (-1)

5. `a = ~32767 ; // -32768`

32767 → 0111 1111 1111 1111
~32767 → 1000 0000 0000 0000 ↳ (-32768)

6. `a = ~-21; // 20`

-1 → 1111 1111 1111 1111
-21 → 1111 1111 1110 1011 (-1 - 16 - 4)
~-21 → 0000 0000 0001 0100 → (20)

7. `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)

8. `a = ~-32768 ; // 32767`

9. `a = ~-1234 ; 1233`

(ex2) unsigned int a;

1. a = ~5 ; // 65530

5 → 0000 0000 0000 0101
~5 → 1111 1111 1111 1010 (65535 - 5 = 65530)

2. a = ~-1 ; // 0

-1 (65535) → 1111 1111 1111 1111
~-1 → 0000 0000 0000 0000 → (0)

- 1's complement value is not equals to negation of given input
- Always 2's complement value is equals to negation of given input.
- 2's complement = 1's complement + 1

(10) what's the 2's complement of 31?

Ans:- (31)

31 →	0000	0000	0001	1111
1's Compl. of 31 →	1111	1111	1110	0000
+1 →	0000	0000	0000	0001
	1111	1111	1110	0001

(-1 -16 - 8 - 4 - 2 = - 31)

(9) Left Shift operator :-

- In left shift operator, towards from the left side 'n' no. of bits need to be dropped and towards from right side empty places need to be filled with 0's
- In left shift operator, when we are dropping the bits towards from left side, then all 1's will be shifted towards left side by n' places, so automatically value will be increased.

(3x) `int a;`

1. a = 10 << 1 ; //20
2. a = 10 << 2 ; //40
3. a = 10 << 3 ; // 80
4. a = 10 << 4 ; //160

10 →	0000	0000	0000	1010
10<<1 →	0000	0000	0001	0100 → 20 (16+4)
10<<2 →	0000	0000	0010	1000 → 40 (32+8)
10<<3 →	0000	0000	0101	0000 → 80 (64+16)
10<<4 →	0000	0000	1010	0000 → 160 (128+32)

5. a = 1 << 1 ; 2
6. a = 2 << 1 ; 4
7. a = 3 << 1 ; 6
8. a = 100 << 1 ; 200
9. a = 255 << 1 ; 510

note: n<<1 = 2n

10. a = 1 << 15 ; // -32768

1 →	0000	0000	0000	0001
1<<15 →	1000	0000	0000	0000 → (-32768)

11. a = 32767 << 15 ;

32767 → 0111 1111 1111 1111

32767 << 15 → 1000 0000 0000 0000 → (-32768)

12. a = 32767 << 12 (-4097)

32767 → 0111 1111 1111 1111

32767 << 12 → 1111 0000 0000 0000

(-1 -2048 - 1024 - 512 - 256 - 128 - 64 - 32 - 16 - 8 - 4 - 2 - 1)

13. a = 12345 << 16 ; 0
14. a = 32767 << 16 ; 0
15. a = 1 << 16 ; 0

note: n << 16 = 0 for n > 0

- For any +ve integer, left shift 16 (<<16) always makes the value as '0', because +ve data representation will be there in 16-bit towards from left side.

1. a = -5 << 1 ; -10
   a4 << 1 ;
2. a = -5 << 2 ; -20
   a4 << 2 ;
3. a = -5 << 3 ; -40
   a4 << 3 ;
4. a = -5 << 4 ; -80
   a4 << 4 ;

-1 → 1111 1111 1111 1111
-5 → 1111 1111 1111 1011
4 → 0000 0000 0000 0100
(-5)a4 → 1111 1111 1111 1011
-5 << 1 → 1111 1111 1111 0110 (-1 -8 -1 = -10)
-5 << 2 → 1111 1111 1110 1100 (-1 -16 -2 -1 = -20)
-5 << 3 → 1111 1111 1101 1000 (-40)
-5 << 4 → 1111 1111 1011 0000 (-80)

value = x << n ;
⇒ x * 2^n

⑤ Right shift operator :-

- In right shift operator towards from right side, n' no. of bits need to be dropped and towards from leftside empty places need to be filled with 0's if it is +ve number.
- In right shift operator, when we are working with -ve number empty places need to be filled with 1's, so that sign bit will not be modified always we will get -ve value only.

on right shift operator, 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 will be decreased.

ex) int a;

1. a = 20 >> 1 ; 10
2. a = 20 >> 2 ; 5
3. a = 20 >> 3 ; 2
4. a = 20 >> 4 ; 1
5. a = 20 >> 5 ; 0

20 → 0000 0000 0000 0100
20 >> 1 → 0000 0000 0000 1010 → 10
20 >> 2 → 0000 0000 0000 0101 → 5
20 >> 3 → 0000 0000 0000 0010 → 2
20 >> 4 → 0000 0000 0000 0001 → 1
20 >> 5 → 0000 0000 0000 0000 → 0

6. a = 1 >> 1 ; 0
7. a = 2 >> 1 ; 1
8. a = 4 >> 1 ; 2
9. a = 100 >> 1 ; 50
10. a = 500 >> 1 ; 250

note: a = n >> 1 ; $\frac{n}{2}$

11. a = 32767 >> 14 ; (1)

32767 → 0111 1111 1111 1111
32767 >> 14 → 0000 0000 0000 0001

12. a = 32767 >> 15 ; 0
13. a = 1234 >> 15 ; 0
14. a = 1 >> 15 ; 0

note: a = n >> 15 ; 0

- For any +ve integer, right shift is (>> 15) makes the value as 0, because +ve data representation is available in 15 bits towards from right side.

value = x >> n ; $x/2^n$ (for x > 0)

1. a = -10 >> 1 ; -5
   ~9 >> 1 ;
2. a = -10 >> 2 ; -3
   ~9 >> 2 ;
3. a = -10 >> 3 ; -2
   ~9 >> 3 ;

-1 → 1111 1111 1111 1111
-10 → 1111 1111 1111 0110
9 → 0000 0000 0000 1001
~9 → 1111 1111 1111 0110

-10 → 1111 1111 1111 0110

-10>>1 → 1111 1111 1111 1011 (-1-4=-5)

-10>>2 → 1111 1111 1111 1101 (-1-2=-3)

-10>>3 → 1111 1111 1111 1110 (-1-1=-2)

-10>>4 → 1111 1111 1111 1111 (-1)

④ 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 Pf("%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

2|3 → 0000 0000 0000 0011 → 3

2^3 → 0000 0000 0000 0001 → 1

(ex)

Printf("%d %d %d", 30&20, 30|20, 30^20);

30 → 0000 0000 0001 1110

20 → 0000 0000 0001 0100

30&20 → 0000 0000 0001 0100 → 20

30|20 → 0000 0000 0001 1110 → 30

30^20 → 0000 0000 0000 1010 → 10

(ex) printf("%d %d %d", 0&-1, 0|-1, 0^-1); 0, -1, -1

④ printf("%d %d %d", 10&-11, 10|-11, 10^-11); 0, -1, -1

⑤ Printf("%d %d %d", 32767&-32768, 32767|-32768, 32767^-32768);

(0, -1, -1)

note:

n & -n = 0

n | -n = -1

n ^ -n = -1

operators in C' :-

- In C' prog language, total no of operators are 44 and according to the priority these operators are :-

1. (), [], →, . ; struct to member
   ↓ parenthesis, ↓ subscript, ↓ pointer to member
2. +, -, ++, --, !, ~, *, &, sizeof, (type)
   ↳ address of, ↳ indirection
3. *, /, %
4. +, -
5. << (L-shift), >> (R-shift)
6. <, <=, >, >=
7. !=, ==
8. & (bitwise AND)
9. ^ (" XOR)
10. | (" OR)
11. &&
12. ||
13. ?:
14. =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
15. ,

Sizeof

- It is an operator & keyword in C' language.
- By using 'sizeof' operator, we can find the size of given i/p data.
- 'sizeof' is a unary operator, which always returns unsigned integer value.

Ex)
```
void main()
{
    int i;
    float f;
    char ch;
    printf("\n Size of int : %d", sizeof(i));
    printf("\n Size of char : %d", sizeof(ch));
    printf("\n Size of float : %d", sizeof(f));
    getch();
}
```

Q)

```
Size of int : 2
Size of char : 1
Size of float : 4
```

ex) sizeof(int) → 2
sizeof(short) → 2
sizeof(long) → 4
sizeof(signed) → 2
sizeof(unsigned) → 2

sizeof(char) → 1
sizeof('A') → 2
↓↓
sizeof(65) → 2

note) Size of character → 1 byte
Size of character constant → 2 bytes

sizeof(12) → 2
sizeof(12L) → 4
sizeof(12u) → 2

note) By default, character constant returns an integer value, that's why size of the character constant is 2 Bytes.

sizeof(float) → 4
sizeof(double) → 8
sizeof(longdouble) → 10

sizeof(12.6) → 8 (double)
sizeof(12.6f) → 4 (float)
sizeof(12.6L) → 10 (long double)

sizeof(3.0) → 8
sizeof(2.5) → 8

- By default any kind of real value is double, so size becomes 8 bytes.

10/10/11

ex)
```
void main()
{
    int i;
    i=5;
    printf("\n size1 = %d", sizeof(i)); // 2
    printf("\n size2 = %d", sizeof(i*6L)); // 4
    printf("\n size3 = %d", sizeof(i/2.0)); // 8
    printf("\n size4 = %d", sizeof(i/2.0f)); // 4
    printf("\n size5 = %d", sizeof(i=i*10)); // 2
    printf("\n i = %d", i); // 5
}
```

- When we are working with "sizeof" operator, then any kind of expressions can be evaluated except assignment operator.

- On 'sizeof' operator assignment or assignment related expressions will not be evaluated.

eg:
```
int a=50;
printf("%d", sizeof(a=a+10)); // 2
printf("%d", a); // 50
```

(ex) void main ()

```
{
    int i ;
    i=10;
    printf ("\n size of i : %d", sizeof (++i));
    printf ("\n i = %d", i);
}
```

++i ⇒ i = i+1

↑ depends on assignment op^r.

↓ will not be evaluated.

op:

```
Size of i : 2
i = 10
```

(ex) void main()

```
{
    int a, b, c ;
    a=b=1 ;
    c = ++a > 1 || ++b > 1 ;
    printf (" a = %d  b = %d  c = %d", a, b, c) ;
}
```

op:

```
a = 2  b = 1  c = 1
```

- When we are working with logical OR operator, then if any one of the expression is true, then return value is '1'.

i.e. In logical 'OR' operator, when the left side expression is true, then right side part will not be evaluated.

(ex) void main()

```
{
    int a, b, c ;
    a=b=2;
    c= ++a < 2 || --b < 2 ;
    printf (" a = %d  b = %d  c = %d", a, b, c) ;
}
```

op:

```
a = 3 , b = 1  c = 1
```

c = ++a < 2 || --b < 2

= (3 < 2) || (1 < 2)   (3 ← a, 1 ← b)

= 1

- In logical 'OR' operator, right side expression will be evaluated when left side expression is false.

ex5
```
void main ()
{
    int a, b, c, d;
    a=1; b=2; c=3;
    d = ++a < 1 || ++b > 2 || --c < 3;
    Printf (" a=%d b=%d c=%d d=%d", a, b, c, d);
}
```

op:
```
a=2 b=3 c=3 d=1
```

d = ++a < 1 || ++b > 2 || --c < 3
(2 < 1) || (3 > 2)
0 1
1

ex6
```
void main ()
{
    int a, b, c, d;
    a=2; b=4; c=6;
    d = --a > 2 || ++b < 4 || --c > 6;
    Printf (" %d %d %d %d", a, b, c, d);
}
```

op:
```
1 5 5 0
```

d = --a > 2 || ++b < 4 || --c > 6
(1 > 2) || (5 < 4) || (5 > 6)
0 || 0 || 0
0

ex7
```
void main ()
{
    int a, b, c;
    a = b = 1;
    c = ++a < 1 && ++b > 1;
    printf (" a=%d b=%d c=%d", a, b, c);
}
```

op:
```
a=2 b=1 c=0
```

c = ++a < 1 && ++b > 1
= (2 < 1)
0

×

- When we are working with logical 'AND' operator, then both expressions or all expressions must be true, then only it returns '1'.
- In logical 'AND' operator, when the left side expression is 'false' then right side part will not be evaluated.

(8) void main()
{
int a, b, c;
a=2; b=4;
c= ++a > 2 && --b > 4;
printf("%d%d%d", a,b,c);
}

o/p: 3 3 0

c = ++a > 2 && --b > 4
(3 > 2) && (3 > 4)
1 && 0
0

(9) void main()
{
int a, b, c;
a=1; b=3;
c = ++a > 1 && --b < 3;
printf("a=%d b=%d c=%d", a, b, c);
}

o/p: a=2 b=2 c=1

(10) 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 0

(11) 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 2 3 1

note: Logical 'AND' has highest priority, when all are binary operators.

d = ++a > 1 || ++b > 2 && --c < 3
(2 > 1) || (3 > 2) && (3 < 3)
1 || 1 && 0
1 || 0
1

124

- When we are working with binary operators with the combination of unary then always priority need to be given for unary operators only.

(ex 11)
```
void main()
{
  int a, b, c, d;
  a = 2; b = 4; c = 6;
  d = --a > 2 || --b > 4 && ++c > 6;
  printf("%d %d %d %d", a, b, c, d);
}
```

o/p: 1360

d = --a > 2 || --b > 4 && ++c > 6

(1 > 2) || (3 > 4)

0 0

(ex 13)
```
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 3 2 1

(ex 14)
```
void main()
{
  int a, b, c, d;
  a = 4; b = 2; c = 3;
  d = --a < 4 && ++b > 2 || --c < 3;
  printf("%d %d %d %d", a, b, c, d);
}
```

o/p: 3 3 3 1

(ex 15)
```
void main()
{
  int a, b, c, d;
  a = 3; b = 5; c = 4;
  d = --a > 3 && ++b < 5 || --c < 4;
  printf("%d %d %d %d", a, b, c, d);
}
```

o/p: 2 5 3 1

--a > 3 && ++b > 6 || --c < 4

0

125

0 || 1 => 1

```
void main()
{
    int a;
    a = 10, 20, 30;
    printf("a = %d", a);
}
```

a = 10, 20, 30;

o/p: a = 10

- Assignment operator having highest priority than comma (,) operator, that's why first value will be assigned to 'a'.

```
void main()
{
    int a;
    a = (10, 20, 30);
    printf("a = %d", a);
}
```

a = (10, 20, 30);

L → R

- When we equal priority is occurred for comma (,), then it should be evaluated from left to right.

```
void main()
{
    int a;
    a = 10;
    if (a = 0)
        printf("welcome %d", a);
    else
        printf("Hello %d", a);
}
```

a: 10 → 0

if (a = 0)

if(a)

if (0)

else

o/p: Hello 0

```
void main()
{
    int a;
    a = 5;
    if (a = 5)
        printf("welcome %d", a);
    else
        printf("Hello %d", a);
}
```

o/p: welcome 5

Ex20

```
void main()
{
    int a;
    a=0;
    if (a=10)
        printf("welcome %d", a);
    else
        printf("Hello %d", a);
}
```

o/p: welcome 10

a → 10

if (a=10) → if(a) → if(10) → welcome 10

** if (a==10), then o/p:- Hello 0

Ex21

```
void main()
{
    int a;
    a=5;
    if (a=10,1,0)
        printf("welcome %d", a);
    else
        printf("Hello %d", a);
}
```

o/p: Hello 10

a → 5

if (a=10,1,0) → if(a,1,0) → if(10,1,0) → 1 → R → if(0) → else

Hello 10

x 22)
```
void main()
{
    int a;
    a = 10;
    if (a = 0, 1, 2)
        printf ("welcome %d", a);
    else
        printf ("Hello %d", a);
}
```

O/P: welcome 0

a: 10 → 0

if (a = 0, 1, 2) ↓

if (a, 1, 2)

if (0, 1, 2)

↓ L→R

if (2) ↓

welcome 0

x 25)
```
void main()
{
    int i;
    i = 2;
    switch (i = 5, 0, 1)
    {
        case 1 : printf (" A : %d", i);
                 break;
        case 2 : printf (" B: %d", i);
                 break;
        case 3 : printf (" C: %d", i);
                 break;
        default : printf (" D: %d", i);
    }
}
```

O/P: A : 5

i: 2 → 5

switch (i = 5, 0, 1) ↓

switch (i, 0, 1)

↓

switch (5, 0, 1)

L→R ↓

switch (1)

A : 5

Ex 4

```
void main()
{
    int i;
    switch(i = printf("Hi"), printf("bye"))
    {
      case 1: Printf("one: %d", i);
              break;
      case 2: printf("Two: %d", i);
              break;
      case 3: printf("Three: %d", i);
              break;
      default: printf("D: %d", i);
    }
}
```

o/p: Hi Bye Three: 3

switch(i = printf("Hi"), Pf("bye"))
(a) (3)

i
2

switch(i = 2, 3)
↓
switch(i, 3)
↓
switch(2, 3)
↓ 2 → R
switch(3)

Hi Bye Three: 2

Ex 5

```
void main()
{
    switch(3)
    {
      int a; // int a = 1;
      a = 1; // warning: unreachable code
      case 1: printf("Hello %d", a);
              break;
      case 2: printf("Hello %d", a);
              break;
      case 3: printf("Bye %d", a);
              break;
    }
}
```

o/p: Bye garbage/junk

ex26)
```
void main()
{
    int i;
    i = 2;
    switch(i)
    {
        i=10; // skip this stmt.
        case1 : printf(" Hello %d", i);
                break;
        case2 : printf("Hi %d", i);
                i=20;
        case 3 : printf(" Bye %d", i);
    }
}
```

o/p

Hi 2 Bye 20

ex27)
```
void main()
{
    int a, b;
    for (a=1, b=5; a<=10, b<=8; a++, b++)
        printf("\n %d %d", a, b);
    printf("\n a=%d b=%d", a+10, b+10);
}
```

o/p

1	5
2	6
3	7
4	8
a=15	b=19

11/10/12

POINTERS

- A pointer is a derived datatype in C', which is constructed from fundamental datatype of C'-language.
- A pointer is a variable, which holds address of another variable.

Advantages :-

↳ By using pointer, we can access a variable, which is defined outside of the functio
↳ By using pointer, we can handle the data structures more efficiently.
↳ pointers can increase the execution speed of the program.

~~Pointer~~

syntax:

```
Datatype *ptr ;
```

- When we are working with pointers, we require to use following Operators.
  1. & (address of)
  2. * (indirection / dereference of object at location / value at address)
- Address of operator (&) always returns base address of a variable.
- Starting cell of any variable is called baseaddress.
- Indirection operator (*) always returns value of the address.

e.g.:
```
void main()
{
    int a;
    int *ptr;
    ptr = &a;
}
```

↓↓

- 'a' is a variable of type an integer, it is a value type variable, which holds an integer value.
- 'ptr' is a variable of type int *, it's an address type variable, which holds an integer variable address

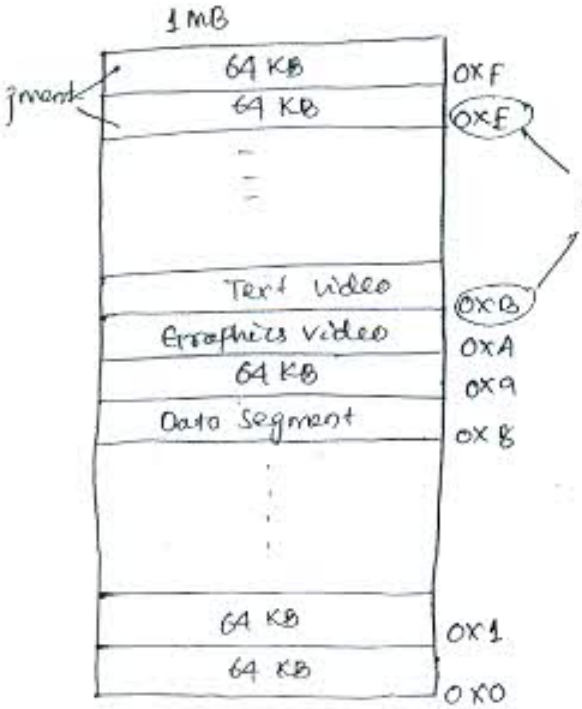
Ex1
```
void main()
{
    int i;
    int *ptr;
    ptr = &i;
    i = 10;
    printf("\n %p %p", &i, ptr);
    printf("\n %d %d", i, *ptr);

    *ptr = 20;
    printf("\n %u %u", &i, ptr);
    printf("\n %d %d", i, *ptr);
}
```

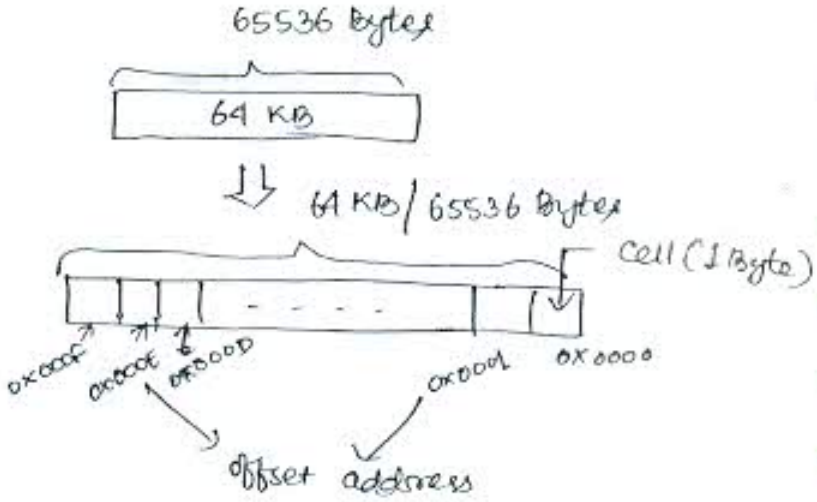
output :

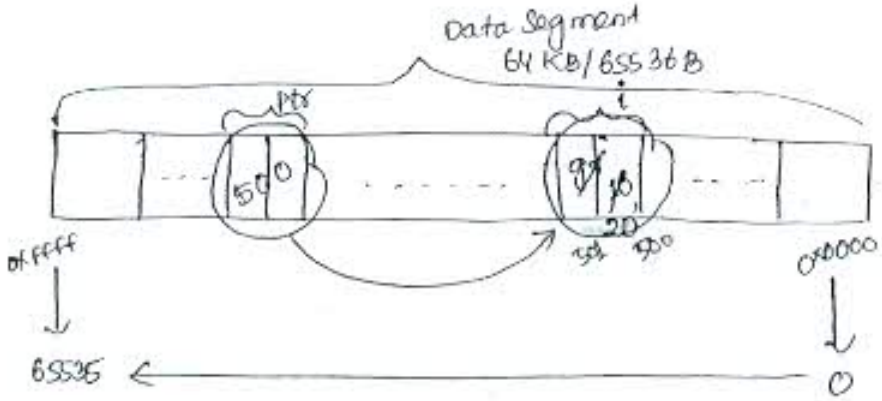
Addrs (H)	Addrs (H)
10	10
Addrs (D)	Addrs (D)
20	20

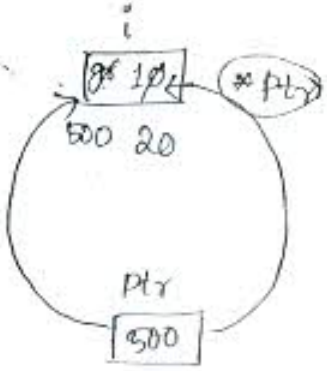
T.C--3.0
↓
8086
↓
DOS 16bit O.S
↓
1MB (occupied space by C program).
↓
16 segments
↓
64 KB (each segment capacity)

1 MB = 64 x 16 KB









500 → 0000 0001 1111 0100
0 1 F 4

O/P :-

0X01F4	0X01F4
10	10
500	500
20	20

↳ 'ptr' always holds address of an integer variable, that's why :-
'&i' and 'ptr' values are same.

↳ Always '*ptr' will returns value of the address, that's why :
'i' and '*ptr' values are same.

↳ TC-3.0 designed on 8086 architecture, i.e. DOS 16 bit O.S. is the platform.

↳ On TC-3.0, whenever a C application is executing, it occupies only 1 mb data.

↳ This complete 1 mb data is devided into 16 equal parts called, segments.

↳ Each & every segment is having an unique identification value, called segment address, which starts from 0x0 and ends with 0xF.

↳ Among those 16 segments, 9th segment; i.e. 0x8 is called data segment.

↳ Wherever we are declaring any variable, it occupies physical memory in data segment only.

↳ Among those all segments, 11th segment; i.e 0xA is called Graphics video segment and 0xB (12th segment) is called text video segment.

↳ Each segment capacity is 64 Kb only; i.e. 65536 Bytes.

↳ Each and every Byte having an unique identification value, called offset address.

↳ This complete 64 Kb data is divided into small partitions called cells.

↳ Each cell capacity is 1 Byte only and offset address range is 0x0000 to 0xffff.

↳ On ~~Borlandcompiler~~ implementation, when we are printing the physical address, then go for "%p, %a, %e, %Le & %Lp" format specifiers only.

↳ By using "%p, %a & %Lp", we can print the physical addresses in hexadecimal format.

↳ By using "%e and %Le", we can print the physical addresses in decimal format.

↳ '%p, %x and %u' will print 16 bit physical address address and '%Le, %Lp' will print 32 bit physical address.

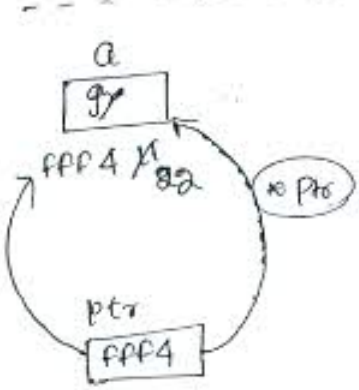
note ↳ For printing the physical addresses, we can't use "%d" format specifier, because:

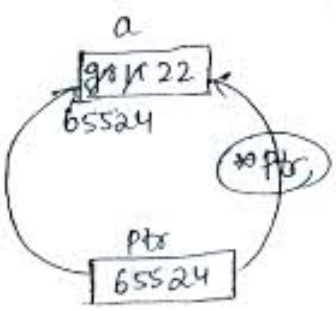
(1) physical addresses are available in hexadecimal format from the range of 0x0000 to 0xffff and in decimal format 0 to 65535, so this range will not be supported by %d

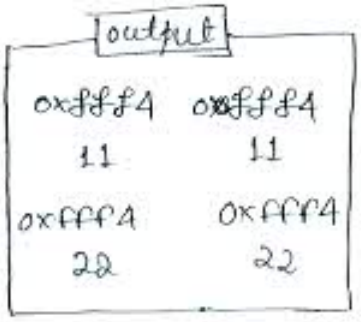
range - ranges to

(2) Physical addresses will not be available in negative format, but '%d' will print negative data also.

Ex2
```
void main()
{
    int a;
    int *ptr;
    ptr = &a;
    a = 11;
    printf("%x %x", &a, ptr);
    printf("\n %d %d", a, *ptr);
    *ptr = 22;
    printf("\n %p %p", &a, ptr);
    printf("\n %d %d", a, *ptr);
}
```

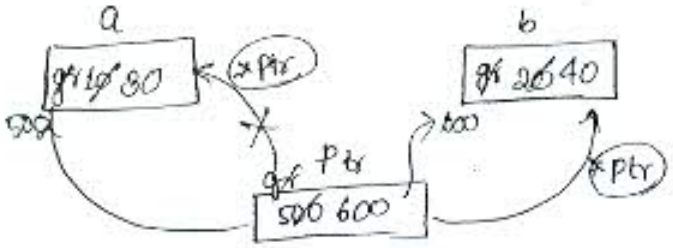






- When we are using '%x' format specifier, then physical addresses will be printed in lower case, '%p' will print the physical address in upper case

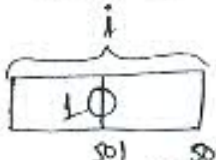
Ex3
```
void main()
{
    int a, b;
    int *ptr;
    ptr = &a;
    a = 10; b = 20;
    Printf("\n %d %d %d", a, b, *ptr); // 10 20 10
    *ptr = 30;
    ptr = &b;
    *ptr = 40;
    Printf("\n %d %d %d", a, b, *ptr); // 30 40 40
}
```



O/P:-
10 20 10
30 40 40

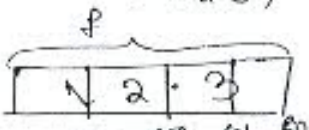
- In implementation, we require to change the pointer location from one variable to another variable, then by reassigning the address we can change the pointer direction.

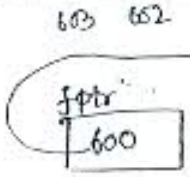
```
int i; ← 2B
int *iptr; ← 2B
iptr = &i;
i = 10;
```




```
*iptr;
(500)
int *iptr;
↓
2B *iptr;
↓
2B (500, 501)
↓
(10)
```

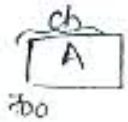
```
float f; ← 4B
float *fptr; ← 2B
fptr = &f;
f = 12.3;
```

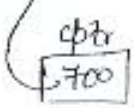





```
*fptr;
(600)
float *fptr;
↓
4B (600, 601, 602, 603)
↓
(12.3)
```

```
char ch; ← 1B
char *cptr; ← 2B
cptr = &ch;
ch = 'A';
```






```
*cptr;
700
char *cptr;
↓
1B (700)
↓
(A)
```

- Different types of variables holds different types of data, that's why sizes will be different.
- Any kind of pointer size is same only, because it holds similar type of data; i.e. address.
- In DOS OS., addresses are limited, because it doesn't contains memory management.
- In DOS based compiler (TC-3.0 or 16 bit compiler or DOS based compiler or BOS based compiler), the size of the pointer is 2 bytes only, because addresses are limited from the range of 0x0000 to 0xffff (0 to 65,535).

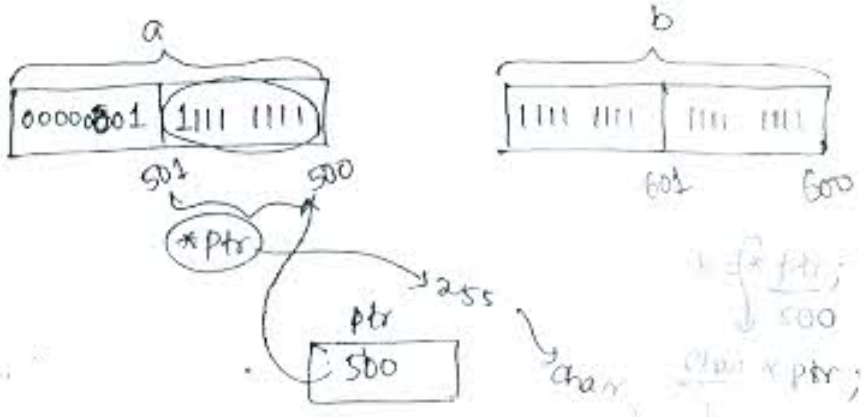
- On windows O.S., at compile time millions of addresses can be created, because it contains proper memory management.
- On windows based compiler (TC-4.5 or 32-bit compiler or gcc compiler or windows/unix/linux), the size of the pointer is 4 bytes, because millions of addresses will be created in 32-bit compilers.
- Any type of pointer size is 2 bytes only, because it holds common data from 0 to 65,535.
- When we are working with pointers any kind of pointer will hold single cell information only, i.e base address of a variable.
- On pointer variable, when we are applying the indirection, then it dereference to pointer type, so depending on pointer type from given address, 'n' no. of bytes will be accessed.
- Any kind of pointer can hold any kind of variable address, but we can't manipulate the data properly, because indirection operator behaviour is datatype dependent
- On implementation, when we are manipulating the integer variable, then go for "int *", float variable → "float *" and character variable → "char *" only, because we can see the difference, when we are applying indirection operator.

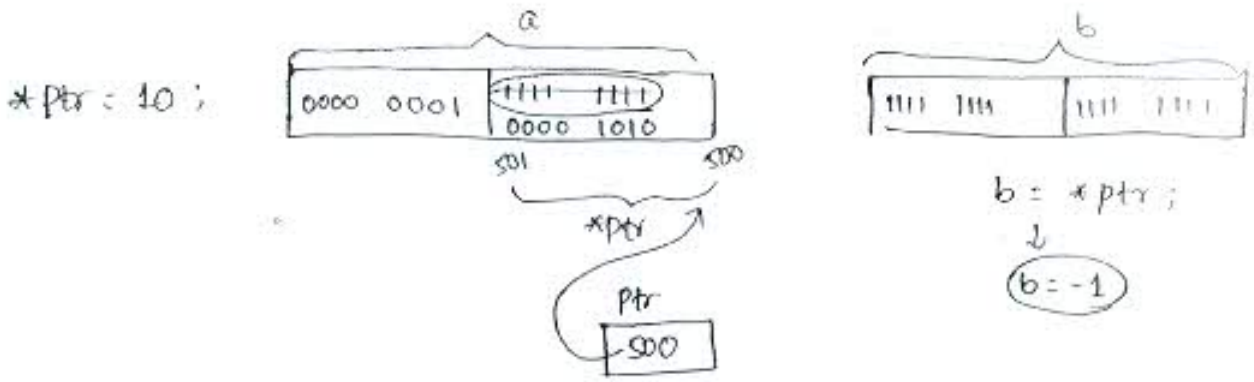
(ex4)
```
void main()
{
    int a, b;
    char *ptr;
    ptr = &a;
    a = 511;
    b = *ptr;
    Printf("\n %d %d %d", a, b, *ptr);
    *ptr = 10;
    printf("\n %d %d %d", a, b, *ptr);
}
```

o/p:-

511 -1 -1
256 -1 10

511 = 0000 0001 1111 1111





- On integer variable, when we are applying the character pointer, then it can access and manipulate only 1-byte data.

note When we are working with pointers, we'll get following warning messages:

① suspicious pointer conversion :-

↳ This warning message, we'll get when we are assigning a variable address into different type of pointer.

- suspicious pointer conversion will not be allowed in C++.

② non portable pointer conversion :-

↳ This warning message, we'll get when we are assigning value type data to a pointer.

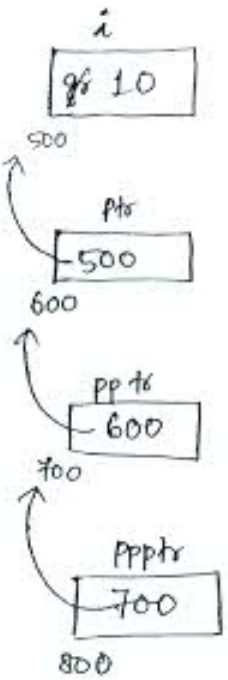
Pointer to pointer :-

- It's a concept of holding a pointer address into another pointer.
- Pointer to pointer concept can be applied upto 12 stages, but generally there is no limitations.
- When we are increasing the pointer to pointer relation, then automatically performance will be decreased.

- syntax:

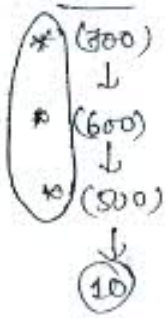
Pointer : Datatype *ptr ;
P 2 P : Datatype **pptr ;
P2P2P : Datatype ***pppptr ;
P2P2P2P : Datatype ****ppppptr ;

(10)
```
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
⑤ &ptr = 600
⑥ pptr = 600
⑦ *pptr = 500
⑧ **pptr = 10
*(600)
⑨ *(500)
⑩

⑨ &pptr = 700
⑩ ppptr = 700
⑪ *pppptr = 600
⑫ **pppptr = 500
⑬ ***pppptr = 10



→ TC-3.0 designed on 8086 architecture, in this architecture memory models are divided into six types. i.e :-

① Tiny
② Small
③ Medium
④ Compact
⑤ Large
⑥ Hugo

- By default any application memory model is small in TC-3.0.
- In Turbo C++-4.5, memory models are divided into 4 types :-
  i) Small
  ii) Medium
  iii) Compact
  iv) Large.

- By default any application memory model is 'large' in TC++-4.5.
- Always memory model decides that "what type of pointer needs to be created?".
- Depending on the memory model, pointers are classified into three types :-

i) near pointer
ii) far pointer
iii) huge pointer

i) near pointer :-

↳ which pointer variable can handle only one segment of one 1 MB data, is called near pointer.

↳ Always, near pointer handles only one segment, i.e 'data segment'.
↳ The size of the near pointer is 2 bytes.
↳ When we are increasing the near pointer value, then it increase offset value address only.
↳ When we are comparing the near pointers, then it compares offset address only.
↳ By using 'near' keyword, we are creating near pointer.
↳ By default, any type of pointer is near only.

ii) far pointer :-

↳ which pointer variable can handle any segment of one 1 MB data, is called far pointer.
↳ when we are working with far pointer, it can handle any segment from the range of 0x0 to 0xf, but at a time only one segment.
↳ The size of the far pointer is 4 bytes.
↳ When we are increasing the far pointer values, then it increase offset address only.
↳ When we are comparing the far pointers, then it compares segment address along with the offset address also.
↳ By using 'far' keyword, we can create 'far pointers'.

iii) huge pointer :-

↳ which pointer variable can handle any segment of 1 Mb data, is called huge pointer.

> By using huge pointer, we can handle any segment from the range of '0x0 to 0xF' but at a time only one segment.

> when we are increasing the huge pointer value, then it increases offset address along with segment address also.

> when we are comparing the huge pointers, then it compares normalization value

↳ 'Normalization' is a process of converting 32 bit physical address into 20 bit hexadecimal format.

↳ The size of the huge pointer is 4 bytes.

↳ By using 'huge' keyword, we can create huge pointers.

↳ In implementation, when we require to handle more than 64 KB data, then go for huge pointers.

Normalization formula :-

① In decimal :-

Physical add. = (Segment Add) * 16 + offset address

② In hexadecimal :-

Physical Add. = (Segment Add.) * 0x10 + offset address.

(ex) what's the normalization value of 0x52486791 ?

Sol :- Physical huge Address :- 0x52486791.

Segment Address :- 0x5248

Offset Address :- 0x6791

64 KB :-
Set
↓
Rows
↓
Cols

Physical Add. = (Segment Add. * 0x10) + offset Add.

= (0x5248 * 0x10) + 0x6791

= 0x52480 + 0x6791

= 0x58C11

```
  0x52480
+ 0x6791
---------
  0x58C11
```

decimal
- 1 * 10 = 10
- 12 * 10 = 120
- 123 * 10 = 1230
- 1234 * 10 = 12340

octal
- 01 * 010 = 010, 1 * 8 = 8
- 012 * 010 = 0120, 10 * 8 = 80
- 0123 * 010 = 01230, 83 * 8 = 664

hexa decimal
- 0x1 * 0x10 = 0x10
- 0x12 * 0x10 = 0x120
- 0x123 * 0x10 = 0x1230

near

① 0x8 + Segment no

② 2B (offset)

far

① 0x0 to 0xF

② 4B (offset, segment)

huge

① 0x0 to 0xF

② 4B (offset, segment)

19/10/12

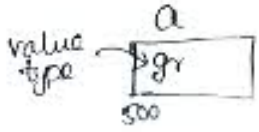
Wild pointer / bad pointer :-

Uninitialized pointer variables are, their pointer variables are not initialized with ~~with~~ any variable address, and these are called as wild pointer.

- wild pointer is also called bad pointer, because without assigning any variable address it points to some memory location.

- ex:-

```
void main()
{
    int a;
    int *ptr; // wild pointer
}
```



Wherever we are declaring any pointer always recommended to initialize with any variable address or else make it null.

Null pointer :-

- Which pointer variable is initialized with 'null', is called null pointer.

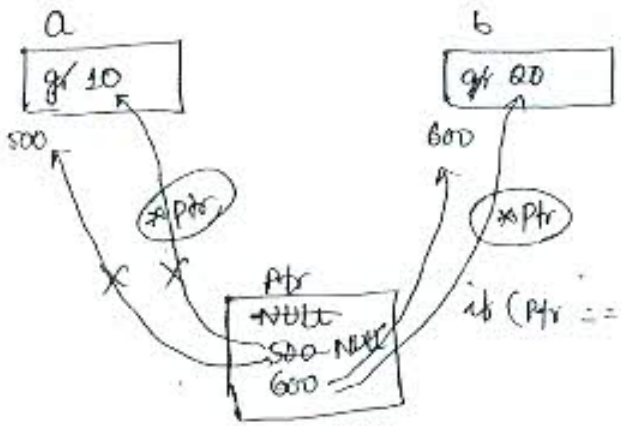
Null pointer is not holding any variable address until, we are not assigning any variable data.

Syntax:-

```
Datatype * ptrs = NULL;  → <stdio.h> should be included.

Datatype * ptrs = (Datatype *)0 ;
```

Ex:-

```
void main()
{
   int a, b ;
   int * ptr = (int *)0 ;
  // int * ptr = NULL ; <stdio.h>

   if ( ptr == (int *) 0) // if (ptr == NULL)
   {
      ptr = &a ;
       a = 10;
       printf("\n value of a : %d", *ptr ) ;
       ptr = (int *)0;
   }
   if (ptr == (int *)0)
   {
       ptr = &b ;
       b = 20;
       printf ("\n value of b : %d", *ptr);
   }
}
```



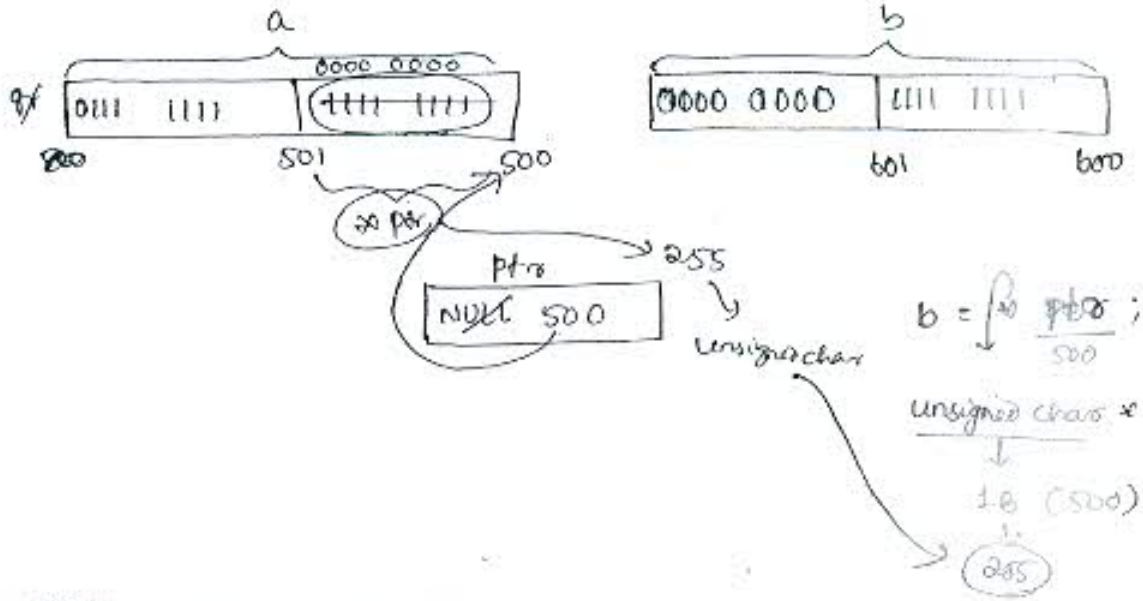
value of a : 10

(ex) int * ptr = 0 ; // non portable pointer conversion
(value typ)

(ex2)
```
void main()
{
    int a, b;
    unsigned char *ptr = (unsigned char *) 0;
    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	255	255
32512	255	0

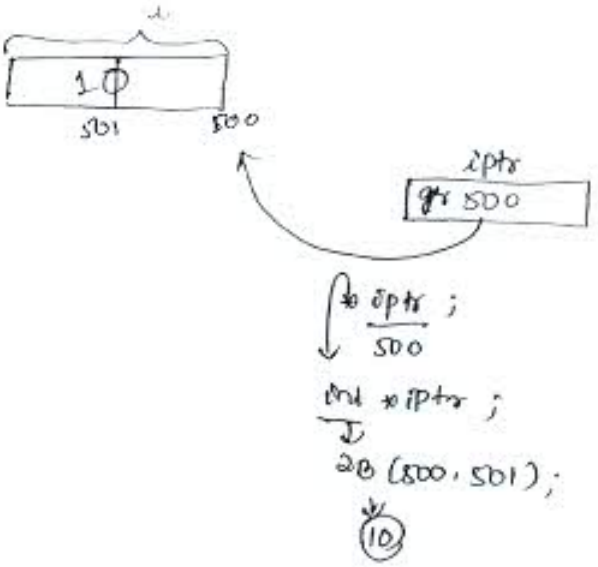


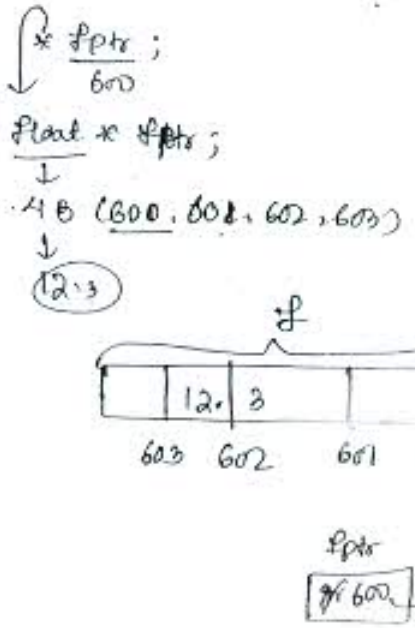
- On integer variable, when we are applying unsigned character pointer, then it can access and manipulate only 1 Byte data.

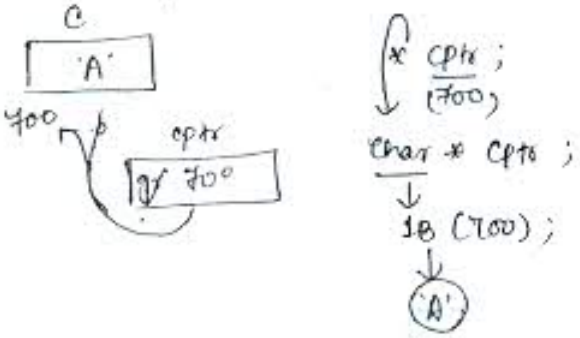
void pointer :-

- Generic pointer of C and C++ is called void pointer.
- Generic pointer means it can access and manipulate any kind of data property.
- The size of the void pointer is 2 Bytes.
- When we are working with void pointer to access the data, typecasting process must be required or else it gives an error.
- When we are working with void pointer, arithmetic operations are not allowed, i.e. incrementation or decrementation of the pointer is not possible.
- When working with void pointer, type specification will be decided at time of execution only.

(200)
```
void main()
{
    int i;
    float f;
    char ch;
    int *iptr;
    float *fptr;
    char *cptr;
    iptr = &i;
     i = 10;
    Printf("%d %d", i, *ptr);

    fptr = &f;
    f = 12.3;
    Printf("%f %f %f", f, *fptr);

    cptr = &ch;
     ch = 'A';
    Printf("%c %c", ch, *cptr);
}
```



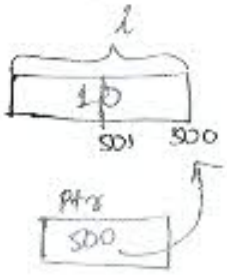


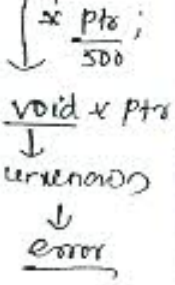


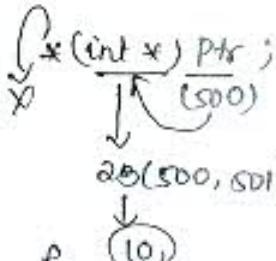
- In the above program, in place of constructing three pointers, we can create a single pointer, which can manipulate & access the data property; i.e. void pointer is required

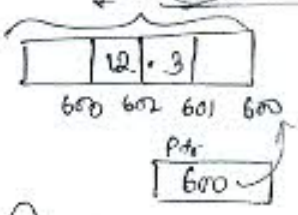
Ex 4)

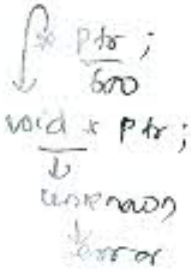
```
void main()
{
    int i;
    float f;
    char ch;
    void *ptr;
    ptr = &i;
    i = 10;
    // Printf("\n %d %d", i, *ptr); error
    //Printf("\n %d %d", i, *(int *)ptr);

    ptr = &f;
    f = 12.3;
    Printf("\n %f %f", f, *(float *)ptr);

    ptr = &ch;
    ch = 'A';
    Printf("\n %c %c", ch, *(char *)ptr);
}
```

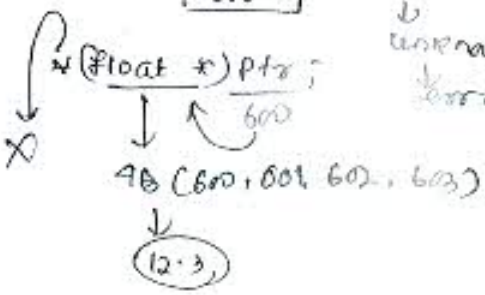


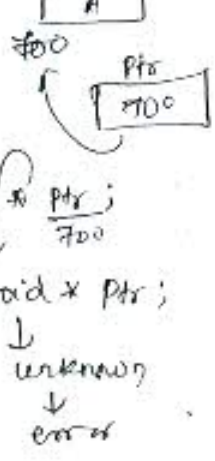


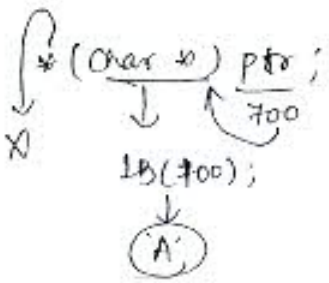












Arithmatic operations on pointer :-

```
int i1, i2;
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 ; previous address

⑥ --p2 ; } previous address
p2 = 145;

⑦ p1 * p2 ; }
⑧ p1 / p2 ; } error.
⑨ p1 % p2 ; }

Pointer rules :-

Rule 1

Addrs + Number = Address (Next Address)
Addrs − Number = Address (Prev. Address)
Address ++ = Address (Next Addrs)
Address −− = Address (Prev. Addrs)
++ Addrs = Address (Next Addrs)
−− Addrs = Address (Prev. Addrs)

Rule 2

Address − Address = Number (No. of elements)
= Size diff / size of (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 use relational operator and conditional operator between two pointers. (<, >, <=, >=, ==, !=, ? :)

Address > Address = T/F
Address >= Address = T/F

Rule 5

We can't perform bitwise operations between two pointers like :-

Address & Address
Address | Address
Address ^ Address
~ Address

} Illegal.