DELTA Pg No.
Date / /

Software – As per the industrial standard a digitalized (GUI) automated system is called software.

When the software where is providing graphically user interface then it is called digitalized.

Without human interaction if the process is completed then it is called automated system.

A software is nothing but a collection of programs.

A program is a set of instructions which is designed for a particular task.

A number of programs combining together like a single unit it is called software tool or software component

Software are classified into two types –

1) System software
2) Application software

(1) **System software** – The software design for general purpose & does not having any limitation it is called system software.

System softwares are classified into 3 types

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

DELTA Pg No.
Date / /

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

Ex:- application softwares are classified into two types

① Application Packages - ex- MS Office, Oracle

② Special Purpose softwares - ex- Tally

– MS Office is a microsoft product which maintain the information in document format.

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

– By using Tally we can maintain the information of accounts

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

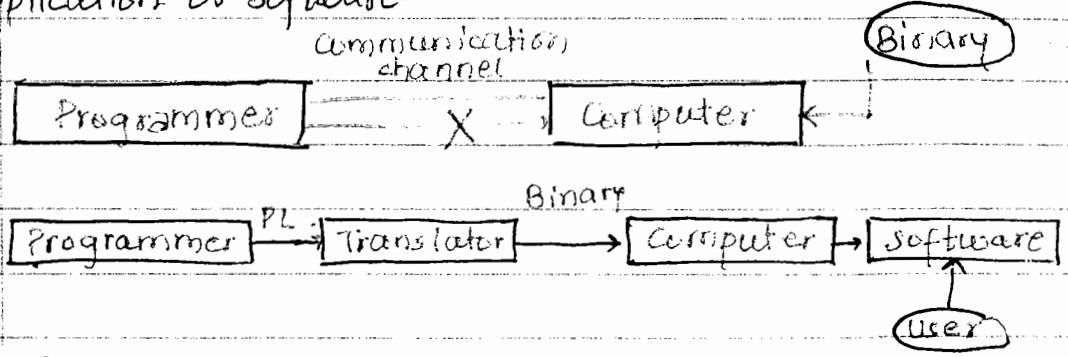
Computer knows only one language i.e binary lang

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

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

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

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



Translators -

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

Translators are classified into 3 types -

① Compiler
② Interpreter
③ Assembler

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

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

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

As per the performance wise, recommended to use compiler

As per the development wise recommended to use an interpreter

PROGRAMMING LANGUAGE

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

Programming lang. is classified into two types-

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

(1) HIGH LEVEL PROGRAMMING LANGUAGE

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

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

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

(2) LOW LEVEL PROGRAMMING LANGUAGE

This programming lang. is also called assembly lang.

C Programming Language

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

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

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

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

HISTORY OF C

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

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

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

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

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

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

DELTA Pg No.
Date / /

"K and R-C"

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

ASCII → American Standard code for information Interchange.

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

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

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

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

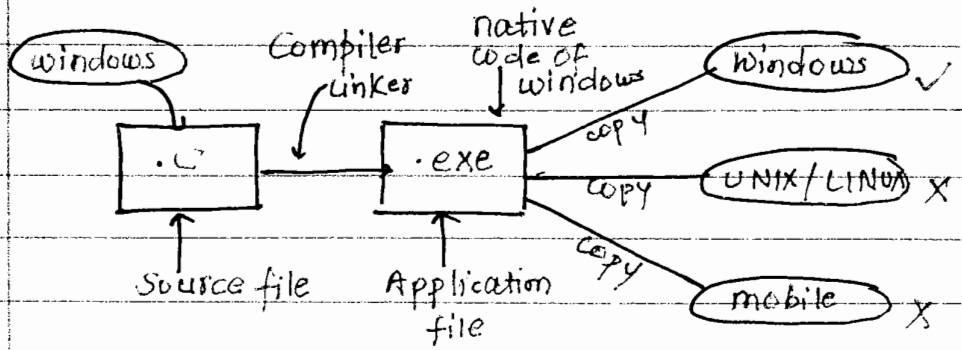
Applications of C

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

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

Advantages of C

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



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

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

Platform dependent, Independent

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

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

C Programming language

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

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

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

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

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

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

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

28/5/2015

DELTA Pg No. Date / /

Characteristics of C

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

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

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

(1) Keyword - It is a reseive word, some meaning is already available to this word & that meaning automatically recognizable by compiler.

In C prog. lang. we are having 32 Keywords ex:- if, else, void, for, break - - - etc.

(2) Operator - It is a special kind of symbol which performs a particular task.

In C prog. lang. we are having 44 operators ex- +, -, * ...

3) Seperators - By using seperator, we can seperate, an individual unit called token.

In C prog. lang. total no. of seperators are 14. ex:- , ; : ' ' " " { } space, etc

4) Constants - It is a fixed one never change during the execution of program

In C, C constants are classified into 2 types -

1) Alpha numeric constants
2) Numeric constants

DELTA Pg No.
Date / /

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

Alpha numeric constants are classified into two types -

a) Character constant

b) String constant

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

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

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

ex:- "freshjobs" "Hello"

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

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

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

a) Integer

b) Float

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

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

Ex- 12.3, 14.85, 98.10

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

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

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

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

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

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

- Unary Operator
- Binary Operator
- Ternary Operator

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

Binary takes 2 operands and ternary is required 3 operands.

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

DELTA Pg No.
Date / /

Assignment Operator :-

→ It is a binary Operator. 1)

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

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

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

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

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

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

29

Arithmetic Operator :-

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

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

DELTA Pg No.
Date / /

* Binary Arithmetic Operators –

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

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

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

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

1. * / %
2. + –
3. =

29/5/2015

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

DELTA Pg No.
Date / /

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

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

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

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

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

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

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

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

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

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

DELTA | Pg No. | Date / /

Modulus Operators (%) :-

This operator returns remainder value.

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

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

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

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

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

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

DELTA Pg No.
Date / /

→ We can't apply it for float datatype

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

Syntax:-

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

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

→ % 10 always provide last digits of input value.

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

DELTA Pg No.
Date / /

Relational and logical Operators

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

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

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

→ Relational Operators are :-

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

→ logical Operators are :-

&&, ||, !

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

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

04/8/2015

DELTA Pg No.
Date / /

Relational OPerators

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

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

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

L = R ; → assignment

L == R ; compares

comparision = equals

L != R → not equals

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

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

DELTA Pg No.
Date / /

1. a = 5 !₀ = 5; 0

2. a = 2! = 8; 1

3. a = 2 > 5 !₀ = 5 < 8; 1

4. a = 1 !₀ = 5 < 8; 0

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

1 !₀ = 0 == 1 !₀ = 1

1 == 1 !₀ = 1

1 !₀ = 1

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

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

&&	||	!
T T → T	TT → T	T → F
T F → F	TF → T	F → T
F T → F	FT → T	
F F → F	FF → F	

Every non-zero should be called as 1.

DELTA Pg No.
Date / /

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

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

OR Combinations

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

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

1 != 0 || 0 != 1

1 || 1

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

1 != 2 || 1 != 5

1 || 1

NOT Combinations

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

   0 != 5 ;
4. a = !(2 < 5 && 2 > 5) ; 1

   !(1 && 0)

Points

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

DELTA Pg No.
Date / /

- C Programming Language is called CASE Sensitive Language i.e. upper & lower case contents both are different
- In C Programming language all existing keywords, and pre-defined functions, all are available in lower case only. (1)
- To write a C Program, we need **IDE** (Integrated Development Environment)

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

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

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

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

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

DELTA | Pg No. | Date / /

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

Diff. b/w Turbo C++ and GCC

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

(2) Inc/dec

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

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

05-6-15

1st Prog

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

O/P: Welcome

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

DELTA | Pg No.
Date / /

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

Printf():-

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

Syntax :-

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

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

DELTA Pg No. Date / /

FORMAT SPECIFIERS

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

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

* Designing a prog on linux OS.

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

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

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

O/P: Balututorials

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

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

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

DELTA Pg No.

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

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

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

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

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

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

TOKEN

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

② A C prog. is a combination of tokens.

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

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

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

O/P: Error

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

O/P: Hello

1. printf ("Hello"); Hello

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

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

function call missing

" " 10 not seperated by comma

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

function call missing

20, seperator then, compiler waits for one more argument

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

statement missing

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

10 welcome 20

FAILURE

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

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

102030 no clarity

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

10 20 30

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

10, 20, 30

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

DELTA Pg No.
Date / /

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

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

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

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

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

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

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

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

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

O/P 0 1

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

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

* VARIABLE Declarations

(i) Name of the memory location is called variable

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

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

(iv) In 'C' prog. lang. variables required to declare on top of the prog. after opening the body b4 writing first statement.

DELTA Pg No.
Date / /

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

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

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

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

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

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

DELTA Pg No.
Date / /

Syntax to Initialize a Variable

Data Type variable = Value;

Ex-

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

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

DELTA Pg No.
Date / /

because -

Predefined reserved words are possible to redefine but Keywords are not possible to redefine. like printf is keyword not defined in java, C#, etc.

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

1. CAMEL NOTATION -

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

eg: ~~Total~~ int TotalSalary;

2. Hungarian Notation -

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

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

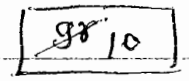
In declaration of the variable after creating the memory it stores garbage value until we are assigning the data

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

DELTA Pg No.
Date / /

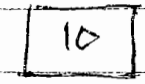
```
int a;
a=10;
```

a




```
int a=10;
```

a



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

vi file2.c

2. for enabling typing, press i button

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

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

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

Run / load : ./file2

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

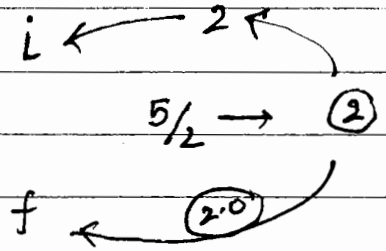
DELTA Pg No.
Date / /

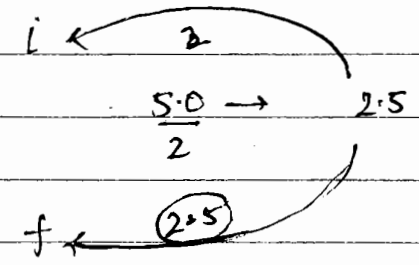
→ Operator behaviour is always operand dependent only i.e depends on input values only, behaviour of operator will be changed.

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

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

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





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

Control Flow Statements

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

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

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

(1) Selection Statements

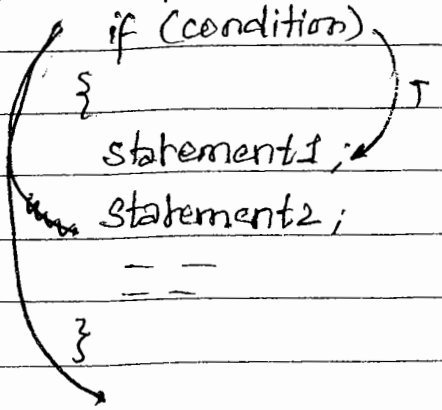
→ These are also called decision making statements

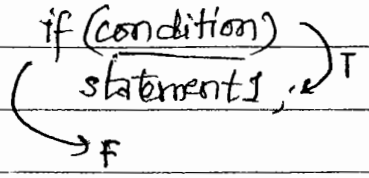
→ By using them, we can create conditional oriented block.

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

DELTA Pg No.
Date / /

Syntax to if :-





→ Constructing the body is always optional.

→ Body is recommended to use when we having multiple statements

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

* Else :-

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

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

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

Syntax

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

DELTA Pg No.
Date / /

Syntax

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

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

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

if → ms
else → s

③
```
if (condition)
  statement 1;
else
  statement 2;
```
T → statement 1;
F → statement 2;

if - s
else s

④
```
if (condition)
  statement 1;
else
{
  Statement 2;
  statement 3;
}
```
T → statement 1;
F → { Statement 2; statement 3; }

if - s
else - ms

PROGRAM

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

// if (1 && 1)

WARNING MESSAGE
When

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

O/P : ABNITC Welcome

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

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

PROGRAM

(1)

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

O/P : NITC

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

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

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

DELTA Pg No.
Date / /

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

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

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

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

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

else scope starts only after if scope only.

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

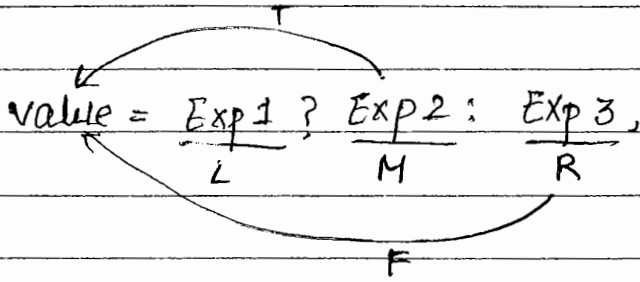
CONDITIONAL OPERATORS (? :) →

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

DELTA Pg No.
Date / /

and left side argument is treated like condition.

Syntax:



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

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

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

• int a;

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

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

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

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

   No. of ? and : are equal

DELTA | Pg No. | Date / /

4. `a = 2<5!=1 ? 10 : 5>2 ? 20 : 30;`

O/P : L 1 m 1 | 2 2 R

upto 1st ? → L

after 1st ? Before 1st : → m

after 1st : Before ; → R

`1 != 1` | `5>2 ? 20 : 30` → L m R

O/P = 20

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

L M R

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

1 2 2 1

L m R

O/P → 10

* int a.

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

L 1 m 1 R 2 2 3 3

L m R

L m R

O/P → 30

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

L m R

False → then right side.

`5>2 ? 30 : 40`

L m R

True then m (30)

DELTA Pg No.
Date / /

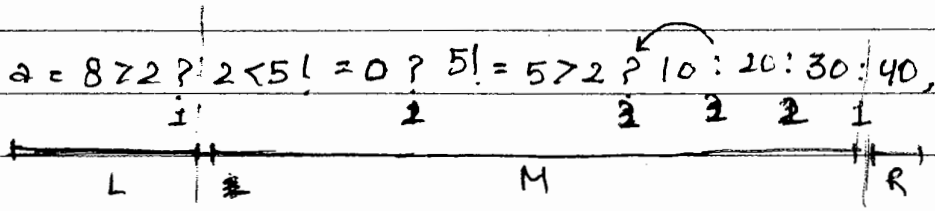
→ When we are working with multiple ? and : then initially we required to convert the expression into 3 arguments.

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

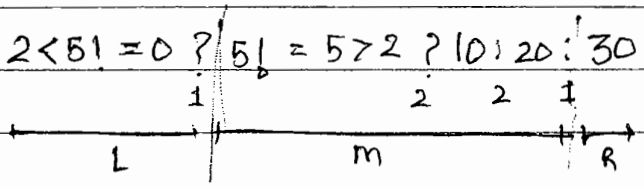
→ According to numbering process, for every ? one unique no. required to assign and for every ':' corresponding ? no. required to assigned.

→ Upto 1st question mark, it is called left argument after 1st ?, b4 1st ? corresponding : should be middle argument and remaining complete part is Right side argument

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



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



5! = 5>2 ? 10 : 20

L m R

5! = 1

O/P : 10

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

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

L m R

L m R

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

L m R

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

L m R

O/P : 70

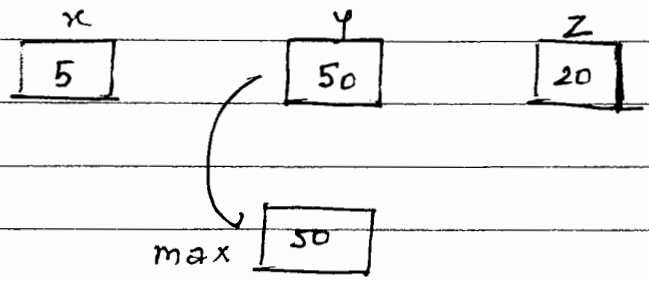
DELTA Pg No. Date / /

DELTA Pg No.
Date / /

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

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

PROGRAM




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

O/P : Max value is: 50

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

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

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

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

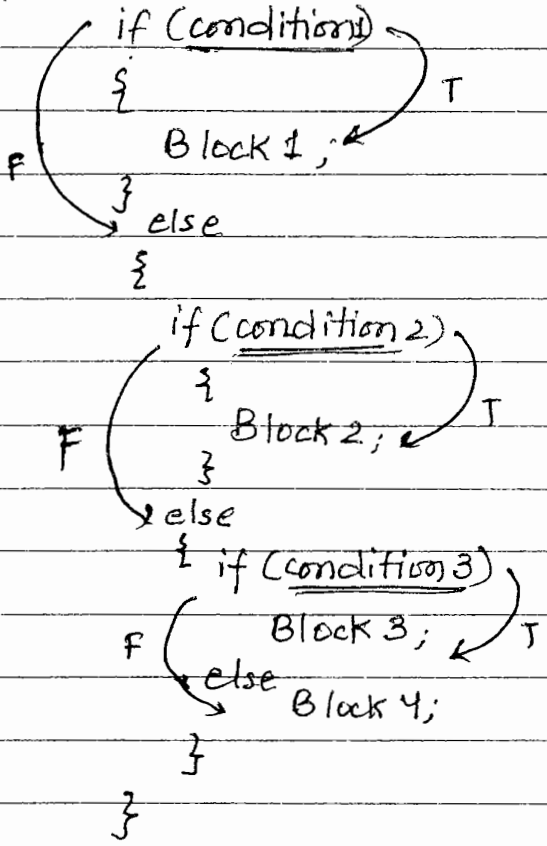
*
```
void main()
{
  int x, y, z, max;
  x=5; y=50; z=20;

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

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

DELTA Pg No.
Date / /

NESTED If-else

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

Syntax:

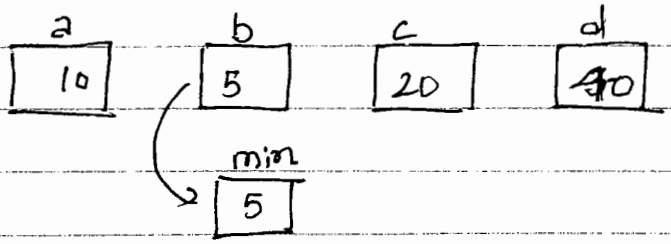


* According to syntax, if condition 1 is true then block 1 is executed, if it is false then control will pass to else part.
* Within the else part if condition 2 is true then block 2 will executed, if it is false then control will pass to nested else.
* Within the nested else, if condition 3 is true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

DELTA Pg No. Date / /

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

— Prog:




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

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

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

* **Scanf**

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

Syntax:-

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

* **clrscr**

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

* **getch()**

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

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

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

* **Comments**

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

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

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

1) Single line
2) multi-line

DELTA Pg No.
Date / /

→ Single line comments can be provided by using //

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

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

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

Prog:

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

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

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

1) Serial no. consider as 4235

2) Tariff data is 1 – 50 1.75
51 – 150 3.75

DELTA BOOKS

// 151 - 250 5.00

>= 251 6.50

3) Previous reading value is 1234

4) min. charged amt. is 40 rs.

5) Service tax need to applied to 12.36%

~~Void main()~~

~~#~~ include <stdio.h>

include <stdlib.h>

{

int sno, cread, pread = 1234, nunits, t;

float ops;

printf ("Enter sno:");

scanf ("%d", &sno);

if (sno = 4321)

{

printf ("Enter current reading");

scanf ("%d" &cread);

if (cread > = pread)

{

nunits = cread - pread;

if (nunits > 251)

{

t = nunits - 250

ops = t* 6.50 // 4th

ops = ops + 100*5.00; // 4+3rd

ops = ops + 100*3.75; // 4+3+2nd

ops = ops + 50*1.75; // 4+3+2+1st

}

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

return EXIT-SUCCESS; // return 0;

}
```

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

// Run / Load command : ./ebill

O/p : Enter sno : 4321

Enter current reading : 1234

Total Amount : 44.94

DELTA Pg No.
Date / /

* Loops

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

⇒ Basic Purpose of loop is code repetition

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

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

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

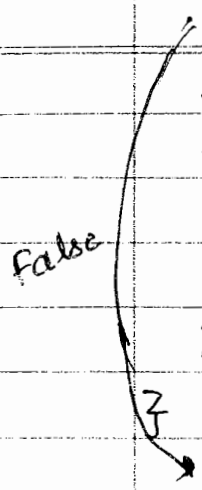
(1) While loop

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

Syntax :-

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

True



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

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

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

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

o/p: 1 3 5 7 9

DELTA Pg No.
Date / . /

* **for loop :-**

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

1) Initialisation
2) Condition
3) Iteration

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

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

P8

while () ⟶ error

for (; ;) ⟶ valid

- When the condition part is not given in for loop, then it repeats infinite times beoz condn part is replaced with non-zero value (True value).
- When we are working with for loop, it repeats in anticlock direction.
- Always prechecking process occurs when we are working with for loop i.e b4 execution of statement block condn part is executed.

while (0) ⟶ No repetition

for (;0,) ⟶ 1's it will repeat

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

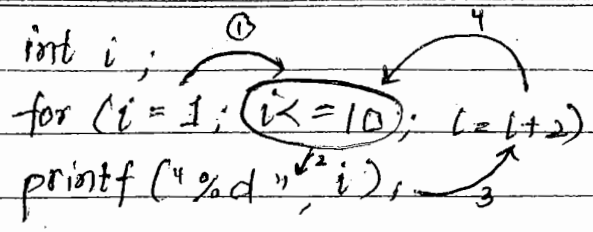
```
int i;
i = 0;
while (i)  -> No repetition
for (; i;) -> No rep.
```

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

Prog:

```
void main()
{
```

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



O/P : 1 3 5 7 9

3) **do - while :-**

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

Syntax :

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

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

Prog:-

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

O/P: 1 3 5 7 9

Working with while loop

(1) Increment order

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

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

(2) Decrement Order

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

DELTA | Pg No.
Date / /

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

Task 1

2 4 6 8 10 12 14 16 18 20

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

Task 2

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

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

DELTA PAGE
DATE / /

Prog :- Enter a value : 30

1 3 5 6 7 9 11 12 13 15 17 18 19 21 23 24
25 27 29 30.

```
void main()
{
  int i, a
  i = 1;
  Printf ("
          ", &a);
  while (i<=a)
  {
    printf("  ", i);
    i = i+1.
    if (i%5==0)
    {
      printf ("
              ");
    }
  }
}
```

OR

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

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

Prog [Fibonacci Series]

Enter a value : 100

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

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

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

i: 0 → 1 → 2

j: 1 → 1 → 3

a: 1 → 2 → 5

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

Again

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

DELTA Pg No.
Date / /

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

How many values you want to print? – 15

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

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

→ Enter 2 values: 10 200

13 21 34 55 89 144

→ Enter 2 values : 5 200

5 8 13 21 34 55 89 144

Prog 1

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

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

o–

:

3

.

given

:");

Auto-alignment concept

DELTA Pg No. Date / /

Prog

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

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

Auto alignment

DELTA PG No.
Date / /

* \n

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

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

Case 1:

Enter 2 values : 10 20 (n1, n2)

10 12 14 16 18 20

or

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

10 12 14 16 18 20

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

20 18 16 14 12 10

or

Enter 2 values : 21 10 (n1, n2)

20 18 16 14 12 10

case 3: Enter 2 values : 10 10

Both values are same

DELTA Pg No.
Date / /

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

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

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

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

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

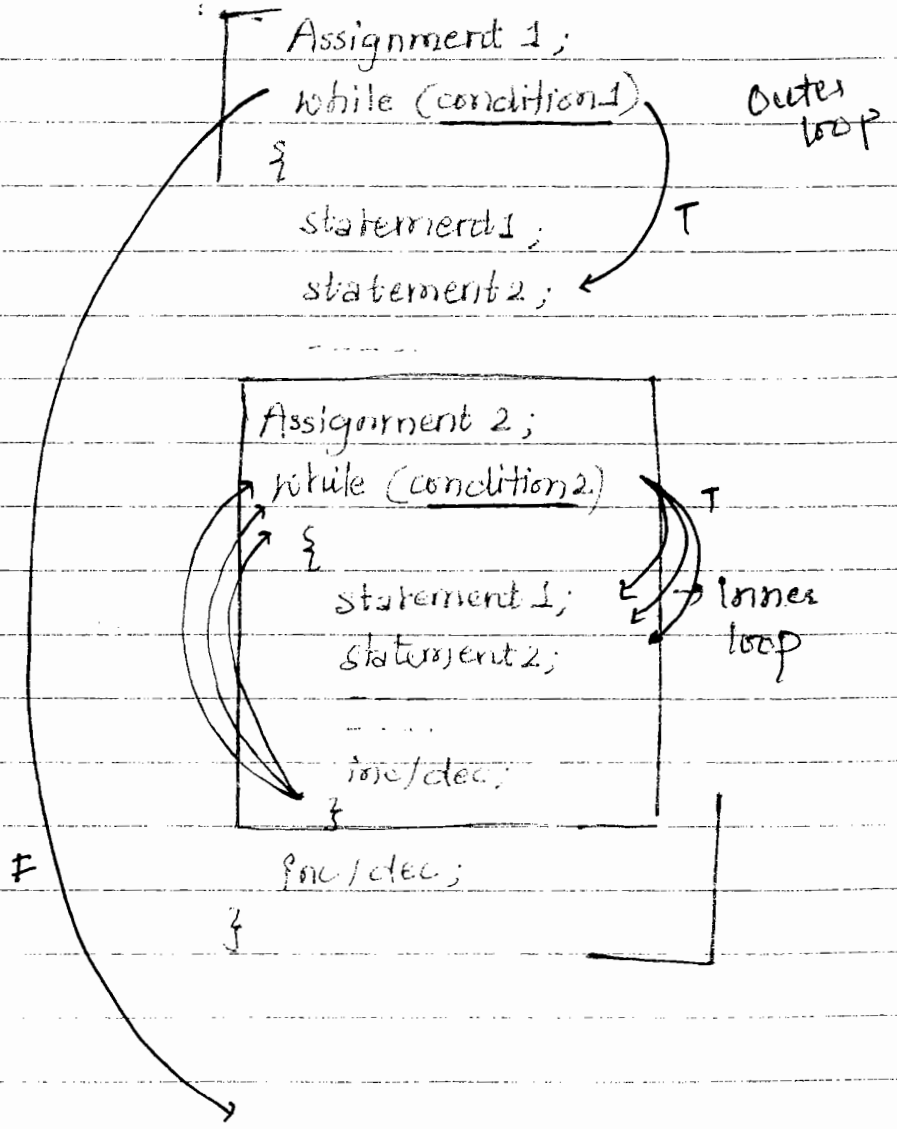
Auto fill

DELTA Pg No.
Date / /

* NESTED LOOPS

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

Syntax :



DELTA Pg No.
Date / /

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

Task

Prog:

Enter 2 values : 2 5

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

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

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

i: 1, 2, 3, 4, ..., 10

2 * 1 = 2 ... 3 ...

2 x 2 = 4

n: 2, 5

Task 1

Enter a value : 6

1

1 2

1 2 3

1 2 3 4 → outer loop

1 2 3 4 5

1 2 3 4 5 6 → inner loop

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

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

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

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

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

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

Task2:

Enter a value : 6

Decrement order

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

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

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

Task3:

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

Task4:

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

Task5:

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

DELTA Pg No.
Date / /

Task6: Enter a value: 6

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

Task7: Enter a value: 6

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

Task8: Enter a value: 6

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

Task9: Enter a value: 8

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

Task 10: Enter a value: 6

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

Task 11: Enter a value: 6

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

Task 12 Enter a value: 6

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

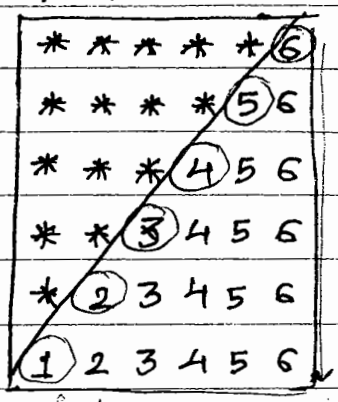
Task 13 Enter a value: 5

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

15/6/2015

DELTA Pg No. Date / /

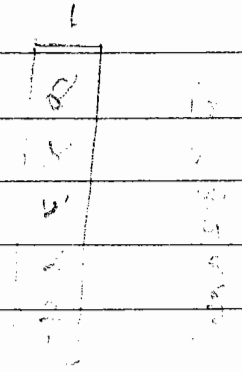
Task 14: Enter a value : 6



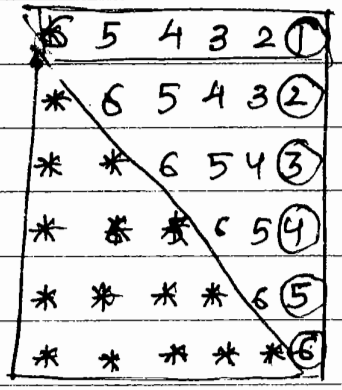
s	i
5	6
4	5
3	4
2	3
1	2
	1

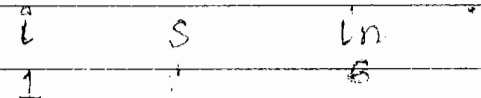
```
void main()
{
  int n, i, in, s;
  clrscr();
  printf("Enter a value : ");
  scanf("%d", &n);
  i=n;
 while (i>=1)
  {
    printf("\n");
    s=1;                    // s=i-1;
    while (s<=i-1)          // while (s>=1)
  {
```

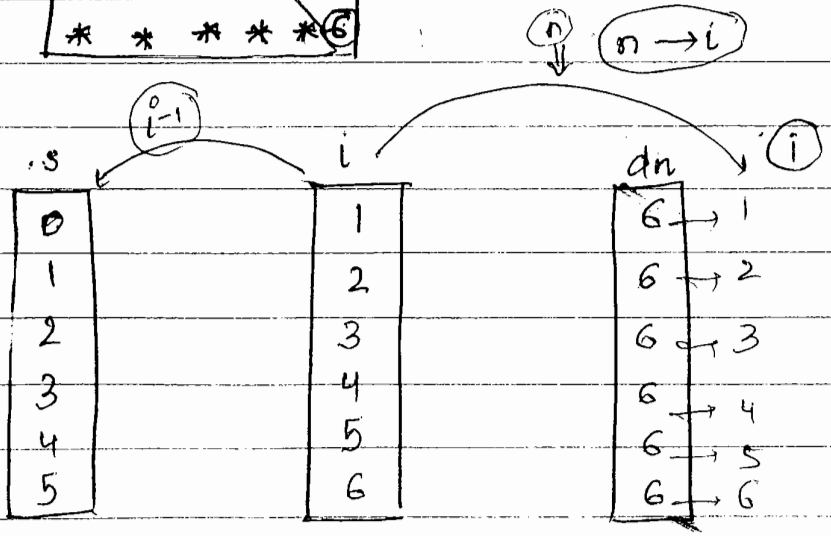
```
while (s <= i-1)
    printf ("*");
    s=s+1;          //s=s-1;
}
in = i;
while (in <= n)
{
    printf ("%d", in);
    in = in+1
}
i = i-1;
}
getch();
}
```



Task 15: Enter a value : 6



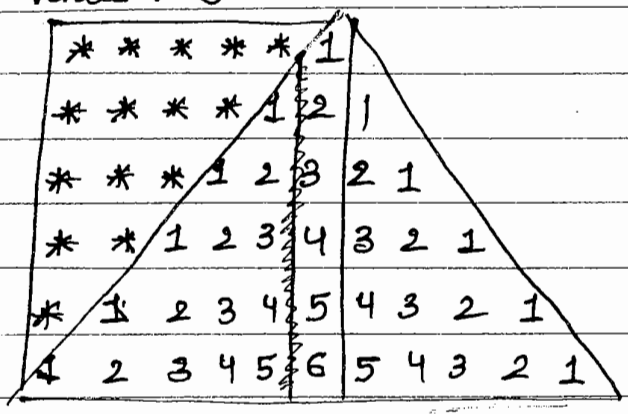


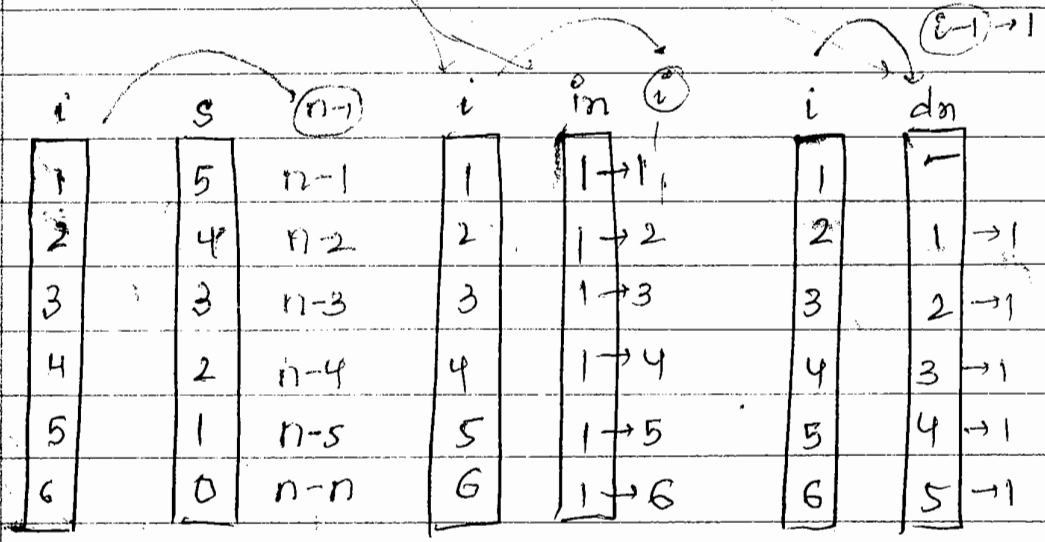


```
void main()
{
 int i, n, dn, s;
 clrscr();
 printf("Enter a value: ");
 scanf("%d", &n);
 i=1;
 while (i<=n)
  {
    printf("\n");
    s=i-1;
    while (s>=1)
    {
      printf("*");
      s=s-1;
    }
 dn=n;
 while (dn>=i)
  {
    printf("%d", dn);
     dn = dn - 1;
    }
     i = i+1;
  }
 getch();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

DELTA | Pg No. | Date / /

Task 16. Enter a value : 6





Tas

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

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

Task17:

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

DELTA Pg No.
Date / /

Task 18: Enter a value : 6

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

Task 19: Enter a value : 6

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

Task 20:

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

Task21:

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

Task22: Enter a - value : 6

Task 22:-

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

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

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

Unary Arithmetic Operators

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

DELTA Pg No.
Date / /

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

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

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

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

i.e `b = a;`

O/P :- a = 2 b = 2

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

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

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

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

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

O/P :- a = 0 b = 0

DELTA | Pg No. | Date / /

Syntax 4:- `b = a--` post decrement

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

O/P :- a = 0 b = 1

Order of priority :-

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

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

→

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

O/P = 9

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

O/P = 9

DELTA | Pg No.
Date / /

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

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

O/P :- 12

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

$a = a + a + a;$

a

1 2 3 4 ⇒ a=4

Pre → Add → Assignment

firstly evaluate all the pre operators then substitute the value.

Thus, a = 12

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

Pre → Add → Assgn → Post

a

5 4 3

$a = a + a + a$

$3 + 3 + 3$

$a = 9$

a=8

a

10

DELTA Pg No.
Date / /

dtil
ble.

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

a = a + a + a
= 2 + 2 + 2
a = 6 after post a = 8

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

a = a++ + ++b;
a = a + b ✓
= 50 + 51;
= 101
after post ⇒ a = 102

a: 102
b: 157

b = a + b ✓
b = 103 + 51
b = 154 ⟶ after post = 155

DELTA | Pg No.
Date / /

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

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

af

$a = a + b$

$a = 5 + 4$

$a = 9 - 1 \Rightarrow$ [a = 8] ↘ Then

[b = 4]

$b = a + b$

$b = 7 + 4$

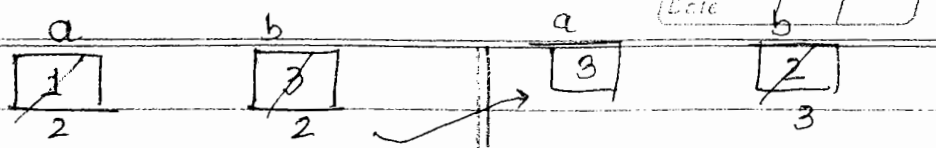
$b = 11$

O/P:- [a = 7] [b = 10] ← [b = 10]

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

Fo

DELTA Pg No.
Date



$a = a - b + a$

$2 - 2 + 2$

$a = 2$

after post $\boxed{a = 3}$

$b = b - a + b$

$b = 3 - 3 + 3$

$b = 3$

After that $\boxed{b = 4}$

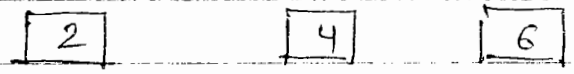
$a = 3$ but $a--$

$\boxed{a = 2}$

$\boxed{b = 4}$

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

For a



$a = a + b - c$

$a = 3 + 4 - 5$

$\boxed{a = 2}$

DELTA Pg No.
Date / /

For b :-

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

$b = a - b + c$

$= 2 - 4 + 5$

$\boxed{b = 3}$

For c :-

a	b	c
3	3	4

$c = a + b - c$

$= 2 + 4 - 4$

$\boxed{c = 2}$

DELTA | Pg No. | Date / /

II model

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

stack (LIFO) approach

passes this way.

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

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

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

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

pre (+1) post (+1) remains same

O/P :- 6 4 4

→

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

pre-subtract two.

O/P :-

```
3 3 5
1
```

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

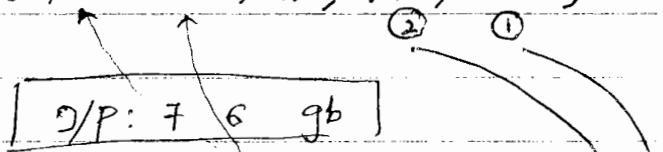
O/P :- 11 10 6

we can assign values inside printf

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

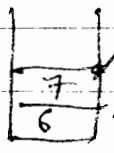
O/P :- 2 1 4

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



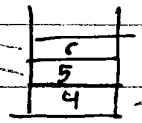
O/P: 7 6 gb

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



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

O/P :- 6 5

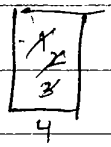


4 remains in stack.

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

O/P : 4 24 4 64

a



int b = ++a * ++a * ++a ← initialisation works acc. to stack

2 * 3 * 4

24

a = 4 b = 24

In stack DMA post both are same.

d = ++c * ++c * ++c ← assignment Works acc. to register

d = c * c * c

4 * 4 * 4

64

a = 4 d = 64

DELTA Pg No.
Date / /

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

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

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

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

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

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

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

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

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

}

O/P: 4 8 4 9

→ void main()

{

int a;

a = 1;

printf ("\n%d", ++a * ++a * ++a);

a = 1;

printf ("\n%d", a++ * ++a * a++);

a = 1;

printf ("\n%d", ++a * a++ * ++a);

a = 1

printf ("\n%d", a++ * a++ * a++);

a = 1

printf ("\n%d", a++ * a++ * ++a);

a = 1

printf ("\n%d", ++a * a++ * a++);

}

O/P :

// 2 * 3 * 4 = 24

// 1 * 3 * 3 = 9

// 2 * 2 * 4 = 16

// 1 * 2 * 3 = 6

// 1 * 2 * 4 = 8

// 2 * 2 * 3 = 12

d);

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

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

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

O/P : 12 4
2 6
7 8

DELTA | Pg No.
Date / /

INTEGER

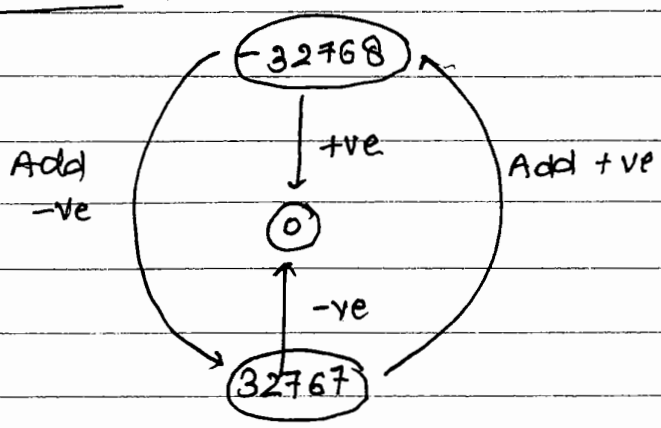
→ On doc based compiler, size of ~~com~~ integer is 2 bytes and range from -32768 to +32767

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

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

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

int i; / short i;



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

→ V

DELTA PG No.
Date / /

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

O/P : a = -127 b = 200

200 * 200
40000 -32768
32767 (n-1) + 7233
7233 -25536

a = 200 * 200 / 200
= -25536 / 200
= $\frac{-25536}{2} \times 10^{-2}$
= -12768×10^{-2}
= -127.68
a = -127

b = 200 * 200 * 200
1 * 200

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

DELTA Pg No.
Date / /

60000 → -32768

(n) 32767 → 27233 (n-1) → 27233 → -5536

$a = 360 * 200 / 300;$

$= -5536 / 300$

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

$= -1845$

$a = -18$

$b = 360 / 200 * 300;$

$= 1 * 300$

$b = 300$

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

O/P: Welcome -32768

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

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

O/P : Welcome 32767

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

O/P: Hello -32768

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

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

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

DELTA | Pg No.
Date / /

a = b++<=3

1 <= 3

∴ b = 2

a = 1 and b = 2

Then 2 <= 3 ∴ **b = 3**

a = 1 and b = 3

Then 3 <= 3 ∴ **b = 4**

a = 1 and b = 4

Then 4 <= 3 ∴ **b = 5**

and **a = 0**

a = 10 **b = 15**

→

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

O/P: 1 6

1 7

1 8

0 9

a = 10 **b = 19**

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

dummy statement

O/P: a=10

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

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

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

dummy statement

O/P: 5 10 5

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

dummy condition

it will not consider this condition.

O/P: ABC NIT

DELTA Pg No.
Date / /

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

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

O/P: ABCNITD

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

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

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

DELTA Pg No.
Date / /

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

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

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

scope is close here

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

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

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

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

dummy loop

O/P: No output with inf loop

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

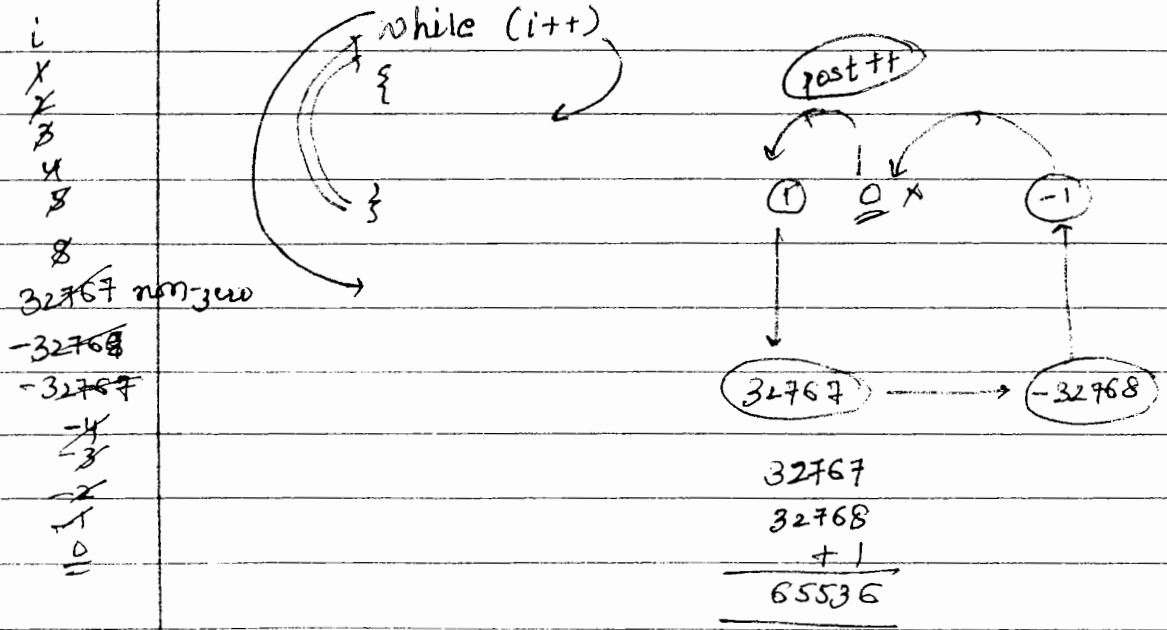
DELTA Pg No.
Date / /

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

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

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

O/P: ~~i=1~~ (65536)



→ void main()

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

dummy loop

O/P: i = 32767

(-1)

while (i--> -32768)
{
}

-32768 > -32768

+32767

→ void main()

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

post

O/P: i = -1 (65536)

→ Void main()

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

O/P: i = -32768

DELTA Pg No.
Date / /

break & Continue

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

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

O/P: 1 2 3 4

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

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

20
18
16
14
12

O/P : 20 18 16 14 12

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

O/P:- 1 3 5 7 9

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

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

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

O/P: 10

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

5

dummy condition

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

DELTA PG No.
Date / /

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

O/P: 5

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

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

O/P: Error misplaced break

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

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

DELTA | Pg No.
Date / /

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

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

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

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

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

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

O/P:- 1 3 5 5 5 5 --- inf. loop

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

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

O/P: No output with inf loop

DELTA Pg No.
Date / /

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

dummy loop

O/P: Error misplaced continue

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

Working with 'for' loop :-

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

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

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

O/P :- 2 4 6 8 10

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

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

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

- When for loop is not having the body

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

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

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

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

→ Void main()

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

a	b
~~9~~	~~9~~
~~8~~	~~8~~
1	~~7~~
1	~~6~~
1	~~5~~
0	~~4~~
a=10	~~b~~ = 14

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

* Perfect Number

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

Ex- 6 → 1+2+3 → (6)

28 → 1+2+4+7+14 → (28)

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

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

→ void main()

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

n	n/2
28	14

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

To get list of Perfect Numbers.

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

O/P : Enter a value : 10000

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

Armstrong No.

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

There are 4 armstrong numbers

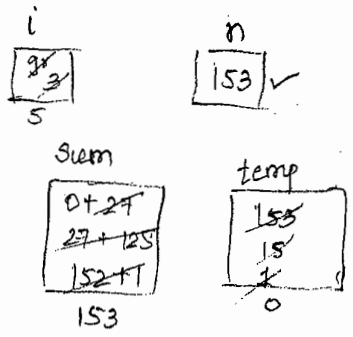
→ 153 → $1^3 + 5^3 + 3^3$
$1 + 125 + 27$ } 153

→ 370 → $3^3 + 7^3$
$27 + 343$ } 370

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

→ 407 → $4^3 + 7^3$
$64 + 343$ } 407

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



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

How to get list of numbers

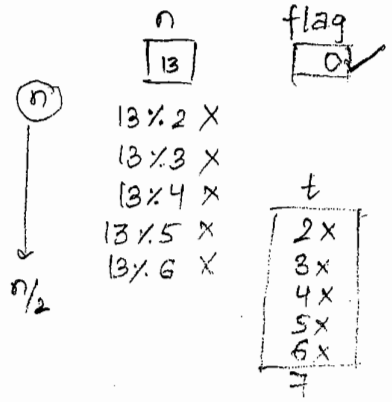
```
Void main()
{
 int n, n1, temp, i, sum=0;
 clrscr();
 printf("Enter a value: ");
 scanf("%d", &n1);
 for(n=153; n<=n1; n++)
 {
  sum=0;
  for(temp=n; temp!=0; temp/=10)
  {
   i= temp%10;
   sum+=(i*i*i);    //sum+=pow(i,3);  <math.h>
  }
  if(n==sum)
    printf("\n%d. ARMSTRONG NO.: %d", ++count,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

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

* Prime No.

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

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



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

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

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

O/P: Enter 2 values : 15 2170

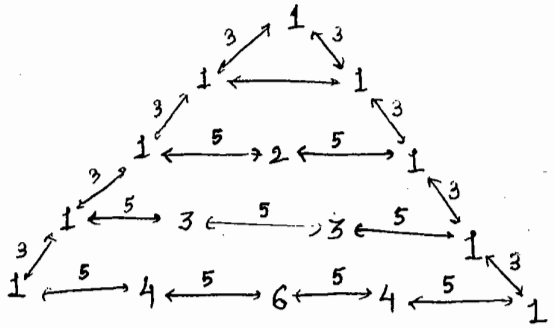
1. PRIME NO : 17
2. PRIME NO : 19
3. PRIME NO : 23

⋮

22/6

PASCAL TRIANGLE

Enter no. of rows = 5




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

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

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

r: 5

s: 41→1, 37→1, 34→1, 31→1, 20→1

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

K: 0, 1, 2, 3, 4

r: X, 2, 3, 4, 5 ~~5~~ ~~4~~ ~~3~~

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

INTERVIEW QUESTIONS

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

O/P :- Welcome
a = 7

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

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

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

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

O/P: 10 Hello 200
a = 12

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

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

L ← R
10
9
R

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

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

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

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

→ void main()

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

O/P :- Three
Two6
One5
a=5

void main()

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

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

→ void main()

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

O/P : Hai
a=3

a = 5>2 ? pf("Hai") : pf("Bye");
T 3

→ void main()

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

O/P : HiHello
a = 5

1) Hi ⇒ 2 value
2) Not operator

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

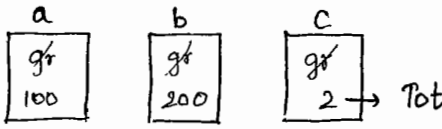
assignment operator i.e error

O/P: Error L value required

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

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

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



→ Total no. of values provided by user.

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

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

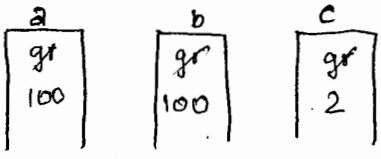
# void main()
{
int a, b, c;
a = b = 100;
c = scanf("%d %d", a, &b);

222

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

printf("\n a = %d b = %d c = %d", a, b, c);
}
// Input values are 111 222

a = 700 b = 222 c = 2



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

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

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

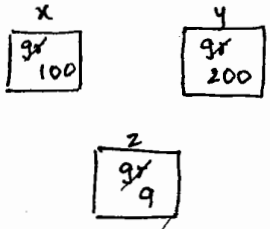
```
void main()
{
  int a,b,c;
  a=10; b=20;
  c= scanf("%d %d %d");
  printf("\n a=%d b=%d c=%d", a,b,c);
}
```

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

23/6

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

L ← R

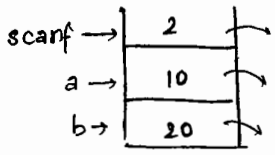


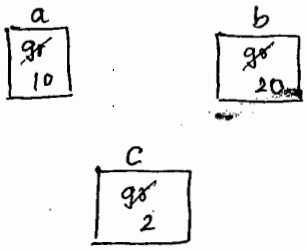
O/P:
```
Welcome 2
x=100 y=200 z=9
```

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

2 10 20

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



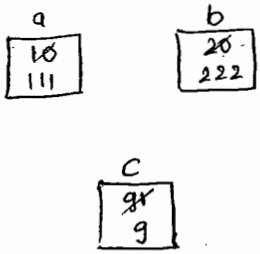


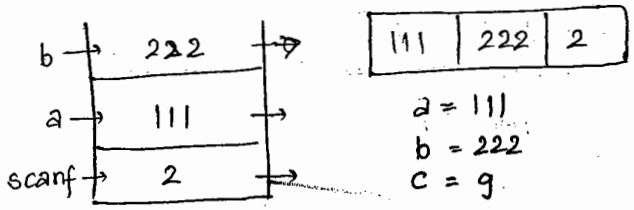
→ After passing the data into printf related stack then it is not possible to update by using scanf function.

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

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

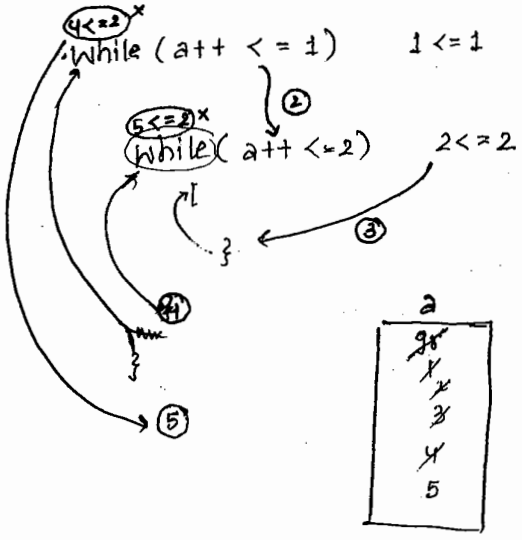
O/P :- a = 111, b = 222 c = 9





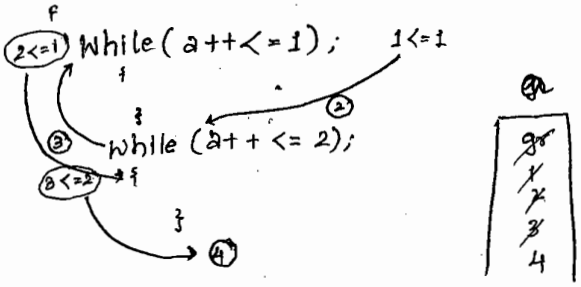

```
→ void main()
  {
     int a ;
     a = 1;
     while (a++ <= 1) ;
     while (a++<=2);
     printf ("a =%d", a);
  }
```

O/P :- a = 5



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

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

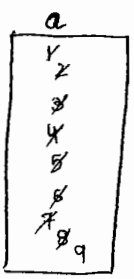


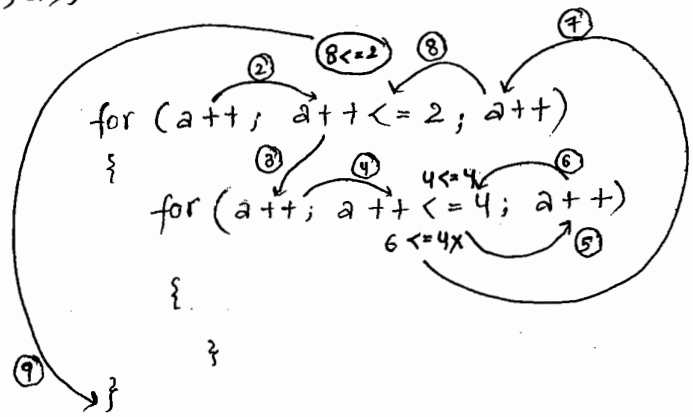
O/P: a = 4

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

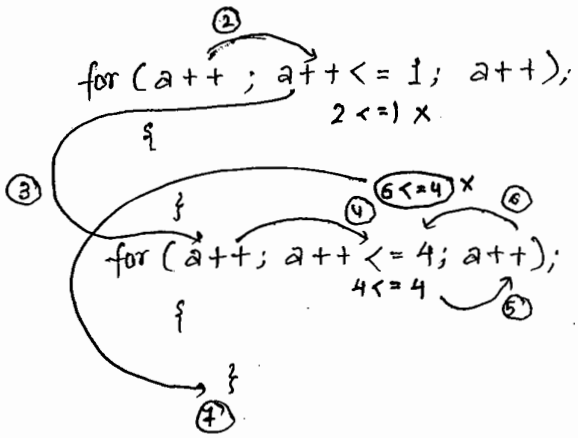
O/P: a = 9

⇒






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



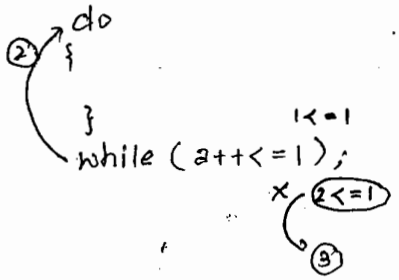
→
```
void main()
{
  int a;
  a = 1;
   do
   while (a++ <= 1);
   printf ("a = %d", a);
}
```

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

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

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

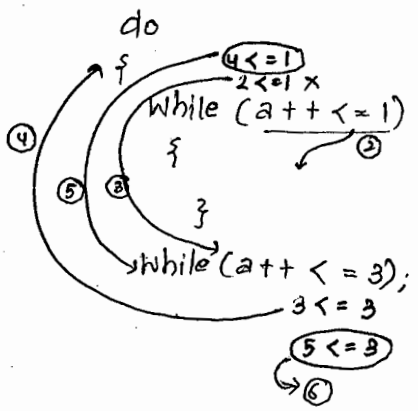
```
void main()
{
  int a;
  a = 1;
   do;
   while (a++ <= 1);
   printf ("a = %d", a);
}
```



O/P : **a = 3**

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

O/P :- **a = 6**



SWITCH (Switch);

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

→ Multiple choices can be constructed by using case Keyword.

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

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

Syntax :-

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

  default: Block-n;
}
```

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

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

O/P :- 23D

→ void main()

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

O/P :- A B

→ void main()

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

O/P :- D

→ void main

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

O/P :- B2

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

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

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

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

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

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

O/P: Apple TV

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

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

24/Jun

→ void main()

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

variable type

case 'a' :

case 'b' :

case 'c' :

O/P: Error Constant expression Required

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

→ void main()

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

O/P: A

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

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

O/P : Error, Duplicate Case

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

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

O/P: Error misplaced continue

case, default → switch body

loop body [break, continue]

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

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

O/P : Error misplaced case, default and break

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

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

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

goto

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

Syntax:

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

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

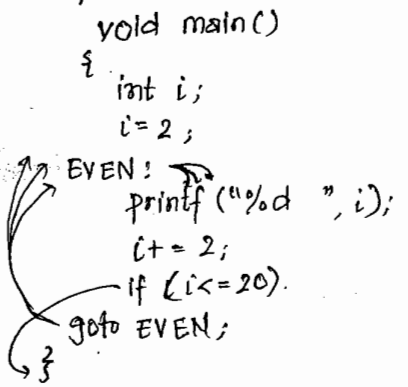
O/P : ABCD

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

O/P: ABCDXY

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

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



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

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

O/P: AxyBCxyBExyBC .... infinite loop

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

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

O/P: Error undefined label 'NIT'

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

→ switch program

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

O/P: D

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

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

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

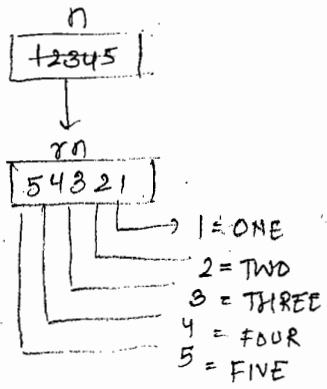
case ~~default~~ :
casedefault :
O/P : 2

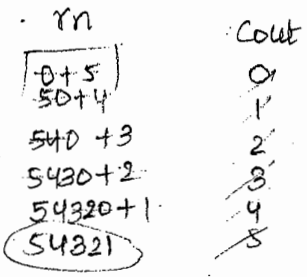
O/P : Error

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

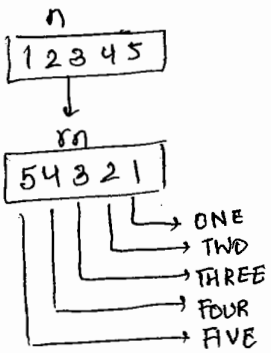
```
void main()
{
 long int n, rn;
 int count = 0;
 clscr();
 printf (" Enter a value");
 scanf (" %ld ", &n);
 while(n)
{ rn = rn*10 + n%10;
 ++count;
 n = n/10;
}
while (rn)
 { switch (rn % 10)
    {
      case 0: printf (" ZERO");
              break;
      case 1 : printf (" ONE");
               break;
      case 2 : printf ("TWO");
               break;
      case 3 : printf ("THREE");
               break;
      case 4 : printf ("FOUR");
               break;
      Case 5 : printf ("FIVE");
               break;
      Case 6 : printf ("SIX");
               break;
      Case 7 : printf ("SEVEN");
               break;
```

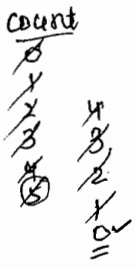
```
        case 8: printf("EIGHT");
                break;
        case 9: printf("NINE");
                break;
    }
    rn = rn/10;
    --count;
}
  while(count>0)
  { printf("ZERO");
    --count;
  }
  getch();
}
```

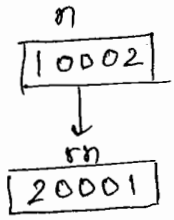


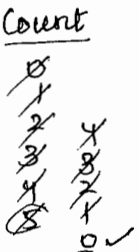


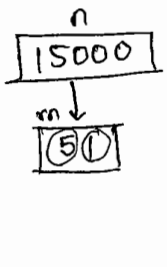
Other cases

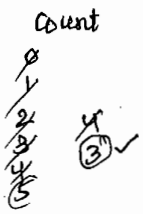


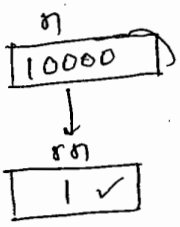


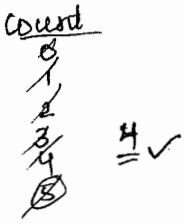












25/Jun

Program - CALENDER

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

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

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

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

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

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

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

long for storing long value of int

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

If cases like 1 2 3 4 5

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

⇒ if leap year

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

To find Weekday

* 01-01-01 ⟶ Monday

18 Jan-01 → 4 Thursday
18 % 7 = 4

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

20-Mar-01 ⇒ 2 Tuesday
31
28
20
79 % 7

final now
yy
2015
mm
06
dd
25

current 2015

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

7 35598

2015
31
28
31
30
31
25

dp → 7,35,774

Jan, Mar, May, July, Aug, Oct, Dec } 31
April, June, Sept, Nov } 30

Feb 28/29

For leap year

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

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

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

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

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

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

DATA TYPES :-

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

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

a) char
b) int and
c) float types

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

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

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

Type	Size	Range	%o	Ex :-
char	1 byte	-128 to 127	%c	'a' 'A' '#' '\n'
unsigned char	1 byte	0 to 255	~~%cd~~	
int	2 bytes	-32768 to 32767	%d	-25.5 0 -5
unsigned int	2 bytes	0 to 65,535	%u	25u 32768u
long int	4 bytes	-2,147,483,648 to +2147,483,648	%ld	45l -5l 40000l
unsigned long	4 bytes	0 to 4,294,967,295	%lu	100lu 51lu
float	4 bytes	$\pm 3.4 * 10^{\pm 38}$	%f	-35f 7.5f
double	8 bytes	$\pm 1.7 * 10^{\pm 308}$	%lf	-3.5, 7.5
long double	10 bytes	$\pm 3.4 * 10^{\pm 4932}$	%Lf	-3.5L, 7.5L

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

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

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

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

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

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

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

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

→ signed, unsigned are called signed specifiers

→ short, long are called size specifiers

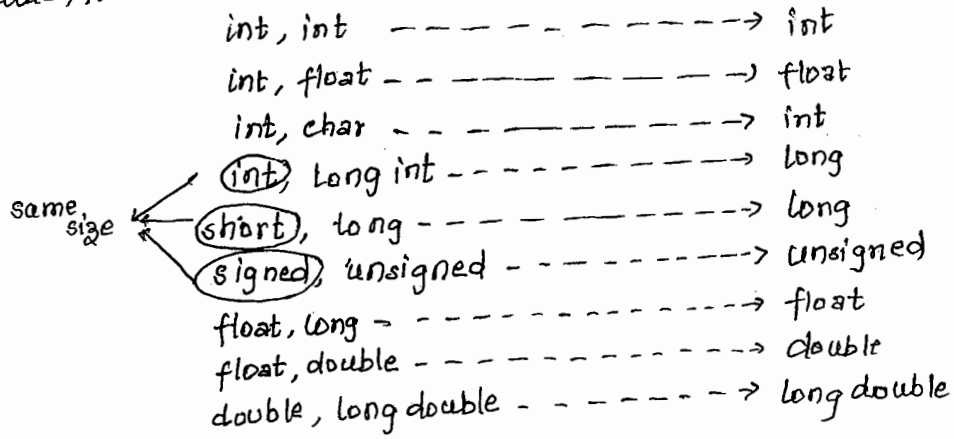
```
Void main()
 {
   int i;
   long int l;
   float f;
   i = 32767+1;
   l = 32767+1;
   f = 32767+1;
   printf("%d %ld %f", i, l, f);
}
```

O/P:- -32768 -32768 -32768.000000

Automatic Operations on Data Type.

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

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



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

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

O/P:- -32768 32768 32768.000000

Type Casting

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

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

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

1) Implicit type casting
2) Explicit type Casting

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

→ Implicit Type Casting is under control of compiler.

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

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

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

→ Explicit Type Casting is under control of programmer.

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

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

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

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

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

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

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

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

O/P :- Welcome

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

.0
.5
} float == double

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

O/P :- Welcome

i → signed value
u → unsigned value

if (i > u)
signed > unsigned

-1 (signed)

65535 (unsigned) > 10

NUMBER SYSTEM

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

Number systems are classified into 4 types.

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

1) Decimal Number System -

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

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

2) Octal Number System

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

Eg :- 012, 034, 0765, 01457

3) Hexadecimal Number System

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

Eg :- 0X1234, 0XA1B2, 0XFEAB

	DECIMAL (10) 0-9 %d	OCTAL (8) 0-7 %o	HEXADECIMAL (16) 0-9ABCDEF %x
①	100	0144 8 | 100 8 | 12 - 4 1 - 4	0X64 16 | 100 6 - 4
②	127	0177 8 | 127 8 | 15 - 7 1 - 7	0X7F 16 | 127 7 - 15 (F)
③	32767	077777	0X7FFF
④	65535	0177777	0XFFFF

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

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

BINARY NUMBER SYSTEM

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

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

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

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

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

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

No. of bits	No. of Combination	max value	max value representation
1	2 (2^1)	1 (2^1-1)	0001
2	4 (2^2)	3 (2^2-1)	11
3	8 (2^3)	7 (2^3-1)	111
4	16 (2^4)	15 (2^4-1)	1111
5	32 (2^5)	31 (2^5-1)	001 1 1111
6	64 (2^6)	63 (2^6-1)	111 1111
7	128 (2^7)	127 (2^7-1)	1111 1111
8	256 (2^8)	255/-1 (2^8-1)	1 1111 1111
16	65536 (2^{16})	65535/-1 ($2^{16}-1$)	1111 1111 1111 1111

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

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

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

→ left side → most significant bit is called Sign bit

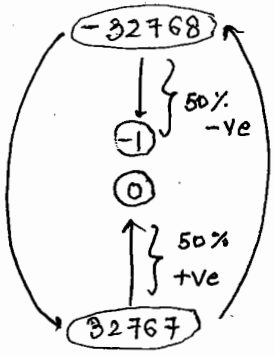
→ Always sign bit will decides the written value of binary representation i.e +ve or -ve.

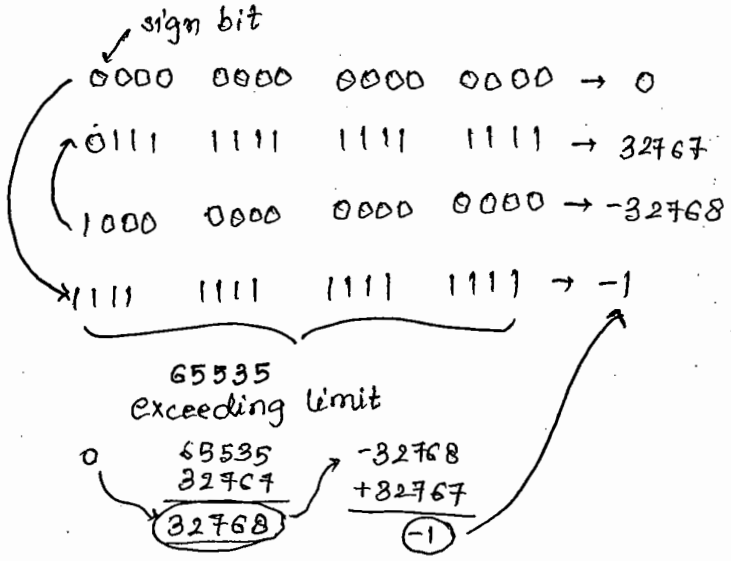
→ If sign bit is 0 and remaining all bits are zero then it gives min. value of +ve sign i.e zero

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

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

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





32767 → 0111 1111 1111 1111
\+ 1 → 0000 0000 0000 0001
1000 0000 0000 0000 ⇒ -32768

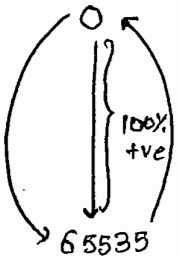
unsigned int u ;

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

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

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

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



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

char ch ;

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

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

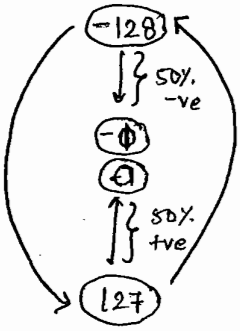
→ Among those all representations 50% provides +ve data & remaining 50% provides +ve data

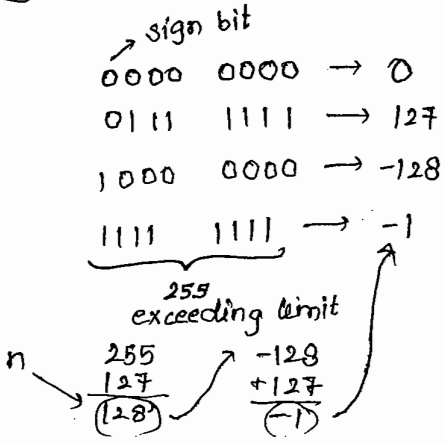
→ If sign bit is 0, and remaining all bits are zeros then it gives min. value of +ve sign i.e (0)

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

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

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





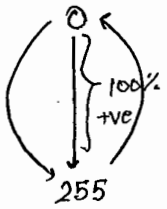
unsigned char ch ;

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

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

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

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



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

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

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

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

8|65535
8|8191 → First count in binary forms and then solve

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

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

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

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

076543
~~0111 0110 0101 0100~~
111 110 101 100 011 octal

0111 1101 0110 0011 Hexa
7 D 6 3
2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^{9} 2^{8} 2^{7} 2^{6} 2^{5} 2^{4} 2^{3} 2^{2} 2^{1} 2^{0}

%x → 7D63
%d → 32099
%u → 32099

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

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

0XFABC

F → 1111
A → 1010
B → 1011
C → 1100

} 16 bit

1 7 5 2 7 4

1111 1010 1011 1100

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

64188 − 32767 = 31421 (n)

31420 − 1848 (n−1)

-32768

%u = 61488
%d = -1348
%o = 175274

BITWISE OPERATORS

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

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

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

⟶ int a;

a) a = ~5

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

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

65535 − 5 = 65530 (n)

65530 − 32767 = 32763 (n−1)

-32768 + 32762 = -6

b) a = ~239

Sign bit approach

239 → 0000 0000 1110 1111

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

sign bit

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

Calculation approach

65535 → 1111 1111 1111 1111

~239 → 1111 1111 0001 0000

65535 − 239 = 65296

65296 / 32767 → 32529 (n-1)

−32768 → 32528 − 240

(c) a = ~0; −1

0 ---> 0000 0000 0000 0000

~0 ---> 1111 1111 1111 1111 (−1)

d) a = ~32767; −32768

32768 ------> 0111 1111 1111 1111

~32767 ------> 1000 0000 0000 0000 −32768

e) a = ~−19; 18

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

−19 → 1111 1111 1110 1101 }

~−19 → 0000 0000 0001 0010 (18)

f) a = ~−128, 127

−1 → 1111 1111 1111 1111

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

~−128 → 0000 0000 0111 1111 (127)

---

unsigned int u;

1. u = ~15 65520

15 → 0000 0000 0000 1111

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

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

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

Ans -29 (xactly negation value).

```
29 ----------> 0000 0000 0001 1101
~29 ---------> 1111 1111 1110 0010
+ 1 ---------> 0000 0000 0000 0001
---------------------------------
               1111 1111 1110 0011   (-29)(-1-16-8-4)
```

LEFT-SHIFT OPERATORS.

1) In Left-shift Operators, towards from left side n no. of bits required to draw & towards from right side (empty) places required to be filled with (zeros).

2) When we are dropping the bits towards from left side den all 1's will be shifted towards left side by n places, so automatically value is increased.

int a;

1. a = 10 << 1;

```
    10 -> 0000 0000 0000 1010
10<<1 -> 0000 0000 0001 0100   20(16+4)
```

2. a = 5<<2;

```
    5 -> 0000 0000 0000 0101
5<<2 -> 0000 0000 0001 0100   20 (16+4)
```

3. a = 20<<3;

```
    20 -> 0000 0000 0001 0100
20<<3 -> 0000 0000 1010 0000   160 (128+32)
```

4. a = 15<<4;

```
    15 -> 0000 0000 0000 1111
15<<4 -> 0000 0000 1111 0000   240 (128+64+32+16)
```

5. a = 1<<1 2

6. a = 100<<1; 200

7. a = 500<<1; 1000

8. a = 1500<<1; 3000

9. a = 1<<15; -32768

```
    1 -> 0000 0000 0000 0001
1<<15 -> 1000 0000 0000 0000   (-32768)
```

10. a = 32767 << 15 → −32768

32767 → 0111 1111 1111 1111

32767<<15 →

11. a = 1 << 16; → 0

12. a = 1234 << 16; → 0

13. a = 32767 << 16; → 0

* For any +ve integer, left shift (16) makes the value as zero bcoz +ve data representation is available in 16 bit towards from left side.

int a;

1. a = -10 << 1;
   ~9 << 1;
   -1 → 1111 1111 1111 1111
   -10 → 1111 1111 1111 0110

2. a = -10<<2;
   ~9 <<2;
   -10<<1 → 1111 1111 1110 1100 → -20 (-1-16-2-1)
   -10<<2 → 1111 1111 1101 1000 → -40 (-1-32-4-2-1)

3. a = -10<<3;
   ~9 <<3;
   -10<<3 → 1111 1111 1011 0000 → -80 (-1-64-8-4-2-1)

Value = x<<n; → valid upto n = 14.
x * 2^n

RIGHT-SHIFT OPERATOR

1) In right-shift Operator, towards from right side n no. of bits required to draw & towards from left side empty places required to be filled with zeros for +ve no.s only.
2) In right-shift operator, for -ve no., empty places required to be filled with 1's so sign bit will not be modified & we will get -ve values only.
3) When we are dropping the bits towards from right side then all 1's will be shifted towards right side by n places, so automatically value is decreased. (for +ve only).

int a;

1. a = 10>>1;
   10 → 0000 0000 0000 1010
   10>>1 → 0000 0000 0000 0101 → (5)
2. a = 15>>2;
   15 → 0000 0000 0000 1111
   15>>2 → 0000 0000 0000 0011 → (3)
3. a = 20>>3;
   20 → 0000 0000 0001 0100
   20>>3 → 0000 0000 0000 0010 → (2)
4. a = 25>>4;

25 → 0000 0000 0001 1001

25 >> 4 → 0000 0000 0000 0001 (1)

5. a = 100 >> 1; 50

6. a = 1000 >> 1; 500

7. a = 1234 >> 1; 617

8. a = 1 >> 15; 0

9. a = 32767 >> 15; 0

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

int a;

Value = x >> n;
= $x/2^n$

1. a = -5 >> 1;

-4 >> 1;

-1 → 1111 1111 1111 1111

-5 → 1111 1111 1111 1011

-5 >> 1 → 1111 1111 1111 1101 · -3 (-1-2)

-5 >> 2 → 1111 1111 1111 1110 -2 (-1-1)

-5 >> 3 → 1111 1111 1111 1111 -1

Remaing Operators (BITWISE AND, OR, XOR)

a	b	a & b	a | b	a ∧ b
1	1	1	1	0
0	1	0	1	1
1	0	0	1	1
0	0	0	0	0

1. printf ("%d %d %d", 2&3, 2|3, 2^3);

O/P :- 2 3 1

2 → 0000 0000 0000 0010

3 → 0000 0000 0000 0011

2 & 3 → 0000 0000 0000 0010

2 | 3 → 0000 0000 0000 0011

2 ^ 3 → 0000 0000 0000 0001

2. printf ("%d %d %d", 25 & 30, 25 | 30, 25 ^ 30);

```
25 → 0000 0000 0001 1001
30 → 0000 0000 0001 1110
-------------------------
25 & 30 → 0000 0000 0001 1000   (24)
25 | 30 → 0000 0000 0001 1111   (31)
25 ^ 30 → 0000 0000 0000 0111   (7)
```

3. printf ("%d %d %d", 0 & -1, 0 | -1, 0 ^ -1)

```
0 → 0000 0000 0000 0000
-1 → 1111 1111 1111 1111
-------------------------
0 & -1 → 0000 0000 0000 0000   (0)
0 | -1 → 1111 1111 1111 1111   (-1)
0 ^ -1 → 1111 1111 1111 1111   (-1)
```

OPERATORS IN 'C'

In 'C' prog. lang., We are having 44 operators, acc. to the priority those operators are- (45 operators but , is not opr, it is seperator so 44)

1. ( ), [ ], →, . — 1st category operators
   (Paranthesis) (subscript) (Pointer to member) (struct to member)

2. +, -, ++, --, !, ~, *, &, sizeof, (type) → 2nd category
   (* ↓ 3rd direction) (& address of)

3. *, /, % — 3rd category

4. +, - — 4th category

5. << (L. shift), >> (R. shift) — 5th category

6. <, >, <=, >= — 6th category

7. !=, == — 7th category

8. & (Bitwise AND) — 8th category

9. ^ (B. XOR)

10. | (B. OR)

11. &&

12. ||

13. ?:

14. =, +=, -=, *=, /=, %=
    &=, |=, ^=, <<=, >>=

15. ,

Size Of

→ sizeof is a operator cum Keyword in 'C' lang.

→ By using sizeof operator, we can find size of an I/P argument.

→ Sizeof operator is a unary operator which always requires 1 argument of type, of datatype & returns unsigned integer value which always greater than zero.

```
void main()
{ int i;
  char ch;
  float f;
  clrscr();
  printf ("\nsize of int : %u", sizeof(i));
  printf ("\nsize of char : %u", sizeof ch);
  printf ("\nsize of float : %u", size of f);
```

→ with paranthesis
→ without paranthesis
} Both way, u can call.

O/P :- size of int : 2
size of char : 1
size of float : 4

sizeof (int) ⟶ 2
sizeof (short) ⟶ 2
sizeof (signed) ⟶ 2
sizeof (unsigned) ⟶ 2
sizeof (long) ⟶ 4
sizeof (25) ⟶ 2
size of (25.0) ⟶ 2
size_of (25.1) ⟶ 4

---

sizeof (char) ⟶ 1
sizeof ('A') ⟶ 2

```
char ch;
ch = 'A';
```

sizeof(ch) ⟶ 1

- sizeof character or character variable is 1 byte but sizeof character constant is 2 bytes

- By default character constant returns an integer value i.e Ascii value of a character constant that's why size of (Char constant) is (2 bytes).

```
sizeof (float)          ⟶ 4
sizeof (double)         ⟶ 8
sizeof (long double)    ⟶ 10B
sizeof (short float)    ⟶ Error
sizeof (signed double)  ⟶ Error
sizeof (12.8)           ⟶ 8  // double
sizeof (12.8f)          ⟶ 4  // float
sizeof (12.8L)          ⟶ 10 // long double
sizeof (12.0)           ⟶ 8
sizeof (12.5)           ⟶ 8
```

```
void main()
{ int i ;
  i = 10;
  printf ("\n size 1 : %u ", sizeof (i));
  printf ("\n size 2 : %u ", size of (i/5.0));
  printf ("\n size3 : %u ", sizeof (i * 51));
  printf ("\n size4 : %u ", sizeof (i/2.0f));
  printf ("\n size5 : %u", sizeof (i = i * 10));
  printf ("\n i = %d", i);
}
```

O/P :

```
size 1 : 2    // int
size 2 : 8    // int/double ⟶ double
size 3 : 8    // int * long ⟶ long
size 4 : 4    // int/float ⟶ float
size 5 : 2    // int * int ⟶ int
 i = 10
```

- In sizeof Operator any kind of expressions can be evaluated except assignment.
- In sizeof Operator directly or indirectly, assignment related expressions are not evaluated

```
void main()
{
  int a ;
  a = 10;
  printf ("\n size of a : %u", size of (++a));     // a = a+1;
```

```
    printf ("\n a = %d", a);
}
```

O/P :- size of a : 2
a = 10

→
```
void main (){
    int i;
    i = -1;
    if (i > sizeof(i))    // Here signed is compared with unsigned
      printf (" Welcome");
    else printf ("Hello");
}
```

O/P :- Welcome

if (i > sizeof(i))

-1 (signed) → 65535 (unsigned) > 2

2 bytes (unsigned)

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

O/P :- 2 > 1 || right side will not be evaluated. | 2 1 1 |

- In logical OR operator, if anyone of the expression is true then return value is 1
- In logical OR operator, when left side expression is true right sigle will not be evaluated.

→
```
void main()
{ int a,b,c;
  a = 1; b = 2;
  c = --a > 1 || ++b > 2;
  printf ("%d %d %d", a,b,c);
}
```

O/P :- | 0 3 1 |

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

O/P :- 2 1 3 1

In logical 'OR' operator, evaluation required to stop when expression becomes true

→ void main()
```
{
  int a,b,c;
  a=b=1;
  c= ++a<1 && ++b>1;
  printf("%d %d %d", a,b,c);
}
```
O/P :- 2 1 0

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

→ Void main()
```
{
  int a,b,c;
  a=1; b=2;  ⟶  c= ++a>>1 && --b<2;
  printf("%d %d %d", a,b,c);
}
```
O/P :- 2 1 1

→ void main()
```
{
  int a,b,c,d;
  a=1; b=2; c=3;
  d= ++a>1 && --b>2 && ++c>3;   2>1 && 1>2
  printf("%d %d %d %d", a,b,c,d);
}
```
O/P :- 2 1 3 0

- In logical 'AND' operator, required to stop the evaluation when the expression became false.

OR, AND Combinations

→ void main()
```
{
  int a,b,c,d;
  a=1; b=2; c=3;
  d= ++a>1 || --b<2 && ++c>3;
  printf("%d %d %d %d", a,b,c,d);
}
```
True ×
2>1 || (1<2)

O/P: 2 2 3 1

(1) (exp1) || exp2 && exp3
When true ↓ this will not be evaluated

2) exp1 (False) || exp2 (False) && exp3 ; ✗

3) exp1 || exp2 (True) && exp3 ; ✓

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

O/P :- 2 2 4 1

1) exp1 (F) && exp2 || exp3;

2) exp1 (T) && exp2 (T) || exp3 ; ✗

3) exp1 (T) && exp2 || exp3 ;
(exp2 → F)

2/7/2015.

POINTERS...

→ Pointer is a derived data type in 'C' which is constructed from fundamental datatype of 'C' language

→ Pointer is a variable which holds address of another variable.

Advantages of Pointer

1) **Data Access**

- 'C' programming language is a procedure-oriented lang. i.e applications are developed by using functions.
- From one func^n to another func^n, when we required to access the data, pointer is required

2) **Memory Management**

- By using pointers only, dynamic m/m allocation is possible.

3) **Database/Datastructures**

- Any kind of data structures required to develop by using pointers only, if datastructures are not available then database is not possible to create.

4) **Performance**

- By using pointers, we can increase the execution speed of the program.

→ When we are working with pointers, we required to use following operators:-

1. & :- address of operator
2. * :- In direction operator or dereference operator or object at location or value at address.

→ Address of operator always returns base address of variable.
→ Starting cell address of any variable is called base address.
→ Indirection operator always returns value of a address; i.e what address we are passing, from that address corresponding value will be retrieved.

Syntax to create a pointer

```
Datatype *ptrName;
```

→ Acc. to syntax, Indirection operator must be required b/w datatype & ptrName.
→ Space is not mandatory to place in declaration of pointer variable.

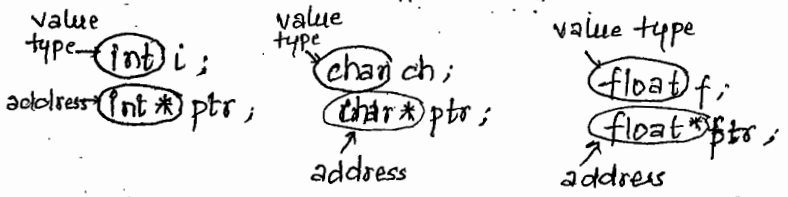
Syntax to initialise a pointer

```
Datatype variable;
Datatype * ptr = &variable;
```

```
⇒ void main()
  {
    int i;
    int * ptr;
  }
```

→ i is a variable of type ~~and~~ integer, it is a value type variable which holds an integer value

→ ptr is a variable of type an int*, it is a address type variable which holds an integer variable address



Memory Management in TC3.0 (Memory Architecture)

→ TC3.0 designed on 8086 Architecture

→ The platform of 8086 Architecture is dos 16 bit os

→ On this architecture, when we are designing a 'c' application, it occupies 1 Mb m/m only.

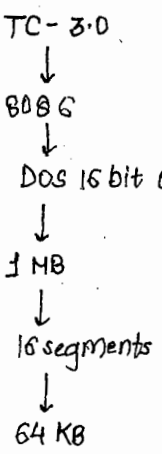
→ This complete 1 MB data is divided into 16 equivalent parts called segments

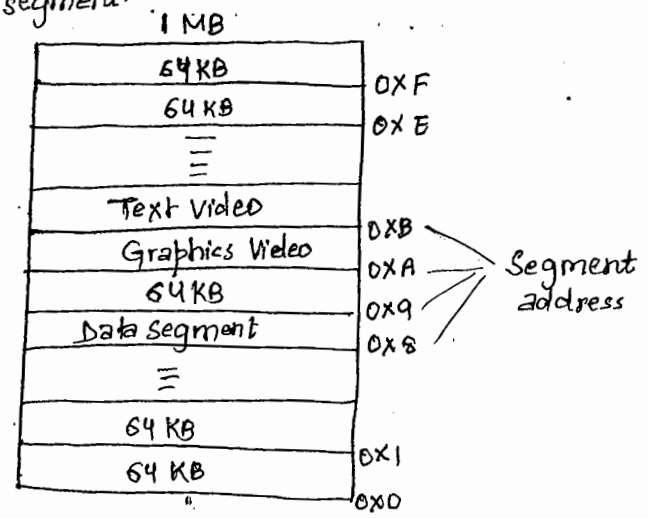
→ Each and every segment having a unique identification value called Segment address which starts from 0x0 and ends with 0xF.

→ Among those all segments, 9th segment is called data segment i.e 0x8, 11th segment is called Graphics Video Segment & next segment is called Text Video Segment

→ whenever we are creating any type of variable, then that variable will occupy the m/m within the data segment only.

→ Graphics related resources are available in Graphics Video segment & printing process will take place in text video segment.





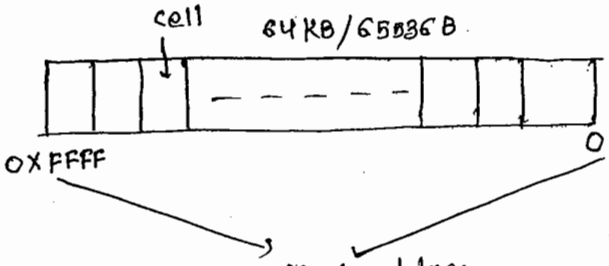
→ Each segment capacity is (64 KB) only.

→ This complete (1 KB) data is divided into small partitions called cells

→ Each cell capacity is [1 byte] only.

→ Each and every cell having a unique identification value called offset Address which starts from 0X0000 and ends with 0X FFFF

Physical add. of a variable means combination of offset Address and Segment Address.



Low order - right side

High order - left side

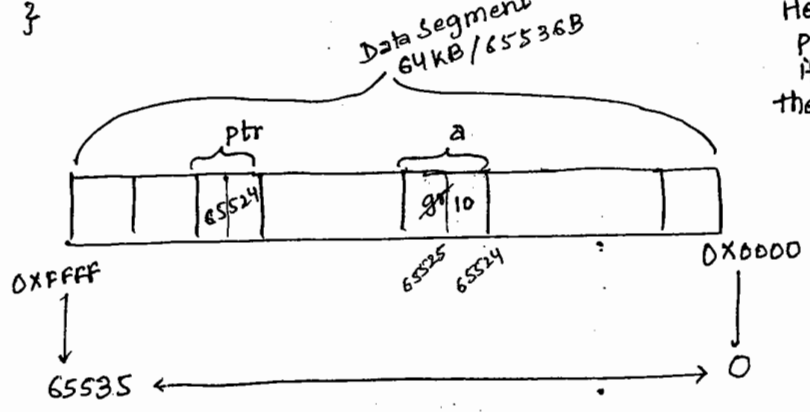
always acc. to standard

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

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

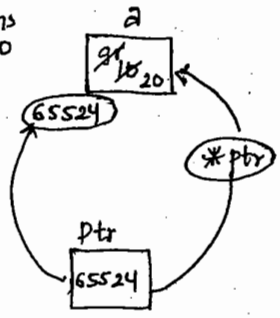
%p → will print physical address only in 16 bit not in 32 bit

O/p :-

Imaginary output	Address (H)	Address (H)
	10	10
	Address (D)	Address (D)
	20	20



Here in code
ptr = &a
if ptr contains address of a
then *ptr = 20
it contains value of 20



65535 → 1111 1111 1111 1111

65524 → 1111 1111 1111 0100
F F F 4

O/p :-

FFF4	FFF4
10	10
65524	65524
20	20

→ In implementation, when we required to print the addition of a variable then we are required to use %p, %x, %u, %lp or %lu format specifier.

→ %p, %x, %lp → will print the address in the form of (Hexadecimal.)

→ %u, %lu → will print the address in the form of (decimal.)

→ %p, %x, %u → will print 16 bit physical address i.e offset value only.

→ %lp, %lu → will print 32 bit physical address i.e segment address & offset address also

NOTE :- For printing the address of a variable, we can't use %d format specifier because-

1) Addresses are available in the form of Hexadecimal from the range of 0X0000 to 0XFFFF in decimal (0 to 65536) so this range is not compatible.

2) There is no any -ve values are available in addresses but %d will print -ve data also

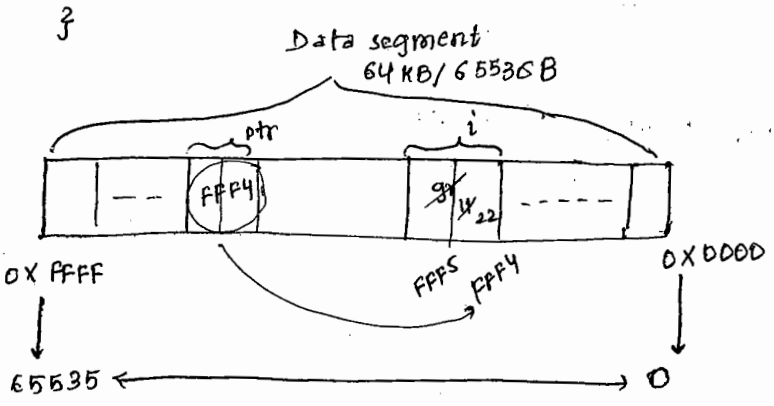
---

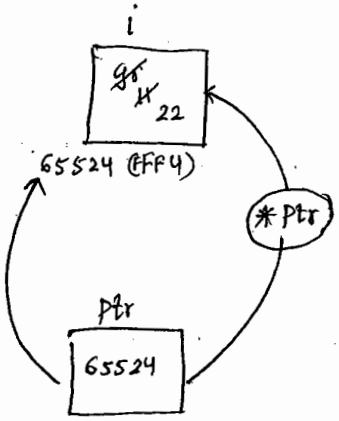
```
→ void main()
  {
     int i;
     int * ptr;
     ptr = &i;
     i = 11;
     printf("%p %p", &i, ptr);
     printf("\n%d.%d", i, *ptr);
     * ptr = 22;
     printf("\n %x %x", &i, ptr);
     printf("\n%d %d", i, *ptr);
  }
```

O/P :-

FFF4	FFF4
11	11
fff4	fff4
22	22

0X0000 → 0000 0000 0000 0000
0XFFFF → 1111 1111 1111 1111
0XFFF4 → 1111 1111 1111 0100





%p format specifier → prints the address in form of Hexadecimal & alphabets are printed in Upper case.

%x format specifier → prints the address in form of Hexadecimal & alphabets are printed in lower case.

→ Address of operators always returns base address i.e starting cell address of a variable.

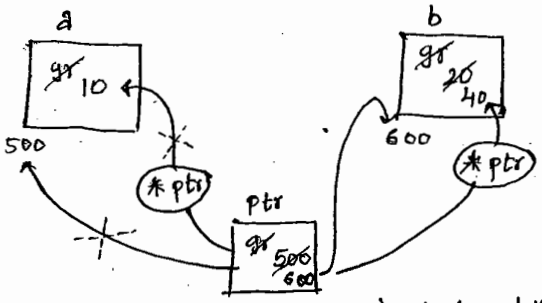
→ In previous prog, ptr holds an integer variable address thats why &i and ptr in direction operator always returns value of the address.

→ i and *ptr values are same because ptr is holding address of i.

```
void main()
{  int a, b;
   int *ptr;
   ptr = &a;
   a = 10;
   b = 20;
   printf ("\n%d %d %d", a, b, *ptr);
   *ptr = 30;
    ptr = &b; - now pointer is pointing to b
    b = 40;
    printf("\n %d %d %d ", a, b, *ptr);
}
```

O/P :-

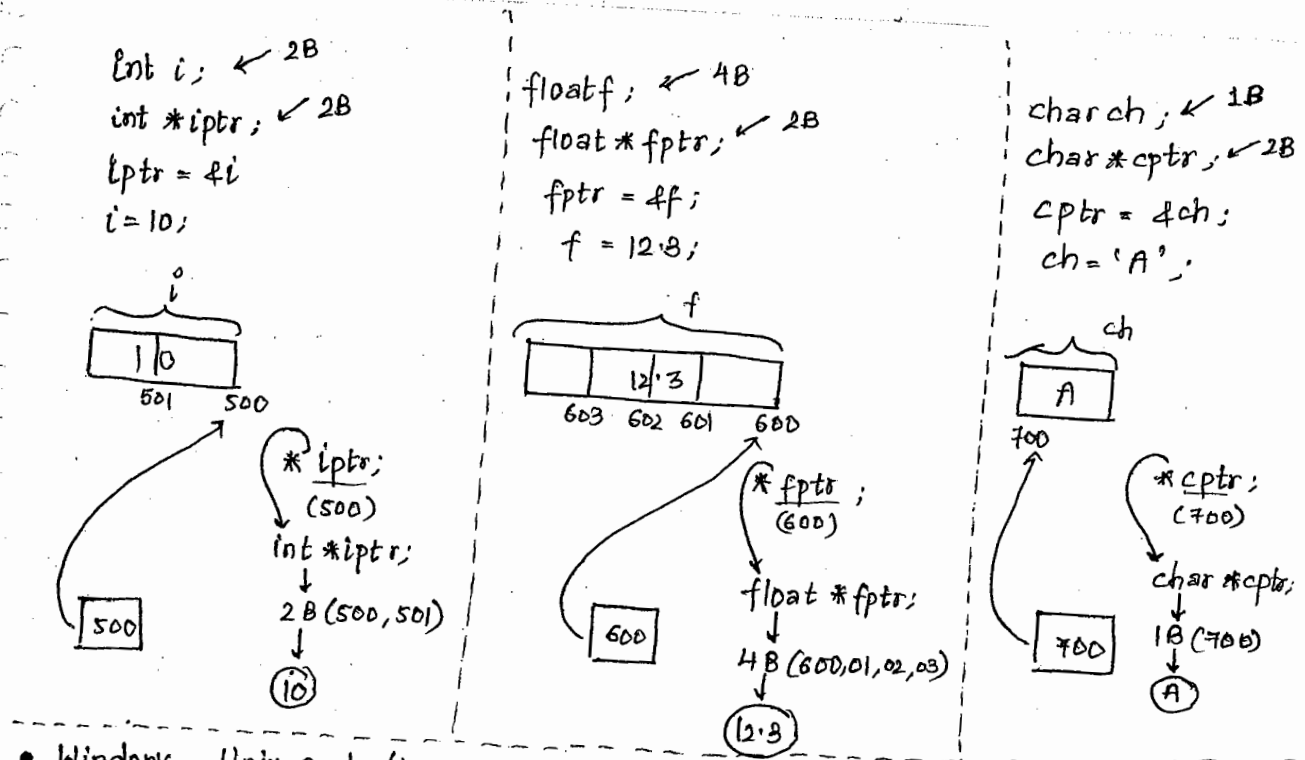
10	20	10
30	40	40



In implementation when we required to shift the pointer from one variable to another variable then we are required to reassign address of a variable to ptr

DATA ACCESS MECHANISM USING POINTERS

- Different types of variables are having diff. types of sizes bcoz internal content is diff.
- Any type of pointer having same size only because internal content is addres which is common for any type of variable.
- DOS os doesn't having any m/m management that's why at run time addresses are limited.
- On DOS based compiler (TC-3.0, 8086 based, 16 bit compiler). The size of pointer is 2 bytes bcoz addresses are limited.



- Windows, Unix and Linux OS are having proper m/m management that's why addresses are not limited (depends on RAM capacity)
- On windows based compiler (TC-4.5, c-free, Devctt, gcc compiler, 32 bit compiler) The size of the pointer is 4 bytes because addresses are unlimited.
- On 64-bit OS, when we are using 64 bit compiler then size of the pointer is 8 bytes

32 bit OS, 16 bit compiler size of pointer is 2 bytes
64 bit OS, 16 bit compiler not compatible
only 32 bit compiler and 64 bit compiler will work.

OS/Compiler → / ↓	16 bit	32 bit	64 bit
16 bit	2B	2B	X
32 bit	X	4B	4B
64 bit	X	X	8B

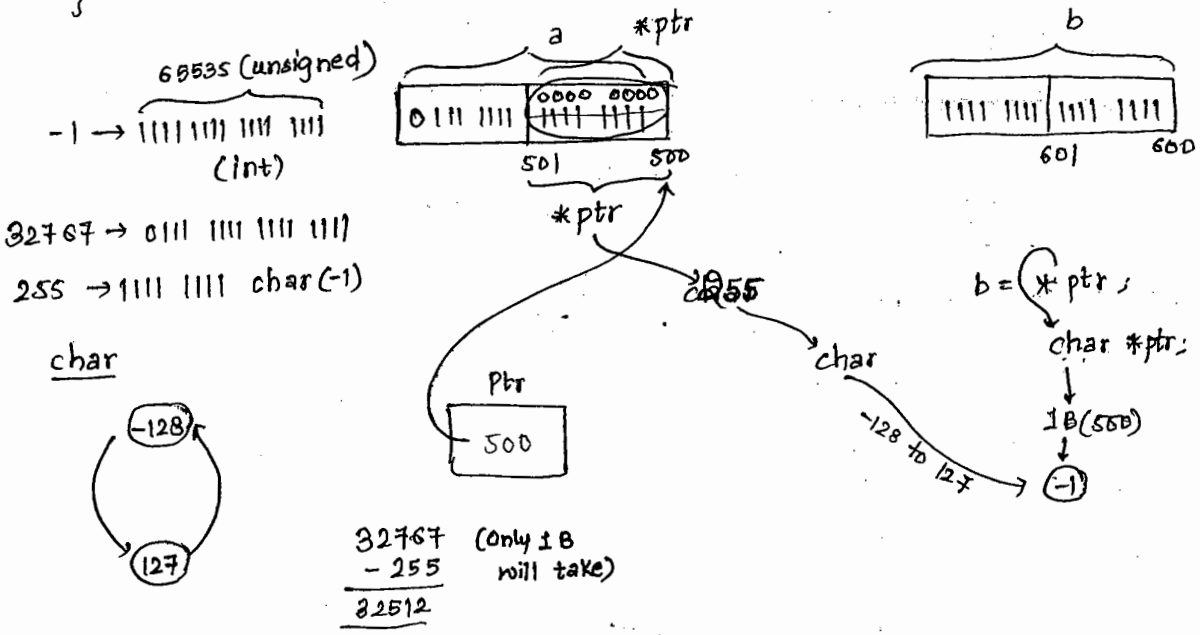
- Any type of pointer size is 2 bytes only because it maintains the offset address from the range of 0X0000 to 0XFFFF, so for holding this many address we require 2 B data
- Any type of pointer holds single cell info only i.e Base address but data can be present in multiple cells.

- On address when we are applying indirection operator then it deference to pointer type, So depending on pointer type given base address (in no. of bytes will be accessed)
- In 'C' prog. lang, any type of pointer can holds, any kind of variable address because address doesn't maintain any type information.
- In implementation when we are manipulating an integer variable then go for an int*, float variable float* and character variable is char* because we can see the difference when we are applying indirection operator.

→
```
void main()
{
  int a,b;
  char *ptr;
  ptr = &a;
  a = 32767;
  b = *ptr;
  printf("\n%d %d %d", a, b, *ptr);

  *ptr = 0;
  printf("\n%d %d %d", a, b, *ptr);
}
```

O/P :-

32767	-1	-1
32512	0	0

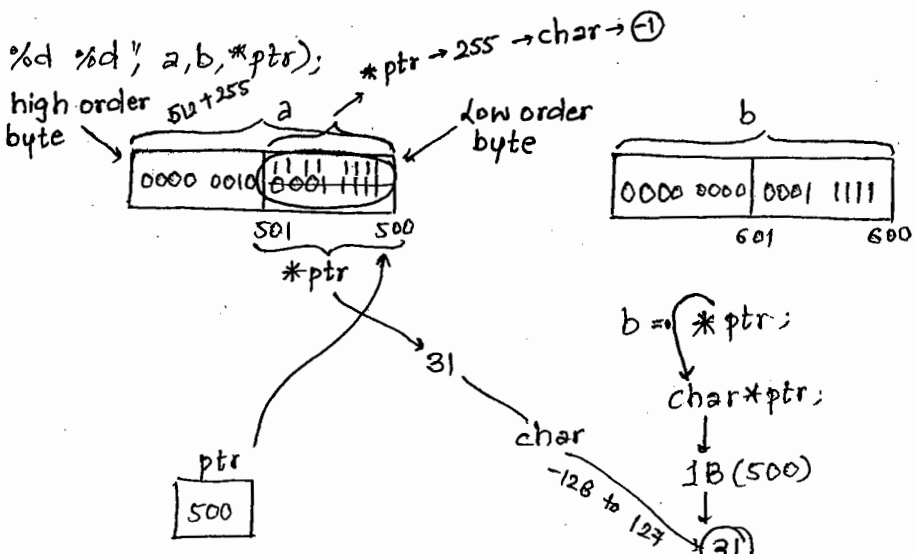


- On integer variable, when we are applying char pointer, then it can access and manipulate only (1Byte) data, because indirection operator behaviour is data type dependent.

→
```
void main()
{ int a, b;
  char *ptr;
  ptr = &a;
  a = 548;
  b = *ptr;
  printf ("\n%d %d %d", a, b, *ptr);

  *ptr = 255;
  printf ("\n%d %d %d", a, b, *ptr);
}
```

O/P :-

548	31	31
767	31	-1



→ All integer variable, when we are applying char ptr then always it will access low order byte data only.

→ When we are working with 2 byte integer, then 1st byte is called low order byte and 2nd byte data is called high order byte.

→ When we are working with pointers, generally we are getting following errors :-

1. Suspicious pointer conversion :-
   - This warning msg occurs when we are assigning address of a variable in diff. type of pointer.
   - These conversions are not allowed in C++.
2. Non-portable pointer conversion :-
   This warning msg occurs when we are assigning value type data to a pointer.

Ex-
```
void main()
{
  int a;
  int *ptr;
  ptr = a
}
```

ARITHMETIC OPERATIONS ON POINTERS :-

```
int P1, P2;
int *P1, *P2;
P1 = &i1;
P2 = &i2;
```

1) P1 + P2 ; Error
2) P1 + 1 ; Next address
3) ++P1 ; } Next address
   P1++ ; }
4) P2 - P1 ; No. of elements
5) P2 - 1 ; Pre address
6) --P2 ; } Pre Address
   P2-- ; }

1) P1 * P2 ; } Error
   P1 * 2 ; }
2) P1 / P2 ; } Error
   P1/5 ; }
3) P1 % P2 ; } Error
   P1 % 2 ; }

POINTER RULES

Rule 1 :-

Address + Number = Address (Next Address)
Address - Number = Address (Pre Address)
Address ++ = Address (Next Address)
Address -- = Address (Pre Address)
++ Address = Address (Next Address)
-- Address = Address (Pre Address)

Rule 2 :-

Address - Address = Number (No. of elements)
= Size Diff / sizeof (Datatype)

```
int *p1 = (int*)100
int *p2 = (int*)200
p2 - p1 = 50
200 - 100 → 100/size of (int)
```

Rule 3 :-

Address + Address = Illegal
Address * Address = Illegal
Address / Address = Illegal
Address % Addres = Illegal

Rule 4 :-

We can't perform bitwise operation b/w 2 pointers like

Address & Address = Illegal

Address | Address = Illegal

Address ^ Address = Illegal

~ Address = Illegal

Rule 5 :-

We can use relational operator and conditional operator b/w two pointers (<, >, <=, >=, ==, !=, ?:)

Address > Address = T/F

Address >= Address = T/F

Rule 6 :- We can find size of a pointer using sizeof operator

Pointer-to Pointer

→ It is a procedure of holding the pointer address into another pointer variable.

→ In C prog. lang., pointer to pointer relations can be applied upto **12 stages.**

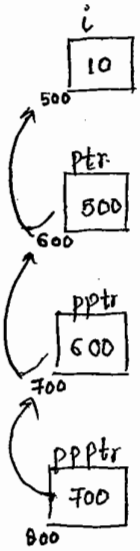
→ For a pointer variable, we can apply **12 indirection operators.**

→ When we are inc. pointer to pointer relations then performance will be decreased.

Syntax :-

pointer :	Datatype *ptr;
P2 pointer :	Datatype **ptr;
P2 P2 pointer :	Datatype ***ptr;
P2 P2 P2 p :	Datatype ****ptr;

```
void main()
{
  int i;
  int *ptr;
  int **pptr;
  int ***ppptr;
  ptr = &i;
  pptr = &ptr;
  ppptr = &pptr;
  i = 10;
}
```



① &i → 500

2) ptr → 500

3) i → 10

4) *ptr → 10

1) &ptr → 600

2) pptr → 600

3) *pptr → 500

4) **pptr → 10

*(600)
↓
*(500)
↓
10

1) &ptr → 700

2) ppptr → 700

3) *ppptr → 600

4) **pptr → 500

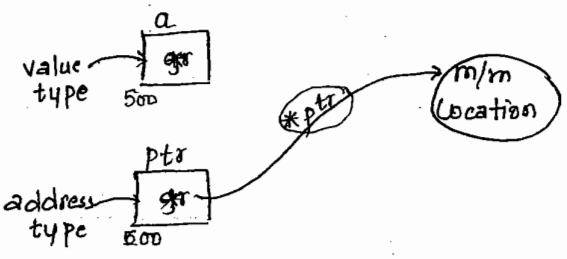
5) ***ptr → 10

*(700)
↓
*(600)
↓
*(500)
↓
10

→ Wild pointer :-

- Uninitialized pointer variable or the pointer which is not initialized with any variable add. is called wild pointer
- Wild pointer is also called bad pointer bcz without assigning any variable address, it is pointing to a m/m location

```
void main()
{
  int a;
  int *ptr; // Wild or bad pointer
}
```



→ When we are working with pointers, always recommended to initialise with any variable address or make it NULL.

NULL Pointer :-

- The pointer variable which is initialised with null value it is called null pointer
- Null pointer doesn't points to any m/m location until we are not assigning the address
- Sizeof Null pointer also is 2 bytes acc. to Doc compiler.

Syntax :

Datatype * ptr = NULL ; <stdio.h>

Eg :- int* ptr = NULL;

char*ptr = NULL;

float *ptr = NULL;

Datatype * ptr = (Datatype *) NULL;

Ex: int * ptr = (int *) NULL;
char *ptr = (char*) NULL;
float *ptr = (float *) NULL;

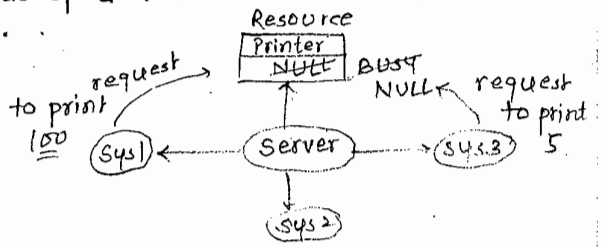
Datatype * ptr = (Datatype*) 0;

Ex: int *ptr = (int*)0;
char* ptr = (char*)0;
float *ptr = (float*)0;

- NULL pointer can be used like error code of a function
- NULL can be used like a constant integral value.

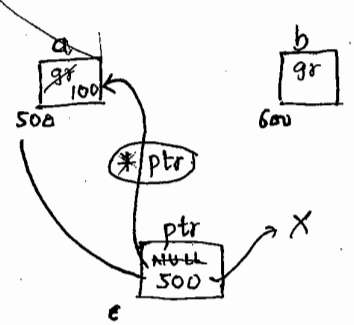
```
int x = NULL;          // int x = 0;
```

- By using NULL, we can check the status of a resource, i.e it is busy with any process or free to utilize.




```
→ void main()
{
  int a, b;
  int *ptr = (int*)0;
            // int*ptr = NULL;   <stdio.h>
  if (ptr==0)
  {
    ptr = &a;
     a = 100;
  }
  if (ptr==(int *)0)
  {
    ptr = &b;
     b = 200;
  }
  printf (" Value of *ptr : %d", *ptr);
}
```

O/P: Value of *ptr = 100



1) Check whethe ptr is null or not

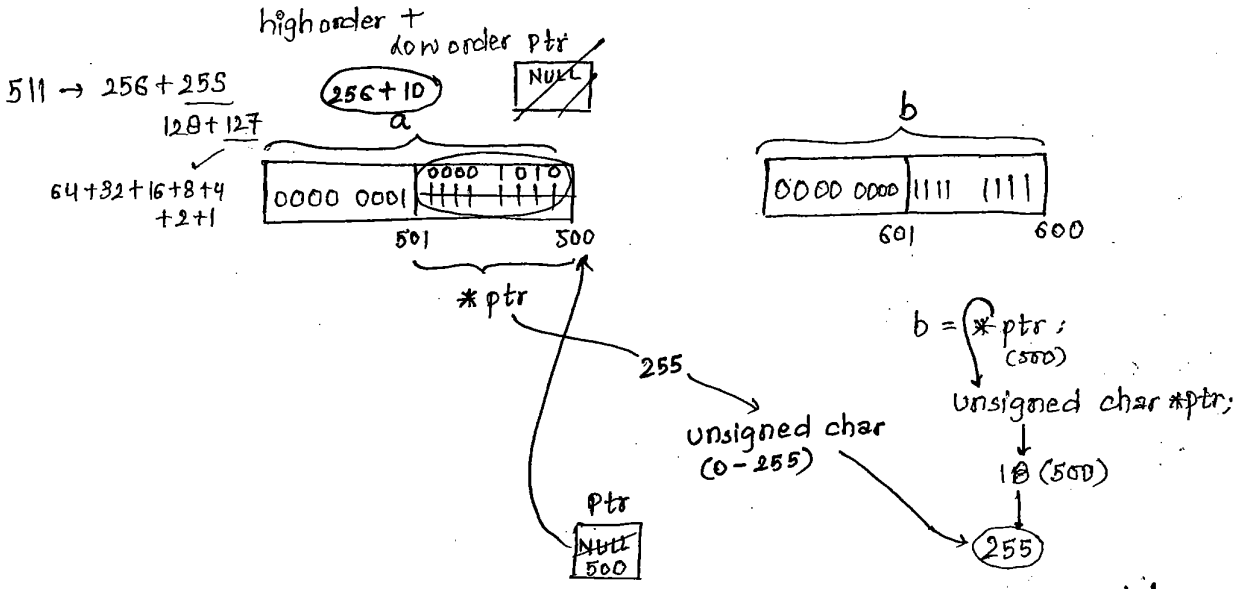
if (ptr==0)
NULL==0 True
{
ptr = &a
then put 500
In a = 100
}
then ptr ≠ NULL
not access b

→ void main()

```
{ int a,b;
  unsigned char *ptr = (unsigned char*)0;
  ptr = &a;
  a = 511;
  b = *ptr;
  printf("\n%d %d %d", a, b, *ptr);
  *ptr = 10;
  printf("\n%d %d %d", a, b, *ptr);
}
```

O/P:

511	255	255
266	255	0



- On integer variable when we are applying unsigned character pointer then it can access and manipulate 1B data only bcz indirection operator behaviour is data type dependent.

- TC-3.0 designed on 8086 Architecture.
  In this Architecture m/m models are classified into 6 types i.e.
  (1) Tiny
  (2) Small
  (3) Medium
  (4) Compact
  (5) Large
  (6) Huge

→ By default, any application m/m model is small in TC 3.0

→ In TC-4.5 m/m models are classified into 4 types i.e.

- Small
- Medium
- Compact
- Large

→ By default, any application m/m model is large in TC 4.5

- Always m/m model will decides that what type of pointer required to create.
- Depending on m/m model pointers are classified into 3 types :-
  1. near pointer
  2. far pointer
  3. huge pointer

1) near pointer - The pointer variable which can handle only 1 segment of 1MB data it is called near pointer.

→ near pointer doesn't points to any other segments except data segment.

→ The size of near pointer is 2 bytes.

→ When we are incrementing the near pointer value then it inc. Offset address only

→ When we are applying the relational operators on near pointer then it compares Offset address only

→ By default, any type of pointer is near only.

→ When we are printing the address, by using near pointer, then we required to use %d or %x or %u format specifier only

→ By using near Keyword, we can create near pointer.

2) far pointer - The pointer variable which can handle any segment of 1MB data, is called far pointer.

- When we are working with far pointer, it can handle any segment from the range of 0X0 - 0XF, but at a time only 1 segment.
- The size of far pointer is 4 bytes bcz it holds segment & offset add. also.
- When we are incrementing the far pointer value, then it increases offset address only.
- When we are applying relational operators on far pointer then it compares segment address along with offset address.
- When we are printing the address by ~~prin~~ using far pointer then we required to use %lp or %lu format specifier
- By using far Keyword, we can create far pointer.

3) huge Pointer:- The pointer variable which can handle any segment of 1MB data is called huge pointer

- When we are working with huge pointer, it can handle any segment from the range of 0X0 – 0XF, but at a time only 1 segment.
- When we are working with huge pointer, then it occupies 4B of m/m coz it holds segment & offset address also.
- When we are incrementing the huge pointer value, then it increase segment address along with offset address
- When we are comparing the huge pointer, it compares normalisation value.
- When we are printing the address by using huge pointer then we required to use %lp or %lu format specifier.
- By using huge Keyword, we can create huge pointer.

* Normalisation – is a process of converting (32 bit) physical bit into (20 bit) hexadecimal format.

IN DECIMAL

Physical address = (Segment address) * 16 + Offset Address

IN HEXADECIMAL

Physical address = (segment Address) * 0x10 + Offset Address

What is the normalisation value of 0X12345678 huge address ?

huge address :- 0X12345678

Segment address :- 0X1234

Offset Address :- 0X5678

Physical address = (Segment Address) * 0x10 + Offset Address,

= 0X 1234 * 0X10 + 0X 5678

= 0X 12340 + 0X 5678

= 0X 179B8

In binary :- 0001 0111 1001 1011 1000

→ One's is moved to ten's
→ 10's to 100's
→ 100's to 1000's
→ 1000's to 10,000's

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

01 * 010 = 010
012 * 010 = 0120
0123 * 010 = 01230

1 * 8 = 8
10 * 8 = 80

0X1 * 0X10
0X12 * 0X10
0X1234 * 0X10
= 0X12340

1 * 16 = 16
18 * 16 =

Void Pointer

→ Generic Pointer of C and C++ is called void Pointer

→ Generic Pointer means it can access and manipulate any kind of data properly. → (multiple datatype, 1 pointer)

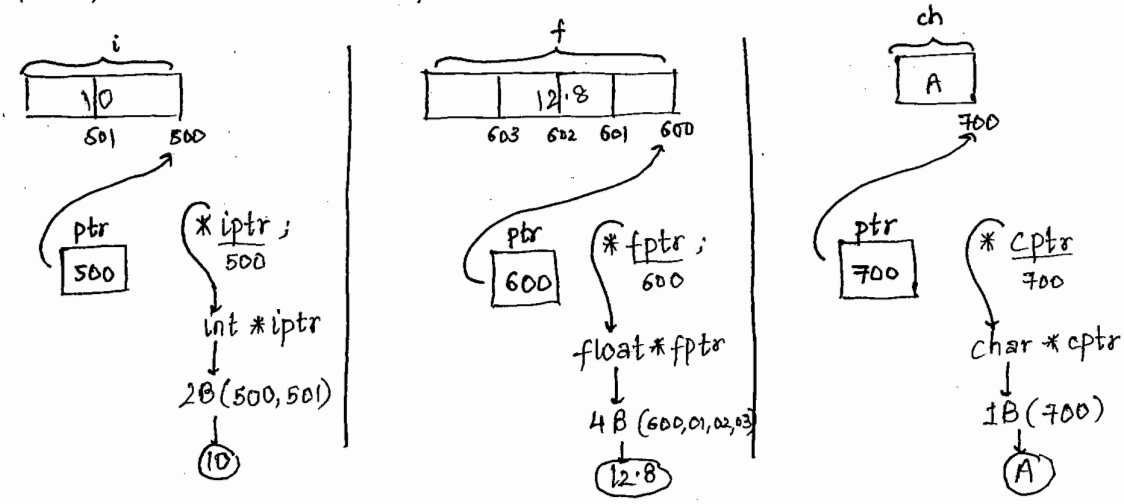
→ Size of void Pointer is 2 bytes

→ By using void Pointer, when we are accessing the data then we required to use Type Casting.

→ When we are working with void Pointer, Type specification will be decided at run time only.

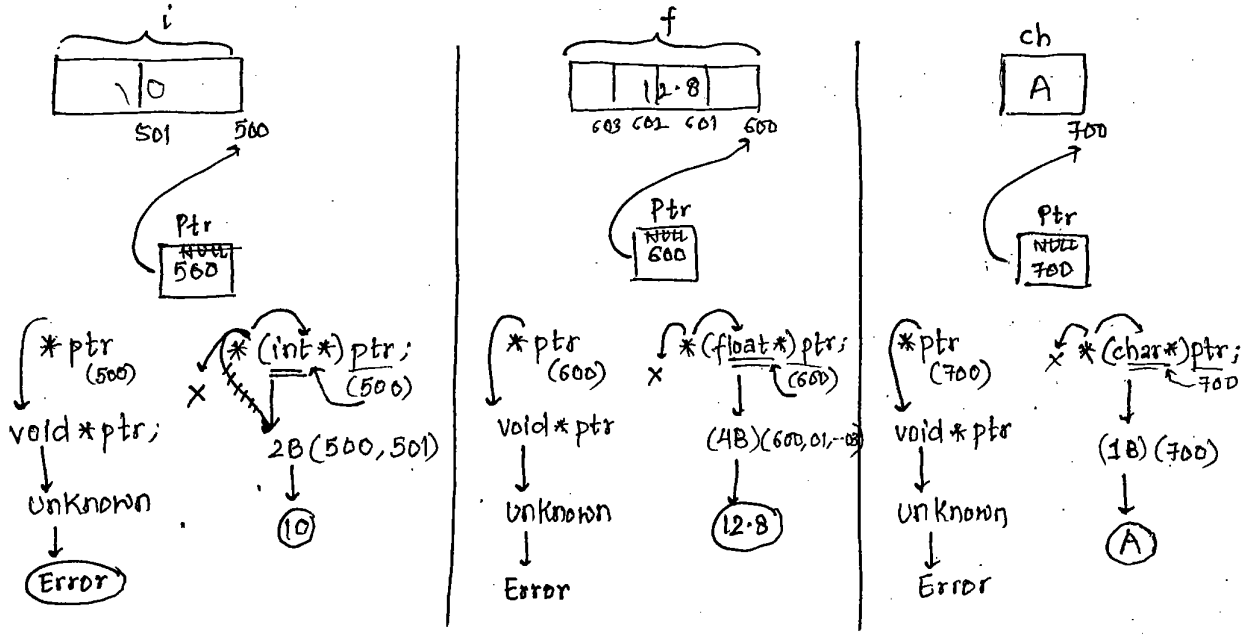
→ When we are working with void Pointer, arithmetic operations are not allowed, i.e. Incrementation and decrementation of pointer is restricted.

```
void main()
{
  int i;
  float f;
  char ch;
  int* iptr = (int*)0;
  float *fptr = (float*) 0;
  char* cptr = (char*) 0;

  iptr = &i;
  i = 10;
  printf ("\n%d %d", i, *iptr);

  fptr = &f;
  f = 12.8;
  printf ("\n%f %f", f, *fptr);

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

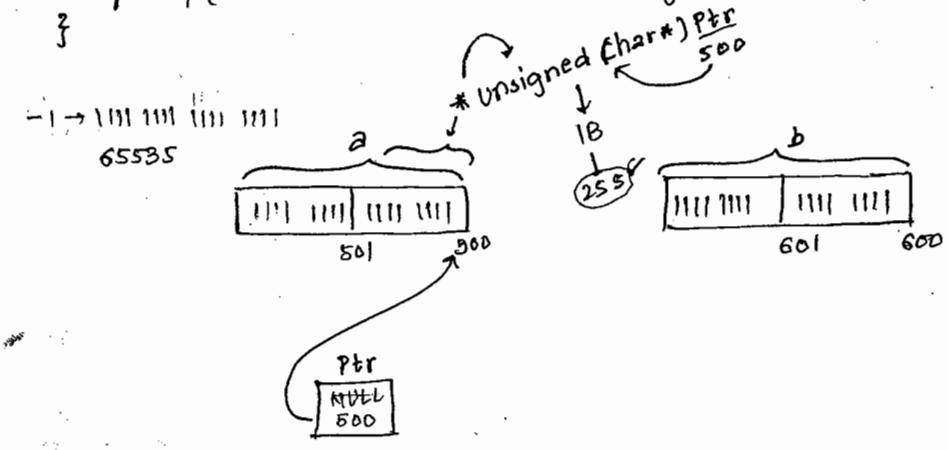


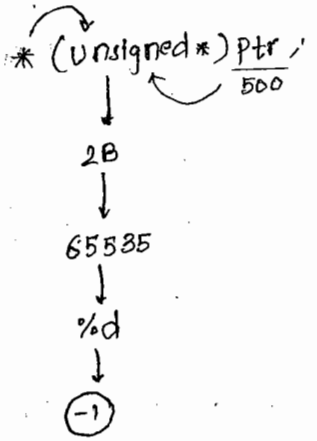
→ In previous prog., in place of constructing 3 types of pointers, we can create a single pointer variable which can access & manipulate any kind of variable properly. i.e void pointer required to use.

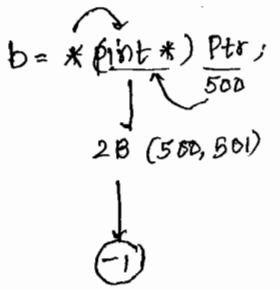
```
★★ void main()
   {
    int i;
    float f;
    char ch;
    void *ptr;
    ptr = &i;
    i = 10;                          // *(int*)ptr = 10;
    printf ("\n%d %d", i, *(int*)ptr);
    ptr = &f;
    f = 12.8;
    printf ("\n%f %f", f, *(float*)ptr);   // *(float*)ptr = 12.8;
    ptr = &ch;
    ch = 'A';
    printf ("\n%ch %c", c, *(char*)ptr);   // *(char*)ptr = 'A';
```

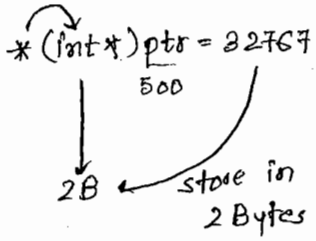


```
void main()
{
    int a,b;
    char* ptr;
    ptr = &a;
    a = -1;

    b = *(int*)ptr;
    printf("\n%d %d %d", a, b, *(unsigned*)ptr);

   *(int*)ptr = 32767;
    printf("\n%d %d %d", a, b, *(unsigned char*)ptr);
}
```









O/P :-

-1	-1	-1
32767	-1	255

Explaination