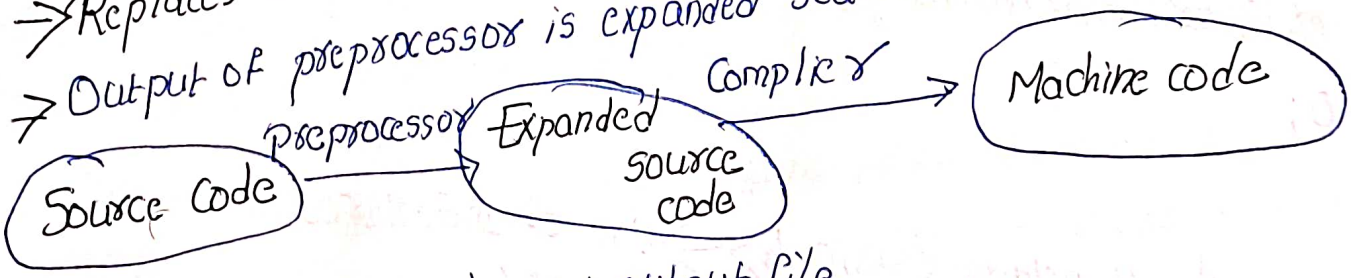
History of Computing:-

Features of C:-

→ procedual language
→ Middle level language
→ Direct access to memory through pointers } → System level features
→ Bit manipulation using bitwise operators
→ Writting assembly code with C code
→ popular choice for system level apps
→ Wide Variety of built in functions, standard libraries and header files
↳ printf

Example in C

// → Comments
include <stdio.h>
↓
It is a preprocessor. it replaces text (starting with #) with actual conten (-
→ Replaces before the compilation begins
→ Output of preprocessor is expanded source code



stdio.h → Standard input output file
- → header file
- → Contains declarations (prototypes) of functions like printf, scanf etc

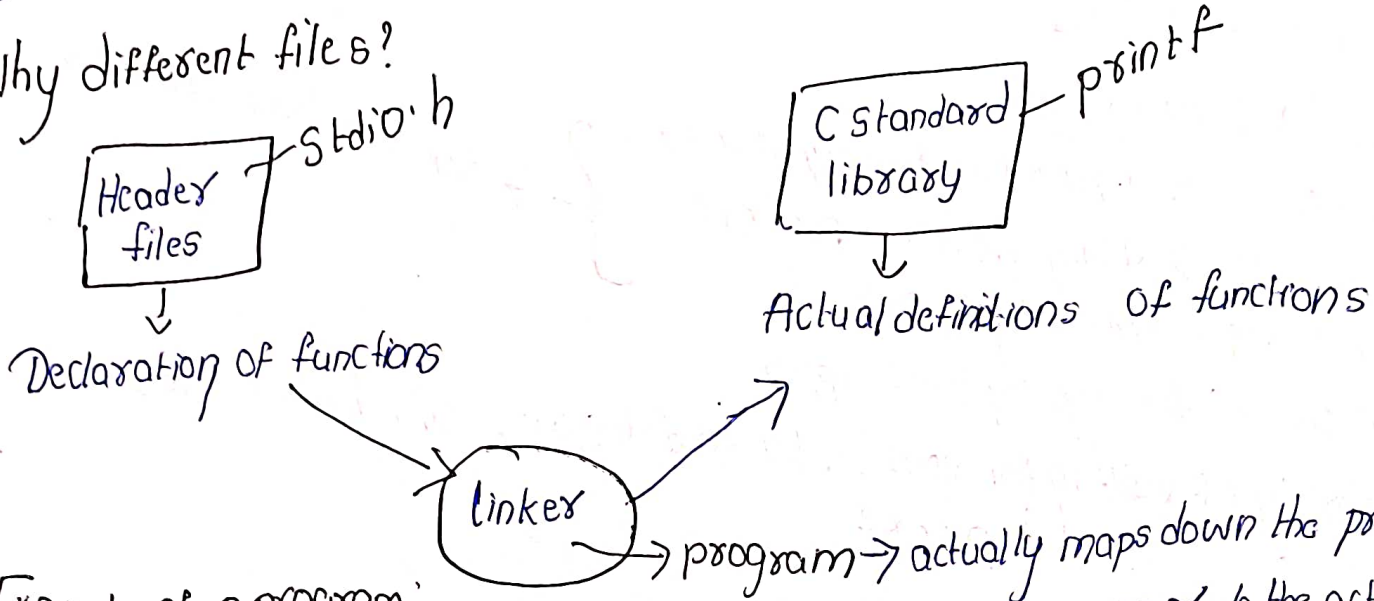
Syntax of a function:-

Return_typename_of_function (parameter_type name_of_parameter, parameter_typename_of_parameter ...)
{

Set of statements

}

Why different files?



Example of a program:-

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

program → actually maps down the prototypes mentioned by pre-processor to the actual codes written in this standard library. It simply maps and doesn't copy and paste content like preprocessor

→ Comment out the #include <stdio.h> → Use of undeclared identifier 'printf'

→ Remove the semicolon after printf → expected ';' after expression

→ Remove int return type from main → C requires a type specifier for all declarations

→ Instead of 0 in return write any integer you want → no error

→ Replace 0 with some character like 'a' → no error

→ Remove semicolon ; after return → expected ';' after return statement

Variable:-

⇒ Declaring variable before we use

Declaration: announcing the properties of variable to the complier

Properties: Size of the variable and name of the variable

Defination of variable: allocating memory to a variable

Most of the time declaration and defination will be done at same time

Example: int var; → Data type: how much space a variable is going to occupy in memory
→ Name of variable

Initialization:- int var = 3; (Initially it is storing and we can change after)

Variable = vary-able

⇒ Declaration of variable must be done always one time and use multiple times

Variable naming conventions:-

Variable name:- Composed of letters or combination of letters (both upper and lower case) and digits

Rule: don't start varname with digit → beginning with underscore is valid but not recommended → Case Sensitive → Special characters (@, #, %, ^, $, * ...) not allowed in the name of variable

→ Blanks or White spaces not allowed → Don't use keywords to name your variables. → Don't use long names for variables.

Basic Output function - printf:-

printf ("%d", var); %d is a place holder for variable

"d" means decimal

printf ("%d %d %d", var1, var2,);

error:

```
#include <stdio.h>
int main()
{
    int two = 2, three = 3, six = 6;
    int result = (two + three) * six / three;
    printf ("(%d + %d) * %d / %d", two, three, six, three);
    printf ("%d", result);
    return 0;
}
```

170
7

Fundamental data types – integer:-

Wanna know size programmaticly? ⇒ Use "size of" operator

⇒ Size of is a unary operator and not a function

Range:- Upper and lower limit of some set of data for example {0,1,2,3,4}

Range: 0 to 4

Decimal number system (base 10) ⇒ Range: 0 to 9

Binary number system: Machine understandable number system also called as base 2 number system ⇒ Range 0 to 1

For 4 bit data ⇒ Range 0 to 15 — for maximum of n bit $2^n - 1$

Rage of Integer

for 2 bytes (16 bits) → Unsigned range: 0 to 65535 (by applying $2^n - 1$)

Signed range: -32768 to +32767

2s Complement range: $-(2^{n-1})$ to $+(2^{n-1}-1)$

If Pc supports → Unsigned range 0 to 4294967295
4 bytes → Signed range : -2147483648 to +2147483647
(32 bits)

Modifiers :

Long and short :- If integer is 4 byte → long int may be 8 bytes
→ short int may be 2 bytes

Syntax :- Size of (short int) (or) sizeof(long int)

⇒ Size of(short) <= sizeof(int) <= Size of(long)

Note :- bydefault int some_Variable_name; is signed integer Variable
→ Unsigned int some_Variable_name; allows only positive values

Programming Examples :-

(1)

```
#include <stdio.h>
#include <limits.h>   → To determine min and max value of system
int main()
{
    int var1 = INT_MIN;
    int var2 = INT_MAX;
    printf("range of signed integer is from : %d to %d", var1, var2);
    return 0;
}
```

O/P :- ⇒ -2147483648 to 2147483647

(2)

```
#include <stdio.h>
#include <limits.h>
int main()
{
    unsigned int var1 = 0;
    unsigned int var2 = UINT_MAX;
    Print("%u to %u", var1, var2);   → unsigned decimal value
    return 0;
}
```

O/P :- 0 to 4294967295

(3)

```
#include <stdio.h>
#include <limits.h>
int main()
{
  short int var1 = SHRT_MIN;
  short int var2 = SHRT_MAX;
  printf("%d to %d", var1, var2);
  return 0;
}
```

O/P:- -32768 to 32767

(4)

```
short unsigned int var1 = 0;
short unsigned int var2 = USHRT_MAX;
printf("%u to %u", var1, var2);
return 0;
}
```

O/P:- 0 to 65535

⇒ short → long %u → %ld %u → %Lu → unsigned long integer

(signed long integer)

if sizeof(long int) = 4 bytes then sizeof(long long int) = 8 bytes

else if sizeof(long int) = 8 bytes then sizeof(long long int) = 8 bytes

→ Exceeding the unsigned and signed range:-

Let's take an example of 3bit (unsigned range)

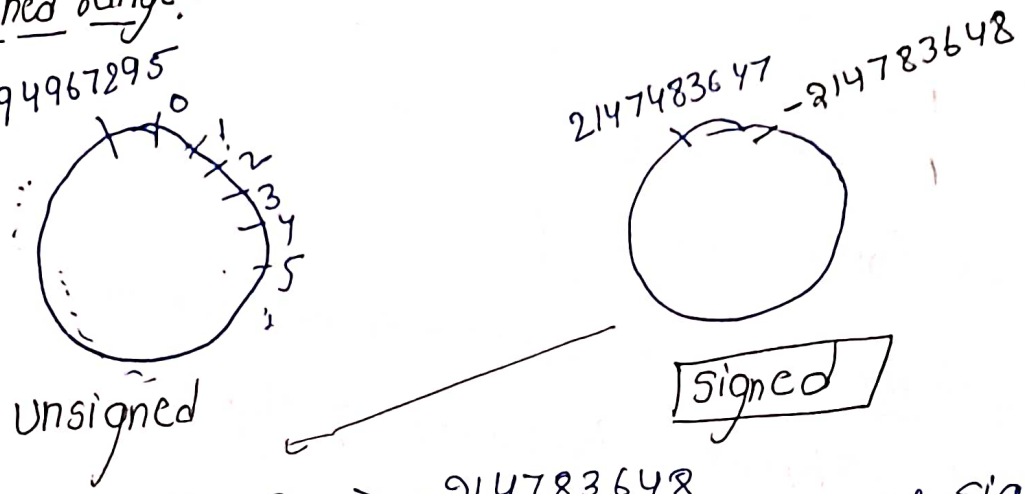
000 → 0
001 → 1
010 → 2
011 → 3
100 → 4
101 → 5
110 → 6
111 → 7
1000 → 0
1001 → 1
9

It looks like 8 mod function

⇒ So it returns 0 when 8 is there and 1 when 9 and 1 to 7 it is cycle

for 32 bit unsigned data → Mod 2^{32} | for n bit unsigned data → Mod 2^n

Signed range:-



⇒ 2147483648 → -2147483648
-2147483649 → 2147483647 → max of signed

Fundamental data type → char -

Char Eswar = 'N'; (only single not double quotes)

Char data type can have only character. It can't hold the string to it

Char Eswar = 65;

Char only hold 1 byte of information

%c → format specifier for character

Example:

```
#include <stdio.h>
int main()
{
  Char var = 65;
  Printf("%c", var);
  return 0;
}
```

O/P :- A (acc to Ascii table)

Size and range of Character

Size	Range
1byte = 8bits	Unsigned → 0 to 255
	signed → -128 to +127

Signed Vs Unsigned

-128 =	1000000	+128 =	1000000
-127 =	1000001	+126 =	1000010
+129 =	1000001	+130 =	1000010

So they print same char (acc to Ascii table)

float, double and long double :- used to represent fractional numbers

Float ⇒ 4 bytes Double ⇒ 8 bytes long double ⇒ 12 bytes

Size of these data types totally depend from system to system

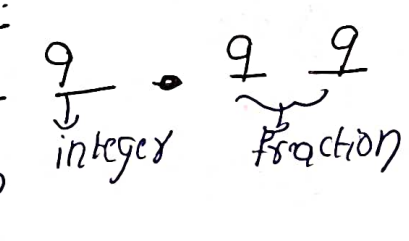
Float → IEEE 754 Single precision floating point

Double → IEEE 754 double precision floating " %f → place holder

long double → Extended precision floating "

fixed point Representation

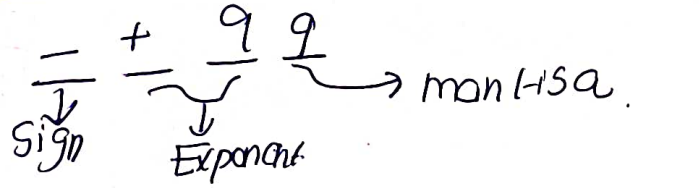
Ex:



That's Why floating point representation choose over fixed point representation.

Floating point representation :-

Example:



$0.M \times Base^{Exp}$

$Min :- -0.9 \times 10^{+9}$

$Max :- +0.9 \times 10^{+9}$

float var1 = 3.142fm - - - - -

printf("%.16f\n", var1);

%f is a place holder for long double

%Lf is a place holder for double

float ⇒ upto 6 decimal points

double → upto 15 decimal points

long double → upto 18 decimal points

accurate after that everything will change

⇒ division (or) multiplication (or) addition (or) sub

⇒ int/int → int int/float → float float + float / float

%s → string of character s

printf("%d", printf("%s", "Hello World!"));

O/P:- Hello World 12

printf("%10s", "Hello");

O/P:- _____ Hello

5 whitespaces

10 characters wide

it will print first second printf then first

```
Char c = 255;
  c = c+10;
printf("%d", c);
  return 0;
```

exceeding the range

⇒ 28

256 | 265

1 → 9

Ans ⇒ 9

int i ⇐> signed int i

Signed i; → Because integer is implicitly assumed
Unsigned i;
long i;
longlong i;

```
Unsigned i = 1;
  int j = -4;
Printf("%d", i+j);
  return 0;
}
```

→ check it %d

it returns signed Integer (%d)

Defining scope of variable:-

Scope = lifetime:

Local variable → Inside the function after that automatically destroys
global variable → Out-side the function
⇒ System choose local than global

What is AUTO Modifier:-

Auto means automatic
Variables declared inside a scope by default are automatic variables

⇒ for auto variable if the value is not assigned it produces some garbage value.

⇒ on other hand, global variable by default initialized 0

Extern modifier:-

int var;

Declaration and definition

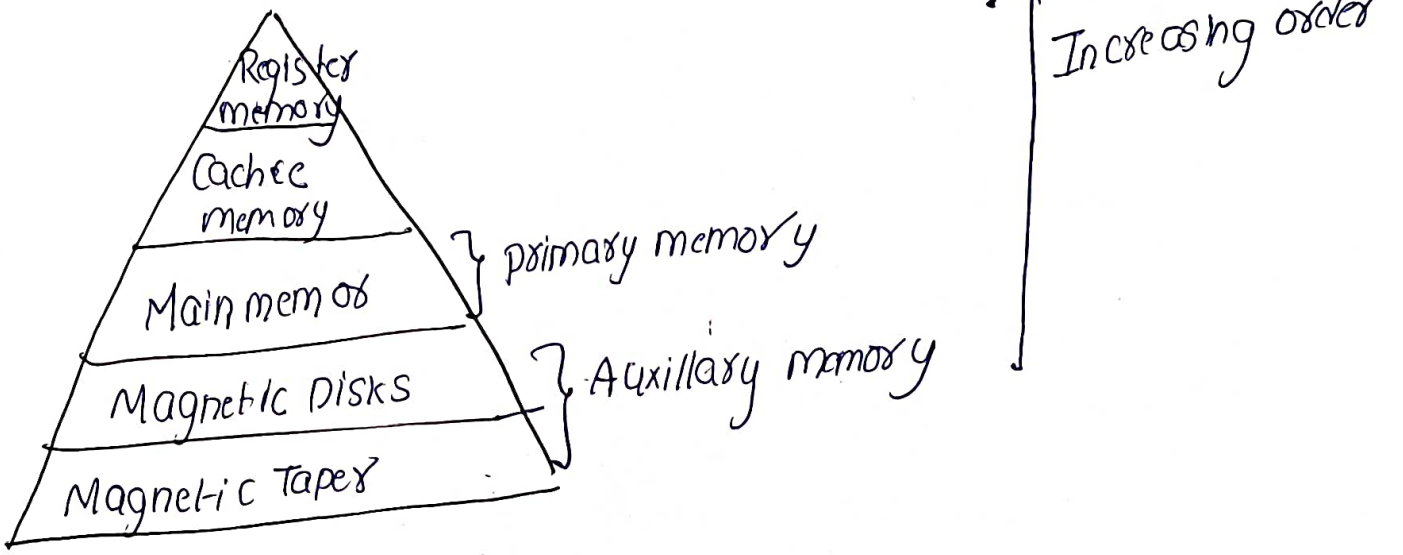
extern int var; → Declaration → not allocating memory

⇒ Extern is shortname for external

⇒ Used when a particular file needs to access a variable from another file

Multiple times of declaring is allowed

If Extern int var = 9; (then it is declared and defined



What is register modifier?

Syntax:- register int var;

⇒ Register keyword hints the Compiler to store a variable in register memory

⇒ This is because access time of register memory is so high

Constants:

- using #define
- using const

Syntax: # define (Name) value

Also called Macro

Job of preprocessor (not compiler) to replace NAME with value

```
#include <stdio.h>
#define PI 3.14159
int main () {
    printf ("%.5f", PI);
    return 0;
}
```

O/P:- 3.14159

⇒ Please don't add semicolon at the end
⇒ Choosing capital letters for NAME is a good practice
⇒ Whatever inside double codes will not be printed
⇒ We can use macros like functions (video)
We can write multiple line using \ → video → We can't declare another variable
6) First expansion then evaluation → video
7) Some predifined micros like __DATE__, __TIME__ - Can print date and time

Syntax of Const:- `Const int var1;` (Read only value)

Can't assign var1 to another value only that is fixed

quiz

```
int var = 052
printf ("%d", var);
```

If 0 is present it's treated as octal value

(052)$_8$ → $5 \times 8^1 + 2 \times 8^0 = 42$

"%d" → format specifier

Scanf → Scanformatted string

Accept character, string and numeric data from user using input - keyboard

Scanf also use format specifiers like printf

Ex: %d to accept input of integer .....

```
int var;
Scanf("%d", &var);
```

→ address of operator

Why &? → While scanning the input, scanf needs to store that input data somewhere

=> To store this input data, scanf needs to know the memory location of a variable

That's why we use &var

Quiz

```
int var = 0x43FF;
printf("%X", var);
```

→ It represents the hexadecimal value

"%X" → format specifier for hexadecimal

What if "%d"

Operators in C:-

Arithmetic operators	+, -, *, /, %
Increament/decreament operators	++, --
Relational operators	==, !=, <=, >=, <, >
logical operator	&&, ||, !

Bitwise operators → &, ^, |, ~, >>, <<

Assignment operators → =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=

Other operators → ?; & * sizeof() ,

Arthmetic operations → +, -, *, /, %
These are binary operator.

Operator precedence and associativity

Precedence	Operators	Associativity
highest ↓	*, /, %	left to right
lowest	+, -	left to right

All are same (same precedence)

Note: Associativity is used only when two (or) more operators are of same precedence
for example: +, - (same predence we use associativity / do coding for this)

Increament and decreament operator:-

→ Increament operator is used to increament the value of a variable by one. Similarly, decreament operator is used to decrament the value of a variable by one

```
int a=5;
a++;
a=6
```

a++ = a+1

```
int a=5;
a--;
a=4
```

a-- = a-1

⇒ Both are unary operators.

a++a; ✗ (only unary operator)

Pre-increament operator

++a;

--a;

post-increament operator

a++;

a--;

⇒ You can't use rvalue before or after increament/decreament operator

Example: (a+b)++; ✗ (error)

++(a+b);

lValue (left value): Simply means an object that has an identifiable location in memory (ie having an address).

⇒ In any assignment statement "lvalue" must have the capability to hold data

⇒ So it must be variable ⇒ Lvalue cannot be a function expression (like a+b) or a constant (like 3, 4 etc).

rValue (right value): Simply means an object that has no identifiable location in memory.

⇒ Anything which is capable of returning constant value

→ Expression like a+b will return constant value

a++;

holddata a = (a+1); return constant

(a+b)++;

(a+b) = (a+b) + 1; ✗

hold data

hold data

difference b/w preincreament/decreament and post increament/ decreament value

Pre → means first increament/decreament then assign it to another variable

Post → first assign it to another variable then increament/decreament

```
x = ++a;
```

x: 6 | a: 6

```
x = a++;
```

x: 5+0 | a: 5+1 = 6

a=4 b=3; a+++b ?

Token generation:-

→ lexical analysis is first phase in compliation process

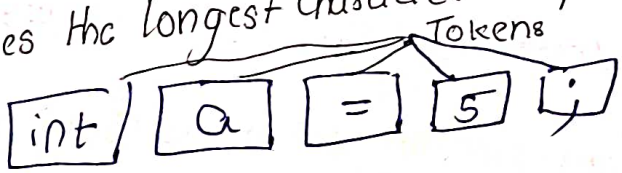
→ Lexical analyzer (scanner) scans the whole source program and when it find meaningful sequence of characters (lexemes) then it converts it into a token

Token: lexemes mapped into token-name and attribute-value.

Ex: int -> <keyword, int>

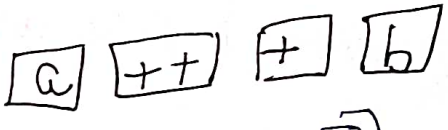
→ It always matches the longest character sequence

int a = 4;



question a=4; b=3;

a+++b



4+3

= 7

Postincreament → first use value in eqn then increament it

a + + + b (space between the two +)

[a] [+] [++b]

[4 + 4 = 8]

Preincreament :- First increament the value and then use in the equation after completion of equation

a++++++b

[a++] [+] [++b]

⇒ 4 + 4 = 8

[&& &&] - do ↳ no meaning

a = 4; b = 4;

Relational operators ==, !=, <=, >=, <, >

Comparing two values

→ All relational operators will return either True (or) False

Logical operator: && , || , ! → NOT

AND OR

&& and || are used to combine two conditions

&& → Returns True when all conditions under True or else False

|| → returns True (one) (or) more conditions (or) else False

! operator is used to complement when the condition under consideration

! → returns True when conditions is false and vice versa

```
a = 5
if (!(a == 6))
  print("Eswar");
```

Short Circuit in Case of &&:- Simply means if there is a condition anywhere in the expression that returns false, then rest of conditions after that will not be evaluated

Video ⟶ check with --a Where (a=1,)

Short circuit in case of ||:- Simply means if there is a condition anywhere in the expression that returns true, then rest of the conditions will not be evaluated.

Video

Bitwise operators

1) It does bitwise manipulation

&, |, ~, <<, >>, ^

difference between bitwise and logical operators:-

char x = 1, y = 2;

8bits

0001
0010

if (x & y)
printf ("Result of x & y is 0");

if (x && y)
print ("Result of x && y is 1");

left left shift operator

First operand << second operand

whose bits get left shifted ↓ decides number of places to shift

⇒ left shifting is equivalent to multiplication by 2^right operand

var=3; var<<1 output: 6 (3×2¹)

var<<4 output: 48 (3×2⁴)

Right shift — first operand >> second operand

var=3 var>>1 (3/2)=1

Right shifting is equivalent to division by 2^right operand

var>>3 (3/2³)=0

XOR —

Inclusive OR

→ Either A or B is 1 or Both are

⇒ Including both

XOR

Exclusive OR

Either A is 1 or B is 1 then output is 1 but both A and B are 1 then output is 0

× Excluding both

int a=4 b=3;

a=a^b;
b=a^b;
a=a^b;

a=4^3

a=7

b=7^3

b=4

a=7^4

a=3

0110
0011
0101

4+1 = 5

0101
0011
0110

0101
0011
0110

(7) (3)

Assignment operator :- (Binary value)

Lvalue R-value.

This operator copies R-value to L-value

var = 5 ;
L-value rvalue

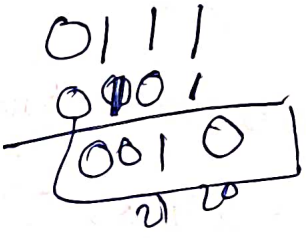
+= -= *= /= %= <<= >>= &= |= ^=

Assignment operator is having least precedence expect comma operator

a+=1 ⇒ a=a+1

char a = 7

a^=5



(7) (2)

a+=3

a = a+3
= 2+3 = 5

Conditional Operator :-

result = (marks > 33) ? 'p' ; 'f' ;

⇒ quick facts about conditional operator :-

⇒ Conditional operator is only ternary operator available in list of operators in clanguage.

Exp1 ? Exp2 : Exp3

int result;
result 0 ? 1 ; 2 ;
1
Nested

Comma (,) operator:-

=> Comma Operator is used as seperator

=> Comma Operato can be used as an operator

```
int a = (3,4,8);
printf("%d", a);
```

Comma returns right most operator and simply rejects everything but Evaluate it

a=8

```
int var = (printf("%s\n", Hello), 5);
printf(var);
```

③ Comma operator having least precedence

```
int a = 3,4,8;   — Compile
```

Bracket has highest precedence among all operators

Precedence of operators :- Precedence of a operator comes into picture when in an expression we need to decide which operator will be evaluated first. Operator with higher precedence will be evaluated first

Associativity of Operators :- Associativity of operators come into picture when precedence of operators are same and we need to decide which operator will be evaluated first

Left to right

()→paranthesis in function calls (highest precedence)

If suppose = operator is having greater precedence then, fun will belong to = operator and therefore it will be treated as a variable.

Member access operater (->.) :-

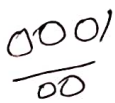
They are used to access members of structures

postfix Increment/decreament (++, --)

⇒ Precedence of Post fix Increament/decrement operator is greater than prefix Increament/Decreament

⇒ Associvity of postfix operator is left to right

⇒ Associvity of prefix operator is right to left



If Else:- Nested if

Switch:- Switch is a great replacement to long else if constructs

```
int x = 2;
switch (x)
{ case 1: printf ("x is 1");
          break;
  case 2: printf ("x is 2");
          break;
  default: printf ("x is another number");
           break;
```

Facts related to switch:-

1) You are not allowed to add duplicate cases
2) only those expression are allowed in switch which results in an integral constant value (not floating point)
3) Floating constants are not used in case value even case (3*4+5) - allowed case (1.5) - not allowed
4) variables are not allowed in case statement but macros are allowed
5) default can be placed anywhere inside switch. it will still get evaluated if no match is found

Importance of loops:

Syntax of while:-

```
while (expression)
{
 St1;
 St2;
 :
}
```

for loop

Syntax of for loop:

```
for (initialization ; condition ; increament/decreament)
{
 Statements;
}
```

Difference b/w While loop and do while loop:-

While:

```
int i=0
While (i>0)
{  printf ("%d", i)
   i--;
}
```

do-While

```
int i=0;
do
{  printf ("%d", i);
   i--;
}
While (i>0);   → semicolon main
```

In dowhile first body will be excuted then checks the condition

Whenever you wanna excute body atleast once you must use dowhile over while.

Break:- used to terminate from the loop

Continue: Similar to break but instead of termanating from the loop, it force to excute the nex iteration of the loop

Loops question 1:-

```
i=1024;
for( ; i ; i>>=1)
    print("Hello,World");
```

i=0, i; (i)

⇒ 11 times helloworld would print

Question 2:-

```
int i=0;
for(i=0; i<20; i++)
{
  switch(i)
  {
    case 0: i+=5;
    case 1; i+=2;
    case 5; i+=5;
    default: i+=4;
  }
  printf("%d", i);
```

X

o/p: 5 10 15 20

no break

Op:- 16,21

Question:

```
int i=0;
for (print("one\n"); i<3 && printf(""); i++)
    {
    printf("Hi\n");
}
```

"one\n"; i<3 ✓

Ole!

One
hi
hi
hi

{ unsigned int i=500;

→ 0 to 4294967295 → after this 0 will com e

```
while (i++ != 0);
printf("%d", i);
return 0;
}
```

→ Semicolon encountered no body for this it just simply check the condition untill 0 encountered

0 != 0

i++ = i = 0+1 = 1

int x=3;

if (x==2); x=0; → ends because of semicolon → it will assign to it

if (x==3) x++;

else x+=2; → x=2

4

5 >= 5 - [1-1]

5 >= 5

Pyramid of stars

→

for (int i=0; i<=n, i++);

for (j=0;

for pyramid we need 3 rows → 5 colums

4 rows → 7 colums

5 rows → 9 colums

6 rows → 11 colums

n rows → (2n-1) colums

(2n+1)

$j = 2j - 1$

for(int i=1; i<=rows, i++);

for (int j=1; j<=colums; j++);

if (j >= n-(i-1) && j <= (n+(i-1)))

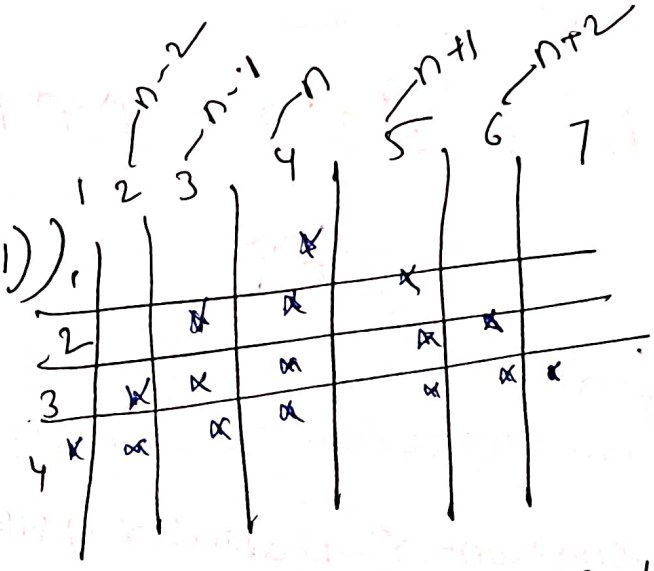
{ print("*");

}

else

{ print(" ");

}



n=5

i-2+1

i-2-1

n-(i-1)

n+(i-1)

for(int i=0; i<=n; i++)

for(int j=0; j<=(2n-1); j++)

if (j >= n-(i-1))

10-1

11

i >= 5-(0-1)

i >= 3 ✗

i <= 5+(-2)

i <=

Palindrome:

Palindrome: Number (or) a word of phase if read backwards gives same number (or) a word or phase

Ex: 12321 — 12321/10

Armstrong:-

⇒ An armstrong number of order n is a number in which each digit when multiplied by itself n no. of times and finally added together, result same number

371 ⇒ 3*3*3 + 7*7*7 + 1*1*1 = 27 + 343 + 1 = 371

371

Problem:- Write a program to check wheather number is strong (or) not

Strong number:- is a number in which the sum of factorial of individual digits of a number is equal to original number

145 = 1! + 4! + 5! = 145

Q) Write a program to add two numbers without using '+' operator

(Idea is to use increament and decrament Operators

Floyd's triangle

1	2	3	4	5
1				
2	3			
4	5	6		
7	8	9	10	
11	12	13	14	15

n+1

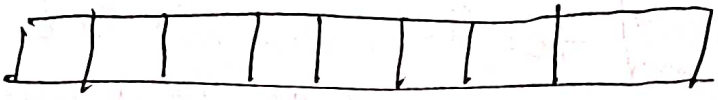
if i=j

Arrays:- An array is a data structure containing a no. of data value [all of which are of same type]

Data structure → format for organizing and storing data

Also, each data structure is designed to organize data to suit a specific purpose.

for example :-



One dimensional array

a = [5 | 10 | 15 | 20]

Declaration and define 1d array:-

Syntax data_type name [no.of elements];

Ex: int arr [5];

⇒ Compiler will allocate a contiguous block of memory of size = 5 * sizeof(int)

Can we specify the length of array as 5.6 (or) other than integer

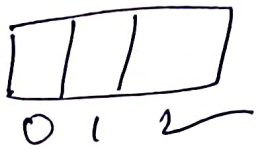
✗ (No. length of array can be specified by an positive integer constant expression like int arr [2/3];

Specifying the length of an array using macro is considered to be an excellent practice → Why? → because changing can be done only once if we use MACRO.

```
#define N 10
int arr[N];
```

Accessing elements from 1d array:-

⇒ array_name[index]



arr[0]; arr[1]; and so on

Initializing 1d array:

	→ you can add more
arr[5] = {1,2,5,67,32};	arr[] = {1,2,5,67,32};

```
int arr[5];                    X  not prefered method
arr[0]=1;
arr[1]=2;
arr[2]=5;
arr[3]=67;
```

```
int arr[5];
for(i=0; i<5; i++){
    scanf("%d", &arr[i]);
}
```

⇒ int arr[10] = {45,6,2,78,5,6};

The remaining locations of the array are filled by value 0

Why not int arr[10] = {}; Because, this is illegal. You must have to specify atleast 1 element and illegal to add more elements more than specified value

Designated initialization of Arrays:-

```
int arr[10] = {1, 0, 0, 0, 0, 2, 3, 0, 0, 0};
```

⇒ `int arr[10] = {[0] = 1, [5] = 2, [6] = 3};`

⇒ This way of initialization is called designated initialization and each number in square brackets is said to be designator

Reverse order:-

```
int main() {
    int a[9] = {34, 56, 54, 32, 67, 89, 90, 32, 21};
    int i;
    for (int j = n-1; i >= 0; i--)
```

Input: 67827 → O/p:- Yes [7 repeated twice] do yourself

Using size of operator with array:-

`Size of (name_of_arr) / size of (name_of_arr[0])`

Introduction of multidimensional array:-

Declaration and syntax:- array of array

```
data_type name_of_array[size1][size2]....[sizeN];
```

For example

```
int a[3][4]; // Two dimensional array
int a[3][4][6]; // Three dimension array
```

Size calculation:- Size of multidimensional

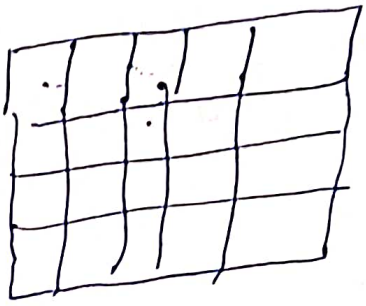
the size of all dimension

Ex: Size of a[10][20] = 10 × 20 = 200 (elements can be stored)

2dimensional

int arr[5][5];

rows ↑ columns ←



4×5

Intilizaing two dimensional array:-

method 1:- int a[2][3] = {1,2,3,4,5,6};

	0	1	2
0	1	2	3
1	4	5	6

method 2:- int a[2][3] = {{1,2,3}, {4,5,6}}

	0	1	2
0	1	2	3
1	4	5	6

Accessing of element:-

Using row index and column index

a[0][1] = 2

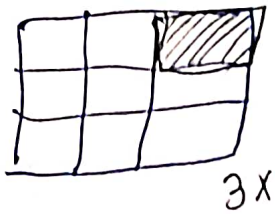
How to print 2d array element:-

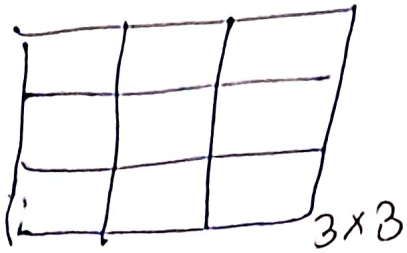
Two nested for loops

```
for (i=0; i<2; i++)
{ for (j=0; j<3; j++)
  {
    print ("a[i][j]");
  }
printf("\n")
}
```

Introduction to 3d arrays:-

```
int arr[2][3][3];
```





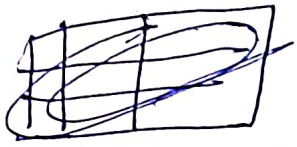

```
arr[0][0][2]
```

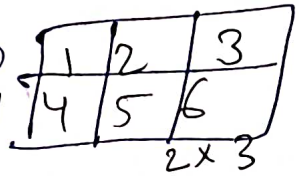
row

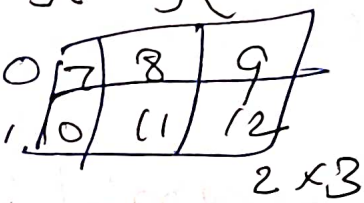
column

first two 2d array

Intilazing 3d array:- `int a[2][2][3] = {{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};`







Printing 3d arrays

```
for(int i=0; i<2 ; i++)
    for(int j=0 ; j<2; j++);
        for(int k=0 ; k<3 ; k++);
```

Write a program that reads a 5x5 array of integers and prints the row sum and Column sum;

8	3	9	0	10
3	5	17	1	1
2	8	6	23	1
15	7	3	2	9
6	14	2	6	0

Row total :- 30 27 40 36 28

Column total :- 34 37 37 32 21

```
int i,j;
 int sum = 0
for(i=0; i<5; i++)
{
 for(j=0; j<5; j++)
  {
  sum += a[i][j]
  }
  printf(" %d", sum);
   sum=0;
}
```

```
int k, l;
 int sum = 0
for(j=0; j<5; j++)
{
reverse
```

Matrix Multiplication :-

⇒ Inorder to multiply two matrices # columns of 1st matrix = # rows of 2nd matrix

⇒ Also, Size of resultant matrix depends on # rows of 1st matrix and # colums of 2nd matrix

— do your abacon

Constant arrays in c :- Either One dimensional (or) multidimensional arrays can be made constant by starting declaration with keyword constant. (const)

```
const int a[10] = {1,2,3,4,5 . . . 10}; // read only location
```

advantage :-

⇒ It assures us that the program will not modify the array which may contain some valuable info

Variable length of array:-

int a[n]; → variable

Pointers:- pointer is a special variable that is capable of storing some address

⇒ It points to a memory location where the first byte is stored

Syntax for declaring pointer variable:-

data-type *pointer_name

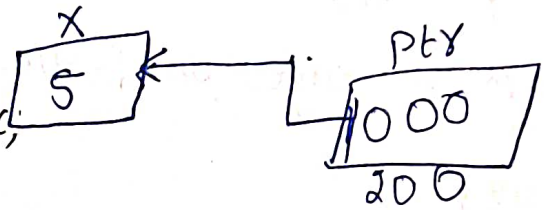
↳ Here data type refers to the type of the value that the pointer will point t

for example

int *ptr; ← points to integer val char *ptr; ← points to char valu

int x=5;
int *ptr;
ptr = &x;

} int x=5, *ptr=&x;



Value of operator/- indirection operator/dereference operater is an operator that is used to access the value stored at the location pointed by pointer

int x=5; *ptr = &x;
printf("%d", *ptr); O/P : 5

int x=10, *ptr=&x; *ptr=4;
printf("%d", *ptr);

⇒ Never apply the indirection operator to uninitialized pointer

Assinging value to an uninitilized pointer is dangerous

```
int *ptr;
*ptr = 1;
```

(segemanation fault) caused by program trying to read (or) write an illegal memory location

Pointer assignment:-

```
int i = 10;
int *p, *q;
q = &i;
q = p;   O/P:- 10,10
```

not same =
but *q & *p

```
int i=10; j=20; &p; *q;
p = &i; q = &j;
*q = *p;
printf(*q, *p); → 10,10
```

```
i=1  *p = &i; int *q;
q = p; *q = 5; print(p)
```

(segemanation fault) — functions left

Application of pointer to find largest and smallest element in array:-

```
int a[] = {23, 45, 6, 98}
int min, max;
min = max = a[0]
```

Returning pointers:-

```
int a[];
 int n = size of (a) / size of a[0];
 int * mid = findMid (a, n);
printf ("%d", *mid);
  return 0;
}
int * findMid (int a[], int n)
{
  return &a[n/2];
}
```

Word of Caution:- Never ever try to return the address of an automatic variable / local variable of the function.

For example:

```
int *fun()
{
  int i=10; //local variable
  return &i;
}
int main() {
    int *p = fun();
    printf ("%d", *p);
}
```

Imp:

```
int *p = &i;   { First statement is declaration and second is simple assignmen
p = &i;          statem why isn't in second statement p is preceed by *symb
```

Sol: In C, * Symbol has different meanings depending on the context in which it is used

At the time of declaration, * Symbol is not acting as an indirection operator

* Symbol in first statement tells the Compiler that p is pointer to an integer

But if we write *p = &i then it is wrong, because here * Symbol indicates the indirection operator and we can't assign the address to some Integer variable

Therefore in second statement there is no need of * Symbol infront of p.

It simply means we are assigning the address to a pointer

(2) O/P:

```
Void fun (const int *p)
{
  *p = 0;
}
int main(){
  Const int i = 10;
  fun(&i);
  return 0;
}
```

⇒ error

3) How to print the address of a variable?

⇒ Use %p as a format specifier in printf function

```
int i = 10;
int *p = &i;
print("%p", p);
```

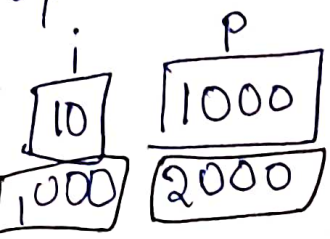
→ in hexadecimal

1) If i is a variable and p points to i, which of following are aliases of i

a) *p b) *&p c) &p d) *i e) *&i

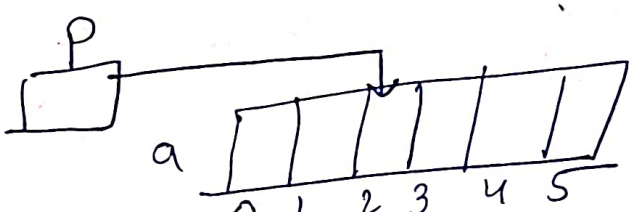
i = 10;

*p = &i; ⇒ *(1000) = 10



- i = 10
- *&p = *(&p) = *(2000) = 1000
- &p = &p = 2000
- (a) *i = *(10) doesn't make sense
- (e) *(&i) = *(1000) = 10

Pointer Arithemetic (addition) :-



Initially, if p points to a[i], then $p = p + j \equiv \&a[i+j]$

$p = \&a[i]$

$p = 1000$

$p = p + 1 \Rightarrow p = 1000 + 1 * 4$

$p = 1004$

H/w We have pointer p and array a

p contains address of 3rd element $p = \&a[2]$

$\Rightarrow p = p + 2 \Rightarrow p = \&a[4]$

Pointer arithmetic Subraction:-

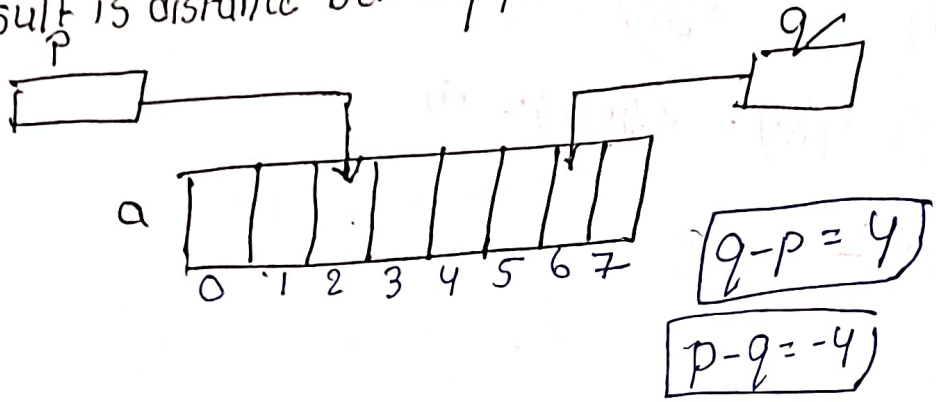
p = p-3 (viceversa) for addition

Initially, if p points to a[i], then $p = p - j \equiv \&a[i-j]$

$p = p - 3 \equiv p = 1012 - 3 \times 4$

Subracting one pointer from another pointer

Result is distance between pointers



q - p = 4

p - q = -4

Performing arithmetic on pointers which are not pointing to array element leads to undefined behaviour.

```
int main() {
    int i = 10;
    int *p = &i;
    printf("%d", *(p+3));
}
```

⇒ Different outputs everytime

⇒ If two pointers are pointing to different arrays then performing subraction between them leads to undefined behaviour

Post increament:-

a[] = {5,16,7,89}

*p = &a[0];

O/p:- 5

*(p++); → printf

again *p → O/p:- 16

pre increament *(++p) → O/p:- 16

And decreament operator is also same //

Comparing two pointers:-

→ Use relational operators (<, >, <=, >=) and equality operator (==, !=) to Compare pointers

→ only possible when both pointers point to same array.

→ output depends upon the relative positions of both pointers

Calculate the sum of elements of arrays using pointers:-

```
for (p = &a[0]; p <= n; p++)
  sum += *p;
```

Using array Name as pointer:-

Fact:- Name of an array can be used as a pointer to first element of array

for example:-

```
int main () {
    int a[5];
    *a = 10;
    printf ("%d", a[0]);
    *(a+1) = 20;
```

It is true that we can use array names as pointers but assigning a new address is not possible

```
int a[] = {1,  . . 3}
printf("%p", a+1);
```

↓ we are trying to assing address a[0] to arrayname (a)

Assigning a = a+1 ✗

printf("%p", a+1); ↓ Accessing it is possible

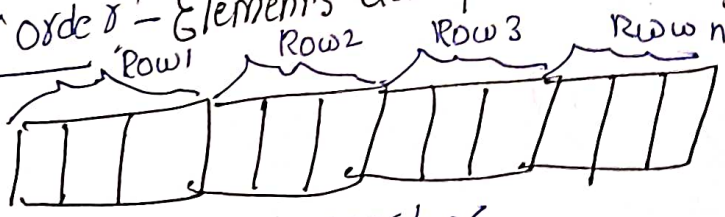
Reverse a series of numbers using pointers :- do it

Passing array name as an argument to a function :- do it (vid

Using pointer to 2d array :-

difference b/w row major and Column major order :-

Row major order :- Elements are stored row by row



Similary Column major order

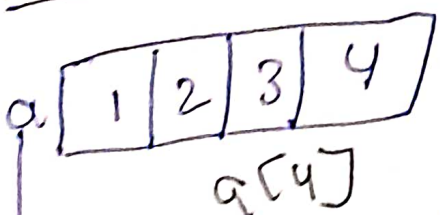
C follows row major order

	C0	C1
r0	1	2
r1	3	4

Using pointers :-

```
for (p = &a[0][0]; p <= &a[row-1][col-1]; p++)
    printf("%d", *p);
```

Address arthemetic of 2d :-

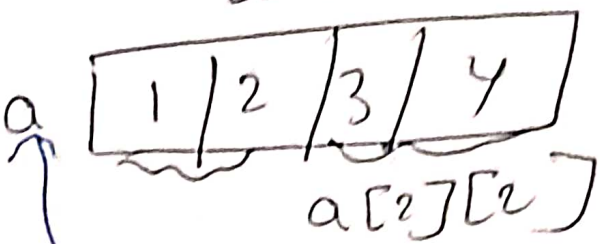


↳ pointer to first element of array

a ⇒ . . .

*a = 1

2d array



pointer to first 1D array

**a = 1

*(a+1)+1 → last element address

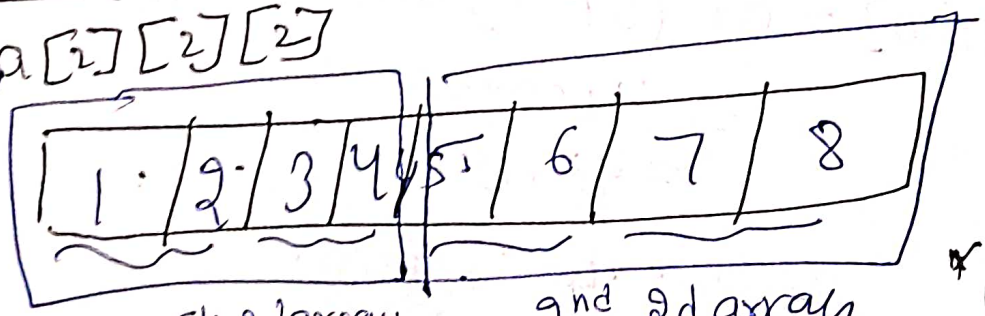
= *(*a) ⇒ *(*a[0][0]

= a[0][0] = 1

**(a+1) → second dimensional 1st element

3d array :-

int a[2][2][2]



↳ pointer to first 2d array

a+1 → pointer to second 2d array

**a = 1

***(a+1) = 5

(*(a a+1)) *(**a+1)

((a+1))+1

Pointers:

Char a[100][100]

Assuming that main memory is byte: array is stored starting from memory address 0, the address of a[40][50] is

formula: $\&a[i][j] = BA + [(i - lb_1) \times NC + (j - lb_2)] \times C$

Where BA = Base address of Whole 2d array

NC = no. of columns C = Size of data type of elements stored in array

a[0 ... 99][0, ... 99]

BA = 0 NC = 100

$\&a[40][50] = 0 + [(40-0) \times 100 + (50-0) \times 1$

$= 4000 + 50$

$= 4050$

x+3

200

2000 + 3*4 ⇒ 2012

2000 + 3*4*3 2036

2036

2036, 2036, 2036

LB1 = 0

Pointer pointing Entire array:-

```
int (*p)[5] = &a;
```

p → pointer to five integer elements

```
int (*p)[5] = &a;
printf("%d", **p);
```

a | 1 | | | | |
1000

O/p of following program:-

```
int a[][3] = {1,2,3,4,5,6};
int (*ptr)[3] = a;
printf("%d %d", (*ptr)[1], (*ptr)[2]);
++ptr;
printf("%d %d", (*ptr)[1], (*ptr)[2]);
```

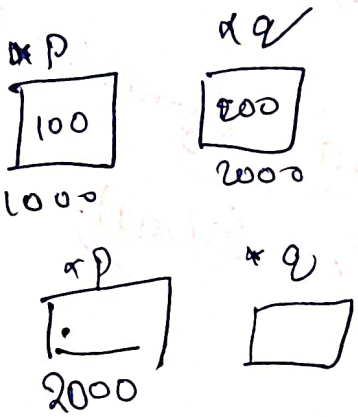
O/p:- 2, 3

O/p:- 4 5, 6

2 3 5 6

Gate question:-

```
void f(int *p, int *q)
{ p = q;
  *p = 2;
}
int i = 0, j = 1
int main(){
    f(&i, &j);
    printf(i, j);
    return 0;
}
```



O/P:- 0 2

$a = \{12, 7, 13, 4, 11, 6\}$

else if (*a % 2 == 0) return *a + f(a+1, n-1);

else return *a - f(a+1, n-1);

*a + f(a+1, n-1)

→ f(7, 5)

→ *a - f(13, 4)

→ *a - f(4, 3)

→ *a + f(11, 2)

⇒ *a + f(6, 1)

→ *a

⇒ 15

P

int c, *b, **a; → points to pointer

c = 4, b = &c a = &b → address of pointer

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

}

x = 4, *py = 4,

**pp3 = 4

int y, 3;

**pp3 = *p1 5;

3 = 5

*py += 2;

y = 6

5 + 7 + 7

19

Basics of recursion:-

Defination of Recursion:- is a process in which a function calls itself direc

or indirecty

for example;

```
int fun()
{
  ...
  fun();
}
```

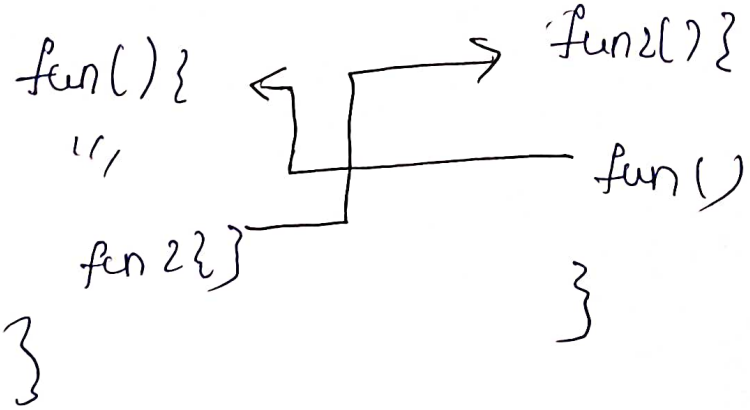
Idea:-

1) divide the problem into smaller sub-problems
2) Specify the base condition to stop the recursion

Types of recursion:- 1) direct recursion 2) Indirect recursion
3) Tail recursion 4) Non tail recursion.

① Direct recursion:- A function called direct recursive if it calls the same function again.

② Indirect recursion:- A function (let say fun) is called indirect recursive if it calls another function (fun2) and then fun2 calls fun directly (or) indirectly



③ Tail recursion:- A recursive function is said to be tail recursive if the recursive call is last thing done by the function. There is no need to keep

record of previous state

```
Void fun(int n) {
    if(n==0)
      return;
    else
      printf("%d",n);
      return fun(n-1)
}
int main() {
    fun(3);
    return 0;
}
```

reverse of Non-tail recursive

4) Non-tail recursive:- A recursive function is said to be non-tail recursive. if the recursive call is not the last thing done by function. After returning back, there is some something left to evaluate

non tail:

∴ fun(18)

```
else
    return 1 + fun(4)   2
                ↓       =3
           1 + fun(2)   1
           1 + fun(1)   1+0
```

Advantages and disadvantages:-

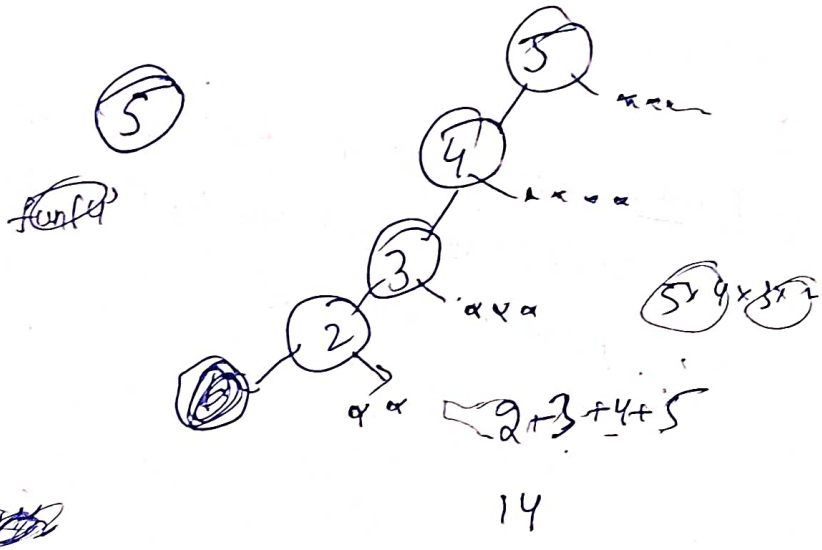
Every recursive program can be modeled into iterative program but recursive programs are more elegant and requires less lines of code.

disadvantage:- It requires more space (activation record stack space) than iterative programs.

```
get(n-1);
get(n-3);
```

5
4
3
2
1
0

```
int i = 0
if (n>1)
  fun1(n-1)
```



(3)

```
fun(5)
x=1, k;
(n==1)
for (k=1; k<n; ++k)
x = x + fun(k) + fun(n-k)
x = 1 + fun(k)
```

n=5

1 + fun(k) + fun(n-k)

fun(1)

1

1+1+fun(n-k)

5-1

1+1+fun(4)

Strings:-

String literal (or string Constant) is a sequence of characters enclosed within double quotes.

"Hello word" → in printf

%s → is a place holder

printf("%s", "- - - - - - - - - - - APJ. Adul kalam"); Error

printf("%s", "- - - - - - - - - - -\-APJ. Adul Kalam"); ✓

"\" → next line

and also " " → for another line.

Storing string literal:-

first argument to printf and scanf function is always a string literal

But, What we are actually passing to printf/scanf?

String literals are stored as an array of characters

Indicates the end of the string

Null character

Total 6 bytes of read only memory is allocated

In C. Complier treats a string literal as a pointer to first character.

So the printf (or) scanf, we are passing a pointer to first character of a string literal

Both printf and scanf functions expects a character pointer (char*) as an argument

Assigning string literal to a pointer:-

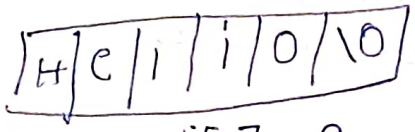
Char *ptr = "Hello World!";

⇒ ptr contains the address of first character of string literal

As writting "Hello" is equivalent to writting pointer to first character

Therefore, we can subscript it to get some character of string literal

"Hello"[1] is equalivalent to pointer to 'H'[1].



pointer to 'H'[1] = e

String literal cannot be modified. it causes undefined behaviour

Char *ptr = "Hello";

*ptr = 'M';

String literals are known as string constants

String literal and character constant are not same

"H" ≠ 'H'.

Represented by a pointer to character 'H'

Represented by an integer (72)

Declaring a string variable:-

A string variable is one dimensional array of characters that is capable of holding a string at a time

⇒ Extra ram for null character

char s[6] = "Hello";

Although it seems like "Hello" in above example is a string literal but it is not. When a string is assigned to a character array, then this character array is treated like other types of array we can modify it's characters

But we can modify characters

for(i=0; s[i] != '\0'; i++)

Writting string using printf and puts function:—

char *ptr = "HelloWorld";

printf("%s", ptr);

"%.ns" is used to print just apart of string where n is no. of characters

"%m.ns" → field of length

Write string using puts:— Puts() function is a function declared in <stdio.h> library and is used to write strings to output screen.

Also, puts() function automitically writes a new line character after writting the string to output screen

char *s = "Hello";

puts(s); Hello

puts(s); Hello

o/p:—

Reading strings using scanf and gets functions:—

Using scanf(). We can read a string into string variable (charracter array)

Char a[20];

Bcanf("%s", a); printf("%s", a);

Scanf() doesn't store the White Space characters in the string variable. It only reads characters other than Whitespaces and store them in the specified character array until it encounters a White-space character.

Reading string using gets:- In order to read an entire line of input, gets() function can be used.

```
char a[10];
gets(a)
printf("%s",a);
```

As scanf and gets function are risky to use. Hence it is advisable to design our own input function

We want our input function

→ It must continue to read the string even after facing whitespaces characters

→ It must stop reading the string after facing newline character

→ discard extra characters → return no. of characters it stores in the character array (9)

Video

getchar() function is used to read one character at a time from the user input. It returns an integer equivalent to Ascii code of the character

putchar() in C:-

Prototype int putchar(int ch)

Putchar accepts an integer argument (which represents a character it wants to display) and returns an integer representing the character written on screen

```
int ch;
for(ch = 'A'; ch <= 'Z'; ch++)
    put-char(ch);
    return 0;
}
```

→ write

Introduction to functions :-

function is basically a set of statements that takes inputs, perform some computation and produce output

Syntax:- Return-type function_name (set-of-inputs); → not neccassary

Return-type: type of data type

function_name: Name of fn

Why functions' - There are two imp reasons of Why we are using functions:

→ Reusability: Once the function is defined, it can be reused over and over again

→ Abstraction: If you are just using function in your program then you don't have to worry about how it works inside

Ex: Scanf fn:

Ex:→ Video

function declaration in C:-

When we declare a variable, we declare it's properties to the compiler

for ex:- int var; prop:- Name-var ; data type = int

Similary, function declaration (also called function prototype) means declaring the properties of funetion to compiler

for ex:- int fun(int, char);

Properties: Name of function : fun

Return type - int

No of parameters :- 2

Type of parameter 1 : int

" " " " 2 : char

⇒ It is not neccessary to put name of parameters in function prototype

for ex:- int fun(int var1, char var2);
not neccessary to mention

Is it always neccessary to declare the function before using it?

⇒ Not neccessary but it is prefered to dedare the fn before using it

function defination:- function defination consits of block of code which is capable of performing some specific task

⇒ Don't forget to mention the prototype of a function. In that prototype no need of mentioning ~~names~~ names of parameter

int add (int, int);

```
int main()
{
  int m=20, n=30, sum;
  sum=add(m,n);
  printf("%d",sum);
}
int add(int a, int b)
{
  return(a+b);
}
```

While calling a function you should not mention return type of function. Also you should not mention the data types of arguments

It is imp to mention both data type and name of parameters.

add int a=
int b=

Difference b/w an argument and a parameter?

Parameter:- Variable in declaration and definition of the function

Argument:- is actual value of the parameter that get passed to function

Parameter → formal parameter Argument → actual parameter
↓↓ received by a function ↓↓ passed to function

Call by value:- Here values of actual parameters will be copied to formal parameters and these two different parameter store values in different locations

```
int x=10, y=20;
fun(x,y);
printf("%od %od", x,y);
```

```
int fun(int x, int y)
{
  x=20;
  y=10;
}
```

O/p:- 10,20

Call by reference:- Here both actual and formal parameters refers to same memory location, therefore any changes made to formal parameters will get reflected to actual parameters.

Here instead passing values, we pass addresses

```
int x=10, y=20;
fun(&x, &y);
printf("%od %od", x,y)
```

```
int fun(int *ptr1, int *ptr2)
{ *ptr1 =20;
  *ptr 2=10;
}
```

O/p:- 20,10

Strings continuation:-

Solved problem:-

printf("%c", '\n'); → character ✓

printf("%c", "\n"); → string ✗

putchar('\n'); → character ✓

putchar("\n"); → string ✗

puts('\n'); → character ✗

puts("\n"); → string ✓

printf("%s", '\n'); ✗

printf("%s", "\n"); ✓

Introduction to C string library and Strcpy (string copy) function:-

There are some operations which we can perform on strings

Ex: Copy strings, concententate strings, select strings and so on ✓

<string.h> → library contains all the required fn's for performing string operations

So we just have to include header file #include <string.h> in our program

Strcpy function:-

Prototype:- char* strcpy (char *destination, const char* source)

↑ It isn't modified. That is why it is constant

Strcpy returns the pointer to the first character of the string which is copied in the destination. Hence if we use %s, then whole string will be printed on the screen

We can also chain together a series of strcpy calls

strcpy(str3, strcpy(str2, str1));
→ returns string

Caution:- In the call to strcpy(str1, str2) there is no way the strcpy will check whether the string pointed by str2 will fit in str1

If the length of the string pointed by str2 is greater than length of character array str1 then it will be an undefined behaviour

To avoid this we can use strncpy

strncpy(str2, str1, sizeof(str2));

O/P:- Characters mentioned upto str2 will be printed

strncpy will leave the string in str2 (destination) without a terminating Null character if the size of str1 (source) is equal to or greater than size of str2 (destination)

To avoid this:

After this line add

str2[sizeof(str2) - 1] = '\0';

Strlen (String length function) :-

→ determine the length of string

Prototype :- `size_t strlen (const char* str);`

- size_t → Unsigned integer type atleast 16 bits.
- not length of array

⇒ It doesn't count Null Character

Strcat (String Concatenate) function :-

prototype :- `char * strcat (char* str1, const char* str2);`

Strcat function appends the content of string str2 at the end of string str1

Caution :- An undefined behaviour can be observed if size of str1 isn't long enough to accomodate the additional characters of str2

Strncat is the safest version of Strcat :-

It appends the limited no. of characters specified by third argument passed to it

Note :- Strncat automatically adds NULL characters at end of resultant string.

```
Strncat (str1, str2, sizeof(str1) - strlen(str1) - 1);
```

- sizeof(str1) → Size of array
- strlen(str1) → size of string
- -1 → Creating null characters

Strcmp (string comparison) function:-

Prototype:- int strcmp (const char *.s1, const char *s2);

⇒ Compares two strings s1 & s2

⇒ Returns value:- Less than 0, if s1<s2
Greater than 0 if s1>s2
Equal to 0, if s1==s2

ASCII character set:-

* All upper case letters are less than all the lower case letters
⇒ Digits are less than letters (48-57)
⇒ Spaces are less than all printing characters (space character has the value 32 in ASCII set)

Array of strings:-

char fruits[][12] = {"2oranges", "2 Apples", "3bannanas", "1pineapple"};

⇒ So lot of memory get wasted as oranges etc sizes are less

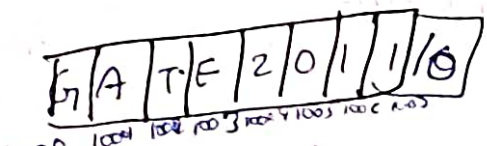
Array of pointers:- This is one dimensional array.

char *fruits[] = {.....};

Eg
```
Char p[20];
Char *s = "abcde";
Int length = strlen(s)
for(int i=0; i<length; i++)
    P(I) = s[length-i];
```
first charecters becomes null character

Printf function will print everything before the null character and will not see anything after null character.

```
Char C[] = "Gate 2011";
Char *p = C;  // address of first character
printf("%s", p + P[3] - P[1]);
```



↓

1000 + E - A

1000 + 4

1004

2011 ⇒ output

```
Void foo(char *a)
{ if(*a && *a != ' ')
  { foo(a+1);
    putchar(*a);
  }
}
```

ABCD EFGH\0

// means if a character pointed by a pointer 'a' is neither a null character and nor a blank character then continue else stop

Solve →

O/P - DCBA

Consider the program:-

```
Void fun1(Char *S1, Char *S2) {
    Char *temp;
    temp = S1;
    S1 = S2;
    S2 = temp;
}
```

→ Just address are changing locally

```
Void fun2 (Char **S1, Char **S2) {
    Char *tmp;
    tmp = *S1;
    *S1 = *S2;
    *S2 = tmp;
}
```

3000 4000

```
int main() {
    char *str1 = "Hi", *str2 = "BYE";
    fun1(str1, str2); printf("%s %s", str1, str2); → Hi BYE
    fun2(&str1, &str2); printf("%s %s", str1, str2); BYE
    return 0;
}
```

3000, 4000

HI

H	I	'\0'

H	i	\0

1000 1001 1002

1000

3000

B	Y	E	\0

2000 2001 2002 2003

2000

4000

```
int main()
{
Char *c = "GATECSIT2017";
 Char *p = c;
 printf("%d", (int) strlen(c+2[p] - 6[p] - 1));
 return 0;
}
```

*(2+p) *(6+p)

1000 T - I - 1

strlen(1010) → 17 ⇒ (2)

(int)(2) strlen returns a size_t type data therefore (int) will convert the size_t type to int type. This is called type casting

Structures in C:-

I want to store all the info about cars which are available in my garage.

Array has the capability to store more than one elements but they all must be of same type.

⇒ structure is a user defined data type that can be used to group elements of different types into single type.

Declaring structure variable:-

Example structure of car

```
struct {
    char *engine;
    char *fuel_type;
    int fuel_tank_cap;
    int seating_cap;
    float city_mileage;
} car1, car2;
```

⇒ Video

It is in global scope

With the help of (.) → dot structure we can access member of structure

```
int main()
{
    car1.engine = struct with car...
}
```

char

char engine[10];

Structure Types (Using structure tag):-

⇒ Video

If structure is in global scope ⊗ User define the values. But if structure is in local scope

User can't define the values inside in it

Need of Creating a type

Structure tag is used to identify a particular kind of structure

Using the typedef keyword

Syntax:- typedef exisiting-data-type new-data-type

typedef gives freedom to user by allowing them to create their own types

Video Useful for writting own data type instead of struct-

Initializing & Accessing the Structure Members:-

Not allowed

```
Struct abc {
        int p=23;
        int g=34;
    };
```

Wrong way of initialization

```
typedef abc {
        int p;
        int g;
    } abc;
int main ()
{
    abc x = {23,34};
}
```

Accessing members of structure:-

We can access members of structure Using dot(.) operator

Designated intilazation:- This allows structure members to be initialized in any order

Video

Array of Structure:- Instead of declaring multiple variables, we can also declare an array of structure in which each element of the array will represent a structure variable.

Video

Accessing members of structures Using structure pointer:-

```
Struct abc {
    int x;
    int y;
};
int main() {
    Struct abc a = {0,1};
    Struct abc *p = &a;
Print f ["%od %od", ptr->x, ptr->y);
```

ptr->x is equivalent to
(*ptr).x
(a).x)

Structure padding in C:-

How memory is allocated to structure members?

⇒ When an object of some structure type is declared then some contiguous block of memory will be allocated to structure members.

Structure padding:-

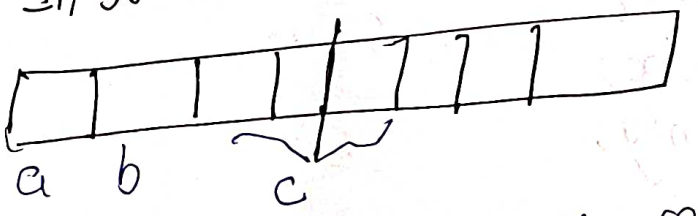
Processor doesn't read 1 byte at a time from memory

It reads 1 word at a time

If we have 32 bit processor then it means it can access 4 bytes at a time which means word size is 4 bytes. If 64 → 8 bytes at a time

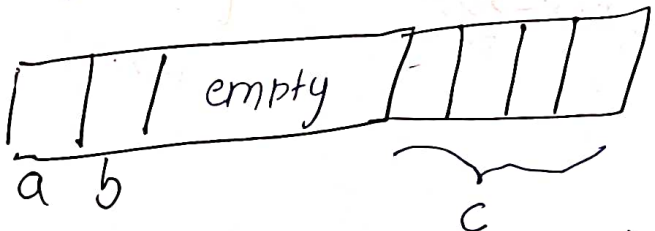
```
Struct abc {
    char a; // 1 byte
    char b; // 1 byte
    int c; // 4 byte.
} var;
```

In 32 bit architecture



It's an unnecessary wastage of CPU cycles

Then the padding concept come into picture:-



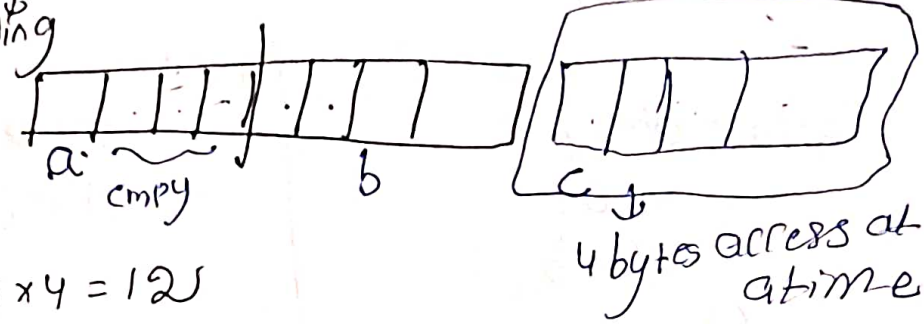
Total = 1 byte + 1 byte + 2 bytes + 4 bytes. = 8 bytes

If we change the order What happens :-

Example

Alignment Consideration

```
char a;
int b;
char c;
};
```



for 32 bit architecture 3 × 4 = 12

Structure packing in C :- Because of structure packing, size of the structure becomes more than size of actual structure.

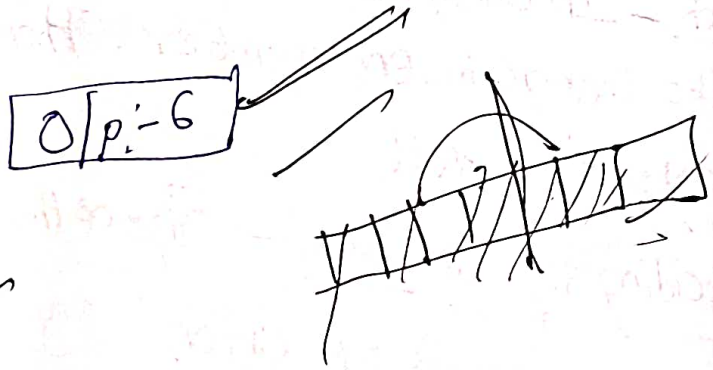
Due to this some memory will get wasted

We can avoid the wastage memory by simple writing #pragma pack(1)

#pragma is a special purpose directive used to turn on (or) off certain features

#pragma pack(1)

```
struct abc {
    char a;
    int b;
    char c;
} var;
int main() {
    printf("%d", sizeof(var));
}
```



Introduction to Unions:- Union is a user defined data type but unlike Structures, Union members share same memory location.

Example:

```
struct abc {
        int a;
        Char b:
    };
```

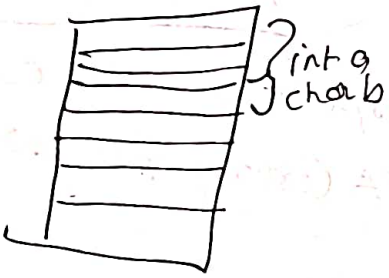
a's address ⇒ different
b's address ⇒

```
Union. abc {
    int a;
    char b;
};
```

a, b } address → both are same

memory location

Struct abc



Union abc

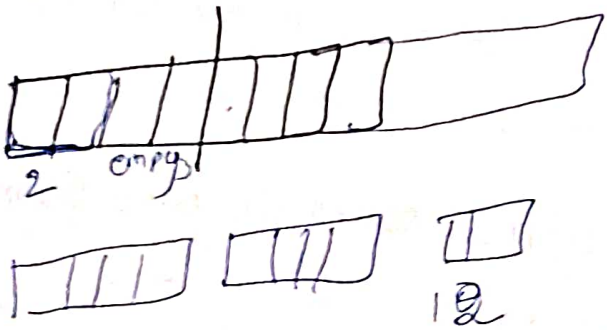
Fact:- In Union members will share same memory allocation if we make change in one member. Then it will be reflected to other member's as well

Deciding size of Union:- Size of the Union is taken acc to size of largest member of Union.

Accessing members Using pointers:—

We can access members of Union through pointers by using the arrow (->) operator

→ Video



Applications of Unions:—

Books: Title, Author, no. of pages, price

Shirts: - Color, size, design

price

→ Video

Application of Union to save memory space

post-b" Video

Enumeration:—

An enumeration is a user defined type which is used to assign names to integral constants because names are easier to handle in program

```
enum bool {false, true};
int main() {
    enum bool var;
    var = True;
    Print ("%d", var);
} return 0;
```

If we don't assign values to enum names then automatically compiler will assign values to them starting from 0

Need of Enumeration:-

Two imp reasons:-

→ Enums can be declared in local scope

→ Enums names are automatically initialized by the compiler

→ Two (or) more names can have same value.

```
int main() {
    enum point {x=0, y=0, z=0};
    printf("%d %d %d", x, y, z);
    return 0;
}
```

⇒ We can assign values in any order. All unassigned names will get values as value of previous name +1

```
int main() {
    enum point {y=2, x=34, t, z=0};
    printf("%d %d %d %d", x, y, z, t);
    return 0;
}
```

@ 34, 2, 0, 35

⇒ Only integral values are allowed

No float (or) nothing.

fact 4:- All enum constants must be unique in their scope.

Video

question:- The following structures are designed to store info about objects on a graphics screen.

struct point {int x, y;};

struct rectangle { struct point upper_left, lower_right;};

A point structure stored the x and y coordinates of a point on the screen.

A rectangle structures stores the coordinates of the upper left and lower right corners of the rectangle.

Write a function that accepts rectangle structure r as an argument and computes the area of r

Video (or) solve by own

(9) struct {

double a;

union {

char b[4];

double c;

int d;

} e;

char f[4];

} s;

20

{Assume that Complier leaves no holes b/w members.

Packing is considered over padding

8

16

4

4

4

8

4

3

64

Understanding Void pointer:- It is a pointer which has no associated data type with it. It can point to any data type and can be typecasted to any type.

Eg:

```
int main()
{
  int n=10;
  void *ptr=&n;
  printf("%d", *(int*)ptr);
  return 0;
}
```

We cannot dereference a void pointer

Use of Void pointer:- malloc and calloc fn returns a void pointer. Due to this reason, they can allocate a memory for any type of data

Null pointers:- Null pointers is a pointer that does not point to any memory location. It represents an invalid memory location. When a null value is assigned to a pointer, then pointer is considered as "NULL" pointer.

Ex:

```
int main() {
    int * ptr = NULL;
    return 0;
}
```

Uses

→ It is used to initialize a pointer when that pointer isn't assigned any valid memory address yet

→ Useful for handling errors when using malloc function

→ The value of NULL is 0. We can either use NULL or 0. But this 0 is written in context of pointers and is not equivalent to integer 0

→ Size of null pointer depends on computer size

→ It is good practice to initialize a pointer as NULL

Dangling pointers:- It is a pointer which points to some non-exisiting memory location.

Ex:

```
int main()
{ int * ptr = (int *) malloc (sizeof(int));
  ::
  free(ptr);
  return 0
```

still ptr contains address

To avoid dangling pointers | De initialise it to N

→ Video Example.

Wild pointers :- Wild pointers are also known as uninitialized pointers. These pointers usually point to some arbitrary memory location and may cause a program to crash or

Basis of dynamic memory allocation :-

Memory allocated during compile time is called static memo The memory allocated is fixed and can't be increased (or) decreased during run time

The process of allocating memory at the time of execution is called dynamic memory allocation

Malloc() → It is a built in function declared in header file <stdlib.h>

Syntax: (Void *) malloc (size_t size) → malloc fn simply allocates memory block acc to size specified in the heap and on success it returns a pointer pointing to first byte to allocated memory else returns NULL

size_t is defined as unsigned int

Why void pointer? - malloc doesn't have an idea of what it is pointing to. It merely allocates memory requested by user without knowing the type of data to be stored inside memory. Void pointer can be typecasted

Calloc (). - It is a function is used to dynamically allocate multiple blocks of memory.

It is different from malloc in 2 ways:-

Calloc() needs two arguments instead of one

Syntax- void * calloc (size_t n, size_t size)

no. of blocks

size of each block

* memory allocated by calloc is initialized to zero. It is not the case with malloc. Memory allocated by malloc with garbage value.

calloc → clear allocation

malloc → memory allocation

realloc() → function is used to change the size of memory block without losing old data

Syntax void *realloc (void *ptr, size_t newsize)

pointer previously allocated memory

free ()

Void free (ptr)

- Structures and functions -

Structure members as argument - Just like variable we can pass structure members as arguments to a fu

call by reference - Instead of passing copies of structur members, we can pass their address.

```
Struct charset {
        char s;
        int i;
};
Void key value (char *s, int *i) {
        scanf ("%c %d", s, i);
}
int main () {
    int j;
    struct charset cs;
    keyvalue (&cs.s, &cs.i);
    printf ("%c %d", cs.s, cs.i);
    return 0
}
```

Passing Structure Variable as an argument:-

Instead of passing structure mem individually. it is a good practise to pass a struct variable as argument-

Unlike arrays, name of structure variables are not pointers

```
struct point{
    int x;
    int y;
};
void print (struct point p){
    printf ("%d %d\n", p.x, p.y);
}
int main (){
    struct point p1 = {2 3, 4 5};
    struct point p2 = {56, 90};
    Print (p1);
    Print (p2);
    return 0;
}
```

Passing pointer to structure as an argument:-

If the size of structure is very large then passing the copy of whole structure is not efficent

Better pass address of structure

→ Use the arrow operator (->) to access the stru members inside the called function.

as last program but-

```
Void print (struct point *ptr) {
    printf ("%d %d\n", ptr->x, ptr->y);
}
```

Returning a structure variable from the functio

① changing values.

```
struct point p1 = {23, 45};
struct point p2 = {56, 90}
p1 = edit(p1);
p2 p2 = edit (p2)

struct point edit {struct point p} {
    (p.x)++;
    p.y = p.y + 5;
    return p;
}
```