C

1. Introduction of C

→ History of Computing → Binary format → Assembly language

Assembly — portable ✗

C — portable ✓, less lines of code

2. Features :-

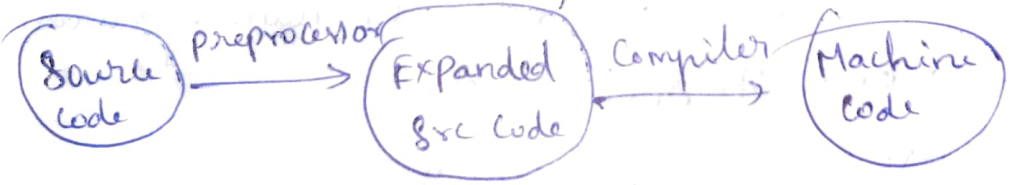
→ procedural programming

High Level → Less efforts

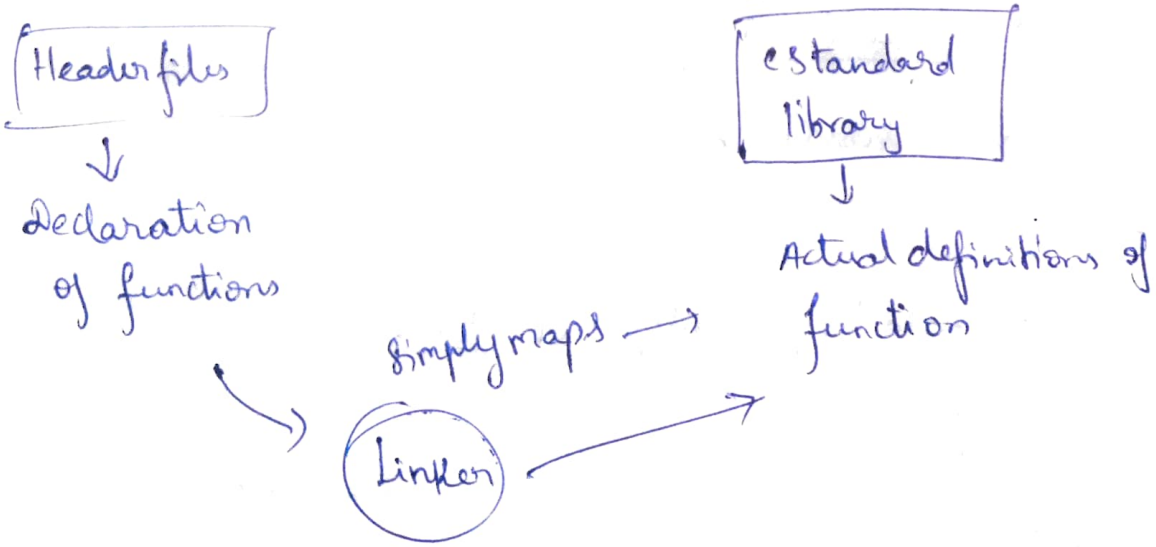
Low level → More efforts ← need more details for users.

Degree of Abstraction

→ Middle level language
→ Direct access to memory through pointers
→ Bit manipulation
→ Writing assembly code within C code
→ Wide variety of built in functions, standard libraries and header files.



stdio.h → Contains declarations of printf, Scanf etc.



3. Introduction to variables

Declaration → announcing properties of variable

properties → 1. Size
2. Name of variable

Definition: → allocating memory to a variable

Most of time declaration and definition will be done at the same time.

int var;
↳ Data type: how much space a variable is going to occupy in memory. (Depends on memory)

4. Naming Conventions to variable.

Rule 1:- Don't start variable name with digit.

2:- Begining with underscore is valid but not recommended.

3:- Case sensitive. upper case letters are different from Lower case letter.

4. Special characters not allowed in name of variable.

5. Blanks or white spaces not allowed.

6. Don't use keywords to name your variables.

5. Basic output function - printf:-

%d → placeholder for variable.
↳ decimal

6. Fundamental data type - Integer (part1).

2 bytes, 4 bytes

'sizeof' is a unary operator and not a function.

Range:- upper and lower limit of some set of data

Decimal Number System:- → Base 10 number system.

Range: 0 to 9

10^2 10^1 10^0
5 6 8

Binary Number System:-

Range: 0 to 1

2^3 2^2 2^1 2^0
1 1 0 0

4 bit data:-

2^3 2^2 2^1 2^0
0 0 0 0 min value = 0

1 1 1 1 Max Value = 15 → formula $2^n - 1$

Range of 4 bit data: 0000 to 1111

0 15

Range of integer: unsigned range: 0 to 65535 $(2^n - 1)$

(2 bytes) 16bits signed range: -32768 to +32767

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

(4bytes) 32bits unsigned range: 0 to 4294967295 $(2^n - 1)$

signed range: -2147483648 to +2147483647

7. part-2)

→ modifiers (short, long, signed, unsigned)

→ if int → 4bytes short int 2bytes

long int 8 bytes

sizeof(short) <= sizeof(int) <= sizeof(long)

By default int some_variable_name; is signed integer variable

unsigned int some_variable_name; allows only positive values.

<limit.h>

%u → printing unsigned integer values

<limits.h> → INT_MIN %d
INT_MAX %d] default signed
UINT_MAX %u → unsigned int
SHRT_MIN %d
SHRT_MAX %d] → short int
USHRT_MAX %u → short unsigned int

Note:-

1. %ld → print "long integer" equivalent to "signed long integer".
2. %lu → print "unsigned long integer".
3. %lld → print "long long integer"
4. %llu → print "unsigned long long integer".

8. Exceeding the valid range of data types

3 bit unsigned range exceeding condition:-

2^2	2^1	2^0	
0	0	0	→ 0
0	0	1	→ 1
0	1	0	→ 2
0	1	1	→ 3
1	0	0	→ 4
1	0	1	→ 5
1	1	0	→ 6
1	1	1	→ 7

value 8 ← (1) [0 0 0] → 0
(1 → fourth bit)

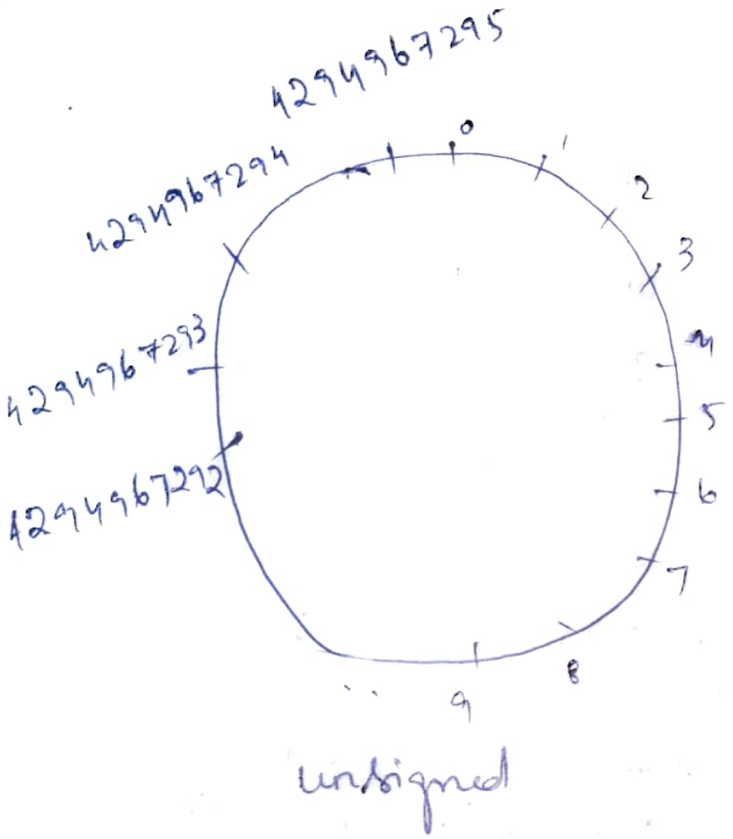
9 ⇒ 1 [0 0 1] → 1
10 ⇒ 1 [0 1 0] → 2
11 ⇒ 1 [0 1 1] → 3

This Mod 8 (or) Mod 2^3

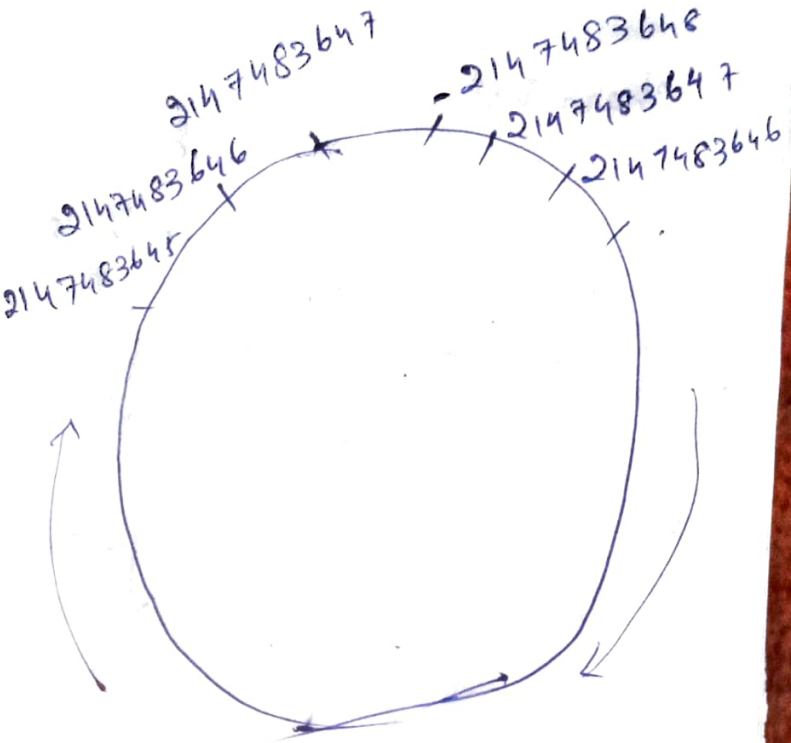
1 Mod 8 = 1
2 Mod 8 = 2
8 Mod 8 = 0
9 Mod 8 = 1

If we try to exceed range (> 7), we will back to (0 → 7)

For 32 bit unsigned data → Mod 2^{32}
For n bit unsigned data ⇒ Mod 2^n
Clock → simply Mod 12 function.



unsigned



signed

Eg:-
```
#include "stdio.h"
int main()
{
int var = 2147483648;
printf var;
}
```

o/p:-
-2147483648

exceeding signed max value (2147483647)

Eg2:
```
#include "stdio.h"
int main()
{
int var = -2147483649
print var;

}
```

o/p:-
2147483647

9. Character :-

ASCII

Size :-

1 byte = 8 bits

Range :-

unsigned : 0 to 255

signed : -128 to +127

```
char c = 65;
printf("%c", c);
```

o/p :- A

$-2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

-128 = 1 0 0 0 0 0 0 0] → same

+128 = 1 0 0 0 0 0 0 0

-127 = 1 0 0 0 0 0 0 1] same

129 = 1 0 0 0 0 0 0 1

-1 = 1 1 1 1 1 1 1 1] same.

255 = 1 1 1 1 1 1 1 1

```
int main()
{
char var = 128;
printf("%c", var);
}
```

```
int main()
{
char var = -128;
printf("%c", var);
}
```

10. Float, Double & long double :-

for representing fractional numbers.

Float → 4 bytes

double → 8 bytes

long double → 12 bytes

Fixedpoint representation

- 9 . 9 9

sign | Integer | Fraction

min value = -9.99

max value = +9.99

Floating point representation :-

- + 9 9 9

sign | exponent | mantissa

formula :- $(0.M) \times Base^{Exp}$

min value = $-(0.9) \times 10^{+9}$

max value = $+(0.9) \times 10^{+9}$

decimal point is not fixed.

```
float var1 = 3.14159265358979 32;
double var2 = 3.14159265358979 32;
long double var3 = 3.14159265358979 32 13 46
printf (".1..16f\n", var1)
printf (".1..16f\n", var2)
printf (".1..21Lf\n", var3)
```

(Capital)

o/p:-

→ 3.14159274101257 32

1 2 3 ... 7 upto 7 only float precise

→ 3.14159265358979 31

16 digits only double

→ 3.14592653589793213359

18 digits only long double

II) Questions:-

① int main()
{
printf(".1.d", printf(".1.s", "Helloworld!"));
return 0;
}

o/p

Hello world! 12

→ .1.s used to print "string of characters".

→ printf not only print the contents of screen, also returns the number of characters that it succesfully prints on the screen. \n (Including newline \n)

② int main()
{
printf (".1.s\n", "HELLO");
printf (".1.0s", "Hello");
}

→ HELLO
Hello

3) int main {
char c = 255;
c = c+10;
printf("%d", c);
}

o/p:- 9

exceeding
255 0

12) Scope of Variables - Local vs Global :-

Defining Scope:- Scope = Lifetime

Area under which variable is applicable (or) alive.

Strict definition:- A block (or) region where variable is declared, defined and used when block (or) region ends, variable is automatically destroyed.

```
int var=10; ——> Global
int main()
{
  int var=3; ——> Local
  printf("%d\n", var);
  fun();
  return 0;
}
int fun()
{
  printf("%d", var);
}
```

o/p:

3
10

13) Modifiers - auto and extern

Auto modifier:-

→ Auto means automatic

→ Variables declared inside a Scope by default are automatic variables.

Syntax: auto int some_variable_name;

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

Equal

```
#include "stdio.h"
int main()
{  auto int var;
   return 0;
}
```

→ If auto variable is not initialized, by default it will be initialized with some garbage (random) value.

→ on other hand global variable by default initialized too.

Extern modifier :-

extern int var

↓

declaration

→ Extern is short name for external.

→ used when particular file needs to access variable from another file.

→ multiple declarations are allowed but not definition.

⇒

① When we write extern some_data_type some_variable_name; no memory is allocated. only property of variable is announced.

② Multiple declarations of extern variables is allowed within the file. This is not case with automatic variable.

③ Extern variable ~~goes~~ says to compiler "go outside from my scope and you will find definition of variable that i declared"

④ Compiler believes that whatever the extern variable said is true and produce no error. Linker throws an error when finds no such variable exist.

⑤ when an extern variable is initialized, then memory is allocated and it will be considered defined.

(14) Modifier - Register

Syntax:- Register datatype Variable name

register int var;

→ Register Keyword hints the Compiler to store a variable in register memory.

→ This is done because access time reduces greatly for most frequently referred variables.

→ This is the choice of Compiler whether it's puts the given variable in register (or) not

→ Usually Compiler themselves do the necessary optimizations.

(15) Static

1. Static variable remains in memory even if it is declared within block on other hand automatic variable is destroyed after the completion of function in which it was declared.
2. Static variable if declared outside the scope of any function will act like global variable but only within file in which it is declared.
3. You can only assign a constant literal (or value) to a static variable.

(16) Constants (P1) → Types 1. Constant 2. #define

Constant → Something that never change.
once defined cannot be modified later in Code.

Using #define:- → also called Macro

#define (Name) value

→ Job of preprocessor (not compiler) to replace Name with value.

→ Don't add semicolon at end
→ choosing Capital letters for NAME is good practice
→ whatever inside double quotes " " won't get replaced.

Eg:- #define Value 89

```
int main()
{
printf("Value is %d", Value);
return 0;
}
```

→ not replaced by

→ We can use macros like functions.
→ We can write multiple line using \ ← Important
→ First expansion then evaluation
→ Some predefined macros like __DATE__, __TIME__ can print current date and time. [%s, %s].

(17) Constants (P2)

Using CONST :-

Syntax:- Const Some-data-type some-variable-name

Eg:- Const int var = 67;

When above statement is placed at global, No functions do not access to change the value

(18) Questions:-

①
```
int main() {
int var = 052;
printf("%d", var);
return 0;
}
```

→ Not 52; it will be octal (because '0' is placed)

052
↳ 2×8⁰ ⇒ 2
↳ 5×8¹ ⇒ 40
42

Ans:- 42

(2) #define STRING "%s\n"
#define NESO "WELCome to Neso Academy!"
int main() {
printf(String, NESO); Ans: WELCome to Neso Academy!
return 0;
}
Not as "WELCome to Neso Academy!".

(19) Basic input function:-

Scanf - stands for Scan Formatted string
Accept character, string and numeric data from the user
Using standard input - Keyboard.
%c → accept input of character type
%d → accept input of integer type
%s → accept string
Why &:- → while scanning ip, Scanf needs to store that input data somewhere.
→ To store this i/p data, Scanf needs to know the memory location of variable.

(20) Questions:-

(1) %x → print small hex
%X → print capital hex

(2) static int i;
static int i=27;
static int i; => 0
int main() not 27.
{
static int i;
printf("%d", i);
}

(1) (2) Sometimes scanf function doesn't read input, why? (see Bank.c structure code)-

Sometimes Scanf doesn't work. It simply doesn't wait for user Input !, why? (Eg:- See Bank.c)

Sometimes when we enter the input, after that user press 'ENTER'. Scanf simply reads the input that provided before 'ENTER'. After that 'ENTER' key is valued and stored in the input buffer. It is not cleared. So when we call 'Scanf' next time like Scanf("%c",&Char), Scanf will read the input buffer. But it reads the 'ENTER' in input buffer that actually previously stored in it. So now it skips the user input ((ie) It doesn't wait for user input).

Issue:- Input buffer is not cleared.

Solution:- Scanf(" %c",&Char)
↳ provide white space.

(or)

getchar() → before using scanf.

6) Introduction to operators:-

Logical → &&, ||, !

Bitwise → &, ^, |, ~, >>, <<

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

Other operators → ?: & * sizeof()

7) Arithmetic operators:-

↳ All are binary operators

precedence ↓	operators	Associativity
Highest	*, /, %	Left to right
Lowest	+, -	Left to right

8) Increment & decrement operator:-

Increment operator:- - Increment value of variable by one.

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

a=6

Decrement operator:- - Decrement value of variable by one.

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

a=4

→ Both are unary operators - because they are applied on single operand

a++ ✓ a++a; ✗

Pre-Increment operator:- ++a post-increment operator a++

pre-decrement operator:- --a post-decrement operator a--

→ rvalue (right value (constant))

→ You cannot use 'rvalue' before or after increment/decrement operator.

Eg:- (a+b)++; error

++(a+b); error

error:- lvalue required as increment operand.

lvalue (left value) :-

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

- any assignment statement, "lvalue" must have the capability to hold the data
- lvalue must be a variable because they have capability to store the data.
- lvalue cannot be 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 a constant expression (or) value.
- Expressions like a+b will return some constant value.

(a+b)++; → error

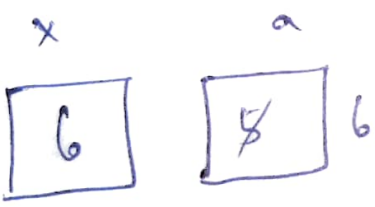
++(a+b) → error

error: lvalue required as increment operand:-

↳ Compiler is executing a variable as an increment operand but we are providing an expression (a+b) which does not have capability to store the data.

Pre-Increment:-

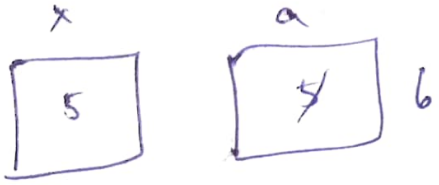
```
x = ++a;
```



x = 6

Post-Increment

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



x = 5.

(24) Lexical analysis → See video.

(25) Relational operators:-

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

All relational operators will return either True (or) false

(26) Logical operators:- &&, ||, !=

Concept of short circuit in logical operators:-

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

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

Eg:- of &&:-

```
#include <stdio.h>
int main()
{
int a=5,b=3;
int incr;
incr=(a>b)&&(b++);
printf("%d\n",incr);
printf("%d",b);
return 0;
}
```

(1) → True → Going to evaluate

o/p:- 1
4

of &&:-

```
#include <stdio.h>
int main()
{
int a=5,b=3;
int incr;
incr=(a<b)&&(b++);
printf("%d\n",incr);
printf("%d",b);
return 0;
}
```

→ false → So this part will not be evaluated

o/p: 0
3

Eg:- of ||:-

Simply means if there is Condition ~~anywhere in expression~~ that returns true, then rest of

```
#include <stdio.h>
int main()
{
int a=5, b=3;
int incr;
incr=(a>b)||(b++);
printf("%d\n",incr);
printf("%d",b);
return 0;
}
```

(2) → true → so this will not be evaluated.

o/p:- 1
3

(27) Bitwise operators:-

AND:- 7 → 0 1 1 1

4 → 0 1 0 0 &

4 ← 0 1 0 0

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

OR:- 7 → 0 1 1 1

4 → 0 1 0 0 |

7 ← 0 1 1 1

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

NOT ~:- 7 → 0 1 1 1

~

8 ← 1 0 0 0

A	~A
0	1
1	0

Eg:
```
#include <stdio.h>
int main()
{
char x=1, y=2; // x=1 (00000001), y=2(00000010)
if(x&y)        // 1&2 = 0(00000000)
printf("Hi\n");
if(x&&y)       // 1&&2 = True&&True = True = 1
 printf("Hi\n");
return 0;
}
```

(28) Left shift operator:- first operand << second operand

How left shift works?

- first operand → whose bits gets left shifted
- second operand → Decides no. of places to shift the bits

① Var << 1

↓

Left shift by position

Var = 3

3 = 0000 0011

↓

0000 011_

↓

6 = 0000 0110 ← Trailing position filled with zero

② Left shifing is Equivalent to multiplication by 2^rightoperand.

Eg 1: Var = 3;
Var << 1; o/p:- 6 [3 × 2^1]

Eg 2: Var = 3;
Var << 4; o/p:- 48 [3 × 2^4]

(29) Right shift operator:

① Var >> 1

↓

Right shift by position

Var = 3

3 = 0000 0011

↓

_0000001

↓

= 00000001

↓

Leading position filled with zero

② Right shifting is Equivalent to division by $2^{\text{right operand}}$

Eg:- Var = 3
Var >> 1;
o/p:- 1 [3/2¹]

Eg:- Var = 32
Var >> 4
o/p:- 2 [32/2⁴]

③ Bitwise XOR operator:-

Inclusive OR

* Either A is 1 (or) B is 1 or Both are 1, then the output is 1.
* Including Both

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

X - OR

Exclusive OR

* Either A is 1 (or) B is 1 then output is 1 but when both A and B are 1 then output is 0.
* Excluding both

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	0

7 → 0 1 1 0
4 → ∧ 0 1 0 0
3 ← 0 0 1 1

④ Assignment operator:-

-> Values to variable Can be assigned using assignment operator.
-> Requires two values - (L values and R-values)
-> This operator Copies R-Value to L-Value

Eg:- a = 5 (-> assignment)
a: L Value, 5: R Value

+= first addition then assignment
-=, *=, /=, %=, <<=, >>=, &=, |=, ^=

Eg:- ~~a++~~ a+=1
a = a+1

⇓
Shorthand assignment operators.

(32) Conditional operators in C :- ? :

```
char result;
int marks;
if (marks > 33)
{
 result = 'p';
}
else
{
result= 'f';
}
```

⇒

```
Char result;
int marks;
result = (marks > 33) ? 'p' : 'f';
```

→ It is the only Ternary operator available in list of operators in C.

→ As in `Expression 1 ? Expression 2 : Expression 3`, expression 1 is the boolean expression. If we simply write '0' instead of some boolean expression than that simply means FALSE and therefore Expression 3 will get evaluated.

Eg:-

```
int result;
result = 0 ? 2 : 1
```

result

1

(33) Comma (,) operator :-

① Comma operator can be used as "separator".

Eg:-

```
int a = 3, b = 4, C = 8;
```

⇓

Multiple definitions in Single line

≡

```
int a = 3;
int b = 4;
int C = 8;
```

② Comma operator can be used as an "operator".

Eg1:- `int a = (3,4,8);`
`printf("%d",a);`

⇒ Comma operator returns the rightmost operand in the expression and it simply evaluates the rest of operands and finally reject them.

o/p:- 8

Eg2:- `int var = (printf("%s\n","HELLO!"),5);` → This value will be returned to var after evaluating the first operand
`printf("%d",var);` → It will simply not rejected. first evaluated and then rejected.

o/p:- HELLO!
5

③ It is having least precedence among all the operators available in C.

Eg1:-
```
int a;
a=3,4,8;
printf("%d",a);
```
≡
```
int a;
(a=3),4,8;
printf("%d",a);
```

o/p:- 3 :- Because '=' having high precedence than ','.

Eg2:-
```
int a=3,4,8;
printf("%d",a);
```
o/p:- error

≡
```
int a=3; int 4; int 8;
```
↓ error

Note:- Here Comma is behaving like a separator

⇒ Comma acts like a separator within function calls and definitions, variable declarations and enum declarations.

Eg 3:- int a;
a = (3, 4, 8);
printf("%d", a);

=

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

O/p :- 8

→ Bracket has the highest precedence than any other operator.

(1)
(34) precedence and associativity :-
(35) operators in C (solved ins)
(36) operators in C (solved)
(37) C programming
(38) If-else
(39) Switch:
(40) for and while loop
(41) do-while
(42) Break, Continue
(43) Conditional and loops
(44) " "
(45) " "
(46) " "
(47) " "
(48) Pyramid of stars
(49) Palindrome
(50) Armstrong number
(51) Strong number

(52) prime number
(53) Add 2 numbers without plus
(54) " " halfadder
(55) fibonacci
(56) floyd's Triangle
(57) Binary to Decimal
(58) power of integer
(59) leap year
(60) check if it perfect number
(61)

(61) function :-

Two reasons:-

(1) Reusability :-

(2) Abstraction :- Eg :- scanf function. If we are using function in our program then we don't have to worry about how it works inside.

(62) function declaration :-

- It is needed when before use it.
- It is not ~~defined~~ needed if we ~~use~~ define before it use.

(63) function definition :-

Parameters :- It is a variable in declaration and definition of the function. (or) parameters are passed to functions

Argument :- It is the actual value of parameter that gets passed to the function. (or) parameters received by function.

Note :- parameter is also called as formal parameter and argument is also called as Actual parameter.

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

int add (int a, int b)
{
    return (a+b);
}
```

Arguments (or) Actual parameters

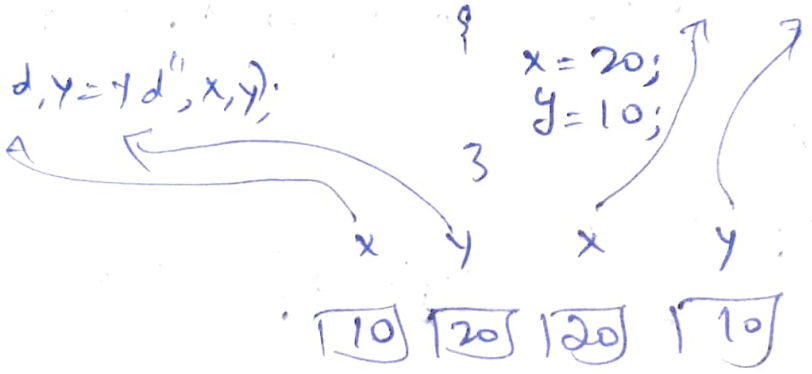
parameters (or) formal parameters.

(b) Call by value :-

Here values of actual parameters will be copied to formal parameters and two different parameters store value in the different locations.

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

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

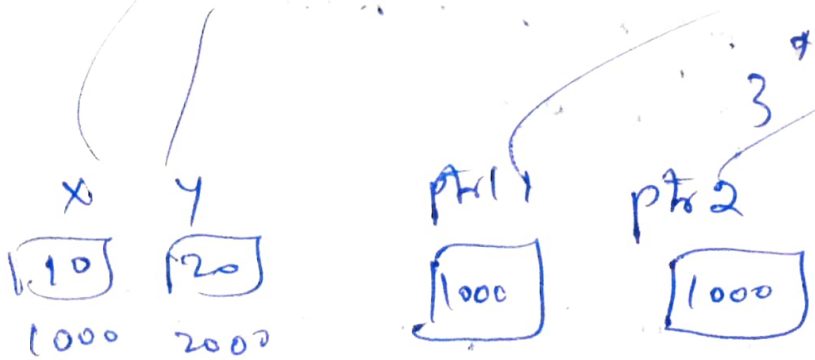


Call by reference :-

Here both actual and formal parameters refers to same memory location. Therefore any changes made to the formal parameters will get reflected to the actual parameters.

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

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



(65) Static functions:-

→ In C, functions are global by default.

→ This means if we want to access the function outside from the file where it is declared, we can allow it easily.

→ Now if we want to restrict this access, then we make our function static by simply putting a keyword "static" in front of the function.

→ Static functions are restricted to files where they are declared.

→ Reuse of the same function in another file is possible.

(66) Static and Dynamic Scoping:-

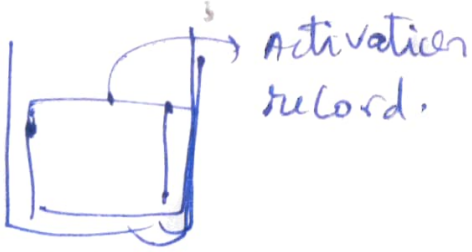
Stack:-

→ Stack is a container (or memory segment) which holds some data.

→ Data is retrieved Last In First Out (LIFO) fashion.

→ Two operations:- push and pop.

Stack:-



Stack / Call stack

Activation Record:- It is a portion of stack which is generally composed of

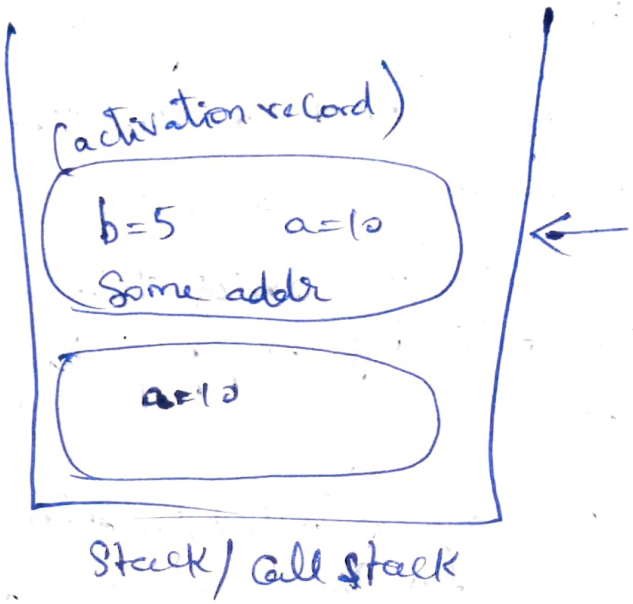
(1) Locals (local variables) of the Callee.

(2) Return address to the Callee.

(3) Parameters of the Callee

Eg:

```
int main()
{
 int a=10;
  a=fun1(a);
 printf("%d",a);
 }
int fun1(int a)
{
  int b=5;
    b=b+a;
   return b;
}
```



(67) Why scoping?

It is necessary if you want to reuse variable names.

Example:

```
int fun1()
{
  int a=10;
}
int fun2()
{
  int a=40;
}
```

Static Scoping:

Definition of variable is resolved by searching its Containing block (or) function. If that fails, then Searching the outer block and so on.

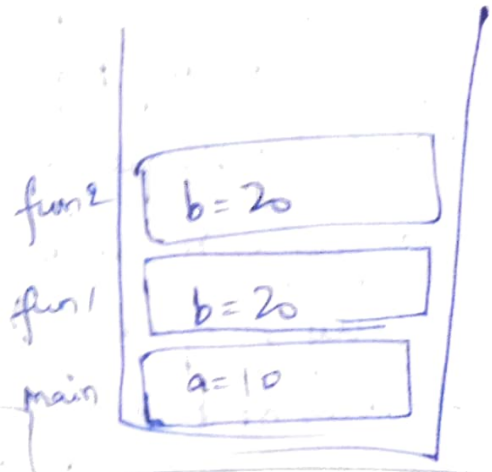
```
int a=10,b=20;
int fun()
{
  int a=5;
  {
    int c;
    c=b|a;
    printf("%d",c);
  }
}
```

o/p:- 4

(67)

Example:-

```
int fun1(int);
int fun2(int);
int a=5;
int main()
{
int a=10;
 a=fun1(a);
 printf("%d",a);
}
int fun1(int b)
{
 b=b+10;
 b=fun2(b);
 return b;
}
int fun2(int b)
{
 int c;
 c=a+b;
 return c;
}
```



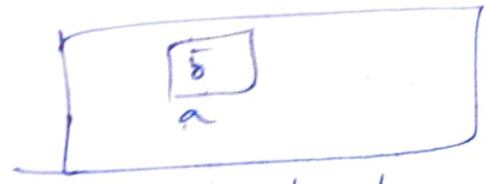
Call stack

(70)

Static Scoping:

In fun2, a is not there. So comes out of scope and found (a=5) in Initialized data segment.

o/p = 25



Initialized Data Segment

Dynamic Scoping:

In fun2, a is not there. So it goes to fun1.
In fun1 also, a is not there. So again goes to main. In main, it finds a=10. So o/p=30.
Here it will not choose a=5, because it is dynamic Scoping

(71) Dynamic scoping:- Refer previous page for example.

In dynamic scoping, definition of variable is resolved by searching its containing block and if not found, then searching its calling function and if still not found then function which called that calling function will be searched and so on.

(72):-

→ In most of the prog. language, static scoping is followed instead of dynamic scoping.

→ Languages including Algol, Pascal, C, Haskell, Schame etc., are statically scoped.

→ Some languages including SNOBOL, APL, Lisp etc., are dynamically scoped.

→ As C follows static scoping, therefore it is not possible to see programmatically, how dynamic scoping works in C.

(73) → Static and Dynamic :-(on1):-

(74) - Static and Dynamic (on2):-

(75) - Recursion in C :-

It is a process in which a function calls itself directly (or) indirectly

(76) → How to write recursive functions:-

```
fact()
{
    if() } → base condition
    {
    }
    else } → recursive procedure
    {
    }
}
```

→ ① Divide the problem into smaller Sub problems

Eg: Calculate fact(4)

Fact(1) = 1

Fact(2) = 2 × fact(1)

Fact(3) = 3 × Fact(2)

Fact(4) = 4 × Fact(3)

Generally $\boxed{Fact(n) = n * Fact(n-1)}$ ⇒ recursive procedure

② Specify the base condition to stop the recursion:-

Base Condition is the one which doesn't require to call the Same function again and it helps in stopping the recursion.

Eg: Fact(1) = 1 ⇒ Chosen as base Condition.

Types of Recursion:-

(1) Direct recursion:-

A function is called direct recursive if it calls the same function again.

Structure of Direct recursion:-

```
fun()
{
// some code
fun();
// some code
}
```

(ii) Indirect recursion:-

A function (let say fun) is called indirect recursive if it calls another function (let say fun2) and then fun2 calls fun directly (or) indirectly.

WAP to print numbers from 1 to 10 in such a way that when number is odd, add 1 and when number is even, subtract 1.

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

```
void odd();
void even();
int n=1;
void odd() {
if(n<=10) {
printf("%d", n+1);
n++;
even();
}
return;
}
```

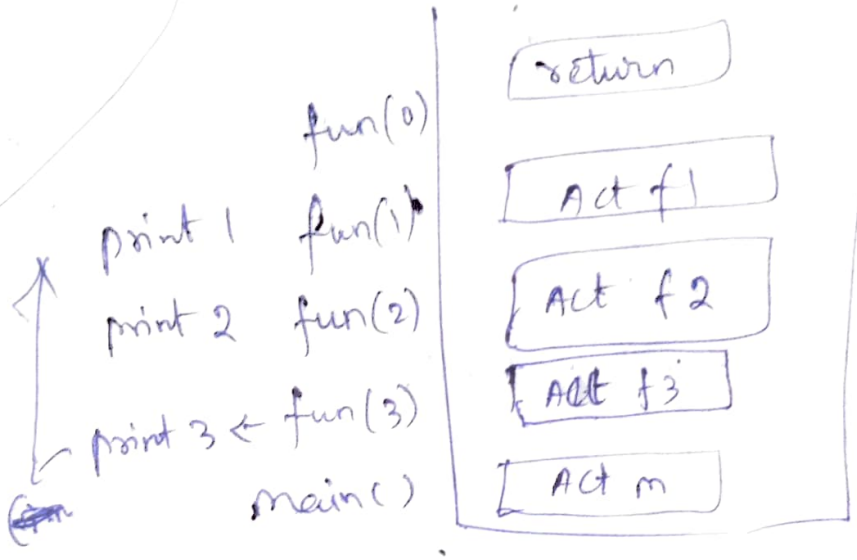
```
void even() {
if (n<=10) {
printf("%d", n-1);
n++;
odd();
}
return;
}

int main() {
odd();
}
```

78 ~~iii) A Types:~~

(iii) Tail recursive:- A recursive function is said to be tail recursive if the recursive call is the last thing done by the function. There is no need to keep record the previous state.

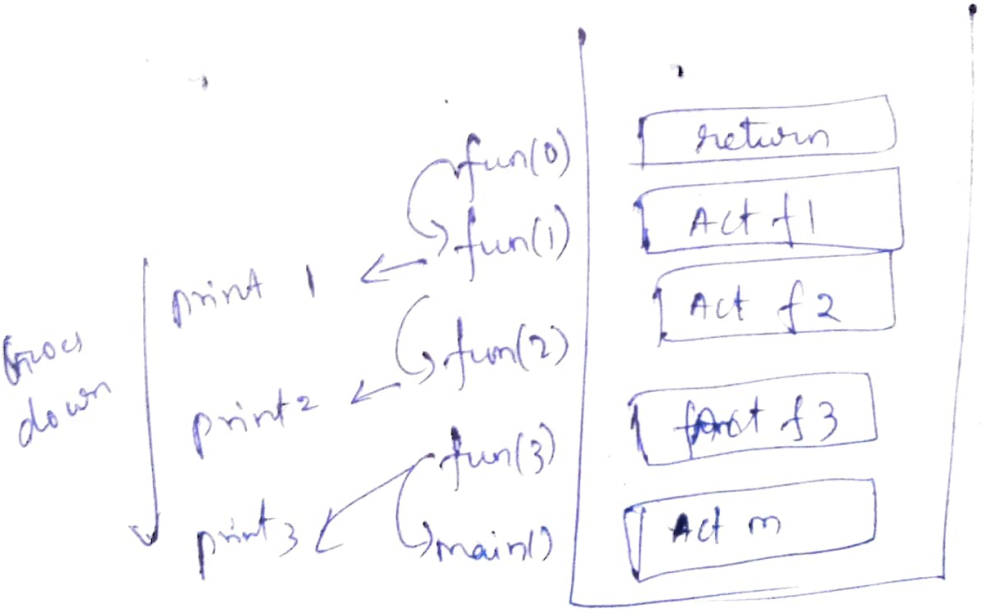
```
void fun (int n){
if (n==0)
  return
  else
   printf ("%d",n);
  return fun(n-1);
}
int main(){
fun(3);
return 0;
}
```



o/p:- 3 2 1

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

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



o/p:- 1, 2, 3

79. Advantage of Recursion :

Iterative :

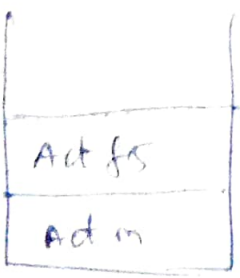
```
int fact(int n){
int res=1;
while(n!=0){
  res=res*n;
  n--;
}
return res;
int main(){
printf("%d", fact(5));
return 0;
}
```

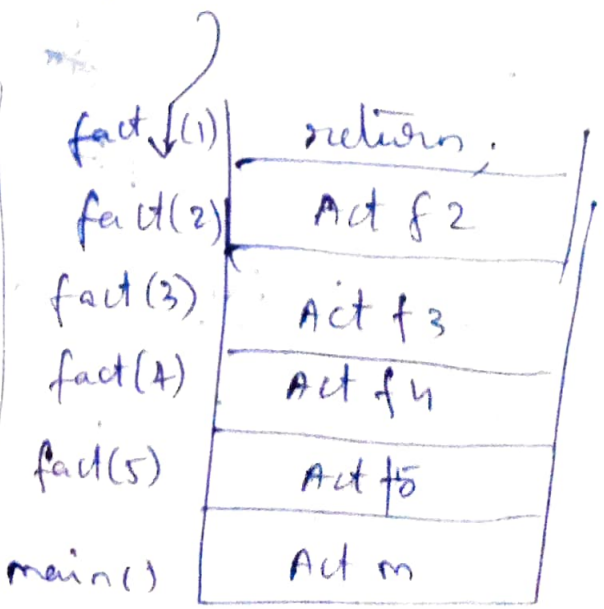
→ Recursive are more elegant and requires less lines of code

Recursive :

```
int fact (int n){
  if (n==1)
  return 1;
  else
  return n*fact(n-1);
}
int main(){
printf("%d", fact(5));
return 0;
}
```

Disadvantage : → Recursive require more stack space than iterative programs.





Iterative (or) Recursion ?

That depends.

→ If more elegant need, go for recursion.

→ If more efficient is needed, go for iteration.

80. Recursion (solved problem) :-

81.
82
83
84

Recursion

85 → Quiz

86:- Array:-

Before that we need to Know "Data Structure".

Data structure is a format for organizing and storing data.

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

For example: Array is data structure which you can visualize as follows.

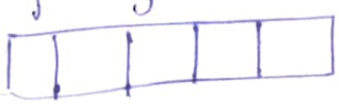
→ An array as a large chunk of memory divided into smaller block of memory and each block is capable of storing a data value of some type.

What is array? An array is a data structure containing a number of data values (all of which are of same type).

87. Declaration of array:

Syntax: data_type name of the array [no. of elements]

For example:- an array of integers can be declared as follows.

int arr[5]; arr 

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

→ Length of the array can be specified by any positive integer constant expression.

Best practice :-

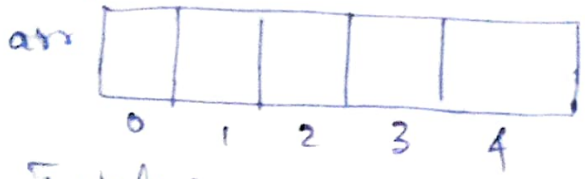
specifying the length of an array using Macro is considered to be an excellent practice.

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

88. Accessing array element:-

To access an array element, just write

array_name[index]



Accessing first element : arr[0]
Accessing second element: arr[1]
and so on...

89. Initializing an array:-

Method 1:-

```
arr[3]={8,9,10};
```

Method 2:-

```
arr[] = {8,9,10}
```

Method 3:-

```
int arr[5];
arr[0] = 8;
arr[1]=9;
arr[2]=10;
```

=> Not preferable

Method 4:-

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

-> If no. of elements are lesser than length of the array then the rest of the locations are automatically filled by value 0.

-> Easy way of Initialization of an array with all zeros is given by:

```
int arr[10]={0};
```

-> At the time of initialization, never leave those flower brackets {} empty and also never exceed the limit of no. of elements specified by the length of an array.

```
int arr[3]={};
int arr[3]={8,9,10};
```

} X illegal.

90. Designated Initialization of arrays:-

Sometimes we want something like this.

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

How great it would be if I don't have to explicitly write these zeros.

So,

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

↳ This way of Initialization is called designated Initialization.

→ And each no. in the square brackets is said to be a designator.

Advantages:-

① No need to bother about entries containing zeros.

int a[15] = {1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0}

→ int a[15] = {[0] = 1, [5] = 2};

② No need to bother about order at all.

int a[15] = {[0] = 1, [5] = 2}; } Both are same.

int a[15] = {[5] = 2, [0] = 1};

Be Careful!

If the length of an array is 'n', then each designator must be between 0 and n-1.

int a[5] = {[0] = 4, [4] = 78}; ✓

int a[5] = {[0] = 4, [5] = 78}; ✗

↓

index is wrong;
it should be n-1, 5-1 = 4.
(max)

What if I won't mention the length?

→ Designators could be any non-negative number.

→ Compiler will deduce the length of the array from the largest designator in the list.

```
int a[] = {[5] = 90, [20] = 4, [1] = 45, [49] = 78};
```

↓

Because of this ~~descriptor~~ designator maximum length of this array would be 50.

We can do mix also:

```
int a[] = {1, 7, 5, [5] = 90, 6, [8] = 4};
```

↓ equal as below

```
int a[] = {1, 7, 5, 0, 0, 90, 6, 0, 4};
```

But if there is a clash, then designated initializer will win.

```
int a[] = {1, 2, 3, [2] = 4, [6] = 45};
```

↓ equal to

```
int a[] = {1, 2, 4, 0, 0, 0, 45};
```

91. Array 8n

92. Array 9n.

93. Count array elements using sizeof() operator:-

Size of 1 array element × no. of elements = size of whole array

$$\text{no. of elements} = \frac{\text{size of whole array}}{\text{size of 1 array element}}$$

$$\text{length of an array} = \frac{\text{sizeof(name\_arr)}}{\text{sizeof(name\_of\_arr[0])}}$$

94. Multidimensional array :-

Multidimensional arrays can be defined as an array of arrays.

General form of declaring N- dimensional array is as follows:

data-type name-of-array [size1][size2]....[sizeN];

For example:

int a[3][4]; // TWO Dimensional array

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

Size Calculation:-

Size of multidimensional array can be calculated by multiplying the size of all dimensions.

For example:

Size of a[10][20] = 10×20 = 200
= 200×4 = 800 bytes

We can store upto 200 elements in array

Size of a[4][10][20] = 4×10×20 = 800
= 800×4 = 3200 bytes

We can store upto 800 elements in this array

95. Introduction to two dimensional arrays:-

Syntax: data-type name-of-array[x][y] where x and y are representing the size of the array.

Visualizing 2D array:-

int arr[4][5];

rows # Columns

Initialize 2D:-

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

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

Accessing 2D array elements:-

→ using row index and Column index

Printing 2D array elements:-

→ By using two nested loops.

96. Introduction to 3D array:-

Initialize 3D:-

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

Method 2: (Better method)

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

printing 3D array elements:-

→ By using three nested for loops.

97. Multidimensional arrays cont

98. Matrix multiplication.

99. Matrix multiplication.

100. Constant arrays in c:-

Either one dimensional (or) Multi-dimensional arrays can be made Constant by starting the declaration with keyword Const.

For example:- Const int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

a[1] = 45; ⇒ produces error.

Advantages:-

→ It assures us that program will not modify the array which may contain some valuable information.

→ It also helps the Compiler to catch errors by informing that there is no intention to modify this array.

(If anyone tries to change, Compiler threw error and doesn't allow other programmer to change it).

101: Variable length in arrays:-

Ask user to provide ~~read~~ length of an array. → scanf("%d", &n);
(ii) After getting that, declare array. int a[n];

Advantages:

→ At the time of execution, we can decide the length of the array.
→ No need to choose the fix length while writing the code.
→ Even arbitrary expressions are possible

Example: int a[3+i+5];
int a[k/7+2];
int a[i+5];

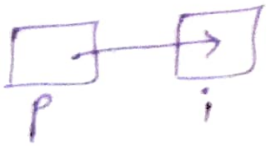
~~Disadv~~ points to be noted:

(1) variable length arrays cannot have static storage duration.
↳ means static but we are varying length

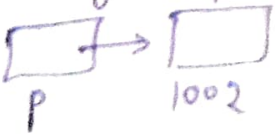
(2) Variable length array does not have the initializer.

102:- Introduction to pointer:-

→ pointer is a special variable that is capable of ~~storing~~ storing some address.



→ It points to a memory location where first byte is stored.



(103):- Declaring and initializing pointer variable:

General syntax for declaring pointer variables:

data-type * pointername

Here data-type refers to the type of the value that the pointer will point to.

For example:

```
int *ptr;   <-- points to integer value
char *ptr;  <-- points to character value
float *ptr; <-- points to float value
```

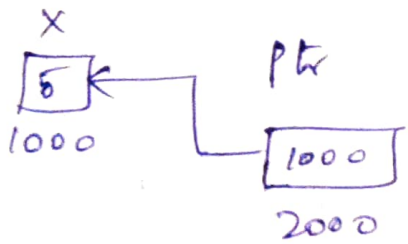
Need of ADDRESS of operator:-

→ Simply declaring pointer is not enough.

→ It is important to initialize pointer before use.

→ One way to initialize a pointer is to assign address of some variable.

```
int x=5;
int *ptr;
ptr=&x;
   ↳ address of operator
```




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

is equivalent to

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

(ii) Value of operator in pointer:-

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

```
int x=5;
int *ptr;
ptr=&x;
printf("%d", *ptr);
```

→ value of operator. It says go to the address of object and take what is stored in the object

→ We can also change the value of the object pointed by pointer.

Aword of Caution:-

① Never apply the indirection operator to the uninitialized pointer.

- for example.

```
int *ptr;
printf("%d", *ptr);
```

② Assigning value to an uninitialized pointer is dangerous.

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

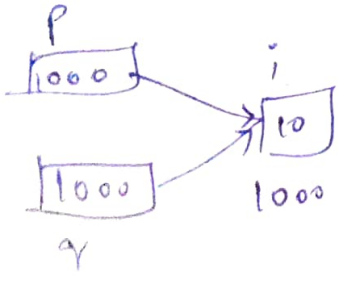
o/p:- Segmentation fault (SIGSEV)

↳ usually segmentation fault is caused by program trying to read (or) write an illegal memory location.

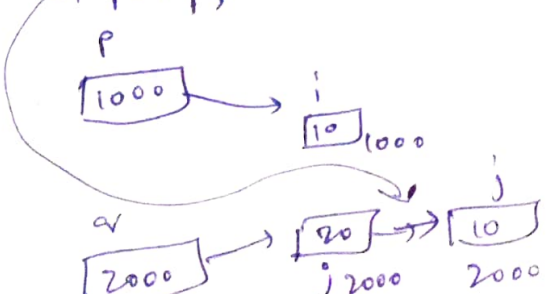
(05) pointer assignment :-

Note that q=p is not same as *q=*p;

```
int i=10;
int *p, *q;
p=&i;
q=p;
```




```
int i=10, j=20;
int *p, *q;
p=&i;
q=&j;
*q=*p;
```



106. pointer application (Finding the largest & smallest Elements) :-

107. Returning pointers :-

Find mid of the array :-

```
int main()
{
int a[] = {1,2,3,4,5};
int n = sizeof(a)/sizeof(a[0]);
int *mid = findmid(a,n);
printf("%d", *mid);
return 0;
}
```

```
int *findmid(int a[], int n)
{
  return &a[n/2];
}
```

o/p :- 3.

Word of Caution :-

Never ever try to return the address of an automatic variable

for example :-

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

Warning :- function returns address of local variable.

108. pointers (Important Questions) :-

Q1 :- Consider the following two statements.

```
int *p=&i;
    p=&i;
```

First statement is the declaration and second is simple assignment statement. why isn't in second statement, p is preceded by * symbol?

Solution:- Inc * symbol has different meanings depending on the context in which it's used.
At the time of declaration, * symbol is not acting as an indirection operator.
* symbol in the first statement tells the compiler that p is a pointer to an integer.
But if we write *p=&i then it is wrong, because here * symbol indicates the indirection operator and we cannot assign the address to some integer variable.
Therefore in the second statement, there is no need of * symbol infront of p. It simply means we are assigning the address to a pointer.

Qn2:- What is the o/p of following prgm.

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

o/p:-
Error: Assignment of read-only location *p

Qn-3:- How to print the address of a variable?
Solution:- use %p as a format specifier in printf function

```
int main(){
 Int i=10;
 int *p=&i;
 Printf("The address of variable i is %p",p);
 return 0;
}
```

o/p:- The address of variable i is 0x7ffd5b7a787c

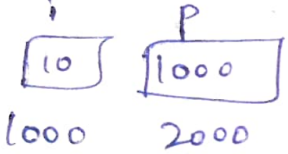
Qn4:- If i is a variable and p points to i, which of following expressions are aliases of i?

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

Solution:- option (a) and (e) are aliases of variable i

Example:-

```
int i=10;
int *p=&i;
```



a) *p = *(1000) = 10

b) *&p = *(&p) = *(2000) = 1000

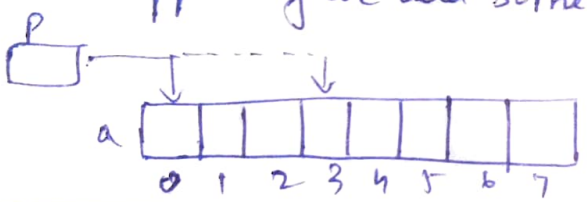
c) &p = 2000

d) *i = *(10) doesn't make sense

e) *&i = *(&i) = *(1000) = 10

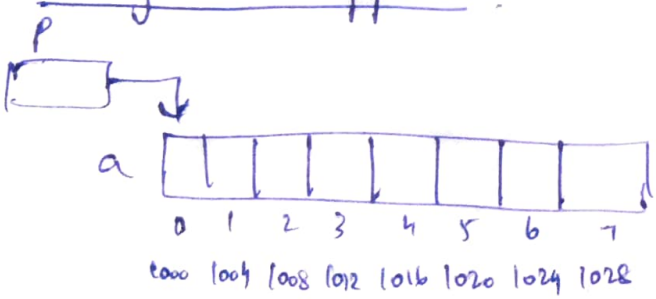
109. pointer Arithmetic: (Addition)

What happens if we add some integer to pointer?



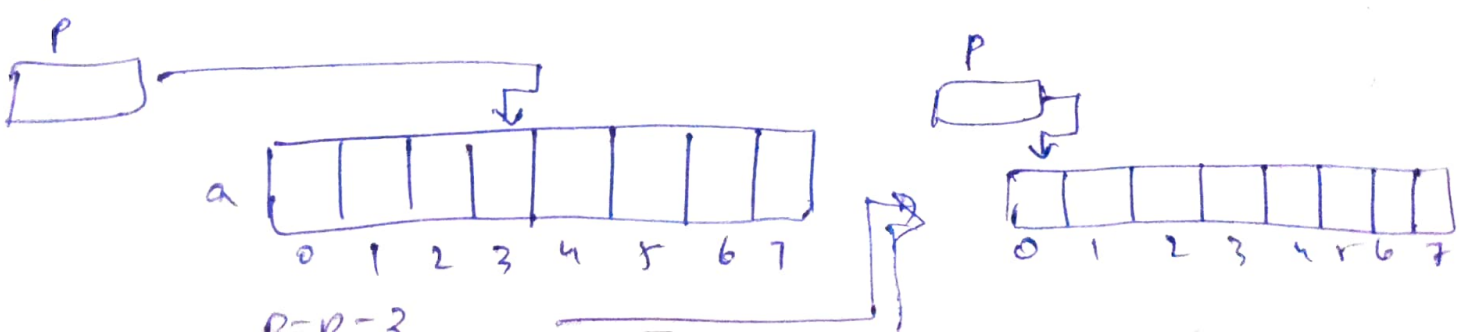
p=p+3 means moving the pointer 3 positions in forward direction.

Actually what happened?



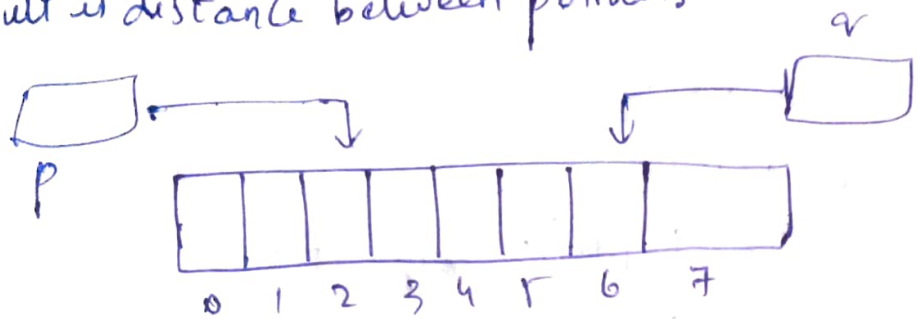
p=p+1 ≡ p=1000+1×4

110. pointer arithmetic subtraction:-



Subtracting one pointer from Another pointer

Result is distance between pointers



q-p=4

Eg: q=1028; p=1008 ⇒ q-p ⇒ 1028-1008 ⇒ 16

(ie) 16 bytes → but distance (index) = 16 / 4 (size of integer) = 4.

undefined behaviours :-

① 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));
return 0;
}
```

Different o/ps everytime.

② If two pointers are pointing to different arrays then performing subtraction between them leads to undefined behaviour.

```
int main()
{
int a[]={1,2,3,4};
int b[]={10,20,30,40};
int *p=&a[0];
int *q=&b[3];
printf("%d", q-p);
return 0;
}
```

undefined behaviour

Online:-
7

Offline:-
-1

111. pointer arithmetic (Increment & DeCrement):-

post Increment:-

```
int main() {
int a[] = {1,2,3,4};
int *p = &a[0];
printf("%d", *(p++));
printf("%d", *p);
return 0;
}
```

o/p: 1 2

Pre-Increment:-

```
int main() {
int a[] = {5,6,7,8,9,10};
int *p = &a[0];
printf("%d", *(++p));
return 0;
}
```

o/p: 6

pre and post decrement:-

```
int main() {
int a[] = {1,2,3,4,5};
int *p = &a[2];
printf("%d", *(--p));
printf("%d", *(p--));
return 0;
}
```

o/p: 2 2

112. pointer arithmetic - Comparing pointers:-

→ use relational operators (<, >, <=, >=) and equality operators (==, !=) to Compare pointers.

→ Only possible when both pointers point to same array.

→ Output depends upon the relative positions of both the pointers.

Example:-

```
int main(){
int a[] = {1,2,3,4,5,6};
int *p = &a[3];
int *q = &a[5];
printf("%d\n", p<=q);   ⟶ o/p: 1
printf("%d\n", p>=q);   ⟶ o/p: 0
q = &a[3];
printf("%d", p==q);     ⟶ o/p: 1
return 0;
}
```

113. pointers (pgm-1)

pgm - Calculate the sum of elements of array using pointers :-

```
int main(){
int a[] = {11,22,36,5,2};
int sum = 0, *p;
for(p=&a[0]; p<=&a[4]; p++)
  sum += *p;
printf("sum is %d", sum);
}
```

o/p:- sum is 76.

114. Using array name as a pointer:-

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

For example:-

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

o/p: 10.

old pgm on (113).

New program

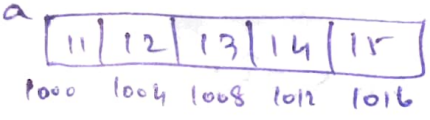
```
int main() {
int a[7] = {11,12,13,14,15};
int Sum=0,*p;
for(p=a; p<=a+4; p++)
  Sum+=*p;
printf("Sum is %d", Sum);
}
```

O/P:- Sum is 65.

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

For example:-

```
int main() {
int a[7] = {11,12,13,14,15};
printf("%d", a++);
return 0;
}
```



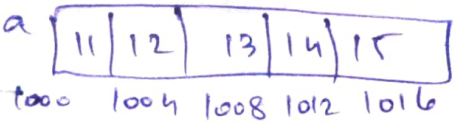
We are trying to assign address 1004 to array name 'a'.
Recall that name of array indicates the base address of array (i.e) 1000
We cannot change this.

There is no problem in the code below.

For example:-

```
int main(){
int a[7] = {11,12,13,14,15};
printf("%d", a+1);
return 0;
}
```

O/P: 12



Here we are not trying to assign some new address to 'a'.
We are simply accessing the address of the second element of array.

In more simpler,

```
int main(){
int a[] = {11,12,13,14,15};
int *p = a;
printf("%d", *(++p));
return 0;
}
```

o/p: 12

115. pointers (pgm 2):- Reverse a series of numbers using pointers.

```
#include <stdio.h>
#define N 5
int main(){
int a[N], *p;
printf("Enter the no of elements in array: ", N);
for (p=a; p<=a+N-1; p++)
  scanf("%d", p);
printf("Elements in reverse order: \n");
for( p=a+N-1; p>=a; p--)
  printf("%d", *p);
return 0;
}
```

116. passing Array name as an argument to a function:-

```
int add(int b[], int len)
{ int sum=0, i;
  for(i=0; i<len; i++)
    sum += b[i];
  return sum;
}
int main(){
int a[] = {1,2,3,4};
int len = sizeof(a)/sizeof(a[0]);
printf("%d", add(a, len));
return 0;
}
```

We can also write int *b;

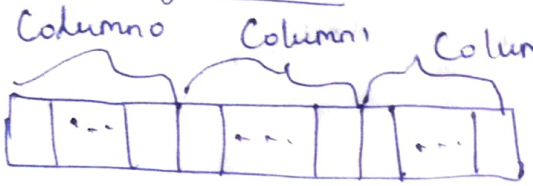
→ We are not passing the whole array. We are just passing the base address of the array.

117. Using pointers to print 2D arrays :-

Difference between Row major and Column major order

Row major order :- Elements are stored row by row

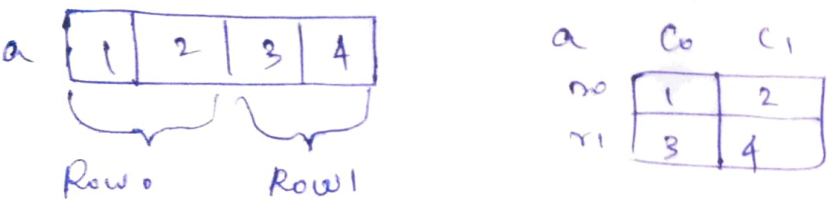
Column major order :- Elements are stored Column by Column.



Note :- C stores multidimensional arrays in row major order.

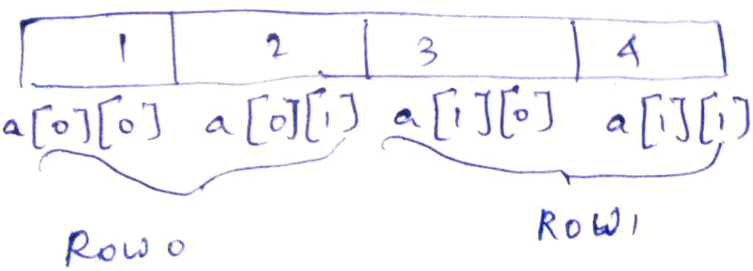
Traditionally,

```
for (i=0; i<row; i++)
for (j=0; j<column; j++)
  printf("%d", a[i][j]);
```



Now Same above is using pointers :- Advantage :- [single line].

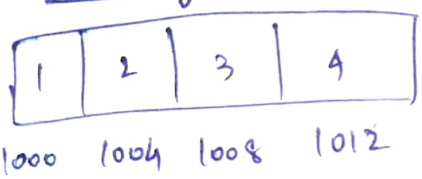
```
for (p=&a[0][0]; p<&a[row-1][column-1]; p++)
printf("%d", *p);
```



row = 2, column = 2

118:- processing Multidimensional array Elements (or) Address Arithmetic of Multidimensional Arrays :-

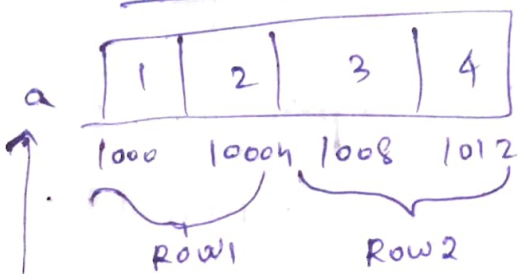
1D - Array:



int a[4];
means array of 4 integers
pointer to first element of the array

a => 1000
*a = 1

2D - Array:



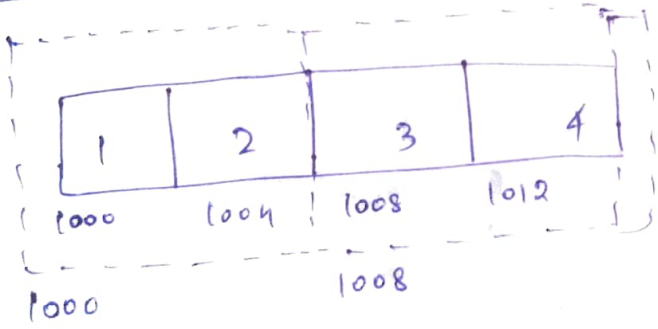
int a[2][2];

Two 1D arrays

Each of which Contains 2 elements

pointer to first 1D array

2D-array:



(1) a => 1000 (pointer to 1st 1D array)

a+1 => 1008 (pointer to 2nd 1D array)

*a => * (pointer to 1st 1D array)

(or)

*a => pointer to 1st element of 1st 1D array.

(i.e) *(a+0) => a[0] => &a[0][0] } Same

So **a => 1 [value of 1st element of 1st 1D array]

② *(a+1) ⇒ pointer to first element of 2nd 1D array.

(or)

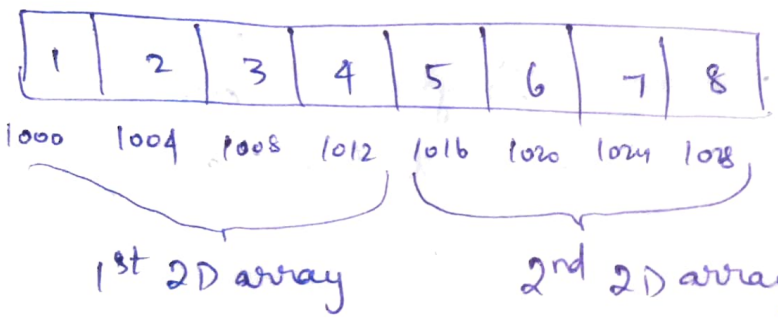
*(a+1) ⇒ *(pointer to 2nd 1D array)

**(a+1) ⇒ value of first element of 2nd 1D-array ⇒ 3.

③ *(a+1)+1 ⇒ pointer to second element of 2nd 1D array.

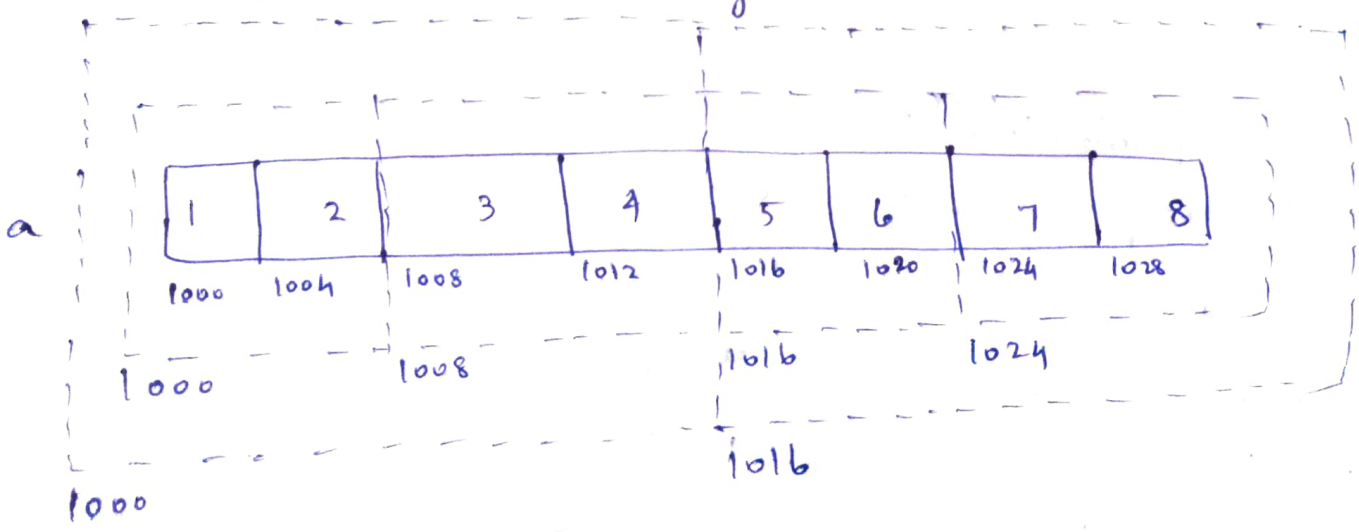
((a+1)+1) ⇒ value of second element of 2nd 1D array ⇒ 4

3D array:-



int a[2][2][2];

- Pointer to first 2D array
- Two 2D-arrays
- Each of which Contains two 1D arrays
- Each of which Contains two elements



a = pointer to 1st 2D array = 1000

a+1 = pointer to 2nd 2D array = 1016

*(a+1) = pointer to 1st 1D array of 2nd 2D array

((a+1)) = pointer to 1st element of 1st 1D array of 2nd 2D array

**(*(a+1)) = value of 1st element of 1st 1D array of 2nd 2D array

= 5 = a[1][0][0]

① Now want to access 2nd element of the array (i.e) value '2'.

a = pointer to 1st 2D-array

*a = pointer to 1st 1D-array of 1st 2D-array

**a = pointer to 1st element of 1st 1D-array of 1st 2D-array.

(**a+1) = pointer to 2nd element of 1st 1D-array of 1st 2D-array.

*(**a+1) → value of "

= 2

② Now want to second last element of array using pointer arithmetic (i.e) '7'.

a = pointer to 1st 2D-array

a+1 = pointer to 2nd 2D-array

*(a+1) = pointer to 1st 1D-array of 2nd 2D-array

(*(a+1)+1) = pointer to 2nd 1-D array of 2nd 2D-array.

((a+1)+1) = pointer to 1st element of 2nd 1D-array of 2nd 2D-array.

**(*(a+1)+1) = value of "

= 7

119. pointers (program3) :-

Consider the following declaration of two dimensional array in c

Char a[100][100]

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

a) 4040 b) 4050 c) 5040 d) 5050

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

Where BA = Base address of whole 2D array

NC = Number of Columns

c = Size of data type of elements stored in array (in bytes)

$a[lb_1 \cdots ub_1][lb_2 \cdots ub_2]$

Now:- $\&a[40][50]$

Formula:- BA=0, NC=100, c=1byte, $a[0 \cdots 99][0 \cdots 99]$

$\&a[40][50] = BA + [(i - lb_1) * NC + (j - lb_2)] \times c$

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

$= 0 + [40 * 100 + 50] \times 1$

$= 0 + [4000 + 50]$

$= 4050$

120. pointer (program 4):-

What is the o/p of following C Code? Assume that address of x is 2000 (in decimal) and an integer requires four bytes of memory.

```
#include <stdio.h>
int main()
{
unsigned int x[4][3] = {{1,2,3},{4,5,6},
                        {7,8,9},{10,11,12}};
printf("%u, %u, %u", x+3, *(x+3), *(x+2)+3);
}
```

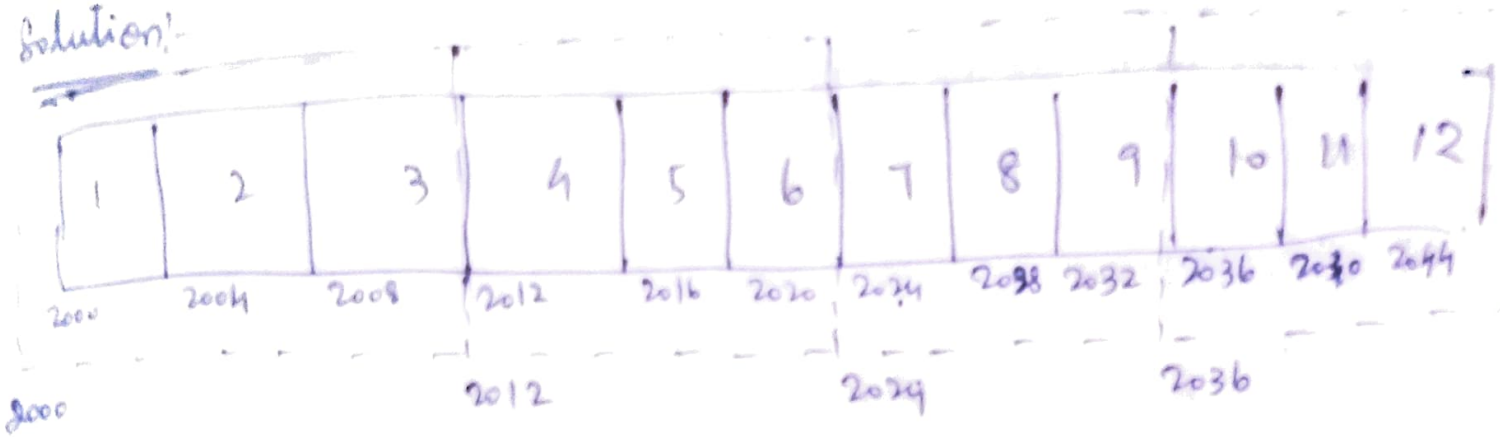
a) 2036, 2036, 2036

b) 2012, 4, 2204

c) 2036, 10, 10

d) 2012, 4, 6

Solution:-



(i) $x+3 \Rightarrow$ ~~$2000+(3\times4)$~~ $\Rightarrow (2000+3)*4 \Rightarrow$ ~~$2002*4$~~

$\Rightarrow 2000+(9\times4) \Rightarrow 2036$

(ii) $*(x+3) \Rightarrow 2036$

(iii) $*(x+2)+3$:-

$\Rightarrow x+2 \Rightarrow 2024$

$\Rightarrow *(x+2) \Rightarrow 2024$

$\Rightarrow *(x+2)+3 \Rightarrow 2024+(3\times4) \Rightarrow 2036$

So solution is a) 2036, 2036, 2036

121. pointer pointing to an Entire array:-

```
#include "stdio.h"
int main(){
int a[5]={1,2,3,4,5};
int *p=a;
printf("%d",*p);
return 0;
}
```

pointer to first element of array

Now:-

```
int main(){
int a[5]={1,2,3,4,5};
int (*p)[5]=&a;
printf("%p",p);
return 0;
}
```

→ what is this?

int (*p)[5];

↳ pointer to whole array of 5 elements

How to read?

int (*p)[5];

↑

A pointer

int (*p)[5];

↓

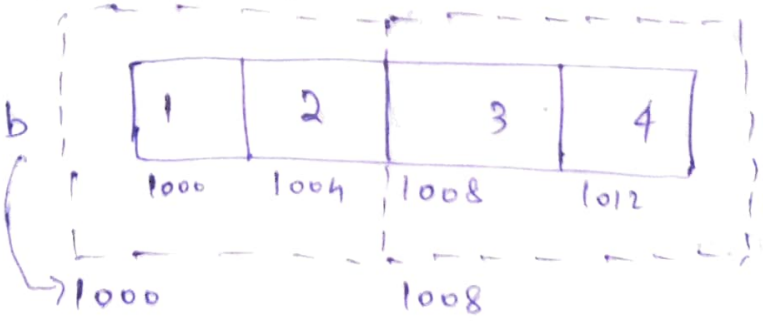
to five integer elements.

Passing address:- int (*p)[5] = &a;

Here we are passing &a instead of a why?

To understand this, Consider

Ex:- 2D array



*b = pointer to 1st element of 1st 1D array

&*b = b = pointer to 1st 1D array.

* → goes inside of imaginary box

& → goes outside of imaginary box

Similarly in this case we can see 'a' is name of whole one dimensional array but it points to 1st element of 1-D array.

Now we want to go outside that means we want address of whole 1-D array. 'a', Right now this points to first element of 1D-array, we want address of whole 1D-array, so we put '&' infront of a (i.e) &a

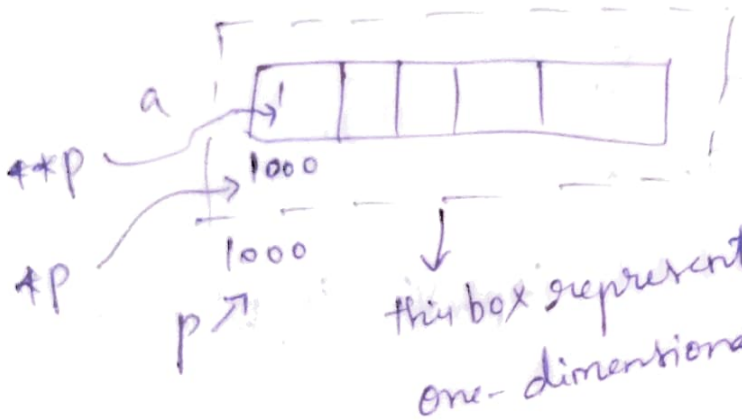
-> we want to go outside, actually not inside

-> If we put &a, we want address of whole dimensional array of all 5 elements

printing the value:-

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

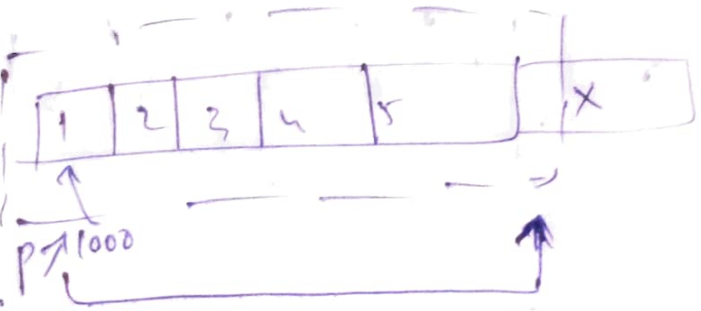
o/p: 1



So the Concept is pointer pointing to whole dimensional array instead of pointing single element.

In the above scenario, don't do p+1;-

```
int (*p)[5]=&a;
printf("%d", **p);
printf("address of p+1: %u\n", p+1);
```



↳ Now shifting to whole 1-D after 5 elements (ie) shifting.

Because store address of whole dimensional array.

note:- If we do `int (*p)[5] = a` [(ie) trying to assign first element address of an array only]

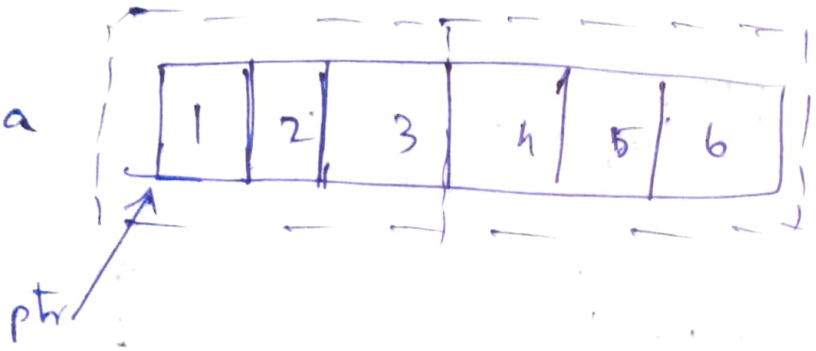
It will throw error "Cannot convert int * to int(*)[5] in initialization."

122. pointer pointing to entire array (solved problem).

o/p pgm:-

```
int main()
{
 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]);
return 0;
}
```

o/p:- 2 3 4 5



Here we do (*ptr)[3]=a not (*ptr)[][3]=&a.

⇓ assign first elem

points to address of 1D array of 3 elements only

⇓ points to whole dimension.

If we do (*ptr)[3]=&a,

it will throw error, "Cannot convert int(*)[2][3] to int(*)[3] in initialization".

because trying to assign whole dimensional array's address (having 3 elements in 2D array) to pointer which can hold only 3 elements.

123. pointers (program 5):-

what is the o/p of following c program:

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

a) 2 2
b) 2 1
c) 0 1
d) 0 2

124. pointers (progm 6):-

```
#include<stdio.h>
int f(int *a, int n)
{
if(n<=0) return 0;
else if(*a%2==0) return *a+f(a+1, n-1);
else return *a-f(a+1, n-1);
}
int main()
{
int a[]={12, 7, 13, 4, 11, 6};
printf("%d", f(a, 6));
getchar();
return 0;
}
```

a) -9
b) 5
c) 15
d) 19

125. pointer (prgm 7):-

o/p of c pgm:-

```
int f(int x, int *py, int **ppz)
{
  int y, z;
  **ppz += 1;
    z = **ppz;
    *py += 2;
     y = *py;
     x += 3;
 return x+y+z;
}
void main()
{
 int c, *b, **a;
 c=4, b=&c, a=&b;
printf("%d", f(c,b,a));
}
```

options:-

a) 18

b) 19

c) 21

d) 22

Assume:-

4	100	200
100	200	300
c	b	a

→ f(4, 100, 200)

x=>4, py=100, ppz=200;

ppz = (ppz) + 1;

**ppz = 5

z = 5

*py = (*py) + 2

*py = 5+2

*py = 7

y = 7

x = 4 + 3 = 7

return 7 + 7 + 5 = 19

126. pointer (prgm 8):-

```
void swap (int **x, int *y)
{
 static int *temp;
  temp=x;
  x=y;
  y=temp;
}

main()
{
  printtab();
  printtab();
}

void printtab()
{
 static int i, a=-3, b=-b;
  i=0;
  while (i<=4)
  {
    if((i++)%2==1) continue;
    a=a+i;
    b=b+i;
  }
  swap(&a,&b);
  printf("a=%d, b=%d\n", a, b);
}
```

a) a=0, b=3
   a=0, b=3

b) a=3, b=0
   a=12, b=9

c) a=3, b=6
   a=3, b=6

d) a=6, b=3
   a=15, b=12

127. pointer (program 9):-

What should be contents of the array 'b' at end of the program?

```
#include "stdio.h"
int main()
{
 int i,j;
 char a[2][3]={ {'a','b','c'},{'d','e','f'} };
 char b[3][2];
 char *p=*b;
 for(i=0; i<2; i++) {
   for (j=0; j<3; j++) {
     *(p+ 2*j +i) = a[i][j];
   }
 }
}
```

options:-

a) a b
   c d
   e f

b) a d
   b e
   c f

c) a c
   e b
   d f

d) a e
   d c
   b f

128. String literals:-

Definition:- String Literal (or string Constant) is sequence of characters enclosed within double Quotes.

For example:- "Hello Everyone", "Hi"

%s is a placeholder :-

```
int main()
{
printf("%s","Hello Everyone");
return 0;
}
```

Note :- Double Quotes are important.

Continuing string Literals:-

```
printf("%s","you have to dream before your dreams become true.
--A.P.J.Abdul Kalam");
```

Here we want --A.P.J.Adbul Kalam in single line in o/p, and also I am not willing to give this in a single line (i.e) ~~I expect the same above in o/p~~ But 'C' does not support this in coding.

It gives Error.

To avoid this error :-

```
printf("%s","you have to dream before your dreams become true.\
--A.P.J.Abdul Kalam");
```

Because of this, error is not generated.

This method is called "splicing".

It also having disadvantage. Let see below example.

```
int main()
{
    printf("%s\n","you have to dream before your dreams become true.\
--A.P.J.Abdul Kalam");
    printf("%s","you have to dream before your dreams becomes true.
"--A.P.J.Abdul Kalam");
}
```

This is due to '/' introduced. It focuses indention for this.

O/P:- Here Indentation is '1' tab (ie) 4 spaces.

you have to dream before your dreams become true. [ ] -- A.P.J. Abdul Kalam

you have to dream before your dreams become true. -- A.P.J. Abdul Kalam,

12g. storing of string literal:-

printf ("Earth");

"Earth" is a string literal.

First argument to printf and scanf function is always a string literal.

But ~~why~~ what we are actually passing to printf/scanf!

Let's understand how string literals are stored?

String literals are stored as an array of characters.

E	a	r	t	h	\0

← Indicates the end of the string.

↑ null character.

Total 6 bytes of read only memory is allocated to above string literal.

'\0' character must not be confused with '0' character. Both have different ASCII codes. '\0' has the code 0 while '0' has the code 48

OK. Fine. But what we are actually passing to printf/scanf?

In C, Compiler treats a string literal as pointer to the first character.

E	a	r	t	h	\0

1000

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

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

printf ("Earth")

Passing "Earth" is Equivalent to passing the pointer to letter 'E'.

130. Assigning string literal to a pointer:-

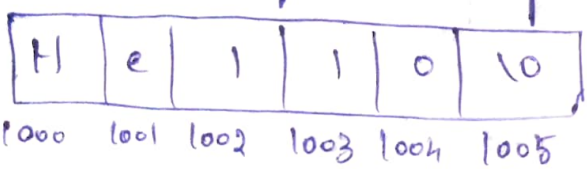
Char *ptr = "Helloworld!".

Recall:- Writing string literal is Equivalent to writing pointer to the first character of the string literal.

Char *ptr = "Hello";

As writing "Hello" is Equivalent to writing pointer to the first character, Therefore, we Can subscript it to get some character of string literal.

"Hello"[i] is Equivalent to pointer to 'H'[i]



pointer to 'H'[i] = *(pointer to 'H' + i)

pointer to 'H'[i] = *(1000 + i) = *(1000 i) = 'e'

Similarly

"Hello"[0] => 'H'

"Hello"[1] => 'e'

"Hello"[2] => 'l'

"Hello"[3] = 'l'

"Hello"[4] = '0'

Points to be Noted:-

String literal Cannot be modified. It Causes undefined behaviour.

Char *ptr = "Hello";

*ptr = 'M'; ⟶ This modification is not allowed.

String literals are also known as String Constants.

They have been allocated to read only memory. So we Cannot alter them.

But character pointer itself has been allocated read-write memory. So the same pointer can point to some other string literal.

(iv) char *ptr = "Hello";

*ptr = 'H' → this is not allowed.

but ptr = "Hi"; → This is allowed.

printf("%s", ptr); o/p: Hi

Note:- printf("%s", *ptr); → should not come → No o/p.

131. String Literal Vs. Character Constant:-

String literal and character constant are not same.

"H" ≠ 'H'

"H": Always represented by a pointer which points to character 'H'.

'H': Represented by an integer (ASCII Code: 72)

printf("\n"); ≠ printf('\n');

✓ ✗ (error)

printf expects a pointer to character not the integer.

That's why we give " " → string literal.

132. Declaring a String Variable:

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

For example:- char s[6];

Note:- Always make the array one character longer than string.

If length of string is 5 characters long then don't forget to make extra room for NULL character.

Failing to do the same may cause unpredictable results when prog is executed as some c libraries assume the strings are Null terminat

Initializing a string variable:-

Example:- char s[6] = "Hello";

Although it seems like "Hello" in the above example is string literal but it is not.

When a string is assigned to character array, then this character array is treated like other types of array. We can modify its characters.

char s[6] = "Hello";
char s[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; } They both are equal

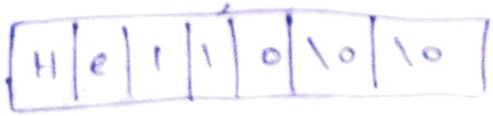
Recall:- We cannot modify a string literal

char *ptr = "Hello"; } error
*ptr = 'M';

But we can modify a char array
char s[6] = "Hello"; } No problem
s[0] = 'M';

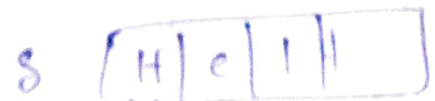
Short length initializer:-

char s[7] = "Hello";



Long length initializer:-

char s[4] = "Hello";



rest of the part is truncated.

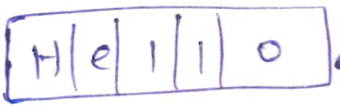
```
printf("%s", s);
```

Warning :- Initializer - string for array of chars is too long.

o/p :- Hell? → o/p is undefined behaviour.

Equal length initializer :-

```
char s[5] = "Hello";
```



No \0 at the end.

Eg :-

```
int main(){
char s[5] = "Hello";
Char t[5];
int i;
for(i=0; s[i] != '\0'; i++)
  t[i] = s[i];
printf("%s", t);
return 0;
}
```

o/p :- [undefined behaviour]
Hello[?] Hello[?]

To avoid :-

```
int main(){
char s[6] = "Hello";
char t[6];
int i;
for(i=0; s[i] != '\0'; i++)
   t[i] = s[i];
t[i] = "\0";
printf("%s", t);
return 0;
}
```

o/p :-
Hello

133. Writing strings using printf and puts function:-

```
Char *ptr = "Hello world";
printf ("%s", ptr);
```
o/p:- Hello World

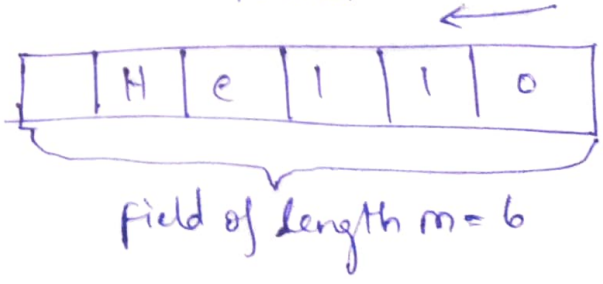
① "%.ns" is used to print just a part of the string where 'n' is the number of characters to be displayed on the screen.

```
Char *ptr = "Hello World";
printf ("%.5s", ptr);
```
o/p:- Hello

② "%.m.ns" is used to print just a part of the string where 'n' is the number of characters to be displayed and 'm' denotes the size of the field within which string will be displayed.

Q:
```
Char *ptr = "Hello world!";
printf ("%.5s", ptr);
printf ("%.6.5s", ptr);
```
o/p:- Hello
Hello

Note that characters are filled from last when field is used.



③ puts() function is a function declared in <stdio.h> library and used to write strings to the output screen.

Also puts() function automatically writes a newline character after writing the string to the o/p screen.

```
char *s = "Hello";
puts(s);
puts(s);
```
o/p:-
Hello
Hello

134. Reading strings using Sclanf and gets():-

Using Sclanf, we can read a string into string variable (character array)

```
char a[10];
printf("Enter the string:\n");
sclanf("%s",a);
printf("%s",a);
```

Like any array ~~name~~ name, a is treated as a pointer to first element of the array. Therefore there is no need to put &.

I/P: you are most welcome

O/P:- you ← Why only "you" is stored?

Ans: Sclanf() doesn't store white space characters in the string variable. It only reads characters other than white spaces and store them in the specified character array until it encounters a white-space character.

I/P: you are most welcome.

↑ when white-space is seen by Sclanf, it stops and hence only 'you' is stored in specified character array.

In order to read an entire line of input, gets() function can be used.

```
char a[10];
printf("Enter the string:\n");
gets(a);
printf("%s",a);
```

I/P:- you are most welcome.

your program may crash!! → Due to array size just 10. But we do overflow.

- Both gets and Sclanf() functions have no way to detect when the character array is full.
- Both of them never checks the maximum limit of input characters. Hence they may cause undefined behaviour and

probably lead to buffer overflow error which eventually caus the program to crash.

Although Scanf() has way to set limit for number of characters to be stored in character array.

By using %ns, where n indicates the number of characters allowed to store in the character array.

```
Char a[10];
Printf("Enter the string: \n");
Scanf("%.9s", a);        → Receives 9 character and '\0' is for null.
Printf("%s", a);
```

I/p:- youaremostwelcome

o/p:- youaremos.

- But unfortunately, gets() is still unsafe.
- It will try to write the characters beyond the memory allocated to the character array which is unsafe because it will simply overwrite the memory beyond memory allocated to character array.
- Hence it is advisable to not use the gets() function.

135. Designing the i/p function using getchar():-

As Scanf and gets function are risky to use. Hence it is advisable to use own i/p function.

We want our i/p function to have following functionalities.

1. It must continue to read the string even after seeing white space characters.
2. It must stop reading the string after seeing newline character.
3. It must discard extra characters.
4. And it must return no. of characters it stores in the character array.

```
#include <stdio.h>
int input(char str[], int n)
{
int ch, i=0;
while((ch=getchar())!='\0')
 if(i<n)
  str[i++]=ch;
 str[i]='\0';
 return i;
}
int main(){
char str[100];
int n=input(str, 5);
printf("%d %s", n, str);
return 0;
}
```

→ getchar() function is used to read one character at time from the user input.

It returns an integer Equivalent to ASCII Code of an integer

'\0' ASCII is 10.

	H	e	l	l	o	'\0'
str	72	101	108	108	111	10

I/P: Hello, How are you?

O/P: 5 Hello

136. 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 the screen.

Note:-

Always remember that character is internally represented in binary form only. It doesn't make any difference if you write int ch instead of char ch.

Eg:

```
#include <stdio.h>
int main()
{
  int ch;
  for(ch='A'; ch<='z'; ch++)
   putchar(ch);
  return 0;
}
```

137. String (problem-1):-

Identify which of the following Calls don't work properly and give the reason for the same.

a) printf("%c", '\n'); → This works.

b) printf("%c", "\n"); → This doesn't work. Because passed argument is string literal.

c) putchar('\n'); → This works.

d) putchar("\n"); → This doesn't work. putchar() expects ~~string~~ li characters/Integer as an argument not a string literal.

e) ~~put cho~~ puts('\n'); → This doesn't work. puts() expects string literal.

f) puts("\n") → This works.

g) printf("%s", '\n'); → This will not work.

h) printf("%s", "\n"); → This will work.

138. Introduction to cstring library:-

There are some operations which we can perform on strings.

For example: Copy strings, Concatenate strings, select substrings and soon.

<string.h> library Contains all the required functions for performing string operations.

So we just have to include this header file "#include <string.h>" in our program and we are good to go.

strcpy (string Copy) function:-

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

↑ It isn't modified. That's why it is Constant.

strcpy is used to copy a string pointed by source (including NULL) to destination (character array)

Eg:-

```
#include <stdio.h>
#include <string.h>
int main(){
char str1[]="Hello";
char str2[10];
printf("%s\n", strcpy(str2,str1));
printf("%s", str2);
return 0;
}
```

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

O/P: Hello
Hello

strcpy (string copy) function:-

We can also chain together a series of strcpy calls.

```
#include <stdio.h>
#include <string.h>
int main(){
Char str1[10]="Hello";
Char str2[10];
Char str3[10];
Strcpy(str3, strcpy(str2,str1));
printf("%s %s", str2, str3);
return 0;
}
```

O/P:-

Hello Hello

In Call to strcpy(str1, str2), there is no way 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 the character array str1 then it will be an undefined behaviour.

To avoid this, we can call strncpy function.

```
strncpy(destination, source, sizeof(destination));
```

Eg:-

```
#include <stdio.h>
#include <string.h>
int main() {
Char stra[6]= "Hello";
Char strb[4];
strncpy(strb, stra, sizeof(strb));
printf("%s", strb);
return 0;
}
```

str1 → strb, str2 → stra

o/p:
Hell

→ If we do
strcpy(strb, stra);
Cause undefined behaviour.

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

Eg:

```
int main()
  char stra[6]="Hello";
  char str2[6];
  strncpy(strb, stra, sizeof(strb));
  strb[sizeof(strb)-1]='\0';
  printf("%s", str2);
  }
```

o/p:
Hello

139. String length function:-

The strlen (string length) function:

prototype:- size_t strlen (const char* str);

↑ size_t is an unsigned integer type atleast 16 bit.

→ strlen() function is used to determine the length of the given string.

→ To this function, we should pass the pointer to the first character of string whose length we want to determine.

→ And as result, it returns length of the string.

Note: It doesn't count Null character ('\0')

Eg:
```
#include <stdio.h>
#include <string.h>
int main(){
char *str = "HelloWorld";
printf("%d", strlen(str));
return 0;
}
```
o/p: 11

(or)
```
char str[] = "HelloWorld";
```
o/p: 11

(or)
```
char str[100] = "Hell...";
```
o/p: 11

Because it calculates the length of the string and not the length of the array.

140. 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.

It returns the pointer to resulting string str1.

Eg:

```
int main()
{
char str1[100], str2[100];
strcpy(str1, "Welcome to");
strcpy(str2, "our Academy");
strcat(str1, str2);
printf("%s", str1);
return 0;
}
```

o/p:- Welcome to our Academy

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

Eg: char str1[15], str2[100];
in the above example.

undefined behaviour
Because you are trying to add more characters than the memory allocated.

strncat function:-

→ It is the safer version of strcat.

→ It appends the limited number of characters specified by the third argument passed to it.

Note: strncat automatically adds NULL character at the end of the resultant string.

Eg:-

```
int main(){
char str1[5], str2[100];
strcpy(str1, "He");
strcpy(str2, "llo!!!");
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
printf("%s", str1);
return 0;
}
```

sizeof(str1) - strlen(str1) - 1
= 5 - 2 - 1

sizeof(str1) → Size of the array
strlen(str1) → Size of the string
1 → Creating room for Null character.

14) STRCMP (String Comparison) function:

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

→ Compares two strings s1 and s2

→ Returns value

Less than 0, if s1 < s2

Greater than 0, if s1 > s2

Equal to 0, if s1 == s2

Notice Board:

Attention!!!

In ASCII character set

* All upper case letters are less than all lower case letters (upper case letters have ASCII codes between 65 and 90 and lower case letters have ASCII codes between 97 and 122)
* Digits are less than letters (0-9 digits have ASCII codes between 48 and 57)
* Spaces are less than all printing characters (space character has the value 32 in ASCII set)

Strcmp considers s1 < s2 if either one of the following conditions is satisfied:

→ when the first i characters in s1 and s2 are same and (i+1)st character of s1 is less than that of s2.

Eg:1:

```
int main(){
char * s1 = "abcd";
char * s2 = "abce";
if(strcmp(s1,s2) <0)
printf("s1 is less than s2");
else
printf("s1 is greater than (or) equal to s2");
return 0;
}
```

o/p:- s1 is less than s2.

Example 2:- In example 1,

Char * S1 = "abce";
Char * S2 = "bbce";

o/p:- S1 is less than S2.

Example 3:- In example 1,

Char * S1 = "bace".
Char * S2 = "abce";

o/p:- S1 is greater than or equal to S2

Example 4:- In example 1,

Char * S1 = "abcd";
Char * S2 = "abcd";

o/p:- S1 is greater than or equal to S2

And also one more Condition:-

StrCmp Conviders S1<S2 if either one of the following Conditions is Satisfied:

→ All the characters of S1 match S2, but S1 is shorter than S2.

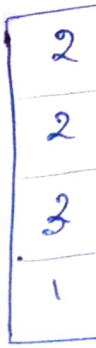
```
int main() {
Char *S1 = "abc";
Char * S2 = "abcd";
if(STRcmp(S1,S2) < 0)
printf("S1 is less than S2");
else
printf("S1 is greater than or equal to S2");
return 0;
}
```

o/p: S1 is less than S2

142. Array of Strings:-

Char fruits[3][12] = {"2 Oranges", "2 Apples", "3 Bananas", "1 pineapple"};

→ predetermined size (This is disadvantage)



Ther

Char

fru

0
1
2
3

143.

Conv

Char

Char

int

int i;

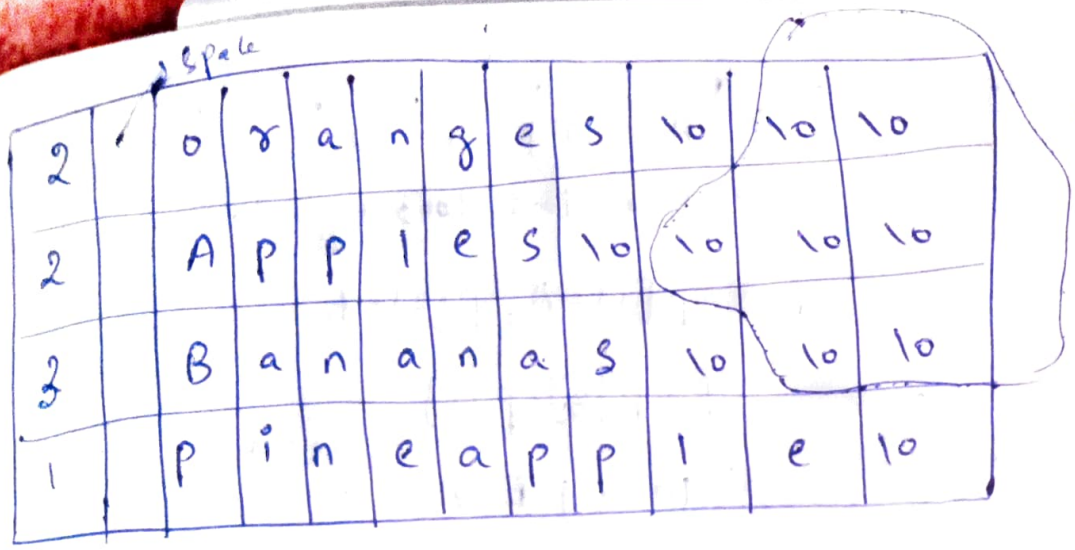
for(i

P[i

printf

→ print

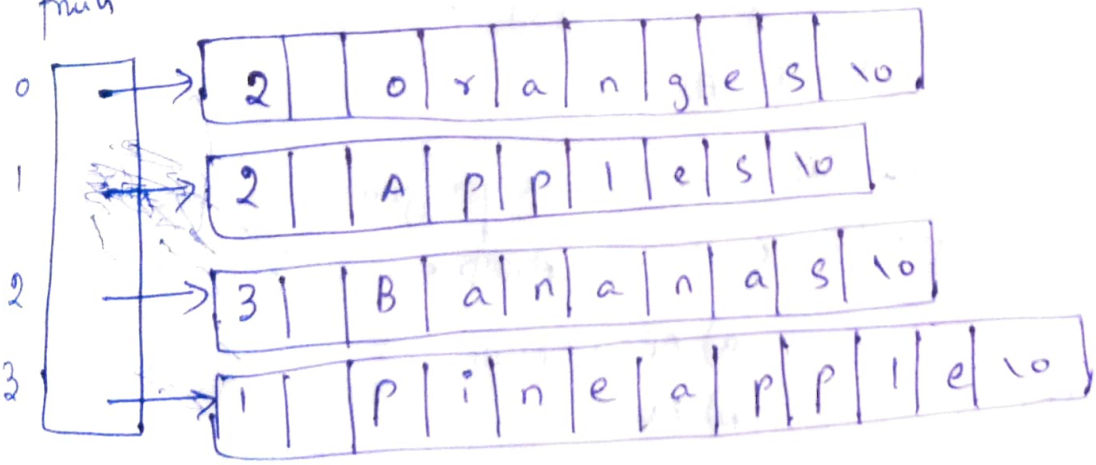
anyth



→ A Lot of space is simply wasted

There is an alternative → Array of pointers.

```
char *fruits[] = {"2 Oranges", "2 Apples", "3 Bananas", "1 pineapple"};
```



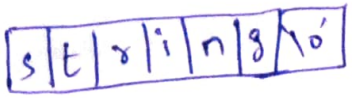
143. String (solved problem 2):-

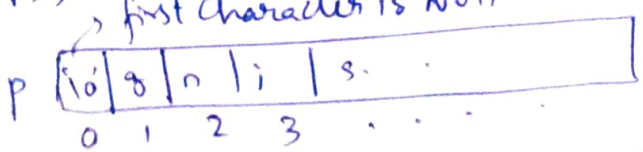
Consider the following c program segment

```
char p[20];
char *s = "string";
int length = strlen(s);
int i;
for(i=0; i<length; i++)
  p[i] = s[length-i];
printf("%s", p);
```

o/p:-

a) gnirts
b) string
c) gnirt
d) No output is printed





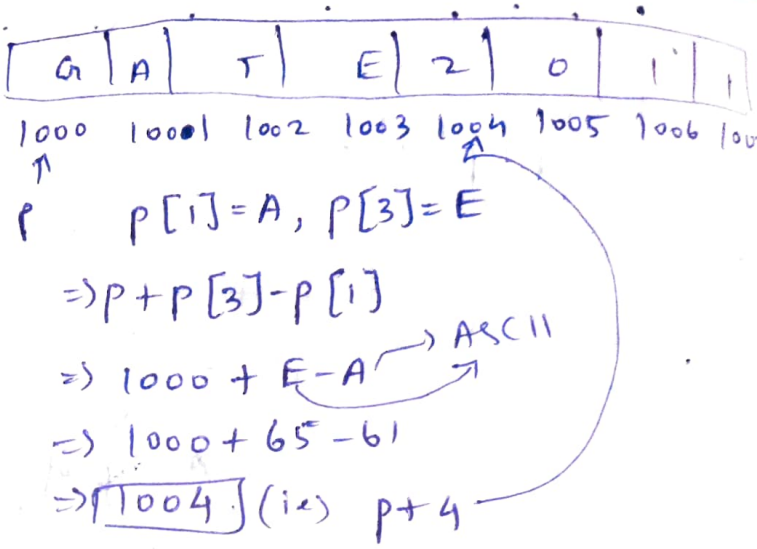
→ printf function will print everything before NULL character and will not see anything after null character. Therefore nothing will be printed on the screen.

144. Strings (solved problem 3):-

what does the o/p?

```
Char c[] = "GATE2011";
Char *p = c;
printf("%s", p + p[3] - p[1]);
```

a) GATE2011
b) E2011
c) 2011
d) 011



What happens if I Don't know ASCII Codes of all Characters? Just assume A=1, then E will be 5.
So E-A => 4.

145. (printf):- Consider the following function written in c programming language. The output of the below function on input "ABCD EFGH" is

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

a) ABCD EFGH
b) ABCD
c) HGFE DCBA
d) DCBA

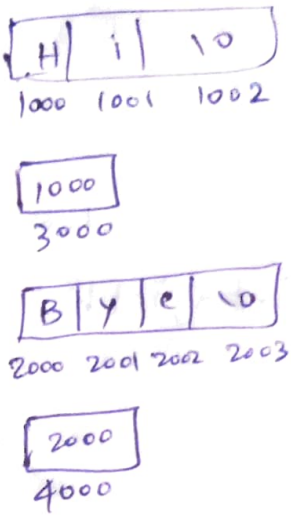
If *a && *a != ' ' means if a character pointed by pointer 'a' is neither a NULL character and nor a blank character then Continue else stop.

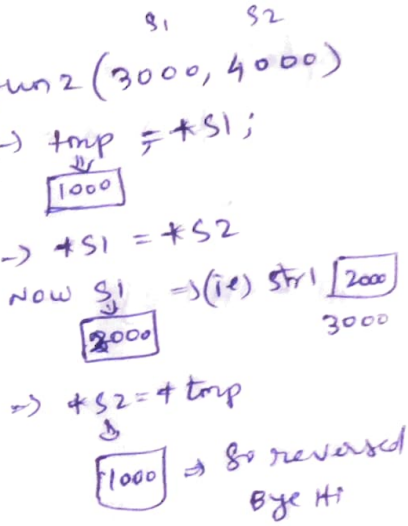
146. Strings (solved problem 5):-

```
void fun1(char *s1, char *s2)
{
  char *tmp;
  tmp = s1;
  s1 = s2;
  s2 = tmp;
}
```

```
void fun2(char **S1, char **S2)
{
char *tmp;
tmp = *S1;
*S1 = *S2;
*S2 = tmp;
}
int main(){
char *Str1 = "Hi", *Str2 = "Bye";
fun1(Str1, Str2); printf("%s %s\n", Str1, Str2);
fun2(&Str1, &Str2); printf("%s %s", Str1, Str2);
return 0;
}
```

o/p: Hi Bye
Bye Hi

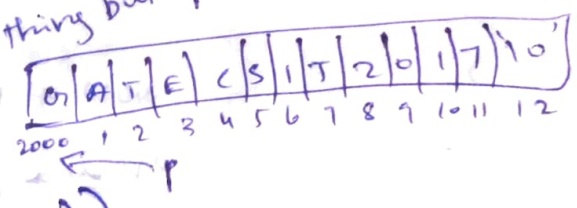




147. Strings (solved problem 6):

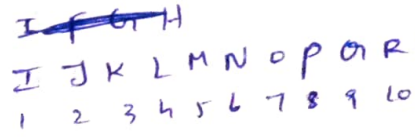
```
#include <string.h>
#include <stdio.h>
int main()
{
char *c = "GATEGSIT2017";
char *p = c;
printf("%d", (int) strlen(c + 2[p] - 6[p] - 1));
return 0;
}
```

nothing but *(2+p), *(6+p)



so → c + 2[p] - 6[p] - 1
= 2000 + *(p+2) - *(6+p) - 1
= 2000 + *(2002) - *(2006) - 1
= 2000 + 'T' - 'I' - 1
= 2000 + 12 - 1 - 1
= 2010
=) Strlen(2010)

We don't know ASCII of T, I.
So Assume I = 1.



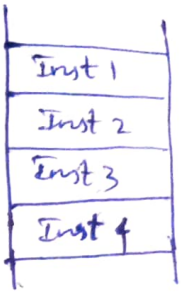
Null

o/p => 2 [don't add '\0']

148. Function pointers in C :-

Function pointers are like normal pointers but they have the capability to point to a function.

```
int fun (int a, int b)
{
  return a+b;
}
```



Example:-
How to declare a pointer to an array?

```
int main()
{
int *ptr[10]; ← Wrong
}
```

Note:-
Always remember precedence of operators.
(ie) precedence of [] is higher than *.
So, * is fighting with [] to take ownership of ptr.

So now, ptr is an array of 10 pointers pointing to integers.
This is not that we want ort? we want pointer to array not the array of pointers.

Solution:-

```
int main()
{
int (*ptr)[10];
}
```

now * belongs to ptr.
ptr is a pointer which pointing to an array of 10 integers.

149. Application of function pointers in C:-

Suppose we want to call a function named fun() at certain point in time in our code.

```
int func (int a, int b)
{ return(a+b);
}
```

```
int main() {
printf("%d", fun(2,5));
return 0;
}
```

There is nothing that has to be decided at runtime

There are some situations in which user has to decide which function has to be called at particular point in time.

Let say we want to design a Calculator which has the capability to perform addition, subtraction, multiplication and division.

Here the user will decide which operation he/she wants to perform.

Suppose, we have decided to create separate functions for these operations.

Now we want user to decide which function has to be called at the run-time.

One way is to use if/switch Case expressions

Another way is to use function pointers.

using function pointers;

```
//Designing a Calculator program using function pointers
#include<stdio.h>
#define ops 4
float Sum (float a, float b) { return a+b; }
float sub (float a, float b) { return a-b; }
float mult (float a, float b) { return a*b; }
float div (float a, float b) { return a/b; }

int main(){
float (*ptr2func[ops])(float, float) = { sum, sub, mult, div };
int choice;
float a,b;
Printf("Enter your choice: 0 for sum, 1 for sub, 2 for mult, 3 for div: \n");
Scanf("%d", &choice);
Printf("Enter the two numbers: \n");
```

```
scanf("%.1f %.1f", &a, &b);
printf("%f", ptr2fun[choice](a,b));
return 0;
}
```

150. Introduction to structure:-

problem statement:- I have a garage.
I want to store all information about cars which are available in my garage.

Creating variables for 100 cars → Not good idea.

Array is also not a good option → Array has capability to store more than one elements but they all must be of same type.
- our requirement is to store data of different types.

what happens if we can define our own type which can accomodate all the required types?

Structure is one stop solution.

→ Definition:- Structure is user defined data type that can be used to group elements of different types into single type.

151. Declaring structure variable:-

Eg:
```
struct {
    char * Engine;
    char * fuel_type;
    int fuel_tank_cap;
} car1, car2;
```

Eg:
```
struct {
    char * engine;
} Car1, Car2;
```

⇒ It is in global scope. Hence it is visible to all the functions.

```
int main() {
Car1.engine = "DDis 190 Engine";
Car2.engine = "1.2L Kappa";
printf("%s\n", Car1.engine);
printf("%s", Car2.engine);
return 0;
}
```

O/P:-

DD is 190 Engine
1.2L Kappa Dual VVT

152. Structure types (Using structure tags) :-

```
#include <stdio.h>
struct {
    char *name;
    int age;
    int salary;
} emp1, emp2;

int manager()
{
 struct {
 char * name;
 int age;
 int salary;
 } manager;
 manager.age = 27;
 if(manager.age > 30)
  manager.salary = 65000;
 else
   manager.salary = 55000;
 return manager.salary;
 }

   int main()
{
printf("Enter the Salary of employee 1: ");
scanf("%d", &emp1.salary);
printf("Enter the salary of employee 2: ");
scanf("%d", &emp2.salary);
printf("Employee 1 Salary is: %d\n", emp1.salary);
printf("Employee 2 Salary is: %d\n", emp2.salary);
printf("Manager's Salary is %d", manager());
return 0;
}
```

Need of creating a type:-

```
struct{
 char *name;
 int *age;
 int *salary;
} emp1, emp2;

int manager()
{
 struct {
 char *name;
 int age;
 int salary;
 } manager;
 .
 .
}
```

Here "manager" variable need to be create as local. meanwhile emp1 and emp2 are as global. Both structure are same. If we use manager above, it will be come global. But our requirement is to make manager as local.

→ So again need to do statements in the function. This makes redundancy

To avoid, we have to use "tag"

```
int manager()
{
 struct employee manager;
 ...
}
```

```
struct employee {
    char *name;
    int age;
    int salary;
};
```

structure tag

structure tag: It is used to identify a particular kind of structure

This is also valid.

```
struct employee {
    char *name;
    int age;
    int salary;
} emp1,
```

53. structure types (using typedef) :-

Syntax:- typedef existing_data_type new_data_type.

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

for example:-

```
#include <stdio.h>
typedef int INTEGER;
int main(){
INTEGER var=100;
printf("%d",var);
return 0;
}
```

```
Struct Car{
char engine[50];
float city_mileage;
};
nt main(){
Struct Car c1;
}
```

⇒

```
typedef struct Car {
        char Engine[50];
        float city_mileage;
        } Car;
int main(){
    struct Car c1;
}
```

(struct Car → old type; Car → new type)

Note:- Car becomes a new data type

54. Initializing & Accessing structure Members:-

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

Wrong way of Initialization.

Allowed:-

```
struct abc { int p;
             int q;
           };
int main()
{ struct abc x={23,34};
}
```

Accessing members of structure:

We can access members of the structure using dot (.) operator.

```
Struct Car {
    int fuel_tank_Cap;
} C1, C2;
int main() {
C1.fuel_tank_Cap = 45;
C2.fuel_tank_Cap = 30;
printf("%d %d", C1.fuel_tank_Cap, C2.fuel_tank_Cap);
return 0;
}.
```

o/p: 45, 30.

155. Designated Initialization:-

Designated initialization allows structure members to be initialized in any order.

```
Struct abc {
    int x;
    int y;
    int z;
};
int main() {
Struct abc a = {.y = 20, .x = 10, .z = 30};
printf("%d %d %d", a.x, a.y, a.z);
return 0;
}
```

→ Don't forget to use dot operator while accessing the members of structure otherwise x = 20, y = 10, z = 30

o/p: 10, 20, 30.

Declaring an 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.

```
struct Car {
    int fuel_tank_cap;
    float city_mileage;
};
int main() {
struct Car c[2];
int i;
for(i=0; i<2; i++)
{
printf("Enter car %d fuel tank capacity:", i+1);
scanf("%d", &c[i].fuel_tank_cap);
printf("Enter car %d city_mileage", i+1);
scanf("%f", &c[i].city_mileage);
}
printf("\n");
for(i=0; i<2; i++)
{
printf("\nCar %d details:\n", i+1);
printf("fuel tank capacity: %d\n", c[i].fuel_tank_cap);
printf("Seating capacity: %d\n", c[i].seating_cap);
printf("City mileage: %f\n", c[i].city_mileage);
}
return 0;
}
```

157. pointer to structure Variable:-

program:-

```
Struct abc {
 int x;
 int y;
 }
int main () {
 Struct abc a={0,1};
 Struct abc *ptr = &a;
 printf("%d %d", ptr->x, ptr->y);
 return 0;
 }
```

ptr is a pointer to some Variable of type Struct abc.

ptr->x => (*ptr).x
=> (*&a).x
=> a.x

158. structure padding:-

Eg:
```
Struct abc {
   Char a;// 1 byte
   Char b;// 1 byte
   int c; // 4 bytes
 } var;
```
Total size: 6 bytes
↓
This is wrong.

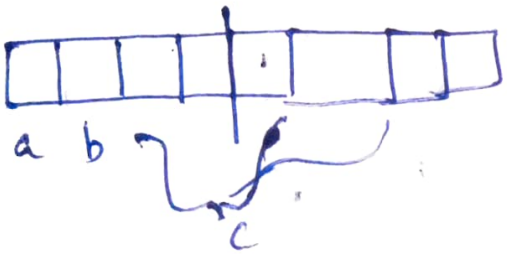
Structure padding:-

-> 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 we have 64 bit processor then it means it Can access 8 bytes at a time which means word size is 8 bytes.

In 32 bit architecture,



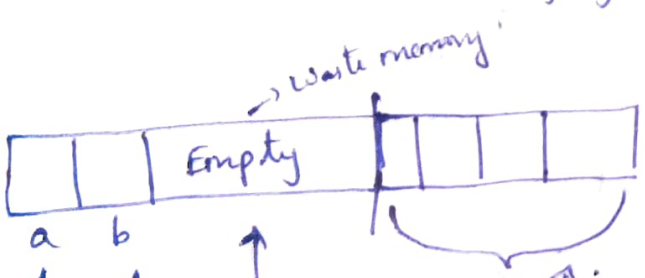
In 1 cpu cycle, char a, char b and 2 bytes of int c can be accessed.

There is no problem with char a and char b but whenever we want the value stored in variable c, 2 cycles are required to access the contents of variable c. In first cycle, 1st 2 bytes can be accessed and in 2nd cycle last 2 bytes.

⇒ It is unnecessary wastage of CPU cycle.

⇒ we can save the number of cycles by using concept called "padding".

Now,

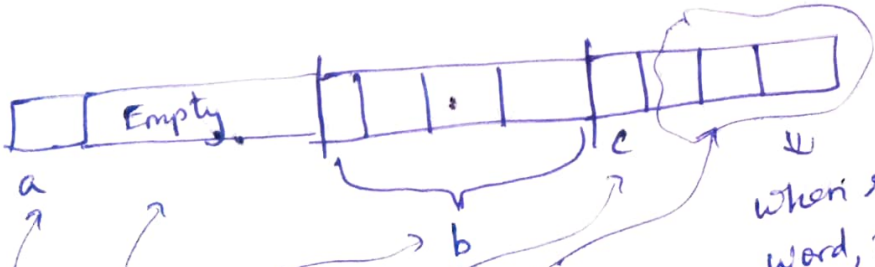


∴ Cycles are reduced

Total: 1byte + 1byte + 2bytes + 4bytes = 8 bytes. and 'c' can be read in 1 cpu cycle.

But what happens if we change the order of members?

```
Struct abc{
char a;
int b;
char c;
};
```



⇒ 1 + 3 + 4 + 1 + 3

⇒ 12 = 3 words accessed.

When read as word, this comes as additionally..

159. structure packing:-

Because of structure padding, size of structure becomes more than size of the actual structure. Due to this some memory will get wasted.

We can avoid the wastage of memory by simply writing

#pragma pack(1).

Eg:

```
#pragma pack(1)
struct abc {
    char a;
    int b;
    char c;
} var;
int main() {
printf("%d", sizeof(var));
}
```

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

Note:-

cpu cycles are wasted here.

o/p:- 6 [instead of 12]

→ It is all up to us.

→ If we don't want to waste cpu cycles, structure ~~packing~~ padding is available.

→ If we don't want to waste memories, structure packing can be used. [#pragma pack(1)]

159. structures in c (solved problem 1):-

predict the o/p:-

```
#include <stdio.h>
struct point
{
 int x, y, z;
};
int main()
{
struct point p1={.y=0, .z=1, .x=2};
printf("%d %d %d", p1.x, p1.y, p1.z);
return 0;
}
```

a) 2 0 1
b) Compiler Error
c) 0 1 2
d) 2 1 0

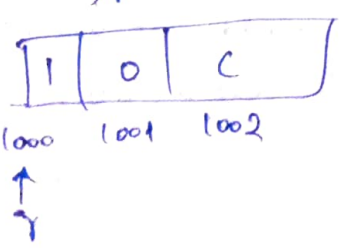
160. structures in c (solved prblm 2):-

```
#include <stdio.h>
struct ournode {
 char x, y, z;
};
int main(){
struct ournode p={'1', '0', 'a'+2};
struct ournode *q=&p;
printf("%c, %c", *((char*)q+1), *((char*)q+2));
return 0;
}
```

o/p:-

a) 0, c
b) 0, a+2
c) '0', 'a+2'
d) '0', 'c'

Type Casting



162. structures in C (solved prblm 3)

```
struct node
{
  int i;
  float j;
};
struct node *s[10];
```

define s to be

a) an array each element of which is a pointer to structure of type node.

b) A structure of 2 fields, each field being a pointer to an array of 10 elements.

c) A structure of 3 fields: an integer, a float, and an array of 10 elements.

d) An array, each element of which is a structure of type node.

Here fighting occurs between * and [], due to precedence high, brackets will win.
So, array of pointers which will point to structure type.

163. Introduction to unions in C:-

union is a user defined data type but unlike structures, union members share same memory location.

Example:

```
struct abc {
  int a;
  char b;
};
```

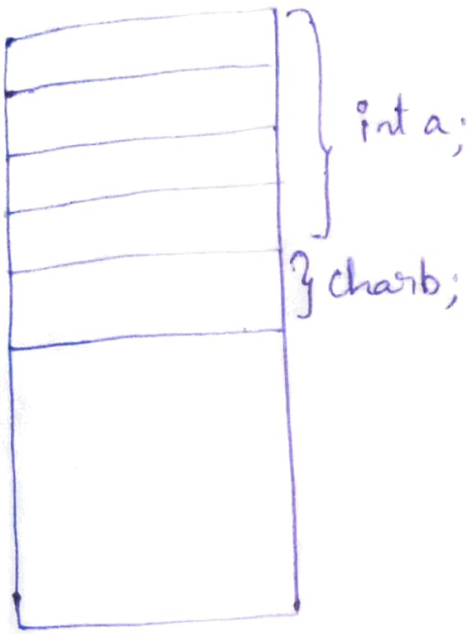
a's address = 6295624
b's address = 6295628

```
union abc {
  int a;
  char b;
};
```

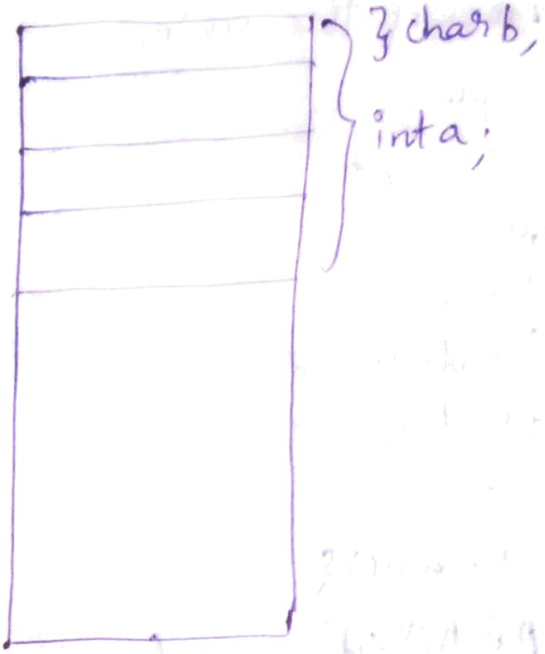
a's address = 6295616
b's address = 6295616

Memory allocation

struct abc



union abc



FACT:-

In union members will share same memory location. If we make changes in one member then it will be reflected to other members as well.

Example:-

```
union abc {
 int a;
 Char b;
}Var;
int main(){
Var.a = 65;
printf("a = %d\n",Var.a);
printf("b= %c",Var.b);
return 0;
 }
```

o/p:-

a = 65
b = A

Deciding the size of union:-

Size of the union is taken according to the size of the largest member of the union.

Example:-

```
union abc {
int a;
char b;
double c;
float d;
};
int main() {
printf("%d", sizeof(union abc));
return 0;
}
```

o/p:
8

Accessing Members using pointers:-

We can access members of union through pointers by using the arrow (->).

```
union abc {
int a;
char b;
};
int main() {
union abc var;
var.a = 90;
union abc *p = &var;
printf("%d %c", p->a, p->b);
return 0;
}
```

o/p:
90 Z

of unions in c (solved prblm 1):-

```
struct{
short s[5];
union {
float y;
long z;
}u;
}t;
```

Assume that objects of type short, float and long occupy 2 bytes, 4 bytes and 8 bytes respectively. The memory requirements for variable t, ignoring alignment considerations is

a) 22 bytes
b) 14 bytes
c) 18 bytes
d) 10 bytes

Application of unions (part-1)

A store sells two kinds of items: 1. Book 2. shirts

Store owners want to keep records of above mentioned items along with relevant information.

Books: Title, Author, number of pages, price
Shirts: Color, size, design, price

Initially, they decided to create a structure like below.

```
struct store
{
double price;
char *title;
char *author;
int num_pages;
int colour;
int size;
char *design;
};
```

This structure is perfectly usable but only price is common property in both items and rest are individual.

```
#pragma pack(1)
struct store
{
 double price; // 8 bytes
 char * title; // 8 bytes
 char * author; // 8 bytes
 int num_pages; // 4 bytes
 int colour; // 4 bytes
 int size; // 4 bytes
 char *design; // 8 bytes
};
```

char pointer

```
int main(){
 struct store book;
 book.title = "The Alchemist";
 book.author = "paulo Coelho";
 book.num_pages = 197;
 book.price = 23; // dollars
 return 0;
}
```

Book variable doesn't possess these properties. Therefore its a wastage of memory.

same

now

```
int main(){
 struct store book;
 printf("%ld bytes", sizeof(book));
 return 0; }
```

o/p:- 44 bytes

We can save lot of space if we start using unions.

Same application in union

```
#pragma pack(1)
struct store{
double price; // 8 bytes  ⟶  8 bytes
union{
    struct {
      char * title; // 8 bytes
      char * author; // 8 bytes
      int num_pages; // 4 bytes
    } 20 bytes
  } book;
```

```
struct {
int colour; // 4 bytes
int size; // 4 bytes
char * design; // 8 bytes
} shirt;
} item;
};
```

16 bytes

```
int main () {
struct store s;
s.item.book.title = "The Alchemist";
s.item.book.author = "paulo coelho";
s.item.book.num_pages = 197;
printf("%s\n", s.item.book.title);
printf("%ld", sizeof(s));
return 0;
}
```

o/p:-
The Alchemist
28

max(20 bytes, 16) ⇒ 20
+
8
28 ⇒ 28

28 < 44. Hence saving memory space.

16. Application of unions (part-2):-

Creating your own mixed type data structure.

```
typedef union {
int a; char b; double c; // 8 bytes
} data;
int main() {
data arr[10]; // ⇒ size = 80 bytes
arr[0].a = 10;
arr[1].b = 'a';
arr[2].c = 10.178
// and so on
return 0;
}
```

Succesful in creating an array containing mixed type data.

```
typedef struct {
int a; char b; double c; // size = 13 bytes
} data;
int main()
{
data arr[10]; ⇒ size = 130 bytes
arr[0].a = 10;
arr[1].b = 'a';
arr[2].c = 10.178;
// and so on
return 0;
}
```

167. Enumeration in C :-

An Enumerated type 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;
printf("%d", var);
return 0;
}
```

o/p :- 1

But we can also use #define to assign names to integral Constants. Then why do we even need enum?

Need of enumeration :-

Reason 1 :- Enums can be declared in the local scope.

```
int main() {
enum Bool { False, True } var;
var = True;
printf("%d", var);
return 0;
}
```

This enum is not visible outside this main function.

But #define must be used in before main (ie) in Global scope only

Reason 2 :- Enum names are automatically initialized by the Compiler.

```
int main() {
enum Bool { False, True } var;
var = True;
printf("%d", var);
return 0;
}
```

False is initialized to 0 and True to 1.

Some Important facts:-

Fact 1:- Two or more names can have same values.

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

o/p:- 0 0 0

Fact 2:- 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;
}
```

o/p: 34 2 0 35

Fact 3:- Only Integral values are allowed.

```
int main(){
enum point {x=34, y=2.5, z=0};
printf("%d %d %d", x, y, z);
return 0;
}
```

error:- Enumerator value for 'y' is not an integer constant.

Fact 4:- All enum constant must be unique in their scope.

```
int main(){
enum point1 {x=34, y=2, z=0};
enum point2 {x=4, p=25, q=1};
printf("%d %d %d %d %d", x, y, z, p, q);
return 0;
}
```

Error:- Redeclaration of enumerator 'x'.

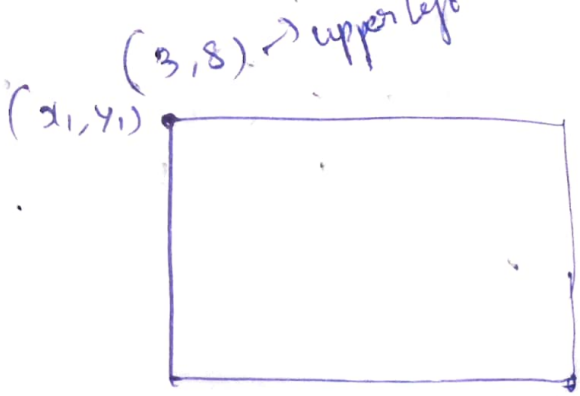
168. Qn: The following structures are designed to store the information about objects on a graphics screen.

Struct point {int x, int y};

Struct rectangle { struct point upper_left, lower_right; };

A point structure stores x and y co-ordinates of a point on the screen. A rectangle structure stores the co-ordinates 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.



Condition:- $x2 > x1$ and $y1 > y2$ for rectangle. (14 > 3 and 8 > 2).

Code:-

```
#include <stdio.h>
struct point { int x; int y; };
struct rectangle {struct point upper_left, lower_right; };
int area (struct rectangle r)
{
  int length, breadth;
  length = r.lower_right.x - r.upper_left.x;
  breadth = r.upper_left.y - r.lower_right.y;
  return length * breadth;
}
```

```
int main() {
struct rectangle r;
printf("Enter the upper left co-ordinates of rectangle : \n");
scanf("%d %d", &r.upper_left.x, &r.upper_left.y);
printf("Enter the lower-right co-ordinates of rectangle : \n");
scanf("%d %d", &r.lower_right.x, &r.lower_right.y);
printf("Area of rectangle : %d", area(r));
return 0;
}
```

Note :- I have done the same with pointer and also include the length, breadth condition. refer area_rect_struct.c file.

9. Structure & unions in c (solved prblm),

Qn1:- Suppose that s is the following structure:

```
struct {
  double a; // 8 bytes
  union {
     char b[4]; // 4 bytes
     double c; // 8 bytes
     int d;  // 4 bytes
  } e; // 8 bytes
  char f[4]; // 4 bytes
} s;
```

How much space will c Compiler allocate for s ? (Assume structure packing).

→ 20 bytes

Qn2:-

```
union {
   double a; // 8
   struct {
      char b[4]; // 4
      double c; // 8
      int d;  // 4
   } e; // 16
   char f[4]; // 4
} u;
```

How much space will c Compiler allocate for u ? (Assume structure packing)

→ 16 bytes

170. File handling → alone topic - 18 lectures

171 - 184. Data structures.

185. Understanding Void pointer :-

Void pointer is a pointer which has no associated data type with it. It can point to any data of 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;
}
```
o/p:- 10

```
If printf("%d", *ptr);
```
↳ we cannot dereference a void pointer. o/p:- Error.

Why we use void * and do typecast? we can use int *? why this unwanted thing?

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

Syntax:- `void *malloc(size_t size);`

Note:- Malloc and Calloc are used to allocate memory at runtime. Malloc, Calloc doesn't care about data type, simply creates the memory block and return address of that block as void pointer.

186. Null pointer :-

What is Null pointer?

A Null pointer is a pointer that doesnot point to any memory location. It represents an invalid memory location.

When a NULL value is assigned to pointer, then pointer is considered as NULL pointer.

Uses:-

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

Eg: int main(){ → It is also Null pointer.
int *ptr = NULL; "Null" word is also null pointer.
return 0;
}

2. useful for handling errors when using Malloc function.

Eg:-

```
int main(){
int *ptr;
ptr = (int *) malloc (2 * sizeof(int));
if(ptr == NULL)
printf("Memory Could not be allocated");
else
printf("Memory allocated Succesfully");
return 0;
}
```

Facts about Null pointer:-

1. 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 the integer 0.

Eg: int main(){
int *ptr = NULL;
printf("%d", ptr);
return 0;
}

o/p: 0

and

int *ptr = 0;
printf("%d", ptr);

o/p: 0

2. Size of NULL pointer depends upon the platform and is similar to the size of the normal pointers.

Eg:- int main(){
printf("%d", sizeof(NULL)); o/p:- 8.
return 0;
}

Best practices:-

-> It is good practice to Initialize a pointer as NULL.

-> It is good practice to perform NULL check before dereferencing any pointer to avoid surprises.

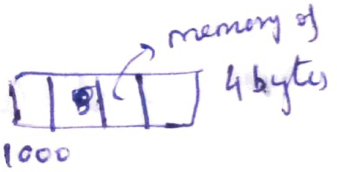
187. Dangling pointer:-

A dangling pointer is a pointer which points to some non-existing memory location.

Eg):-

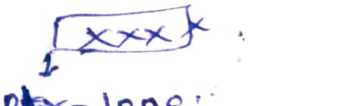
```
int main()
{
int *ptr = (int*)malloc(sizeof(int));
. . .
. . .
free(ptr);
return 0;
}
```

Assume



ptr = 1000;

when we free(), actually release allocated memory. (i.e) Deallocating.



ptr = 1000;

Now pointer is still pointing to the deallocated memory (i.e) Dangling.

Solution:-

```
. . .
. . .
free(ptr);
ptr = NULL;
```

-> Now ptr is no more dangling.

Example 2:-

```
#include <stdio.h>
int *fun()
{ int num=10;
  return &num;
}
int main() {
int *ptr = NULL;
ptr = fun(); 
printf("%d", *ptr);
return 0;
}
```

→ returning local variable address

→ obviously ptr will become dangling pointer.

o/p :- Segmentation fault

Dashboard

8 Cars → Port
5 → following
3 → Nevada

2020 April - Sept
2021 April - Sept

(18)

What is Wild pointer?

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) misbehave.

Eg:
```
int main()
{
    int *p;
    *p = 10; ⟶ wild pointer
    return 0;
}
```

How to avoid wild pointers?

① Initialize them with address of a known variable.

```
int main()
{
    int var = 10;
    int *p;
    p = &var; ⟵ No more a wild pointer
    return 0;
}
```

② Explicitly allocate the memory and put values in the allocated memory.

Example:
```
int main()
{
    int *p = (int *)malloc(sizeof(int));
    *p = 10;
    free(p);
    return 0;
}
```

189:- Static Memory Allocation:-

Memory allocated during Compile time is called static memory. The memory allocated is fixed and cannot be increased (or) decreased during run time.

Eg:-
```
int main()
{
 int arr[5]={1,2,3,4,5};
}
```

Memory is allocated at Compile time and fixed

Problems faced in static memory allocation:-

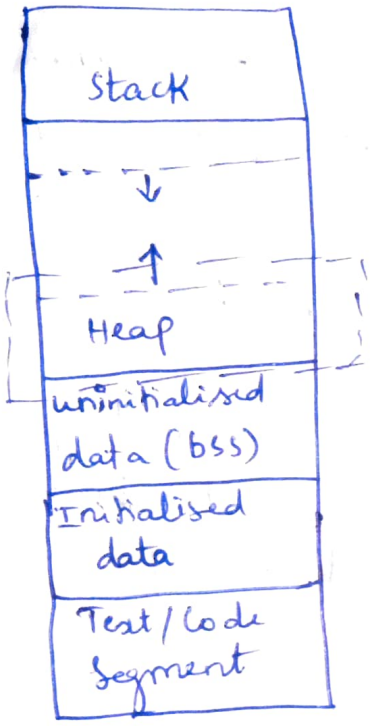
→ If you are allocating memory for an array during Compile time then you have to fix the size at the time of declaration. Size is fixed and user cannot increase (or) decrease the size of the array at run time.

→ If values stored by user in the array at run time is less than the size specified then there will be wastage of the memory.

→ If values stored by user in the array at run time is more than the size specified then the program may crash (or) misbehave.

→ Dynamic memory allocation:-

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



Heap is the segment of memory where dynamic memory allocation takes place.

Unlike Stack where memory is allocated or deallocated in a defined order, heap is an area of memory where memory is allocated or deallocated without any order at the random.

There are certain built-in functions that can help in allocating (or) deallocating some memory space at run time.

→ pointers play an important role in dynamic memory allocation.

→ Allocated memory can only be accessed through pointers.

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

Built-In functions:-

→ malloc()
→ calloc()
→ realloc()
→ free()

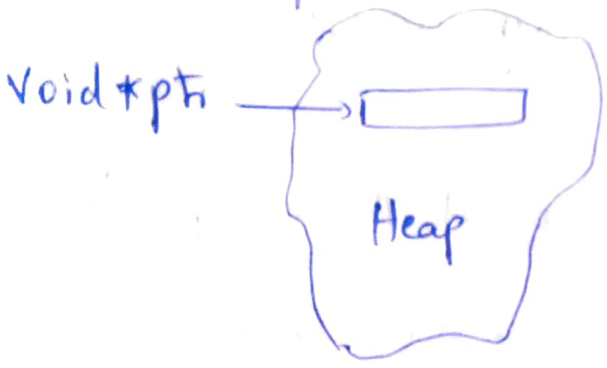
190. Malloc():-

- Malloc is a built-in function declared in header file <stdlib.h>
- Malloc is the short name for "memory allocation" and is used to dynamically allocate a single large block of the contiguous memory according to the size specified.

Syntax:- (void *) malloc (size_t size)

malloc function simply allocates a memory block according to the size specified in the heap and on success it returns a pointer pointing to first byte of the allocated memory else returns NULL.

Size_t is defined in <stdlib.h> as unsigned int.



Why void pointer?

→ malloc doesn't have an idea of what it is pointing to.

→ It merely allocates memory requested by the user without knowing type of the data to be stored inside the memory.

Void pointer can be typecasted to an appropriate type

(int *) ptr = (int *) malloc(4)

→ malloc allocates 4 bytes of memory in the heap and the address of the first byte stored in the pointer ptr.

Eg:-

```
int main(){
int i,n;
printf("Enter the number of integers:");
Scanf("%d",&n);
int *ptr = (int *)malloc(n*sizeof(int));
if(ptr == NULL){
printf("Memory not available");
exit(i); }
for(i=0;i<n;i++){
printf("Enter an integer: ");
Scanf("%d",ptr+i); }
for(i=0;i<n;i++)
printf("%d",*(ptr+i));
return 0;
}
```

19. CALLOC

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

It is different from malloc in two ways.

① → calloc() needs two arguments instead of just one.

Syntax: void *calloc(size_t n, size_t size);

n: No. of blocks

size: Size of each block

Example:

```
int *ptr = (int*) calloc(10, sizeof(int));
```

An Equivalent malloc call-

```
int *ptr = (int*)malloc(10 * sizeof(int));
```

② Memory allocated by calloc is initialized to zero.

> It is not the case with malloc. Memory allocated by malloc is initialized with some garbage value.

Note:- malloc and calloc both return NULL when sufficient memory is not available in heap.

CALLOC stands for clear allocation.

MALLOC stands for Memory allocation.

192. Realloc():-

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

Syntax:- `void * realloc(void *ptr, size_t newsize);`

`ptr` → pointer to previously allocated memory.

Eg:- `int *ptr = (int *)malloc(sizeof(int));`

`ptr = (int *)realloc(ptr, 2*sizeof(int));`

→ This will allocate memory space of 2*sizeof(int).

→ Also, this function moves the contents of old block to a new block and data of old block is not lost.

→ We may lose the data when new size is smaller than old size.

→ Newly allocated bytes are uninitialized.

Eg:-
```
int main()
{ int i;
  int *ptr = (int *)malloc(2*sizeof(int));
  if(ptr == NULL)
  {
  printf("Memory not available!");
  exit(1);
  }
```

```
printf("Enter two numbers:\n");
for(i=0;i<2;i++){
 scanf("%d", ptr+i);
 }
ptr=(int *)realloc(ptr, 4*sizeof(int));
if(ptr==NULL)
{
 printf("Memory not available!");
 exit(1);
}
printf("Enter 2 more integers:\n");
for(i=2;i<4;i++)
scanf("%d", ptr+i);
for(i=0;i<4;i++)
printf("%d", *(ptr+i));
return 0;
 }
```

193. free():-

free() function is used to release the dynamically allocated memory in heap.

Syntax: void free(ptr)

The memory allocated in heap will not be released

automatically after using the memory. The space remains there and can't be used.

It's programmer's responsibility to release the memory after use.

Eg:-

```
int main()
{
 int *ptr = (int *) malloc( 4 * sizeof(int));
 .....
 free(ptr);
}
```

Eg:-

```
 int *input()
{
   int *ptr, i;
   ptr = (int *) malloc ( 5 * sizeof(int));
   printf("Enter 5 numbers: ");
   for(i=0; i<5; i++)
    scanf("%d", ptr+i);
    return ptr;
}
 int main()
 {
   int i, sum=0;
   int *ptr = input();
   for(i=0; i<5; i++)
       sum += *(ptr+i);
   printf("Sum is: %d", sum);
   free(ptr);
   ptr = NULL;
   return 0;
 }
```

170. Structures and functions:-

Quick recap:- Structure is a user defined data type that can be used to group elements of different types into a single type.

① Passing structure member as argument:-

Just like variables, we can pass structure members as arguments to a function.

```
Struct Student {
  Char name[50];
  int age;
  int roll_no;
  float marks;
}
Void print(char name[], int age, int roll, float marks) {
 printf("\n %s %d %.2f", name, age, roll, marks); }
int main() {
  Struct Student S1 = {"Nick", 16, 50, 72.5};
  printf(S1.name, S1.age, S1.roll_no, S1.marks); }
```

Call by reference:-

Instead of passing the copies of the structure members, we can pass their addresses (or references).

```
struct charset{
    Char s;
    int i; };
Void Keyvalue (char*s, int*i){
 Slanf("%c%d", s,i); }
 int main(){
  int j;
  Struct charset cs;
  Key value (&(s.s, &(s.i);
  print ("%c%d", cs.s, cs.i);
  return 0;
  }
```

-> having high priority than &.
so we don't need to set brackets.

171. Structure variable as an argument :-

Instead of passing structure members individually, it is a good practice to pass a structure variable as an argument.

unlike arrays, name of the structure variables are not pointers.

It's a pass by value not address.

Eg:-

```
struct point { int x; int y; };
void print(struct point p) {
printf("%d %d\n", p.x, p.y);
int main() {
 struct point p1 = { 23, 45 };
 struct point p2 = { 56, 90 };
 print(p1);
 print(p2);
 return 0;
 }
```

o/p:-

```
23 45
56 90
```

172. Passing pointers to structures as arguments:-

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

Better choice:- pass the address of the structure

* use arrow operator (->) to access the structure members inside the called function.

Eg:

```
struct point {
    int x;
    int y;
    };
```

```
Void print (struct point *ptr){
printf("%d %d\n", ptr->x, ptr->y); }
int main(){
Struct point p1={23,45};
Struct point p2={56,90};
print(&p1);
print(&p2);
return 0;
}
```

173. Returning a structure variable from function:

It is similar to returning a variable from function.

```
Struct point{
 int x;
 int y; };
Struct point edit (struct point p){
 (p.x)++;
  p.y=p.y+5;
 return p;
}
Void print(struct point p){
printf("%d %d\n",p.x,p.y);
}
```

```
int main(){
Struct point p1={23,45};
Struct point p2={56,90};
p1= edit(p1);
p2= edit(p2);
print(p1);
print(p2);
return 0;
}
```

174. Returning a pointer to structure from function:-

```
Struct point{ int x; int y;};
Struct point * fun(int a, int b){
Struct point *ptr = (struct point*) malloc (sizeof(struct point));
 ptr->x= a;
 ptr->y= b+5;
  return ptr;
}
void print(struct point *ptr){
  printf("%d %d\n", ptr->x, ptr->y);
}
```

```
int main() {
Struct point *ptr1, *ptr2;
ptr1 = fun(2,3);
ptr2 = fun(6,9);
print(ptr1); print(ptr2);
free(ptr1); ⟶ releasing heap memory.
free(ptr2); ⟶
return 0;
}
```

Why we use malloc() in this code?

Reason is malloc creates memory in heap. Heap memory is fixed and not be deallocated automatically. So, here returning the address is not problem. But if we do static memory, once fun over, automatically memory is deallocated and we can't return address. This will be problem.

175. Passing array of structures as argument:

Compiler will allocate contiguous block of memory for the data members of the structure.

Eg:-

```
struct point { int x; int y};
void print (struct point arr[]){
 int i;
 for(i=0; i<2 ;i++)
 printf("%d %d\n", arr[i].x, arr[i].y); }
int main(){
struct point arr[2] = {{1,2}, {3,4}};
print(arr);
 return 0;
 }
```

176. Self referencial structures:-

Those are structures in which one or more pointers points to the structure of the same type.

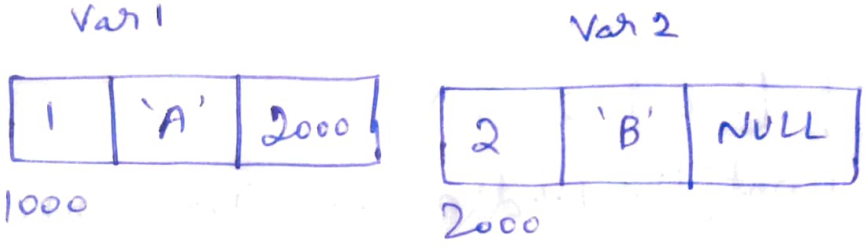
```
struct self {
    int p;
    struct self *ptr; };
```

Eg:-

```
struct code {
    int i;
    char c;
    struct code *ptr;
};
```

```
int main(){
Struct Code Var1;
Struct Code Var2;
Var1.i = 65;
Var1.c = 'A';
Var1.ptr = NULL;
Var2.i = 66;
Var2.c = 'B';
Var2.ptr = NULL;
Var1.ptr = &Var2;
}
```

Visualization



Var1.ptr = &Var2;

Where we use it?

Linked List.