weightage

Monday, July 29, 2024 3:07 PM

c + dsa

total : 18 to 22 marks

C programming -

(i) data types and operators .

(ii) control flow statements : if-else, loops, etc.

(iii) functions and storage classes.

(iv) arrays and pointers.

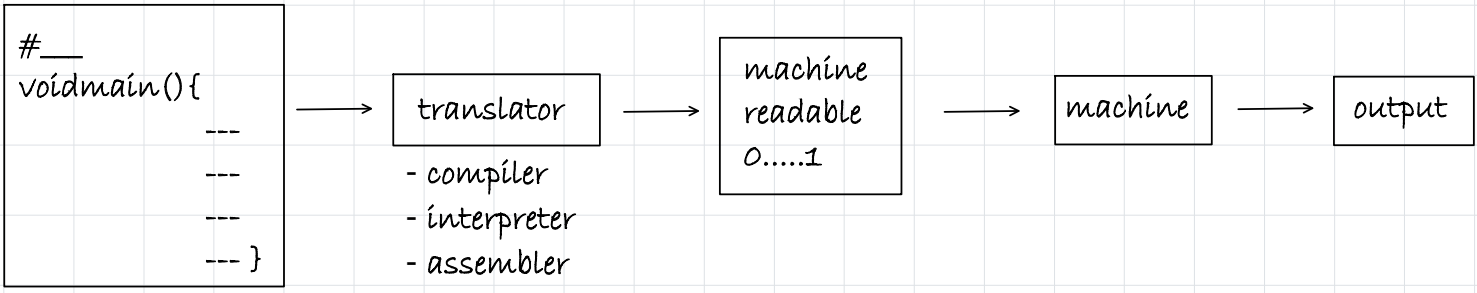
(v) strings.

(vi) structure and union.

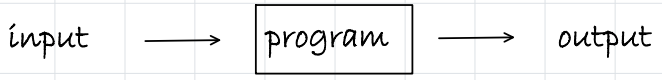
(vii) miscellanous topics.

introduction to C programming

program



ATM machine



credentials (pin number)

money

q. why we cannot write program in 0 an 1?
sol.
- it is difficult to debug the program as a single 0 or 1 can change the entire program and it will be difficult to find out which 0 or 1 was wrong.
- to perform addition we have to give the instructions in 0 and 1 and for every program we have to remember the codes and that will be difficult.
- we humans are not good with numbers.

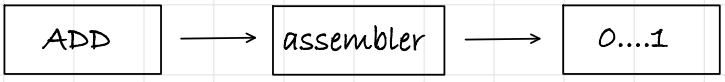
so, instead we started using letters

addition : ADD
subtraction : SUB
multipication : MUL

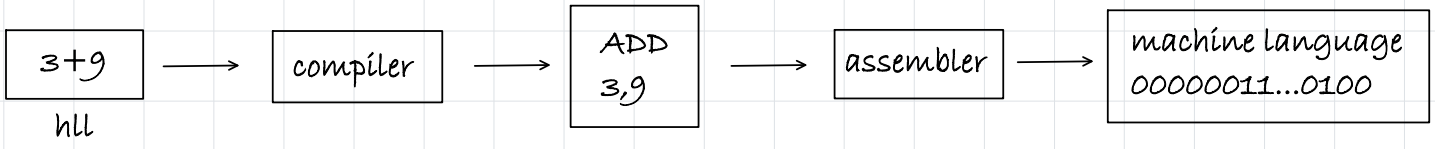
collections of these mnemonics are called assembly language.

00000011 00001001 change into ADD
(machine language) (assembly language)

- high level language :
  3+9
  a+b : hll (human readable)

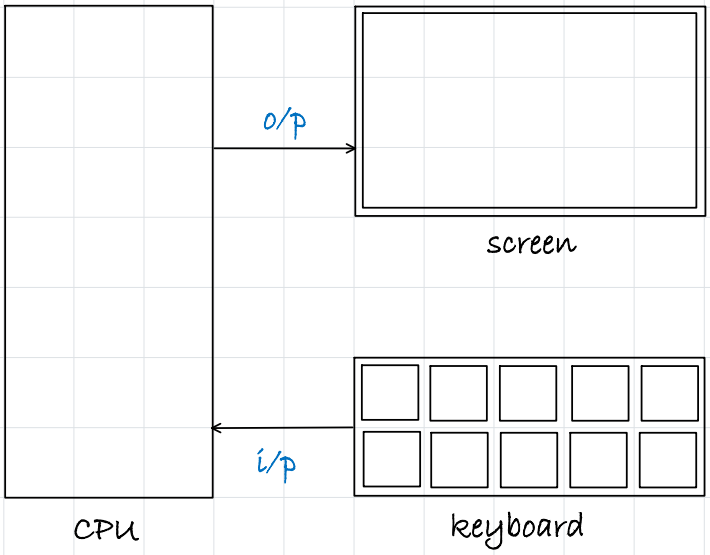


high level language to assembler to machine language :

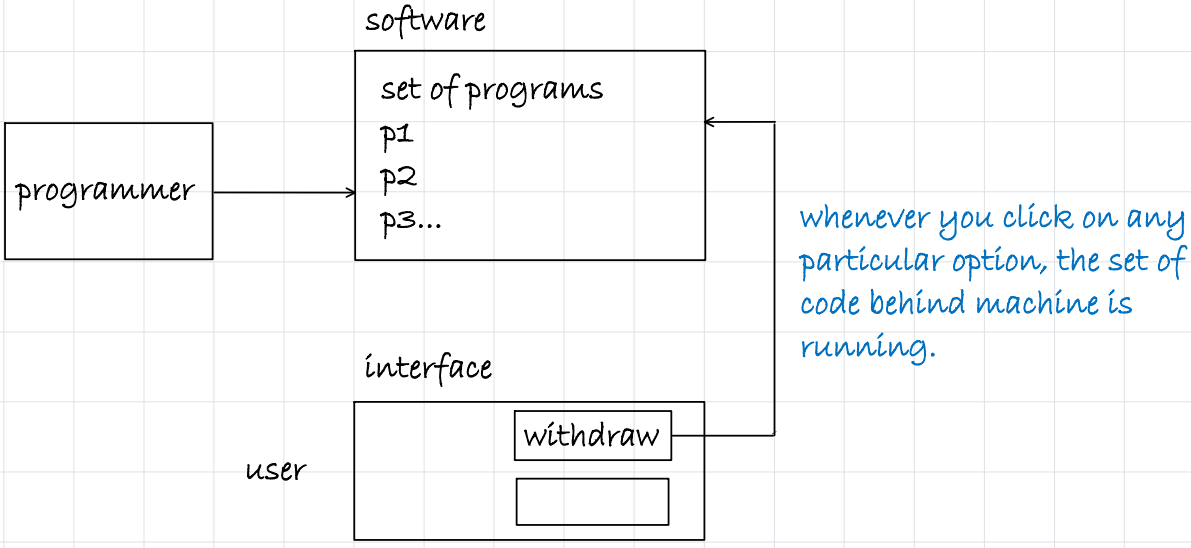


set of rules for program :

(a) every programming language must have the facility to take input coming from the keyboard.
(b) every programming language must have the facility to display output from program to screen.
(c) every programming language must have the set of grammar rules to follow.



programmer creating software for bank and user is accessing the programs through interface :



interface

- abstraction : hiding the important details which are not needed. like for an ATM user it is not neccesary for him to know the code and details about the program running behind the software in order to operate the ATM machine.

q. which software to use for C programs?

C related software.

whenver you install C related software there will be 2 things in picture :

(i) compiler
(ii) library

- what is library and why library?

program

```
#include<stdio.h>
voidmain(){
        printf ("aryan");
                }
```

printf : whatever you provide inside " " will be displayed on screen as it is.

when compiler is compiling the program, it will look for the code behind the printf, we did not write the code of printf but we are simply using it as a template and the code of printf is written inside the header file- #include<stdio.h>

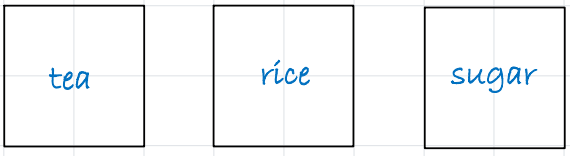
so, here we are using printf through header file (#include<stdio.h>) as an interface.

library : collection of these header files like 'printf'

so as a user we are able to use these pre-built codes through library as an interface

topic - variable
being able to change (vary)

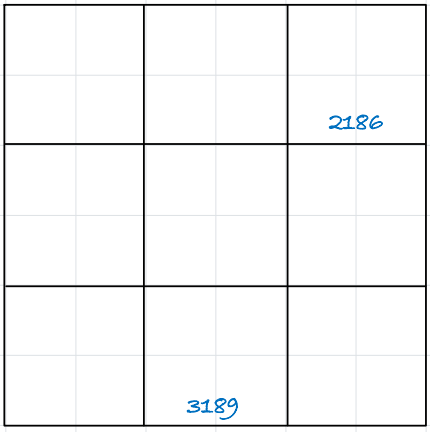
variables are just like the containers in the kitchen.



(a) (b) (c)

(a) (b) (c)

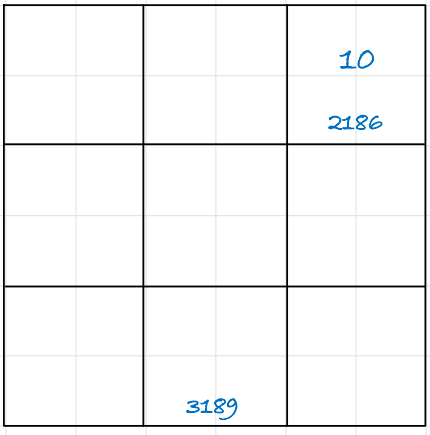
every container have
- types
- capacity



every memory block has an address

memory

randomly storing the data : 10



every memory block has an address

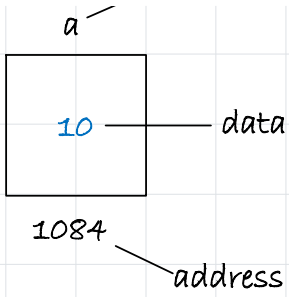
memory

the problem here is how we will retrieve the data?
this box doesnt have any name and there are lakhs of the blocks in the memory.

so in order to store the data we have to name it like "a = 10;"
here'a' is a variable that is basically an idenitifer.

identifier (name of variable)

a



topic - garbage value

the Garbage value is a random value at an address in the memory of a computer. Whenever a variable is defined without giving any value to it, it contains the leftover values from the previous program.

types of data

(i) Google maps

source Delhi

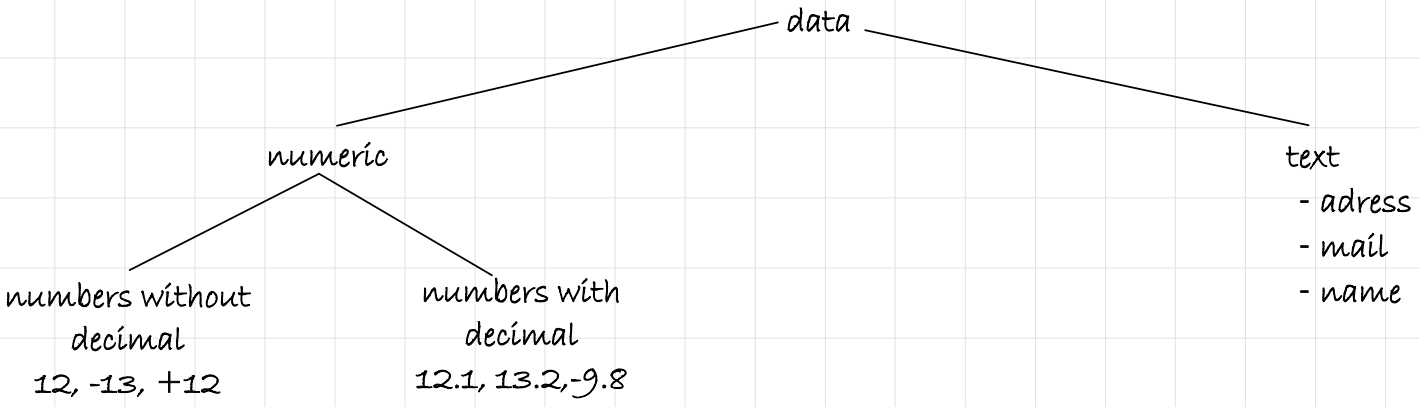
destination GOA

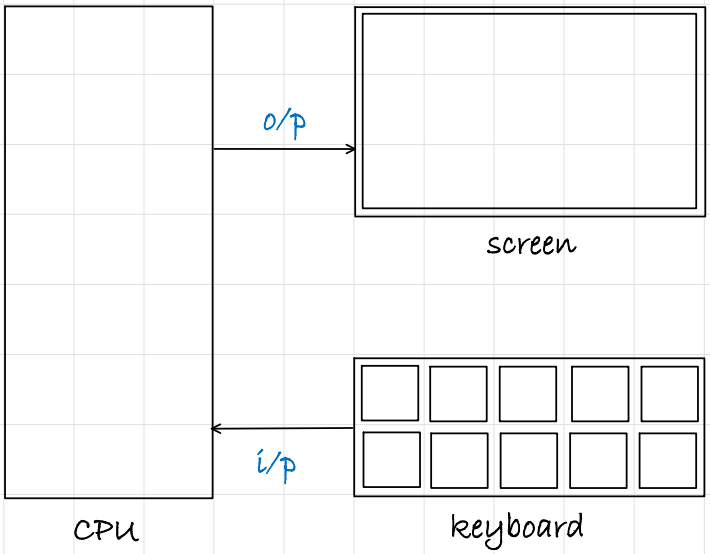
text type data

(ii) ATM machine

pin no. 1234 — numeric type data

data can be of different type.



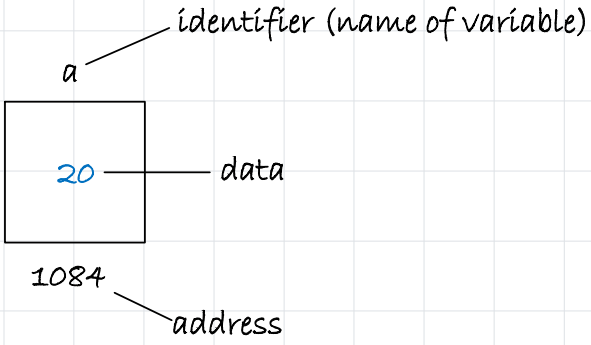


whatever we enter from the keyboard it will be the sequence of 0 and 1's

A : 0000 1111 (binary number)
value : 15

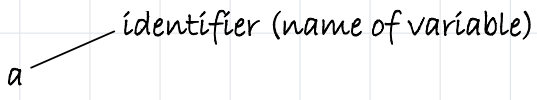
but we have to tell the program ki humein 0000 1111 ko "A" ki tarah deal karna hai ya "15" ki tarah aur iske lie aate hai data types.

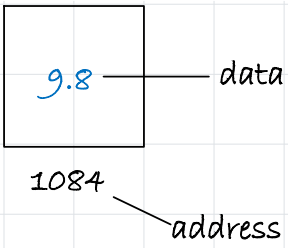
example :
(i) int a = 20;



'int' means integer, number without decimal point.

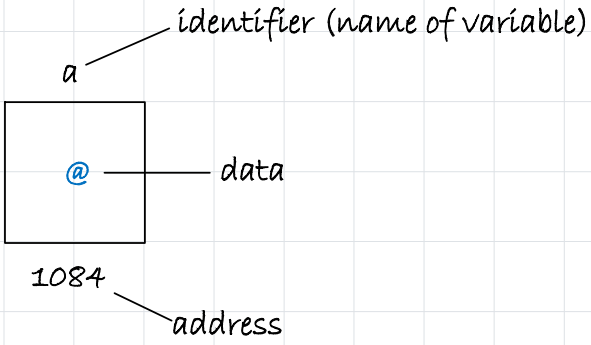
(ii) float b = 9.8;





'float' means number with decimal point.

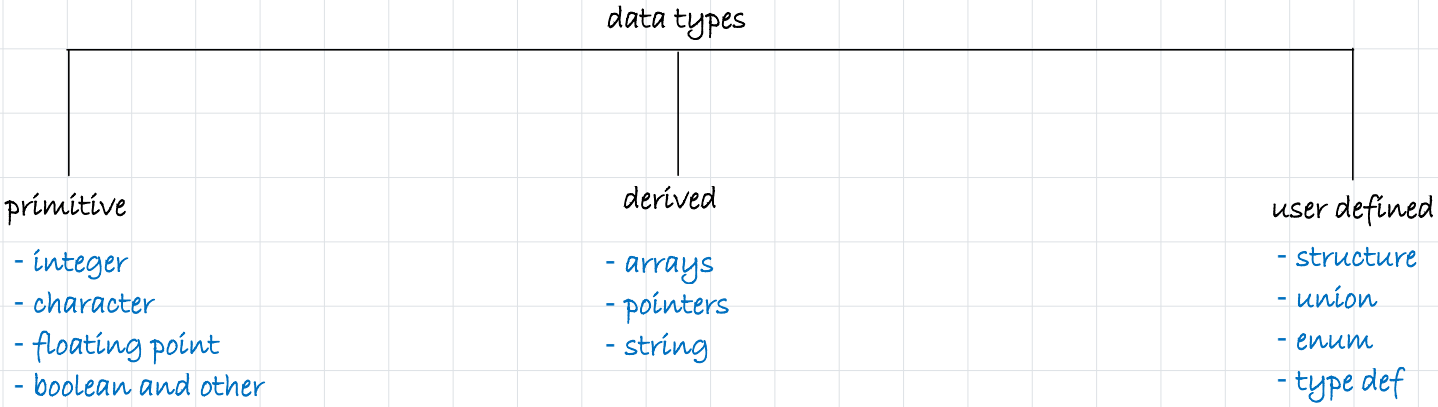
(iii) char c = "@";



to store one symbol 'char' data type is used.

so these keywords char, float, int are reserved words or keywords.

data types and operators



integer

(a) short int (minimum 2 byte) :

(i) signed integer : -12, +12 (negative and positive)

int a = 20; (or) signed int a = 20; is same.

(if you do not write signed or unsigned then compiler will take it as by default signed)

(ii) unsigned integer : 12 (starts from 0)

unsigned int age = 20;

in case of unsigned you have to mention, ex : age cannot be negative.

(b) int :

(i) signed integer : -12, +12

(ii) unsigned integer : 12

(c) long int :

(i) signed integer : -12, +12

(ii) unsigned integer : 12

(d) long long int :

(i) signed integer : -12, +12

(ii) unsigned integer : 12

(ii) floating point

(a) float

(i) signed integer : -12, +12

(ii) unsigned integer : 12

(b) double
(i) signed integer : -12, +12
(ii) unsigned integer : 12

(c) long double
(i) signed integer : -12, +12
(ii) unsigned integer : 12

decimal number system

possible values we can generate :

(i)

1st	2nd
10	10

10^2

(ii)

1st	2nd	3rd
10	10	10

10^3

binary number system

(0,1)

1 digit : 1bit

(i) one bit - 0/1 : 2 possible values can be generated.

(ii) two bit - 0/1 : 2 and 0/1 : 2
2^2=4 possible values can be generated

(iii) eight bit - 2^8 possible values can be generated

unsigned

2bits

0	0
0	1
1	0
1	1

4 values
range : 0 to (4-1)3

4bits

2^4 = 16 values
range : 0 to (16-1)15

range: integer

(i) unsigned range of int

int : 2byte
2^{16} possible values
65,536

range : 0 to 65,535
0 to 2^{16}-1

int : 1byte
2^{8} possible values
256

range : 0 to 255
0 to 2^{8}-1

int : 4byte
2^{32} possible values
4294967296

range : 0 to 4294967296
0 to 2^{32}-1

(ii) signed range of int

int : 2byte
2^{16} possible values
65,536

range : -32,768 to 32,767
-2^{15} to 2^{15}-1

int : 1byte
2^{8} possible values
256

range : -128 to 127
-2^{7} to 2^{7}-1

int : 4byte
2^{32} possible values
4294967296

range : -2147483648 to 2147483647
-2^{31} to 2^{31}-1

for n bits

unsigned	signed
0 to 2^{n}-1	-2^{n-1} to 2^{n}-1

facts about printf

(i) whatever you provide in double quotes printf will print as it is

```
#include<stdio.h>
void main (){
        int a = 20;
        printf ("a");
                }
```

output : a

so, the value is not printing here.

(ii) whatever you provide in double quotes printf will consider it as textual data by format.

```
#include<stdio.h>
void main (){
        printf ("10+10");
                }
```

output : 10+10

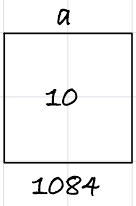
so, the addition is not printing here.

we have to specify the format, for this we need format specifier

format specifier

(i) %d - format specifier for integer

```
#include<stdio.h>
void main (){
        int a = 10;
        printf ("the value
                is %d",a);
                        }
```

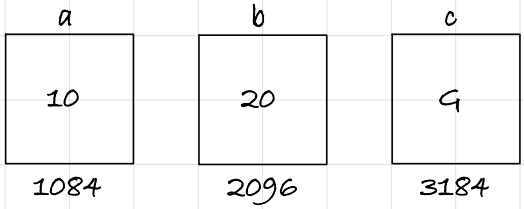


output : the value is 10

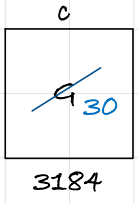
printf ("the value is %d",a); — the value that which needs to be printed is in a. (after comma tells us)

some data in integer form is to be printed.

```
#include<stdio.h>
void main (){
        int a = 10, b = 20, c;
        c = a+b;
        printf ("the sum of %d and %d is %d",a,b,c)

}
```

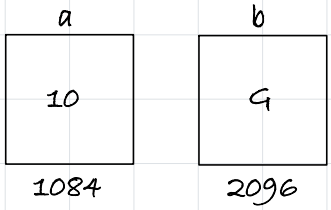


output : the sum of 10 and 20 is 30



initialization

```
#include<stdio.h>
void main (){
        int a = 10, int b ;
        c = a+b;
        printf ("the sum of %d and %d is %d",b,a,c)

}
```

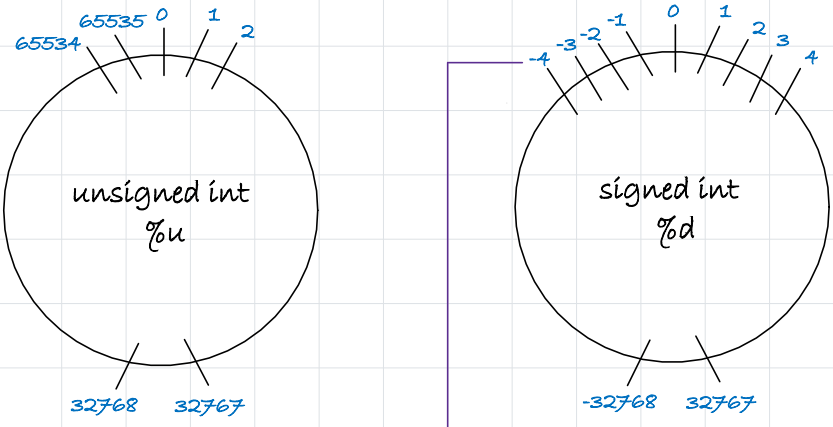


output : the sum of 20 and 10 is 30

(ii) %u - format specifier for unsigned short int and integer
(iii) %ld - format specifier for long integer
(iv) %lld - format specifier for long long integer

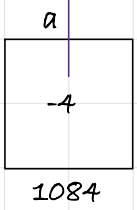
cyclic property

integer %d




```
#include<stdio.h>
void main (){
        int a = -4;
        printf ("%d",a);
                }
```

by default signed



it is available in signed circle

output : -4

```
#include<stdio.h>
void main (){
        unsigned int a = -1;
        printf ("%u",a);
                }
```

it is not available in unsigned circle so it will move anticlockwise and will take the value available at -1 in unsigned circle

output : 65535

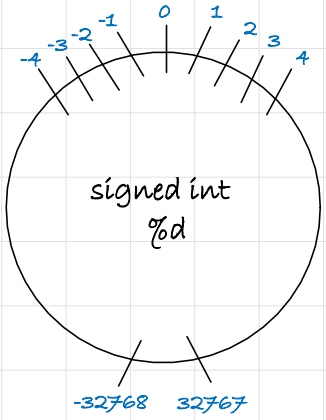
```
#include<stdio.h>
```

i

```
#include<stdio.h>
void main (){
        unsigned int i = -1;
        printf ("%d",i);
                }
```

it is not available in unsigned circle so it will move anticlockwise and will take the value available at -1 in unsigned circle

output : -1



we are printing %d (signed) but the value assigned as unsigned so we have to move twice in the signed circle to reach 65535 and the value available at 65535 in the signed circle is -1

```
#include<stdio.h>
void main (){
        int a = 32768;
        printf ("%d",a);
                }
```

by default signed

output : -32768

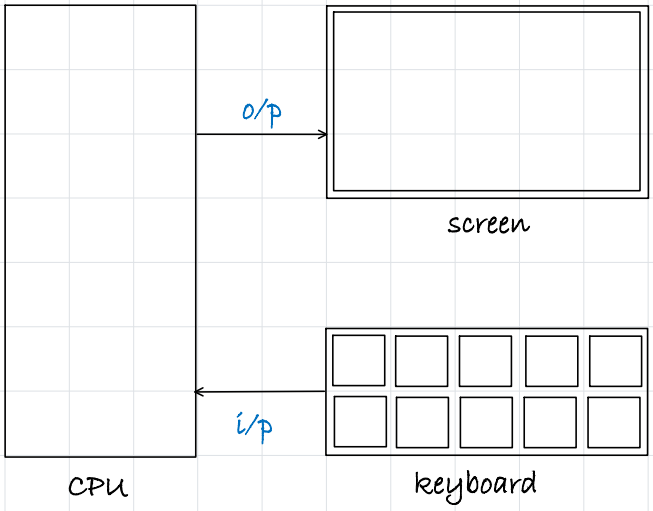
```
#include<stdio.h>
void main (){
        int a = -32770;
        printf ("%d",a);
                }
```

by default signed

output : 32766

topic : scanf

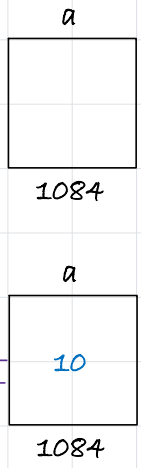
scanf : to read input



(i) scanf ("%d") - the first thing we tell scanf is about format, in which format we are about to give data.

(ii) scanf ("%d",&a); - the second thing we tell scanf is the address, where and in which variable we want to save data.

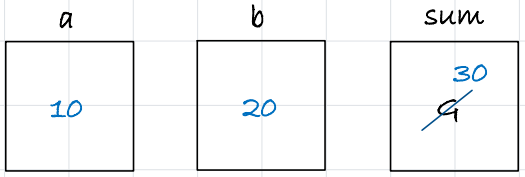
```
#include<stdio.h>
void main (){
        int a;
        printf ("enter a number:" ");
        scanf ("%d",&a);
        printf ("the value is %d",a);
                                }
```



output :
enter a number : 10
the value is 10

as we enter the value 10, 10 will be saved at location a

```
#include<stdio.h>
void main (){
        int a,b,sum;
        printf ("enter 2 numbers:" ");
        scanf ("%d,%d",&a,&b);
        sum = a+b;
        printf ("the sum of %d and %d is %d",a,b,sum);
                                }
```



output :
enter 2 number : 10 20

the sum of 10 and 20 is 30

note : it is not neccesary to write printf before scanf but just to make the program user friendly we use printf

range: char(character)%c

unsigned	signed
0 to 255	-128 to 127

decimal number to binary are possible but how symbols are converted into binary?

program

```
#___
voidmain(){
        int a = 10;
          ---
          ---
          ---
          ---
            }
```

10 : 0000 1010

0100....
0011....
..
..
.
0010....

for that we need a character system

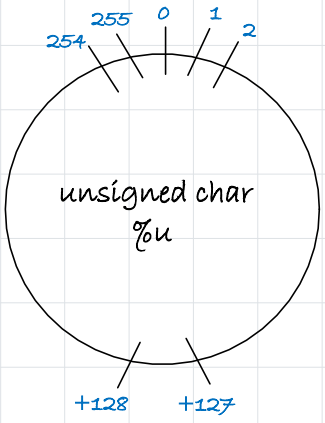
english	assigned number
@ information	001__0000 information

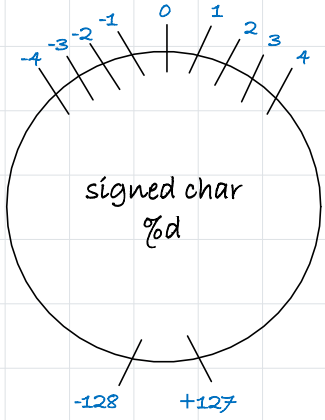
both are information but their representations are different.

for character system there is ASCII code (american standard code for information interchange)

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]






```
#include<stdio.h>
void main (){
        char c = 65;
        printf ("%d"c");
                }
```

c

0100 0001

value stored

but asked for %d, 65 is in %d circle.

output :

65

```
#include<stdio.h>
void main (){
        char c = 128;
        printf ("%d"c");
                }
```

c

-128

value stored

```
        printf ("%d"c");
                    }
```

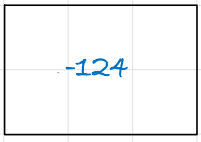
value stored

by default signed — but asked for %d, -128 is in %d circle.

output :
-128

```
#include<stdio.h>
void main (){
        char c = 132;
        printf ("%d"c");
                    }
```

c



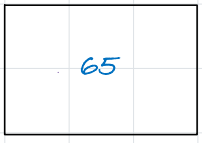
value stored

by default signed — but asked for %d, -124 is in %d circle.

output :
-124

```
#include<stdio.h>
void main (){
        char c = -191;
        printf ("%c"c");
                    }
```

c



256-191=65

value stored

by default signed — asked for %C (unsigned), 65 is A is in %c system.

output : A

topic : escape sequence

(i) \n : it moves the cursor from its current position to the beginning of the next line

```
#include<stdio.h>
void main (){
        printf ("hello everyone");
        printf ("how are you");
                    }
```

output : hello everyone how are you

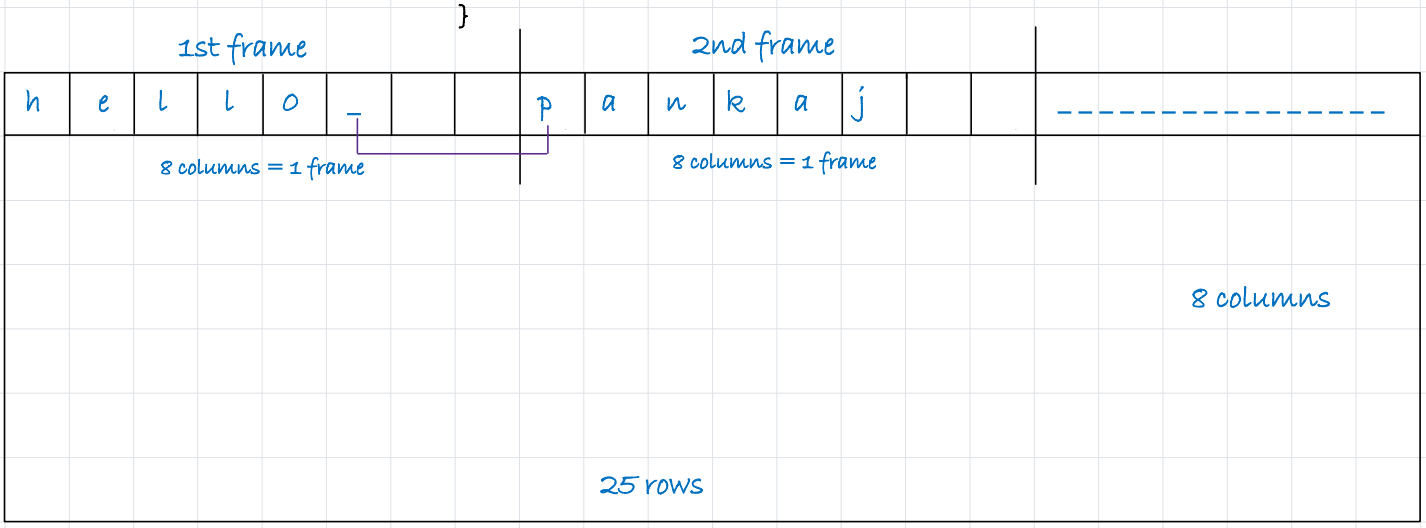
```
#include<stdio.h>
void main (){
        printf ("hello everyone\n");
        printf ("how are you");
                    }
```

output : hello everyone

how are you

(ii) \t: it moves the cursor to next available frame.

```
#include<stdio.h>
void main (){
        printf ("hello/t");
        printf ("pankaj");
                }
```



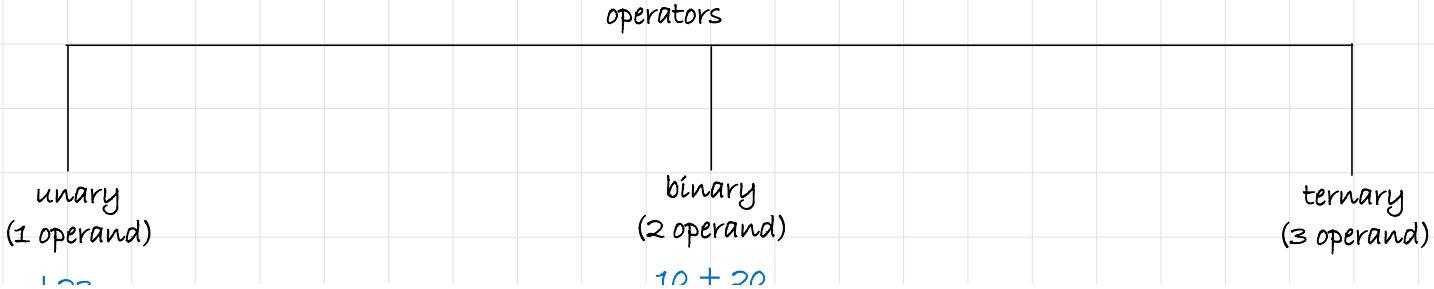
3 spaces are there.

topic : operators

10 + 20=30
10 x 20=200

10, 20 (operand)
+ , X (operators)

there is a value associated with every operations



unary
(1 operand)

+23
-17

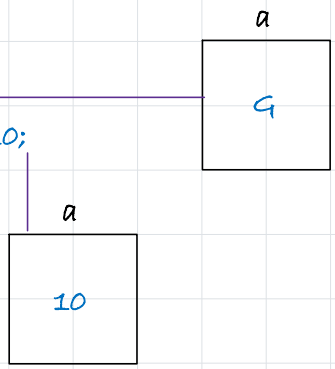
binary
(2 operand)

10 + 20
10 x 20

ternary
(3 operand)

C program : collection of statements

(i) declaration : int a;
(ii) declaration + initialization: int a = 10;
(iii) assignment statement : a=10;



(iv) expression: statement having some value

5+10=15, 15 is a value hence it is a expression
int a; it is not a expression because it does not have any value, instead it is a statement.

(i) every expression is a statement.
(ii) every statement is not a expression.

any value is a valid expression and expression is a statement, so a value always represents a C statement.

valid C program

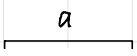
```
#include<stdio.h>
void main (){
                5+2;
                10.2;
                    }
```

(i) assignment operator (=)

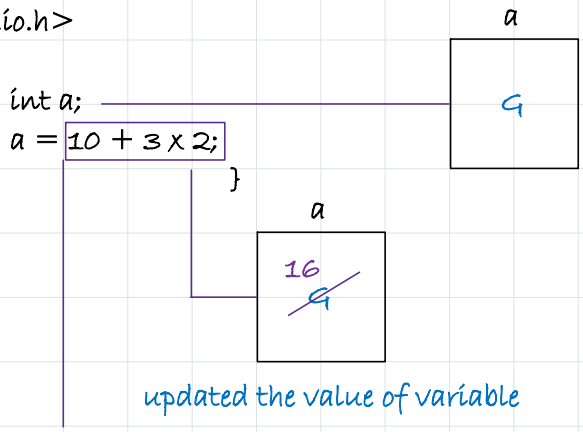
LHS = RHS
it is a binary operator (=) R to L

it updates the value of a variable.

```
#include<stdio.h>
```



```
#include<stdio.h>
void main (){
        int a;
        a = 10 + 3 x 2;
        }
```



updated the value of variable

expression

valid statements:

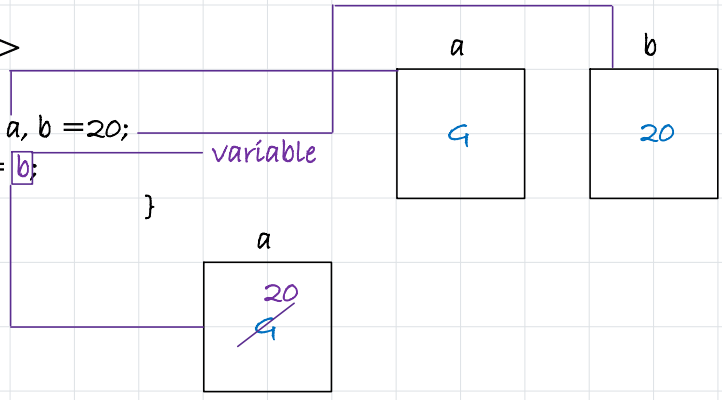
```
#include<stdio.h>
void main (){
        int a;
        a = 10 + 3 x 2;
        }
```

expression

```
#include<stdio.h>
void main (){
        int a;
        a = 30;
        }
```

literal/constant

```
#include<stdio.h>
void main (){
        int a, b =20;
        a = b;
        }
```



Lvalue = R value

R value can be
- expression
- literal/constant

- variable

L value cannot be
- expression
- literal/constant
- must be a variable

(ii) arithmatic operator (+ ,- ,x ,/ ,%)

priority / precedence

- high (X, /, %) L to R
- low (+, -) L to R

priority means how to parenthesize and not how to evaluate

example :
- 8/2+3

(8/2)+3 as the priority of / is more than +

what if both operators are of same priority?
- 8/4 x 2

so, when 2 operators are of same priority (or) multiple occurence of same operator we use associativity

associativity
(i) left to right .
(ii) right to left.

(iii) modulus operator (%)

it is a binary operator

a%b : what is the remainder when a is divided by b?

```
#include<stdio.h>
void main (){
        printf ("%d",12%5);
                }
```

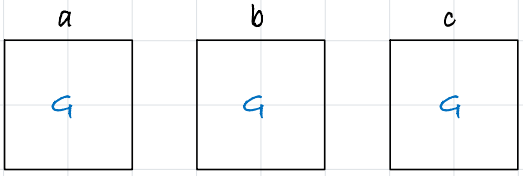
output : 2

note : both operator for % must be integer type otherwise compiler will give errors.

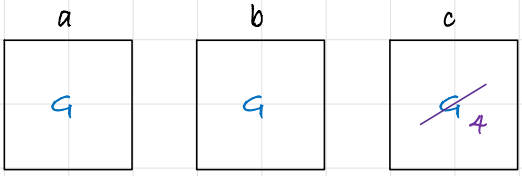
```
#include<stdio.h>
void main (){
        printf ("%d",2%6);
                }
```

output : 2

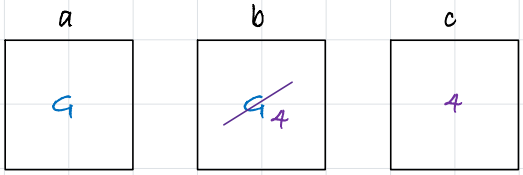
-
```
#include<stdio.h>
 void main (){
        int a,b,c;
        a = b = c = 4;
        printf ("%d %d %d", a, b, c);
                }
```



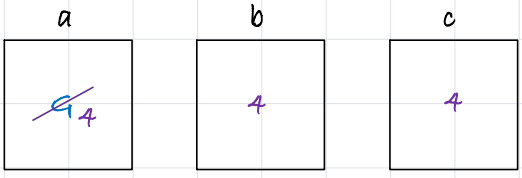
(i) a=b=c=4
a=b=(c=4)

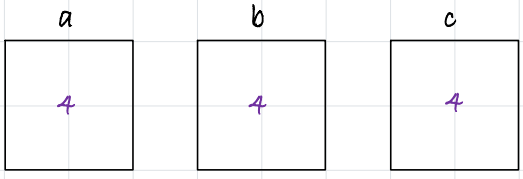


(ii) a=b=4
a=(b=4)



(iii) a=4
a=4





-
```
#include<stdio.h>
 void main (){
        int a,b,c;
        a = b = 4 = c;
        printf ("%d %d %d", a, b, c);
                }
```

L value cannot be constant/literal

result of any operator depends upon operand

(i) 4/2 = 2
4 : int type
2 : int type
result : int type

(ii) 4.0/2 = 2.0
4.0 : floating type (double)
2 : int type
result : floating type (double)

(iv) relational operators

they all are binary operators
- a<b : is a less than b?
- a>b : is a greater than b?
- a<=b : is a less than (or) equal to b?
- a=>b : is a greater than (or) equal to b?
- a==b : is a equal to b?
- a!=b : is a not equal to b?

you can categorise them in true or false.

priority / precedence

- high (<, <=, >,=>) L to R
- low (==, !=) L to R

```
- #include<stdio.h>
  void main (){
                printf ("%d", 10<20);
                                    }
```

output : 1

the statement is true hence the output is 1

```
- #include<stdio.h>
  void main (){
```

```
                printf ("%d", 0>0);
                                }
```

output : 0

the statement is false hence the output is 0

```
- #include<stdio.h>
  void main (){
                printf ("%d", 10<=10);
                                }
```

output : 1

there are two statements
(i) 10<10 : false
(ii) 10=10 : true
hence, the statement is true.

```
- #include<stdio.h>
  void main (){
                printf ("%d", 20<=10);
                                }
```

output : 0

there are two statements
(i) 20<10 : false
(ii) 20=10 : false
hence, the statement is false.

```
- #include<stdio.h>
  void main (){
                printf ("%d", 10==10);
                                }
```

output : 1

(i) 10==10 : true
hence, the statement is true.

```
- #include<stdio.h>
  void main (){
                printf ("%d", 20==10);
```

```
                                                    }
```

output : 0

(i) 20==10 : true
hence, the statement is false.

```
- #include<stdio.h>
  void main (){
                    printf ("%d", 10!=10);
                                                    }
```

output : 0

(i) 10!=10 : false
hence, the statement is false.

```
- #include<stdio.h>
  void main (){
                    printf ("%d", 20!=10);
                                                    }
```

output : 1

(i) 20!=10 : true
hence, the statement is true.

the value (or)output of every relational operator is either 0 or 1

note : unary operator have always higher priority than binary operators

priority	operators	associativity
high ↓ low	+, - (unary)	
	x , / , % (arithmatic)	l to R
	+ , - (binary)	
	< , <= , > , => (relational)	
	== , !=	
	= (assignment)	R to l

```
- #include<stdio.h>
    void main (){
                    int a;
                    a = 10<5==3!=4<7>2;
                    printf ("%d",a);

                                                }
```

a=10<5==3!=4<7>2
a=(10<5)==3!=4<7>2

a=0==3!=(4<7)>2
a=0==3!=1>2

a=0==3!=(1>2)
a=0==3!=0

a=(0==3!)=0
a=0!=0

a=(0!=0)
a=0

output : 0

```
- #include<stdio.h>
    void main (){
                    int i;
```

```
                    i = printf("pankaj")
                    printf ("%d",i);
                                                }
```

output : pankaj6

```
int i;
i = printf("pankaj")
printf ("%d",i);
```

(i) first printf will print pankaj
(ii) now pankaj have 6 symbols and 6 will be assigned to i

```
- #include<stdio.h>
    void main (){
                    printf ("%d",printf("GATE2025"));
                                                        }
```

output : GATE20258

```
- #include<stdio.h>
    void main (){
                    int a;
                    a=  printf ("%d",printf("%d", printf("GATE2025")));
                                                                        }
```

output : GATE202581

(i)printf ("%d",printf("%d",printf("GATE2025")))
- GATE2025 (8 symbols printed)

(ii)printf ("%d",printf("%d, 8))
- 8 (1 symbol printed)

(iii)printf("%d",1)
- 1

```
- #include<stdio.h>
    void main (){
                    int a;
                    a=  printf ("%d",printf("\n%d", printf("GATE20242025")));
                    printf ("%d",a);
                                                                        }
```

output : GATE20242025
1231

(i)printf ("%d",printf("%d",printf("GATE20242025")))
- GATE20242025 (12 symbols printed)

(ii)printf ("%d",printf("\n%d, 12))
- 12 (2 symbol printed + 1 symbol of \n)

(iii)printf("%d",3)
- 1

(v) logical operators

(a) logical AND (&&) | binary
(b) logical OR (||) | binary
(c) logical not (!) | unary

(a) logical AND (&&)

the output value is 1 if both operands are non-zero (true) otherwise the output value is 0.

a	b	a&&b	output value
F	F	F	0
F	F	F	0
T	F	F	0
T	T	T	1

```
- #include<stdio.h>
    void main (){
                printf ("%",12 && 13.8);
                                        }
```

output : 1

```
- #include<stdio.h>
    void main (){
                printf ("%",A && 3.2);
                                        }
```

output : 1

A : 65 (ASCII)

```
- #include<stdio.h>
    void main (){
                printf ("%",0 && 3);
                                    }
```

output : 0

```
- #include<stdio.h>
    void main (){
                printf ("%",13 && 0.0);
                                    }
```

output : 0

(b) logical OR (||)

the output value is 1 if at-least one operand is non-zero (true) otherwise the output value is 0.

a	b	a||b	output value
F	F	F	0
F	T	T	1
T	F	T	1
T	T	T	1

```
- #include<stdio.h>
    void main (){
                printf ("%",12 || 7.0);
                                    }
```

output : 1

```
- #include<stdio.h>
    void main (){
                printf ("%",-12 || 13.7);
                                    }
```

output : 1

```
- #include<stdio.h>
    void main (){
                printf ("%",12.2 || 0);
                                        }
```

output : 1

```
- #include<stdio.h>
    void main (){
                printf ("%",0 || 0.0);
                                        }
```

output : 0

note : just like relational operators, the value (output) of every logical operator is also 0 or 1.

(c) logical NOT (!)

It is also called a negation operator, which means it takes the input operand and negates it

!5 = !(non-zero) = !true = false = 0
!5 = 0

!0 = !(zero) = !false = true = 1
!0 = 1

short circuit evaluation in AND operator

```
- #include<stdio.h>
    void main (){
                int a;
                a = 0 &&___;

  }
```

this will never be evaluated

if the 1st operand is zero for logical AND, then the result is 0 irrespective of the value of second operator.

```
- #include<stdio.h>
    void main (){
```

```
int a;
a = 0 && printf("hello");
printf("%d",a);
                    }
```

output : 0

will never be evaluated

short circuit evaluation in OR operator

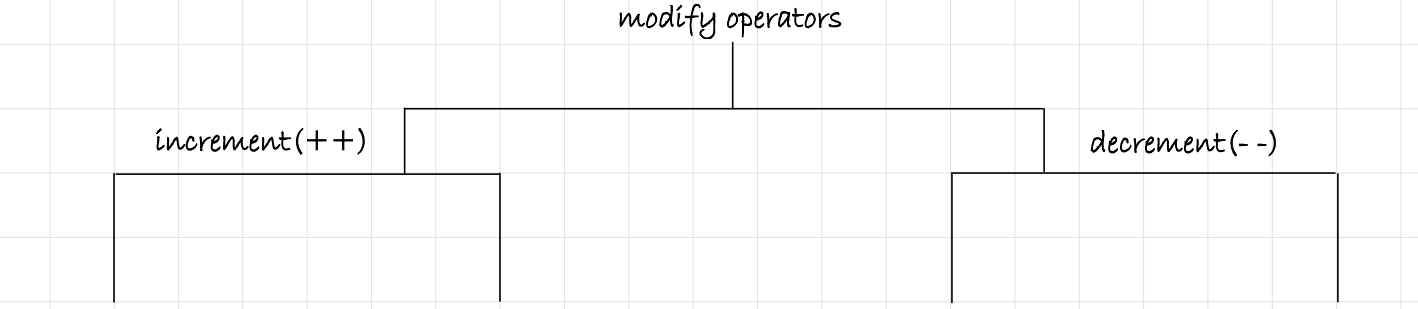
```
- #include<stdio.h>
    void main (){
                int a;
                a = 5 || ____;
                        }
```

this will never be evaluated

output : 1

if the 1st operand is non-zero for logical OR, then the second operand will not be evaluated.

(vi) modify operators



pre- increment

```
#include<stdio.h>
void main (){
        int a=5, b;
        b = ++a;
        printf ("%d", a);
        printf ("%d", b);
                }
```

b = ++a ;
(i) first increase the value of variable
- a = a+1 (5 → 6)

post- increment

```
#include<stdio.h>
void main (){
        int a=5, b;
        b = ++a;
        printf ("%d", a);
        printf ("%d", b);
                }
```

b = a++ ;
(i) first use the value of variable
- b = a (5)

pre- decrement

```
#include<stdio.h>
void main (){
        int a=5, b;
        b = - - a;
        printf ("%d", a);
        printf ("%d", b);
                }
```

b = - - a ;
(i) first decrease the value of variable
- a = a - 1 (5 → 4)

post- decrement

```
#include<stdio.h>
void main (){
        int a=5, b;
        b = a - -;
        printf ("%d", a);
        printf ("%d", b);
                }
```

b = a - - ;
(i) first use the value of variable
- b = a (5)

(i) first increase the value of variable
- a = a+1

(ii) used the updated value. (in expression/assignment)
- b = a

output : 6 6

(i) first use the value of variable
- b = a

(ii) then updated value. (in expression/assignment)
- a = a+1

output : 6 5

(i) first decrease the value of variable
- a = a - 1

(ii) used the updated value. (in expression/assignment)
- b = a

output : 4 4

(i) first use the value of variable
- b = a

(ii) then updated value. (in expression/assignment)
- a = a - 1

output : 5 4

note : if you using modify operator more than once in a sequence point then it is compiler dependent.

```
- #include<stdio.h>
  void main (){
                int a=0, b=2, c;
                c = a++ && - - b;
                printf ("%d", a, b, c);
                                }
```

c = a++ && - - b;

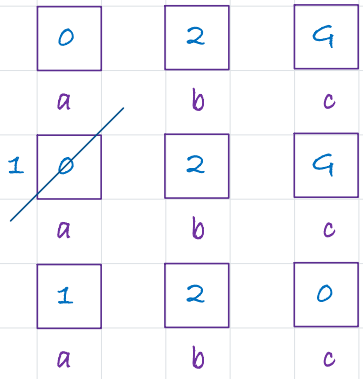
(i) a++
- use the value.
- then increase it.

c = 0 && - - b

short circuit evaluation

c = 0

output : 1 2 0




```
- #include<stdio.h>
  void main (){
                a = pf("pankaj")||pf("rawan")&&pf("hai")
                printf ("%d", a);
                                }
```

a=pf("pankaj")||pf("rawan")&&pf("hai")

a=~~pf("pankaj")~~||pf("rawan")&&pf("hai") pankaj

6

a=6||~~pf("rawan")&&pf("hai")~~ short circuit evaluation

a=1

output : pankaj1

priority	operators	associativity
high ↓	+, - (unary)	
	x , / , % (arithmatic)	
	+ , - (binary)	l to R
	< , <= , > , => (relational) == , !=	
	&& (logical) ||	
low	= (assignment)	R to l

binary number

0 to 1 | 0 | = 2x + 0 = 2x
x
(2 times of old value + 0)
if we add 0

0 to 1 | 1 | = 2x + 1 = 2x
x
(2 times of old value + 1)
if we add 1

(v) bitwise operators : perfomed on bits

(a) bitwise AND (&)
(b) bitwise OR(|)
(c) bitwise XOR (^)
(d) bitwise left shift (<<)
(e) bitwise right shift (>>)
(f) bitwise not (~)

(a) to (e): (binary)
(f): (unary)

(a) bitwise AND (&)

the output value is 1 if both operands are non-zero (true) otherwise the output value is 0.

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

- #include<stdio.h>
  void main (){
  int a =7, b = 10, c;
  c = a&b
  printf ("%d",c);
  }

7 : 0000 0000 0000 0111
10 : 0000 0000 0000 1010
o/p: 0000 0000 0000 0010

output : 2

(b) bitwise OR(|)

the output value is 1 if one operand is non-zero (true) otherwise the output value is 0.

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

- #include<stdio.h>
  void main (){
  int a =7, b = 10, c;
  c = a|b
  printf ("%d",c);
  }

7 : 0000 0000 0000 0111
10 : 0000 0000 0000 1010
o/p: 0000 0000 0000 1111

output : 15

(c) bitwise XOR (^)

the output value is 0 if both operands are same otherwise the output value is 0.

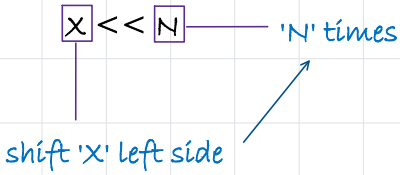
a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

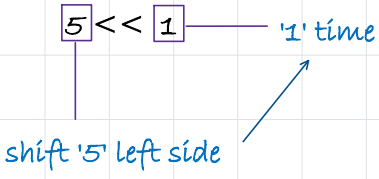
-
```
#include<stdio.h>
    void main (){
                int a =7, b = 13, c;
                c = a^b
                printf ("%d",c);
                                    }
```

7 : 0000 0000 0000 0111
13 : 0000 0000 0000 1101
o/p: 0000 0000 0000 1010

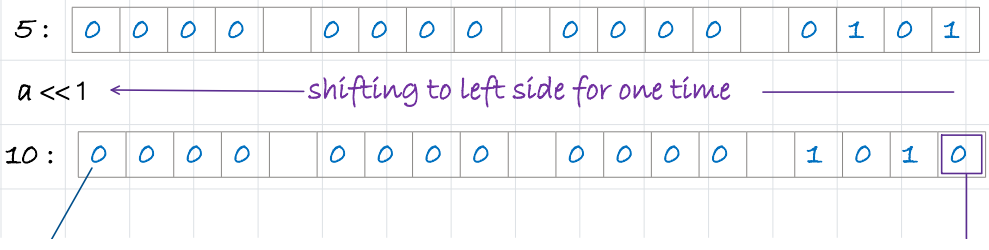
output : 10

(d) bitwise left shift (<<) (binary operator)

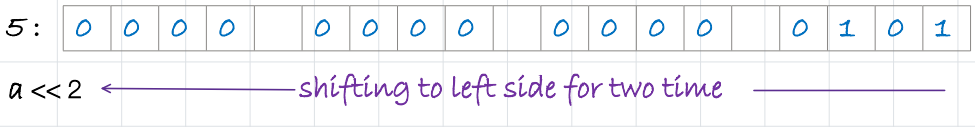




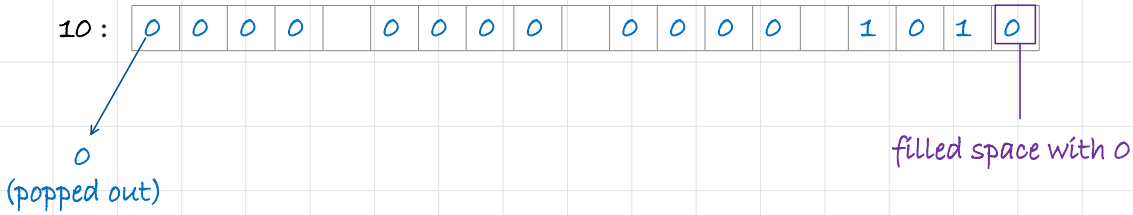
-
```
#include<stdio.h>
    void main (){
                int a, b;
                a = 5;
                b = a <<1
                printf ("%d%d",a,b);
                                    }
```



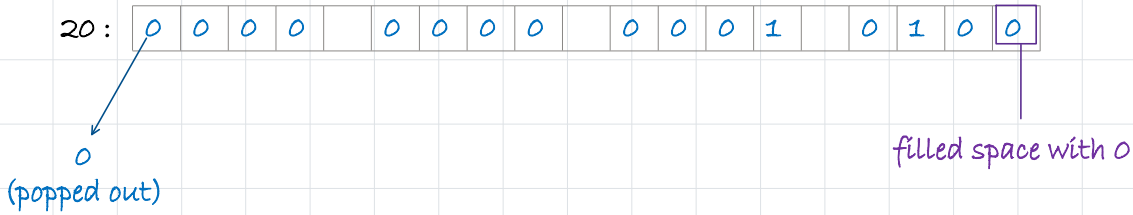

```
- #include<stdio.h>
    void main (){
                int a, b;
                a = 5;
                b = a << 2;
                printf ("%d%d",a,b);
                                }
```



first time :



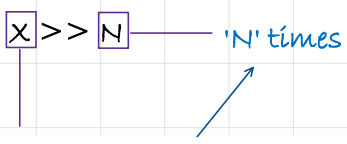
second time :



in general :

a<<2 is a x 2 x 2
$a \times 2^2$
5×2^2
$5 \times 4 = 20$

(e) bitwise right shift (>>)

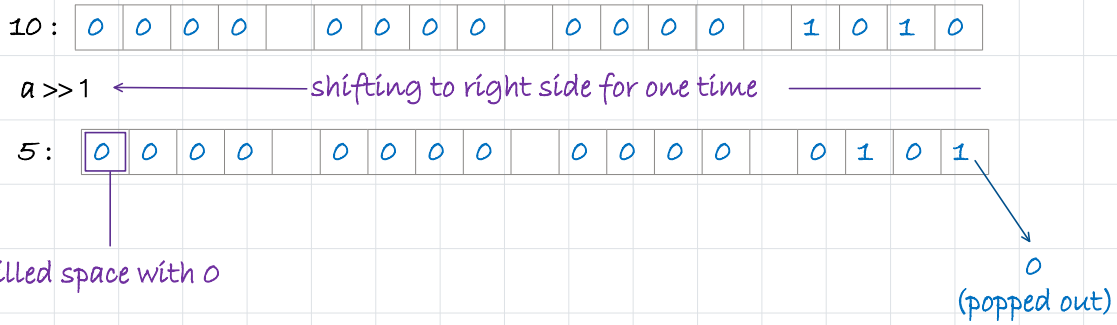


shift 'X' right side

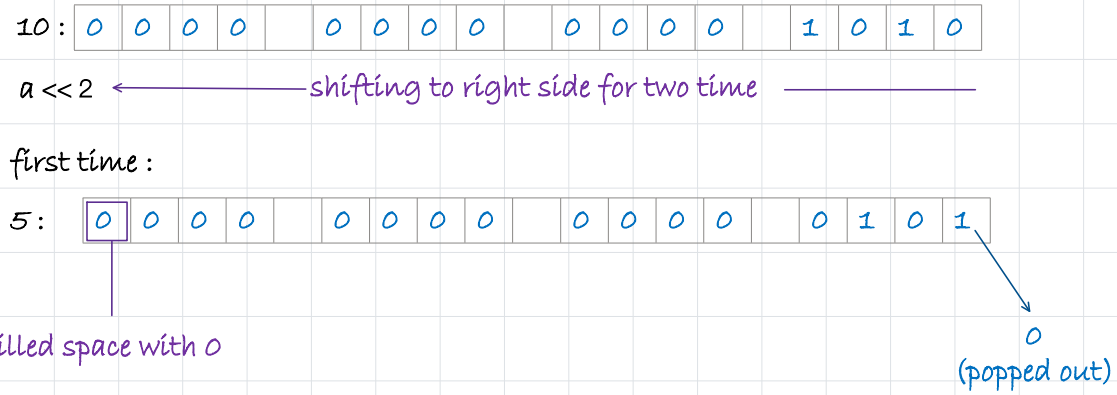
5 >> 1 — '1' time

shift '5' right side

```
- #include<stdio.h>
    void main (){
                int a, b;
                a = 10;
                b = a >>1
                printf ("%d%d",a,b);
                                }
```




```
- #include<stdio.h>
    void main (){
                int a, b;
                a = 10;
                b = a >> 2;
                printf ("%d%d",a,b);
                                }
```



second time :

2 : 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 1 0

filled space with 0

1

(popped out)

in general :

a>>2 is $a/2^2$

10/4 = 2

note :
%d : integer
%o : octal
%x : hexadecimal

literal

decimal

```
int a = 31;
```

prefix (compiler)

```
int a = 027
```

written in octal

```
- #include<stdio.h>
    void main (){
                int a = 027
                printf ("%d,a);
                            }
```

output : 23

prefix (compiler)

```
int a = 0xace
```

written in hexadecimal

```
- #include<stdio.h>
    void main (){
```

a = 10
b = 11

```
- #include<stdio.h>
    void main (){
                int a = 0xace
                printf ("%d,a);
                            }
```

```
a = 10
b = 11
c = 12
d = 13
e = 14
```

output : 2766

- in 2's complement we focus on 0's

1 1 1 1 0 1 1 0

$= -2^3 - 2^0 - 1$

$= -8 -1 -1$

$= -10$

(f) bitwise not (~)

```
- #include<stdio.h>
    void main (){
                int a=48, b;
                b = ~a;
                printf ("%d,b);
                            }
```

output : -49

48 : | 0 0 0 0 | 0 0 0 0 | 0 0 1 1 | 0 0 0 0 |

~a : | 1 1 1 1 | 1 1 1 1 | 1 1 0 0 | 1 1 1 1 |

written in 2's complement

$= -2^5 -2^4 -1$
$= -32 -16 -1$ (or) $= -2^5 -2^4 -1$
$= -49$ $= -(2^5+2^4+1)$
$= -(32+16+1)$
$= -(48+1)$
$= -(a+1)$

```
- #include<stdio.h>
    void main (){
                int a=-10;
                printf ("%d,~a);
                            }
```

output : +9

= -(a+1)
= -(-10+1)
= -(-9)
= +9

questions :

```
- #include<stdio.h>
    void main (){
                int a=028;  ———— (octal 0 to 7)
                printf ("%d,a);
                                }
```

output : compiler error

```
- #include<stdio.h>
    void main (){
                int a, b, c;
                a = -1 > 1;  ———— integer have both negative and positive
                b = -1L > 1;  ———— integer have both negative and positive
                c = -1u > 1;  ———— unsigned integer have only positive, so -1 will be a large positive number
                printf ("%d%d%d,a,b,c);
                                        }
```

output : 0 0 1

```
- #include<stdio.h>
    void main (){
                int a;
                a = printf("pankaj")>>1;
                printf ("%d,a);
                                        }
```

output : pankaj3

a = printf("pankaj")>>1;
a = 6>>1;
$6/2^1 = 3$
a = 3

```
- #include<stdio.h>
    void main (){
```

```
int i=-1, j=-1, k=0, l=2, m;
m = i++ && j++ && k++ || l++;
printf ("%d %d %d %d,i, j, k, l, m);
                                    }
```

output : 0 0 1 3 1

m = i++ && j++ && k++ || l++;

i	j	k	l	m
-1	-1	0	2	G

m = (((-1 && -1) && 0) || 2);

i	j	k	l	m
0	0	1	3	G

m = ((1 && 0) || 2);

m = (0 || 2);

m = 1

i	j	k	l	m
0	0	1	3	1

-
```
#include<stdio.h>
    void main (){
                int x=10, y=5, d, q;
                p = x>9;
                q = x>3 && y!=3;
                printf ("%d %d ,p , q);
                                    }
```

output : 1 1

- p = x>9;
  p = 10>9;
  p = 1

- q = x>3 && y!=3;
  q = 10>3 && 5!=3;
  q = 1 && 1;
  q = 1

(g) ternary operator (?:)

format : exp1 ? exp2 : exp3

evaluated first and if its value is non-zero (true) then the value (o/p) is exp2

otherwise the value (o/p) is exp3

example :

```
- #include<stdio.h>
    void main (){
                int a;
                a = 10 ? 20 : 30;
                printf ("%d, a);
                                }
```

output : 20

a = 10 ? 20 : 30;

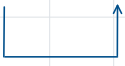
a = 1 ? 20 : 30; (true)

```
- #include<stdio.h>
    void main (){
                int a;
                a = 5<2 ? 20 : 30;
                printf ("%d, a);
                                }
```

output : 30

a = 5<2 ? 20 : 30;

a = 0 ? 20 : 30;



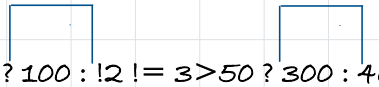
```
- #include<stdio.h>
    void main (){
                int a;
                a = 20>100 ? 100 : !2 != 3>50 ? 300 : 400;
                printf ("%d, a);
                                                        }
```

output : 400

nested

(i) jitne colons (:) utne question (?)

(ii) right to left, colon apne nearest free question mark ke sath attach hoga.



a = 20>100 ? 100 : !2 != 3>50 ? 300 : 400;

a = 20>100 ? 100 : !2 != 3>50 ? 300 : 400;
exp1

a = 20>100 ? 100 : !2 != 3>50 ? 300 : 400;
exp2

a = 20>100 ? 100 : !2 != 3>50 ? 300 : 400;
exp3

a = 20>100 ? 100 : !2 != 3>50 ? 300 : 400;
a = 0 ? 100 : !2 != 3>50 ? 300 : 400;

a = !2 != 3>50 ? 300 : 400;

a = !2 != 3>50 ? 300 : 400;
exp1

a = !2 != 3>50 ? 300 : 400;
exp2

a = !2 != 3>50 ? 300 : 400;
exp3

a = !2 != 3>50 ? 300 : 400;

a = (!2 != (3>50)) ? 300 : 400;

a = (0 != 0) ? 300 : 400;

a = 0? 300 : 400;

a = 400

(h) sizeof

- unary
- works during compile time
- memory in byte
- returns in unsigned int because size cannot be negative

- #include<stdio.h>
  void main (){
  int a=2, b=4;
  printf ("%d, size of (3)); integer type(integer).
  }

output : 4

```
- #include<stdio.h>
    void main (){
                int a=2, b=4;
                printf ("%d, size of (10/2+3));
                                                }
```

(int/int + int) (expression)

output : 4

```
- #include<stdio.h>
    void main (){
                int a=2, b=4;
                printf ("%d, size of (a));
                                            }
```

integer type (variable)

output : 4

```
- #include<stdio.h>
    void main (){
                int a=2, b=4;
                printf ("%d, size of (int));
                                            }
```

integer type (data type)

output : 4

size of can be
- data type
- literal
- variable
- expression

flow control statement

statements that control the flow of execution of statements.

(i) selection statements
- if
- else if
- if-else if else
- switch statement

(ii) iterative statements (repititon)
loops :
- for loop
- while loop
- do while loop

(iii) jump statement
- continue
- break
- exit
- return

(i) selection statements

(a) if

the if in C is the decision-making statement. It consists of the test condition in if block or body. If the given condition is true only then the if block will be executed.

```
if(condition/expression)
{
|    // if body
|    // Statements to execute if condition is true
}————————— scope of 'if'
```

if there are no brackets after (condition/expression) then by default the scope is till first semi colon.

with bracets -
```
if (condition/expression) {
                    s3;
                    s4;
                      }
```

without bracets -
```
if (condition/expression)  ——→  if (condition/expression) {
                                                    s3; }
                                                    s4;
```

```
s3;
s4;
}
```

```
s3;
s4;
```

```
s3; }
s4;
```

```
- #include<stdio.h>
  void main (){
      s1;
      s2;
      if (condition/expression){
      s3;
      s4;
      }
      s5;
      s6;
      }
```

Statements to execute if condition is true

true : S1, S2, S3, S4, S5, S6

false : S1, S2, S5, S6

```
- #include<stdio.h>
  void main (){
      s1;
      s2;
      if (true){
      s3;
      s4;
      }
      s5;
      s6;
      }
```

```
- #include<stdio.h>
  void main (){
      s1;
      s2;
      if (false){
      s3;
      s4;
      }
      s5;
      s6;
      }
```

```
- #include<stdio.h>
  void main (){
      pf ("1");
      if (2<3){
      pf ("2");
      pf ("3");
      }
      pf ("4");
      }
```

true

output : 1 2 3 4

```
- #include<stdio.h>
    void main (){
                    pf ("hey");
                    if (!20)
                    pf ("bhagwan");
                    pf ("bacha lo");
                    pf ("is rawan se");
                                    }
```

no brackets

output : heybachaloisrawanse

```
pf ("hey");
if (!20){
pf ("bhagwan");}
pf ("bacha lo");
pf ("is rawan se");
```

zero (false)

```
 - #include<stdio.h>
     void main (){
                    if (){
                    pf ("bhagwan");}
```

syntax error

```
 - #include<stdio.h>
     void main (){
                    int i=4;
                    if (i<2);
                    pf ("pankaj");}
```

no brackets but semi colon is instanlty after if

```
if (i<2){
        ;
        }
```

output : pankaj

write a program to read a number and if the number is even then print "pankaj"

```
 - #include<stdio.h>
     void main (){
                    int a;
                    printf ("enter a number");
                    scanf ("%d,&a");
                    if (a%2 ==0){
                    printf ("pankaj");}
                                    }
```

to store a number (int) we took a variable "a"

to take the input from keyboard

to store a number in the location "a"

if a is divisible by 2 that gives remainder 0 (even no.)

last bit of even : 0
last bit of odd : 1

another way :

```
- #include<stdio.h>
    void main (){
                int a;                          ——— to store a number (int) we took a variable "a"
                printf ("enter a number");      ——— to take the input from keyboard
                scanf ("%d,&a");                ——— to store a number in the location "a"
                if (a&1==0){                    ——— if a anding 1 that is equal to 0 (even no.)
                printf ("pankaj");}
                                }
```

a=3

```
- #include<stdio.h>
    void main (){
                int a;
                printf ("enter a number");
                scanf ("%d,&a");
                if (3&1==0){
                printf ("pankaj");}
                                }
```

3 : 0000 0000 0000 0011
1 : 0000 0000 0000 0001
    0000 0000 0000 0001

```
if (3&1==0){
            printf ("pankaj");}

if (1==0){
            printf ("pankaj");}

if (0){
            printf ("pankaj");}
```

output : nothing

a=2

```
- #include<stdio.h>
    void main (){
                int a;
                printf ("enter a number");
                scanf ("%d,&a");
                if (2&1==0){
                printf ("pankaj");}
                                }
```

2 : 0000 0000 0000 0010
1 : 0000 0000 0000 0001
    0000 0000 0000 0000

```
if (2&1==0){
            printf ("pankaj");}

if (0==0){
            printf ("pankaj");}

if (1){
            printf ("pankaj");}
```

output : pankaj

(b) if-else

It is an extension of the 'if' in C that includes an 'else' block along with the already existing if block.

```
if (condition/expression) {
    // code executed when the condition is true
}
else {
    // code executed when the condition is false
}
```

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if (condition/expression){
                s3;
                s4;
                }
                else {
                s5;
                s6;
                }
                s7;
                s8;
                }
```

Statements to execute if condition is true

Statements to execute if condition is false

true : S1, S2, S3, S4, S7, S8

false : S1, S2, S5, S6, S7, S8

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if(true){
                s3;
                s4;
                }
                else {
                s5;
                s6;
                }
                s7;
                s8;
                }
```

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if (false){
                s3;
                s4;
                }
                else {
                s5;
                s6;
                }
                s7;
                s8;
                }
```

(i) either code 1 (or) code 2 will execute.

(ii) both will not execute.
(iii) exactly one of them will execute.
(iv) you cannot use else without if.

- #include<stdio.h>
  void main (){

```
if (!2+3)          if (!2+3){
printf ("1");      printf ("1")}
printf ("2");      printf ("2");  ———— error
else               else
printf ("3");      printf ("3");
          }                  }
```

else cannot be without if, else should be after the scope of if ends.
there cannot be statements between if and else.

write a program to take a integer and if the number is even then print "1", otherwise print "0".

- #include<stdio.h>
  void main (){

```
int a;  ———————————————— to store a number (int) we took a variable "a"
printf ("enter a number");  ———— to take the input from keyboard
scanf ("%d,&a");  ———————————— to store a number in the location "a"
if (a&1==0){  ———————————————— if a anding 1 that is equal to 0 (even no.)
printf ("1");}
else {
printf ("0");}
          }
```

(c) if-else-if

one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed.

if none of the conditions is true, then the final else statement will be executed.

```
if (condition/expression)
    statement;
else if (condition/expression)
    statement;
.
.
else
    statement;
```

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if (condition/expression){
                s3;                    ——— Statements to execute if condition is true
                s4;
                  }
                else if (condition/expression){
                s5;                    ——— Statements to execute if condition is true
                s6;
                  }
                else {
                s7;                    ——— Statements to execute if condition is false
                s8;}
                  }
```

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if(true){
                s3;
                s4;
                  }
                else if (con/exp) {
                s5;
                s6;
                  }
                else {
                s7;
                s8;}
                  }
```

output : s1, s2, s3, s4

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if(false){
                s3;
                s4;
                  }
                else if (true) {
                s5;
                s6;
                  }
                else {
                s7;
                s8;}
                  }
```

output : s1, s2, s5, s6

```
- #include<stdio.h>
    void main (){
                s1;
                s2;
                if(false){
                s3;
                s4;
                  }
                else if (false) {
                s5;
```

```
    }
else if (false) {
s5;
s6;
    }
else {
s7;
s8;}
    }
```

output : s1, s2, s7, s8

largest among two numbers

```
- #include<stdio.h>
    void main (){
                int a,b,max;
                printf ("enter a number")
                scanf ("%d %d, &a, &b")
                max = a>b?a:b
                printf ("%d", max);
                                }
```

```
- #include<stdio.h>
    void main (){
                int a,b,max;
                printf ("enter a number")
                scanf ("%d %d, &a, &b")
                if (a>b)
                    max = a;
                else
                    max = b;
                        }
```

largest among three distinct number

```
- #include<stdio.h>
    void main (){
                int a,b,c;
                printf ("enter a number")
                scanf ("%d %d %d, &a, &b, &c")
                if (a>b && a>c){
                    printf("%d is the largest, a);
                else if (b>c){
                    printf("%d is the largest, b);
                else {
                    printf("%d is the largest, c);
```

```
- #include<stdio.h>
    void main (){
                int a,b,c, largest;
                printf ("enter a number")
                scanf ("%d %d %d, &a, &b, &c")
                largest=(a>b && a>c)?a:b>c?b:c
```

a is greater than b and a is greater than c then the answer is a

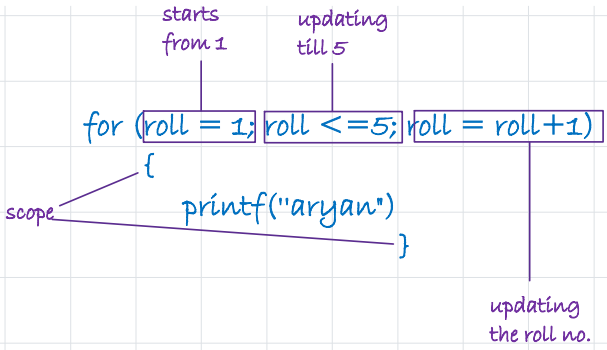
b is greater than c then the answer is b otherwise c

(ii) iterative statements (repititon)

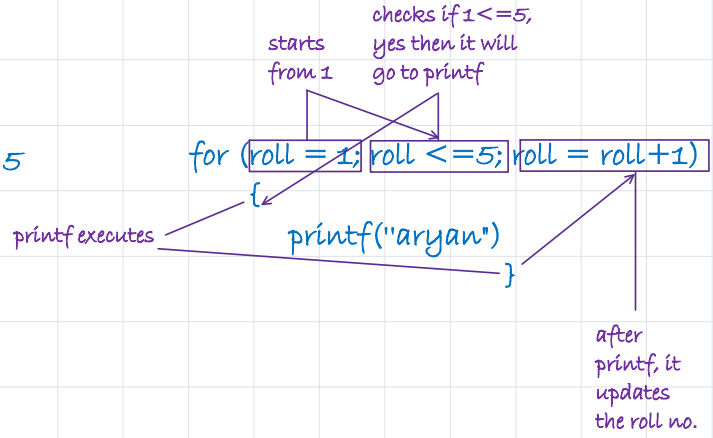
(a) for loop

starts from 1

updating till 5

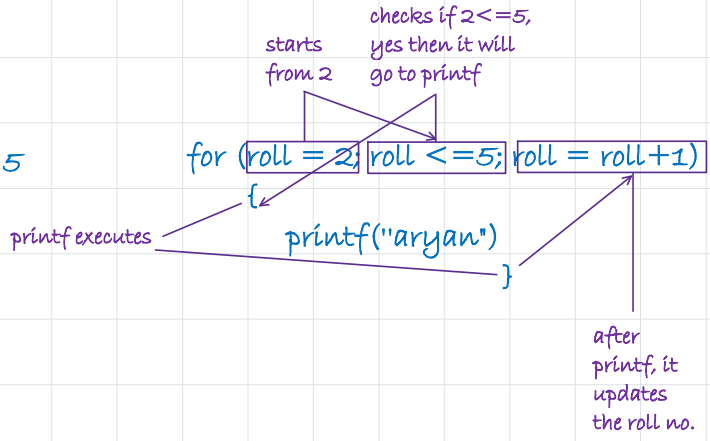


for roll = 1; roll<=5



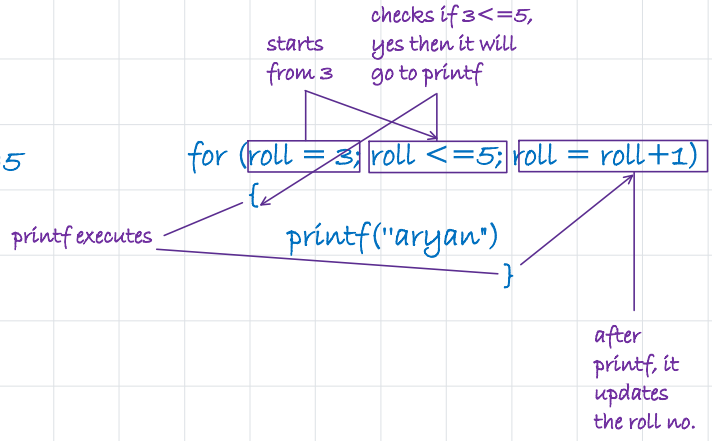
output : aryan

for roll = 2; roll<=5



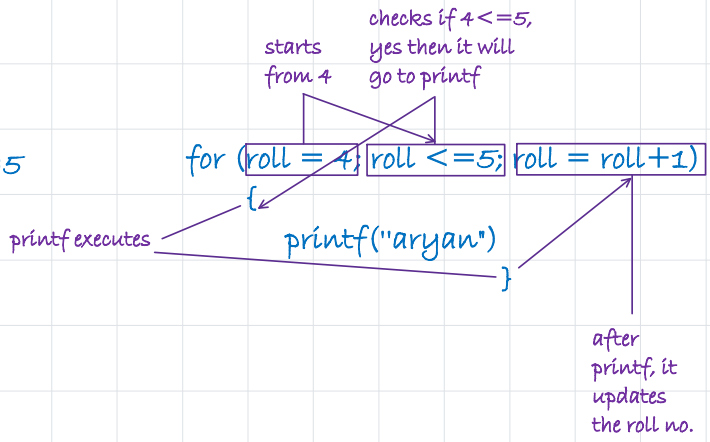
output : aryan

for roll = 3; roll<=5



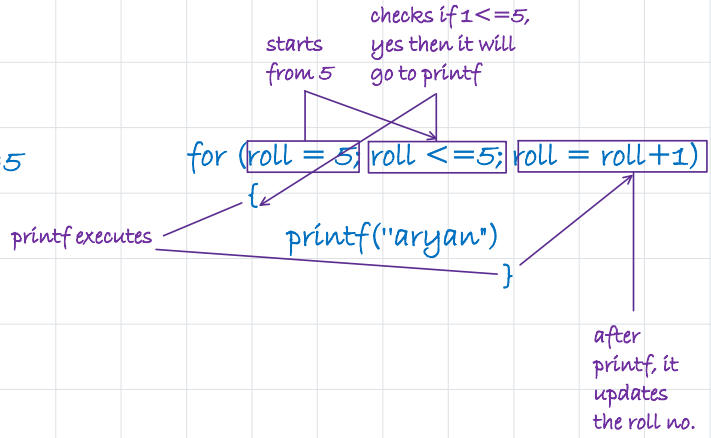
output : aryan

for roll = 4; roll<=5



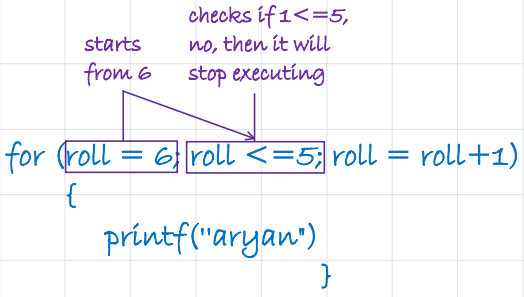
output : aryan

for roll = 5; roll<=5



output : aryan

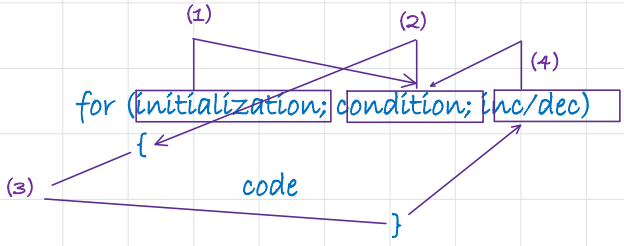
for roll = 6; roll<=5



output : aryan

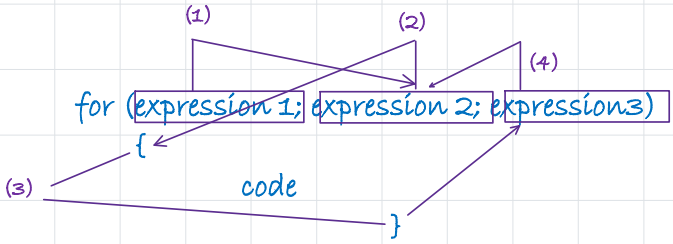
final output : aryan aryan aryan aryan aryan

format :



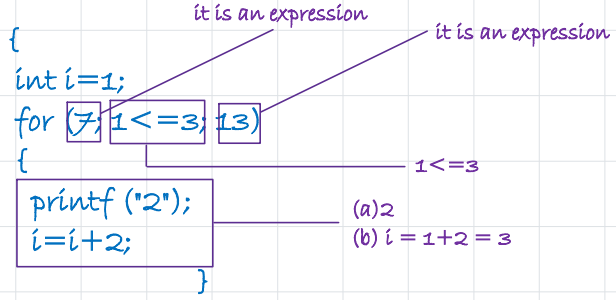
(2), (3), (4) repeats

loop ke ek baar chalne ko iteration kehte hai

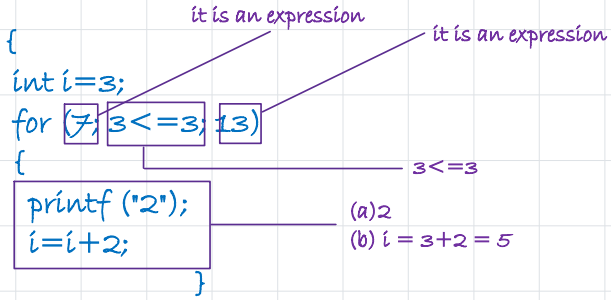


iteration :
exp 1 : 1 time
exp 2 : true = code executes then goes to exp3
updates the value
exp 2 : true = code executes then goes to exp3
updates the value
.
.
repition : exp (2) - code - exp (3)

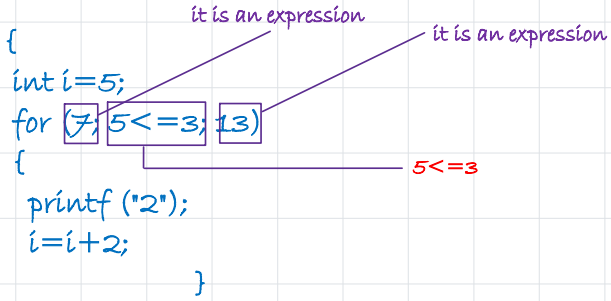
```
- #include<stdio.h>
    void main (){
                    int i=1;
                    for (7; i<=3; 13)
                    {
                     printf ("2");
                     i=i+2;
                                }
```



output : 2



output : 2



final output : 22

```
- #include<stdio.h>
    void main (){
                char ch;
                for (ch=1; ch; ch++)
                    printf ("aryan");
                    printf("end");
                                }
```

```
                {
                char ch;
                for (ch=1; ch; ch++)
                    printf ("aryan");
                    printf("end");
                                }
```

ch = 1,2,3.....127,-128,-129..-1,0(false) - (255)

scope till first semi-colon

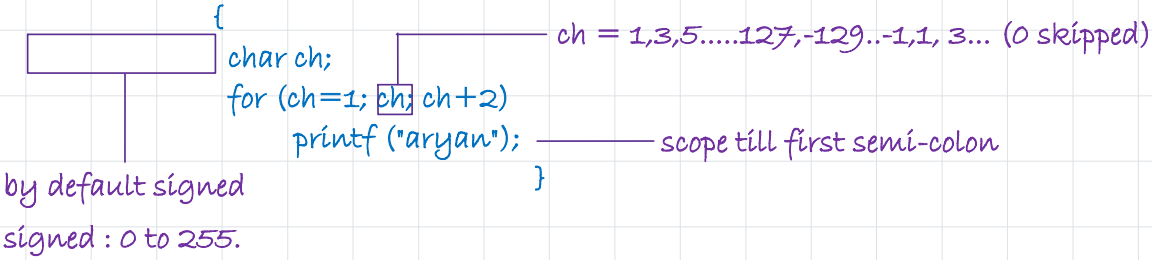
by default signed

signed : 0 to 255.

output : aryan(255times)end

```
- #include<stdio.h>
    void main (){
                char ch;
                for (ch=1; ch; ch+2)
                    printf ("aryan");
```

```
}
```



output : aryan (infinite times in odd)

```
- #include<stdio.h>
    void main (){
                int i =1;
                for (printf("1"); i<=5; printf("3"))
                {
                 printf ("2");
                 i++
                            }
                                                    }

               {
               int i =1;
               for (printf("1"); 1<=5; printf("3"))
               {
                printf ("2");
                i++
                            }
                                                    }
```

output : 123

```
               {
               int i =2;
               for (printf("1"); 2<=5; printf("3"))
               {
                printf ("2");
                i++
                            }
                                                    }
```

output : 23

```
{
int i =3;
for (printf("1"); 3<=5; printf("3"))
{
 printf ("2");
 i++
        }
                                    }
```

output : 23

```
{
int i =4;
for (printf("1"); 4<=5; printf("3"))
{
 printf ("2");
 i++
        }
                                    }
```

output : 23

```
{
int i =5;
for (printf("1"); 5<=5; printf("3"))
{
 printf ("2");
 i++
        }
                                    }
```

output : 23

```
{
int i =6;
for (printf("1"); 6<=5; printf("3"))
{
 printf ("2");
 i++
        }
                                    }
```

output : 1

final output : 12323232323

note : all 3 expressions are optional

```
- #include<stdio.h>
    void main (){
                int i =-1;
                for (i++; i++; i++)
                {
                 printf ("pankaj");
                        }
                                }
```

```
{
int i =-1;
for (-1; i++; i++)
{
 printf ("pankaj");
        }
```

i++ :
(a) use the value
-1
(b) update the value
0

output : nothing

```
 - #include<stdio.h>
     void main (){
                int i =-1;
                for (i++; i++; i++)
                {
                 printf ("pankaj");
                        }
                                }
```

```
{
int i =-1;
for (-1; ++i; i++)
{
 printf ("pankaj");
        }
```

i++ :
(a) use the value
-1
(b) update the value
0

output : pankaj (infinite times-odd)

i : 1, 3, 5...0 (skipped)

++i :
(a) update the value
1
(b) use the value
1

i++ :
(a) use the value
1
(b) update the value
2

```
- #include<stdio.h>
    void main (){
                    char i =-1;
                    for (i++; i++; i++)
                    {
                     printf ("pankaj");
                                    }
                                                    }
```

```
{
 char i =-1;
 for (-1; ++i; i++)
 {
  printf ("pankaj");
                  }
```

output : pankaj (infinite times-odd)

i : 1, 3, 5...0 (skipped)

i++ :
(a) use the value
-1
(b) update the value
0

++i :
(a) update the value
1
(b) use the value
1

i++ :
(a) use the value
1
(b) update the value
2

```
- #include<stdio.h>
    void main (){
                    for (i=1; i<=n; i+2)
                    {
                     printf ("pankaj");
                                    }
                                      }

                   {
                   for (i=1; i<=n; i+2)
                   {
                    printf ("pankaj");
                                    }
                                      }
```

output : $\frac{n}{2}$

```
- #include<stdio.h>
    void main (){
                for (i=1; i<=n; i x 2)
                {
                 printf ("pankaj");
                                    }
                                      }
```

```
- #include<stdio.h>
    void main (){
                for (i=1; i<=100; ix2)
                {
                 printf ("pankaj");
                                    }
                                      }
```

i = 1 : pankaj (2^0)
i = 2 : pankaj (2^1)
i = 4 : pankaj (2^2)
i = 8 : pankaj (2^3)
i = 16 : pankaj (2^4)
i = 32 : pankaj (2^5)
i = 64 : pankaj (2^6)
.
.
i = n(2^k)
last value of i : 2^k

i = 1 : pankaj
i = 2 : pankaj
i = 4 : pankaj
i = 8 : pankaj
i = 16 : pankaj
i = 32 : pankaj
i = 64 : pankaj
i = 128 X

printf will print k+1 times

what is k?

$2^k <= n$
$\log(2^k) <= \log n$
$k \log 2 <= \log n$

$$k <= \frac{\log n}{\log_2}$$

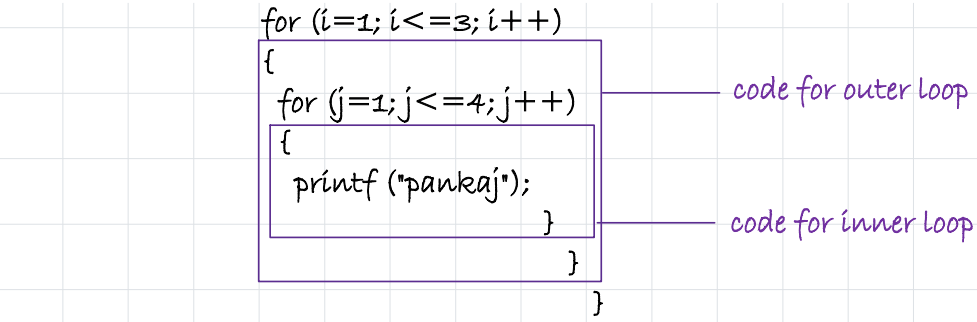
$k <= \log_2 n$

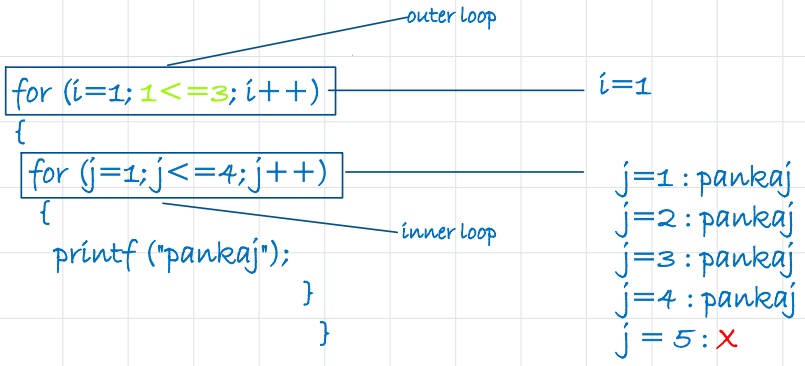
$k = \lfloor \log_2 n \rfloor$
$1 + \lfloor \log_2 n \rfloor$

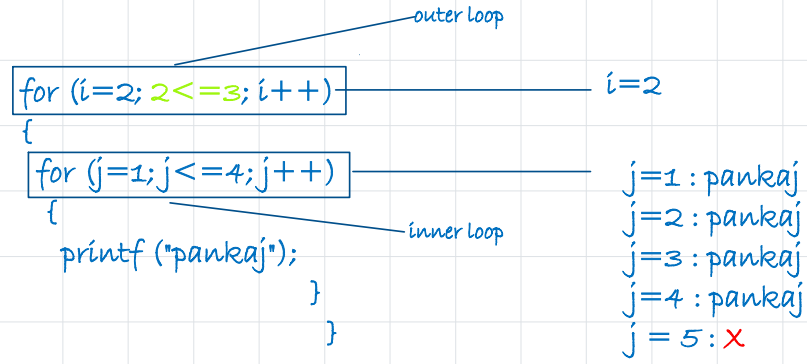
nested loop :

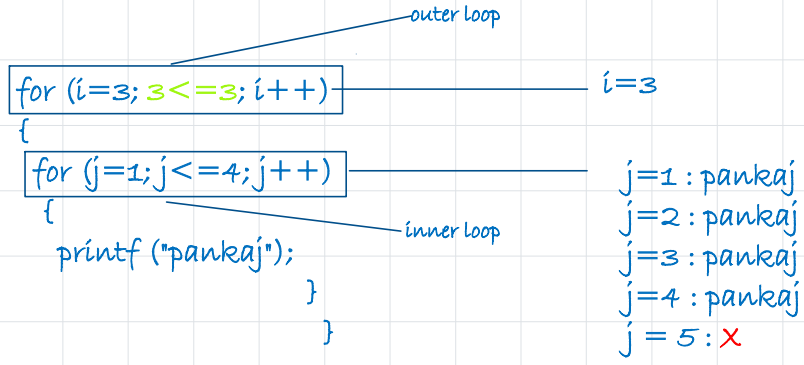
```
- #include<stdio.h>
    void main (){
                for (i=1; i<=3; i++)
                {
                 for (i=1; i<=4; i++)
```

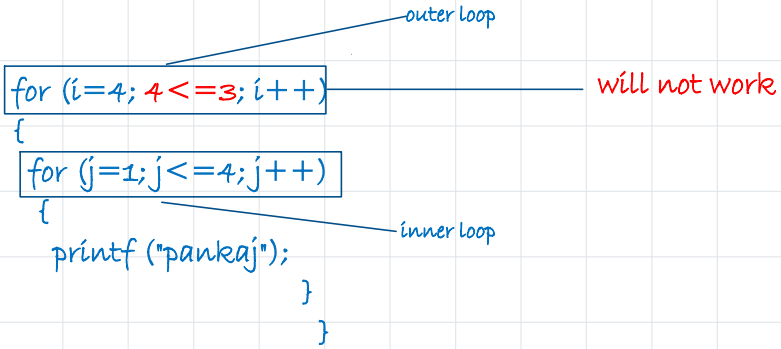
code for outer loop


for (i=1; i<=3; i++)
{
for (j=1;j<=4;j++)
{
printf ("pankaj");
}
}
}
code for outer loop
code for inner loop


outer loop
for (i=1; 1<=3; i++)
{
for (j=1;j<=4;j++)
{
printf ("pankaj");
}
}
inner loop
i=1
j=1 : pankaj
j=2 : pankaj
j=3 : pankaj
j=4 : pankaj
j = 5 : X


outer loop
for (i=2; 2<=3; i++)
{
for (j=1;j<=4;j++)
{
printf ("pankaj");
}
}
inner loop
i=2
j=1 : pankaj
j=2 : pankaj
j=3 : pankaj
j=4 : pankaj
j = 5 : X


outer loop
for (i=3; 3<=3; i++)
{
for (j=1;j<=4;j++)
{
printf ("pankaj");
}
}
inner loop
i=3
j=1 : pankaj
j=2 : pankaj
j=3 : pankaj
j=4 : pankaj
j = 5 : X


outer loop
for (i=4; 4<=3; i++)
{
for (j=1;j<=4;j++)
{
printf ("pankaj");
}
}
inner loop
will not work

output : pankaj pankaj pankaj pankaj
pankaj pankaj pankaj pankaj
pankaj pankaj pankaj pankaj
pankaj pankaj pankaj pankaj

```
- #include<stdio.h>
    void main (){
                for (i=1; i<=3; i++)
                {
                 for (j=1;j<=4;j++)
                 {
                  printf ("%d%d%d,i,j");
                                        }
                   printf("\n");
                                         }
                                          }
```

```
for (i=1; 1<=3; i++)
 {
  for (j=1;j<=4;j++)
   {
    printf ("%d%d%d,i,j");
                        }
     printf("\n");
                  }
```

outer loop

inner loop

i=1

j=1
j=2
j=3
j=4
j = 5 : X

output : 11121314

```
for (i=2; 2<=3; i++)
 {
  for (j=1;j<=4;j++)
   {
    printf ("%d%d%d,i,j");
                        }
     printf("\n");
                  }
```

outer loop

inner loop

i=2

j=1
j=2
j=3
j=4
j = 5 : X

output : 21222324

```
for (i=3; 3<=3; i++)
{
 for (j=1;j<=4;j++)
```

outer loop

i=3

j=1

i = 1	1<=3	j=1 1<=4 printf ("%d%d%d,i,j"); 11
		j=2 2<=4 printf ("%d%d%d,i,j"); 12
		j=3 3<=4 printf ("%d%d%d,i,j"); 13
		j=4 4<=4 printf ("%d%d%d,i,j"); 14
.	.	.
.	.	.
.	.	.
i = 3	3<=3	j=1 1<=4 printf ("%d%d%d,i,j"); 31
		j=2 2<=4 printf ("%d%d%d,i,j"); 32
		j=3 3<=4 printf ("%d%d%d,i,j");

```
{
 for (j=1;j<=4;j++)
  {
   printf ("%d%d%d,i,j");
                        }
    printf("\n");
                  }
```

inner loop

j=1
j=2
j=3
j=4
j = 5 : X

32
j=3
3<=4
printf ("%d%d%d,i,j");
33
j=4
4<=4
printf ("%d%d%d,i,j");
34

output : 31323334

final output : 11121314
21222324
31323334

```
- #include<stdio.h>
    void main (){
                  for (i=1; i<=3; i++)
                  {
                   for (j=1;j<=n;j++)
                   {
                    printf ("pankaj");
                                    }
                                        }
                                          }
```

i = 1
n times

i = 2
n times

i = 3
n times

final output : 3n times

```
- #include<stdio.h>
    void main (){
                  for (i=1; i<=n; i++)
                  {
                   for (j=1;j<=3;j++)
                   {
                    printf ("pankaj");
                                    }
                                        }
```

```
                                                    }
```

```
i = n       i = n       i = n
 -j = 1      -j = 1      -j = 1
 -j = 2      -j = 2      -j = 2
 -j = 3      -j = 3      -j = 3
```

final output : 3n times

```
- #include<stdio.h>
    void main (){
                    for (i=1; i<=n; i++)
                    {
                     for (j=1;j<=n;j++)
                     {
                      printf ("pankaj");
                                        }
                                            }
                                               }
```

```
i = n       i = n                  i = n
 -j = n      -j = n.....  -j = n
```

final output : $(n \times n)\ n^2$ times

```
- #include<stdio.h>
    void main (){
                   for (i=1; i<=n; i++)
                   {
                    for (j=1;j<=n;j=j x 2)
                    {
                     printf ("pankaj");
                                       }
                                           }
                                              }
```

$i = n$ $\quad -j = 1 + \lfloor \log_2 n \rfloor$ $\quad i = n$ $\quad -j = 1 + \lfloor \log_2 n \rfloor$ $\ldots$ $i = n$ $\quad -j = 1 + \lfloor \log_2 n \rfloor$

final output : $n(1 + \lfloor \log_2 n \rfloor)$

```
- #include<stdio.h>
    void main (){
                for (j=128;j>=1;j=j/2)
                {
                 printf ("pankaj");
                                }
                                    }
```

output : $\log_2 128$
7+1 = 8times

j= 128
j= 64
j= 32
j= 16
j= 8
j= 4
j= 2
j= 1

```
- #include<stdio.h>   (dependent loop)
    void main (){
                for (i=1; i<=3; i++)
                {
                 for (j=1;j<=i;j++)
                 {
                  printf ("pankaj");
                                }
                                    }
```

i = 1	1<=3	j=1 1<=1 pankaj
i = 2	2<=3	j=1 1<=2 pankaj j=2 2<=2 pankaj
i = 3	3<=3	j=1 1<=3 pankaj j=2 2<=3 pankaj j=3 3<=3 pankaj

final output :
pankaj
pankaj pankaj
pankaj pankaj pankaj

(b)while loop

```
- #include<stdio.h>
    void main (){
            while (expression/condition)
            {
             code
```

, true/non-zero(code executes)

```
{
 code
 } true/non-zero(code executes)
 }
```
false/zero(code does not execute)

```
- #include<stdio.h>
   void main (){
              for (i=1; i<=n; i++)
               {
                printf ("pankaj");
                          }
                      }
```

≡

```
- #include<stdio.h>
   void main (){
              i=1
              while (i<=n)
              {
               printf ("pankaj");
               i = i+1
                          }
```

```
- #include<stdio.h>
   void main (){
              int i=1
              while (++i<5)
              {
               printf ("%d,i");
                          }
```

```
int i=1
while (2<5)
{
 printf ("%d,i");
          }
```
output : 2

```
int i=2
while (3<5)
{
 printf ("%d,i");
          }
```
output : 3

```
int i=3
while (4<5)
{
 printf ("%d,i");
          }
```
output : 4

```
int i=4
while (5<5)
{
 printf ("%d,i");
          }
```
output : X

final output : 234

note
- the scope of while loop is also till first semi-colon.
- in for loop and while loop we pre-check the condition.

```
- #include<stdio.h>
   void main (){
              while ( )
              {
                  _
```
it is mandatory to give expression/condition in while loop

```
-
    }
```

do while loop

in do while loop we dont have any pre-conditions, so even if the expression/condition is false, it will execute the code at-least once.

```
- #include<stdio.h>
    void main ()
                do {
                    code
                } while (0);
```

code executes once

even if the condition is false the code will execute at-least once

for loop	while loop	while loop

number of iterations are known in advance
 	
for (i=1; i<=10, i++)
{
-
-
}
 	
	number of iterations are not known in advance but some criteria/conditions is there
	
	while ( )
	{
	-
	-
	}
		post-checking of condition
		
		code will execute at-least one time irrespective of wether the condition is true or false.
		
		do {
		code
		} while (0);

code for sum

```
- #include<stdio.h>
    void main (){
                int n, i, sum;
                printf ("enter a number");
                scanf ("%d", &n);
                sum = 0;
                for (i=1; i<=n; i++)
                    sum = sum + i;
                printf("%d", sum);
```

code for product

```
- #include<stdio.h>
    void main (){
                int n, i, prod;
                printf ("enter a number");
                scanf ("%d", &n);
                prod= 1;
                for (i=1; i<=n; i++)
                    prod= prod x i;
                printf("%d", prod);
```

```
        sum = sum + i;
    printf("%d", sum);
                }
```

```
        prod= prod x  i;
    printf("%d", prod);
                }
```

break and continue

(a) break

whenever break is encountered in a loop, it terminates the current loop.

```
- #include<stdio.h>
    void main (){
                for (i=1; i<=10; i++)
                {
                 printf("hello");
                 break;
                      }
                printf("end");
                          }
```

```
{
for (i=1; i<=10; i++)  ——— for i=1
    {
     printf("hello"); ——————— hello executes for i=1
     break; ———————————— break terminates the loop
          }
     printf("END");————————————————
                }
```

output : helloEND

```
- #include<stdio.h>
    void main (){
                for (i=1; i<=10; i++)
                {
                 if (i%4==0)
                 break;
                 printf("%d",i);
                              }
```

```
{
for (i=1; i<=10; i++)
    {
     if (i%4==0)
```

```
        {
         break;
                }
     printf("%d",i);
                  }
```

output : 123

(b) continue

whenever continue is encountered it skip the remaining portion of current iteration and continue with next iteration.

```
- #include<stdio.h>
    void main (){
                  for (i=1; i<=10; i++)
                  {
                   continue;
                   printf("%d",i);
                                }
```

```
{
 for (i=1; i<=10; i++)
      {
       continue;  <---------- this skip to the next iteration so printf
        printf("%d",i);       will never execute.
                    }
```

```
- #include<stdio.h>
    void main (){
                  for (i=1; i<=10; i++)
                  {
                   if (i%4==0)
                   continue;
                   printf("%d",i);
                                }
```

```
{
for (i=1; i<=10; i++)
    {
     if (i%4==0)
        {
         continue;
               }
```

```
        printf("%d",i);
                    }
```

output : 123567910

```
- #include<stdio.h>
    void main (){
                for (i=1; i<=10; i++)
                {
                 for ((i+j)%3==0)
                  {
                    continue;
                    printf("%d%d",i,j);
                                    }
                    printf("\n");
                                }
```

```
{
 for (i=1; i<=10; i++)
    {
     for (j=1; i<=4; j++)
        {
         for ((i+j)%3==0)
            {
            continue;
                    }
            printf("%d%d",i,j);
                                }
            printf("\n");
                        }
```

final output :
111314
2223
313234
414244

i=1
j=1
(1+1)%3==0
2%3==0
output : 11

i=1
j=2
(1+2)%3==0
3%3==0
output : X

i=1
j=3
(1+3)%3==0
4%3==0
output : 13

i=1
j=4
(1+4)%3==0
5%3==0
output : 14

i=2
j=1
(2+1)%3==0
3%3==0
output : X

i=2
j=2
(2+2)%3==0
4%3==0
output : 22

i=2
j=3
(2+3)%3==0
5%3==0
output : 23

i=2
j=4
(2+4)%3==0
6%3==0
output : X

i=3
j=1
(3+1)%3==0
4%3==0
output : 31

i=3
j=2
(3+2)%3==0
5%3==0
output : 32

i=3
j=3
(3+3)%3==0
6%3==0
output : X

i=3
j=4
(3+4)%3==0
7%3==0
output : 34

i=4
j=1
(4+1)%3==0
4%3==0
output : 41

i=4
j=2
(4+2)%3==0
6%3==0
output : 42

i=4
j=3
(4+3)%3==0
7%3==0
output : X

i=4
j=4
(4+4)%3==0
8%3==0
output : 44

topic - switch statement

keyword : used to create selection statement with multiple choices
multiple choices are provided with another keyword (case)

```
switch (n) {

    case 1: code we want to execute if the value of n is 1
            break;

    case 2: code we want to execute if the value of n is 1
            break;

   default: code we want to execute if the cases dosent match
            break;

            }
```

```
switch (2x5+3) {
                    case 1: block of statements
                            break;

                    case 13: block of statements      ———— this block will execute.
                             break;

                    default: block of statements
                             break;
                                }
```

important points about switch statement :
(i) break is optional.
(ii) expression : evaluates to be integer.
(iii) position of default does not matter, it can be anywhere (start, mid, end)
(iv) default is optional.
(v) duplicate case labels are not allowed.
(vi) case labels cannot be variables (a,b,c) and can only contains constant/literals.

- once the case label is matched, all the cases will be of no use and the code will execute sequentially

```
switch (1) {
            case 1: block of statements
                    break;

            case 13: block of statements
                     break;

            case 18: printf("aryan")
```

```
                    break;
```

```
switch (1) {
            case 1:

            case 13:

            case 18: printf("aryan")
                     break;
```

```
switch (1) {
            case :

            case :

            case : printf("aryan")
                   break;
```

sequentially

output : aryan

- whatever code you want to execute must be present in after the colons of the case label and before the break.

```
switch (i+3) {
            case 1: printf("1");
                    printf("one");
                    break;
                    printf ("pankaj");
```

it never gets printed (ignored)

```
            case 13: printf("2");
                     break;
                           }
```

- switch is based on expression/condition so it cannot be empty.

```
switch ( ) {
            case 1: block of statements
                    break;
```

cannot be empty

- if default executes, the rest of case lables will be of no use and code will execute sequentially.

```
int i = 3;
switch (i+3) {
            case 5: printf("5");
                    break;

            default: print ("0");

            case 7: printf("7");
                    break;
```

```
int i = 3;
switch (6) {
```

```
int i = 3;
switch (6) {
```

```
int i = 3;
switch (6) {
        case 5: printf("5");
                break;

        default: print ("0");

        case 7: printf("7");
                break;
```

output : 07

```
int i = 3;
switch (6) {
        case 5: printf("5");
                break;

        default: 0

        case: 7
                break;
```

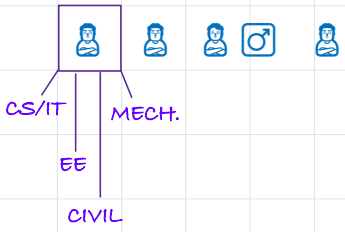
functions and storage classes

functions : functions are designed to perform the specific task.

built in functions : code written by dennis ritchie and we are currenlty using them.
(i) printf
(ii) scanf ()

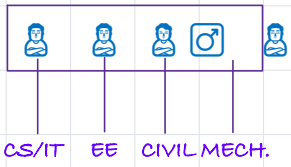
software team

(i) only one person is working



- no proper resource utilisation
- more time

(ii) every person is working



- proper resource utilisation
- less time

functions are designed to perform the specific task.

for example :

```
#include<stdio.h>
void main () {
        int a=10, b=20, ans;
        ans = mul(a,b);
        printf ("%d",ans);
                        }
```

200 ans variable to store the result
ans

there should be variable to hold upcoming values.

```
int mul (int x, int y)
{
 int temp ;
 temp = x * y
 return temp ;
        }
```

mul(a,b) here we are sending the a and b values

20 Y
10 X
200 temp to store the result of multipication
temp

```
#include<stdio.h>
void main () {
        printf ("%d",a);
                        }
```

compiler error because there is no variable a, compiler will not understand without declaration.

note : compilation is done before execution, it is done from top to bottom but execution starts from main.

concept - forward declaration

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

compilation error
error during compile time.

compiler don't know about printf we have to provide the information regarding printf so we have to give the forward declaration to compiler.

to avoid compilation error :

```
#include<stdio.h>          ——— forward declaration for printf
void main () {
        printf ("hello");
                }
```

```
#include<stdio.h>
void main () {
        int a=10, b=20, ans;
        ans = mul(a,b);
        printf ("%d",ans);
                }
```

compiler don't know about this function so have to give compiler the information regarding the function.

function of the header

```
int mul (int x, int y)
{
  int temp ;
  temp =  x * y
  return temp ;
        }
```

body of the function

```
#include<stdio.h>
int mul (int, int)          ——— forward declaration (no memory is allocated here)
void main () {
        int a=10, b=20, ans;
        ans = mul(a,b);
        printf ("%d",ans);
                }
```

arguments

note : if we want to use a function before its definition to avoid compilation error, you provide forward declaration, the forward declaration is for information purpose.

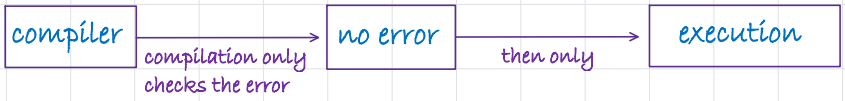
- arguments : jo values function leta hai
- return values : jis type ki value return krta h

but, if we are defining the function before main then the forward declaration is not needed.

```
#include<stdio.h>
int mul (int x, int y)
{
 int temp;
 temp = x * y;
 return temp ;
}
void main () {
        int a=10, b=20, ans;
        ans = mul(a,b);
        printf ("%d",ans);
}
```

we dont need the forward declaration

we have defined the function here



by default the return type of function is int

```
#include<stdio.h>
mul (int, int)
void main () {
        int a=10, b=20, ans;
        ans = mul(a,b);
        printf ("%d",ans);
}
int mul (int x, int y)
{
 int temp ;
 temp =  x * y
 return temp ;
}
```

```
#include<stdio.h>
no forward declaration
void main () {
        int a=10, b=20, ans;
        ans = mul(a,b);
        printf ("%d",ans);
}
```

compiler will store the return type of mul() as int

```
int mul (int x, int y)
{
```

compiler got int so compiler is happy because it matched with information it stored.

```
#include<stdio.h>
no forward declaration
void main () {
        int a=3;
        double b;
        b = f(a);
        printf ("%lf",ans);
}
```

compiler will store the return type of mul() as int

```
double f (int x)
{
```

compiler got double so compiler is not happy and

```
int mul (int x, int y)
{
 int temp ;
 temp = x * y
 return temp ;
              }
```

is happy because it matched with information it stored.

```
double f (int x)
{
 double y = 2.38;
 return ;
              }
```

double so compiler is not happy and will give error because it does not match with information it stored.

so, this confusion and mis-information can be avoided with forward declaration.

q.can we compile a program without main () ?

yes, we can compiler a program without main.

q.can we execute a program without main?

no, we cannot execute without main, the execution starts with main.

(i) if a function return value
store - use.
does not store - no use.

```
#include<stdio.h>
void main () {
              printf ("hello");
                          }
```

print value does not save here so we cannot use the value returned by printf

output : hello

```
#include<stdio.h>
void main () {
              int i;
              i = printf ("hello");
              printf ("%d,i");
                          }
```

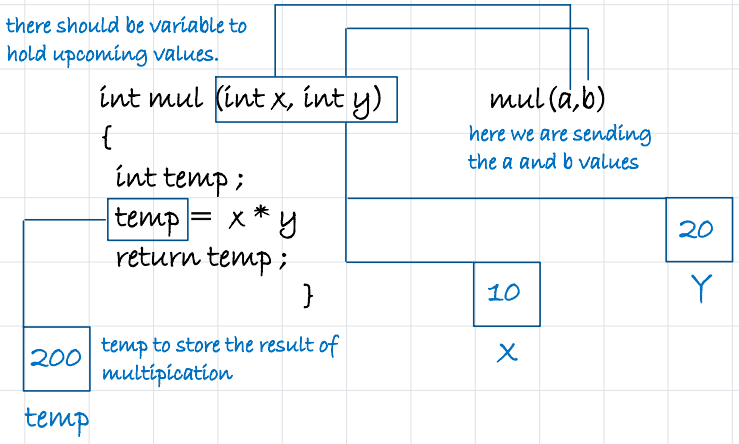
print value does save here in i so we can use the value returned by printf

output : hello5

for example :

```
#include<stdio.h>
void main () {
              int a=10, b=20, ans;
              mul(a,b);
                          }
```

no error but there is no variable to store the value returned.



concept - how function works

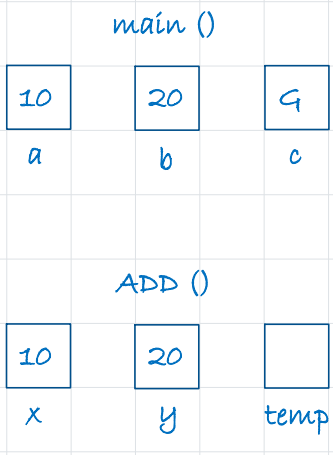
```
#include<stdio.h>
void main () {
        int a=10, b=20, ans;
        ans = ADD (a,b);
        printf ("%d", ans);
                }
```

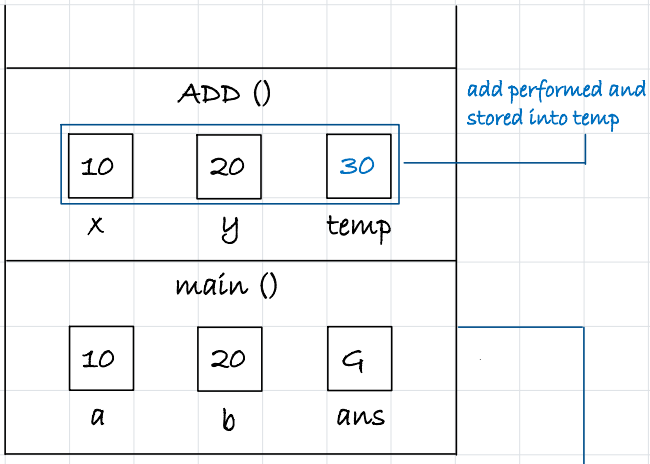
```
int ADD (int x, int y)
{
 int temp ;
 temp = x + y ;
 return temp ;
        }
```

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

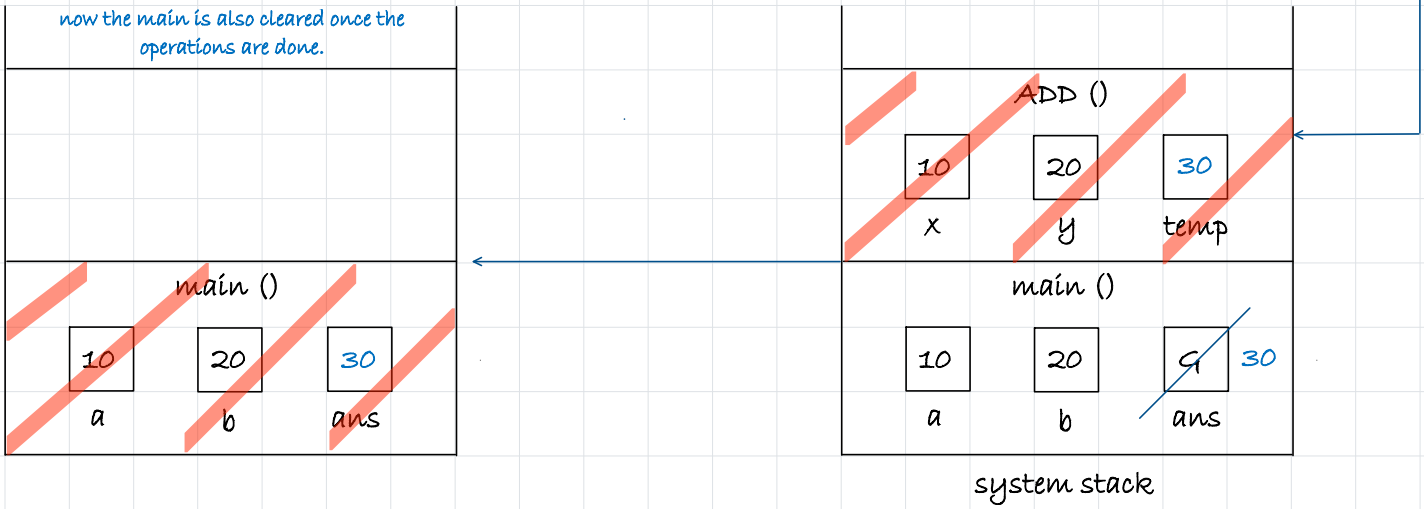
ADD function called

```
int ADD (intx , int y)
{
int temp ;
temp = x + y ;
return temp ;
        }
```





now the value stored and temp returned to ans and the activation record of ADD is cleared.



system stack

final output : 30

concept : formal arguments and actual arguments

```
#include<stdio.h>
void main () {
```

```
int ADD (int x, int y)
{
```

formal arguments

```
#include<stdio.h>
void main () {
        int a=10, b=20, ans;
        ans = ADD (a,b);
        printf ("%d", ans);
                }
```

actual arguments

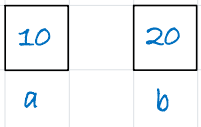
```
int ADD (int x, int y)
{
 int temp ;
 temp =  x + y ;
 return temp ;
        }
```

formal arguments

swapping function :

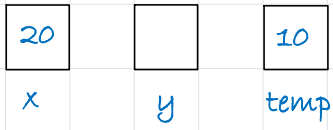
```
#include<stdio.h>
void main () {
        int a=10, b=20, ans;
        printf ("before swap");
        printf ("a=%d, b=%d", a,b);
        swap (a,b);
        printf ("a=%d, b=%d", a,b);
                }
```

```
void SWAP (int x, int y)
{
 int temp ;
 temp =  x ;
 x = y ;
 y = temp ;
        }
```



temp = x ;

x = y ;



y = temp;

but as soon as the function ends the memory returned from stack and x, y and temp does not exist anymore.

so, a and b does not change!

all changes are done at formal arguments and not in actal arguments.

note : changes performed on formal arguments will not be reflected back to actual arguments (parameters are passed by value) - call by value

topic : storage classes

(i) scope : part of code in which a variable is visible (where we can directly access the variable)

(ii) lifetime : duration (active/alive)

(iii) default value : if we do not initialise a variable then what is its default value.

(iv) storage area : where a variable is stored.

scope :

(a) block

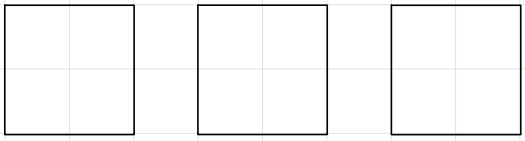
```
{
  —
  —
  —
}
```

(b) file

```
void f1 () {
(local to f1)  int a;
                    }
void f2 () {
(local to f2)  int b;
                    }
```

(c) multiple files

scope of variable : multiple files



auto / local

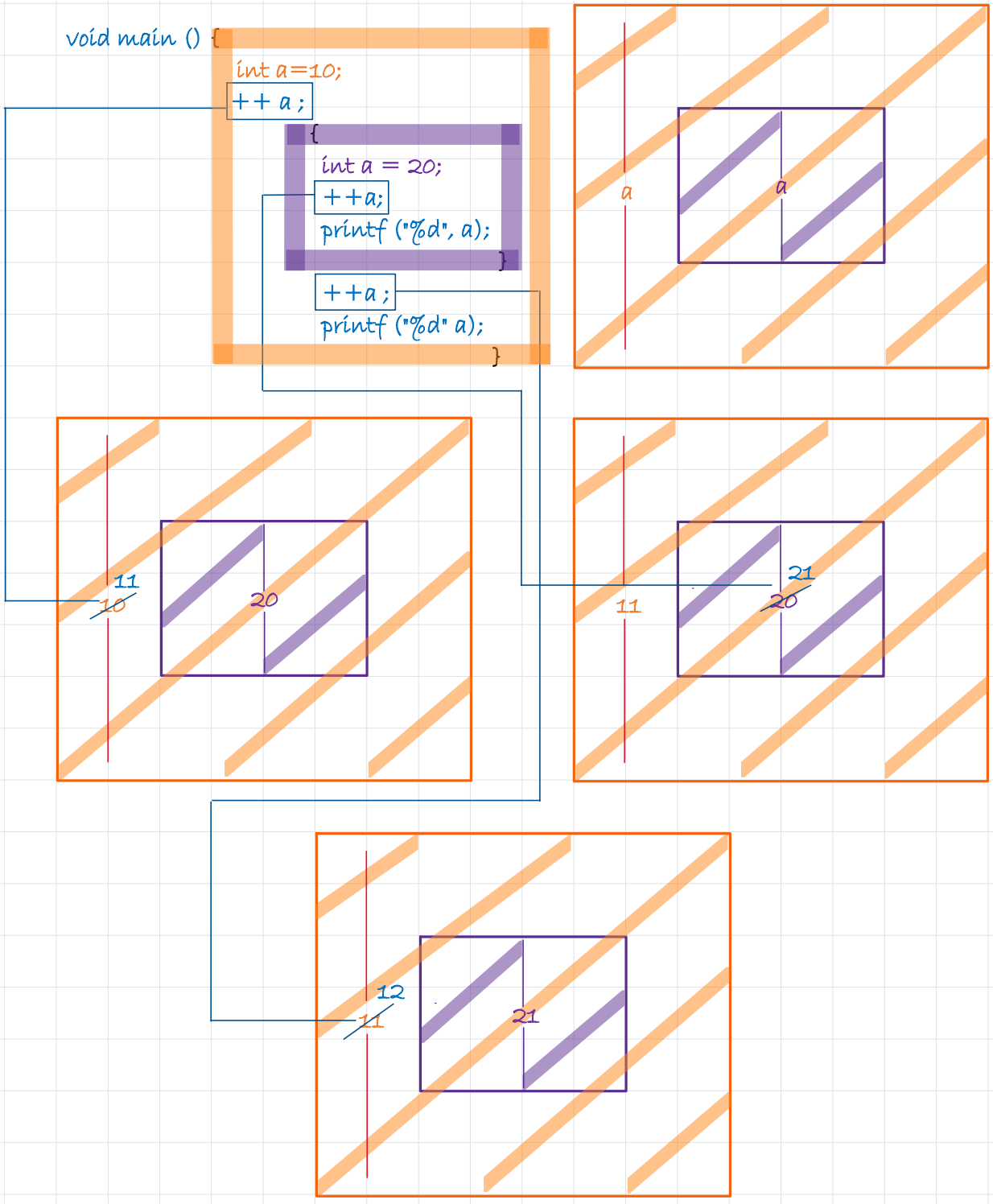
(i) scope : block in which they are defined/declared.

(ii) lifetime : block in which they are defined/declared.

(iii) default value : garbage.

(iv) storage area : stack.

```
#include<stdio.h>
void main () {
                int a=10;
                ++ a ;
                      {
                       int a = 20;
                       ++a;
                       printf ("%d", a);
                                          }
                       ++a ;
                       printf ("%d" a);
                                          }
```



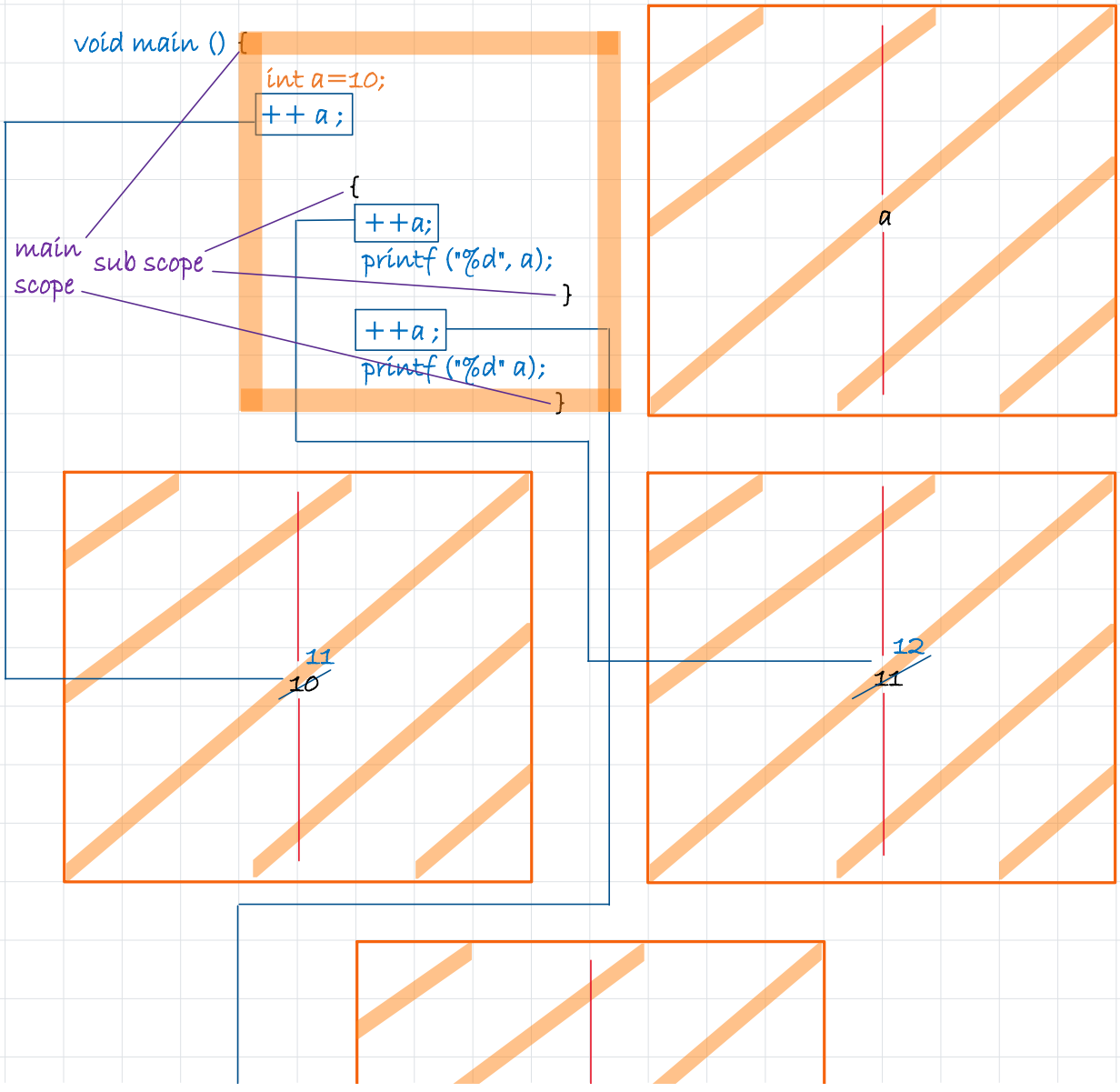
output : 12 and 21

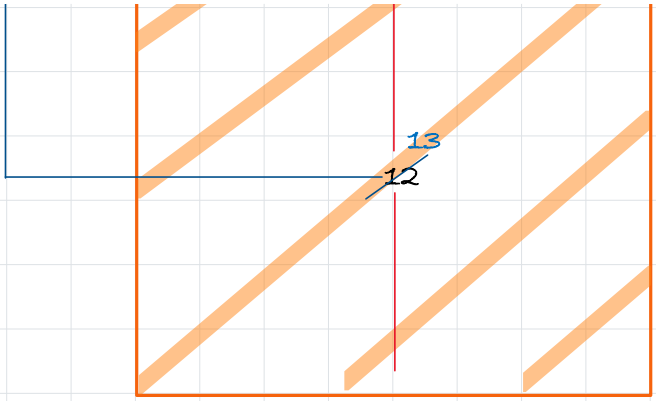
note : by default local scope is more preferred

in a same scope, declaring the same variable twice is not valid.

```
void main () {
        int a;
        int a;          ——— invalid
        }
```

```
#include<stdio.h>
void main () {
        int a=10;
        ++ a ;
            {
             ++a;
             printf ("%d", a);
                        }
             ++a ;
             printf ("%d" a);
                        }
```





output : 12 and 13

note : main scope variable is accesible within sub-scope

```
#include<stdio.h>
void main () {
                int a=10;
                ++ a ;
                        {
                         ++a;
                         printf ("%d", a);
                           {
                            int b = 5;
                            ++a ;
                            printf ("%d" a+b);
                                              }
                         ++b ;
                         printf ("%d%d" a,b);
                                                }
               ++a ;
               printf ("%d%d" a);
                                  }
```

```
#include<stdio.h>
void main () {
                int a=10;
                ++ a ;
                        {
                         ++a;
                         printf ("%d", a);
                           {
                            int b = 5;
```

all modify operators are working on main scope variable.
12

we can access "a" here because this sub-

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

we can access "a" here because this sub-scope is a part of main scope.
13+5=18

invalid because this sub-sub scope is trying to access b of sub-scope.

note : sub-scope variables are not accessible within main scope

auto : they are created automatically when we enter the block in which they are declared and destructed automatically when we exit the block.

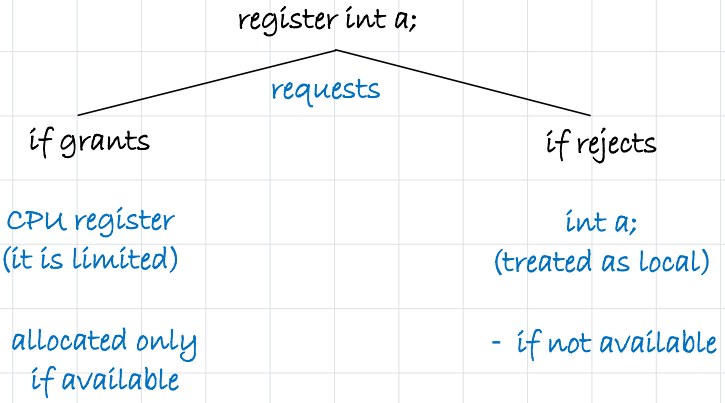
register variable

(i) scope : block in which they are defined/declared.

(ii) lifetime : block in which they are defined/declared.

(iii) default value : garbage.

(iv) storage area : CPU register / stack.



static variable

(i) scope : block in which they are defined/declared.

(ii) lifetime : program.

(iii) default value : 0

(iv) storage area : static area.

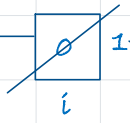
(i) value persists between different function calls.
(ii) no re-declaration.
(iii) they are created only once in program.

```
#include<stdio.h>
void fun() {
            static int i = 0;
            ++i ;
            printf ("%d", i);
                        }
void main (){
             fun();
             fun();
             fun();
                  }
```

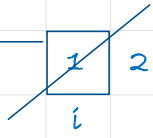
```
void fun() {
            static int i = 0;
            ++i ;
            printf ("%d", i);
                        }
void main (){
             fun();
             fun();
             fun();
                  }
```

(works on compile time, decision before execution time)

(i) fun()
++i

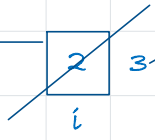


(ii) fun()
++i



→value persist throughout the program

(iii) fun()
++i



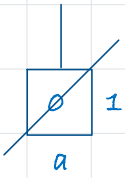
output : 123

difference between local variable and static variable

```
void fun() {
        int a = 0;
        ++a;
        printf ("%d", a);
                }
void main (){
        fun();
        fun();
        fun();
          }
```

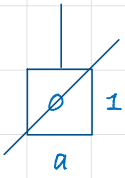
(i) fun()
int a = 0

++i



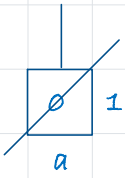
(ii) fun()
int a = 0

++i



(iii) fun()
int a = 0

++i



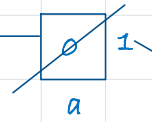
output : 111

value does not persist.

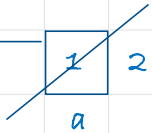
```
void fun() {
        static int x = 0;
        ++x;
        printf ("%d", x);
                }
void main (){
        fun();
        fun();
        fun();
          }
```

(works on compile time, decision before execution time)

(i) fun()
++a

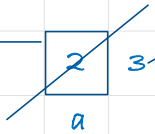


(ii) fun()
++a



no redeclaration because value persist throughout the program

(iii) fun()
++a



output : 1123

value does persist.

```
#include<stdio.h>
void fun() {
            int a = 0;
            static int b = 10;
            ++a;
            ++b;
            printf ("%d%d",a,b);
                                }
void main (){
             int a = 10;
             ++a;
             ++b;
             printf ("%d%d",a,b);
                                }
```

the scope of b is within the block

error, not available

```
#include<stdio.h>
void fun() {
            int a;
            static int b = a;
            printf ("%d%d",a,b);
                                }
```

you cannot initialise the static variable by a variable (always initialise by literals)

note : static variable works on compile time.

global variable

(i) scope : external.

(ii) lifetime :

(iii) default value : 0

(iv) storage area : static area.

global variable is not inside any function, variables are outside all the functions in a global variable.

```
#include<stdio.h>
int x;
void fun() {
            ++x;
            printf ("%d",x); 1
                              }
void g(){
          ++x;
          printf ("%d",x); 2
```

at this point compiler knows what is x

compilation top to bottom

```
void g(){
        ++x;
        printf ("%d",x); 2
                        }
void main(){
            f();
            g();
            ++x;
            printf ("%d",x); 3
```

```
#include<stdio.h>
[redacted]
void fun() {
            ++x;                    no information, compiler dont know about what is x
            printf ("%d",x);        trying to use something which is not defined.
                                }
int x = 10;
void g(){
        ++x;
        printf ("%d",x);
                        }
void main(){
            f();
            g();
            ++x;
            printf ("%d",x);
```

```
#include<stdio.h>
void fun() {
            extern int x;           forward declaration
            ++x;                    hence, no error.
            printf ("%d",x);
                                }
void g(){
        ++x;
        printf ("%d",x);
                        }
void main(){
            f();
            g();
            ++x;
            printf ("%d",x);
                                }
```

local forward declaration:

```
#include<stdio.h>
void fun() {
        extern int x;        ---- to avoid compilation error
        ++x;                      (no memory created)
        printf ("%d",x);
                        }
void g(){
        extern int x;        ---- to avoid compilation error
        ++x;                      (no memory created)
        printf ("%d",x);
                        }
int x = 10;          ---------- the "x" i am using is of global.
void main(){
        ++x;
        f();
        g();
        ++x;
        printf ("%d",x);
                        }
```

instead of local forward declaration, i can do it globally.

```
#include<stdio.h>
extern int x;        ---------- global forward declaration for global variable.
void fun() {
        extern int x;  ---- it is basically re-declaration, so compiler will ignore this information
        ++x;               because global declaration will give the information to compiler.
        printf ("%d",x);
                        }
void g(){
        ++x;
        printf ("%d",x);
                        }
void main(){
        ++x;
        f();
        g();
        ++x;
        printf ("%d",x);
                        }
int x = 10;
```

static + global :

```
#include<stdio.h>
```

global :

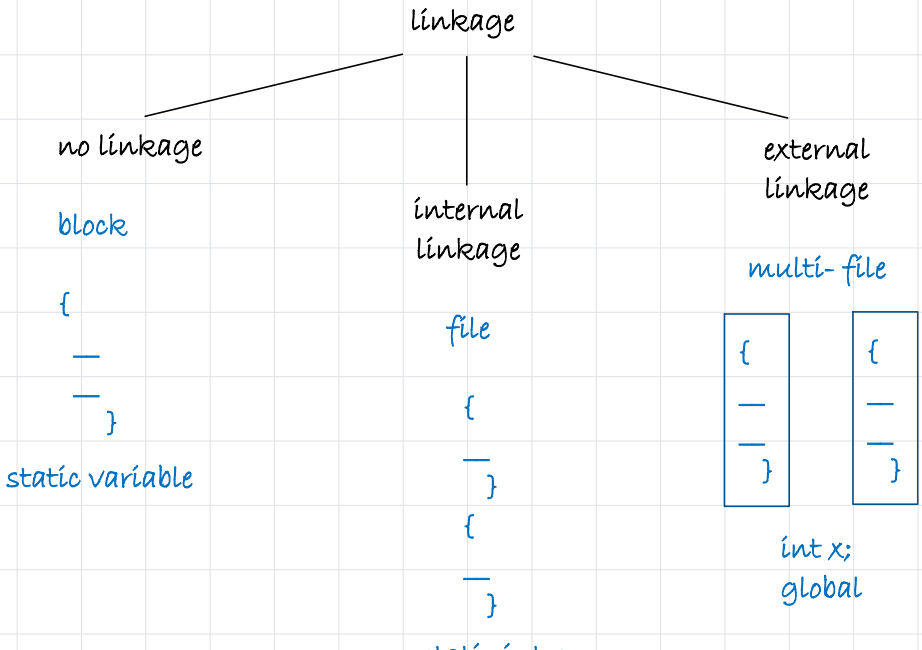
```
#include<stdio.h>
```

static + global :

```
#include<stdio.h>
static int i = 10;
void fun() {
            ____
            ____
                }
void g(){
            ____
            ____
                }
void main(){
            ____
            ____
            ____
                }
```

global :

```
#include<stdio.h>
int i = 10;
void fun() {
            ____
            ____
                }
void g(){
            ____
            ____
                }
void main(){
            ____
            ____
            ____
                }
```



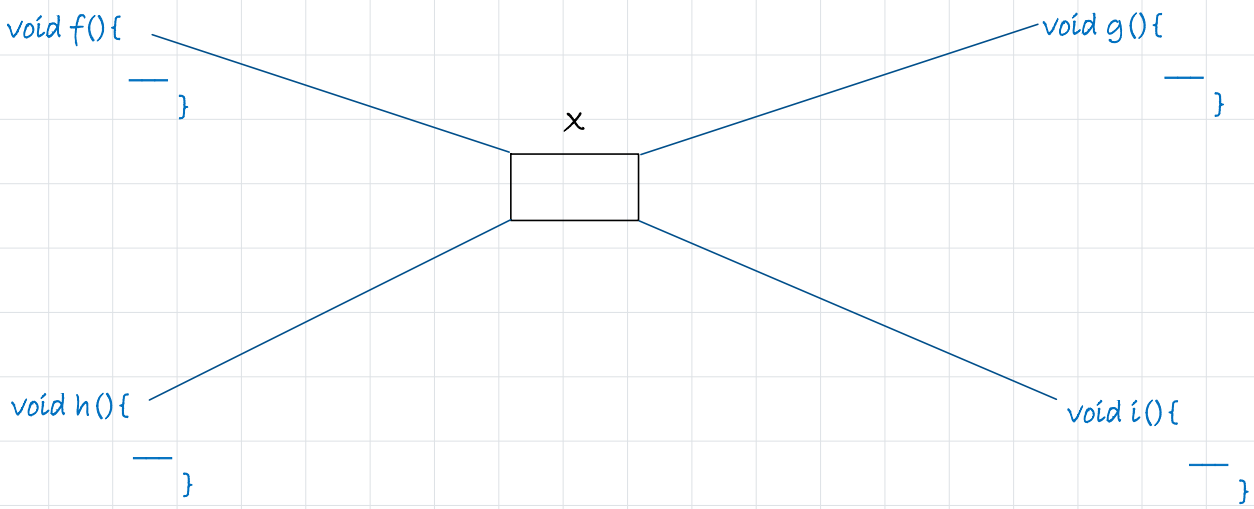
static global : cannot use in multiple environment

```
#include<stdio.h>
static int i ;
static int i ;
int main ()
```

globally redeclaration is allowed

this "x" variable can be mainpulated by these functions only, only these functions have the right to modify data in the variable.

modify data in the variable.



note : global assignment is not possible, only functions can change the value.

```
#include<stdio.h>
static int a ;
static int a ;
static int a  = 90;
static int a  = 100;
```

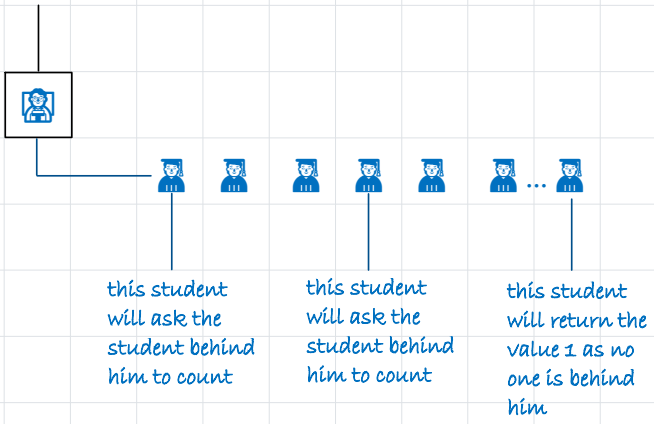
globally redeclaration is allowed

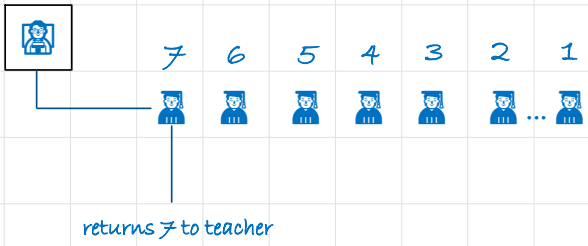
multiple initialization are not allowed, you can only initialize once.

recursion

whenever an operation is defined in terms of itself.

a teacher ask to student how many students behind you are including you?





small work done.

when n is small

- we can answer directly
- easy to solve
- no recursion is required

when n is large

- we cannot answer directly
- not easy to solve
- recursion is required

```
if (n is small) {
                we can answer directly
                easy to solve
                no recusrion is required
                                        }
                else {
                        input is large
                        cannot be answered directly
                        recursion is required
```

```
}
```

(i) code for $n \geq 1$

- i/p: n=2
  o/p : hihi

- i/p: n=3
  o/p : hihihi

- i/p: n=10
  o/p : hihihihihihihihihihi

har recusrive call kuch kaam khud karta hai aur baaki ka kaam recusrion karta hai

```
#include <stdio.h>
void fun (int n) {
        if (n==1)
        {
          printf ("hi");
          return;
              }
        else {
              printf ("hi");
              fun (n-1);
                    }
                     }
```

kuch kaam khud karta hai (when n is small)

to return to the function

baaki ka kaam recusion karta hai (when n is large, eg. n=1000)

to print (n-1) times.

(ii) code for $n > 0$

given a number of k digits, program have to give the output as sum.

- i/p: n=125
  o/p : 8

$n = d_1 d_2 \ldots d_k$

$d_1 + d_2 + \ldots + d_k$

- i/p: n=9
  o/p : 9

- i/p: n=2538
  o/p : 18

- i/p: n=1

o/p : 1

```
#include <stdio.h>
int sum_of.digits (int n) {
                        if (n>0 && n<=9)
                        return n;
                                }
                        else {
                                last = n%10;
                                remain = n/10;
                                last + sum_of.digits (n/10);
                                                        }
                                                         }
```

example :

sum_of.digits(2379)

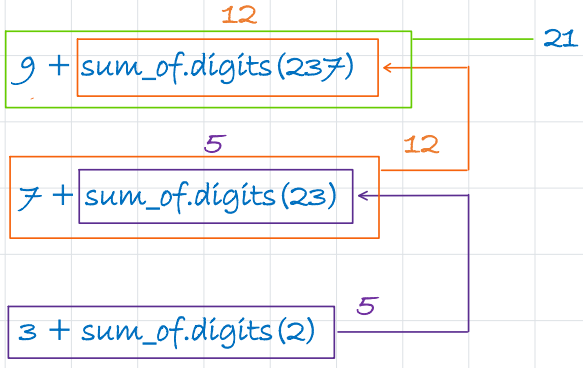
n%10 + sum_of.digits(2379) (n/10)

9 + sum_of.digits(237) 2379 % 10 = 9 (we got last digit)
2379 / 10 = 237 (we got first three digits)

7 + sum_of.digits(23) 237 % 10 = 7 (we got last digit)
237 / 10 = 23 (we got first two digits)

3 + sum_of.digits(2) 23 % 10 = 3 (we got last digit)
23 / 10 = 2 (we got first digit)

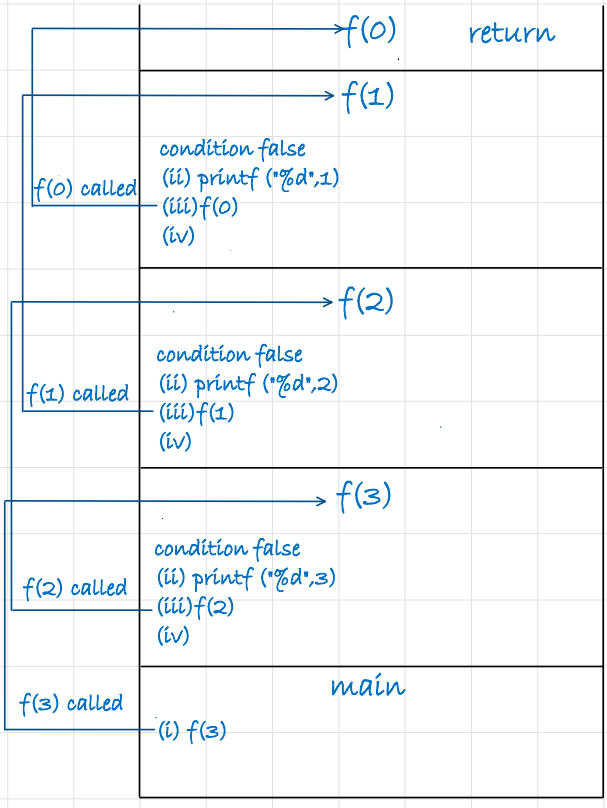
now it will return values in recusrive manner.

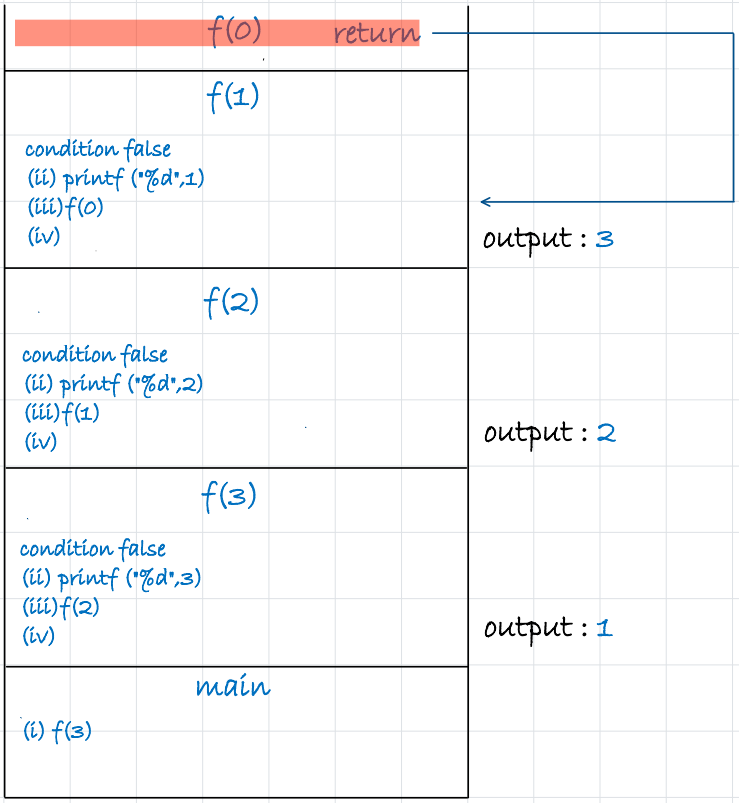


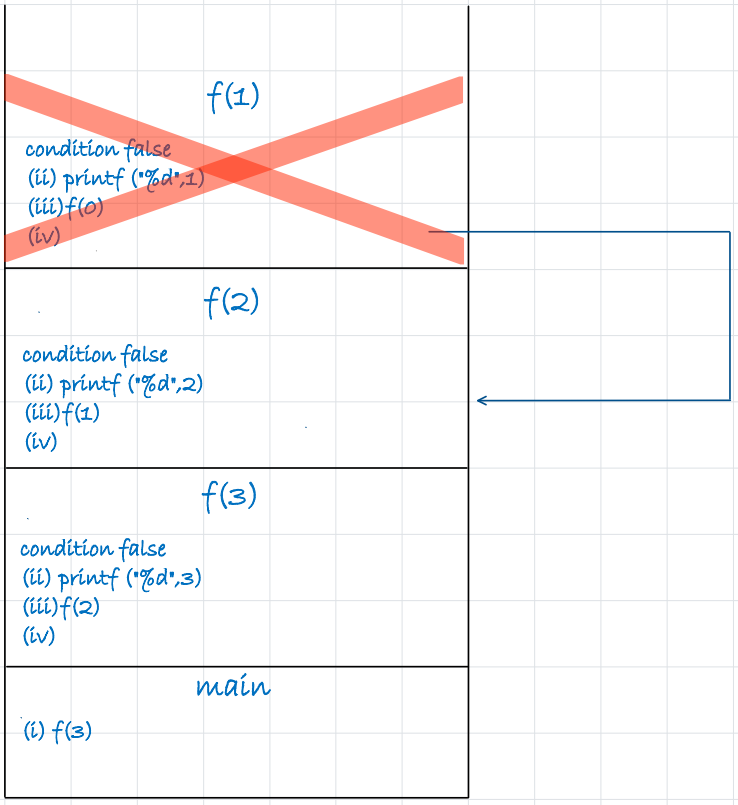
final output : 21

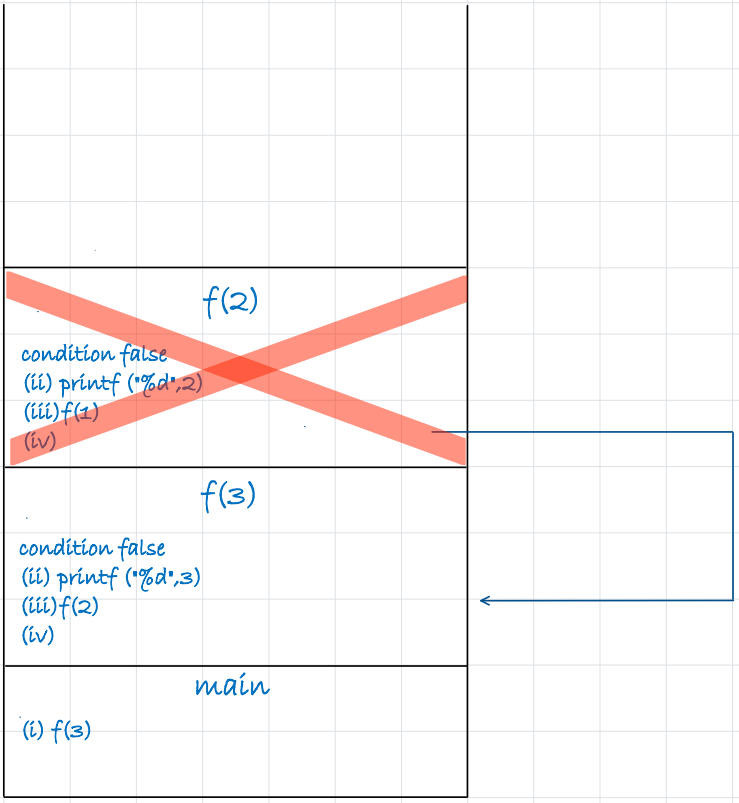
```
#include <stdio.h>
void f (int n) {
                if (n = = 0)
                return n;
                        }
                else {
                      printf ("%d", n);
                      f(n-1);
                           }
                void main {
                           f(3);
                              }
                                }
```

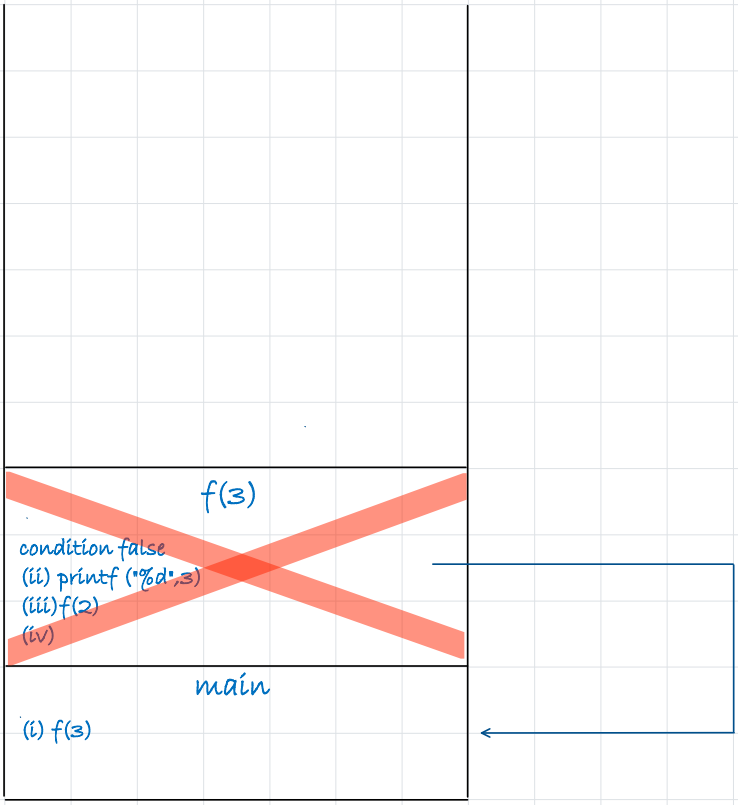
```
check condition {
                if (n = = 0)
                  return n;
                           }
              else {
                (ii)printf ("%d", n);
                (iii)f(n-1);
                (iv)          }
              void main {
                        (i) f(3);
                                 }
                                  }
```




f(0) return
f(1)
condition false
(ii) printf ("%d",1)
(iii) f(0)
(iv)
f(2)
condition false
(ii) printf ("%d",2)
(iii) f(1)
(iv)
f(3)
condition false
(ii) printf ("%d",3)
(iii) f(2)
(iv)
main
(i) f(3)
output : 3
output : 2
output : 1


f(1)
condition false
(ii) printf ("%d",1)
(iii) f(0)
(iv)
f(2)
condition false
(ii) printf ("%d",2)
(iii) f(1)
(iv)
f(3)
condition false
(ii) printf ("%d",3)
(iii) f(2)
(iv)
main
(i) f(3)

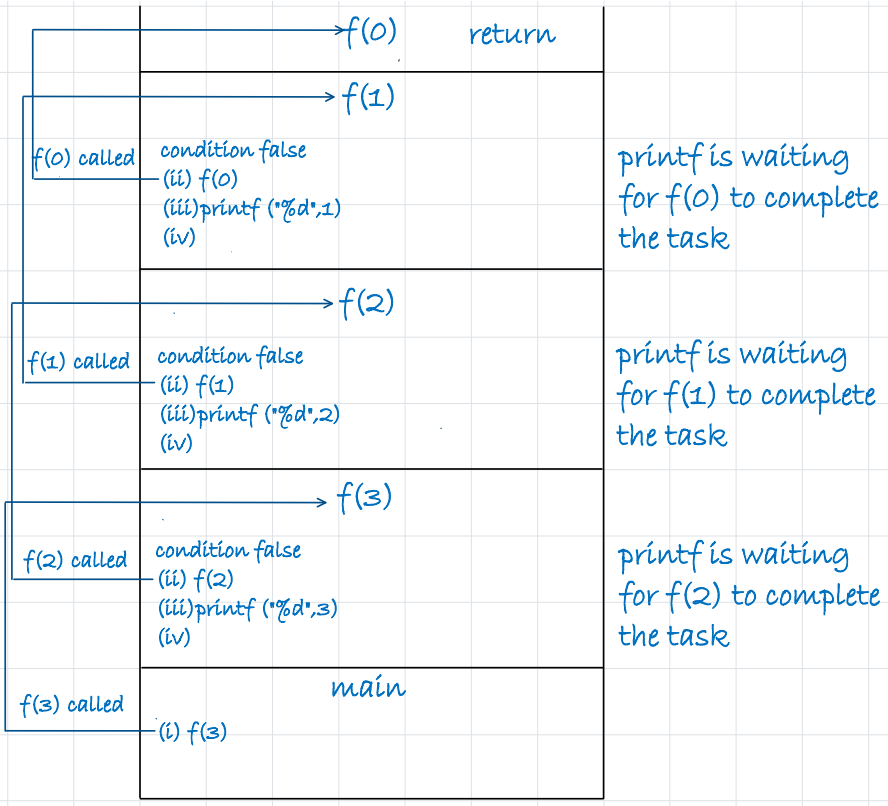

f(2)
condition false
(ii) printf ("%d",2)
(iii) f(1)
(iv)
f(3)
condition false
(ii) printf ("%d",3)
(iii) f(2)
(iv)
main
(i) f(3)

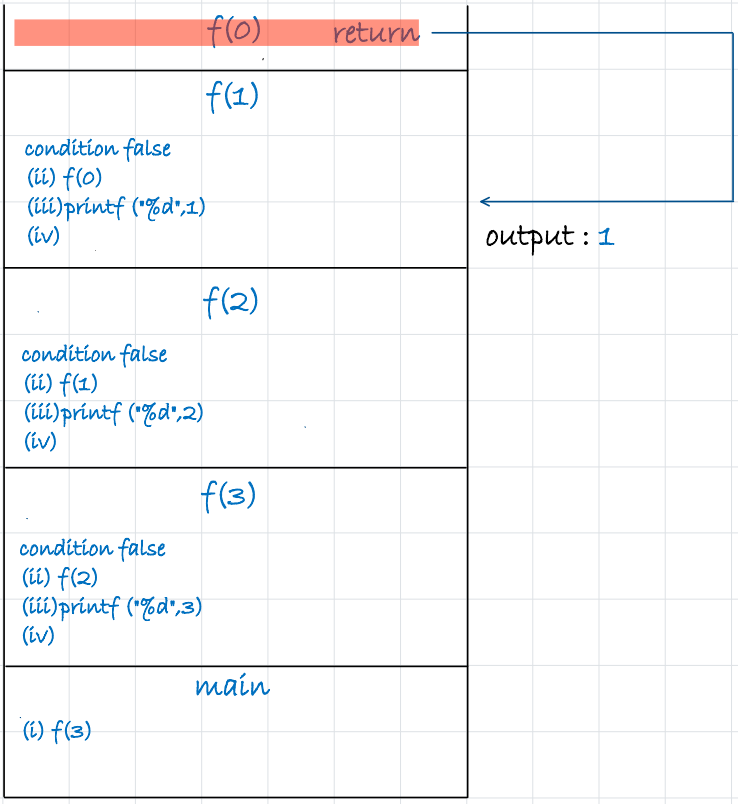

f(3)
condition false
(ii) printf ("%d",3)
(iii) f(2)
(iv)
main
(i) f(3)

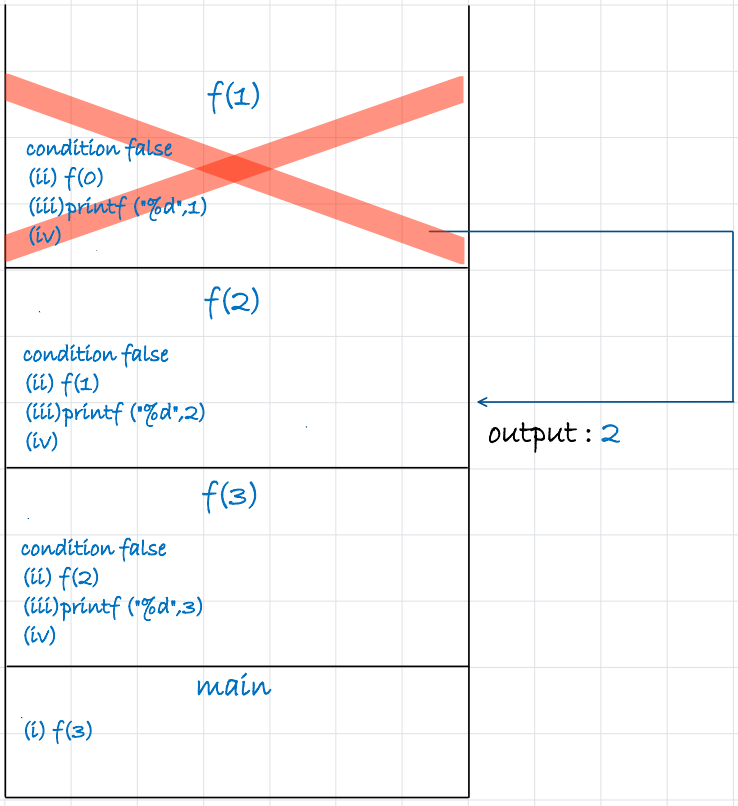
output : 321

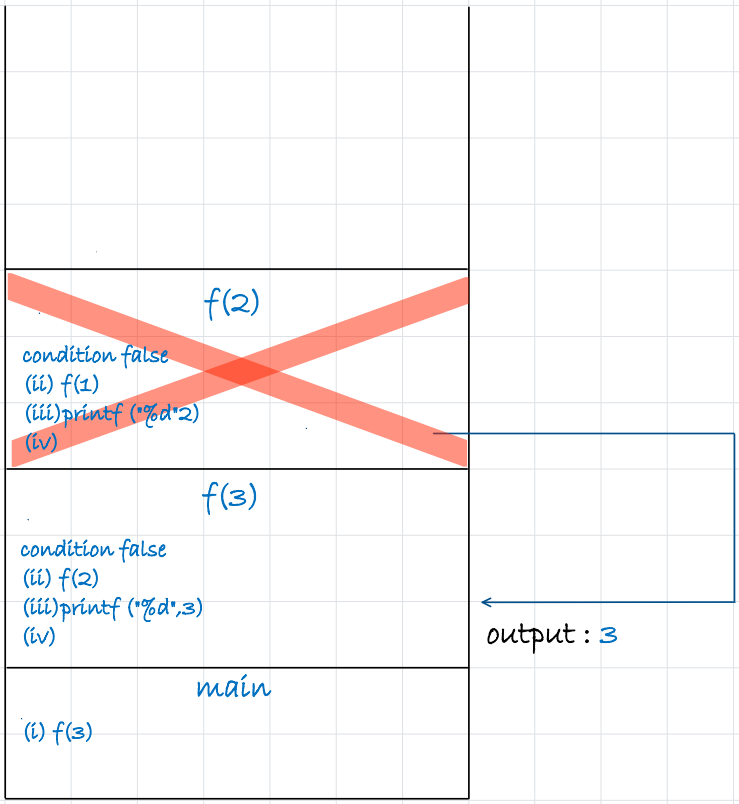
check condition

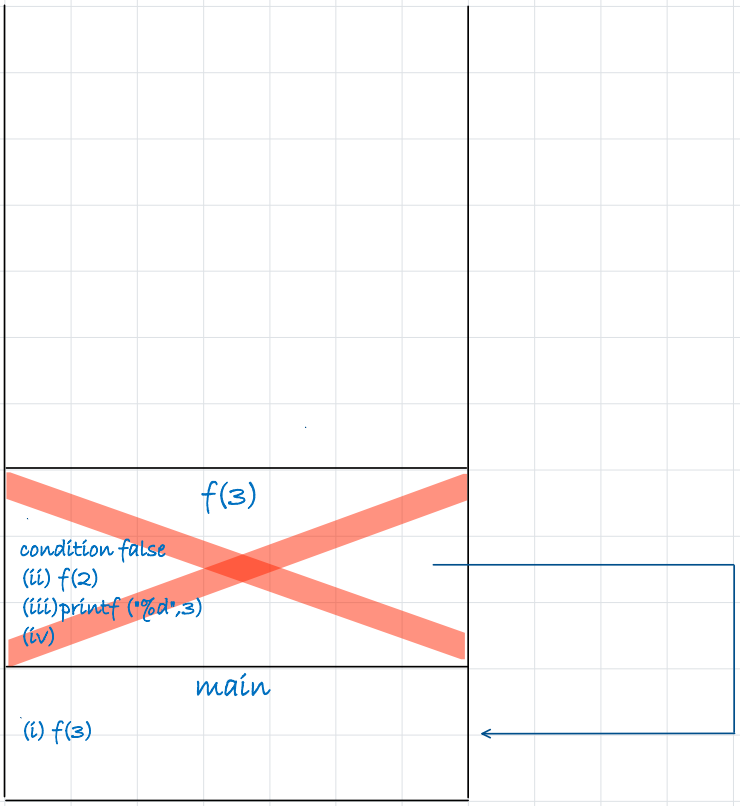
```
{
if (n == 0)
    return n;
        }
else {
  (ii)f(n-1)
  (iii)printf ("%d", n);
  (iv)                    }
void main {
        (i) f(3);
              }
               }
```









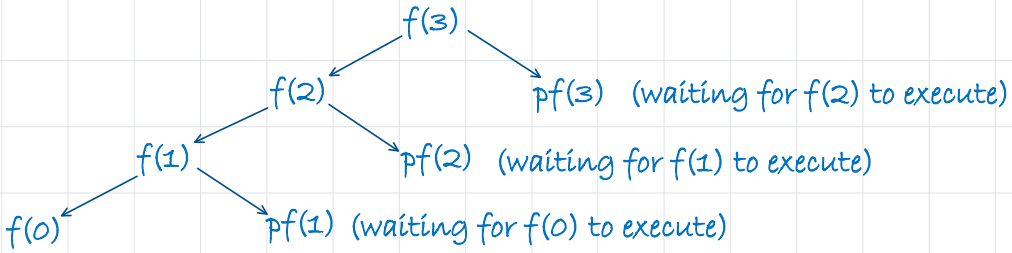


output : 123

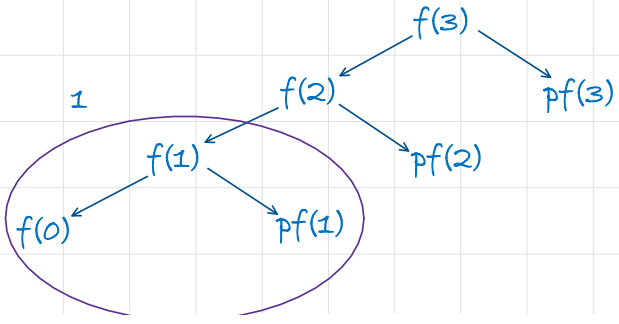
check condition

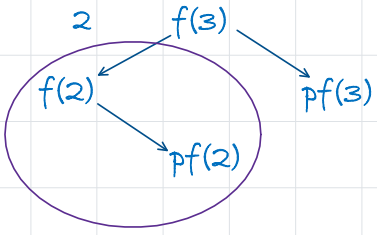
```
{
if (n = = 0)
  return n;
        }
else {
  (ii)f(n-1)
  (iii)printf ("%d", n);
  (iv)                    }
void main {
        (i) f(3);
              }
              }
```

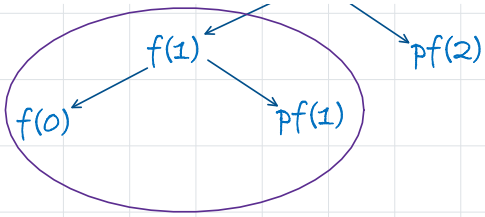
recursion tree

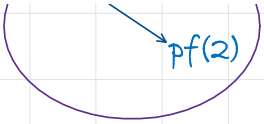


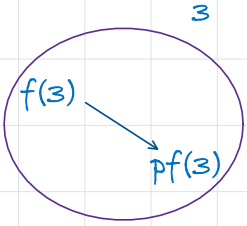
approach : top to bottom and left to right











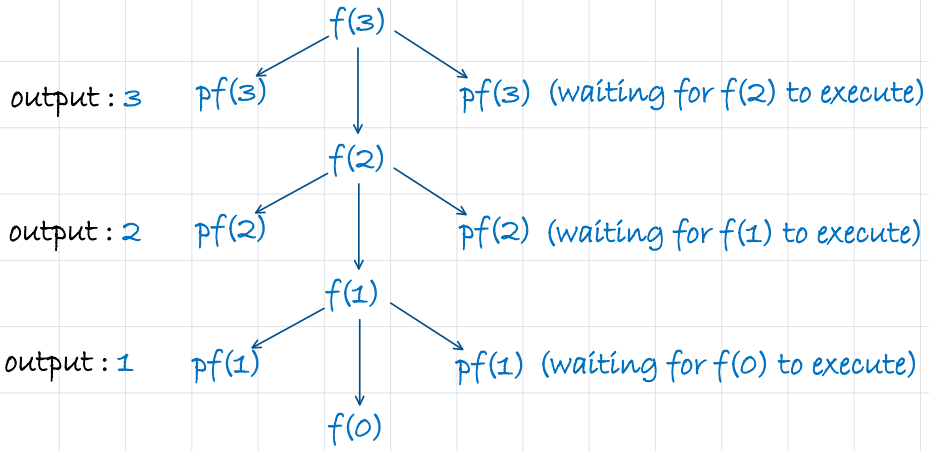
output : 123

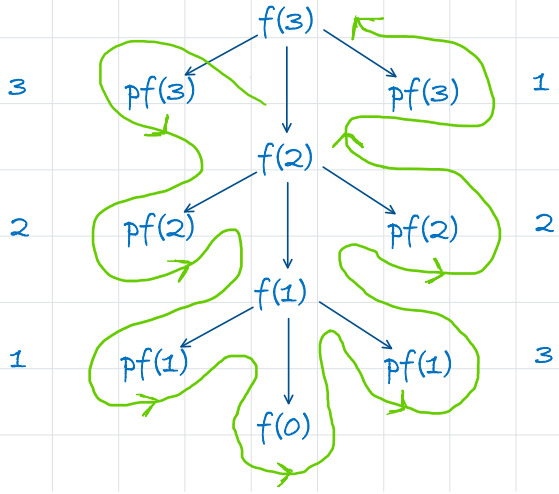
note : statements written after recusrive call execute in opposite order of call.

check condition

```
{
if (n = = 0)
  return n;
        }
else {
  (ii)printf ("%d", n);
  (iii)f(n-1)
  (iv)printf ("%d", n);
  (v)                    }
void main {
        (i) f(3);
              }
              }
```

recursion tree

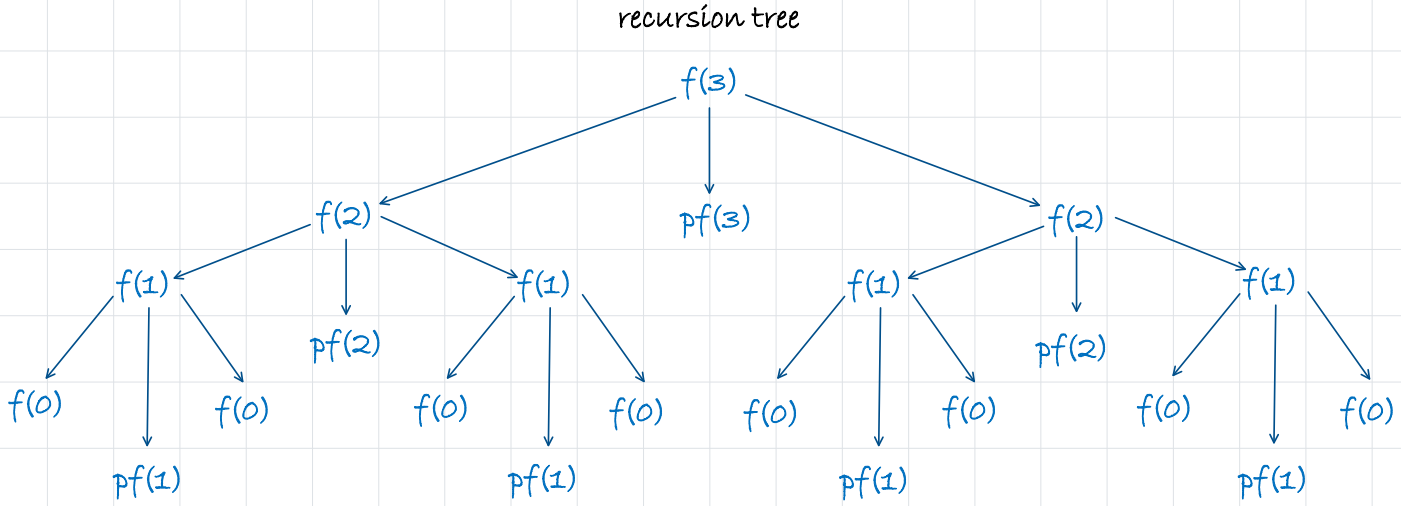


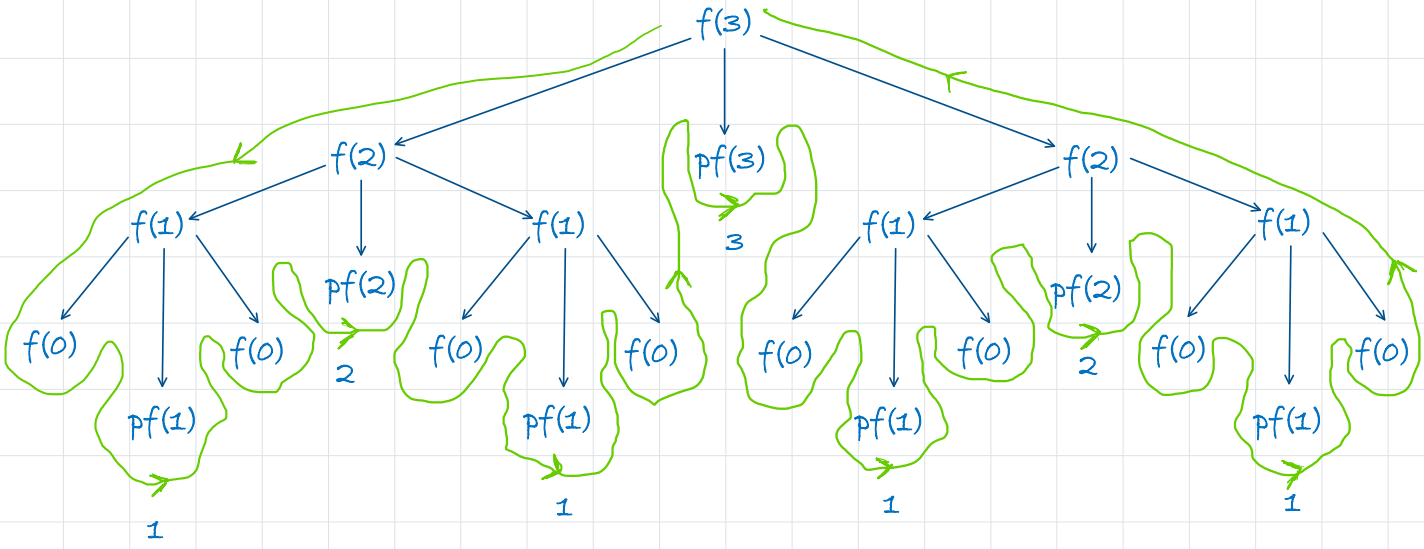


output : 321123

check condition

```
{
if (n = = 0)
  return n;
        }
else {
  (ii)f(n-1)
  (iii)printf ("%d", n);
  (iv)f(n-1)
  (v)       }
void main {
    (i) f(3);
          }
           }
```





output : 1213121

decimal to binary program

2	23	rem
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

binary : 10111

```
#include<stdio.h>
void main f(int n){
                    if (n is small)
                     {
                       easy case (1 and 0)
                       we can answer directly
                       no recusrion is needed
                                              }
                    else
                     {
                       we cannot answer directly
                       recusrion is needed
                                            }
                                              }
```

```
#include <stdio.h>
void f (int n) {
                  if (n = = 0 || n = = 1)
                  {
                   printf ("%d", n);
                   return;
                         }
                  else {
                       f(n/2);
                       printf ("%d", n%2);
                                          }
                                           }
```

2	23	rem
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

quotient : n/2
recursion ispar lagana hai

remainder to be printed in reverse order

power function program

a^b

$a,b > 0$

a^b using recursion

$3^{10} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$

$3^{10} = 3 \times 3^9$

$f(a,b) = a \times f(a, b-1)$

$a^b = a \times a^{b-1}$

```
#include <stdio.h>
int f (int a, int b) {
                        if (b = = 1)
                            return a;
                       else {
                             return a x f(a, b-1);
                                                  }
                                                    }
```

write a recursive code to print the octal equivalent of given no. (n)

```
#include<stdio.h>
void fun (int n) {
                       if (n<=0 && n>=7)
                       printf ("%d", n);
                 else{
                       f(n/8)
                       printf ("%d", n%8);
                                           }
                                             }
```

arrays and address

(i) address

(a) absolute address
(b) relative address

(a) absolute address :

A-106, krishna nagar
mathura-281004 (U.P.)

(b) relative address

we are calculating address using another address

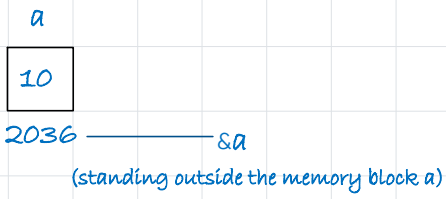
A-101	A-102	A-103	A-104	A-105	A-106

the resident in A-101 telling us about A-106 with respect to himself.

(i) address of operator (&)

int a = 10;

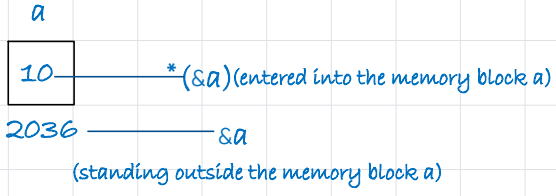
&a = 2036
(memory location 2036)



(ii) value at operator (*)

int a = 10;

&a = 2036
(memory location 2036)



*(&a) = value at (memory location 2036)
*(&a) = 10

now, a= 10 and also, *(&a) = 10
so basically, *&a = a
~~*&a~~
(value at) and (address of) cut each other.

q. why we use array?

- array is a group of similar elements
- collection of homogenous types of elements.

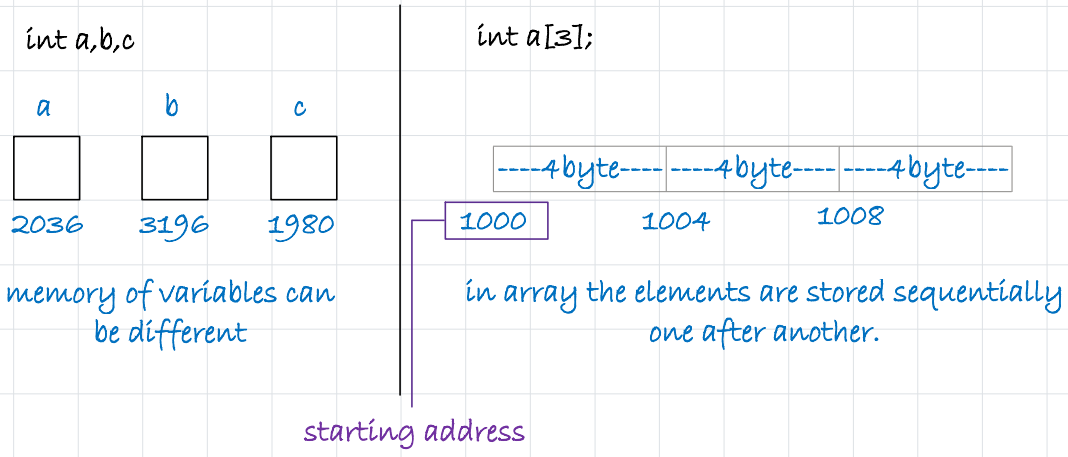
example :

int a,b,c = int a[3]

a,b,c are 3 variables of integer type.

a is group of 3 variables of integer type

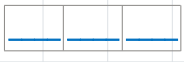
(i) the memory of variables can be different but in array the elements are stored sequentially one after another.



(ii) all 3 elements are represented by same name/entity.

```
int a,b,c;
a = 10;
b = 30;
c = 300;
```

int a[3]; - group/collection of 3 elements.

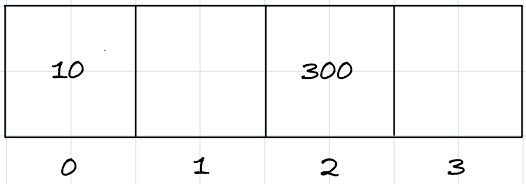


a is array of 3 elements

(iii) all the elements of array are contigiously stored.

(iv) for array there is a unique identification that we call indexes, and in C index always starts from 0.

```
int a[4];
```



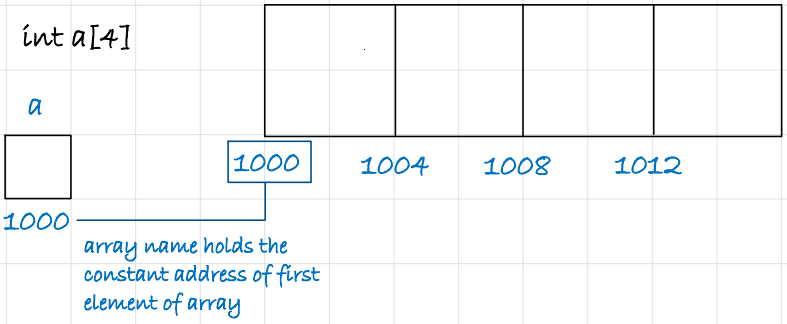

```
a[0] = 10;
a[2] = 300;
```

a — group name

[0] — index no.

(iv) array name represents the constant address of first elements of array.

```
int a[4]
```

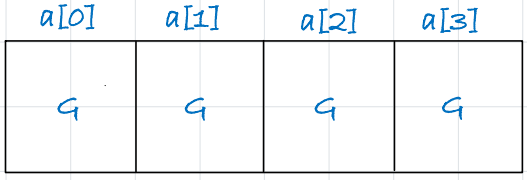


(v) if values are not assigned for array, the default value will be garbage.

```
void main () {
    int a[4];
    printf ("%d", a[0]);
```

local

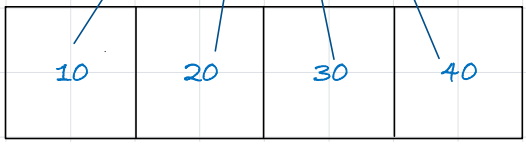
the default value for local is garbage

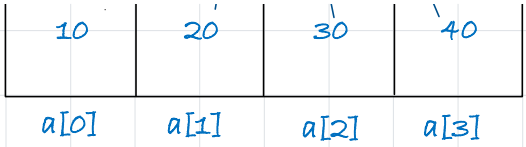


(vi) initialization and declaration, also elements are stored sequentially one after another.

```
int a;      ——— declaration
int a = 1;  ——— initialization
```

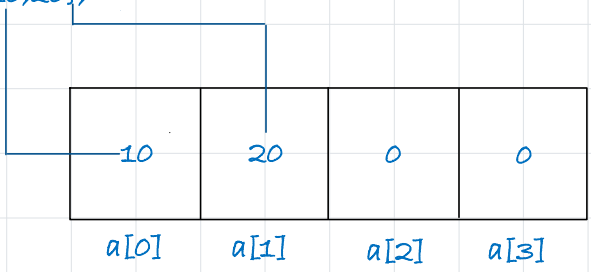
```
int a[4];                      ——— declaration
int a[4] = {10,20,30,40};      ——— initialization
```





(vii) the concept of garbage ends even if we initialise a single value in array, the rest of array will be filled with 0.

```
int a[4] = {10,20};
```



(viii) the array "[]" is invalid because it cannot be empty, the group size is 0.

```
int a[]
```
error

(ix) compiler will assume the elements if we initialise it.

```
int a[] = {10,20,30}  ⟶  int a[3] = {10,20,30}
```

array size 3

(x) whenever you create the array mention the size.

- int a[2+2];
- int a[2 x 3];
- int a[4 x size of(int)];

(xi) it is not meaningful to create array of 0 size because behaviour is undefined.

(xi) array_name is the constant address of first element of array so it cannot be Lvalue of any assignment statement, because Lvalue must be some variable.

array_name = 10;
1000 = 10;

Lvalue = Rvalue — constant

must be some variable — expression — variable

```
void main () {
    int a[4]; {10, 20, 30};
    printf ("%d", a[0]);
```

store information of array

```
void main () {
    int a[4];
    a = {10, 20, 30}
    printf ("%d", a[0]);
```

array_name cannot be L value it is invalid

(xii) array_name is the constant address of first element of array we cannot use modify operators.

2++; invalid

array_name++
++array_name
array_name - -
- - array_name

invalid

a[2]++ valid

int a[4] = {10, 20, 30, 40};

collection of variables — collection of values

(xiii) %u is preffered for array because address cannot be negative hence it is unsigned.

```
void main () {
        int a[4]; {10, 20, 30,40};
        printf ("%u", a); 10
        printf ("%d", &a[0]); 10
```

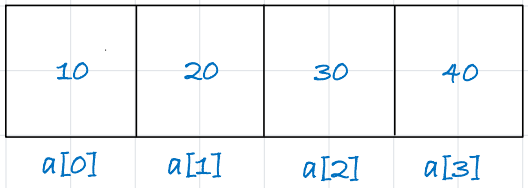
10	20	30	40
a[0]	a[1]	a[2]	a[3]

(xiv) name of array does not represent an address with 2 operators.

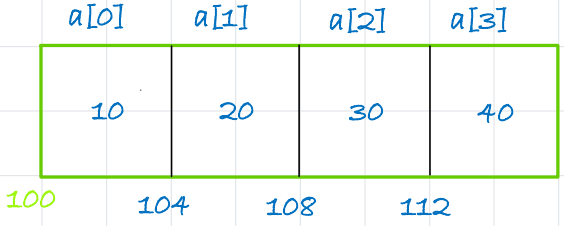
(a) address of operator (&a)

(a) size of operator

```
void main () {
        int a[4]; {10, 20, 30,40};
```

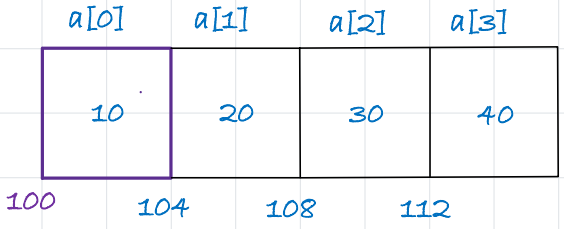


&a : address of whole array : 100 (16byte)



address of whole array is also the starting address of an array.

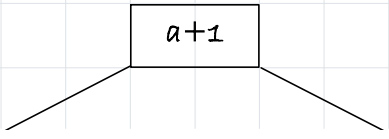
a : address of first element : &a[0] : 100 (4byte)



array name represent the constant address of first element of array.

concept : what is a?

numerical value of a is 100



a + 1

if a is a value (or) simple variable
int a = 100;
pf ("%d",a+1);
101
value + value = value

if a is address pointer variable
- address arithmetic
address + value = address
value+ address = address
address+ address = invalid
address : kiska address hai and uska size kya hai?

if declaration of array is having n-dimension then,
(a) element - anywhere in program, you provide exactly n-dimensions.
(b) address - anywhere in program, you provide less than n-dimensions.

what are dimensions?

one dimension

a[]

int a [4] = {10,20,30,40};

a : 0 dimensional

element?
no, because we
provided 0 dimension

address?
yes, because we provided
less than 1 dimension

two dimension

a[][]

int a [2][3] = {10,20,30,40};

a : 0 dimensional

element?
no, because we
provided 0 dimension

address?
yes, because we
less than 2 dimension

a[0] : 1 dimensional

element?
no, because we
provided 0 dimension

address?
yes, because we provided
less than 2 dimension

a[1][1] : 2 dimensional

element?
yes, because we
provided 2 dimension

three dimension

a[][][]

int a [3][3][3] = {10,20,30,40};

a : 0 dimensional

element?
no, because we
provided 0 dimension

address?
yes, because we provided
less than 3 dimension

a[0] : 1 dimensional

element?
no, because we
provided 1 dimension

address?
yes, because we provided
less than 3 dimension

a[0][1] : 2 dimensional

element?
no, because we
provided 2 dimension

element?
yes, because we provided 2 dimension

address?
no, because it is exactly 2 dimensional

element?
no, because we provided 2 dimension

address?
yes, because we provided less than 3 dimension

a[0][0][0] : 3 dimensional

element?
yes, because we provided less than 3 dimension

address?
no, because it is exactly 3 dimensional

int a [4] = {10,20,30,40};

a+1:

(a) what is a?
- a is address because it is 0 dimensional and array is of 1 dimension.

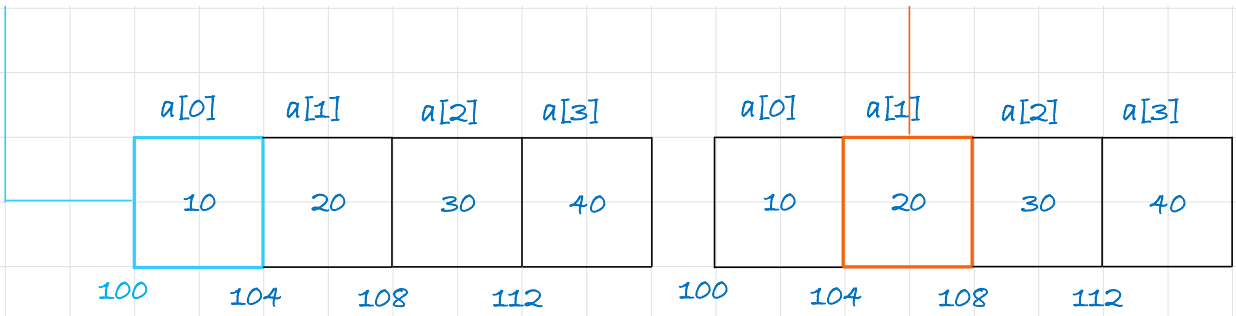
(b) whose address?
- a : array name is the constant address of first element.
&a[0] : 100

(c) size of a[0]?
- 4 bytes.

a+1

&a[0] + 1 x 4
= 100 + 4
= 104

(a+1) = (memory location 100 + 1x(size of))
(a+1) = (memory location 100 + 4)
(a+1) = memory location 104
(a+1) = &a[1]



int a [4] = {10,20,30,40};
a+2:

(a) what is a?
- a is address because it is 0 dimensional and array is of 1 dimension.

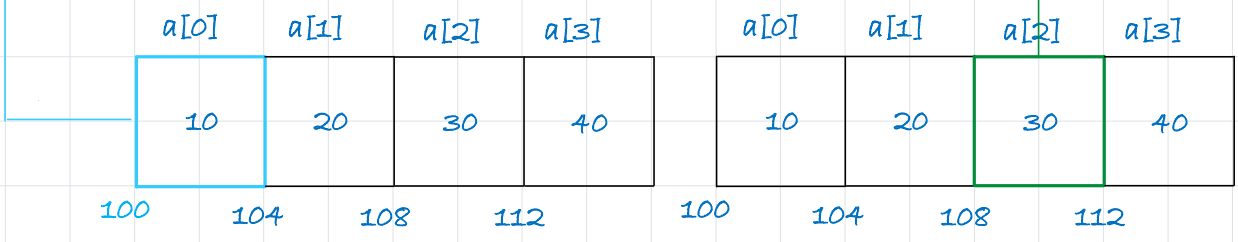
(b) whose address?
- a : array name is the constant address of first element.
&a[0] : 100

(c) size of a[0]?
- 4 bytes.

a+2

&a[0] + 2 x 4
= 100 + 8
= 108

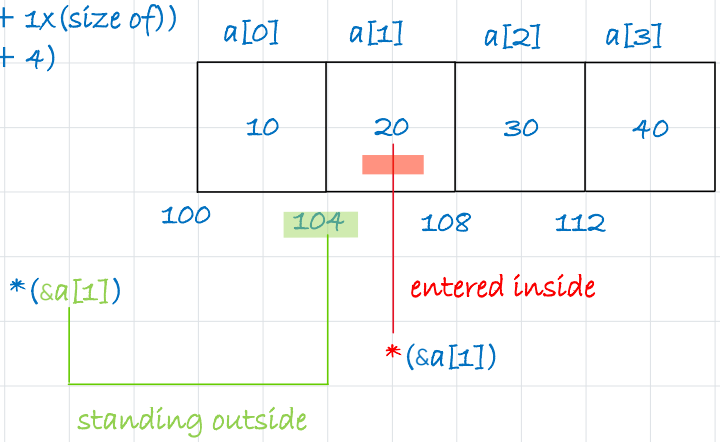
(a+2) = (memory location 100 + 2x(size of))
(a+2) = (memory location 100 + 8)
(a+2) = memory location 108
(a+2) = &a[2]



int a [4] = {10,20,30,40};

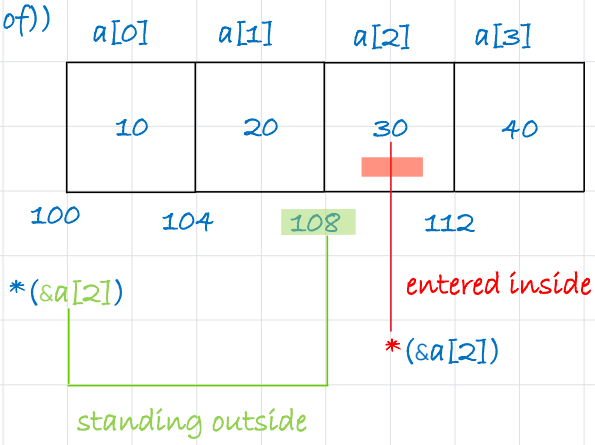
a+1
*(a+1) = value at(memory location 100 + 1x(size of))

a+1
*(a+1) = value at(memory location 100 + 1x(size of))
*(a+1) = value at(memory location 100 + 4)
*(a+1) = value at(memory location 104)
*(a+1) = value at(&a[1])
*(a+1) = a[1]
*(a+1) = 20



int a [4] = {10,20,30,40};

a+2
*(a+2) = value at(memory location 100 + 2x(size of))
*(a+2) = value at(memory location 100 + 8)
*(a+2) = value at(memory location 108)
*(a+2) = value at(&a[2])
*(a+2) = a[2]
*(a+2) = 30

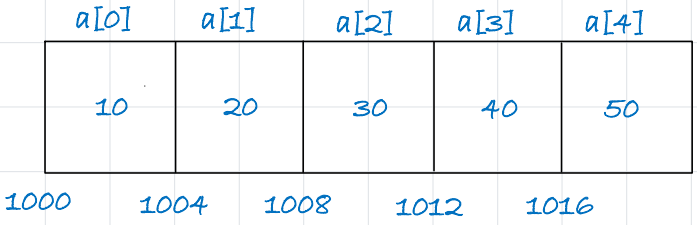


note :
*(a+i) = a[i]
- addition is commutative
a[i] = *(a+i)
a[i] = *(i+a)
a[i] = i[a]

but,
int 4[a] = {10,20,30,40);
in declaration, it is not valid.

```
void main () {
        int a[5]; {10, 20, 30,40,50};
```

a[0]	a[1]	a[2]	a[3]	a[4]



pf("%u", a); address of first element : &a[0] : 1000

pf("%u", &a); address of whole array: &a : 1000

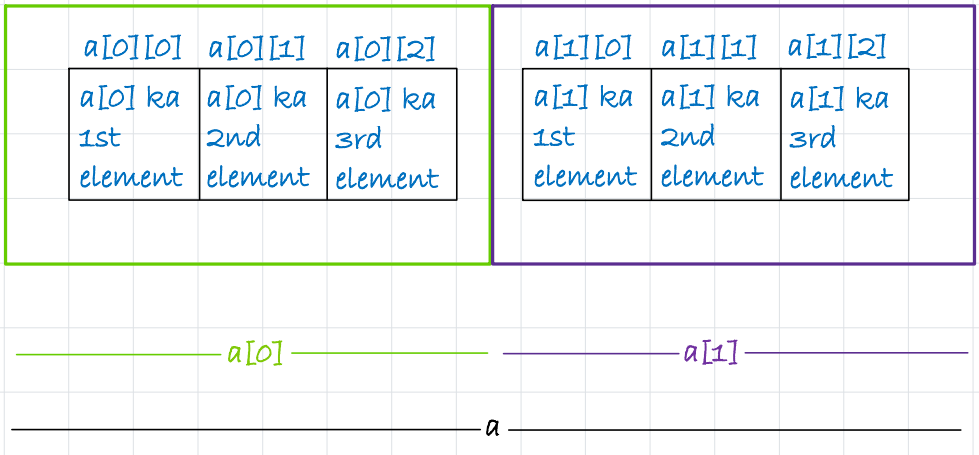
pf("%u", a+1); &a[0]+1 x 4 = &a[0]+ 4 = &a[1] = 1004

pf("%u", &a+1); &a+1 x 20 = 1000+ 20 = 1020

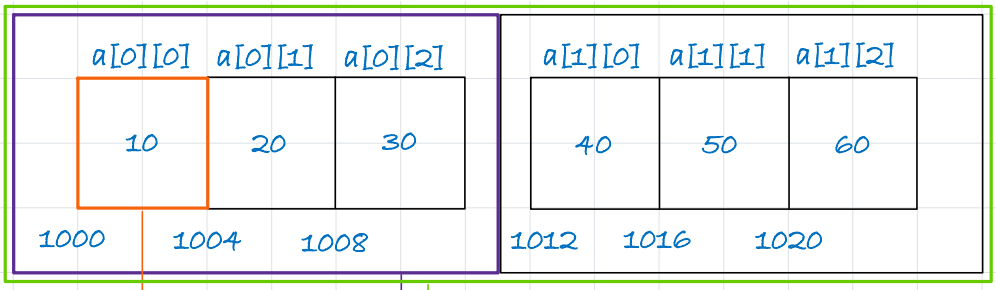
pf("%u", *(a+1); *(&a[0]+1 x 4) = *(1000+ 4) = *(1004) = 20

concept : 2D array

int a[2][3]




```
void main () {
        int a[2][3]; {10, 20, 30,40,50,60};
```



pf("%u", a); address of first element : &a[0] : (12byte) 1000

pf("%u", a[0]); &a[0][0] : 4 byte 1000

pf("%u", &a); address of whole array: &a : 24 byte 1000

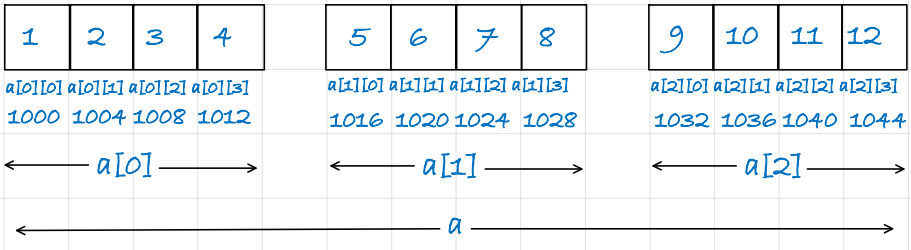
pf("%u", a+1); &a (address of first element) +1 x 12 (size of a[0]) = 1012

pf("%u", &a+1); &a (address of whole array) +1 x 24 (size of whole array) = 1024

pf("%u", a[0]+1); &a[0][0] +1 x 4 (size of a[0][0]) = 1004 (&a[0][1])

a[0] : address (less than n-dimesion)

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



pf("%u", a); address of first element : &a[0] : (12byte) 1000

pf("%u", &a); address of whole array: &a : (48 byte) 1000

pf("%u", a[0]); &a[0][0] : (4 byte) 1000

pf("%u", a+1); &a[0] +1 x 16 (size of array) = &a[0][4] : 1016

pf("%u", a[0]+1); &a[0][0] +1 x 4 (size of array) = &a[0][1] : 1004

pf("%u", &a+1); &a +1 x 48 (size of whole array) = 1048

pf("%u", *a); ~~*&a~~ = a[0] : a[0][0] = 1000

it is address due to less than 2-Dimension

pf("%u", **a); ~~*&a~~ = *(a[0]) = ~~*&a~~[0][0] = a[0][0] = 1

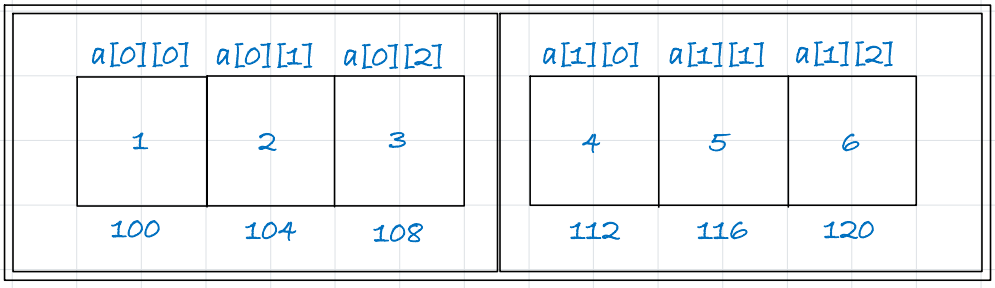
it is address due to less than 2-Dimension

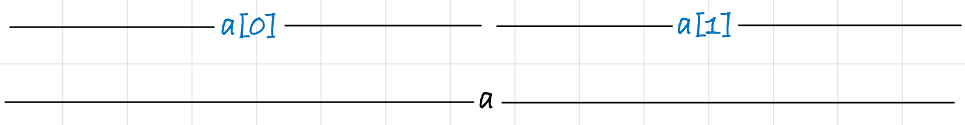
it is element due to exactly 2-Dimension

pf("%u", *a+1); ~~*&a~~+1 = (a[0]+1) = a[0][0]+1 = a[0][1] = 1004

it is address due to less than 2-Dimension

```
void main () {
        int a[2][3]; {10, 20, 30,40,50,60};
```





pf("%u", a[0]); &a[0][0] : 100

pf("%u", a[0]+1); &a[0][0]+1 = a[0][1] = 104

pf("%u", *(a[0]+1); *&a[0]+1 = a[0]+1 = a[0][0]+1 = a[0][1] = 2

it is address due to less than 2-Dimension

it is element due to exact 2-Dimension

pf("%u", *(a[0]+2); *&a[0]+2 = a[0]+2 = a[0][0]+2 = a[0][2] = 3

it is address due to less than 2-Dimension

it is element due to exact 2-Dimension

*(a[0]+j) = a[0][j]

pf("%u", a[1]+1; &a[1][0]+1 = a[1][1] = 116

pf("%u", *(a[1]+1); *&a[1]+1 = a[1]+1 = a[1][0]+1 = a[1][1] = 5

it is address due to less than 2-Dimension

it is element due to exact 2-Dimension

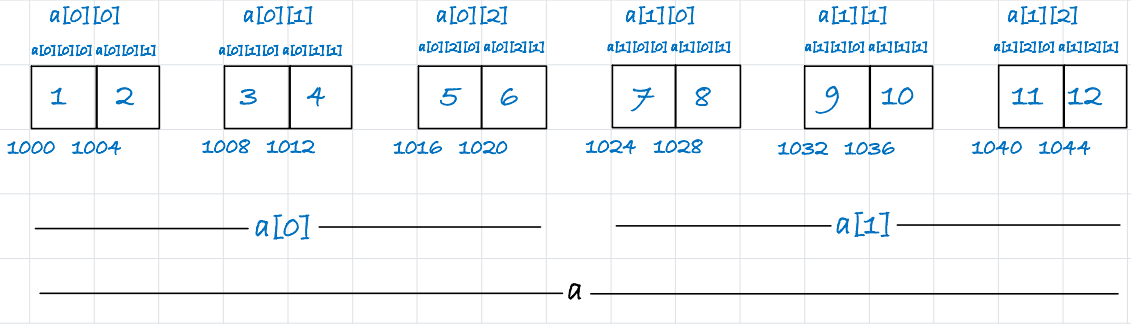
pf("%u", *(a[1]+2); *&a[1]+2 = a[1]+2 = a[1][0]+2 = a[1][2] = 6

it is address due to less than 2-Dimension

it is element due to exact 2-Dimension

*(a[i]+j) = a[i][j]

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



pf("%u", a); &a[0] : 1000 (24 byte)

pf("%u", a[0]); &a[0][0] = 1000 (8 byte)

pf("%u", a[0][0]); &a[0][0][0] = 1000 (4 byte)

pf("%u", &a); 1000 (48 byte)

pf("%u", a+1; &a[0]+1 : 1000+1 x 24 = 1024 (a[1])

pf("%u", a[0][0]+1); &a[0][0][0]+1 = 1000+1x4 =1004 (a[0][0][1])

pf("%u", &a+1); 1000 + 1 x 48 = 1048

pf("%u", *a); *&a[0] = a[0] = a[0][0] = 1000 (8 byte)

pf("%u", **a+1); *(*&a[0]+1) = *(a[0][0]+1) = *&a[0][0] = a[0][0][0]+1 = a[0][0][1] = 1004

pf("%u",***a+1); **(*&a[0]+1) = **(a[0][0]+1) = *(*&a[0][0]+1) = *(a[0][0][0]+1) = 1+1 =2

topic : declaration and initialization

(i) int a [] ; invalid

(ii) int a [] = {10, 20, 30}; valid

if only declaration is there without initialization, it is mandatory (compulsory) to provide the size of each dimension.

(iii) int a[2][3]; valid

in case, we are initializing an array, there is flexibility that you can omit the size of ist dimension.

(iv) int a [][2] = {10, 20, 30}; valid

optional

no other dimension is having such flexibility.

(v) int a [x][3] = {1, 2, 3, 4, 5, 6}; valid

6 elements.

x * 3 = 6

x = 6/3 = 2

so, compiler will assume

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

(vi) int a [x][3] = {1, 2, 3, 4}; valid

4 elements.

x * 3 = 4

x = 4/3 = 1.33 = 2

so, compiler will assume

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

by default, the rest of 2 arrays will be filled with 0

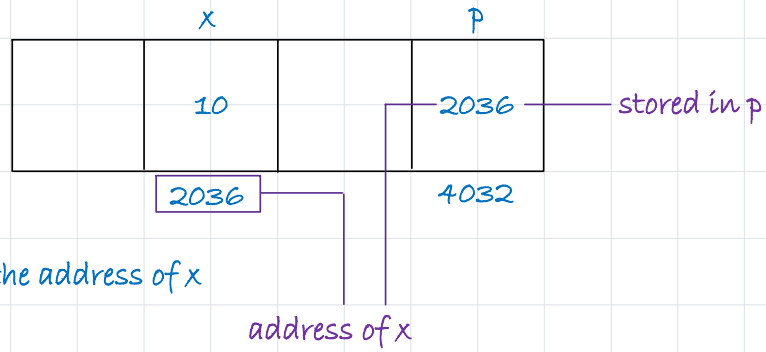
pointers

it is a special variable that are used to store the addresses of other variables.

`int *p`

- p is a pointer to integer.
- p have the address of integer variable.
- p can store the address of some integer variable.

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



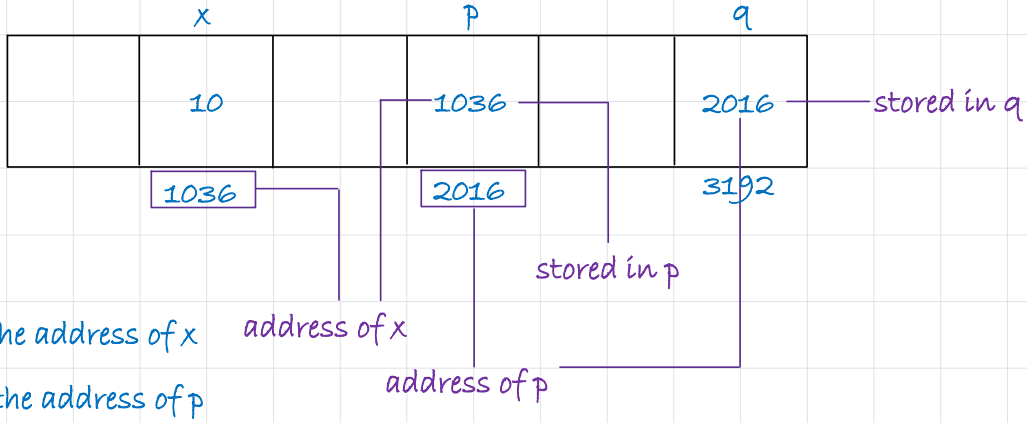
p = &x ——— p is storing the address of x

p = memory location 2036
*p = value at (memory location 2036) = 10

`int **q`

- q is a pointer to pointer to integer.
- q have the address of pointer to integer.
- q can store the address of some pointer.

```
int x = 10;
int *p;
int **q;
p = &x
q = &p
```



p = &x ——— p is storing the address of x

q = &p ——— q is storing the address of p

p = memory location 1036
*p = value at (memory location 1036) = 10
q = memory location 2016
*q = value at (memory location 2016) = 1036
**q = value at (memory location 1036) = 10

pf ("%u",p); 1036 (memory location 1036)

pf ("%u", *p); 10 (value at (memory location 1036))

pf ("%u",q); 2016 (memory location 2016)

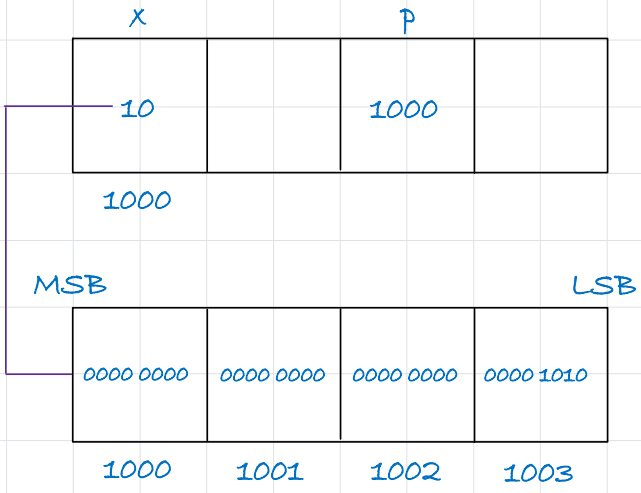
pf ("%u",*q); 1036 (value at (memory location 2036))

pf ("%u",**q); 10 (value at (memory location 1036))

concept : dereferencing

dereferencing : fetching value through address (value at operator lagate hai hum use de-referencing bolte hai)

```
int x = 10;
int *p;
```



pf ("%u",*p); 10

dereferencing

(i) in the case of pointer, everything depends upon declaration.

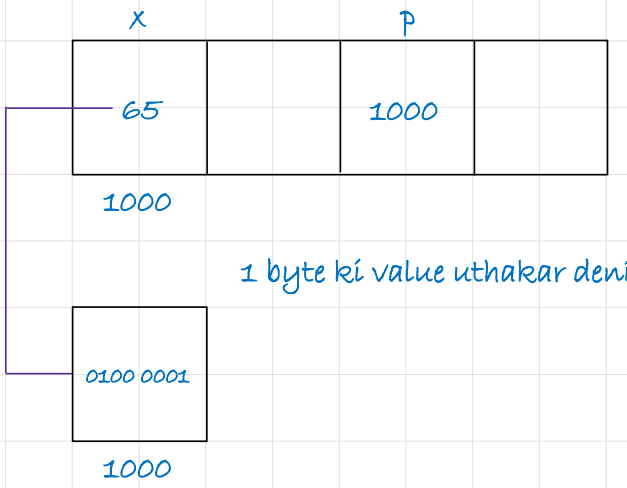
int *p ——— compiler 4 bytes uthayega (4 byte)

char *q ——— compiler 1 byte uthayega (1 byte)

kitne byte uthayega compiler vo depend karta hai declaration par.

kitne byte uthayega compiler vo depend karta hai declaration par.

```
char x = 65;
char *p; (p is a pointer
          to character)
p = &x;
```




```
pf ("%d",*p); 10
```

```
int [4] = {10, 20, 30, 40};
int *p; (p contains the
         address of 4 byte)
p = &a[1];
p = p+1;
```

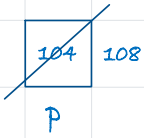
p contains the address of a[1] = 104


```
p = p+1;
```

address + value = address
value + address = address

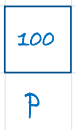
address value

p = 104 + 1 x4
p = 108




```
int [4] = {10, 20, 30, 40};
int *p; (p contains the
         address of 4 byte)
p = &a[0];
p = p++;
p =++p;
```

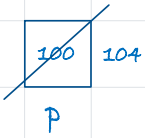
p contains the address of a[0]




```
p = p++;
```

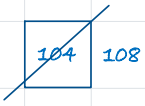
(i) use the value (useless)
(ii) increase the value

p = p+1
p = 100 + 1 x 4
p = 104



p =++p;
(i) increase the value

p = p+1
p = 104 + 1 x 4
p = 108

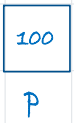


(ii) use the value

difference between updation and updation + storing

int [4] = {10, 20, 30, 40};
int *p =a; (p contains the address of 4 byte)
int *p = a
int *p
p = a

p contains the address of a



a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

only updation	updation + storing
p+2	p = p+2
100 + 1 x 8 = 108	p = 100 + 1 x 8
100	p = 108
p	100 108
	p

int *p; — declaration
p = a; — initialization
int *p=a — declaration+initialization

int [4] = {10, 20, 30, 40};
int *p =a; (p contains the address of 4 byte)
int *p = a
int *p
p = a

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

100 p contains the address of a[0]

p

pf ("%d", *p); value at (memory location 100) : 10

pf ("%d", *(p+1)); value at (memory location 100+1x4)
value at (memory location 100+4)
value at (memory location 104) : 20

pf ("%d", *(p+2)); value at (memory location 100+2x4)
value at (memory location 100+8)
value at (memory location 108) : 30

pf ("%d", *(p+3)); value at (memory location 100+3x4)
value at (memory location 100+12)
value at (memory location 112) : 40

p : p+0
*(p+i) = p[i]

pf ("%d", p[0]); 10

pf ("%d", p[1]); 20

pf ("%d", p[2]); 30

pf ("%d", p[3]); 40

equivalence between arrays and pointers :

arrays : elements store contigiously

pointer : +1 karne par next element

difference between using modify operators on array and pointers :

arrays

int a[4] = {10, 20, 30, 40}

```
a++;
++a;
a--;
--a;
```
— invalid

invalid, because array_name is the constant address of its first element.

pointers

int a[4] = {10, 20, 30, 40}

int *p = a;

p++ ; p = p+1
++p ;
address value

valid, because we can update P.

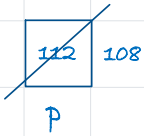
int [4] = {10, 20, 30, 40};
int *p =&a[3];
--p;

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

112 p contains the address of a[3]

p

--p;
(i) decrease the value
p = p - 1
p = 112 - 1X4
p = 108



(ii) use the value

modifying pointers :

(i) pointer + pointer : invalid

(ii) ++ptr, --ptr, ptr++, ptr-- : valid

(iii) ptr + 3 : valid (moving 3 locations in forward direction)

(iv) ptr - 3 : valid (moving 3 locations in backward direction)

when the subtraction of pointers are valid?

ptr 1 - ptr 2 : logically invalid, only when both of them are pointing to elements of same array.

```
int [4] = {10, 20, 30, 40};
int *p *q;
p = &a[3];
q= &a[0];
```

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

112 p contains the address of a[3]

p

100 q contains the address of a[0]

q

$$p\text{-}q = \frac{\text{(actual difference)}}{\text{integer size}}$$

$$\frac{112 - 100}{4} = \frac{12}{4} = 3$$

no. of elements between these 2 address are 3

priority between ++ and *

++ and * : same priority (right to left associativity)

```
int [4] = {10, 20, 30, 40};
int *p = &a[0];
++*p;
*p++;
pf ("%d", *p);
```

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

100 p contains the address of a[0]

p

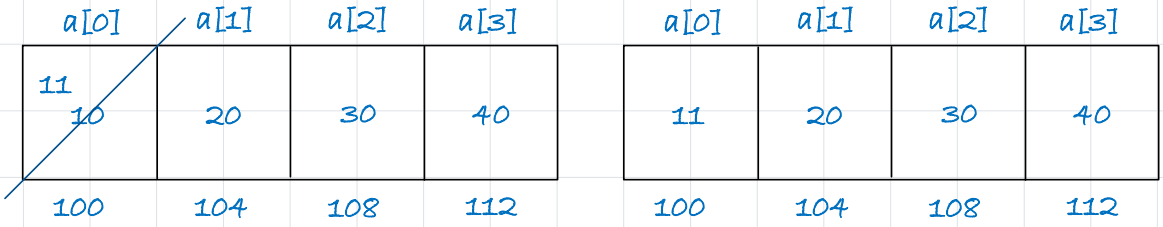
(i) ++*p

++ (*p)

++ (value at (memory location 100))

++ (10)

11



(ii) *p++

*(p++)

(i) use the value (useless)
(ii) increase the value
p = p+1
p = 100 + 1x4
p = 104

104 p contains the address of a[1]

p

*p

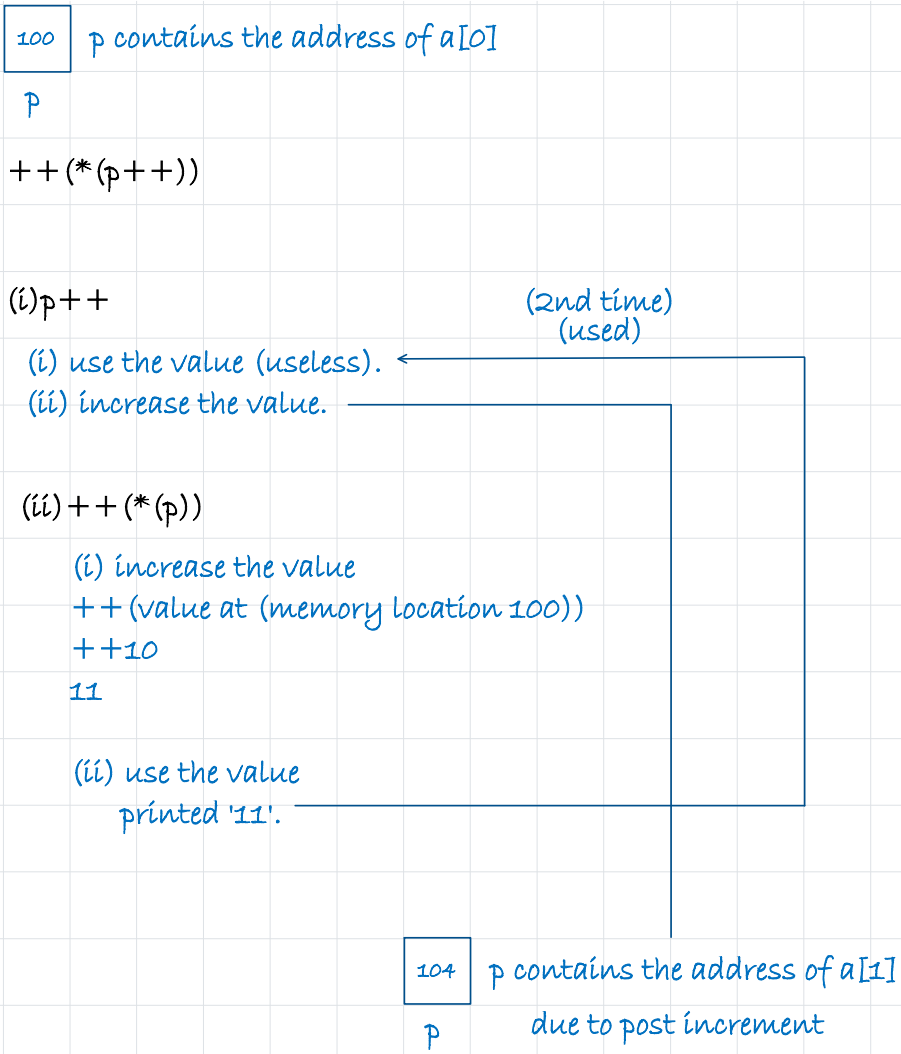
value at (memory location 104)

20

```
int [4] = {10, 20, 30, 40};
int *p = &a[0];
pf ("%d", ++*p++);
```

cannot use multiple modify operator withing a sequence point.

```
int [4] = {10, 20, 30, 40};
int *p = &a[0];
++*p++;
```



topic : complex declarations

declaration involves :

(i) () : functions
(ii) [] : arrays
(iii) * : pointers
(iv) identifier
(v) data type

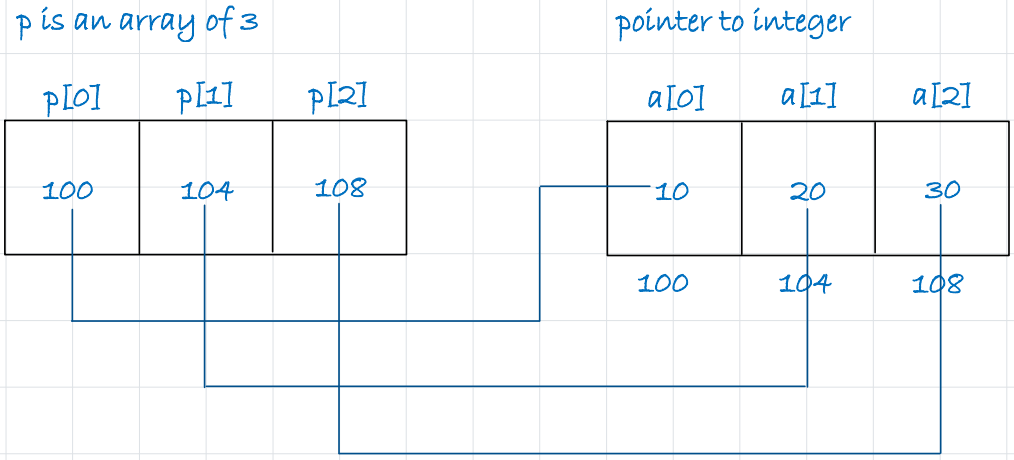
(i) (left to right)
(ii) (right to left)

declaration hamesha identifier ka hi hota hai.

(i) int *(p[3])

(a) (p[3]) : p is an array of 3

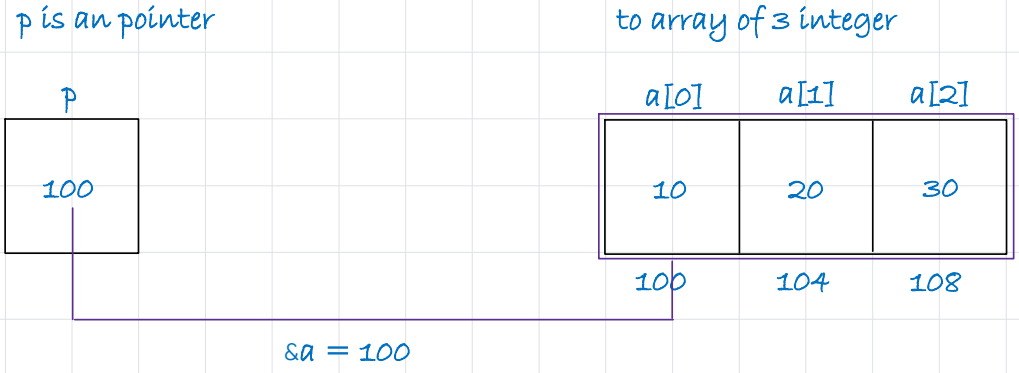
(b) int*(p[3]) : p is an array of 3 pointer to integer



(ii) int *p[3] : int (*p)[3]

(a) (p) : p is an pointer

(b) int (*p)[3] : p is an pointer to array of 3 integer
(address of an array of 3 integer)

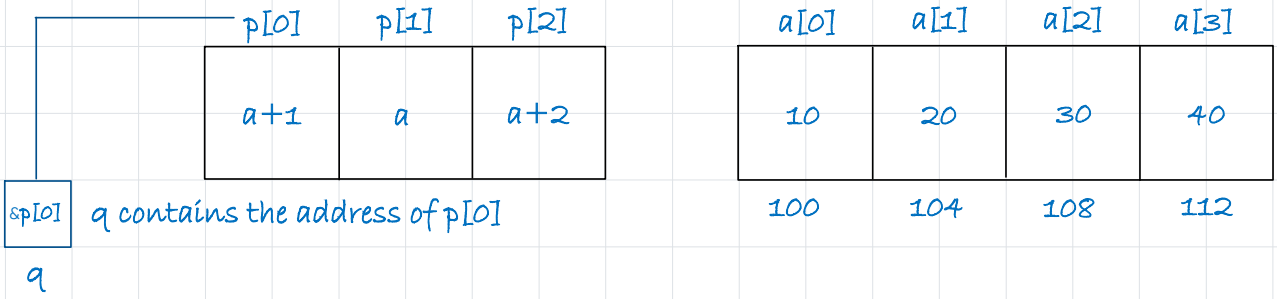


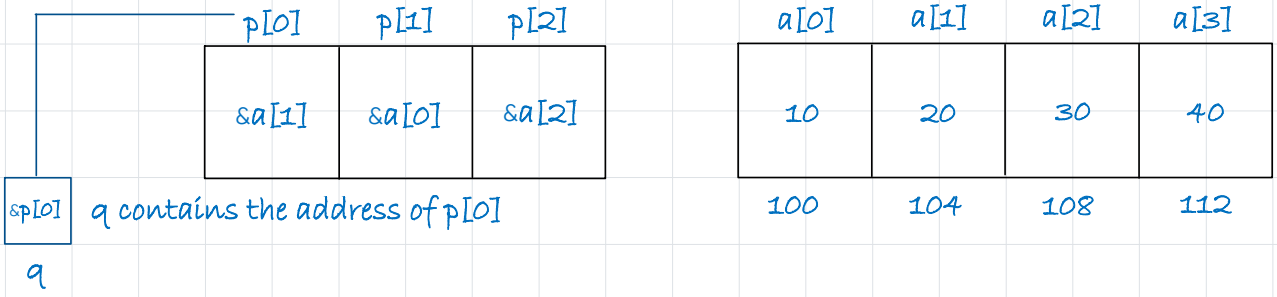
(iii) int *p
p is a pointer to integer
p stores the address of integer

(iv) char *p
p is a pointer to character
p stores the address of character

int [4] = {10, 20, 30, 40};

```
int *p = {a+1, a, a+2}
int **q;
q=p;
```





(i) ++q

(a) increase the value

q = q+1
q = &p[0]+1
q = &p[1]

&p[1] q contains the address of p[1]
q

(b) use the value

(i) *++q : *(++q)

(a) increase the value

q = q+1
q = &p[1]+1
q = &p[2]

&p[2] q contains the address of p[2]
q

(b) use the value
(no meaning, no print)

(ii) pf("%d", *++*q);

(a) *q : *&p[2]
p[2]
&a[2]
108

(b) ++*q : ++*&p[2]
++p[2]
++&a[2]
&a[2]+1
&a[3]

(c) *++*q : *++*&p[2]
*&a[3]
a[3] ——— element
40

swap function

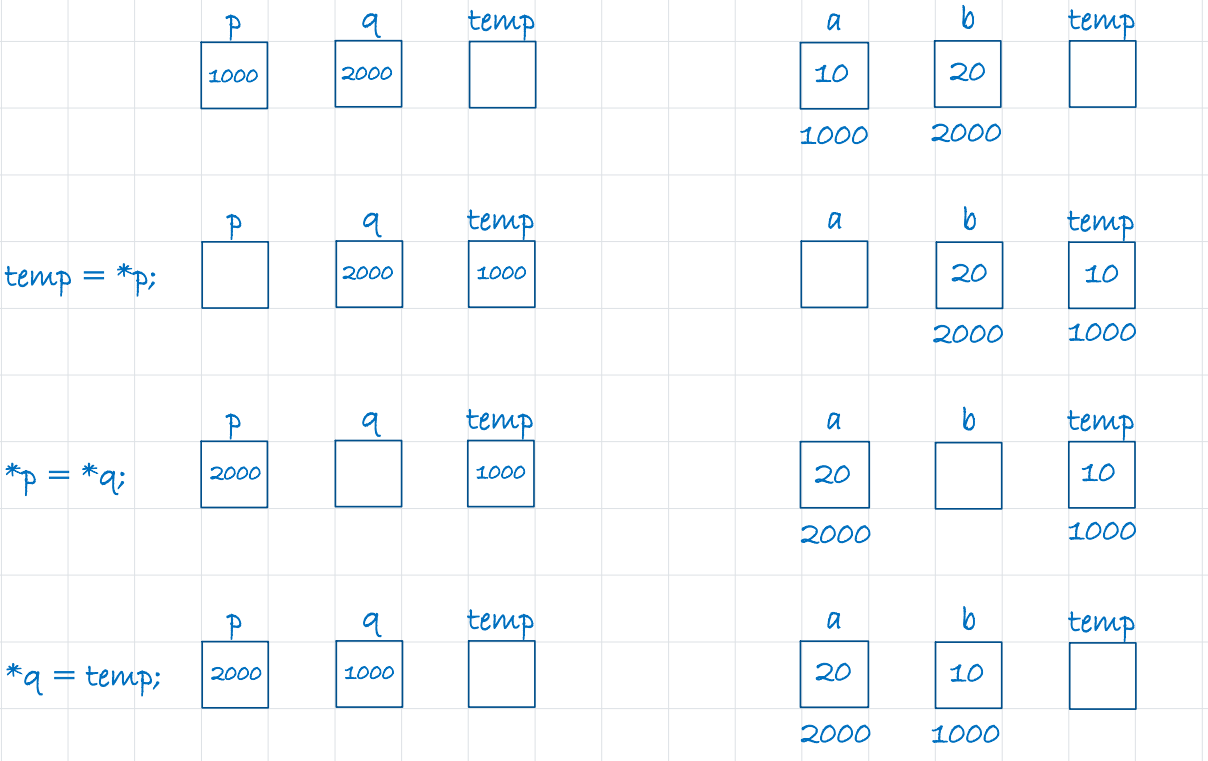
parameters passed by address (call by reference)

```
void swap (int*, int*)
void main () {
        int a = 10, b = 20;
        swap (&a &b);
        pf ("%d%d, a,b);
                }
```

```
void swap (int*p, int*q)
        {
        int temp;
        temp = *p;
        *p = *q;
        *q = temp;
                }
```

to hold the addresses
we have pointer here

we are passing addresses here



changes made in formal arguments reflected on actual arguments because we are swapping

addresses here, we worked on actual argument.

```
void fun (int*)
void main () {
        int a [4] = {10, 20,30,40};
        fun (a);
        pf ("%d%d, a[0], a[1]);
                }
```

```
void fun(int*p)
        {
          ++p;
          ++*p;
                }
```

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

&a[0] p contains the address of a[0]

p

(i) ++p;

(a) increase the value

p = p+1

p = &a[0] + 1

p = &a[1]

&a[1] p contains the address of a[1]

(b) use the value. p

(i) ++*p; ++(*p)

p = *&a[1]

p = a[1]

p = 20

(a) increase the value

++20

21

(b) use the value.

a[0]	a[1]	a[2]	a[3]
10	21 20	30	40
100	104	108	112

pf ("%d%d, a[0], a[1]); 10, 21

```
void fun (int*)
void main () {
```

```
void fun(int*p)
        {
```

```
void fun (int*)
void main () {
        int a [4] = {10, 20,30,40};
        fun (a);
        pf ("%d%d, a[0], a[1]);
                }
```

```
void fun(int*p)
        {
         *++p;
              }
```

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

&a[0] p contains the address of a[0]

p

(i) *p++; *(p++)

(a) use the value.

useless

(b) increase the value

p = p+1

p = &a[0] + 1

p = &a[1]

- *(&a[1])

*&a[1]

a[1]

20

pf ("%d%d, a[0], a[1]); 10, 20

```
void fun (int*)
void main () {
        int a [4] = {10, 20,30,40};
        fun (a);
        pf ("%d%d, a[0], a[1]);
                }
```

```
void fun(int*p)
        {
         ++ *p++;
              }
```

a[0]	a[1]	a[2]	a[3]
10	20	30	40

&a[0] p contains the address of a[0]

p

++ *p++; ++(*(p++))

(i) p++

(a) use the value (2nd time) (used)

useless

(ii) increase the value.

(ii) ++(*p)

++(*&a[0])

++(a[0])

++(10)

11

104 p contains the address of a[1]

p due to post increment

```
void fun (int*)
void main () {
        int a [4] = {10, 20,30,40};
        sum (a, 4);
        pf ("%d%d, a[0], a[1]);
                                }
```

you have to pass the number of elements also, sum till 4.

```
void sum(int*p, int n)
        {
         int s = 0
         for (i=0; i<n; i++);
          {
           s = s + p[i];
           printf ("%d", sum);
                                }
```

p[i] = *(p+i)

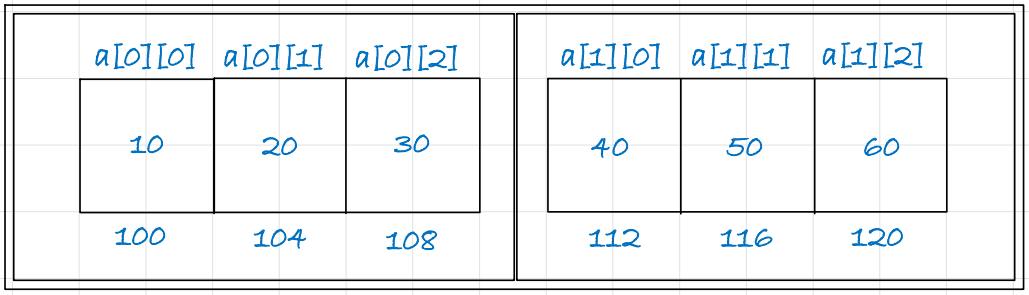
jab bhi aap ek array pass kar rahe hai jisme no. of element of array par kaam ho raha hai toh batana padega.

```
void fun (int*)
void main () {
        int a [2][3] = {10, 20,30,40, 50, 60};
```

```
void fun(int*p)
        {
          ++p;
```

```
void main () {
        int a [2][3] = {10, 20,30,40, 50, 60};
        fun (a[0]);
        pf ("%d%d, a[0][0], a[0][1], a[0][2]);
                                                }
```

```
{
 ++p;
 *p++;
 *++p;
 p - - ;
 *p = 100;
          }
```



a[0] a[1]

a

100 p contains the address of a[0]

p

(i) ++p

(a) increase the value

p = p + 1

p = &a[0][0] + 1

p = &a[0][1]

104 p contains the address of a[0][1]

p

(b) use the value

useless

(ii) *p++

*(p++)

(a) use the value

useless

(b) increase the value

p = p+1

p = &a[0][1] + 1

p = &a[0][2]

108 p contains the address of a[0][2]

p

- *p

*&a[0][2]

24

(iii) *++p

*(++p)

(a) increase the value

p = p+1

p = &a[0][2] + 1

p = &a[1][0]

p contains the address of a[1][0]

(b) use the value

useless

- *p

*&a[1][0]

40

(iv) p - -

p - -

(a) use the value

useless

(b) decrease the value

p = p-1

p = &a[1][0] - 1 (112-4)

p = &a[0][3] (108)

p contains the address of a[0][3]

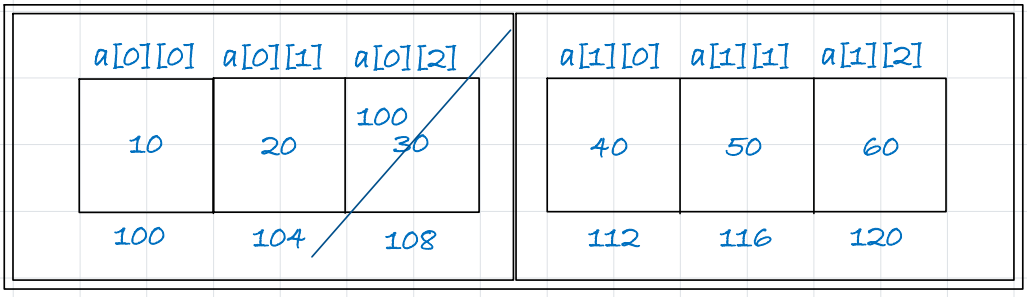
- p

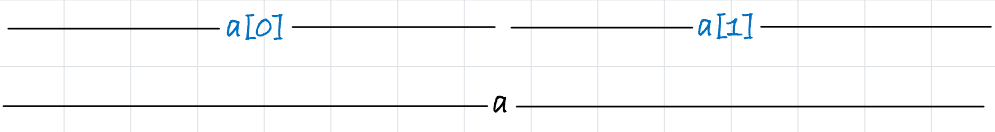
&a[0][3]

(v) *p = 100

*&a[0][3] = 100

(108) = 100





pf ("%d%d, a[0][0], a[0][1], a[0][2]); 10, 20, 100

int *p;

(4 byte)

p++ : 4 byte se increment

p++ : 4 byte se increment

char*q; [ ] (1 byte)

q++ : 1 byte se increment

concept : typecasting

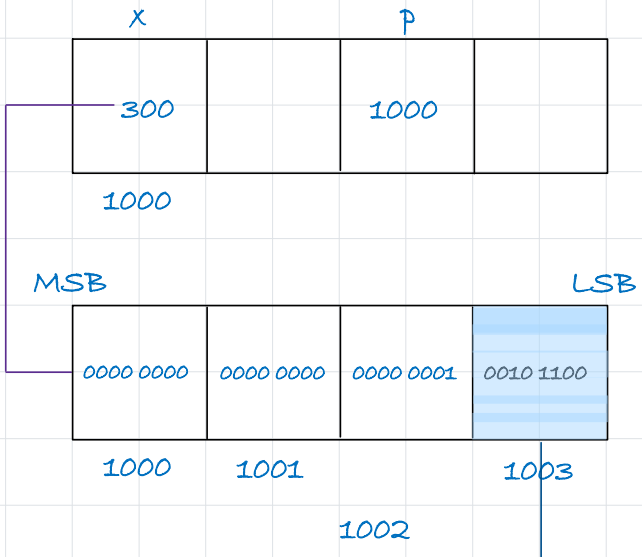
typecasting : process of converting one data type to another data type by the programmer using the casting operator during program design.

```
int a = 300;
char *p;
p = (char*)&a;
```

typecasting (maan le character ka address hai)

```
printf ("%d", *p);
```

44



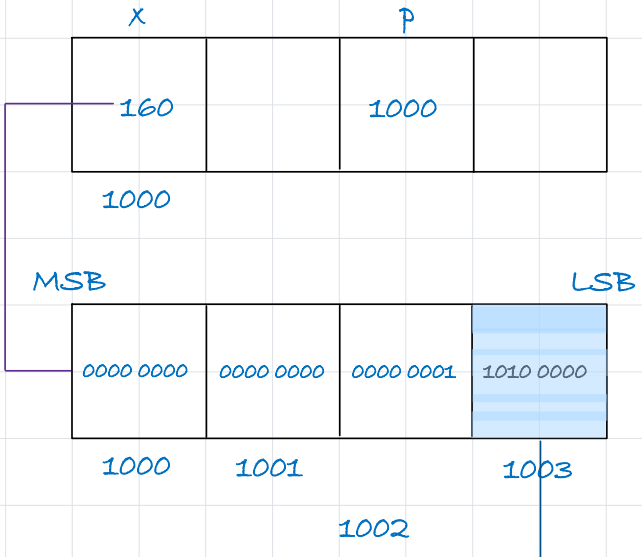
we are using char so only one byte will be dereferenced

```
int a = 160;
char *p;
p = (char*)&a;
```

typecasting (maan le character ka address hai)

```
printf ("%d", *p);
```

-96



we are using char so only one byte will be dereferenced

128+32=160
in cyclic property, 160 is not available
the range is -128 to 127

so, the value will be in negative and we will focus on 0's

-2^6 -2^5 -2^4 -2^3 -2^2 -2^{-1} -2^0 1
1 0 1 0 0 0 0 0

-2^5 -2^3 -2^2 -2^{-1} -2^0 1
-64-16-8-4-2-1
-96

concept : function pointer (pointer to function)

int (*p) ()

p is a pointer to fucntion that takes no argument and it returns a integer value.

int add (int, int)

return type : int

argument list

- int *p(int, int)

p is a pointer to function that takes two integer argument and it returns an integer value.

- int *p(char*)

p is a pointer to function that takes pointer to character as an argument and it returns an integer value.

- int *p(int*)

p is a pointer to function that takes pointer to integer as an argument and it returns an integer value.

- can we pass a value to a function?
yes.

```
int a = 10, b = 20, result;
result = add(a,b);
```

- can we pass address to a function?

- can we pass address to a function?

yes.

```
int a [4] = {10, 20, 30, 40};
fun (a);
```

advantage : we can pass one function as an argument to an another function.

declaration of pointer :

```
int a = 10;
int *p;
```

```
float b = 9.8;
float *ptr;
```

how to declare a pointer to function?

use function pointer.

example : int add (int, int);

int (*p) (int, int); p is a pointer to function that takes two integers as argument as returns its value.

```
#include <stdio.h>
int add (int, int);
int main () {
        p = &add;
        printf ("%d", (*p) (10,20);
        return 0;
        }
int add (int x, int y)
        {
         return x + y;
        }
```

forward declaration

calling the function using pointer to function

body of the function

note : it is important to write the definition of function

subtraction

```
#include <stdio.h>
int sub (int, int);
int main () {
                p = &sub;
                printf ("%d", (*p) (10,20);
                return 0;
                        }
int add (int x, int y)
        {
          return x - y;
                        }
```

product

```
#include <stdio.h>
int add (int, int);
int main () {
                p = &prod;
                printf ("%d", (*p) (10,20);
                return 0;
                        }
int add (int x, int y)
        {
          return x * y;
                        }
```

all are valid

(i) p = &add : (*p) (10,20);

(ii) p = add : (*p) (10,20);

(iii) p = add : (p) (10,20)

(iv) p = &add : (p) (10,20)

```
int [4] = {10, 20, 30, 40};
int *p = {a+3, a+2, a+1,a}
int y;
y = - - p[0] - p[1];
pf("%d",y); 0
pf("%d",*p[0]); 30
```

p[0]	p[1]	p[2]	p[3]
&a[3]	&a[2]	&a[1]	&a[0]

a[0]	a[1]	a[2]	a[3]
10	20	30	40
100	104	108	112

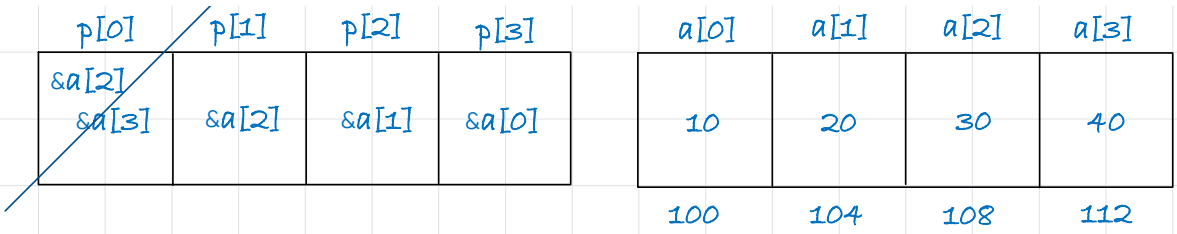
y = - - p[0] - p[1];

(i) - - p[0]

p[0] = p[0] - 1
p[0] = &a[3]-1
p[0] =&a[2]

p[0]	p[1]	p[2]	p[3]
&a[2]			

a[0]	a[1]	a[2]	a[3]



(ii) p[0]-p[1]
&a[2] - &a[2]

$$p\text{-}q = \frac{\text{(actual difference)}}{\text{integer size}}$$

$$\frac{108 - 108}{4} = \frac{0}{4} = 0$$

no. of elements between these 2 address are 0

q.1. which of the following are invalid?

(a) int b[][4]; invalid, because of no initialization

(b) int b[]; invalid, because of no initialization

(c) int b[2][][3]= {1, 2, 3, 4}; invalid, because you cannot omit second array

(d) int b[][2][3]= {1, 2, 3, 4}; — valid

(1) only declaration (without initialization)
it is mandatory/compulsory to provide size of each dimension.

(2) initialization
there is flexibility only for first dimension, you can omit size of first dimension but this flexibility is only valid for first dimension.

q.2. which of the following are valid?

p : int arr [4] = {10, 20, 30, 40};
printf ("%d", *arr++);

array_name is the constant address of first element, we cannot use modify operators on it

q : int arr [4] = {10, 20, 30, 40};
int *ptr = arr;
printf ("%d", *ptr++);

we can use modify operators on pointer hence, it is valid.

q.3. which of the following are valid?

p : int a[5] = {10, 20, 30, 40, 50};
printf ("%d", 4[a]);

it is valid because 4[a] = a[4]

q : int 5[a] = {10, 20, 30, 40, 50};
printf ("%d", a[2]);

invalid, because declaration must be like a[5] not 5[a]

q.4. what will be the output?

```
int a[5] = {10, 15, 20, 25, 30}
printf ("%u", *(a+2)+6); 26
printf ("%u", *(a+*(a+1)-12); 25
```

(a) *(a+2)+6

*a[2] + 6

20 + 6

26

(b) *(a+*(a+1) - 12)

*(a + *(a[1]) - 12)

*(a + 15 - 12)

*(a + 3)

*(&a[0] + 3)

*(&a[3])

25

q.5. what will be the output?

```
int a[5] = {5, 3, 1, 2, 4};
int *p[5] = {a, a+1, a+3, a+2, a+4}; 2
printf ("%u%u", p[3][1], *(*(p+4)-2); 1
```

int *p[5] = {&a[0], &a[1], &a[3], &a[2], &a[4]};

p [3][1]

*(p[3] + 1)

*(&a[2]+ 1)

*(&a[3])

*&a[3]

((p+4)-2)

((p[4]) - 2)

((&a[4]) - 2)

(&a[4]- 2)

*(a[4] - 2)

a[3] *(a[2])

2 1

a[i][j] = *(a[i]+j)

q.6. what will be the output?

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

int *p[5] = {a+3, a+1, a, a+2, a+4};
int *p[5] = {&a[3], &a[1], &a[0], &a[2], &a[4]} ;

ptr = p+3
ptr = &p[0]+3
ptr = &p[3]
```

(i) ptr-p
&p[3] - &p[0]
3

(ii) *ptr-a
&p[3] - &a[0]
&a[2] - &a[0]
2

(iii) **ptr
**&p[3]
*a[2]
*&a[2]
a[2]
1

q.7. what will be the output?

```
void func (int (*ptr)[2])
        {
          **ptr+=1;
          ptr++;
          **ptr x =3;
                    }
void main ()
          {
           int arr [2][2] = {0, 1, 2, 3};
           func (arr);------- &arr[0]
           printf ("%d%d" a[0][0], a[0][1]);
                                          }
```

**ptr+=1; ptr++; **ptr x =3:

**ptr+=1;

arr[0][0] = a[0][0]+1

arr[0][0] = a[0][1]

&arr[0]

ptr

ptr++;

ptr = ptr + 1

ptr = &arr[0] + 1

ptr = &arr[1]

&arr[1]

ptr

**ptr x =3;

**ptr = **ptr x 3

&arr[1][0] = &arr[1][0] x 3

arr[1][0] = 2 x 3

6

q.8. correct or not?

```
int a[3][2] = {1, 3, 5, 7, 9, 11};
int *ptr = a;
```

incorrect because declaration says that pointer will have the address of integer but here we are assigning the address of whole array to pointer.

q.9. what is output?

```
int a[3][2] = {1, 3, 5, 7, 9, 11};
int *ptr = a[1];
++*ptr++;
printf("%d", *ptr);
```

++*ptr++;

(i) ptr++
useless

(ii) ++ (*ptr)
++(*&a[1])
++(a[1])
++(a[1][0])
++5
6

(iii) using post increment.
++*ptr++;
6++;
7

q.10. what is output?

```
int a=5, b=10, c=15;
int *p[3] = {&a, &b, &c};
printf("%d", *p[*p[1]-8];
```

- *p[*p[1]-8]
  p[&b-8]
  *p[b-8]
  *p[10-8]
  *p[2]
  *&c
  15

q.11. what is output?

```
int a[] = {10, 20, 30, 40, 50};
int *p[] = {a, a+3, a+4, a+1, a+2};
int **ptr = p;
ptr++;
printf("%d%d", ptr-p, **ptr);
```

int **ptr = p;

ptr

ptr++;

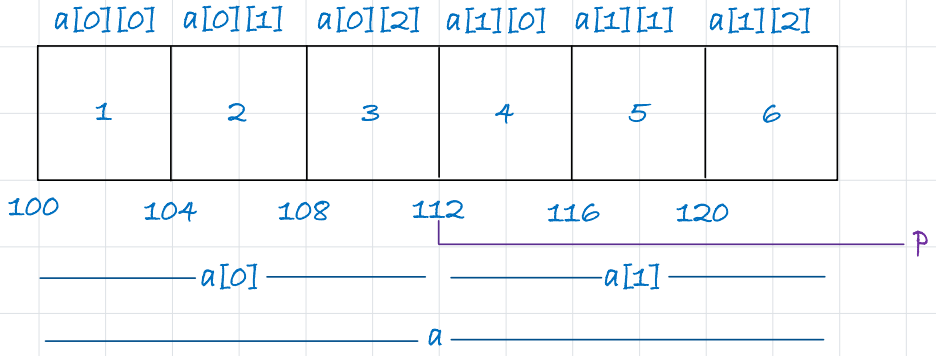
ptr

int *p[] = {&a[0], &a[3], &a[4], &a[1], &a[2]};

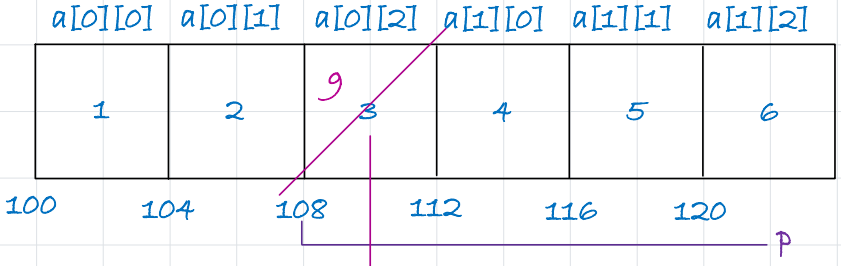
ptr-p	**ptr
&p[1] - &p[0]	**&p[1]
1	*a[3]
	*&a[3]
	a[3]
	40

q.12. what is output?

```
void f(int*);
void main () {
        int a[2][2] = {1, 2, 3, 4, 5, 6};
        f(a[1]);          // &a[1][0]
        pf ("%d%d", a[1][1], a[1][2]);
                                        }
void f(int*p) {
        p--;
        *p = *p x *p;
        p--;
        *p = *p x *p;
                        }
```

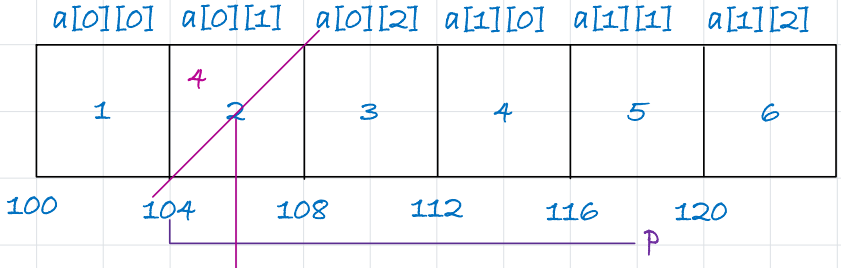


(a) p--;
(i) p = p - 1
p = &a[1][0] - 1
p = 112 - 1x4
p = 108
p = &a[0][2]



(b) *p = *p x *p;
*p = 3 x 3;
*p = 9;

(c) p--;
(i) p = p - 1
p = &a[0][2] - 1
p = 104 - 1x4
p = 104
p = &a[0][1]



(d) *p = *p x *p;
*p = 2 x 2;
*p = 4;

a[1][1]: 5
a[1][2]: 6

q.13. what is output?

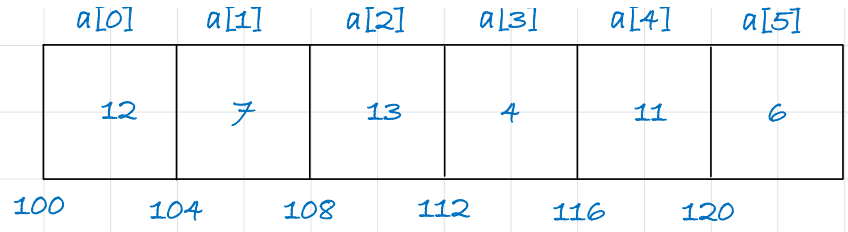
```
int a[4][5] = { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16,17,18,19,20};
pf ("%d", *(*(a+**a+2)+3));
```

- *(*(a+**a+2)+3)
((&a[0]+**&a[0]+2)+3)
((&a[0]+*a[0]+2)+3)
((&a[0]+*&a[0][0]+2)+3)
((&a[0]+a[0][0]+2)+3)
((&a[0]+1+2)+3)
((&a[0]+3)+3)
((&a[3])+3)
((&a[3])+3)
(&a[3])+3
*(a[3])+3
*(&a[3][0])+3
*(&a[3][0])+3
*&a[3][0]+3
a[3][0]+3
16+3
19

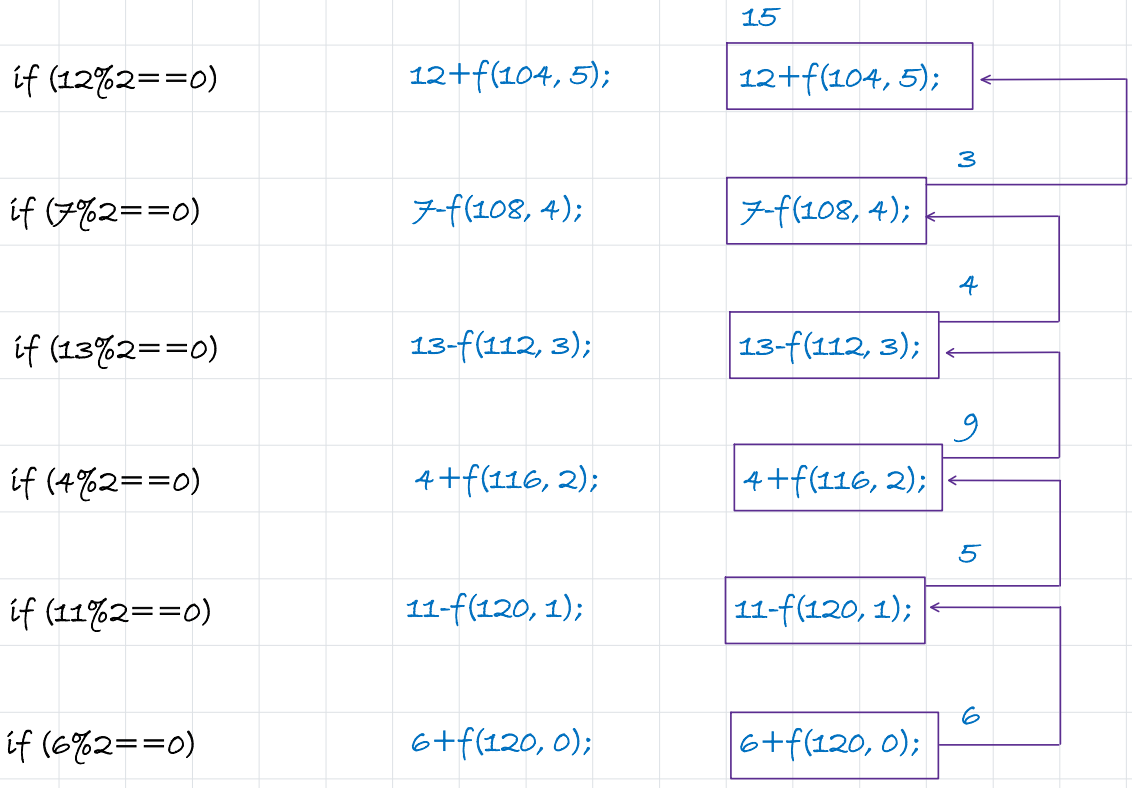
q.14. what is output?

```
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);
                                                }
void main () {
                int a [] = {12, 7, 13, 4, 11, 6};
                pf("%d", f(a,6);
                                }
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]




```
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);
```




```
char a = 60;
int *p;
p = int*&a;
pf ("%d",*p);
```

it is character hence 60 is stored in one byte
but we are trying to fetch integer which means 4 byte from memory but the rest of values are unknown (garbage values).

note : pointer and data must be of same type.

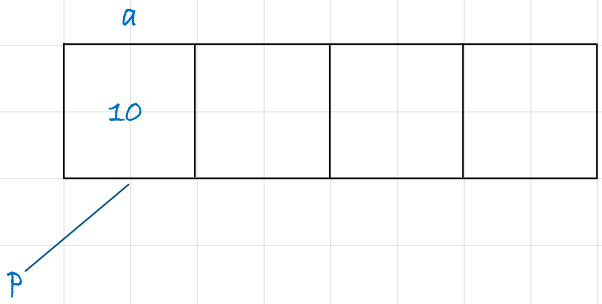
topic : void pointer

void *p : there will be a address in p

int *p : there will be a integer address in p

char *p : there will be a character address in p

```
void *p;
int a = 10;
float b = 4.32;
p = &a;
pf ("%d", *p);
```



compiler will shout here because compiler does not know how much byte it have to pick up, it won't be able to dereference.

we should not dereference any void pointer.

```
pf ("%d", *(int*)p);
```

compiler will not shout here because compiler knows how much byte it have to pick up, it will be able to dereference.

so, typecasting is the solution to dereference void pointer.

points to remember -
(i) we can not dereference a void pointer directly.
(ii) first typecast then only dereference.

concept : arithmatic operators on void pointer.

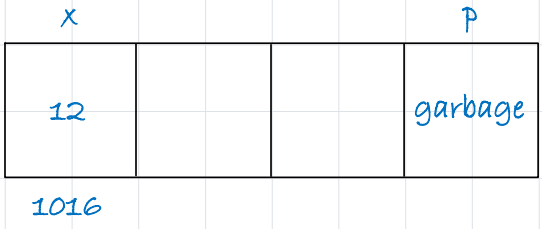
do not perform any arithmatic operation on void pointers.

int *p;	char *p;	void *p;
—	—	—
—	—	—
—	—	—
—	—	—
p = p+2	p = p+2	p = p+2
2 x size of (int)	2 x size of (char)	2 x size of (?)

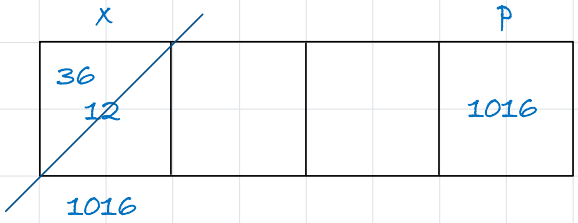
topic : wild pointer

wild pointer is an uninitialized pointer.

```
void main ()
        {
         int *p;
         int x = 12;
         *p = 36;
                }
```

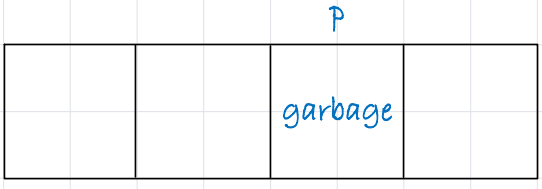


but there is a possibility of 1016 being there in p



so here unintenionally we changed the value of our program.

```
void main ()
        {
         int *p;
           }
```



p ke andar bhi ek garbage value hogi that is being treated as address.

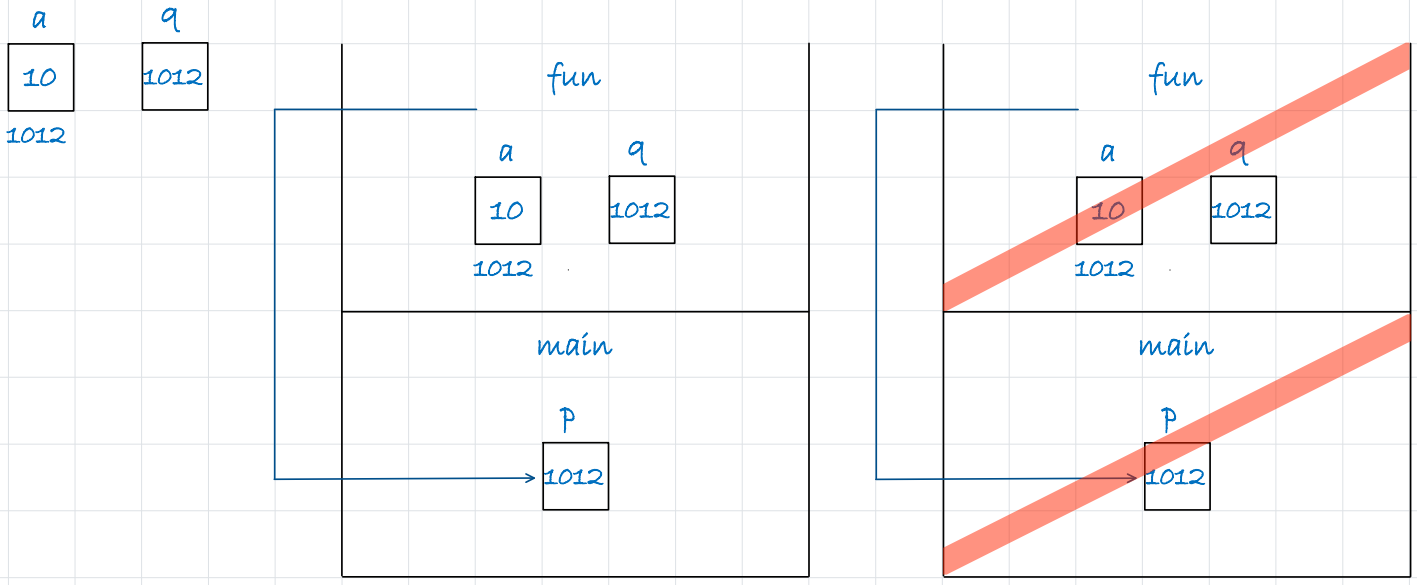
topic : dangling pointer

pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in

C programs.

```
int *fun ()
{
  int a = 10;
  int *q = &a;
  return q;
}
```

```
void main ()
{
  int *p;
  p = fun();
  *p;
}
```



once the value is returned, activation record clears up

p is pointing to fun, but activation record is cleared up and hence p is not pointing to any valid address.

note : local variable ka address return nahi karna chaiye because return karne se activation record clear ho jata hai but if we use static then it remains throughout program.

```
int *fun ()
{
  static int a = 10;
  int *q = &a;
  return &q;
}
```

```
void main ()
{
  int *p;
  p = fun();
  pf ("%d",*p);
}
```

does not save in activation record

topic : null pointer

the null pointer is the pointer that does not point to any location but NULL.
- to differentiate valid pointer/address from invalid.

why 0 is treated as null?

because, operating system guarantees that few first bytes of address in the memory will not be allocated to any program.

int *p = (int*)0;

here we are initialising pointer with null, esi location jo exist nahi karti.

to check if pointer is null :

```
ptr == NULL;
```

Uses of NULL Pointer in C :

(i) to initialize a pointer variable when that pointer variable hasn't been assigned any valid memory address yet.

(ii) to check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer-related code, e.g., dereference a pointer variable only if it's not NULL.

(iii) to pass a null pointer to a function argument when we don't want to pass any valid memory address.

(iv) a NULL pointer is used in data structures like trees, linked lists, etc. to indicate the end.

dynamic memory allocation

types :

(i) malloc ()
(ii) calloc ()
(iii) re-alloc ()
(iv) free ()

(i) malloc () : memory allocation

used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't initialize memory at execution time so that it has initialized each block with the default garbage value initially.

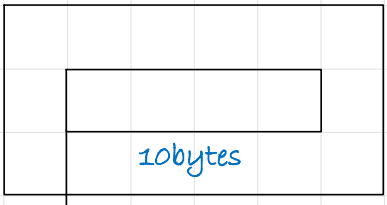
- cheaper and not reliable.
- garbage values.

Syntax of malloc() in C :

```
(void*)malloc (unsigned int);
```

```
(void*) malloc (10);
```

10 byte ka block allocate



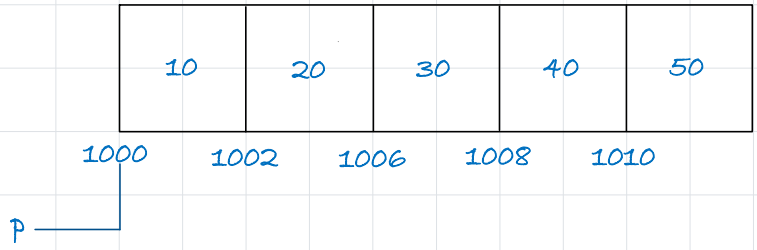
malloc address return karta hai aur us address ko store karne ke lie we need pointer.

example :

assume we are storing 5 integers in the system where integer are of 2 bytes.

total bytes required : 5 x 2 = 10 (bytes allocated)

```
int*p = malloc (10);
scanf ("%d", p+0); 10
scanf ("%d", p+1); 20
scanf ("%d", p+2); 30
scanf ("%d", p+3); 40
scanf ("%d", p+4); 50
```



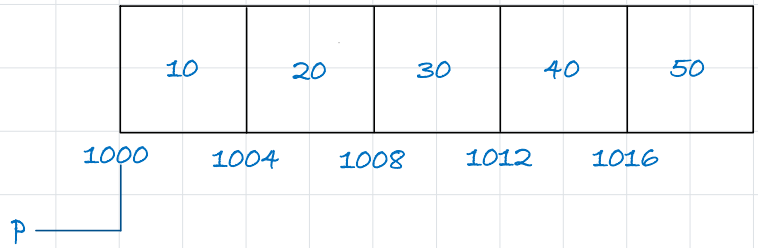
malloc return karega address 1000

if i run the same program on a system where the integer are of 4 byte then the behaviour is undefined because in this program we are storing 5 integer in 10 bytes while in case of 4 bytes/integer we need 20 bytes to store 5 integer.

the solution is; instead of allocating integer bytes we will write "[size of (int)]" so now it becomes compiler or system dependent.

int*p = malloc (5 x size of (int))

```
int*p = malloc (10);
scanf ("%d", p+0); 10
scanf ("%d", p+1); 20
scanf ("%d", p+2); 30
scanf ("%d", p+3); 40
scanf ("%d", p+4); 50
```



malloc return karega address 1000

in this, a piece of code is repeating

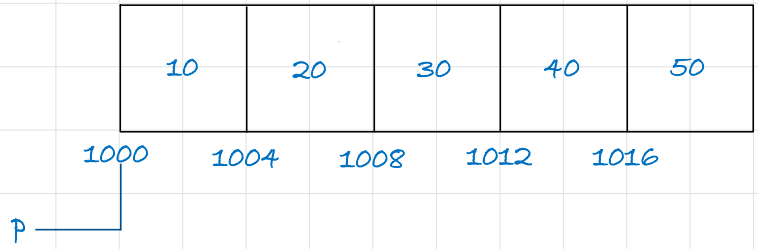
```
int*p = malloc (10);
scanf ("%d", p+0); 10
scanf ("%d", p+1); 20
scanf ("%d", p+2); 30
scanf ("%d", p+3); 40
scanf ("%d", p+4); 50
```

yahan hum for loop ka use kar sakte execute karane ke lie

```
for (i=0; i<5; i++)
    scanf ("%d" p+i);
```

5 values read karega

```
int*p = malloc (10);
for (i=0; i<5; i++)
    scanf ("%d" p+i);
```



p[0] : (memory location 1000)
*(p[0]) = value at (memory location 1000)

```
printf ("%d", *(p+0)); 10
printf ("%d", *(p+1)); 20
printf ("%d", *(p+2)); 30
```

```
printf ("%d", *(p+3)); 40
printf ("%d", *(p+4)); 50
```

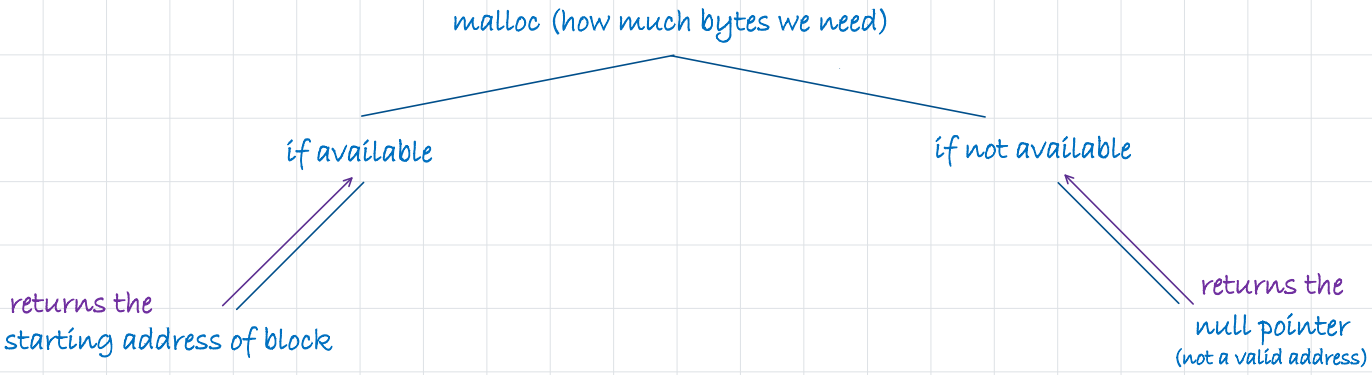
in this, a piece of code is repeating

```
printf ("%d", *(p+0)); 10
printf ("%d", *(p+1)); 20
printf ("%d", *(p+2)); 30
printf ("%d", *(p+3)); 40
printf ("%d", *(p+4)); 50
```

yahan hum for loop ka use kar sakte print karane ke lie

```
for (i=0; i<5; i++)
    printf ("%d" *(p+i));
```

5 values print karega




```
int n, i;
int *p;
printf ("enter the number of elements");
scanf ("%d", &n);
p = malloc (size of (int) x n);
if (p!=null)
                {
                 for (i=0; i<n; i++)
                      scanf ("%d", p+i);
                 for (i=0; i<n; i++)
                      scanf ("%d", p[i]);
                                        }
```

if p is not equal to null then only the code will run means p is not pointing to any invalid address. (block is available)

(ii) calloc() : contiguous allocation

"calloc" or "contiguous allocation" method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.

it is very much similar to malloc() but has two different points and these are:
(i) it initializes each block with a default value '0'.
(ii) it has two parameters or arguments as compare to malloc().
- expensive and more reliable
- initialised with "0" (garbage value se jada known values acchi hoti hai)

Syntax of calloc() in C

```
calloc (no. of block, size of each block)
```

(iii) re-alloc() : re-allocation

"realloc" or "re-allocation" method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc through some pointer is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.
- grow aur shrink kar sakte hai hum isse.e

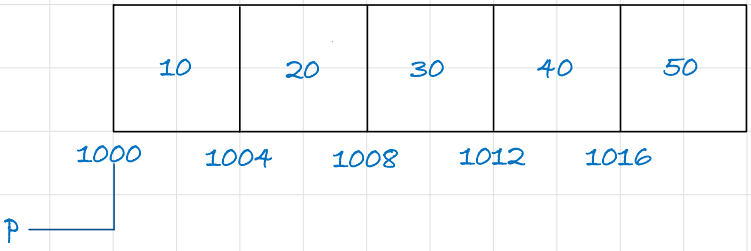
Syntax of calloc() in C

```
ptr = realloc (ptr, newsize);
```

example : (grow)

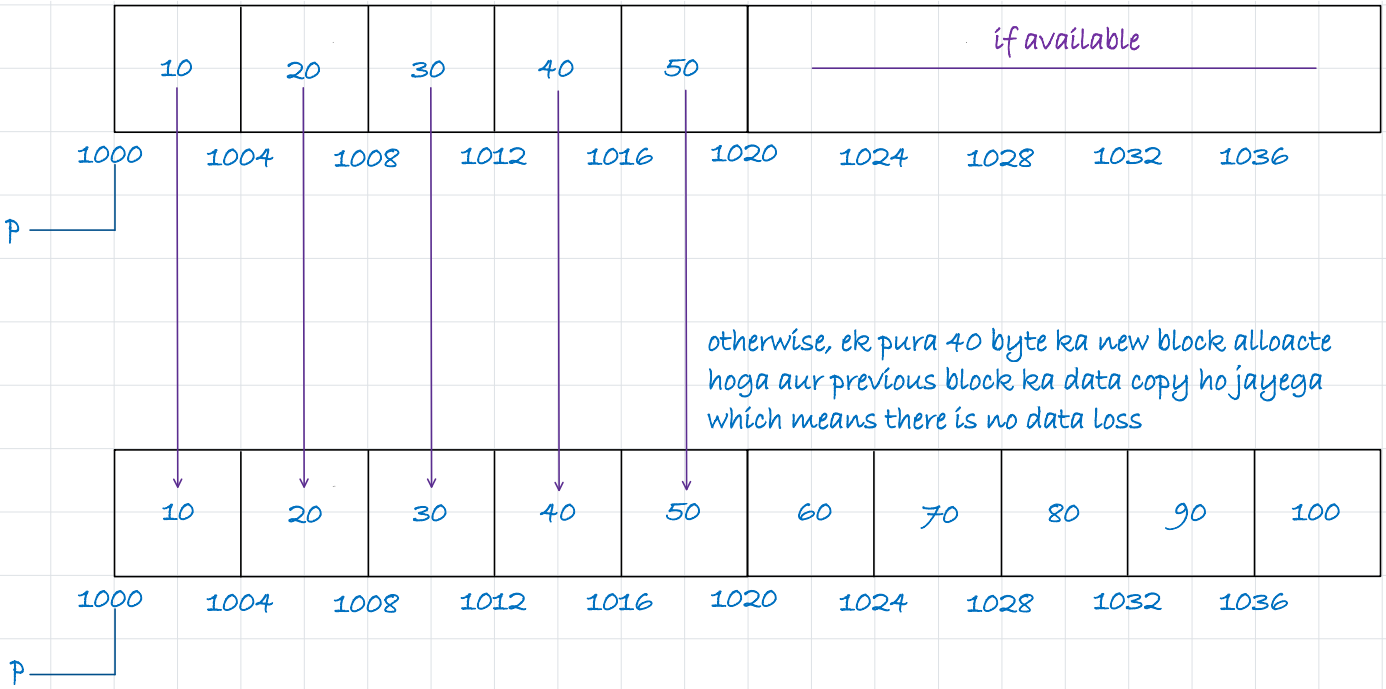
```
int *p = malloc (5 x size of (int))
for {
    ___
    ___
      }

p = realloc (p, 10 x size of (int))
```

int *p = malloc (5 x 4) = 20 byte ka block mil gaya



now, i want to store 10 elements instead of 5

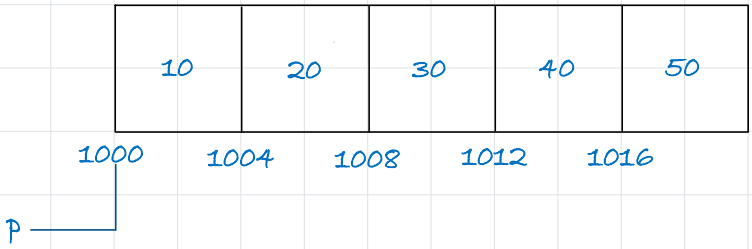
p = realloc (p, 10 x 4) = 40 byte ka block chaiye toh agar memory available hai continiously toh vo hume allocate ho jayegi



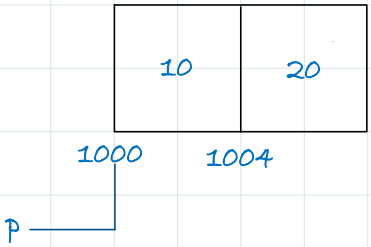
example : (shrink)

```
int *p = malloc (5 x size of (int))
for {
    ___
    ___
      }

p = realloc (p, 2 x size of (int))
```

int *p = malloc (5 x 4) = 20 byte ka block mil gaya



int *p = malloc (2 x 4) = 8 byte ka block ho gaya



it is not a data loss because we delibaterly decided to shrink the memory

- we do not need the elements.

it is the programmer's responsibility to allocate the memory, that means to de-allocate the memory after the work has done is also the responsibility of the programmer.

concept : de-allocation

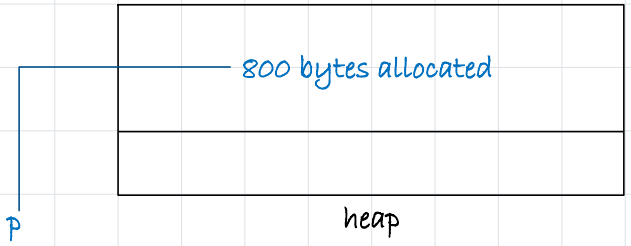
why we need to de-allocate?

```
void fun ( )
        {
          int *p = malloc (200 x size of (int))
          for {
              ___
              ___
              return;
                  }

void main ( )
          {
            fun ();
            fun ();
            fun ();
            fun ();
            fun ();
                }
```

(i) fun () called for first time

how much bytes operation system allocated : 800 bytes

int *p = malloc (200 x size of (int))

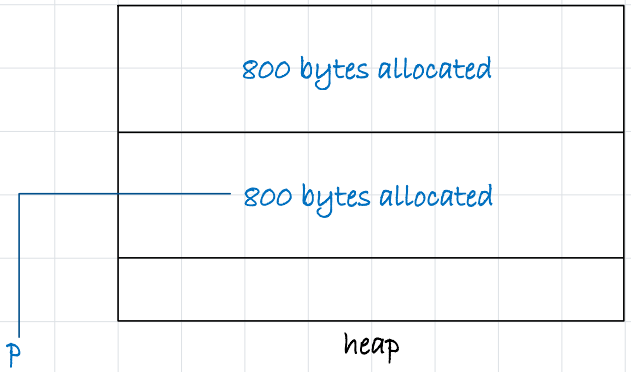


p is pointing to first section

(ii) fun () called for second time

how much bytes operation system allocated : 1600 bytes

int *p = malloc (200 x size of (int))



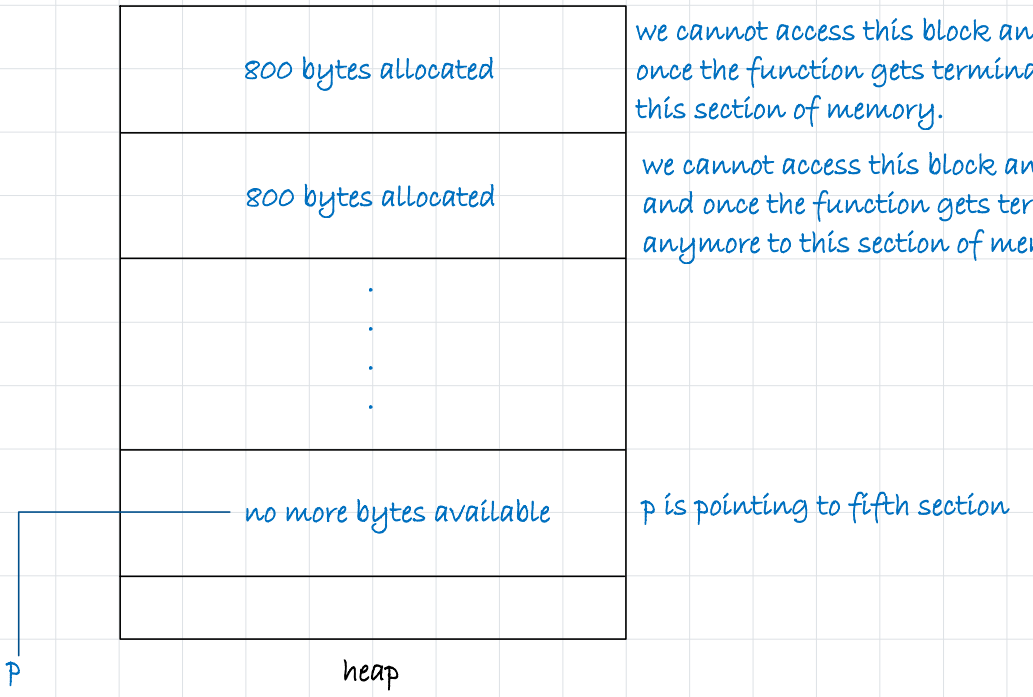
we cannot access this block anymore as p is a local variable and once the function gets terminated p is not pointing anymore to this section of memory.

p is pointing to second section

(iii) fun () called for fifth time

int *p = malloc (200 x size of (int))

how much bytes operation system allocated : 3200 bytes



we cannot access this block anymore as p is a local variable and once the function gets terminated p is not pointing anymore to this section of memory.

we cannot access this block anymore as p is a local variable and once the function gets terminated p is not pointing anymore to this section of memory.

p is pointing to fifth section

no more bytes available in the memory because we used 3200 bytes according to operating system, meanwhile we cannot access any of those blocks. this concept is called as memory leakage means we are not using the memory and we have not de-allocated it yet. here comes the concept of free ()

concept : free()

"free" method in C is used to dynamically de-allocate the memory. the memory allocated using functions malloc() and calloc() is not de-allocated on their own. hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free () in C

```
free (ptr);
```

```
void fun ( )
        {
          int *p = malloc (200 x size of (int))
          for {
              ___
              ___
              free (p);
              return;
                  }

void main ( )
```

```
{
 fun ();
 fun ();
 fun ();
 fun ();
 fun ();
        }
```

strings

sequence of characters terminated by null character ('\0')
string is stored as an array of characters.

the difference between a character array and a string is that the string in C is terminated with a unique character '\0'.

ASCII code of null : 0
format specifier for string : %s

double quotes ke andar by default string hoti hai.

"aryan"

a	r	y	a	n	\0

null will be added by the compiler

string is stored as an array of characters :

char name [10] = "aryan" (char : 1 byte)

a	r	y	a	n	\0				
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009

name

printf ("%s", name); aryan

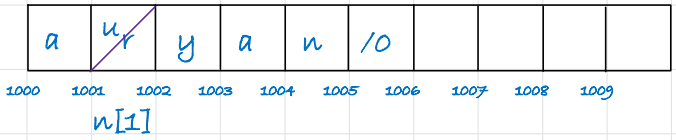
(i) we can individually change the elements (content) of an array.
(read + write)

char name [10] = "aryan" (char : 1 byte)

a	r	y	a	n	\0				
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009

name

name [1] = 'u';



(ii) flexibility to omit the first dimension of an array if initialised.

```
char name [ ] = "aryan"
```

compiler will count;

```
char name [6] = "aryan"
```

(iii) behaviour is undefined if you provide same count in initialization as string characters.

```
char name [5] = "aryan"
```

so, always provide the size of an array one more then the string size.

(iv) if we are providing everything explicitly by own then we also have to provide null.

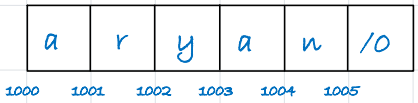
```
char name [ ] = {'a', 'r', 'y', 'a', 'n'};
printf("%s", name); aryan#$%?
```

provide null also

```
char name [ ] = {'a', 'r', 'y', 'a', 'n', '/0'};
printf("%s", name); aryan
```

(v) the address of string is passed to printf, printf checks the content and prints the symbols till null.

```
#include <stdio.h>
void main () {
                char name [ ] = "aryan";
                printf("%s", name);
                                }
```



name : &n[0] : 1000
&n[1] : 1001

&n[2] : 1002
&n[3] : 1003
&n[4] : 1004
&n[5] : 1005

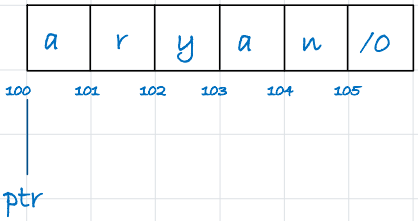
printf("%s", name+2); &n[0]+2 : &n[2] : 1002 : yan

another way to store string :

through pointer to character.

```
#include <stdio.h>
void main () {
                char *ptr = "aryan";
                printf("%s", ptr);
                                }
```

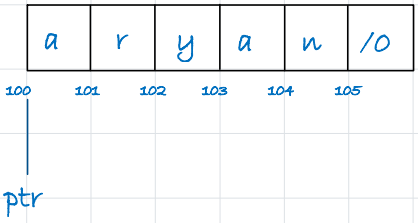
(i) string is the address of its first character.



char *ptr = "aryan";

iska address ptr ke andar jaa raha hai

(ii) we cannot individually change the elements (content) in pointer.
(read only area)

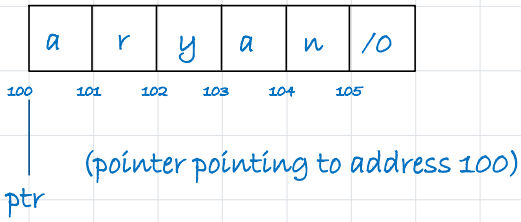


ptr [1] = 'u';

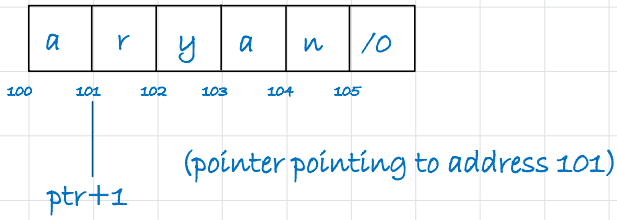
we cannot change the content individually

(iii) modify operators are valid in the case of pointers as it just points to next or previous address.

char *ptr = "aryan";



++ptr : ptr = ptr + 1



format specifier (%s) is optional

```
char name [10] = "aryan"
printf ("%s", name); aryan
printf ("%s",&n[0]); aryan
printf (name); aryan
printf (&name[0]); aryan
```

all four will have the same output

the difference between character array and pointer:

character array	pointer
(i) char name [] = "aryan"; name = "iram";	(i) char *ptr = "aryan"; ptr = "iram";
this is invalid because we cannot assign a complete new string to an array. it can only change individual	char *ptr = "aryan"; a r y a n /0

assign a complete new string to an array. it can only change individual element.

(ii) char name [10] = "aryan";
printf ("%s", name);a

address

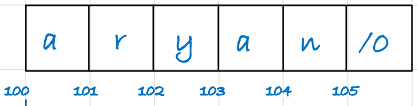
(iii) char arr 1 [] = "hello"

arr1 (arr 1 have the starting address 100)

char arr 2 [] = "hello"

arr2 (arr 2 have the starting address 200)

array representation mai array banta hi banta hai.



ptr (pointer pointing to address 100)

char *ptr = "iram";

ptr (now pointer is pointing to address 200)

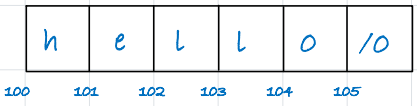
so, in this case we can assign a complete new string but we cannot change the individual element.

in different times there can be different character address. the previous array will be in garbage.

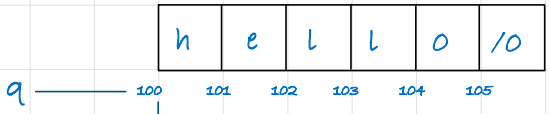
(ii) char name [10] = "aryan";
printf ("%s", ptr+2);

address

(iii) char *p = "hello";
char *q = "hello";



both pointers are pointing to the same location.



(p and q both are pointing to address 100)

pointer representation mai dono ek hi address ko point kar rahe.

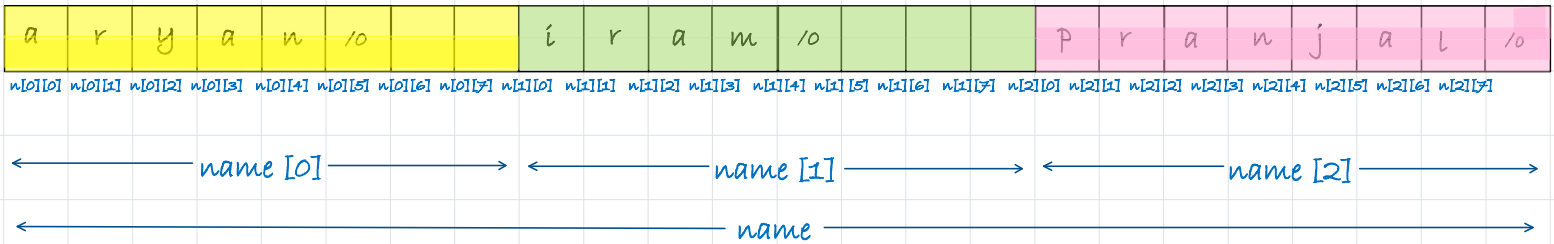
how strings are implemented :

```
char arr1 [] = "aryan"
char arr2 [] = "iram"
char arr3 [] = "pranjal"
```

we can create a 3d array of it

```
char arr [3][8] = {"aryan", "iram", "pranjal"};
```

have to take atleast 8 because "pranjal" have 7 characters.




```
printf ("%s", name[0]); aryan
(address of n[0][0])

printf ("%s", name[0]+2); yan
(address of n[0][0]+2 : n[0][2])

printf ("%s", *(name[0]+2)); y
(value at(n[0][0]+2) : value at (n[0][2])
```

```
name [0] = "aakash" : invalid
it is an array name, we cannot assign a new string to an array.

name [0][0] = 'u' : valid
we can assign a new individual character to an array.
```

how multiple strings are stored using array :

using array of pointer to character.

```
char *p = "aryan"
char *q = "iram"
char *r = "pranjal"
```

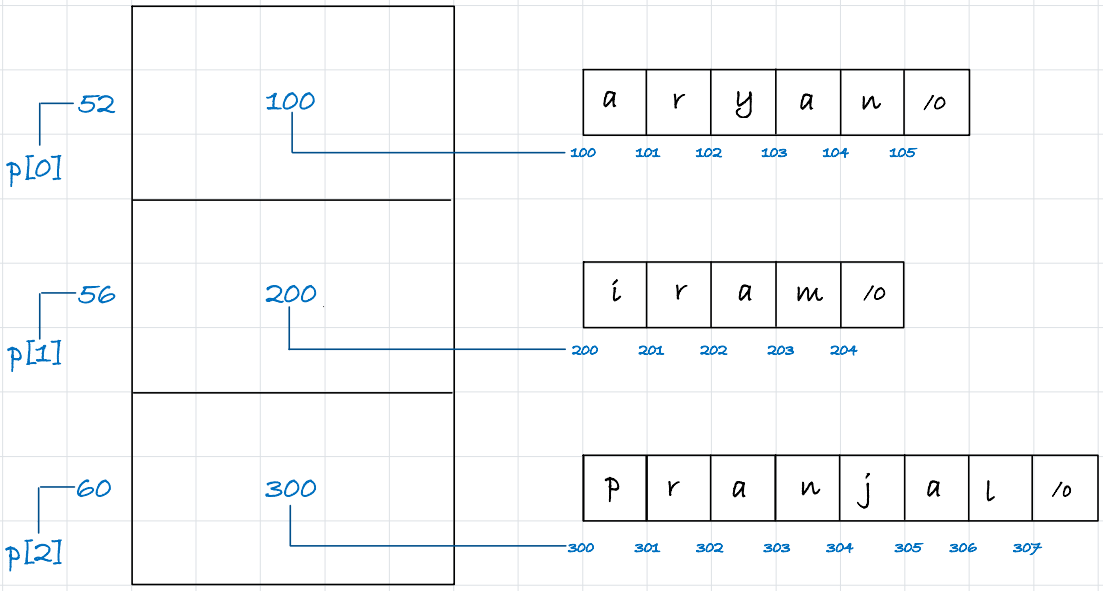
3 pointer to character
they are of same type and for same type element we use array.

char*p[3] = {"aryan", "iram", "pranjal"}

array of 3 pointer to character

char*p[3] = {"aryan", "iram", "pranjal"}

strings stored in read area only.



basically p[3] is an array of address

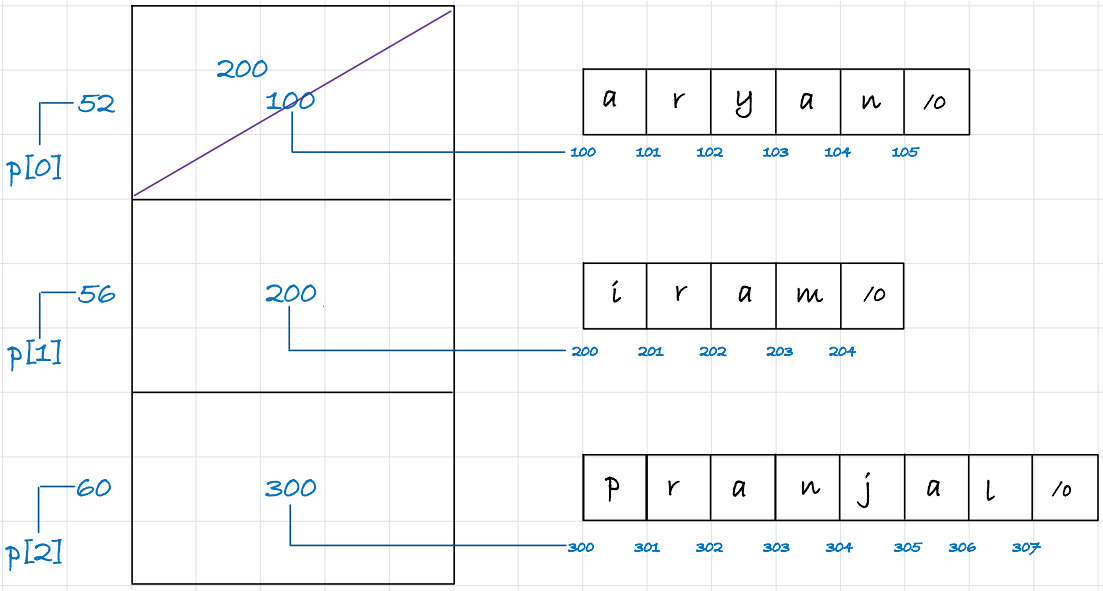
(i) p = &p[0] = memory location 52
(ii) *p = *&p[0] = value at (memory location 52) = 100 (address of character 'a' in "aryan")
- printf("%s", *p); aryan

(i) p+1 = &p[0]+1 = &p[1] = memory location 56
(ii) *p+1 = *&p[0]+1 = *&p[1] = value at (memory location 56) = 200 (address of character 'i' in "iram")
- printf("%s", p[1]); iram

(i) p[1]+2 = &p[1][0]+2 = &p[1][2] = address of 'a' in "iram"
- printf("%s", p[1]+2); am

p[0] = "iram"

strings stored in read area only.



p[0] is now pointing to address 200.

concept: pre-defined functions that works on strings.

(i) strlen
(ii) strcat
(iii) strcmp
(iv) strcpy

(i) strlen

the strlen() function in C calculates the length of a given string till null. it doesn't count the null character '\0'. strlen() function is defined in string.h header file. return type is size_t (which is generally unsigned int)

syntax of C strlen()

```
size_t strlen(const char* str);
```

unsigned int strlen (const char*)

string length cannot be negative

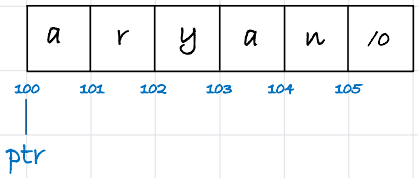
we cannot modify original string

```
char name [10] = "aryan"
int i ;
i = strlen (name)
```
address

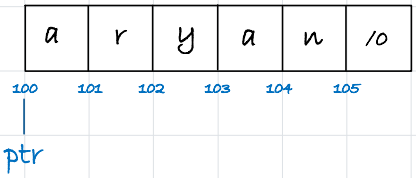
```
char *ptr = "aryan"
int i ;
i = strlen (ptr)
```
address

working of strlen () :

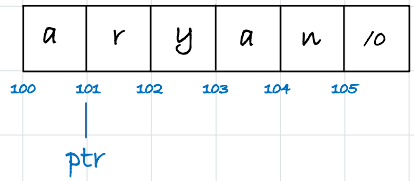
```
#include <stdio.h>
#include <string.h>
void main () {
        int i;
        char *ptr = "aryan"
        i = strlen (ptr)
        printf("%d", i);
                }
```



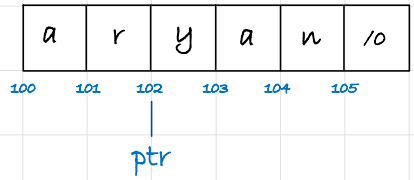
(i) counter : 0
*p = null? no
counter increase to 1
p++ = p = p+1
(points to next address 101)



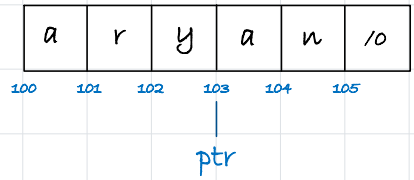
(ii) counter : 1
*p = null? no
counter increase to 2
p++ = p = p+1
(points to next address 102)



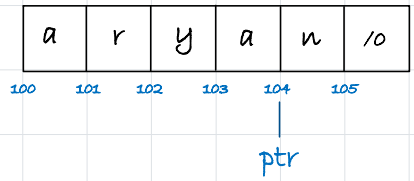
(iii) counter : 2
*p = null? no
counter increase to 3
p++ = p = p+1
(points to next address 10)



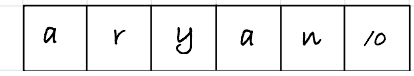
(iv) counter : 3
*p = null? no
counter increase to 4
p++ = p = p+1
(points to next address 14)



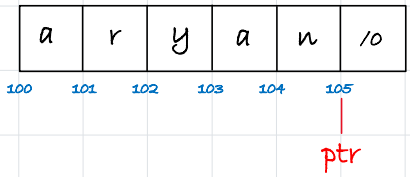
(v) counter : 4
*p = null? no
counter increase to 5
p++ = p = p+1
(points to next address 105)



(vi) counter : 5
*p = null? yes



(vi) counter : 5
*p = null? yes



counter stops.

we can create a for loop for this

```
int mystrlen (const char *p) {
                    int count = 0;
                    while (*p! = '\0')   (until the value at p is not null)
                    {
                     count++;
                     p++;
                           }
                     return count ;
                                  }
```

(ii) strcpy

strcpy is a C standard library function that copies a string from one location to another. It is defined in the string.h header file.

function takes two arguments:

(i) a destination buffer where the copied string will be stored

(ii) a source string that will be copied.

the function copies the entire source string, including the null terminator, into the destination buffer.

it copies the string pointed by source pointer to the buffer/array pointer by destination pointer.

syntax:

```
char* strcpy(char* destination, const char* source);
```

(i) destination: pointer to the destination character array where the content is to be copied.
(ii) source: pointer to the source character array which is to be copied.
(iii)return value: a pointer to the destination string is returned after the strcpy() function copies the source string.

example :

```
char arr [10] ; (we created the array but did not initialise it)
arr = "aryan"; (we cannot initialise the array like this)
```

so, the solution is strcpy

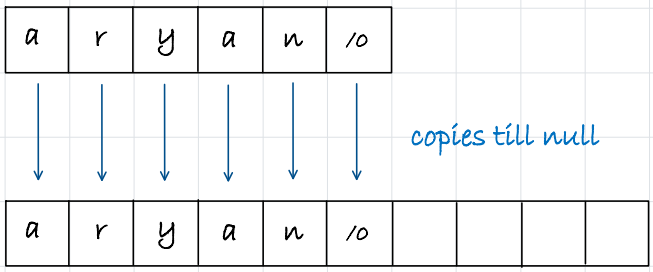
strcpy (arr, "aryan");

"arr" array mai "aryan" string copy karni hai

somewhere in memory

char arr [10];
strcpy (arr, "aryan")

printf ("%s", arr); aryan



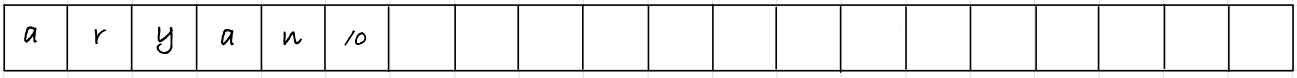
note : do not try to store string in a pointer because pointer ko address milta hai

```
char *ptr;
strcpy (ptr, "aryan");
```

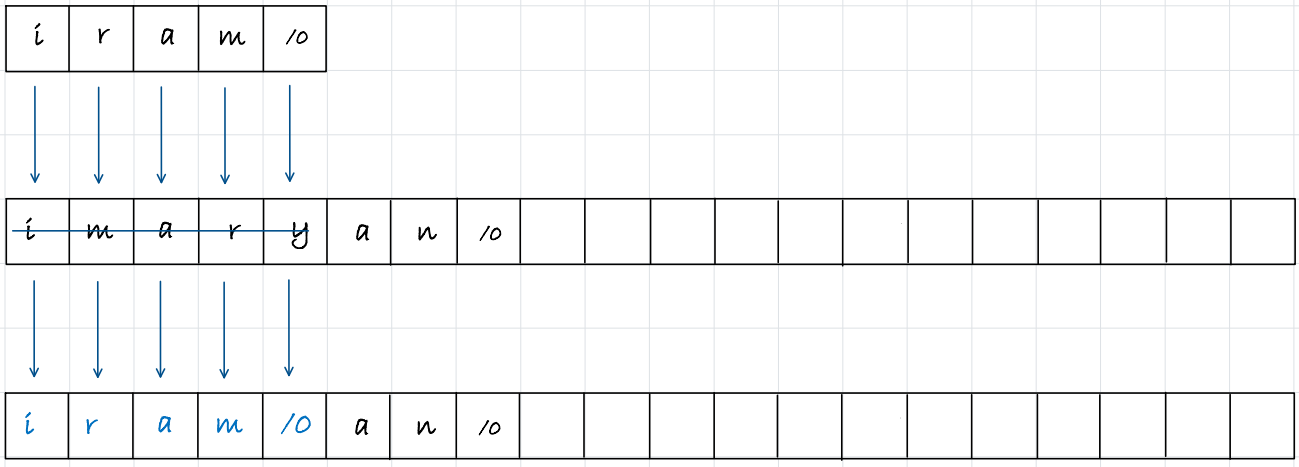
we can overwrite the string in array

```
char arr [20] = "imaryan";
strcpy (arr, "iram");
printf("%s", arr);
```

char arr [20] = "aryan";



strcpy (arr, "iram");



printf("%s", arr); iram

(iii) strcat

strcat() function appends the string pointed to by src to the end of the string pointed to by dest. It will append a copy of the source string in the destination string. plus a terminating null character.

the initial character of the string (src) overwrites the null-character present at the end of the string (dest).

It is a predefined string handling function under string library <string.h> in c

syntax:
char *strcat(char *dest, const char *src);

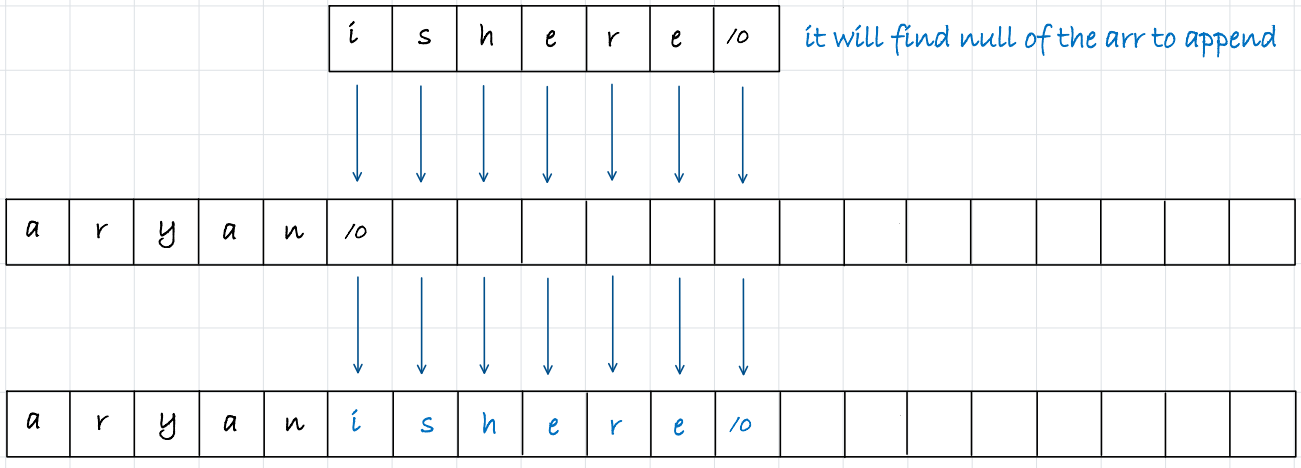
(i) dest: this is a pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string.
(ii) src: this is the string to be appended. this should not overlap the destination.

example :

char arr [20] = "aryan"
strcat (arr, "ishere");

char arr [20] = "aryan";

strcat (arr, "ishere");



note : if you dont want overflow then provide the size in which we can append the new string.

concept : problem with scanf

```
scanf ("%s", arr); aryan ishere
printf("%s", arr); aryan
```

scanf by default first white space par ruk jata hai

that is why we use "^"

```
scanf ("%[^\n]s", arr); aryan ishere
printf("%s", arr); aryan ishere
```

now white space is also allowed, and ab \n par rukega.

- we can also give it the set of characters that we want to allow in our string

```
scanf ("%[a-z, A-Z]s");
```

concept : gets () and puts ()

(i) gets () : reads the character and stores them as a string in str. (white space allowed)

syntax

```
int *gets (char *str);
```

str : pointer to a block of memory (array of char) where the string read is copied as string.

(ii) puts () : prints the string character until null.

syntax

```
int puts (char *str);
```

str : string to be printed.

(q). char s [20] = "aryanishere"

```
printf ("%s", s+s[2]-s[3]);
```

s+s[2]-s[3]	a : x
s+y-a	b : x+1
s+x+24 - x	c : x+3
s+24	.

(q). char s [20] = "pankajsharma"

```
printf ("%s", s+s[2]-s[3]);
```

s+s[2]-s[3]	k : x
s+n-k	l : x+1
s+x+3 - x	m : x+2
s+3	n : x+3

s+x+24 - x
s+24
s[24]
error.

c : x+3
.
.
y : x+24

s+x+3 - x
s+3
s[3]
kajsharma.

m : x+2
n : x+3

(q). printf ("%s",ptr+B-A);

ptr+B-A
ptr + x+1-x
ptr+1
ptr [1]

A : x
B : x+1

(q). char arr [] = "pankajsharma"

printf ("%s", &arr[6]); sharma

printf ("%s", &6[arr]); sharma

printf ("%s", arr+6); sharma

printf (&arr[6]); sharma

printf (&6[arr]); sharma

printf (arr+6); sharma

printf ("%d", sizeof("pankaj")); 7

printf ("%d", sizeof("pankaj\0")); 8

(q). char arr [10] = "aryan";

printf ("%d", strlen(arr)); 6

printf ("%d", sizeof(arr)); 10

structure and union

topic : structure

structure in is a user-defined data type that can be used to group items of possibly different types into a single type.

"struct" keyword is used to define the structure in the C programming language.

the items in the structure are called its member and they can be of any valid data type. additionally, the values of a structure are stored in contiguous memory locations.

how to represent a student in a software?

student
- int roll
- char name
- etc

(i) struct is the keyword used to create user defined data type. just like primitive data type; int, float, char. from now on a new data type exsist : struct student.

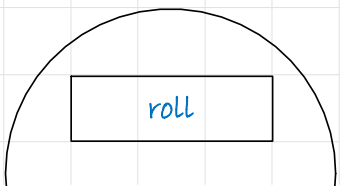
```
struct student {
        int roll;
        char name [20];
                }
```

```
struct student {
        int roll;
        char name [20]="aryan";
                }
```

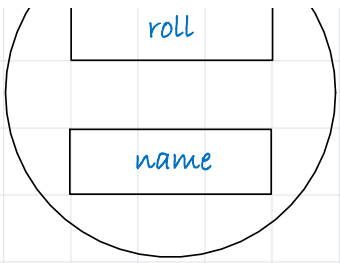
no memory is allocated, this is the information for the compiler that whenever i create the student type structure there will be (int roll and char name) in it as members. we cannot initialise it also because there no memory to store the value.

(ii) memory will only allocate when we create the variable for the data type.

```
struct student {
        int roll;
        char name [20];
                }
void main {
        int ·
```



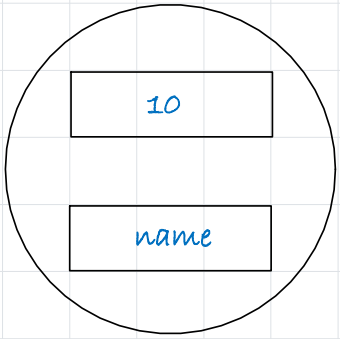
```
                                        }
    void main {
                int ;
                struct student s1;
                                    }
```



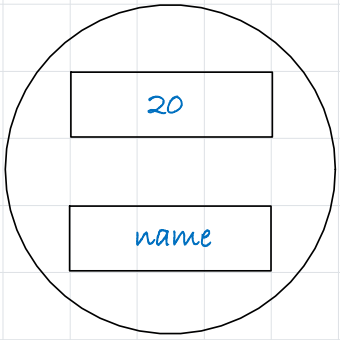
s1

s1 variable is a group of two members
- roll
- name

```
struct student {
                int roll;
                char name [20];
                                }
void main {
                int ;
                struct student s1s2;
                s1.roll = 10;
                s2.roll = 20;
                s1.name = "aryan";
                                }
```



s1



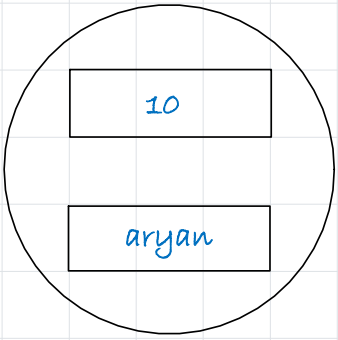
s2

s1.name = "aryan";

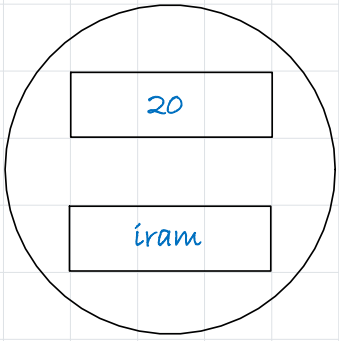
this will not execute because array name cannot be lvalue.

so, we have discussed the solution before

```
strcpy (s1.name, "aryan");
strcpy (s2.name, "iram");
```



s1



s2

- : it is a membership operator, to access a particular member of any structure we need the membership operator.

concept : global vs optional vs typedef

global : defined the template globally, iske variable hum kisi bhi fun mai bana sakte hai

```
struct student {
                int roll;
                char name [20];
                                }
void main {
            int ;
            struct student s1s2;
                                }
void fun (){
             struct student s1s2;
                                }
```

optional: template is optional, iske variable hum ek hi baar bana sakte hai that are defined globally.

```
struct student {
                int roll;
                char name [20];
                                }s1, s2, s3;
void main {
            cannot create variables here
                                }
```

typedef: typedef is a keyword that is used to provide existing data types with a new name.

```
struct student {
                int roll;
                char name [20];
                                }pankaj;
void main {
            pankaj s1, s2;
                                }
```

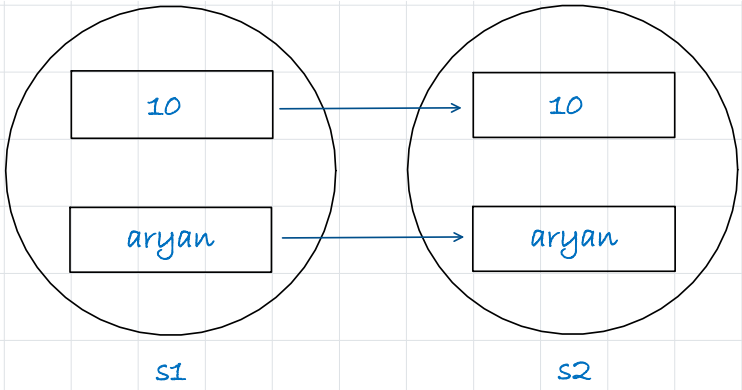
struct student template name changed to pankaj.

(iii) the order of initialisation must be as same as the orde of members.

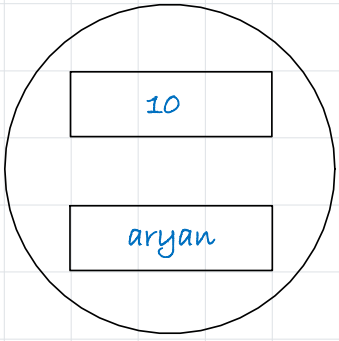
```
struct student {
                int roll;
                char name [20];
                                }
void main {
            int ;
            struct student s = {10, "aryan"};
            {s.roll, s.name} = {10, "aryan"};
            {s.name, s.roll} = {10, "aryan"};
                                }
```

(iv) we can copy the data of one structure to another.

```
struct student {
                int roll;
                char name [20];
                                }
void main {
            struct student s1 = {10, "aryan"};
            struct student s2 = s1;
            struct student s3 = {"aryan"};
                                }
```



(v) the values of a structure are stored in contiguous memory locations.



roll.	name
10	pankaj

topic : union

the union is a user-defined data type that can contain elements of the different data types just like structure. but unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given instance.

(i) union is they keyword that is used to define the data type.

```
union pankaj {
        char i;
        int j;
          }
```

(ii) in case of structures all members get individual (separate) memory space but all members of union share common memory area.

```
union B{
        char i; 1 byte
        int i; 4 byte
```

max (1,4) : 4 byte
sabse bade member ke size ke equal

```
union B{
        char i; 1 byte
        int j; 4 byte
            }
void main {
          union B b;
          printf ("%d", sizeof(b)); 4
                                    }
```

max (1,4) : 4 byte
sabse bade member ke size ke equal
memory allocate hoti hai

topic : scoping

(i) static scoping (lexical scoping) : scope related decision are taken on compile time.

(ii) dynamic scoping : scope related decision are taken on run time.

where to look for the value of a?

```
int a ;
    {
    |   {
    |   |   {
    |   |   | printf ("%d", a);
    |   |   }
    |   }  sub sub scope
    } sub scope
main scope
```

consider the program in a hypothetical language that allows global variables and a choice of static and dynamic scoping.

what is printed under static scoping?

```
int i ;
program main () {
                    i = 10;
                    call f();
                            }
procedure f (1) {
                int i = 20;
                call g();
                        }
procedure g () {
                print (i);
                        }
```

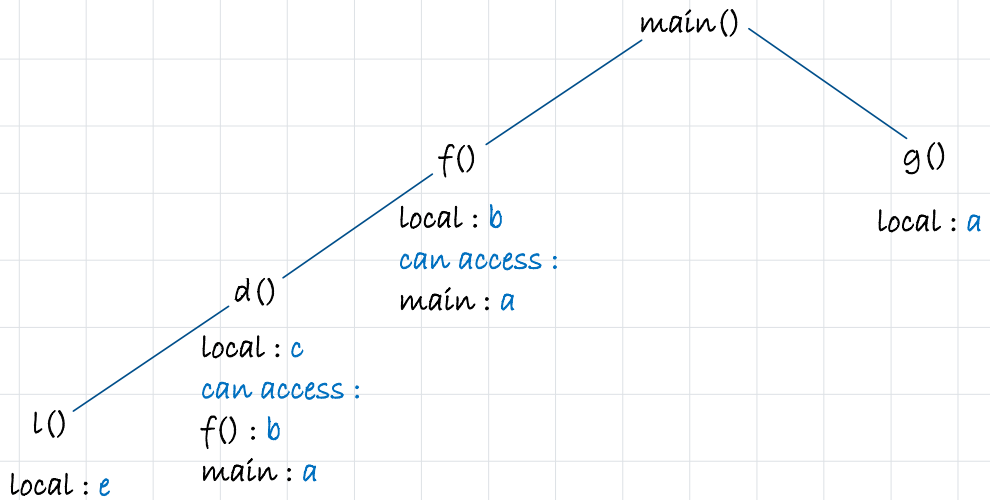
```
in c :
int i ; global variable
void main () {
                i = 10;
                call f(); called the function f()
                        }
void f(1) {
            int i = 20; f have its own local variable 20
            call g(); called the function g()
                    }
void g() {
          print ("%d",i);    (f(1) terminated, it was a local
                          } variable. so the value of i will be 10.
```

what is printed under dynamic scoping? (value nahi mil rahi toh parent function mai search)

```
int i ;
program main () {
                    i = 10;
                    call f();
                              }
procedure f (1) {
                    int i = 20;
                    call g ();
                              }
procedure g () {
                    print (i);
                              }
```

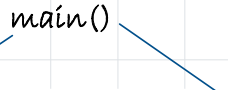
```
in c :
int i ;  global variable
void main () {
                    i = 10;
                    call f(); called the function f()
                              }
void f(1) {
                    int i = 20; f have its own local variable 20
                    call g (); called the function g ()
                              }
void g () {
                    print ("%d",i);    g () does not have its own i, it will look
                                      } in the parent function, so 20 will be
                                        printed)
```

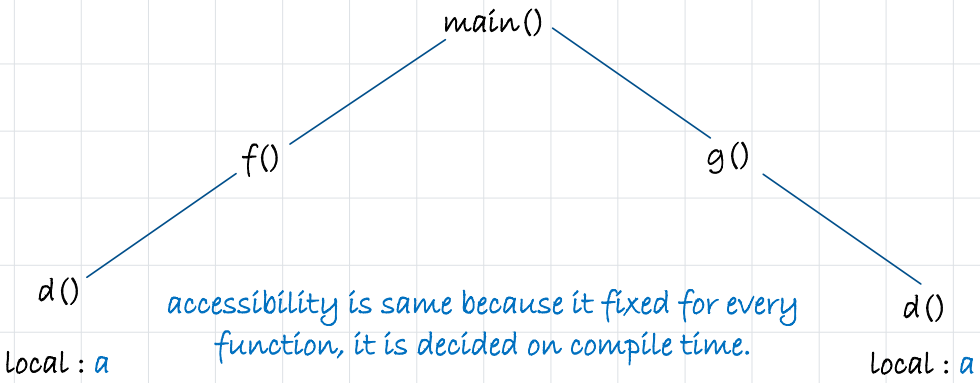
in dynamic scoping, function can access all the parent variables.



accessibility is different because it is not fixed for every function, it is decided on run time (kaunsa function kisko call kar raha).

in static scoping, function can access own variables as well as global variables




main()
f()
g()
d()
local : a
accessibility is same because it fixed for every function, it is decided on compile time.
d()
local : a