Bit representation

Binary digit --- 2 states
--> system understands 0 or 1, true or false, high or low

Byte

---> group of 8 bits -- 1 byte

Character

--> alphabets and symbols
a,b,c,d.....z
---> character code --> ASCII, UTF

a ---> ascii equivalent value of ---> 97 ---> binary
2^7 ---> 128 -- ascii --- 0 to 127

A ---> 65 --- 8bit --->

01000001

2 | 65
2 | 32 -- 1
2 | 16 -- 0
2 8 --- 0
2 4 -- 0
2 2 -- 0
1 -- 0

Virat

Word Representation

8-bit, 32-bit, 64-bit
--> an amount of data processor can fetch and process at one time -- word size

8-bit --- 1byte
32-bit -- 4bytes

0 ---> 0000 0000 0000 0000 0000 0000 0000 0000
A - 65 --- 0000 0000 0000 0000 0001 000001

Integer Number

--> whole number, positive, negative

8-bit

8 ---> 0000 1000

13 -- 32-bit
0000 0000 0000 0000 0000 0000 0000 1101

-13 ---> 2's complement

1's complement + 1

13 ---> 0000 0000 0000 0000 0000 0000 0000 1101
1's ---> 1111 1111 1111 1111 1111 1111 1111 0010
\+ 1
1111 1111 1111 1111 1111 1111 1111 0011

convert 1 to 0 and 0 to 1

Addition
0+0 = 0
0+1 = 1
1+0 = 1
1+1 = 10

11
+1
100

-8 in 8-bit

0000 1000

1111 0111
1
1111 1000

128 + 64 + 32 + 16 + 8
248

$-k = 2^n - k$ where n is number of bits
k is negative integer
$-8 = 2^8 - 8 = 256 - 8 = 248$

$2^{32} - 8 = 4294967288$

-1 in 8-bit, binary and decimal

1111 1111 --- 256 -1 = 255

Floating Point representation

3 parts

sign bit --- negative, positive
exponent --
mantissa -- part of log after decimal point

Float conversion

1. 0.5 combination of 2 part -- integral, fractional

Step 1: convert both integral and fractional into a binary

0

.5 * 2 = $\boxed{1}$.0 ---> 1

.0 * 2 = 0.0 ---> 0

.5 ==== 10

0.5 ----> 0.10

Step 2: convert result of step 1 to standard exponent format

$1.Xe^{\pm y}$

0 . 1 0

---> shift the decimal point either left or right in a way that it makes the value as 1.whatever

1.0

---> number of shift is y
if shifted to right side it will be negative value
else if shifte to left side it will be positive value

y = -1

X is mantissa which is after decimal point value

X = 0000 0000 0000 0000 0000 000

Step 3: Find the exponent using standard bias number

float --- 127

exponent = 127 +/- y = 127 - 1 = 126 = 01111110

0.5 float

sign bit ---> 0
exponent --> 0111 1110
mantissa --> 0000 0000 0000 0000 0000 000

sign	exponent - 8	mantissa - 23
0	0111 1110	0000 0000 0000 0000 0000 000

2. 8.25

Step 1 : convert to binary

8 ---> 1000

.25 --> 010

.25 * 2 = 0.5
.5 * 2 = 1.0
.0 * 2 = 0.0

8.25 ==== 1000.010

Step 2: convert result of step 1 into a standard exponent format

1 0 0 0 . 0 1 0

1 0 0 . 0 0 1 0

1 0 . 0 0 0 1 0

1. 0 0 0 0 1 0

left shift so positive value

y = 3

X = 000010 00000000000000000

Step 3: find the exponent

exponent : 127 + 3 = 130 ---- 1000 0010

sign	exponent - 8	mantissa - 23
0	1000 0010	0000 1000 0000 0000 0000 000

-8.25 ---- all the steps remain same only thing is sign bit will be 1

sign	exponent - 8	mantissa - 23
1	1000 0010	0000 1000 0000 0000 0000 000

1. 10.15
2. 3982.225
3. -5.45

float

1. 10.15

1010.001001.....

never ending number

.15 * 2 = 0.30
.30 * 2 = 0.60
.60 * 2 = 1.20
.20 * 2 = 0.40
.40 * 2 = 0.80
.80 * 2 = 1.60

.15 = 0$\overline{01001}$

.60 * 2 = 1.20
.20 * 2 = 0.40
.40 * 2 = 0.80
.80 * 2 = 1.60

step 2: 1.Xe^+/-y

1010.001$\overline{001}$

1.010001$\overline{001}$

y = 3

X = 010001001 1001 1001 1001 10

sign bit = 0

step 3: exponent = 127 + 3 = 130

10000010

Double --- 64 bit

--> all the steps remain same except the standard bias number and number of bits in exponent and mantissa

1. 8.25

1000.010

2. 1.000010

   y = 3

   X = 000010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 00

   sign bit = 0

3. in case of double the standard bias number is 1023

   exponent = 1023 + 3 = 1026

   10000000010

   0 10000000010 000010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 00

3. 1.7

   float and double

   step 1 :

   $1.1\overline{0110}$

   step 2:
   y = 0

   X (float) = 10110 0110 0110 0110 0110 01
   X (double) = 10110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 011

   step 3:

   float = 0111 1111
   double = 01111111111

   sign = 0

Data types

type of information -- integer, character, floating.......

basic types / primitive types
---> char ---> ascii --- 1 byte of memory is allocated
1 byte integer
---> always enclosed within the single quote
'a','b','c'.........'0'

int ----> size is compiler implementation dependant

gcc, turbo c, BDS, Clang(mac os), visual express........

turbo c -- 2bytes
gcc --- 4bytes

---> numeric data ---- number, hexa, octal

-----> float ---> real values
4 bytes

double ---> 8bytes

---> sizeof ---> returns the total memory allocated for the variable or datatype

Declaration: declare the variable

initialisation: assigning the values to the variables

syntax:
datatype variable_name;

```
int age;
char option;
float attendance;
double avg;
```

int age; declaration
age = 28; initialisation

or int age = 28; ---> definition

sizeof --- value ---- format specifiers --- %zu --- positive integer value
common format specifier for all the system

sizeof(variable_name)
sizeof(datatype)
sizeof variable_name

```
printf("Hello world");
printf("%zu", sizeof(age));
printf("%zu",sizeof(int));
```

to display the value of variable the format specifiers are used

int ---> %d (%i)
char ---> %c
float ---> %f
double --> %lf

By default all the variable will be having garbage value -- positive, negative, 0

scanf --- read the values from the user

scanf("formatspecifier / s", &num);

int num;

num $\boxed{\text{GV}}$

1000 --> assumption

size modifiers

---> tune the width of the datatype

int ---> %d,
octal --> %o, hexa --> %x

short

---> 2 bytes of memory
---> can apply only to int type
---> %hd -- decimal
%hx -- hexa decimal 2bytes
%ho -- octal in 2 bytes

declaration:
short int variable_name
short variable_name;
//default its a int

long

--> int and double

int -- Compiler Implement
Dependant
4bytes / 8bytes
long int variable_name
%ld, %lx, %lo

double -- CID
12bytes / 16bytes
-- %Lf
long double variable_name

long variable_name;
//default - long int variable

long long

---> only applicable for int

---> compiler implementation dep
8bytes / 16bytes

---> %lld

---> long long int variable_name;
long long variable_name;

```
int num = 67;
char ch = 'd';

printf("%c  %d\n", num, ch);
```

signedness

--> instruct the compiler whether the variable is cpable of holding positive or negative value

---> by default integral variables are signed variable that is it supports both positive and negative

1. signed

---> only for integral datatypes --- int and char

char ch;
int num;

OR

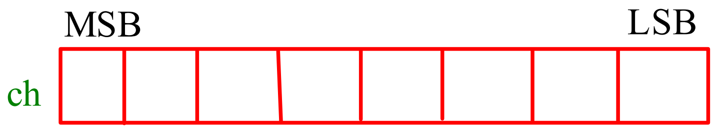
signed char ch;
signed int num;

-12

1

--> you can store both positive and negative value

signed char ch;



MSB -- Most Significant Bit

LSB -- Least Significant Bit

0000 0000 --- 0
1000 0000 --- 128

0000 0001 --- 1

MSB is dedicated for sign
0 -- positive
1 -- negative value

for positive,

0000 0000 to 0111 1111 ==> 0 to 127

for negative,
1000 0000 to 1111 1111 ==> -128 to -1

2s complement

1000 0000
0111 1111
+ 1
~~1000 0000~~ --- -128

1111 1111
0000 0000
+ 1
0000 0001 ---> -1

In general signed character can hold the range = -128 to 127

general form = -2^(n-1) to +2^(n-1) -1, where n is the number of bits

-2^(8-1) to +2^(8-1) -1 = -128 to 127

signed short int;

-2^(16-1) to +2(16-1) -1 = -32768 to +32767

int -- -2^31 to +2^31 -1

signed char ch = 128;

1000 0000
0111 1111
+   1
1000 000 --   -128

signed char ch = 133;

1000 0101
0111 1010
+   1
0111 1011   -- -123

133 - 256 = -123

256 - 133 = -123

2. Unsigned --> only +ve number

--> no special dedication for MSB
---> integral types

unsigned int num;
unsigned char ch;

if unsigned char ch;

0000 0000 to 1111 1111 ---> 0 to 255

general form = 0 to 2^n -1, where n is number of bits

-23

256 -23 = 233

0001 0111
1110 1000
+   1

11101001   ---> 233

format specifier --- unsigned int == %u
unsigned long int == %lu

signed int num; ---> 4bytes

signed short int num; -- short

program
--> set of instructions / set of statements / collection of statement

```
int num;  //declaration statement

printf("");

if()
{
}

functions()
{
}
```

```
if()
{

 if()
{
 }
}
```

Conditional construct

--> depending on the particular condition whether it is true or false, block of code will be executed

avg > 55 ---

syntax:

```
if(condition)
{
    //logic / statement/s
}
```

```
int main()
{
    declaration;

   conditional construct

    .....
}
```

```
int main()
{
    int num = 11;

    if(num < 10)
    {
        printf("num is less than 10\n");
        printf("num is %d\n", num);
    }
  return 0;
}
```

```
if(num < 10)
{
    printf("");
}
printf()
```

if ... else

```
if(condition)
{
}
else
{
...
}
```

else should always followed by the if

there should not be any statement in between the if and else

;

num = 2

```
if (num < 5);
{
        printf("num is smaller than 5\n");
}
```

===

compiler view

```
if(num < 5)//2 < 5
{
    ;
}
{
      printf(".....");
}
```

```
if (num < 5);
{
        printf("num is smaller than 5\n");
}
else
{
        printf("num is greater than 5\n");
}
```

==

```
if (num < 5)
{
    ;
}
{
        printf("num is smaller than 5\n");
}
else
{
        printf("num is greater than 5\n");
}
```

```
if (num < 5)
        printf("num is smaller than 5\n");
        printf("Emertxe\n");
else
{
        printf("num is greater than 5\n");
}
```

===

```
if(num < 5)
{
    printf("num is smaler\n");
}
printf("Emertxe\n");
else
{
      printf("....");
}
```

Error

if .. else if

avg > 55 ---- second class

avg > 75 ---- firstclass

distiction

```
if ( condition )
{
    statement
}
else if( condition )
{
      statement
}
else
{
      statement
}
```

```
num = 6

if (num < 5)//6 < 5 --- true
        {
                printf("num is smaller than 5\n");
        }
        else if (num > 5) 6 > 5
        {
                printf("num is greater than 5\n");
        }
        else
        {
                printf("num is equal to 5\n");
        }
```

```
if( condition )
{

}
```

```
if(condition)
{
}
else
{
}
```

```
if()
{
}
else if()
{
}
```

&& --- logical and
both condition is true

num1 < num2 && num1 < num3

nested if....else

if within another if

```
if(condition)
{
      if(condition)
      {
            statement
      }
}
```

```
if()
{
    if()
    {
    }
    else
    {
    }
}
else
```

avg > 55 -- Second class
avg > 56 and < 70 ---- first class
avg > 71 -- distinction

<50 ---> fail

num1 = 20, num2=10, num3=30

10	20
20	40
30	30

20 < >

switch case

--> single iteration

constant --- label but a constatnt
integer constant, character const, enum(later)

Real values are not allowed -- case 3.4
should be a unique value -- case 10...... case 10
multiple values are not allowed --- case 10,20,30 error
variable name is not allowed(non constant)

int num -- compile time
print("")

errors from developer
when u make mistake in the logic

segmentation fault -- run time error

unexpected result

--> depends on requirements

x = 10;

```
if (x == 100)        1 cycle
{
}
else if(x == 90)     1 cycle
{
}
else if(x == 80)     1 cycle
{
}
else                 3 CPU cycle
{
}
```

for each comparison = 200nsec assume

3 * 200 = 600ns

switch (x) 10 JUMP
{
case 100:
case 90: 1 cycle
case 80:
default: 200nsec
}

Multi iteration

--> statements are executed repeatedly until thhe condition is true
---> while loop, do...while, for

1. while loop
   ---> entry controlled loop
   --> it will start executing the statement only when condition is true
   --> once it starts executing, it will continue the execution until the condition is true

```
int iter;
iter = 0;
while (iter < 5)
{
printf("Looped %d times\n", iter);
iter++;
}
```

1. iter = 0
   0 < 5 -- true - enter the loop
   iter = iter + 1
   = 0 + 1
   iter = 1
2. 1 < 5 true
   iter = 2
3. 2 < 5 true
   iter = 3
4. 3 < 5 true
   iter = 4
5 4 < 5
   iter = 5
6. 5 < 5 -- false
   exit the while loop

Looped 0 times

Looped 1 times

Looped 2 times

Looped 3 time

Looped 4 times

do ... while

---> exit controlled loop
---> first it will enter the loop execute the statement while exiting from loop it will check for the condition, if condition is true continue the loop else stop executing the loop

```
int iter;
iter = 0;
do
{
    printf("Looped %d times\n", iter);
    iter++;
} while (iter < 5);
return 0;
}
```

1. iter = 0
   Looped 0 times
   iter = 1

2. 1 < 5
   Looped 1 times
   iter = 2

3. 2 < 5
   Looped 2 times
   iter = 3

4. 3 < 5
   Looped 3 times
   iter = 4

5. 4 < 5
   Looped 4 times
   iter = 5

6. 5 < 5 -- false
   stop the execution

while	do ... while
1. Entry controlled loop	1. exit controlled loop
2. when condition is true	2. atleast for once the statement will be executed
3. When?	--->
number of times of execution is unknown	menu driven applications
when u know the entry condition	---> atleast for once the menu should be displayed
	---> ATM ---

For loop

1 once   2 ⇐ true   4

```
for ( initialisation ; condition eval ; post expression evaluation)
{
    statement
}
```

3

false

```
int iter;

for ( iter = 0; iter < 5 ; ++iter)
{
    printf("Looped %d times\n", iter);
}
```

1. iter = 0
   0 < 5 --- true
   Looped 0 times
   post eval -- iter = 1

2. 1 < 5
   Looped 1 times
   iter = 2

3. 2 < 5
   Looped 2 times
   iter = 3

4. 3 < 5
   Looped 3 times
   iter = 4

5. 4 < 5
   Looped 4 times
   iter = 5

6. 5 < 5 -- false
   exit the for loop

---> number of execution is known
array --- fixed size

```c
int iter;
iter = 0;

while (iter < 5)
	printf("Looped %d times\n", iter);

iter++;
```

```c
int iter;
iter = 0;

while (iter < 5)
{
	printf("Looped %d times\n", iter);
}

iter++;
```

1. 0 < 5 -- true
   Looped 0 times

until the condition true

2. 0 < 5 -- true
   Looped 0 times

infinite loop

```c
int iter;
iter = 0;
while (iter < 5)
	iter++;

printf("Looped %d times\n", iter);
```

1. 0 < 5 -- tru
   iter = 1

2. 1 < 5 -- true
   iter = 2

3. 2 < 5
   iter = 3

6. 5 < 5 -- false
   exit from the loop

Looped 5 times

```c
int iter;
iter = 0;
while (iter < 5);
{
	printf("Looped %d times\n", iter);
	iter++;
}
```

```c
while (iter < 5)
{
	;
}
{
	printf("Looped %d times\n", iter);
	iter++;
}
```

1. 0 < 5 -- true
   ;

infinite loop without any output

2. 0 < 5 -- true
   ;

```
for (iter = 0; iter < 5; iter++)
{
    ;
}
{
        printf("Looped %d times\n", iter);
}

printf("Iter value outside loop: %d\n", iter);
```

1. 0 < 5 -- true
   ;
   iter = 1

2. 1 < 5
   ;
   iter = 2

3. 2 < 5
   ;
   iter = 3

iter = 5
5 < 5 -- false

Looped 5 times

5

```
iter = 0
for ( ; iter < 5; )
{
    printf()
    iter++;
}
```

```
for( ; ; )
{
}
```

1. initialisation always executes only once first time
2. condition and post evaluation will repeatedly executed until the condition is true

true values and false values

true values ---> positive, negative, character, float, double
false --- 0, "", NULL

Nested for Loop

---> pattern printing
---> 2d array

```
for ( i = 0; i < 3; i++)
{
    for( j = 0 ; j < 2; j++)
    {
          printf("Hello\n");
    }
}
```

1. 0 < 3 -- true
   enter the loop

   1.1 j = 0
       0 < 2 -- true
       Hello
       j = 1
   1.2 1 < 2 -- true
       Hello
       j = 2
   1.3 2 < 2 -- false
       exit the loop

2. i = 1
   1 < 3 -- true

   2.1 j = 0
       0 < 2 -- true
       Hello
       j = 1
   2.2 1 < 2 -- true
       Hello
       j = 2
   2.3 2 < 2 -- false
       exit the loop

3. i = 2
   2 < 3

   3.1 j = 0
       0 < 2 -- true
       Hello
       j = 1
   3.2 1 < 2 -- true
       Hello
       j = 2
   3.3 2 < 2 -- false
       exit the loop

4. i = 3
   3 < 3 -- false
   exit outer loop

i \ j	0	1
0	Hello	Hello
1	Hello	Hello
2	Hello	Hello

```
for ( i = 0 ; i < 5; i++)
{
}
for(j = 0 ; j < 2; j++)
{
}
```

not a nested loop

1. * * * * * *

```
for( i = 0; i < 6; i++)
{
    printf("* ");
}
```

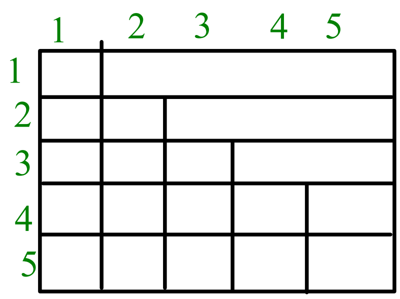
2.

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

```
for(i = 0; i < 5; i++)
{
    for ( j = 0; j < 5; j++)
    {
        printf("* ");
    }
}
```

3.

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



1st row --- 1 column
2nd row -- 2 column
3rd row -- 3 column
4th row -- 4 column
5th row -- 5 column

col = row

```
for(i = 0; i < 5; i++)
{
    for(j = 0; j <= i; j++)
    {
        printf("* ");
    }
    printf("\n");
}
```

Break statement

Used to break the loop on the given condition
used in switch case to exit from the switch

1000 ---> id = 100 -- loop, break

```
for( i = 0; i < 10; i++)
{
    if(iter == 5)
    {
        //logic
        break;
    }
    printf("Iteration %d\n", i);
}
```

1. 0 < 10 -- true
   0 == 5 false

   iteration 0

2. 1 < 10 -- true
   1 == 5 false
   iteration 1

3. 2 < 10
   2==5
   2

4. 3

5. 4

6. 5 < 10 -- true
   5 == 5 -- true

   break -- exit from loop

--> should be within the loop or switch case

Continue

--> only used within the loop
--> will skip that particular iteration and goto the post evaluation or condition evaluation

```
for(i = 0; i < 5; i++)
{
    if( i == 2)
    {
        continue;
    }
printf("%d\n", i);
}
```

1. 0 < 5 -- true
   0 == 2

   0
2. 1 < 5
   1 == 2
   1

3. 2 < 5

   2 == 2
   skip the execution

4. 3 < 5
   3
5. 4

```
for( i = 0; i < 10; i++)
    for(j = 0; j < 100; j++)
```

Outer loop:
- initialisation : 1
- condition : 11
- post eval : 10

1 + 11 + 10 =
22Machine cycle

Inner Loop
- initialisation : 10
- condition : 101 * 10 = 1010
- post eval : 100 * 10 = 1000

2020 MC

22 + 2020 = 2042 Machine Cycle

```
for(i = 0; i < 100 ; i++)
    for(j = 0; j < 10; j++)
```

Outer loop : 1 + 101 + 100 = 202 MC

Inner : 100 + 1100 + 1000 = 2200

2402 Machine cycle

Operators

---> instruct the compiler to do the specific / some operation on operands

```
int num;
num = 7 - 4 * 3 / 2 + 5;
printf("Result is %d\n", num);
```

1. Precedence -- rank

2. Associativity -- left to right
   right to left

7 - 4 * 3 / 2 + 5

7 - (4 * 3) / 2 + 5

7 - ((4 * 3) / 2) + 5

(7 - ((4*3) / 2)) + 5

3 1 2 4

6

grouping -- adding the paranthesis

- * / +

/ * -- same rank, L to R

+ - -- same precedence , L to R

Unary Operators

--> single operand / value
---> ++, --, +, - ....

int x, y = 10;

x = -y;
= -(10)
=-10

int x, y = -10;

x = -y;
= -(-10)
= +10

int x, y = -10

x = +-y; // same precedene, R to L

= + (-(-10))
= +10

Increment and Decrement Operator

---> single operand

Increment

1. Pre increment -- ++operand
   ---> increments the value of operand by 1

   1. Increment first
   2. Use / assign the incremented value

int x = 10, y;

y = ++x;

1. Increment first
   x = x + 1 = 10 + 1 = 11

2. assign
   y = 11

2. Post increment -- operand++
---> increment by 1

1. Use / assign the value first
2. Then incremnet the value of operand

int x = 10, y;

y = x++;

1. Assign the value
y = 10

2. Increment the value of x
x = 11

Decrement Operator

1. Pre decrement --- --operand

decrement the vlaue by 1

int x = 10, y;

y = --x;

x = 9 y = 9

2. Post decement

y = 10

x = 9

sizeof
--> is a operator --- value, variable, datatype
--> returns total memory allocated in unsigned int or long int ---> %u, %lu
%zu -- works on any system
--> sizeof(). sizeof var

sizeof is a compile time operator

--> returns the memory at compile time
---> never reflects the value or changes the value of variable

Type conversion

--> process of converting one type of data to another

```
int x = 10,  y = 4, z;

z = x / y;

 = int / int ==> int

 = 10 / 4
 = 2
```

```
2. int x = 10, y = 4;
   float z;
                         z = x / y;
                           = int / int = int
                           = 10 / 4 = 2
                         z = 2.000000
```

```
3. int x = 10;            z = x / y;
   float y = 4, z;          = int / float;  ---> implicit type conversion(type casting)
```

--> whenever 2 different types of data then compiler will do the type conversion based on hierarchy table

---> lower precedence will be converted to higher precedence type

```
z = 10.000000 / 4.000000

z = 2.500000
```

```
4. int x = 10, y = 4;        z = float / int; --> implicit
   float z;                    = float / float;
                             z = 10.000000 / 4.000000
   z = (float) x / (float) y; explicit
```

```
int num1 = 5,
float num2 = 3
float avg = num1 / num2;
```

Unary conversion

--> char and short both will be converted to integer

```
1. char ch = 12, y = 20, z;
   z = ch + y;                    printf("%d", z);
   z = 12 + 20 = 32
```

assignment

```
int x;
char y = 'a';

x = y;
```

if RHS is of lower rank and LHS is of higher rank then it will promote the value to higher rank

LHS = RHS
x = 97; ---> type promotion

```
float x = 5.4;
int y;

y = x;
```

if RHS is of higher rank and LHS is of lower rank then it will demote the higher rank to lower rank
--- type demotion

y = 5;

Logical Operator

--> &&, ||, !

binary

unary

Truth table

Logical AND --- &&

A	B	Output
False(0)	False(0)	False(0)
False	True(1)	False
True	False	False
True	True	True

For logical AND if any one of the input is false then output will be false else true

Logical OR -- ||

A	B	Output
False(0)	False(0)	False(0)
False	True(1)	True
True	False	True
True	True	True

For OR if any 1 input is true then output will be true else false

3. Logical NOT -- !

A	Output
False	true
True	false

To ptimize the time it will apply short circuit evaluation whenever a logical operator is used

--> in case of logical OR if first statement is true then second statement will not be evaluated

```
int num1 = 1, num2 = 0;

if(num2++ || ++num1)
{
        printf("If: %d %d\n", num1, num2);
}
else
{
        printf("Else: %d %d\n", num1, num2);
}
```

num2 = 0 || 2 -- true
num2 = 1

++num1 || num2++
2 || not evaluated
true
if --> 2, 0

---> in case of logical AND, if first expression is false then second is not evaluated

```
if(num2++ && ++num1)
{
        printf("If: %d %d\n", num1, num2);
}
else
{
        printf("Else: %d %d\n", num1, num2);
}
```

0 -- flase
0++ && num1
false &&
false

1 1

int a = 50, b = -50, c = 0, d = 10;

expr = a || b || c;

expr = ((a || b) || c)

expr = ((50 || not eval) || c)
= (1 || c)
= 1

expr = a || b && c
= a || (b && c)
= (a || (b && c))   A || B

= 50 || not evaluated
= true

expr = a && b || c
= (a && b) || c --> A || B
= (50 && -50) || c
= true || not evaluated
= 1

expr = a || b || c && d
= a || b || (c && d)
= (a || b) || (c && d) --> A || B
= (50 || not) ||
= true || not evaluated

expr = a && b && c || d
= ((a && b) && c) || d --- a || b
= ((50 && -50) && 0)
= false || 10
= true

1. x=y=z=1;

++x || ++y && ++z;

x=2, y=1 , z=1

++x || (++y && ++z);

2 || not evaluated

x = 2

2. x = y = z = 1
++x && ++y || ++z
x = 2, y = 2, z = 1
++x && ++y && ++z
x = 2, y = 2, z = 2

3. x=y=1
z =0

z++ || ((++y && ++z) && ++z)

z = 1 y = 2 z = 3

x = 1

4. x=y=z=-1

++x || (++y && ++z)
x = 0 y = 0 z = -1
++x && ++y || ++z

x = 0, y = -1 z = 0

1. num = 0;

!num = 1

num = 100;
num = !!!!num;
= !!!(!num)
= !!(!(!num))
= !(!(!(!num)))
= !(!(!(0)))
= !(!1)
= !0 = 1

R to L

num = 0;
num = !num++; ==> !(num++)

num = !(num++)

undefined

num = 1

num = !++num;
num = !(2);
num = 0

undefined behavior -- unexpected output

num = ++!num; -- compile time error

num = !num || num++;

num = !num || !num && ++num || num++;

num1 = 0.7;   float

1. 0.7

   Binary

   $0.1\overline{0110}$

.7 * 2 = 1.4
.4 * 2 = 0.8
.8 * 2 = 1.6
.6 * 2 = 1.2
.2 * 2 = 0.4

.4 * 2 = 0.8
.8 * 2 = 1.6
.6 * 2 = 1.2
.2 * 2 = 0.4

2. 1.0110

   y = -1

3. exponent = 127 - 1 = 126 = 0111 1110
   1023 - 1 = 1022 = 01111111110

0 01111110 0110 0110 0110 0110 0110 011   float

0 01111111110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110   double

Notes:
1. IF expression has more than one operator, then check th precedence
2. If precedence is same or same operator is repeated then check associativity
3. if the operands of different types, then type conversion (implicit, explicit)

-1
0000 0000 0000 0000 0000 0000 0000 0001
1111 1111 1111 1111 1111 1111 1111 1110
+   1
1111 1111 1111 1111 1111 1111 1111 1111

10 > 4294967295

10 > -1

=, +=, -=, *=, /=
+= num = num + 10;

num1 += num2 += num3 += num4; R to L

1

2

3

int num1 = 1, num2 = 1;
float num3 = 1.7, num4 = 1.5;

num3 += num4

num3 = num3 + num4;
num3 = 3.2

num2 += num3;
num2 = num2 + num3;
= 1 + 3.2
= 1.000000 + 3.2
= 4.2
num2 = 4 (type demotion)

num1 += num2
num1 = num1 + num2;
num1 = 1 + 4
num1 = 5

Bitwise

---> perform on bits
--> &, |, ^, <<, >> ~

Bitwise &

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR -- |

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise XOR -- ^

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

when 2 input has same bit/value
then result is 0

1. char x = 0x61, y = 0x13

x & y ---> 0110 0001
0001 0011
~~0000 0001~~ ---> 0x01 -- 1

x = 0xAA y = 0x57

x & y = 1010 1010
0101 0111
0000 0010 -- 0x02, 2

x | y --> 0110 0001
0001 0011
0111 0011 --> 0x73, 115

x | y = 1010 1010
0101 0111
1111 1111 --- FF, 255

2. int x = 10, b = 15;

0000 0000 0000 0000 0000 0000 0000 1010
0000 0000 0000 0000 0000 0000 0000 1111

Bitwise XOR - ^

x = 0xBC, y=0x35

x ^ y ---> 1011 1100
0011 0101
1000 1001 --> 0x89, 137

Bitwise complement --> ~
--> 1's complement -- 1 --- 0
0 --- 1
--> unary operator

x = 0x43 signed
~x == 0100 0011
1011 1100 --> 0xBC

Shift

1. Left shift -- <<
2. right shift -- >>

1. Left shift -- <<

--> bits will be shifted to the left side depending on the number of shift

syntax:

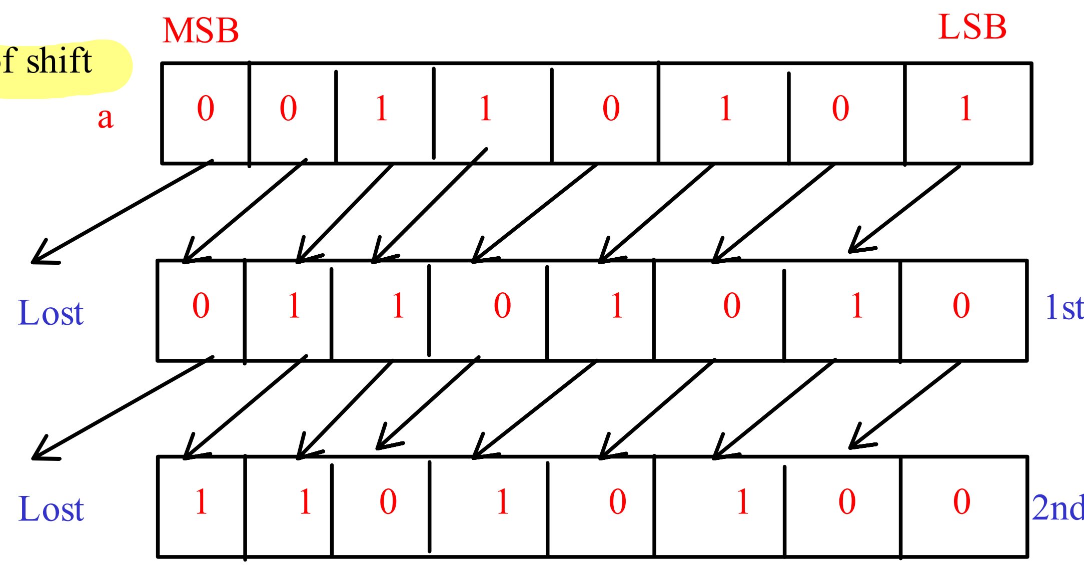
result = value << number of shift

1. char a = 0x35, res;

res = a << 2;

In left shift MSB bit will be lost and LSB is filled with zeroes

res = D4 -- 212



--> Effecient way of multiplying 2 power values

0x35 == 53 === 53 * 2^0 = 53 * 1 = 53
0x6A == 53 * 2^1 = 53 * 2 = 106
0xD4 == 53 * 2^2 = 53 * 4 = 212

0x35 = 3 * 16^1 + 5*16^0

general = value * 2^(number_of_shift)

53 * 2^2 = 53 * 4 = 212 == D4

2. 0x16 << 3

0x16 ===> 22

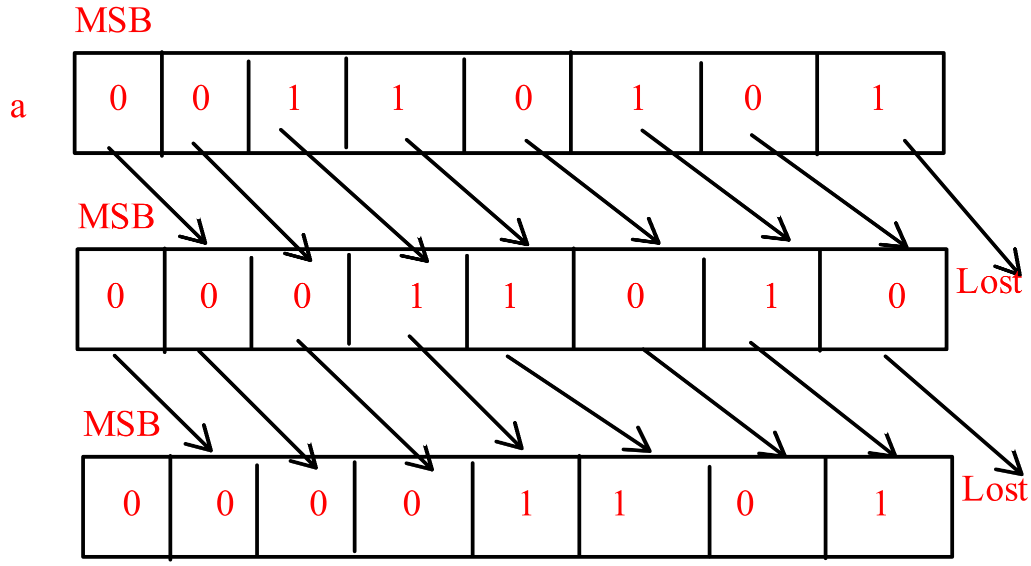
result = 22 * 2^3 = 22 * 8 = 176
== 0xB0

2. Right shift -- >>

--> bits will be shifted to the right side depending on the number of shift

1. Unsigned right shift

--> only positive value



unsigned char ch = 0x35;
ch >> 2

in case of unsigned right shift,
LSB bit will be lost,
MSB is filled with 0

result = 0x0D

right shift is the effecient way of dividing the given value by 2 power values

resultant = value / 2^(number_of_shift)

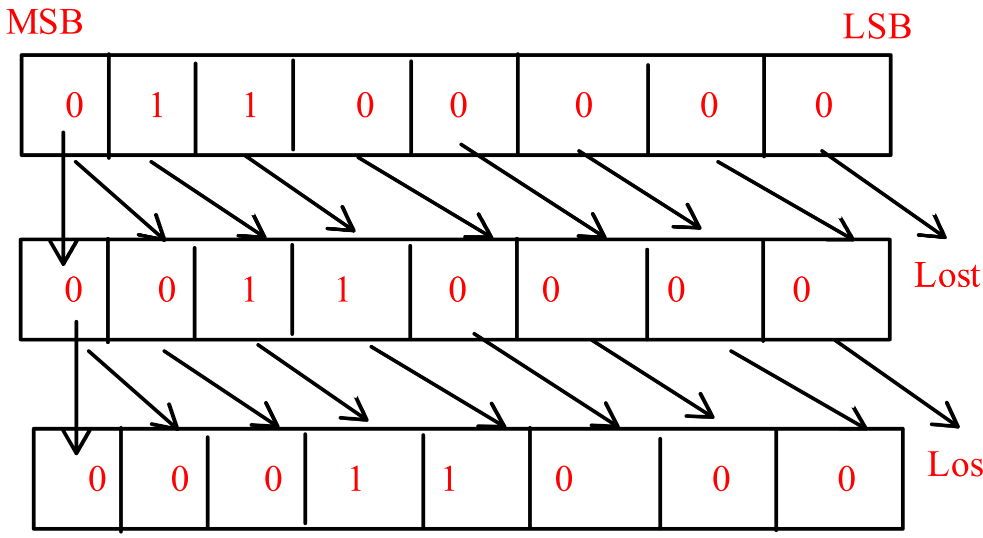
0x35 === 53
53 / 2^2 = 53 / 4 = 13 ---> 0x0D

2. Signed right shift

--> both positive and negative

signed char ch = 96;

ch >> 2
for signed, LSB is lost
MSB is filled with previous
MSB bit



result = 24

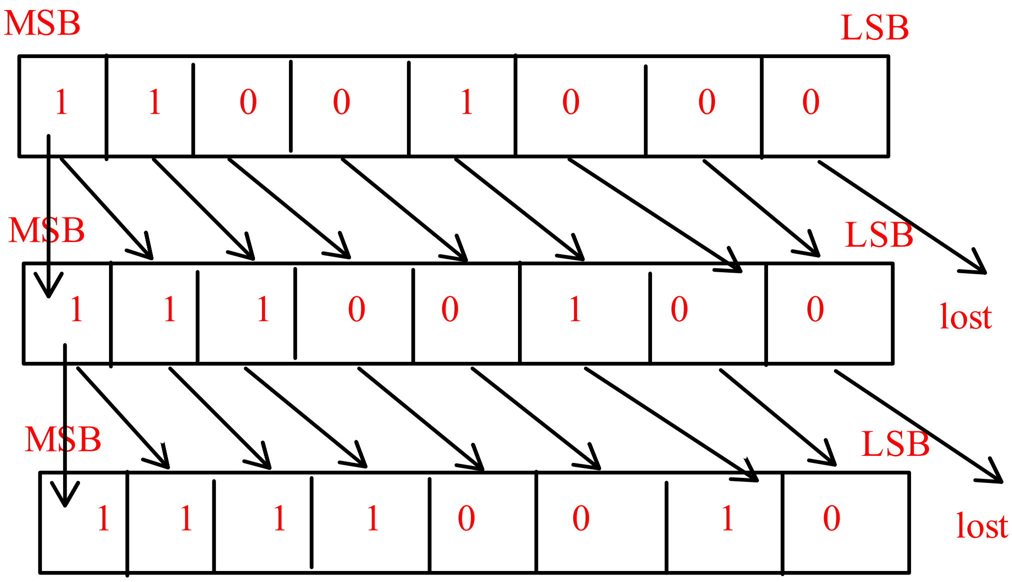
res = 96 / 2^2 = 96 / 4 = 24

signed char ch = -56;

ch >> 2

00111000
11000111
+ 1
11001000

result = -14



1111 0010
00001101
+ 1
00001110 -- -14

result = -56 / 2^2 = -56 / 4 = -14

Notes:
1. Number of shifts should be positive value
val << num, val >> num
else, it will result in undefined behaviour

2. right operand (number of shift) has to be within the width of left operand
e.g, if val is char, then the valid number of shift is 8 times -- 0 to 7
if val is int, 0 to 31
more than this it results in undefined behaviour

3. signed char ch = 96,
int res;

res = ch << 4;
= 96 * 2^4 = 1536
res = 1536

--> after shifting if the resultant value is not within the range of data type then it will be undefined behaviour

%d --- signed int
%u --- unsigned val

```
int count;
unsigned char iter = 0xFA;
for (count = 0; iter != 0; iter >>= 1)
{
    if (iter & 01)
    {
        count++;
    }
}
printf("count is %d\n", count);
```

1. count = 0
1111 1010
00000001
00000000

iter = iter >> 1
iter = 0111 1101

2. 0111 1101
00000001
00000001 -- true

count = 1
iter = 0011 1110

3. 0011 1110
00000001
00000000

iter = 0001 1111
00000001
00000001
count = 2

Why bitwise?

1byte --- 8 bit ---

--> set bit, clear bit, toggle bit......

1. set bit

----> Whatever the bit might be, make it 1
----> need a bitwise operator --- and or, xor, shift
----> Whenever a set bit asked, you have to use bitwise or operator

char value = 0xAA

---> set the bit at position 4

	7	6	5	4	3	2	1	0	LSB
value	1	0	1	0	1	0	1	0	

mask = 0x10
res = value | mask

|

1 0 1 0 1 0 1 0
0 0 0 1 0 0 0 0

1 0 1 1 10 1 0

1 0 1 0 1 0 1 0
0 1 0 0 0 0 0 0 mask = 0x40
1110 1010

pos = 0, 1, 2, 3, 4

generic mask --- genaral mask works with all the values

generic mask = 1 << pos

set bit = value | mask

0 --> 0000 0001
1 ---> 0000 0010
2 ---> 0000 0100
3 ---> 0000 1000
4 ---> 0001 0000
5 ---> 0010 0000
6 ---> 0100 0000
7 ---> 1000 0000

set 3-bits from lsb of the given value

0xAA 1 0 1 0 1 0 1 0 LSB

| 0 0 0 0 0 1 1 1 ---> 0x07

2bits from LSB
0000 0011 --> 0x03

generic mask = (1 << number of bits) - 1

1 << 3
0000 1000
8 - 1 = 7
0 000 0111

1 << 2
0000 0100
4-1 = 3
0000 0011

1. bitwise -- OR, AND, XOR
2. mask

1. set bit --- whatever the bit may be make it 1

value = 0xA3; set bit in the position 4

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

1010 0011
| 0001 0000 --- mask

1011 0011

0 | 1 === 1
1 | 1 === 1

0 | 0 == 0
1 | 0 == 1

1010 0011
0000 0100 --- mask

0000 0001
0000 0010
0000 0100
0000 1000
0001 0000
0010 0000

position ---

generic mask = 1 << position

value | mask

3-bits from LSB

1010 0011
0000 0111 --- 7

1010 0011
0000 1111 --- 15

1 << 3
0000 1000 -- 8 - 1 = 7

1 << 4
0001 0000 -- 16-1 = 15

generic mask = (1 << number 0f bits) - 1

value | mask

2. clear bit

---> whatever the bit may be make it 0

value = 0xAA

--> clear the bit at position 3

---> Whenever clear bit is asked, bitwise AND is the solution

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

0 & 0 == 0
1 & 0 == 0

1010 1010
&1111 0111   F7

0 --> 1111 1110
1 --> 1111 1101
2 --> 1111 1011
3 --> 1111 0111
4 --> 1110 1111
5 --> 1101 1111

generic mask = ~(1 << position)

1010 1010

1 << 3
~0000 1000
1111 0111

value & mask ===> clear bit

clear 3 bits from LSB

1111 1000
1111 1100
1111 0000

generic mask = ~((1 << number of bits) - 1)

1 << 3
0000 1000 - 1
~0000 0111
1111 1000

3. Toggle bit --- xor -- ^

--> 1 -- 0
0 -- 1

0 ^ 1   1
1 ^ 1   0

value = 0xAA

1010 1010
0000 1000
1010 0010

1010 1010
0000 0100
1010 1110

mask = 1 << position

value ^ mask

get bit
--> return the same bit from the given position

0 --> 0
1 --> 1

value = 0xAA

1010 1010
& 0000 0000
0000 0000

1010 1010

unsigned value

value >> position

value >> 2
0010 1010
& 0000 0001
0000 0000 ---> 0

value >> 3
1010 1010
0001 0101
& 0000 0001
0000 0001 ---> 1

--> mask = (1 << position)

value & mask

num = 0xAA;

1 0 1 0 1 0 1 0

1. i = 7
num >> 7
1010 1010
0000 0001
0000 0001
1

num >> 6

1010 1010
0000 0010
0000 0001
0000 0000

num >> 5

10101010
00000101
00000001

0xAA

1010 1010

>> 1 0 1

1010 1010 >> 7
0000 0001
0000 0001
0000 0001

1010 1010 >> 6
0000 0010
0000 0001
0000 0000

1010 1010 >> 5
0000 0101
0000 0001
0000 0001

swap nibble of a byte

nibble ---> 4bits

0xAB ---> swap 0xBA

value >> 4

value << 4

value << 4 | value >> 4

unsigned

1010 1011 >> 4
0000 1010 --> a

1010 1011 << 4
1011 0000 --> b

b | a   a | b

---> 1010 1011 >> 4
1111 1010 ---> fa

B , A

(value & 0x0F) << 4 --> 1010 1011
0000 1111
0000 1011 --> 0B

(value & 0xF0) >> 4 ---> 1010 1011
1111 0000
1010 0000 --> A0

Ternary operator ---> 3 operands

expression/condition ? true_expression : false_expression;

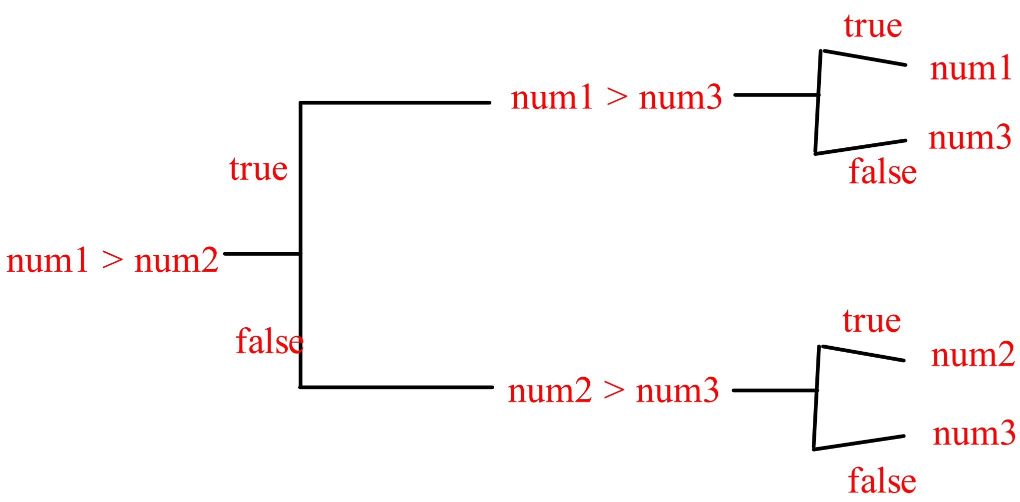
--> same as if else with sing statement

```
if (num1 > num2)
{
    printf("num1 is max\n");
}
else { printf("num2 is max\n")
}
```

===> num1 > num2 ? printf("num1 is max\n") : pf("num2 is max");

```
if(num1 > num2)
{
    if(num1 > num3)
    {
        max = num1;
    }
    else
    {
        max = num3;
     }
}
else
{
      if(num2 > num3)
      {
            max = num2;
       }
     else
      {
           max = num3;
       }
}
```

max= num1 > num2 ? (num1 > num3 ? num1 : num3) : (num2 > num3 ? num2 : num3);



j = 1 > 1 ? 2 && 3 : 4 ? 5 % 6 : 7 : 8;

?: % &&

= 1 > 1 ? (2 && 3) : 4 ? (5 % 6) : 7 : 8;

2 1

R to L

= 1 > 1 ? (2 && 3) : (4 ? (5 % 6) : 7) : 8;

3

= = 1 ? (1 ? (2 && 3) : (4 ? (5 % 6) : 7)) : 8;

4

= (1 ? (2 && 3) : (4 ? (5 % 6) : 7))

= (2 && 3)

j = 1

Comma Operator

int num1, num2, num3; ---> separator

num = (1,2,3);   //right most value is returned

num = 3;   operator

num1 = (x=1+2, y=2+2, z=3+3);

x = 3 y = 4 z = 6

num1 = 6

int x = 1, y = 2;

num1 = (x, x+y);

int i = 0, j = 0;

j = i++ ? i++ : ++i;   0++ ?   i = 1

printf(%d %d, i, j);   ++i   i = 2

j = 2

j = i++ ? i++, ++j : ++j, i++;   ++ , ?:

= (i++ ? (i++, ++j) : (++j)), i++;

= (2++ ? (i++, ++j)   i = 3

j = 3, i++;   i = 4

j = 3

i = 5

num = (1,2,3)

num = 1,2,3;   //separator

num = 1;
2;
3;

j = a, b

expr ? expr1 : expr2;

int i, j;

for(i = 0, j = 0; (i < 5, j < 10); i++, j++)
{
}

printf("%d %d", i, j);

10, 10

1. i =0, j = 0
   0 < 5 - t, 0 < 10

2. i = 1, j = 1
   1 < 5 -- t, 1 < 10 -- t

3. i = 2, j = 2
   2 < 5, 2 < 10

4. i = 3, j = 3
   3 < 5, 3 < 10

i = 5, j = 5
5 < 5 - f, 5 < 10

i = 6, j = 6
6 < 5, 6 < 10

i = 7, j = 7
i = 10, j = 10
10 < 5, 10 < 10 -- false
exit the loop

Overflow and underflow

whenever you try to store the value out of range, then underflow or overflow will ocuur

if more than the maximum range --- then oveflow
if less than the minimum range --- underflow

signed char ch = 127;   -128 to +127

ch = ch + 1;   underflow <   > overflow

128   1000 0000

-k = 2^n -k

2's   0111 1111
               + 1
           1000 0000 ---> -128

where n is number of bits
k is given value

ch = -129;   1000 0001

256 -129 = 127

0111 1110
+   1
0111 1111 ---> + 127

Arrays

--> collection of homogeneous type of data
--> collection of same type of data
--> huge datatype which can store more than one value
--> array will have fixed size

syntax:

datatype array_name[size];

int arr[5];
char carr[5];
float farr[5];
double darr[5];

int age[100]

	[0]	[1]	[2]	[3]	[4]	
arr	GV	GV	GV	GV	GV	
	1000	1004	1008	1012	1016	-- assume

int arr[5];

total memory = size * sizeof(dtatype of array);

index = 0 to size-1

0 to 4

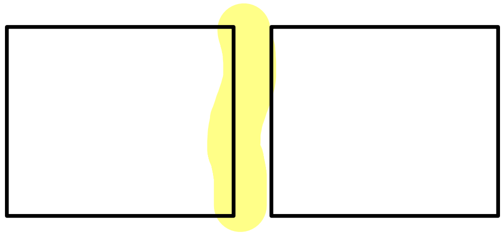
memory = 5 * sizeof(int)
       = 5 * 4
       = 20bytes

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

	[0]	[1]	[2]	[3]	[4]	
arr	1	2	3	4	5	
	1000	1004	1008	1012	1016	valid legal

---> values of array is fetched with the index

arr[0] ---> 1
arr[1] ---> 2
arr[2] --- 3
arr[3] --- 4
arr[4] --- 5



illegal

arr[5] ---> run time error -- logical error

for loop -- size

```
for(i = 0; i < size; i++)
{
    printf("%d\n", arr[i]);
}
```

different ways of declaring an array

1. int arr[5] = {10, 20, 30, 40, 50};

2. int arr[5] = {10, 20, 30};

partially initialised array

	[0]	[1]	[2]	[3]	[4]
arr	10	20	30	0	0
	1000	1004	1008	1012	1016

remaining will be initialised by 0

3. int arr[5];

	[0]	[1]	[2]	[3]	[4]
arr	GV	GV	GV	GV	GV
	1000	1004	1008	1012	1016

4. Without size

int arr[] = {10, 20, 30};

total memory = number 0f elements * sizeof(array_datatype);

memory = 3 * 4 = 12bytes

	0	1	2
arr	10	20	30
	2000	2004	2008

5. int arr[];
   compile time error

int arr1[5] = {1,2,3,4,5}

int arr2 = arr1;

not possible
error

	[0]	[1]	[2]	[3]	[4]
arr1	1	2	3	4	5
	1000	1004	1008	1012	1016

	[0]	[1]	[2]	[3]	[4]
arr2	1	2	3	4	5
	2000	2004	2008	2012	2016

---> name of the array will return the base address(starting address) of the array
---> cannot modify the memory address of array

Reverse elements of the given array

	[0]	[1]	[2]	[3]	[4]
arr2	5	4	3	2	1
	2000	2004	2008	2012	2016

i = 0 swap j = size-1

5 4 3 2 1

incremented decremented

```
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
j--;
i++;
```

```
for (i = 0, j = size-1; i < j ; i++, j--)
{
    temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

1. i = 0, j = 4
   temp = 1
   arr[0] = 5
   arr[4] = 1

2. i = 1, j = 3
   temp = 2
   arr[0] = 4
   arr[3] = 2

i = 2, j = 2

2 < 2   false

3, 4, 5 2 1

1 2 5 4 3

i 0   j 4

3 4 5 2 1

temp = arr[0]

arr[0] = arr[4]   arr[4] = temp

1 4 5 2 1

1 4 5 2 3

```
for (i = 0; i < (size / 2); i++)
{
    temp = arr[i];
    arr[i] = arr[size-i-1];
    arr[size-i-1] = temp;
}
```

```
for( j=size-1;j>=0;j--)
{
    printf("%d\n", arr[j]);
}
```

---> Bubble sort: ascending or descending order

arr [5] = {5, 4, 3, 2, 1}; //ascending

1. 5 > 4 -- true -- swap	4,5,3,2,1
2. 5 > 3 - true -- swap	4,3,5,2,1
3. 5 > 2 - true -- swap	4,3,2,1,5
4. 4 > 3, 4 > 2, 4 > 1	3,2,1,4,5
5. 3 > 2, 3 > 1	2,1,3,4,5
6. 2 > 1	1,2,3,4,5

```
int swap = 0;
for(i = 0; i < size-1; i++)
{
   for(j = 0; j < size-i-1;j++)
   {
        if(arr[j] > arr[j+1])
        {
             temp = arr[j];
             arr[j] = arr[j+1];
             arr[j+1] = arr[j];
             swap = 1;
        }
   }

   if (swap == 0)
      break;
}
```

1,2,3,4,5 ---> sorted

descending

```
if (arr[j] < arr[j+1)
{
   ...
}
```

1. Rotate the array left and right

left rotation

10 20 30 40 50
20 30 40 50 10

right rotation

10 20 30 40 50
50 10 20 30 40

weekly test - 1

Functions

--> block / group of statements

```
{                         if()            for while
    logic                 {
}                         }
```

int num;

How ?

--> it involves 3 steps

1. Function declaration / prototype / signature

--> function is declared before calling
-->Instruction to the compiler that our program contains a function and also tells about the type of input and output

synatx:

```
return_datatype  function_name(arg1_type, arg2_type.......n);

int main()
{
}

int add(int num);          float add(float, float);
int add(int);              float add(float n1, float n2);
```

2. Function definition

---> actual logic / statements to do the specific task

```
function declaration

int main()
{
}

//function definition
return_type function_name(datatype arg1, datatype arg2, ....... n)
{
      logic of task
}

int add(int num)
{
    //logic
    return num;
}
```

3. Function call

--> function will be executed only when you call them

---> inside any other function

```
function declaration

int main()
{
    //call the function
    function_name(arg value....n)
}

function efinition()
{
}
```

```
#include <stdio.h>
//function declaration
int add(int, int);
int main()
{
    int result;
    //call the function
    result = add(10, 20);
    printf("Result is :%d\n", result);   30

    return 0;
}
//function definition
int add(int num1, int num2)
{
    int sum = 0;
    sum = num1 + num2;
    return sum;
}
```

actual arguments / parameters

formal arguments / parameters

10

20

30

30

```c
#include <stdio.h>
// //function declaration
int add(int, int);
int main()
{
    int result, num1, num2;
    //call the function
    printf("Enter 2 numbers\n");
    scanf("%d%d", &num1, &num2);//10 20
    result = add(num1, num2);
    printf("Result is : %d\n", result);

    return 0;
}
//function definition
int add(int n1, int n2)//(2, 4)
{
    int sum = 0;
    sum = n1 + n2;
    return sum;
}
```

calling function

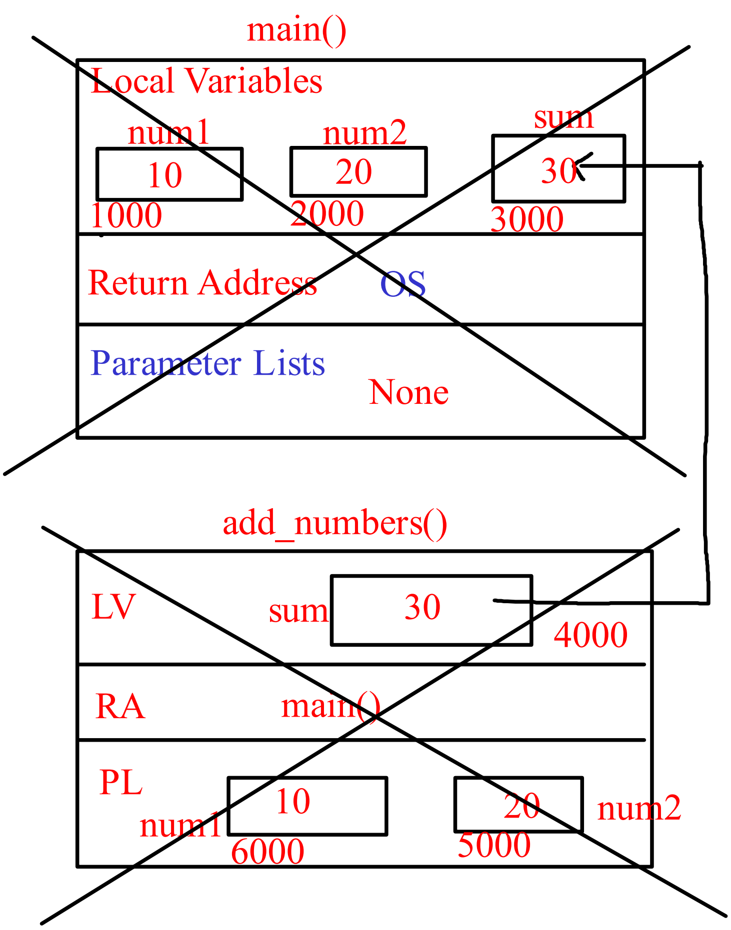
Context switching

called function

Local variables ---> declared within the block
{ }

Return address -- after completing the execution
control should go back to caling

```c
#include <stdio.h>
int add_numbers(int num1, int num2);
int main()
{
        int num1 = 10, num2 = 20;
        int sum = 0;

        sum = add_numbers(num1, num2);
        printf("Sum is %d\n", sum);

        return 0; //successful termination
}
int add_numbers(int num1, int num2)
{
        int sum = 0;
        sum = num1 + num2;
        return sum;
}
```

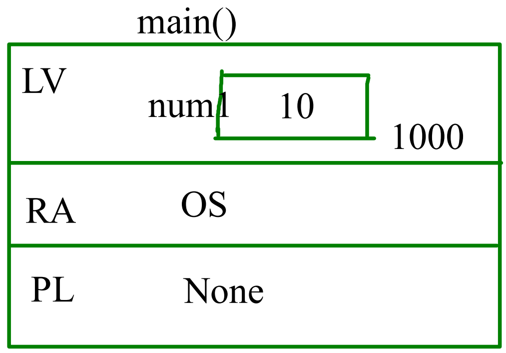


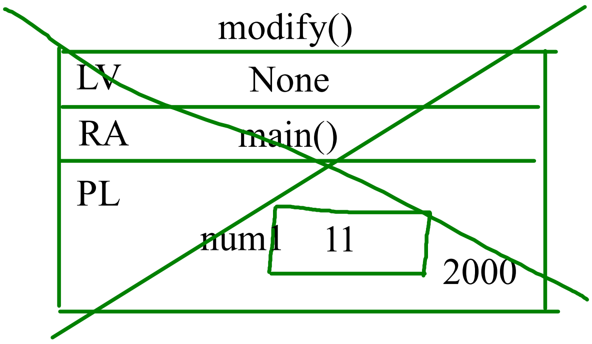
parameter list ---> int func(int num) { }

```
void print_message()
{
    printf("Hello world\n");
}
```

void --- nothing returned

```
#include <stdio.h>
void modify(int num1);
int main()
{
    int num1 = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num1);      10

    modify(num1);

    printf("After Modification\n");
    printf("num1 is %d\n", num1);      10

    return 0;
}
void modify(int num1)
{
    num1 = num1 + 1;
}
```





Pass by value

---> modification done in the function will not be reflected in the actual arguments
---> cannot return the multiple values
---> when u dont need the modified value in original parameter or actual parameter

Types of functions

1. Function without arguments and without a return

```
void  func_name(void)
{
}
```

2. Function with arguments and without return

```
void func_name(int num)
{
    //logic
}
```

3. function without arguments and with return

```
int func(void);
```

4. function with argument and with return

```
int func(int num1)
{
}
```

by default it will consider return type as int and argument type as int

int func(int, int);

implicit int rule

```
func()
{
}
```

```
int func(int)
{
}
```

```
void func(void)
{
}
```

```
void print_message(void)
{
}
```

```
int func()
{
}
```

```
int func(int)
{
}
```

```
int func(void)
{
}
```

```
func(int n)
{
}
```

```
int func(int n)
{
}
```

```
void func(int n)
{
}
```

Pointers

--> variable which holds the address of another variable

datatype var_name;

int num = 90; //value
char ch;
float

syntax:

datatype *pointer_name;

int *ptr;
char *ptr;
float *ptr;
double *ptr;

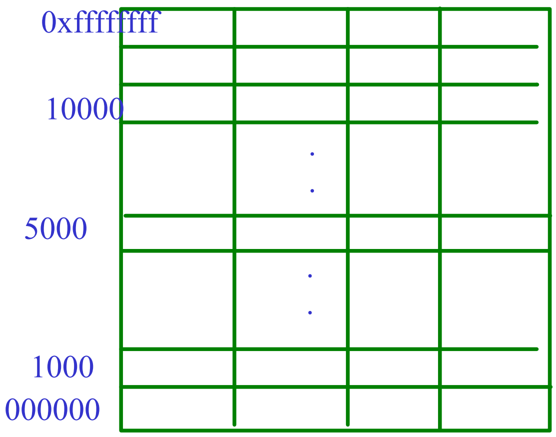
int *ptr = 10; //illegal access

Rule 1 : Pointer is an integer

--> pointer is holding address

---> 100.5
"num"

'a'



32-bit system
4 bytes

64-bit
0xffffffffffffffff
8bytes

int *ptr;

ptr [GV] 1000

int num;

num [ ] 2000
4-bytes

double d; //8

sizeof pointer = system bitness dependant

if 32-bit --- 4bytes
if 64-bit --- 8 bytes regardless of the datatype / modifiers

Rule 2 : Referencing and dereferencing

--> 3 operators ---> *, &, -> (later in the user defined dt)

1. & operator: Referencing

unary operator --- one operand

bitwise & -- binary operator
n1 & n2 --- bitwise

&variable_name ---> fetch the address of the variable

reference means address / memory location

2. * operator --- dereferencing

*pointer_name --- return the value present in the memory / address

int num = 100;

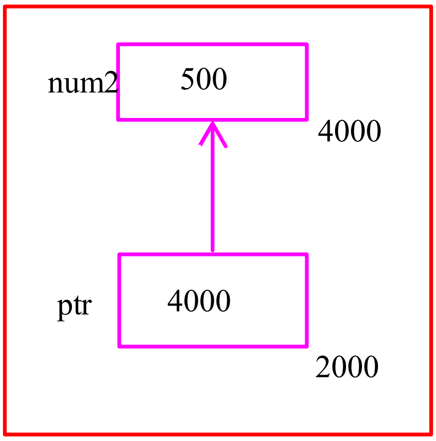
int *ptr;

ptr = #

or

int *ptr = #

int num2 = 500;
ptr = &num2;



int *ptr = #

int num = 100;

invalid -- error

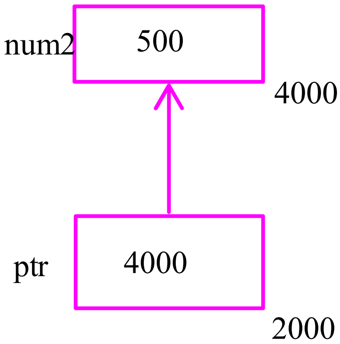
ptr ---> 1000
&ptr ---> 2000

num = 200;
num = 300;

dereferencing --> *

--> *ptr ---> fetch the value from that particular memory

*ptr ---> *4000 ---
instructing the compiler go to memory location 4000
and fetch the value
--- 500



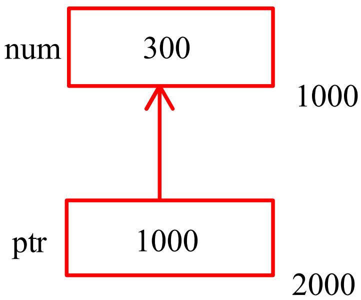
Rule 3 : Pointing means containing

---> whatever the memory location pointer is holding it contains that memory

*ptr = 300;

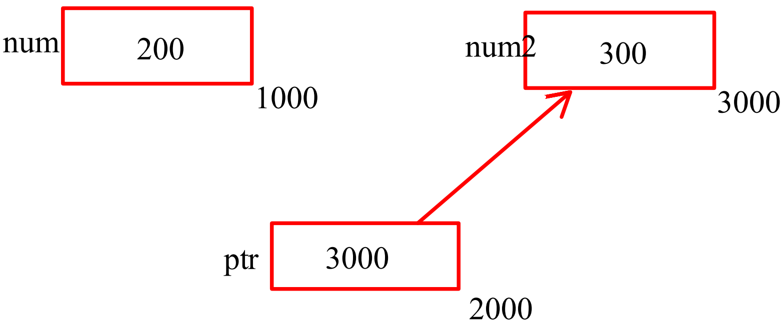
*1000 = 300;

goto memory location 1000 and replace
the value by 300



whatever the modification is done by the pointer will be reflected in the variable and vice versa

```
int num = 200;
   int *ptr = &num;
   int num2 = 100;
   ptr = &num2;
   *ptr = 300;

   printf("num: %d\n", num);      200
   printf("*ptr: %d\n", *ptr);     300
```



Rule 4 : Pointer datatype

int *ptr; ---> 4 / 8

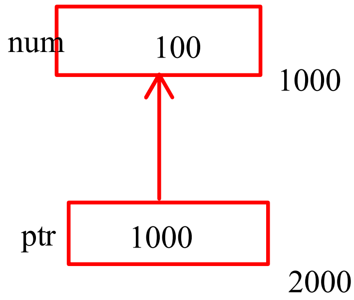
---> dereferencing the pointer how many bytes of datat should be fetched will be known by the datatype

int num = 100;

int *ptr = #

*ptr = *1000 --> 4bytes

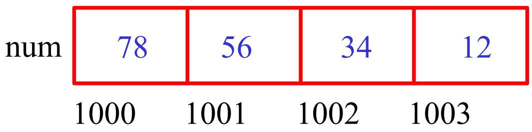
char *ptr; --> 1byte



int num = 0x12 34 56 78;

MSB LSB

little endian system



2 digit in hexa ==== 1 byte

78563412 --- 12345678

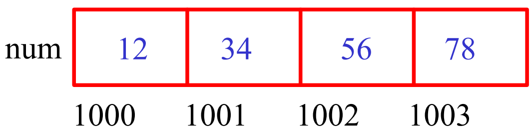
Endianess

---> how data is stored

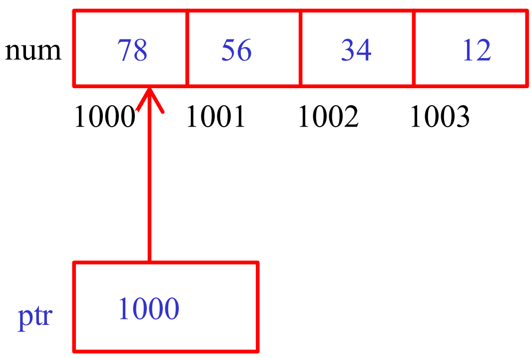
1. Little endian system ---> Lowest Significant Byte will be in the lower address / starting address
most of the processor follows little endian

2. Big endian system
---> Motorola
---> MSB will be in the lower order address/starting address

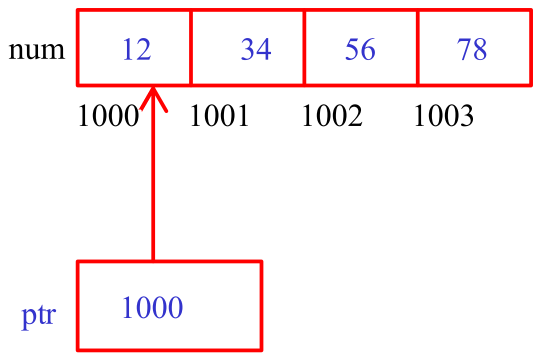
Big endian system



little endian system



Big endian system



char *ptr = #

*ptr ---> 1byte ---> 78

*ptr === 12

int num = 0x56789;

little endian system

num	89	67	05	00
	1000	1001	1002	1003

Big endian system

num	00	05	67	89
	1000	1001	1002	1003

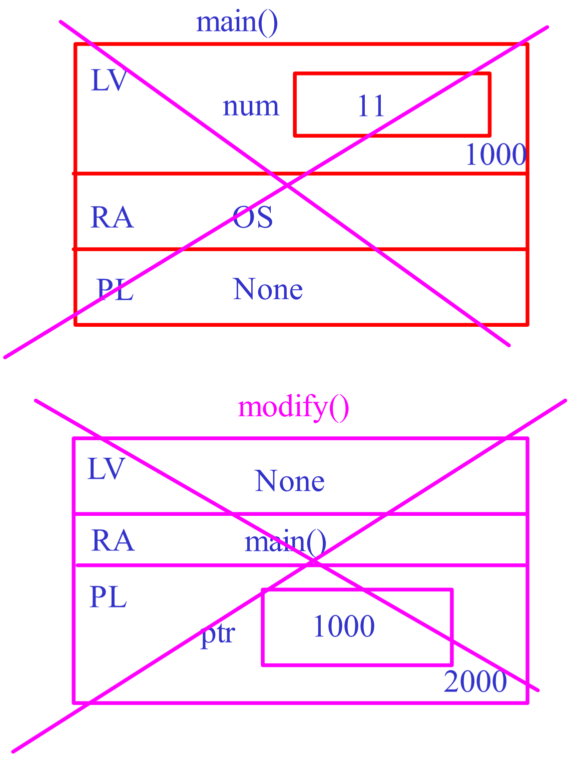
00 05 67 89

Pass by reference

--> pass by value --- calling the function by passing the value
--> calling the function by passing the address of the variable

```
void modify(int *);
int main()
{
        int num = 10;
        printf("Before: %d\n", num);

        modify(&num);
        printf("After: %d\n", num);
}

void modify(int *ptr)
{
        *ptr = *ptr + 1;
}
```

*1000 = *1000 + 1
= 10 + 1 = 11



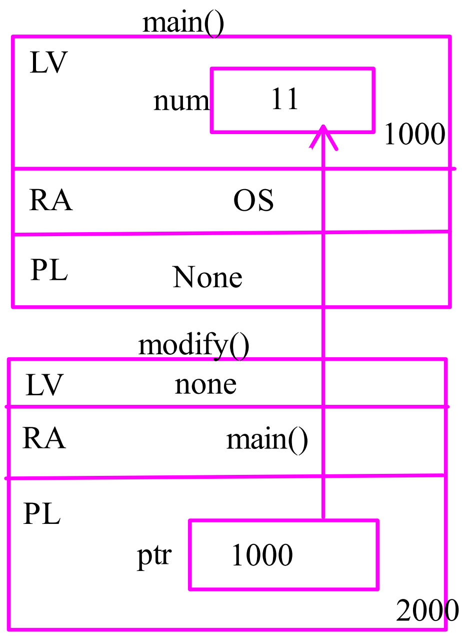
Before : 10

After : 11

---> changes made in the function will be reflected in the actual or original variable

```
void modify(int *);
int main()
{
        int num = 10;
        printf("Before: %d\n", num);

        modify(&num);
        printf("After: %d\n", num);
}

void modify(int *ptr)
{
        *ptr = *ptr + 1;
}
```

address of the num

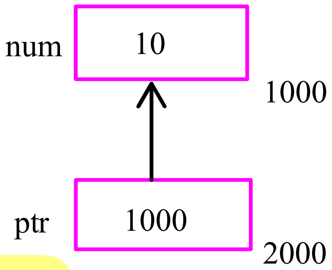


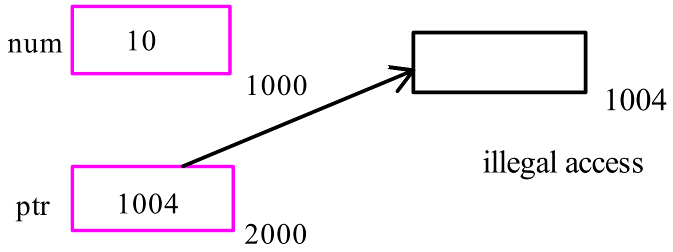
*ptr = *ptr + 1;
= *1000 + 1;
= 10 + 1 = 11

*1000 = 11;
instructing the compiler
goto memory location 1000 and assign the value 11

Rule 5 : Pointer arithmetic

```
---> int num = 10;
     int *ptr = &num;

     ptr = ptr + 1;

     ptr = ptr + 1 * sizeof(datatype of pointer)

     ptr = 1000 + 1 * sizeof(int);
     ptr = 1000 + 4
     ptr = 1004

   ptr++;

   ptr = ptr+1;
```





used in case of arrays --->

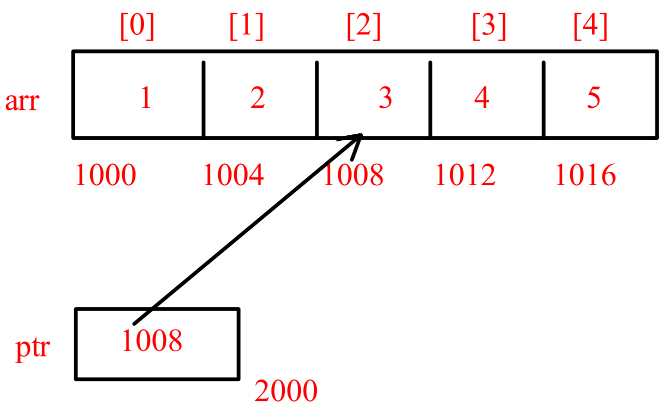
arrays --> collection of similar type of data

```
int arr[5] = {1,2,3,4,5};
int *ptr = arr;
```

sizeof(arr) ---> whole array ---> 5 * 4 = 20bytes

arr ---> base address

&arr -> whole array -- 2d array



*ptr -- *1000 --- 1

ptr++;
ptr = 1000 + 1 * 4 = 1004

*ptr = *1004 ---> 2

```
for (i = 0; i < 5; i++)
{
      printf("%d\n", *ptr++);
}
```

1. i = 0

*(ptr++)
*1000
ptr = 1000 + 1 * 4 = 1004

2. i = 1

*(ptr++)
*1004 == 2
ptr = 1004 + 1 * 4 = 1008

1

2

arr[i] == *(arr + i)

*(ptr+i) ==== ptr[i]

commutative law

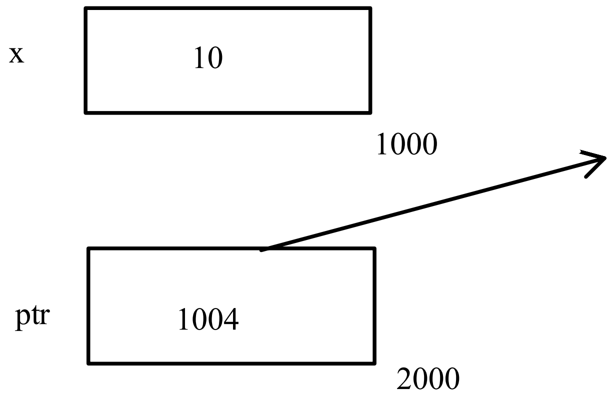
a + b = b + a

*(arr + i) == *(i + arr)
arr[i] == i[arr]

ptr[i] == i[ptr]

1. int x = 10;
   int *ptr = &x;

   *ptr++;
     print   x
     print ptr
     print *ptr

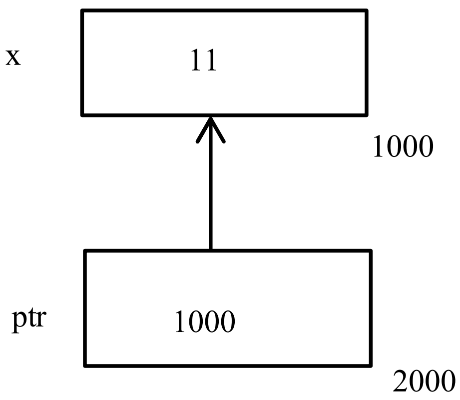


*(ptr++)

*ptr = 10

2. (*ptr)++

   print   x
   print ptr
   print *ptr



*ptr == go to memory location 1000

(*1000)++

++*ptr

Passing array to a function

1. The way you declare an array

```
int arr[5];

void print_array(int arr[5])        //int *arr
{
        //logic                                    print_array(arr);

}
```

2. Without the array size

```
void print_array(int arr[])     //int *arr
{                                          print_array(arr)      //base address
   //logic
}
```

3. Using integer pointer

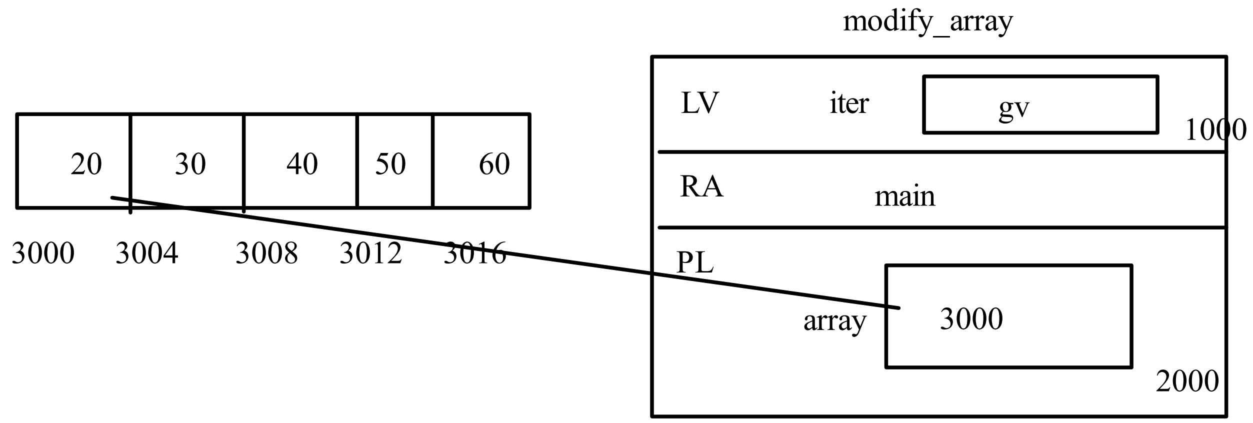
```
void print_arr(int *arr)                        print_array(arr)
{
}
```

4. Pass the size of an array along with the array address
   -- recommended way of passing array to function

```
void print_array(int arr[], int size)
{
}
```

```
int main()
{
   int array[5] = {10, 20, 30, 40, 50};
   printf("Before\n");
   print_array(array, 5);
   modify_array(array, 5);
   printf("AFTER\n");
   print_array(array, 5);

   return 0;
}
```

```
void modify_array(int *array, int size)
{
   int iter;

   for (iter = 0; iter < size; iter++)
   {
      *(array + iter) += 10;
   }
}
```



1. iter = 0

*(array + iter) += 10;
*(3000 + 0) = *(3000 + 0) + 10;

*3000 = 10 + 10 = 20

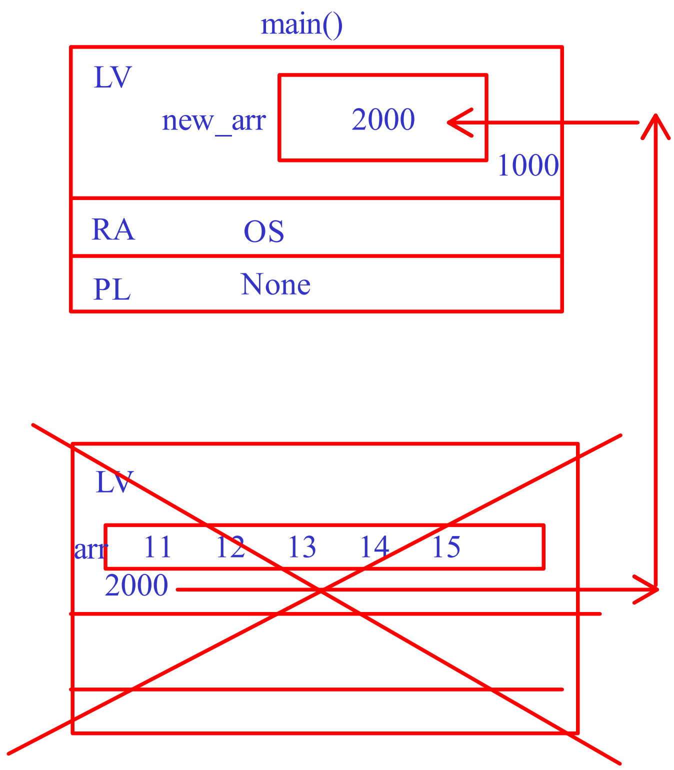
2. iter = 1

*3004 = 20 + 10 = 30

3. iter = 2

*3008 = 30 + 10

```
int main()
{
	int *new_arr;

	new_arr = modify();

	print_array(new_arr, 5);

	return 0;
}
int *modify ()
{
	int arr[] = {1,2,3,4,5}, i;

	for(int i = 0; i < 5; i++)
	{
		arr[i] += 10;
	}
	return arr;
}
void print_array(int arr[], int size) //pass by reference
{
	int i;
	for(i = 0; i < size; i++)
	{
		printf("%d\n", arr[i]);
	}
}
```



static

---> storage class which will instruct the compiler to create the memory in data segment

BSS --- block strated by symbol

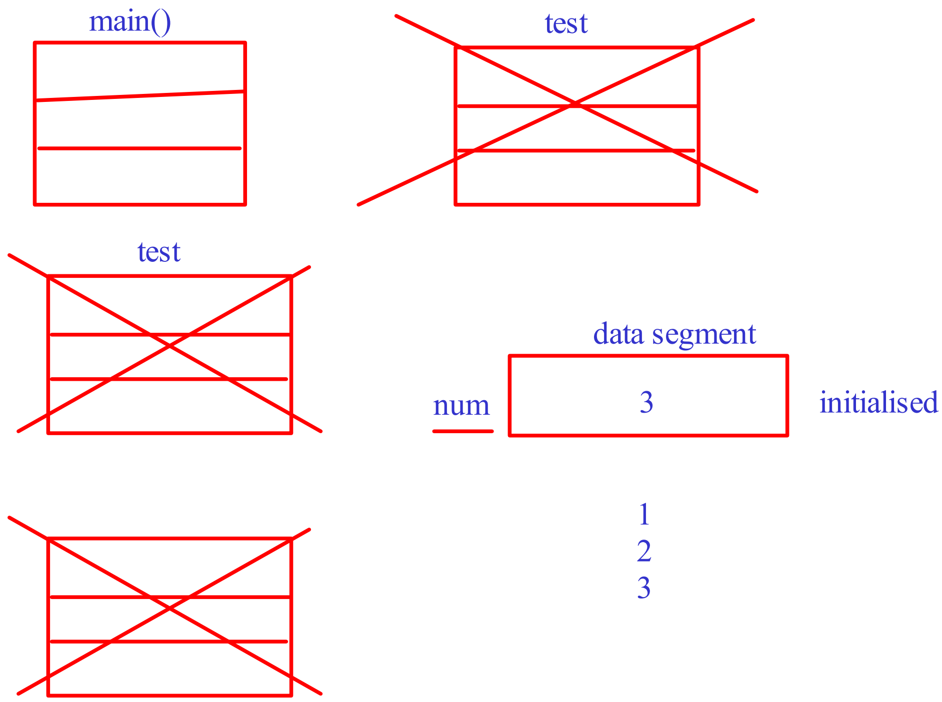
--> static variables are declared without any value then BSS, by defalut it will have 0

initialised

--> declared and initialised with value

--> data segment memory will stay until the program execution (main)

```
int main()
{
    test();
    test();
    test();

    return 0;
}

void test()
{
    static int num = 0;

    num++;
    printf("%d\n", num);
}
```



booking --- 10

user 1 ---> 1 -- 9
user 2 --- 2 -- 7

Recursion

--> calling itself
--> 2 steps:

1. Base condition (exit condition)

2. where to call

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

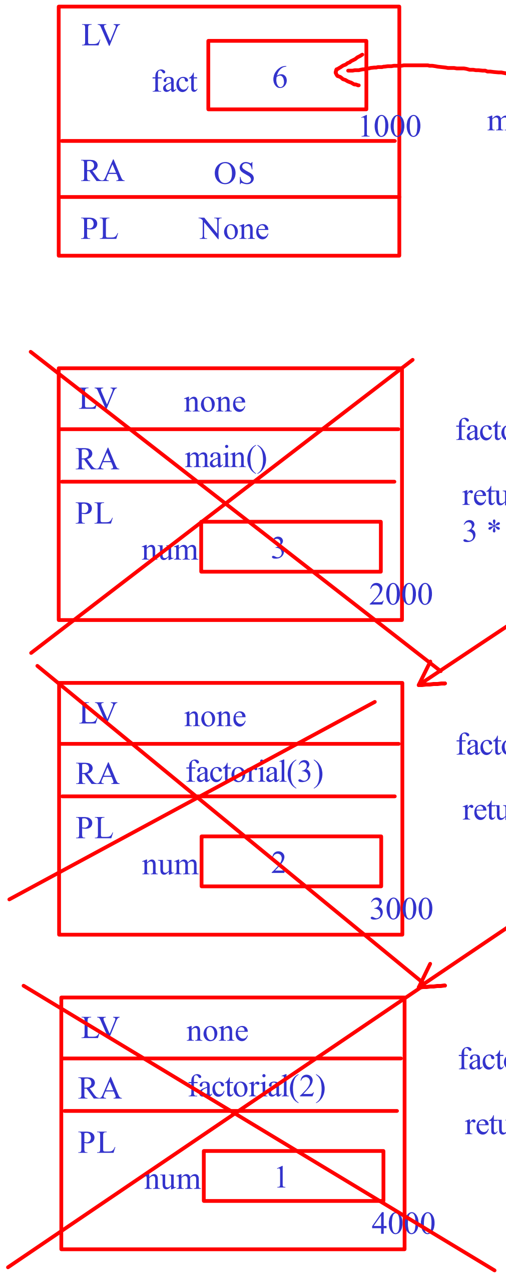
```
void func()
{
    func();
}
```

recursion

factorial of a number -- > 3! ---> 3 * 2 * 1 === n(n-1)!

```
int factorial(int num)
{
    if (num == 1)
    {
        return 1;
    }
    else
    {
        return num * factorial(num-1);
    }
}
```

```
int main()
{

    int fact;

    fact = factorial(3);

}
```



main()

factorial(3)

return 3 * factorial(2)
3 * 2 = 6

factorial(2)

return 2 * factorial(1) == return 2 * 1 = 2

factorial(1)

return 1

```
int factorial(int num)
{
    if (num == 1)
    {
        return 1;
    }
    else
    {
        return num * factorial(num-1);
    }
}
```

When to use recursion vs loop

space

recursion ---> factorial(3) ---> 4 (return value) + 4 paramete = 8 bytes
factorial(2), factorial(1) = 8 + 8 + 8 = 24bytes

factorial(5) = 8 * 5 = 40bytes

loop --> 4 + 4 + 4 = 12bytes

3, 5

--> constraints on the memory/ limitation -- loop
embedded system -- loop

Time

3 stackframe + 3 = 6 machine cycle for recursion

factorial(5)
5 + 5 = 10 mc

loop

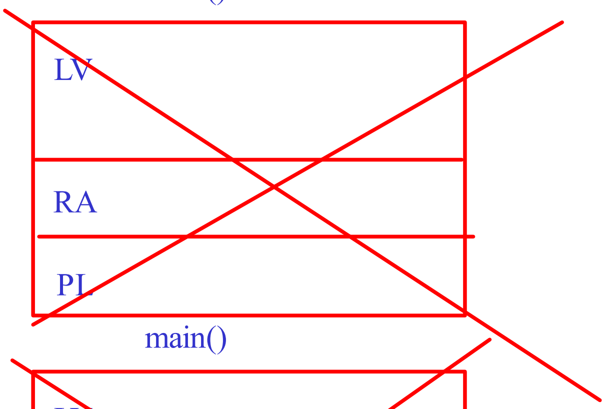
1 stackframe + 1 + 4 condition + 3 operation + 1 + 3 = 13 machine cycle --- factorial(3)

1 + 1 + 6 + 5 + 5 + 1 = 19 mc

recursion is faster than loop

```
int main()
{
	int num = 5;

	if(num-- != 0)
	{
		printf("%d time\n", num);
		main();
	}

	return 0;
}
```

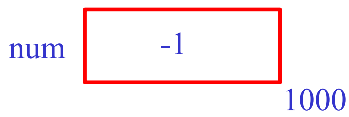
main()



5-- != 0

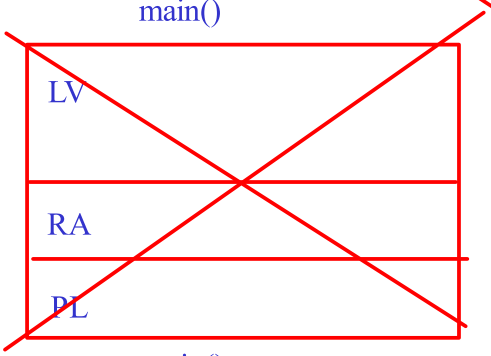
4 time

DS



main is not a true recursive function

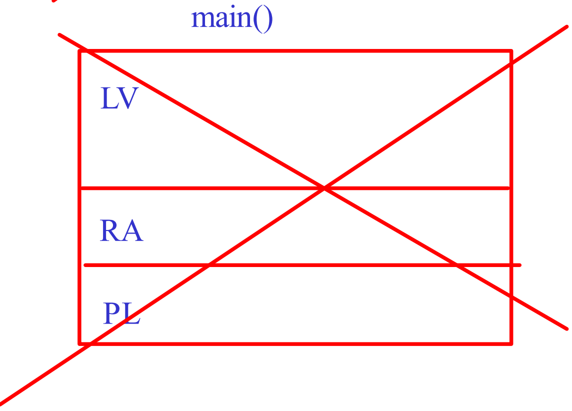
main()



4-- != 0

3 time

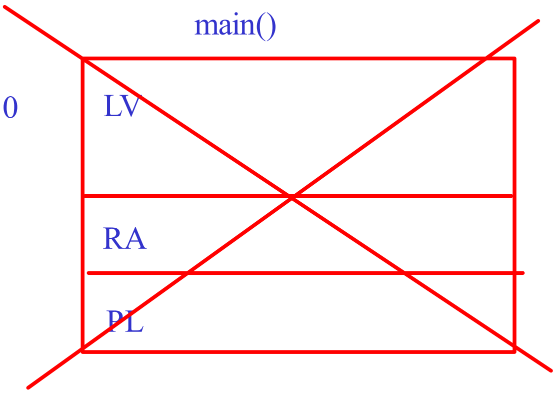
main()



3-- != 0

2 time

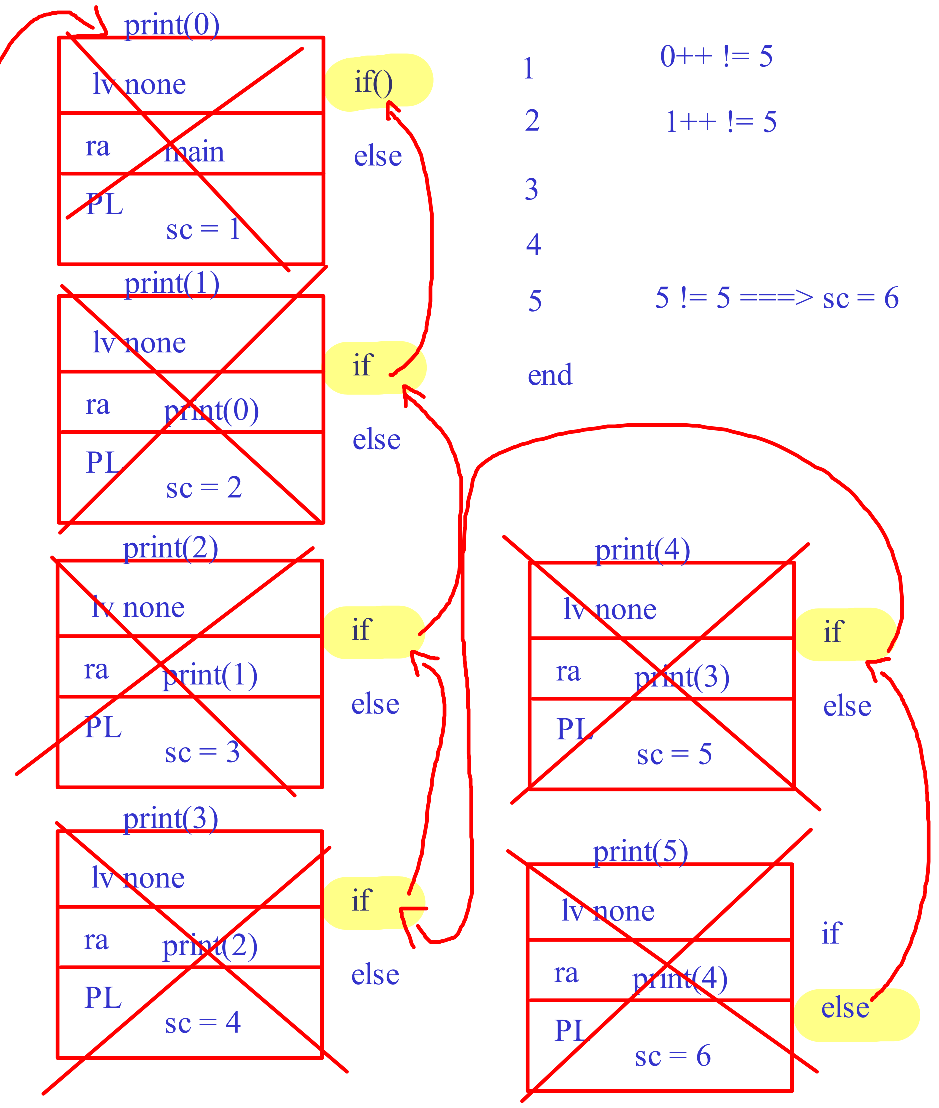
main()



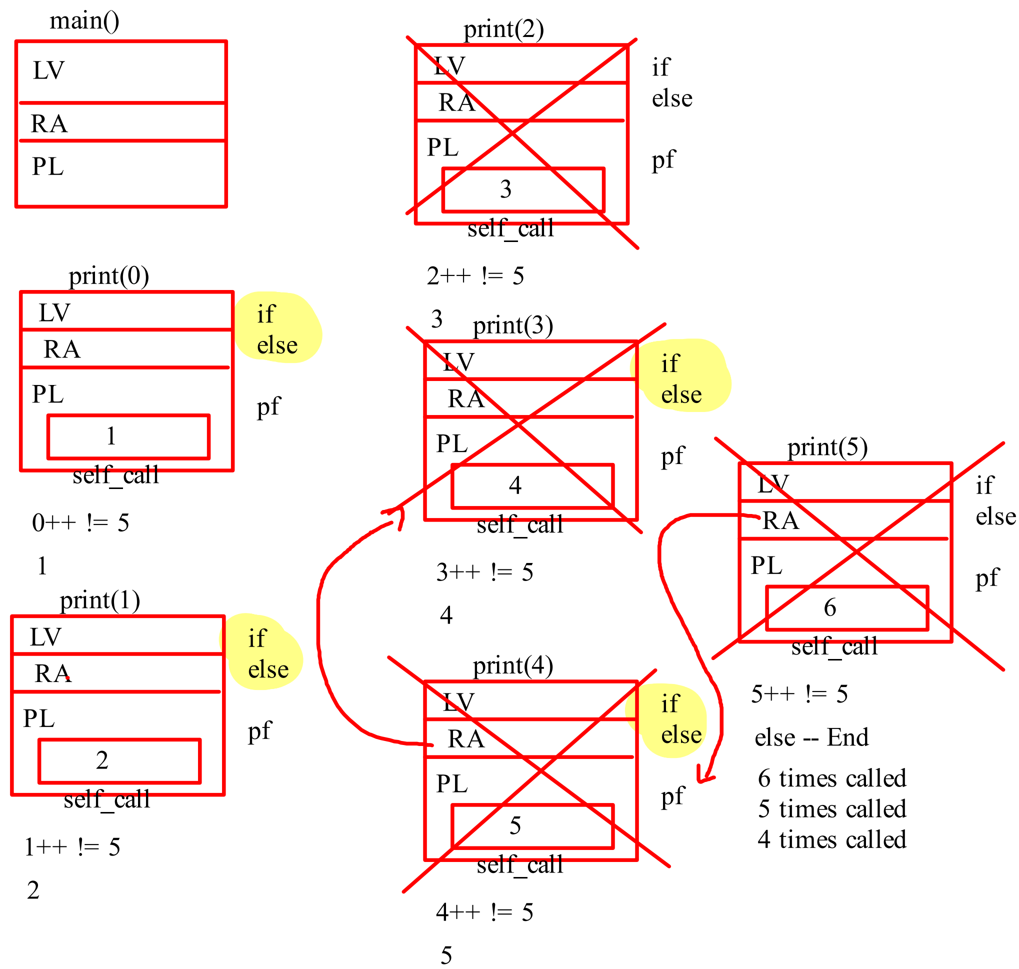
0-- != 0

0 time

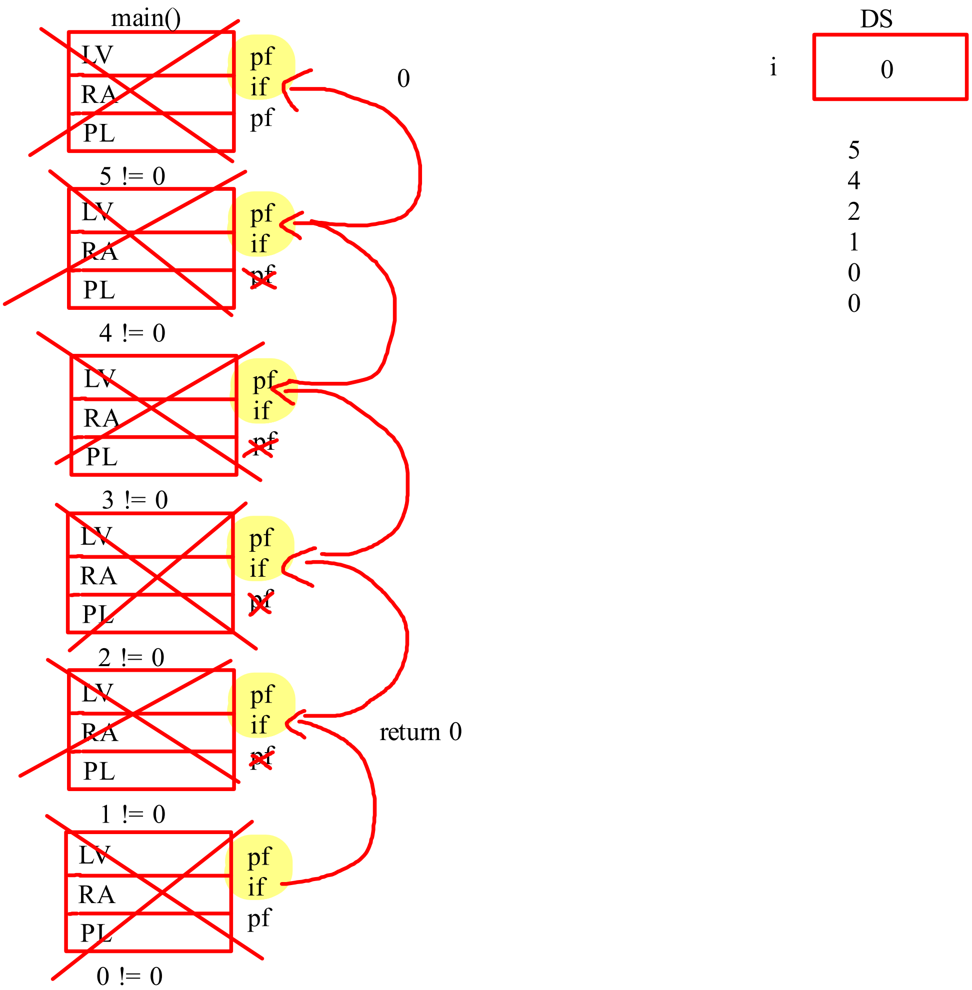
```
void print(int self_call)
{
        if(self_call++ != 5)
        {
                printf("%d times\n",self_call);
                print(self_call);
        }
        else
        {
                printf("End\n");
        }
}
int main()
{
        print(0);

        return 0;
}
```




```
void print(int self_call)
{
        if(self_call++ != 5)
        {
                printf("%d times\n",self_call);
                return print(self_call);
        }
        else
        {
                printf("End\n");
        }
        printf("%d times called\n", self_call);
}
int main()
{
        print(0);

        return 0;
}
```



```
int main()
{
        static int i = 5;

        printf("%d\n", i);

        if (i != 0)
        {
                i--;
                return main();
        }

        printf("%d\n", i);

        return 0;
}
```



return func(); -- exit the function

Strings

Collection of character / array of character terminated by null character

char arr[5] = {'h', 'e', 'l','o'}; --- not string, charcater array

Different ways of decalaring strings

1. char str1[6] = {'h', 'e', 'l', 'l', 'o', '\0'};

size = 6 * sizeof(char) = 6 * 1 = 6bytes

%s --- string, pass base address of the string
str1

	[0]	[1]	[2]	[3]	[4]	[5]
str1	h	e	l	l	o	\0
	1000	1001	1002	1003	1004	1005

2. char str1[] = {'h', 'e', 'l', 'l', 'o', '\0'};

size = 6bytes

	[0]	[1]	[2]	[3]	[4]	[5]
str1	h	e	l	l	o	\0
	1000	1001	1002	1003	1004	1005

3. char str1[6] = {"h" "e" "l" "l" "o" "\0"};

char str1[6] = {"h", "e", "l", "l", "o","\0"};

collection of strings -- 2d array

compile time error

"h" --> "h\0"
"e" --> "e\0"
"h\0" + "e\0" + "l\0" +"l\0" + "o\0" + "\0"
"hello\0"

	[0]	[1]	[2]	[3]	[4]	[5]
str1	h	e	l	l	o	\0
	1000	1001	1002	1003	1004	1005

4. char str1[6] = {"Hello"};

--> implicit null charcater added by the compiler

sizeof(str1);   6 * 1 = 6byte

	[0]	[1]	[2]	[3]	[4]	[5]
str1	h	e	l	l	o	\0
	1000	1001	1002	1003	1004	1005

5. char str1[6] = "Hello";

most commonly used way of string declaration

char str1[] = "Hello";

sizeof(str1) === 5 + 1 = 6 bytes

str1[0] ..... str1[5] -- valid way

Hello 0   undefined

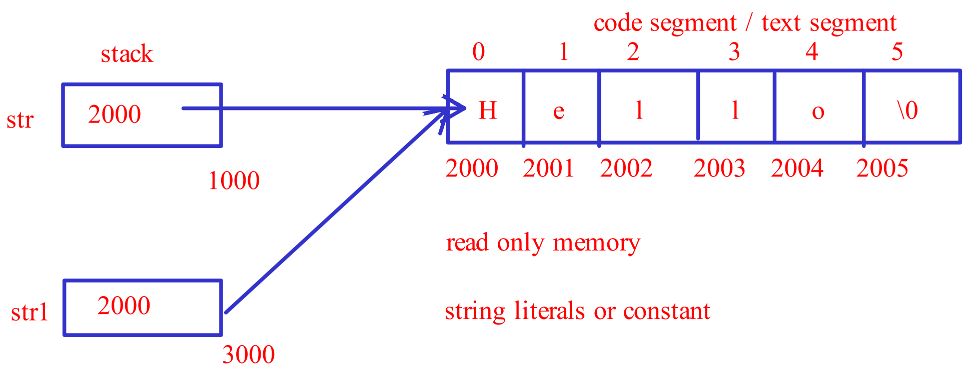
char str= "string"

6. char *str = "Hello";

pointer to an character array

non modifiable string

char *str1 = "Hello";



char *	char[]
1. memory is in code / text segment	1. in stack frame of function
2. Read only , non modifiable string string literals	2. modifiable, array of character
3. sizeof will return the 4/8bytes	3. sizeof array = size * sizeof(char)
4. Memory is shared when 2 pointer has same string	4. not shared

fgets

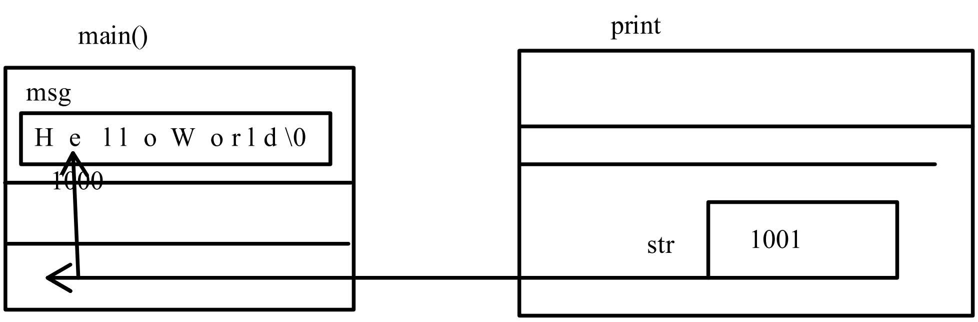
selective scanf

scanf("%10[^\n]", str);

```
char msg[] = "Hello world";

print(msg);
```

```
void print(char *str)
{
    while(*str != '\0')
    {
        putchar(*str);
        str++;
    }
}
```

```
while(*str != '\0')
{
    putchar(*str++);
}
```



*1000 === H != '\0'

Hello World

*1001 === e != '\0'

\0 != \0 -- false

string.h

strlen --- length of the string

sizeof

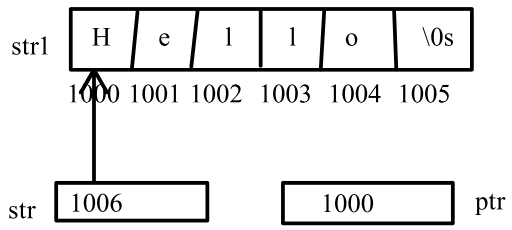
size_t strlen(const char *s);

address of string

user defined datatype
typedef
unsigned int (32 bit)
unsigned long int (64)

s

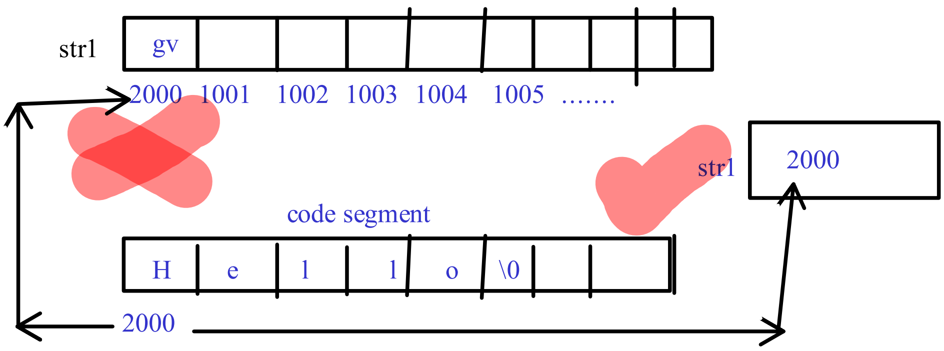
```
size_t my_strlen(const char *str)
{
    char *ptr = str;
    while(*str++);
    return str-ptr-1;
}
```



1006 - 1000 - 1 = 5

```
char str1[] = "Hello"

char str2[] = str1;
```

```
char str1[10];

str1 = "Hello";

puts(str1);
```

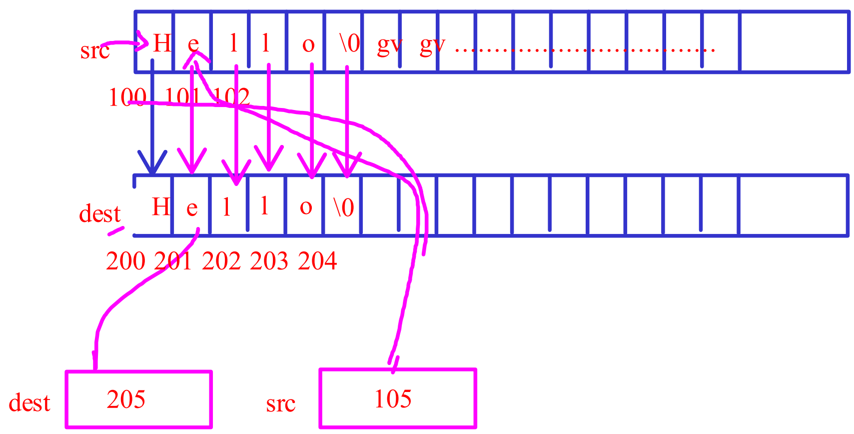


char *strcpy(char *dest, const char *src)

address of destination string

address of source string

address of destination string




```
void my_strcpy(char *dest, char *src)
{
    while(*src)
    {
        *dest = *src;
         src++;
         dest++;
    }
   *dest = '\0';
   //*dest = *src;
}
```

H -- true
e -- true
l -- true
l -- true
o -- true
\0 -- false -- exit the loop

*src != '\0
*src

strcmp -- string compare

--> compare 2 strings

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

integer

string 1 address

string 2 address

if s1 == s2, returns 0
if s1 < s2, returns -ve value
is s1 > s2, returns +ve value

s1 = "RAM", s2 = "RAM"

R == R, 82 == 82, 82-82 = 0
A == A, 65 == 65, 65-65 = 0
M == M, 0
\0 == \0 0 - 0 = 0

0 == strings are equal

s1 = RAM, s2 = ROM

R == R, 82 - 82 = 0
A == O, 65 - 79 = -14

return -14

strings are not equal

s1 = ROM, s2 = RAM

14

strcat -- string concatenation

char *strcat(char *dest, const char *src);

address of destination

destination

source

strcat("Hello ", "good morning");

--> move the pointer of destination till null
---> from src to dest copy character by character until u reach null charcater in the src

substring check

strstr -- to search the given substring in a string

string --- haystack
substring -- needle

char *strstr(const char *haystack, const char *needle);

address of the matched needle in haystack

address of string

address of substring/needle

strstr(haystack, needle);

%s, 106

you

when matching needle is not found then strstr will returns NULL(0 address)
NULL means failure

haystack

H	e	l	l	o		h	o	w		a	r	e		u	?	\0

100 101 2 3 4 5 6 7 8 9 110 11 12 13 14 15 16 17 18 19

needle

h	o	w	\0	

200 201 202 203 204

1. H == h, not equal
2. haystack is incremented, e == h, not equal
3. haystack++, l == h
4. l == h, o == h
6. h == h, equal, haystack++, needle++
7. o == o, equal
8. w == w, equal
9. needle == '\0', stop comparing
10, returns the address of haystack from where the value started same

106 will address

hello! hola how are you?

how

h == h
e == o, not equal
decrement the needle

l == h, l == h, o == h, h == h, o == o, l == w
needle decremented by 2 times
h == h

strtok -- string token

enter your name ---> emer;txe

token = ;

char *strtok(char *str, const char *delim);

address of string

address of token need to be cleared

address of string

strtok("emer;txe", ";");

e == ;, not equal
m == ;, not equal
e == ;, r == ;
; == ;, replace the found delimeter with null charcater in the string

emer\0txe

returns the address from where it started searching for the delimeter

string: hi;'hello:"?bangalore
delim: ;':"?

h == ;, h == ' h == : h == " h == ?
i == ; i == ' i == : i == " i == ?

; == ;

string: hi\0'hello:"?bangalore

delim: ;':"?

' == '

hi\0\0hello\0\0\0bangalore

```
while(ret != NULL)
{
    strtok(NULL, delim);
}
```

//it has to continue the search from previously replaced null charcater
continue the loop until it replaces all the delimeter
compare each charcater of string with character of delimeter

once it reaches the null character in the delim, then returns NULL stdlib.h --- atoi, dynamic memory

atoi --- ASCII to Integer

takes string as a argument and returns integer

char id[10] --> "2434132144"

```
int atoi(const char *nptr);
```

address of string

itoa

integer to ascii

%d as a input, pass integer as a argument and convert it to string

123 === "123"

Storage class

--> sc is a keyword, which will instruct the compiler where should be memory allocated for the variable

1. stack segment (cla)
2. Data segment
3. Heap segment
4. Code / text segment

Register

```
for(i = 0; i < 1000; i++);
```

---> faster accessibility of the variable

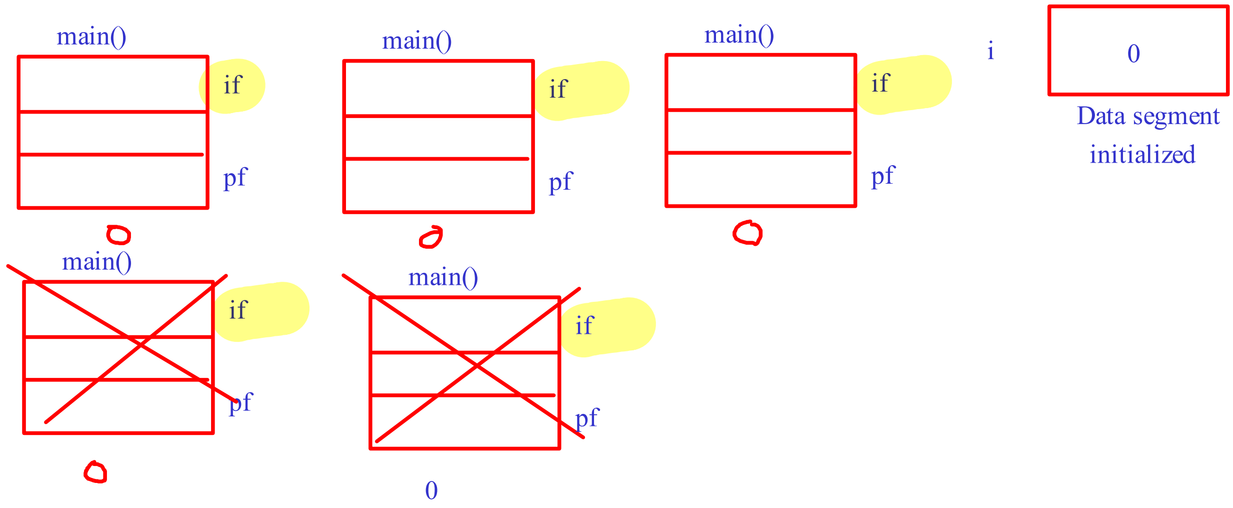
---> register is a closest memory to the CPU
--> directly from register to stdout

--> accessing the address of register variable is not allowed

static

local --
```
{
	static int num;
}
```

```
static int i = 5;

if (--i)
{
        return main();
}

printf("i %d\n", i);

return 0;
```



0

return 0

Global

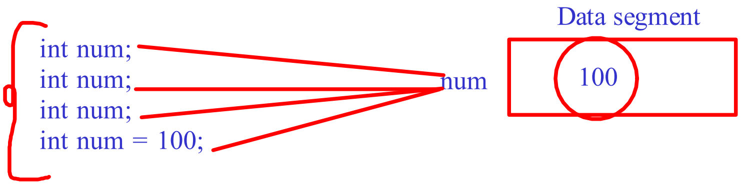
--> variable declared outside the function
--> static and extern
--> memory will be in data segment -- if uninitiliased, then it is BSS, by default value is 0
    if initialised, initialsed block
--> lifetime - program
    static -- file
    extern -- program

one definition rule

---> in global we can have multiple declaration of variable followed by one definition

declaration == int num;

declaration + initialisation = definition



Whenever extern is seen compiler will search in the previous visible decalaration

if previous visible declaration is local then it will ignore it, and goes outside function

gcc file1.c file2.c
a.out

Standard I/O

printf scanf works?

getchar, putchar, getc, putc, sprintf................
buffers

#include <stdio.h> -- angular bracket <> ---> built in headers --- search the file in library path

#include "file.h" -- user defined header files --> search the file in the current directory or folder

--> collection of function protoytpe / signature / declaration
--> user defined dt, macros

.c -- source code

1. Unformatted

--> whatever input is taken from the user will be written directly into the memory

1. getchar ---> read a character from the user

char ch;

ch = getchar();

2. putchar(character) -- print a character on the stdout

putchar(ch);

3. gets() -- read a string
4. puts() -- print the string

5. getc and putc ---> reads a character and prints a character

getc(stdin);

putc(ch, stdout);

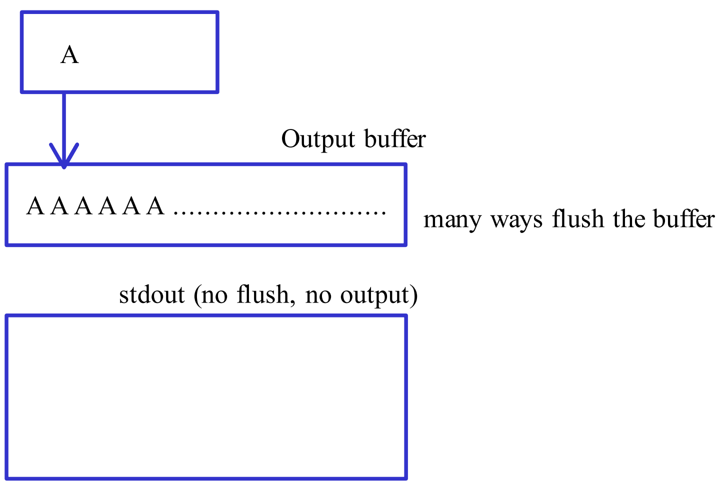
char str[10];

size * sizeof(char)
10 * 1 = 10bytes

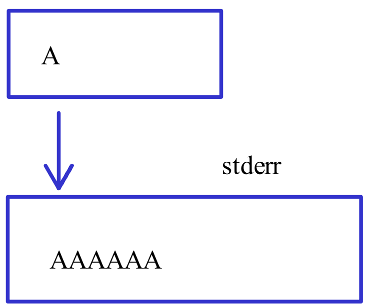
--> gets is dangerous to use because it can override the process when enetered more than the given size

stdout vs stderr?

--> buffers



error -- emergency message -- immediately executed

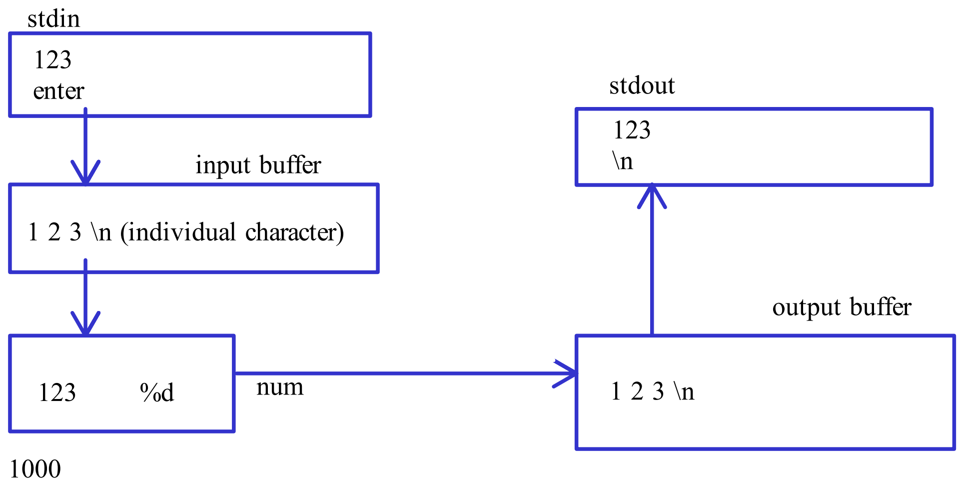


Formatted I/O

---> printf, scanf

int num;

scanf("%d", &num);
printf("%d\n", num);



int printf(const char *format, ...);

address of string

variadic function
any number of arguments

width specifiers

1
10
100
1000

%[x]d where x is number of blocks/division

%4d --> 4 division

space	space	space	1

space	space	1	0

space	1	0	0

precision modifiers

---> number of digit needs to be printed / strings

%[x].[x]d

%3.2d ---> 1

01

3 division and represent 1 in digit

.2f

space	0	1

%12.8s hello world

sp	spa	spa	spa	h	e	l	l	o		w	o

Escape sequence

escape character --> \n, \t tab space

\r ---> carriage return
--> it will bring the cursor back to the beginning of the output

```
printf("Hello World\rEmertxe");
```

Emertxeorld

--> printf returns the number of character printed on the screen

sprintf
--> similar to [printf but with different format

```
int sprintf(char *str, const char *format, ...);
```

returns number of character

address of array / string/ buffers

format of data with message

ellipses variadic function

```
int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string1[] = "sprintf() Test";
    char string2[100];

    sprintf(string2, "Hello world %d %c %f %s\n", num1, ch, num2, string1);

    printf("%s\n", string2);
```

%s -- used to print the string

string1

num1	ch	num2	string1
123	a	12.345	s p r i n t f ( ) T e s t \0
			2000

string2 100bytes

H	e	l	l o w o r l d 1 2 3 a 1 2 . 3 4 5 0 0 0 s p ri nt f Test ( )

1000

```
printf("%s\n", string2);
```

scanf

---> reads the input from stdin

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

successfully read input

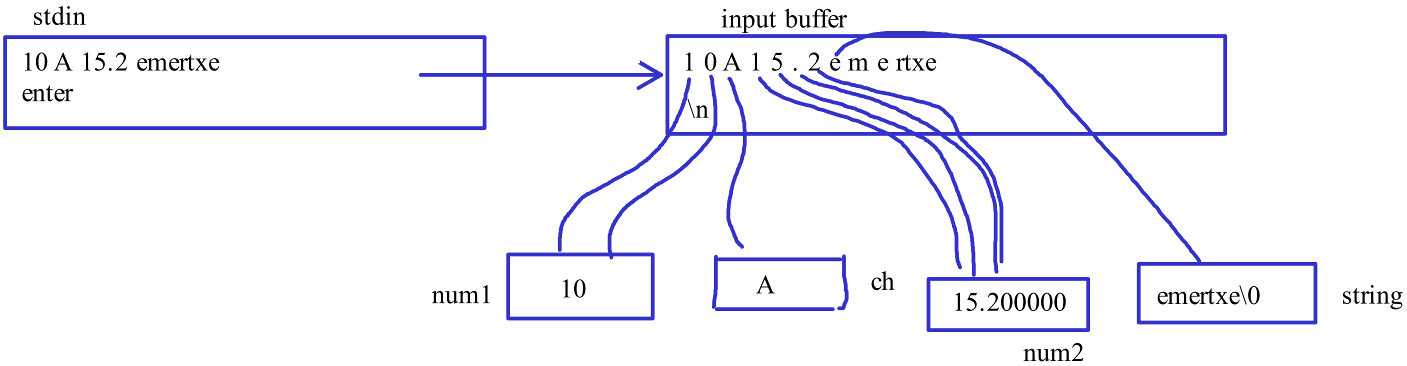
format of input

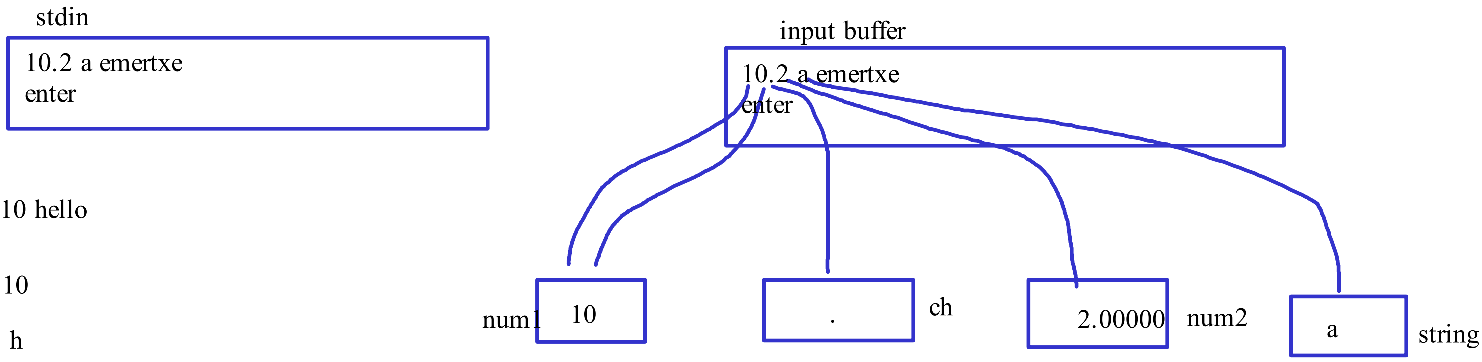
variadic function ellipses

```
int x, y, ret;
ret = scanf("%d%d", &var, &var2);
```

%d%c%f%s, &num1, &ch, &num2, string

10 A 15.2 emertxe





10 hello

10

h

input buffer

scanf("%s", str2);

Emertxe

Emertxe str2

--> %*specifier

%*c ---> ignore the character entered by the user

DD-MM-YYYY

%d%*c%d%*c%d

selective scanf

name --> alphabets

%[a-zA-Z]

int sscanf(const char *str, const char *format, ...);

read input from
array/string/buffer

format specifiers
%d...

variadic function

Buffers

--> temporary storage / memory

input buffer and output buffer

interpreter vs compiler

line by line

whole file at time
a.out

faster

Output buffer

Many ways to flush the output buffer

1. when program terminates

2. using \n

3. When the buffer memory is filled

1024bytes

1024 bytes

1s --- 11character = 11bytes

$\boxed{\text{Hello WorldHelloWorld..........................}}$ 93

1024 / 11 = 93s

4. fflush(stdout);

5. by using scanf

$\boxed{\text{100}}$

6. By disabling the buffer

setbuf(stdout, NULL);

$\boxed{\text{100}}$

equivalent of stderr

int num;

segmentation fault --- stderr

printf("%d\n", num); ---> stdout

%c --> 125, 126, 127, -128, -129

Input buffer

scanf

Pointers revision

variable which holds the address of another variable

datatype *ptr_name;

int *ptr;
char *ptr;

Rule 1: Pointer is an integer

address

sizeof(ptr)

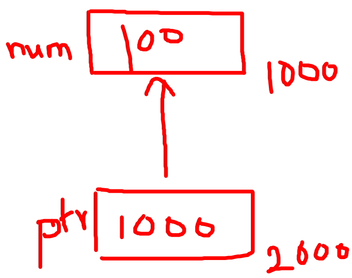
Rule 2: referencing nad dereferencing

& ---> address
* --- fetch the value

int num = 10;

*ptr

int *ptr = #



*1000 === go to memory location 1000 and fetch the value

Rule 3: Pointing means containing

Rule 4: Pointer datatype

how many bytes of data should be fetched

int *ptr;

Endianness

int num = 0x12345678;

78	56	34	12
1000	1001	1002	1003

Rule 5: Pointer arithmetic

int *ptr = #

arrays

ptr++; //ptr = ptr + 1 * sizeof(datatype)

```
*ptr++;

*(ptr++)

*ptr
ptr = ptr + 1 *sizeof(int)
```

command ---> stack command line argument
gcc

```
int main(int argc, char *argv[], char *envp[])
{

}
```

argc --- argument count, number of value/command passed while executing

gcc filename.c
./a.out 1 2 3 4

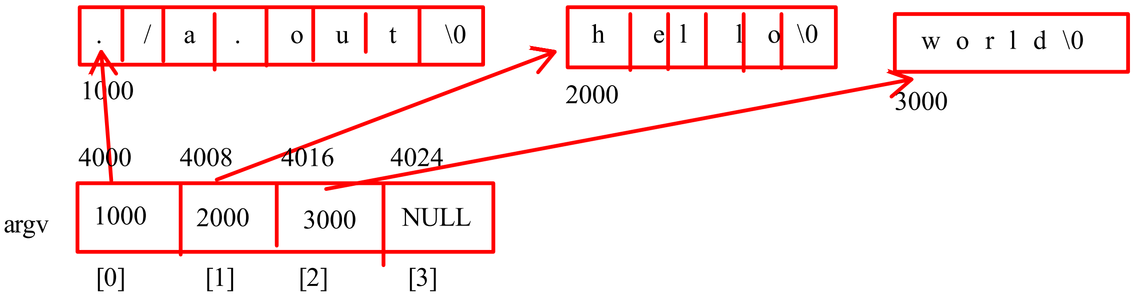
5 arguments

by default these are strings

./a.out hello world

%s --> argv[0] --> ./a.out

argv[1] -- hello
argv[2] -- world



argv --- argument vector --- collection of address of command line arguments

./a.out 10 20 30 40

10+20+30+40 = 100 / 4 = 25

char *envp[]

--> environmental variable

system information
path ---> gcc --> c:\
path --->

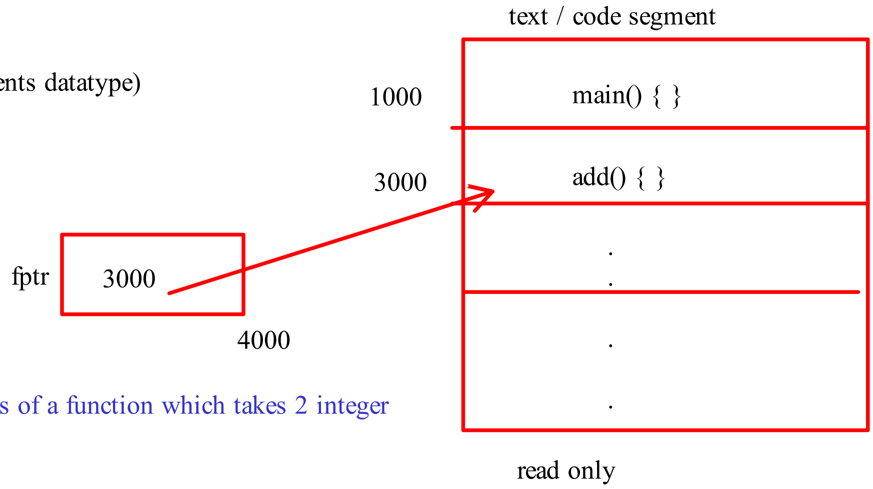
Function pointer

address

Pointer that holds/store the address of the function / point to the function

syntax:
return_datatype (*function_pointer)(list of arguments datatype)

```
int add(int num1, int num2)
{
   //logic
}
```



int (*fptr)(int, int);

fptr is a function pointer that can store the address of a function which takes 2 integer arguments and returns an integer as output

add --- address/memory location of that function add(10,20);

//1000(10,20)

fptr(20,30)

1000(20,30)

```
int *foo1()
{
    static int num;
    return &num;
}
```

```
char *foo2()
{
      static char str[] = "hello";

      return str;
}
```

callback functions

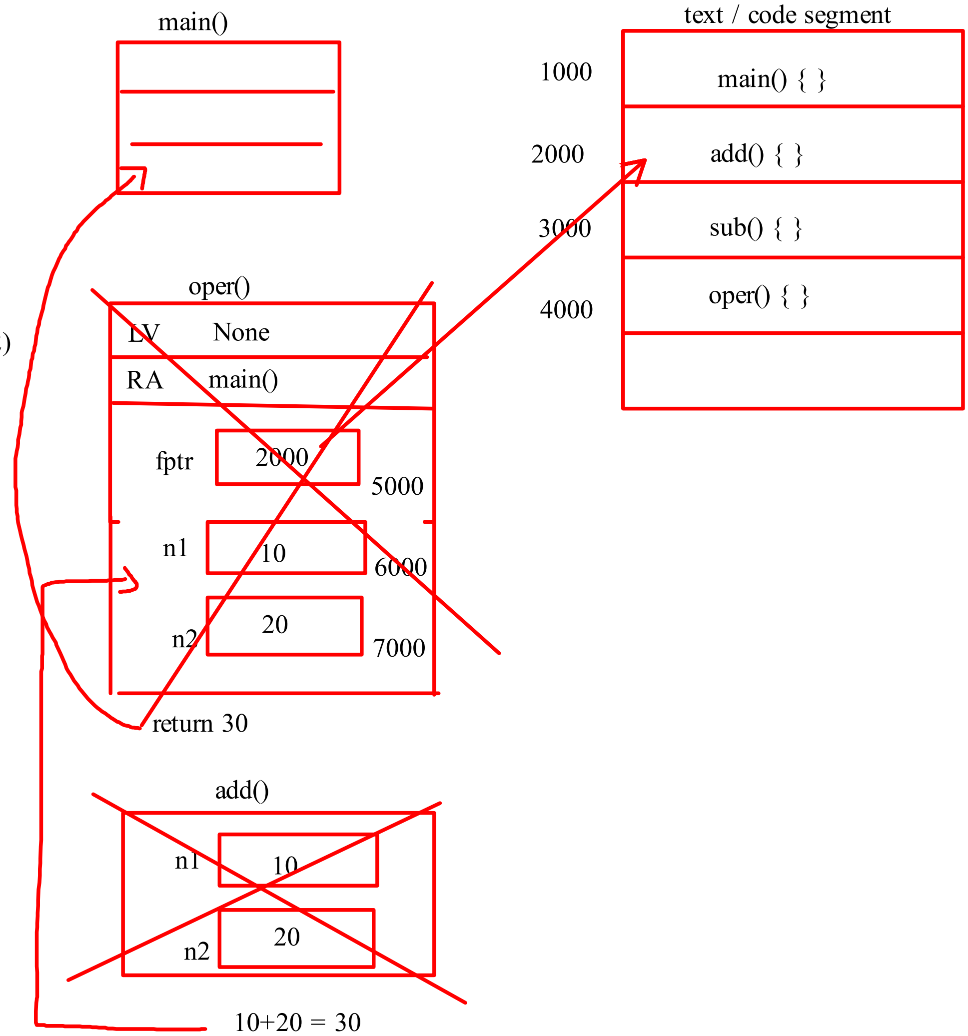
when you pass function as a argument to another function and calling that function within the called function

oper(add, 10, 20)

add()

library implementation ---> atexit(), qsort()

```
int add(int n1, int n2)
{
   return n1+n2;
}

int sub(int n1, int n2)
{
   return n1-n2;
}
int oper(int (*fptr)(int, int), int n1, int n2)
{
     return fptr(n1, n2);
}
int main()
{
    oper(add, 10, 20);
    oper(sub, 10, 20);

}
```



array of function pointer

collection of more than one function address

```
int (*fptr[3])(int, int);

  fptr[0] = add;
  fptr[1] = sub;
  fptr[2] = mul;
```

or

```
int (*fptr[3]) = {add, sub, mul};
```

atexit(function)

---> it will execute the function registered while terminating program

dynamic memory allocation ---

exit(0) -- terminate program immediately

```
int atexit(void (*function)(void));
```

qsort() --- sort any type of data -- array

generic function

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

base address of any array

number of elements of array

sizeof each elements

function pointer

compare function

comparison function

```
int compare_as(const void *ptr1, void *ptr2)
{
        if(*(int *) ptr1 == *(int *)ptr2)
        {
            return 0;
        }
        else if(*(int *)ptr1 < *(int *) ptr2)
        {
             return -1;
         }
         else return 1;
}
```

if ptr1 == ptr1, 0 equal
else ptr1 < ptr2, -1
else ptr1 > ptr2, 1

Variadic Function

function which takes any number of arguments

printf, scanf, sprintf, sscanf...

```
printf("Hello World\n");   1 argument
printf("Hello World %d %d\n", 100, 200); //3
```

...   --- ellipses

```
add(10,20)
add(10,20,30)
add(10,20,30,40);
```

```
int add(int n1, int n2)
{
     return n1+n2;
}
```

```
int main()
{
     sum = add(10,20);
     sum = add(add(10,20), 30);
     sum = add(add(10,20), add(30,40));
}
```

```
int add(int count, ...)
{
     int sum = 0;
    //declare argument pointer
    va_list ap;
    //point the ap to starting of function
    va_start(ap, count);

    for(int i = 0; i < count; i++)
    {
        //fetch the next optional argument
        sum += va_arg(ap, int);
    }

    //end the ap
    va_end(ap);
}
```

```
printf("Hello World %d %f %c\n",  100, 3.4, 'a');
```

"string"

\0

implement my_printf()

```
my_printf("Hello World\n");
```

```
void my_printf(const char *fmt, ...);
int main()
{
    my_printf("Hello World\n");

   return 0;
}

void my_printf("const char *fmt, ...)
{
       va_list ap;
        va_start(ap, fmt);

        while(*fmt)
        {
             putchar(*fmt);
             fmt++;
        }

        va_end(ap);
}
```

```
 char str[] = "Hello";

printf("%s",  str);

 case 's':
        ptr = va_arg(ap, char *);
        fprintf(stdout, "%s", ptr);
        break;
```

Pre processing

one step before the compilation

compilation involves 4 stages

1. Preprocessing
2. Compilation
3. Assembly
4. Linker

1. Preprocessing -- #

    1. Inclusion of header files
    2. Removing the comments
    3. Substitution of macros
    4. conditional compilation

gcc -E filename.c

2. Compilation
---> checks the syntax

.s -- assembly level code

3. Assembly

4. Linking

a.out -- link all the external varioable and function

#include --- header files

2 ways ---> <> ---> predefined or built in -- search the file in dedicated library path
if found the file in the path then load it else error

printf.c --
scanf.c

test.h ---> collection of function declaration and type definitions or macro definition

#include "test.h" ---> search in the current folder or directory, if found load the header else it will search in the dedicated library path, if found then load else error

Macro

Preprocessor directive which is used to give a meaningful name to a constant

3.14 --- PI

substituted in preprocessing stage

text replacement

global declaration

#define

should be defined in capital letter

__FILE__ --- name of the current file executing
__DATE__ --- current system date
__TIME__
__LINE__ --- current line number

__func__ -- function name

2 types of macro
1. object like macro

2. function like macro

pass argument to the macro

```
#define  MACRONAME(arg1, arg2...n)        expression;
```

example

```
#define   ADD(x, y)     x + y            //ADD(10, 20)     10 + 20


in tmain()
{
      int sum;
      sum = ADD(10, 20);            ===            sum = 10 + 20
      printf("%d\n", sum);

}
```

multiple line function like macro

```
#define swap(x,y)        \
int temp;                \
temp = x;                \
x = y;                   \
y = temp;                \
```

macro	function
1. stack frame is not created	stack frame
2. substituted in preprocessing	compiled at compile time, and executed at run time
3. datatype is not required.	datatype is mendatory
4. no context switching	context switching
5. macros are for smaller code	bigger problem statement
6. space is not optimized	space optimization

stringification

#

```
#define WARN_IF(EXP) \
do   \
{    \
x--; \
if (EXP) \
{ \
fprintf(stderr, "Warning: " #EXP "\n"); \
} \
} while (x);
int main()
{
int x = 5;
WARN_IF(x == 0);
return 0;
}
```

```
WARN_IF(x == 0);


do
{
     x--;
     if(x == 0)
     {
          fprintf(stderr, "Warning: " "x == 0" "\n");
     }
}while(x == 0);
```

conditional compilation

---> at the prprocessing stage itself depending on the condition it will either add or remove the code

```
#ifndef  MACRO_NAME
.....
#endif
```

if not define ---> if macro is not defined then load the content of this macro to your .c
if it is defined then ignore

#ifdef

……

if defined -- add or remove code whther macro is defined or not

#endif

User defined datatypes

int, float, double, char --->

student ---> id, batch no, attendance, marks, address…….

```
int id;
char name[20];
char add[50];
```

complex

```
int id;
char name[20];
char add[50];
```

can create our own datatype which is based on the primitive datatype

Structure

collection of elements of different/same type of data under one common name

int, float...
struct Student -- datatype

```
struct StructureName
{
    //data
}
```

```
struct Student
{
    int id;
    char name[20];
    char add[50];
};
```

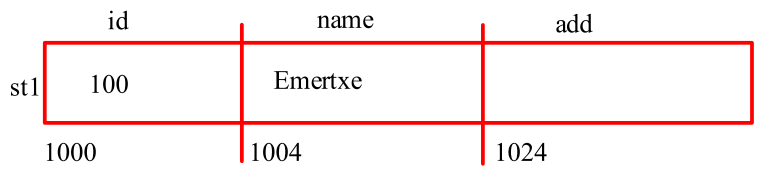
structure member / field

memory is not allocated
memory will be allocated whne you create variable for the structure

struct Student st1;   //memory allocation

assumption, 4 + 20 + 50 = 74bytes

struct Student st1, st2, st3, st4;



dot operator

st1.id = 100;
st1.name = "Emertxe"; //error

strcpy(st1.name, "Emertxe");

char name[20];

name = "Emertxe";

&st1.id --- 1000

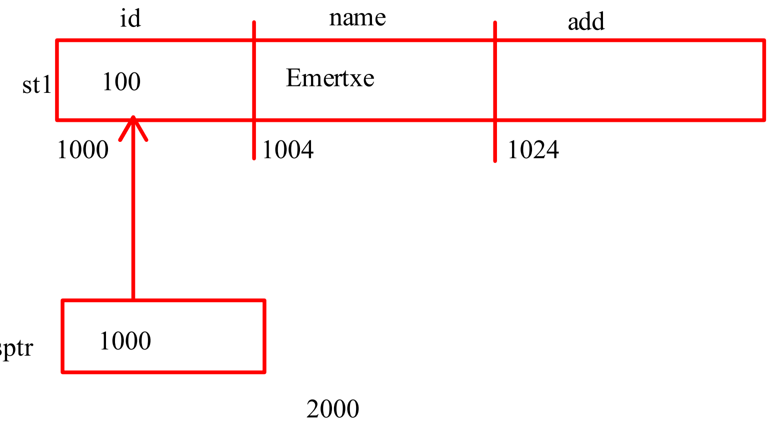
name -- base address

st1.name -- base address

structure pointers

--> pointer -- hold the address of structure

struct Student *sptr = &st1;



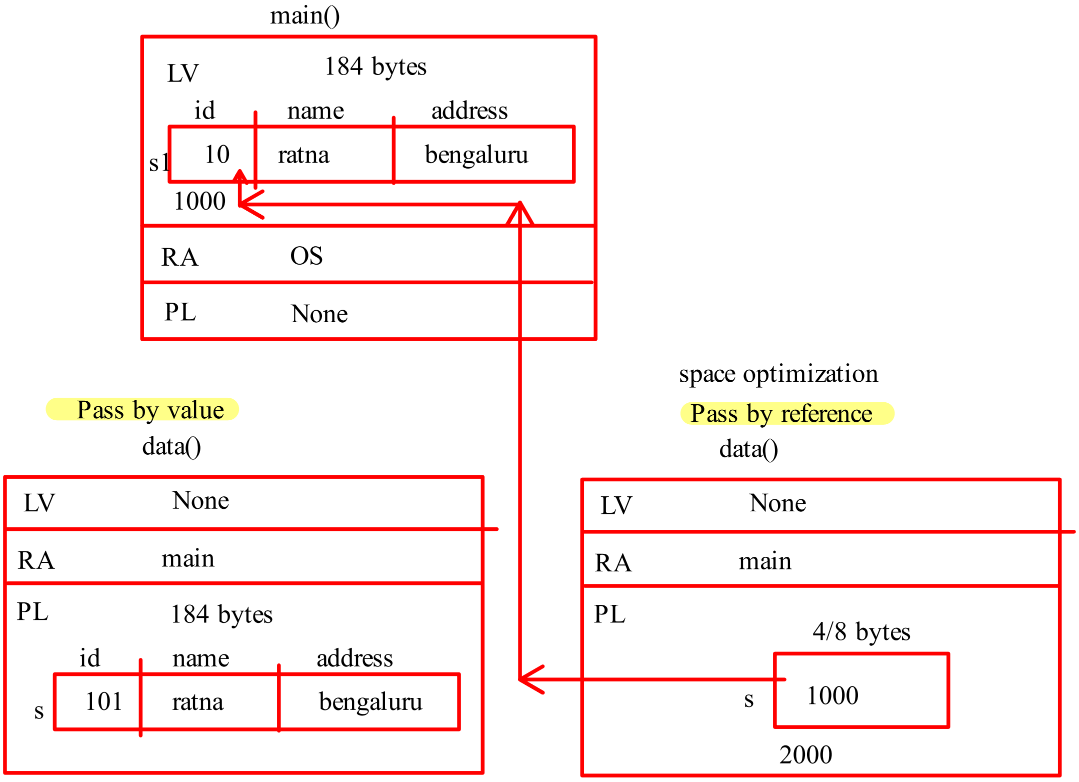
(*sptr).id = 10;

commonly used

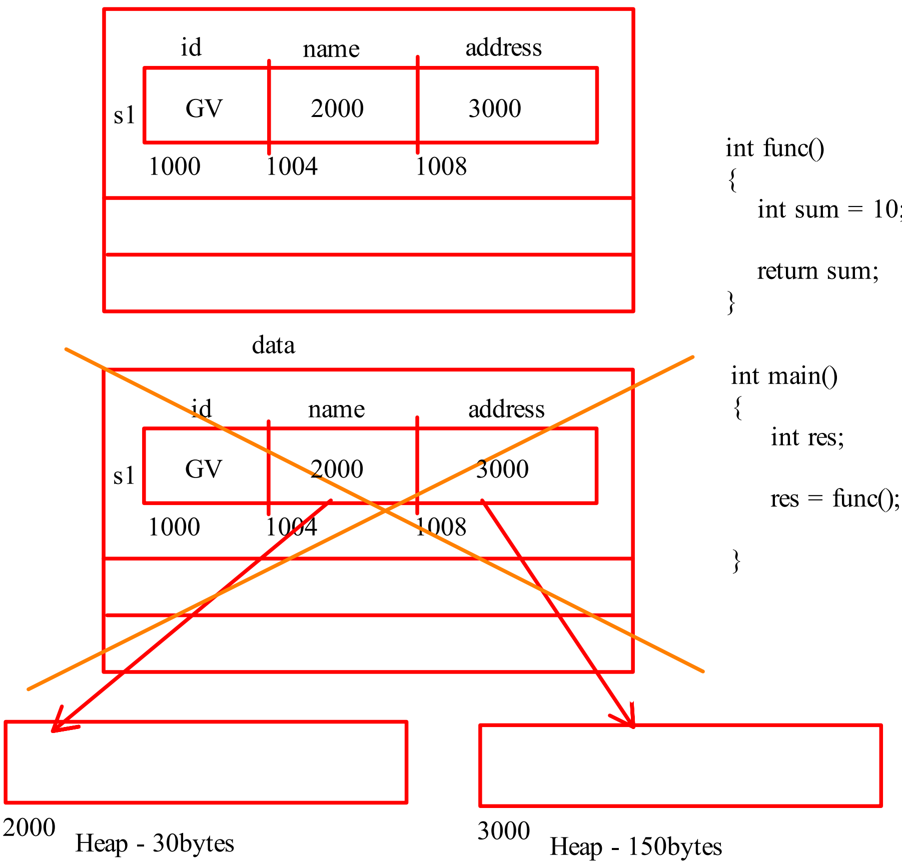
->
sptr -> id = 100;

Pass by value

```c
#include <stdio.h>

struct Student
{
        int id;
        char name[30];
        char address[150];
};

void data(struct Student *s)
{
        s->id = 10;
}

int main()
{
        struct Student s1;

        data(&s1);
        printf("%d\n", s1.id);
        return 0;
}
```



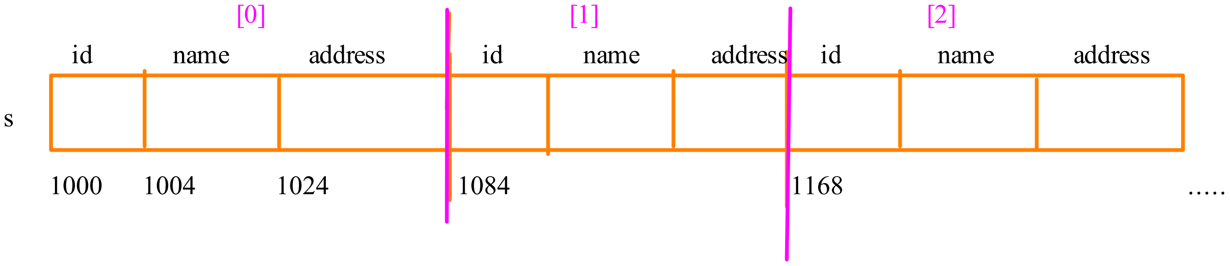

```c
#include <stdio.h>
#include <stdlib.h>

struct Student
{
        int id;
        char *name;
        char *address;
};

struct Student data(void)
{
        struct Student s;
        s.name = (char *) malloc(30 * sizeof(char));
        s.address = (char *) malloc(150 * sizeof(char));

        return s;
}

int main()
{
        struct Student s1;

        s1 = data();
        return 0;
}
```




```c
int func()
{
    int sum = 10;

    return sum;
}

int main()
{
    int res;

    res = func();

}
```

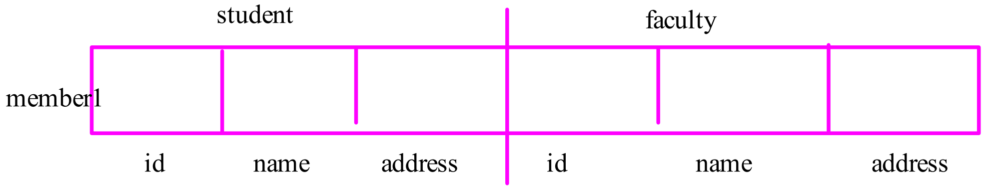
array of structures

```
struct Student s[3];
```



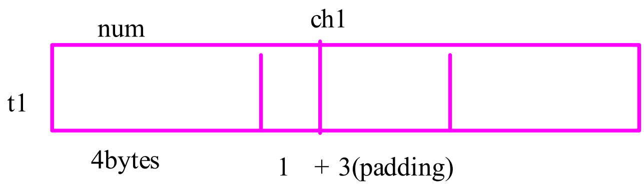
s[0].id, s[0].name

```
#include <stdio.h>

struct College
{
    struct
    {
        int id;
        char name[20];
        char address[40];
    }faculty;
    struct Student
    {
        int id;
        char name[20];
        char address[40];
    }student;
};
int main()
{
    struct College member1;

        return 0;
}
```



member1.student.id = 100;
member1.faculty.id = 200;

```
struct Test
{
    int num;
   char ch1;
};
```



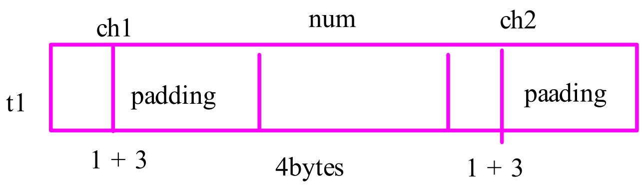

```
struct Test t1;
```

structure padding is done with following 3 rules

1. largest member of structure with respect to memory

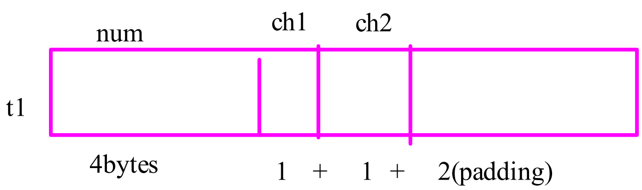
int --- 4bytes

```
struct Test
{
    char ch1;
    int num;
    char ch2;
};
```



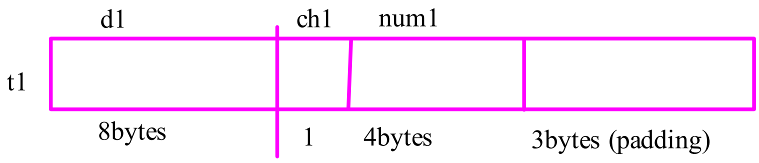
2. Order of member

```
struct Test
{
    int num;
    char ch1;
    char ch2;
};
```



if the immediate member of structure can fit into the padding bytes or not

```
struct Test
{
    double d1;//8
    char ch1;//1
    int num1;//4 + 3bytes
};
```

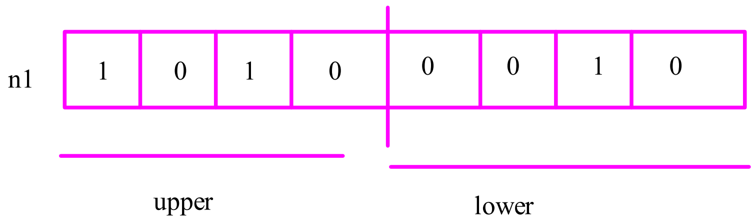


Bitfields

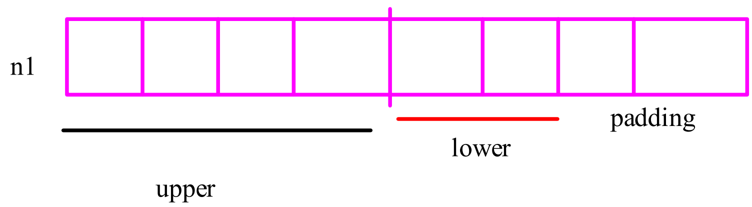
```
struct Nibble
{
        unsigned char lower : 4;
        unsigned char upper : 4;
};
```

```
struct StructureName
{
      datatype member1 : number_of_bits;
      ....
};
```

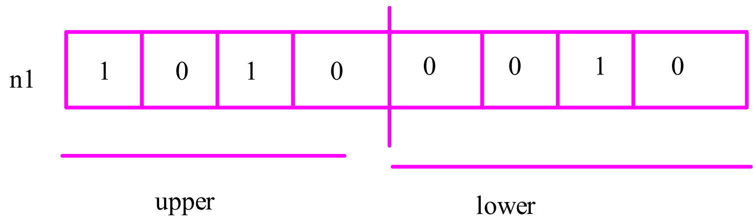
```
struct Nibble n1;

n1.upper = 0xA;

n1.lower = 0x2;
```




```
struct Nibble
{
        unsigned char lower : 2;
        unsigned char upper : 4;
};
```



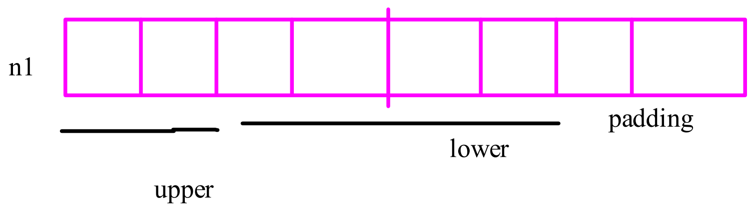

```
1010
0101
+   1
0110  -- -6
```



-6

0 to 2^n - 1

0 to 2^2 - 1
0 to 3




```
%o -- 4bytes --- 0000 0000 0000 0000 0000 0000 0000 0110
                 1111 1111 1111 1111 1111 1111 1111 1001
                                                       1
                 1111 1111 1111 1111 1111 1111 1111 1010
```

%#x --> 0xAB

%#o --> 056

Union --->

memory will be allocated to largest member and it will shared among the other members

```
union Test
{
   char option;
   int id;
   double height;
};

union Test t1;
```

	1000	1001	1002	1003	1004	1005	1006	1007
t1	31	12	00	00	CC	CC	1C	40

height
id
option

```
t1.height = 7.2;
t1.id = 0x1234;
t1.option = '1';
```

34 12 00 00

float, double --> s, m, e

3.2

0 10000000 100 1100 1100 1100 1100 1101

0 1000000 --- 40
0 100 1100 --- 4C
1100 1100 -- CC
1100 1101 --- CD

degree s e m

fb	CD	CC	4C	40
	1000	1001	1002	1003

```
union Endian
{
        unsigned int value;
        unsigned char byte[4];
};
int main()
{
        union Endian e = {0x12345678};
        e.byte[0] == 0x78 ? printf("Little\n") : printf("Big\n");
}
```

value	[0]	[1]	[2]	[3]
e	78	56	34	12

byte

byte[0]

mc -- 8051

psw register ---> Program Status Word

zero flag, overflow flag, auxillary, carry, parity.....

```
union reg
{
    char psw;
    struct
    {
        char zf : 1;
        char pf : 1;
        char af : 1
        char of : 1;
    }PSW;
}
```

```
union reg R1;
R1.psw = some value;
R1.PSW.of = 1;
```

```
void foo1(data x)
{   }

void foo2(data x)
{   }

void foo3(data x)
{   }

void foo4(data x)
{   }
```

```
typedef double data;
int main()
{
data x;
foo1(x);
foo2(x);
.
.
}
```

```
typedef datatype  name_typedef;
```

```
ui num;
uli num;
```

```
typedef  unsigned int ui;
typedef unsigned long int  uli;
```

```
struct Student st1

Student s1;
```

```
typedef struct Student
{


}Student;
```

Enums

--> collection of constant

stdio.h

1. create file pointer

```
FILE  *filepointername;

FILE *fptr;
```

2. open the files which is required to do the specific operaton

```
FILE *fopen(const char *pathname, const char *mode);
```

"path of file"

mode --- mode of file

"r" ---> read mode
---> read only -- read the content from file
---> If file exist in the given path then it will open the file in read mode else it will return NULL
--> If success then file pointer will be pointing to the first byte/position/character of the file

"w" ---> write mode
---> write only -- writes the content to file
--> If file exist in the given path then it will open the file in write mode else it will create the given file in the path
--> If success then file pointer will be pointing to the first byte/position/character of the file
--> if file contains data then it will replaced by new content

"a" ---> append
  --> append only --- writes the content to file
  ---> If file exist in the given path then it will open the file in write mode else it will create the given file in the path
  --> If success then file pointer will be pointing to the last byte/position/character of the file
  --> if file contains data then it will merge/combine old content with new content

"r+" -- read and write
  ---> If file exist in the given path then it will open the file in read mode else it will return NULL
  --> If success then file pointer will be pointing to the first byte/position/character of the file

"w+" --- write and read
  ---> If file exist in the given path then it will open the file in write mode else it will create the given file in the path
  --> If success then file pointer will be pointing to the first byte/position/character of the file
  --> if file contains data then it will replaced by new content

"a+" --- append and read
  --- If file exist in the given path then it will open the file in write mode else it will create the given file in the path
  --> If success then file pointer will be pointing to the last byte/position/character of the file
  --> if file contains data then it will merge/combine old content with new content

fgetc -- fetch/read a character from the file

fputc(ch, stdout);

fgets(ch, 100, filepointer);
fputs(ch, filepointer); write the content of array to file

ftell --> provides the information about the position of file pointer

ferror() and clearerr() -- used to track the error of file pointers

ferror --> error flag = 0
---> error flag will be set

```
struct Test
{                                  4bytes
    int num;                                        16bytes
    char arr[10]; //10 + 2 padding 12
};
```

```
struct Test
{                                  4
    int num;                       20        24bytes
    char arr[17]; //17 + 3
};
```

fprintf and fscanf

fprintf --- used to write the the content into file by converting to a particular format

fgetc, fputc, fgets, fputs --- unformatted

fprintf(file_pointer, "format specifier", var1, var2...)

rewind ---> it will bring the file pointer to the beginning of the file

fscanf --- formatted input function used to read the content from a file ina particular format

fscanf(fptr, "format specifier", &var1, &var2...);

fseek

--> used to move the file pointer to the given position

fseek(fptr, how many bytes needs to be moved, from where);

from where ---> SEEK_SET --- move the file pointer from the beginning of the file

fseek(fptr, 10, SEEK_SET);

-- instructing the file pointer to move to 10th position from the beginning of the file

SEEK_END -- move the file pointer from end of the file

fseek(fptr, -10, SEEK_END);

move the file pointer backwards by given number of bytes

SEEK_CUR
--> used to move the file pointer from the current position

fseek(fptr, 10, SEEK_CUR);

move the file pointer forward from current position by 10 bytes

fseek(fptr, -10, SEEK_CUR);
move backward from current position

fwrite and fread

--> read and write the content from file directly in the form of diagraph (machine code)
--> non human readable
--> large data types like structure, array...

fwrite
--> used to write content to the file in non human readable format

```
fwrite(const void *ptr, size_t size, size_t nmemb,FILE *stream);
```

address of variable from where | positive integer size of variable | number of elements | file pointer to file

```
int num = 10;

fwrite(&num, sizeof(int), 1, fptr);
```

```
fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

address of variable to where | size of variable | number of elements | file pointer from

volatile

--> to avoid the optimization of your code

```
for(int i = 0; i < 0xfffffff; i++);  //remove the code
```

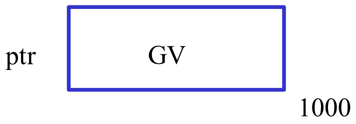
Rule 6: Pointing to nothing

Usr space ---> appications

Kernel --> not allowed to access directly -- 0

int *ptr;

wild pointer



1221424 → gv
-123233 → segmentation
0 → segmentation

int num = 0;

undefined behavior

int *ptr = NULL;

(int) num;

NULL is a predefined macro, which says (void *) 0

pointing to nothing

illegal access -- segmentation fault

void pointer

void -- nothing

incomplete pointer

direct dereferencing is not allowed with void pointer

pointer arithmetic is compiler dependant, in gcc it is possible

ptr++;

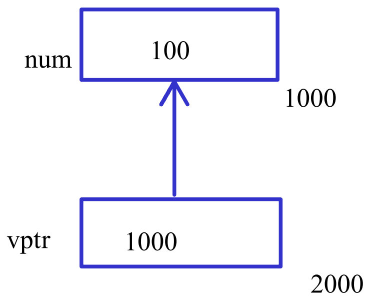
ptr = ptr + 1 * sizeof(datatype)

sizeof(void) -- 1byte

int num = 100;

void *pointer_name;

void *vptr = #



generic function

swap

--> function which works with any type of data
--> void pointer is a generic pointer -- any type of data

double x = 7.2;

step 1:

$111.\overline{0011}$

step 2:

step 3: 1023+2 = 1025

y = 2

1.110011

0	10000000001	1100110011001100110011001100110011001100110011001101

0100 0000 0001 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101

4 0 1 C C C C C C C C C C C C D

8bytes

8 byte

x	CD	CC	CC	CC	CC	CC	1C	40
	2000	2001	2002	2003	2004	2005	2006	2007

vptr	2000
	3000

%x -- 4bytes hexa
%hx ---> 2bytes
%hhx ---> 1byte

```
printf("%hhx\n", *(char *)vptr);          CD
printf("%hhx\n", *(char *)(vptr + 7));    40
```

2000 + 7 = 2000 + 7 * sizeof(void) = 2000 + 7*1 = 2007

```
printf("%hx\n", *(short *)(vptr + 3));
printf("%x\n", *(int *)(vptr + 0));
```

2000 + 3 * 1 = 2003 --- short ---> 2bytes

CCCC

2000 + 0 * 1 = 2000

CD CC CC CC

CC CC CC CD

char int
int

int double
int

num1	78	56	34	12
num2	FF	EF	CD	AB

FF563412
123456FF

78EFCDAB
ABCDEF78

1. FF EF CD AB
2. 78 56 34 12

4times
8times

```
void swap_generic(void *num1, void *num2, int size)
{
        char temp;
        int i;
        for(i = 0; i < size; i++)
        {
              temp = *(char *) num1;
              *(char *) num1 = *(char *) num2;
              *(char *)num2 = temp;
              num1++;
              num2++;
         }
}
```

swap_generic(&num1, &num2, sizeof(int));

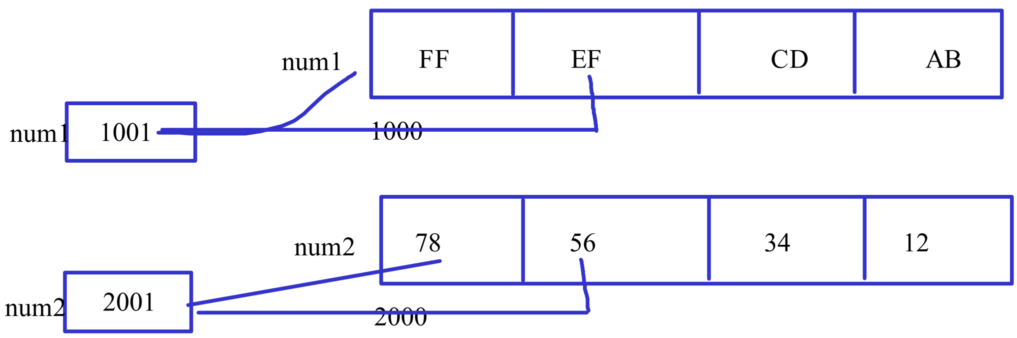
12345678;   int *
char

int *
double

4bytes
8bytes

char * -- loop till the size of datatype

4 bytes
8 bytes



sizeof(double)

for (i = 0; i < size; i++)

*(char *) num1 ---> EF
*(char *) num2 ---- 56

num1++
num2++

Dynamic memory allocation

int num;
float, char......;

static memory / named memory

int arr[10];

dynamic --- allocate memory whenever required, deallocate(delete), extend, shrink

unnamed memory --- heap memory

pointers

malloc, calloc, realloc, free --- stdlib.h

```
int main()
{
      //100 bytes
      //declare pointer
      //use function to allocate memory -- malloc, calloc
     //do the specific task

     //free memory
}
```

malloc

--> allocate memory dynamically in heap

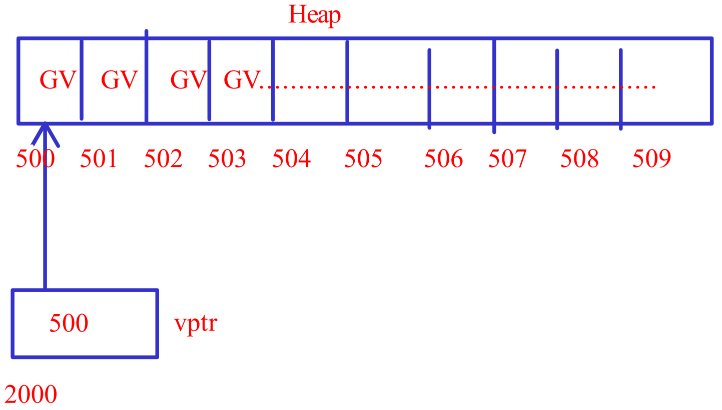
void *malloc(size_t size);

returns the address as void pointer

typedef
unsigned int/long int
positive integer value

--> how many bytes of memory is required

malloc(10);

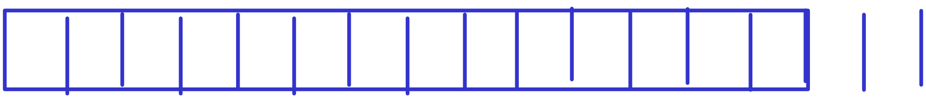


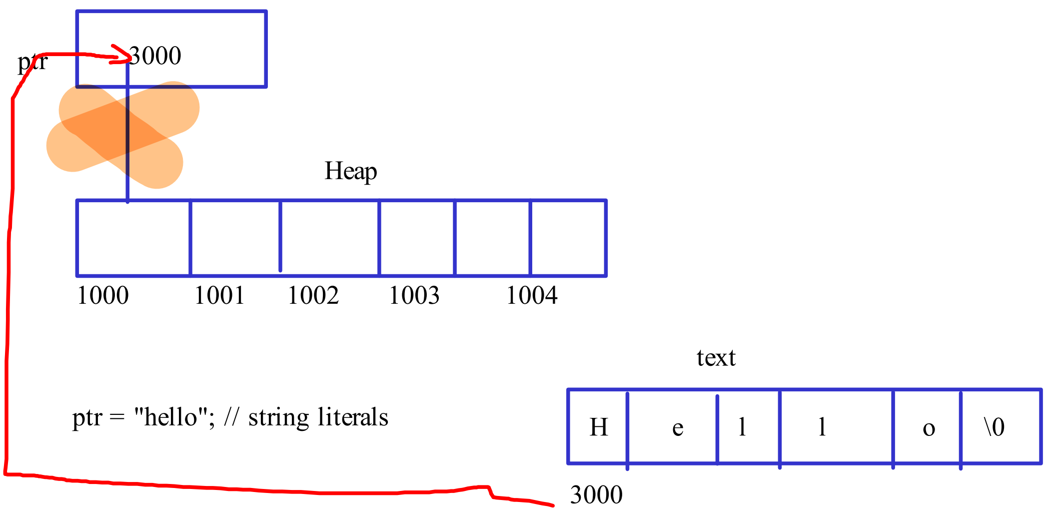
--> if requested memory is not avaiable or did not find it then malloc will return NULL

4 * 4 = 16bytes

malloc(4 * 4);

malloc(4 * sizeof(int));





calloc --- allocate requested memory in heap

void *calloc(size_t nmemb, size_t size);

positive int value
number of members/block/elements

size of each elemnt/block/member

void pointer/address

calloc(4,4); //4 * 4 = 16bytes

calloc(4, sizeof(int));
calloc(4, sizeof(char));

--> search continuous memory in heap if found return the address or else returns null

space ---> both will allocate requested memory, both are same

time
--> calloc function will assign 0 by default

0	0	0	0	0	0	0	0

--> calloc will take some extra time
malloc is bit faster than the calloc

--> malloc takes one value as argument
calloc takes 2 arguments

--> malloc is preffered for basic memory and structure

--> calloc is preffferred for arrays

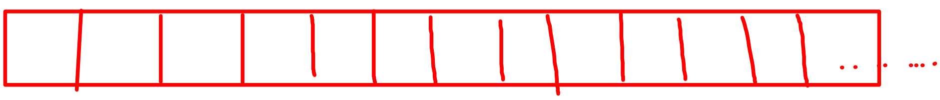
calloc(5, 4); //5*4 = 20bytes

int arr[10];//4*10

malloc(5 * sizeof(int))

Heap

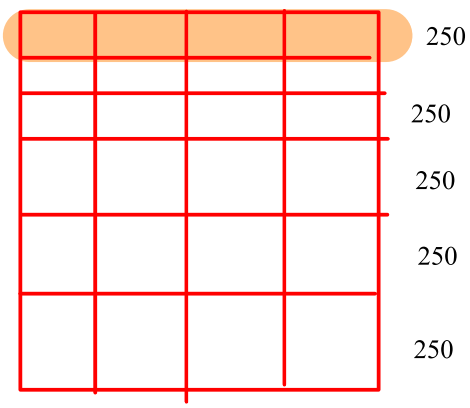
malloc(20)



1000 1001 1002 1003 1004

ptr = malloc(250);

ptr = malloc(250);



memory leakage

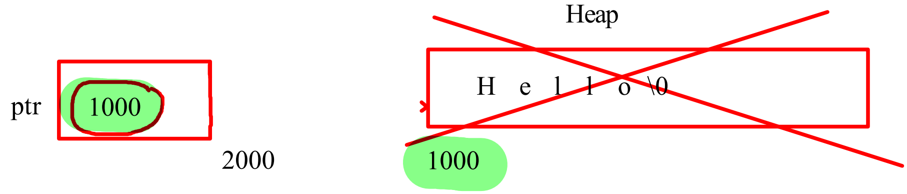
free

---> used to deallocate or free the dynamic memory

void free(void *ptr);

nothing

pointer which holds the memory allocated by malloc or calloc

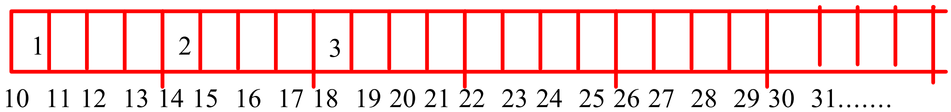


free(ptr)

dangling pointer

ptr = NULL;

```
for(i = 0; i < 10; i++)
{
        printf("%d\n", arr[i]);
}
```



calloc(6, 4); // 6*4 = 24 bytes

*(arr+i)
*(10+0*4)
*10

*(arr+1)
*(10+4)
14

1

int arr[6];

0000 0000 0000 0000 0000 0000 0000 0001

13 12 11 10

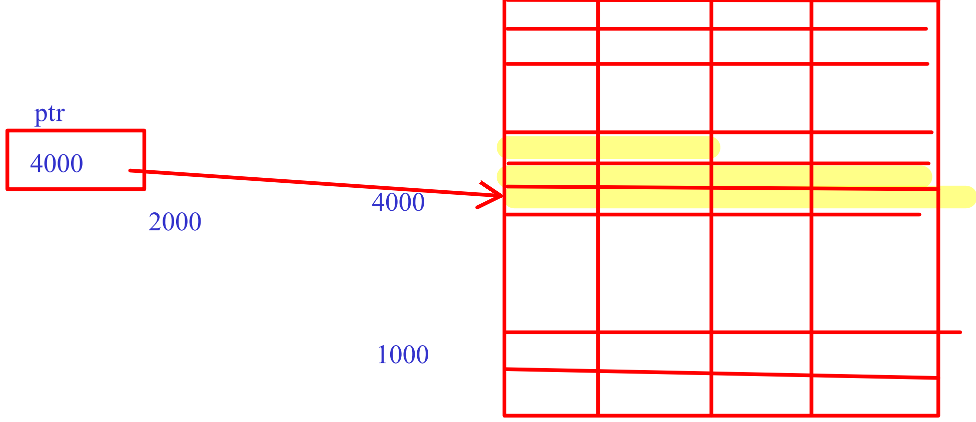
void *realloc(void *ptr, size_t size);

used to either extend / shrink the previously allocated memory

valid address from previously allocated memory

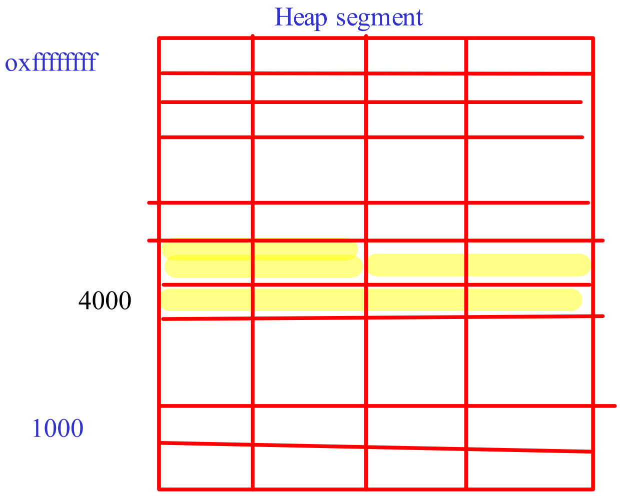
positive integer value

int *ptr = malloc(10);



Memory shrink

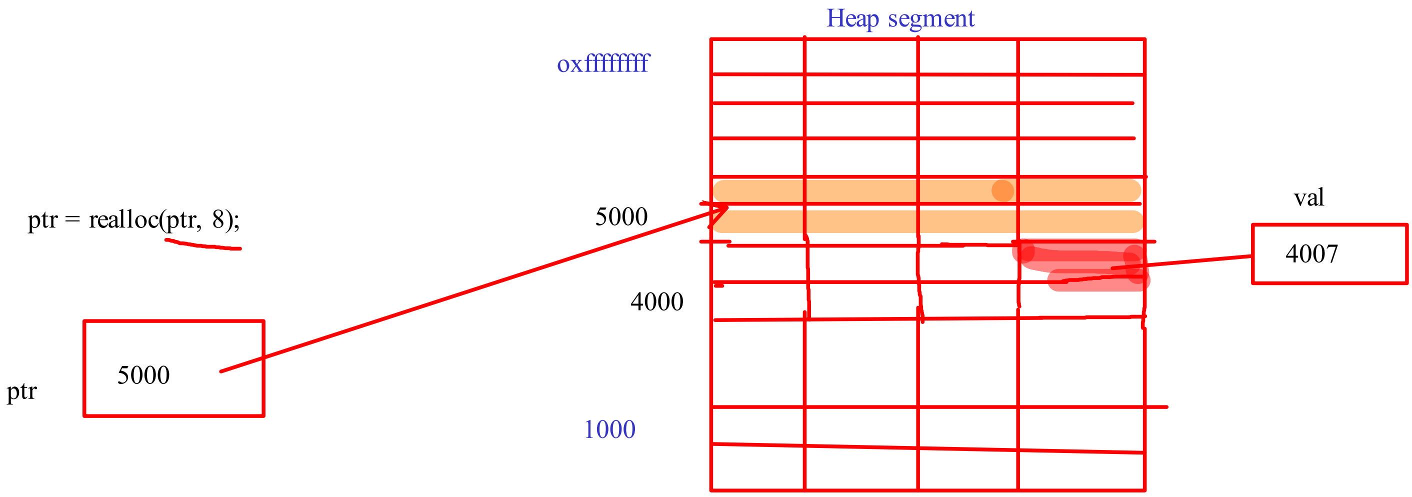
ptr = realloc(ptr, 6);reduce the memory by 10-6 = 4bytes



Extend

ptr = realloc(ptr, 8);

extend the memory by 8 - 6 = 2bytes



if requested extending memory not avaible then returns the null

new_ptr = realloc(ptr, 8);

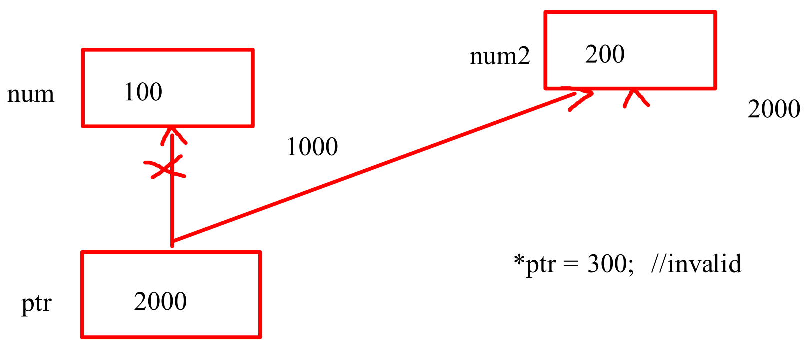
const with pointer

Pointer to a constant

syntax: const datatype *ptr_name;

int num = 100;

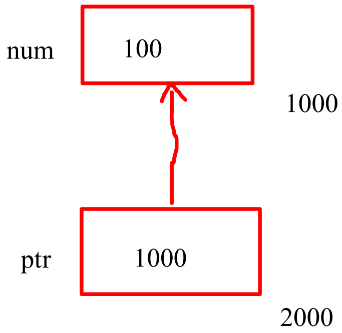
const int *ptr = #



Constant pointer

dtatype *const pointer_name;

the address is constant



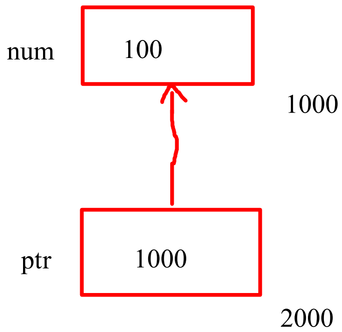
*ptr = 900;

int num = 100, num2 = 200;;
int *const ptr = #

ptr = &num2; //invalid

constant pointer to a contant value

int const * const ptr;
const int * const ptr;



*ptr = 400;
ptr= &num2; both are invalid

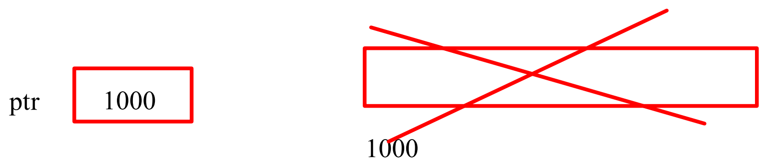
wild pointer

int *ptr;

*ptr = 100;
printf("%d\n", *ptr);

undefined behavior

int *ptr = malloc(10)



*ptr

ptr = NULL;

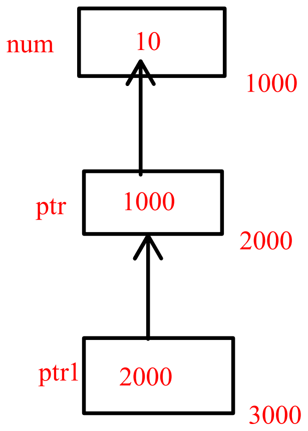
ptr = malloc(10);

int num = 10;
int *ptr = #

*********ptr

address of another pointer

int **ptr1 = &ptr;



*ptr --> *1000 --- 10

*ptr1 --> *2000 --- 1000

**ptr1 --- **2000 ---> *1000 ---> 10

2 Dimensional array

collection of rows and columns -- matrix

syntax:

datatype array_name[rows][columns];

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

number of elements = rows * columns

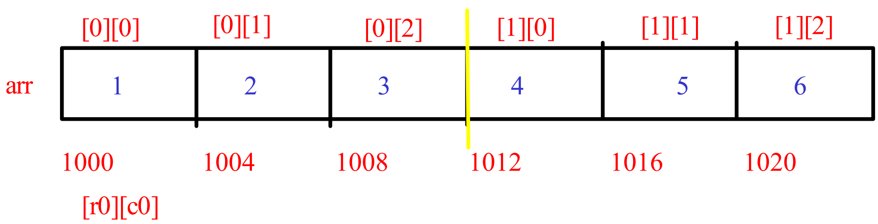
total memory = numberofelements * sizeof(dtatype_array)

total memory = rows * columns * sizeof(datatype);

int arr[2][3];

number of elements = 2 * 3 = 6

total = 6 * sizeof(int)
total = 6 * 4 = 24bytes



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

arr[0][1]
arr[0][2]
arr[0][3]

arr[1][0]
arr[1][1]
arr[1][2]

```
for(row = 0; row < 2; row++)
{
      for(col = 0; col < 3; col++)
      {
            printf("%d\n", arr[row][col])
      }
}
```

arr[i][j]

arr[i] ==> *(arr + i)

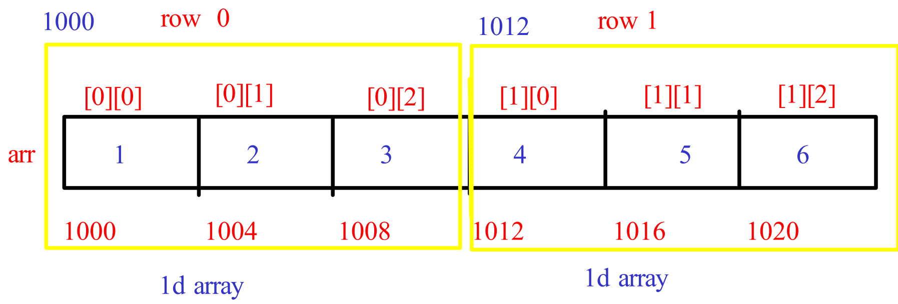
--> replace arr[i] as x

x[j] == *(x+j) == *(x + j * sizeof(datatype_array))

1. *(arr[i] + j) == *(arr[i] + j * sizeof(datatype_array))

2. *(*(arr + i) + j) = *(*(arr + i * sizeof(row)) + j * sizeof(datatype))

1. i = 1, j = 1   arr[1][1]



*(arr[i] + j * sizeof(datatype_array))

*(arr[1] + 1 * sizeof(int))
*(arr[1] + 1 * 4)
*(arr[1] + 4)
((arr + 1 * sizeof(row/1d array)) + 4)
((1000 + 1 * (3 * sizeof(int)) + 4)
((1000 + 1*(3*4)) + 4)
((1012) + 4)
*(1012 + 4)
*1016
5

--> collection of 1d arrays
--> row represents 1d array

3. *(*arr+i))[j]

Array of pointers

array --- collection
pointer -- address of another variable

collection of address

syntax:

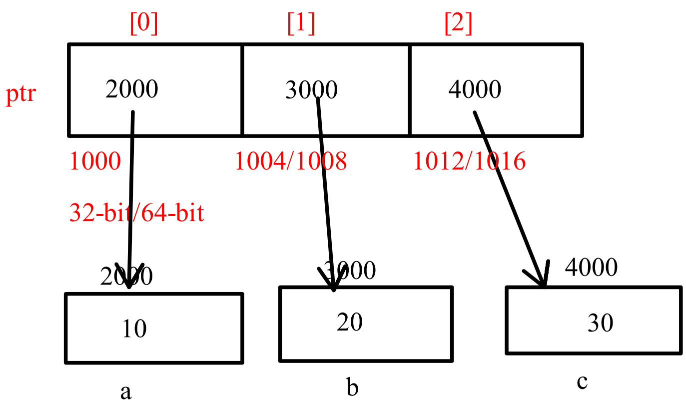
datatype *pointer_name[size];

int *ptr[3];   ptr is a array of pointer which is capable of holding 3 address/memory location

total = size * sizeof(ptr);

total = 3 * 8 = 24bytes

3 * 4 = 12bytes(32 bit)



1000 --- &ptr[0]
1008
1016

ptr[0] -- 2000

*ptr[0] -- *2000
10

int a = 10, b = 20, c = 30;

int *ptr[3];

ptr[0] = &a;
ptr[1] = &b;

ptr[1] = &c;

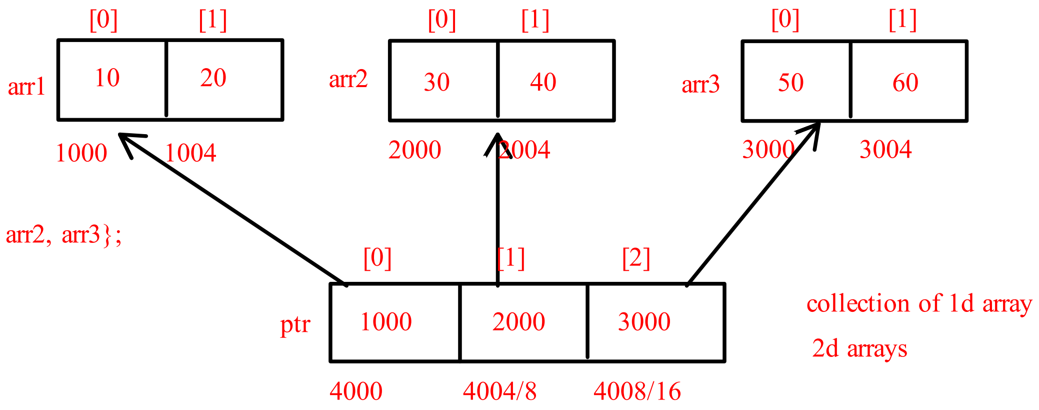
int *ptr[3] = {&a, &b, &c};

*ptr[0] --- 10
*ptr[1] --- 20
*ptr[2] -- 30

```
void func(int *ptr1, int *ptr2, int *ptr3)
{
}

void func(int *ptr[3])
{
}
```

int arr1[2] = {10, 20}, arr2[2] = {30, 40}, arr3[2] = {50, 60};

int *ptr[3] = {arr1, arr2, arr3};



*ptr[0]
*1000
10

ptr[0][0]
ptr[0][1]

ptr[1][0]
ptr[1][1]

ptr[2][0]
ptr[2][1]

*ptr[1]
*2000
30

*ptr[2]
*3000
50

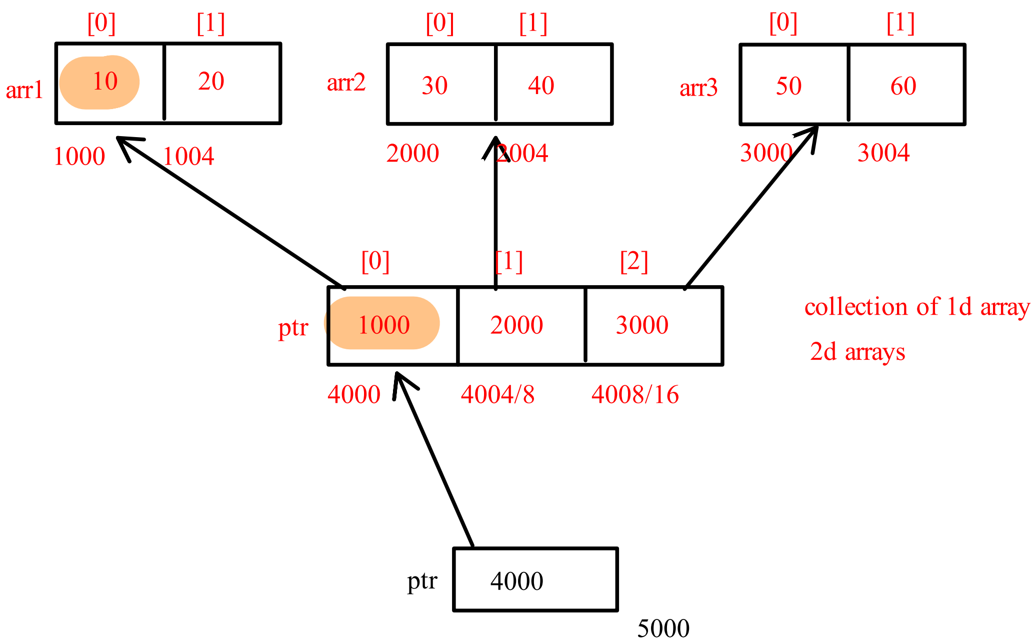
ptr[0] --- 1000
ptr[1] --- 2000
ptr[2] --- 3000

&ptr[0] ---> 4000
&ptr[1] --- 4008

ptr ---> 4000

**ptr --- *4000 --- *1000

*(ptr[0]++)
*(1000++)
*(1000++)




```
void print_array(int **ptr)
{
    int i, j;

    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 2; j++)
        {
            printf("%d\n", ptr[i][j]);
        }
    }
}
```

int *ptr[0]
int **ptr;

**ptr++

((ptr++))

*ptr ---> *4000 ---> 1000
ptr = 4008

*1000 ---> 10

Array of strings

char str[6] = "Hello";

char str[6] = {"h", "e", "l","l","o","\0"};   collection of string

2d arrays

char str[3][8] = {"Array", "Of", "Strings"};

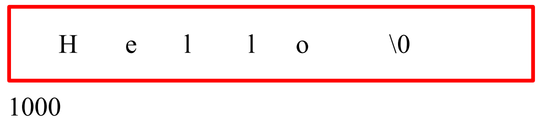
str	1	2	3	4	5	6	7		
1000	A	r	r	a	y	\0			[0]
1008	O	f	\0						[1]
1016	S	t	r	i	n	g	s	\0	[2]

printf("%s\n", str[0]);

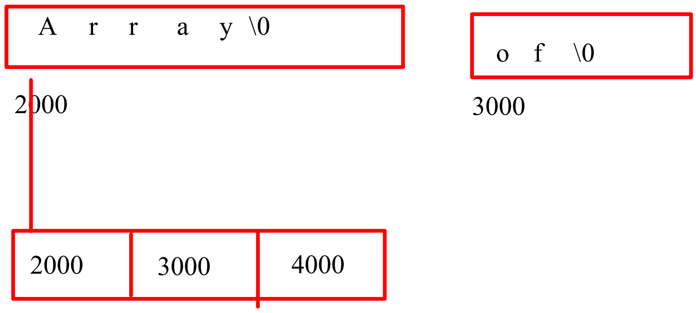
printf("%s\n", str[1]);

Text/code segment

char *str = "Hello";



char *str[3] = {"Array", "Of", "Strings"};



Pointer to an array

--> explicitly used for 2d array

datatype (*ptr_name)[size];

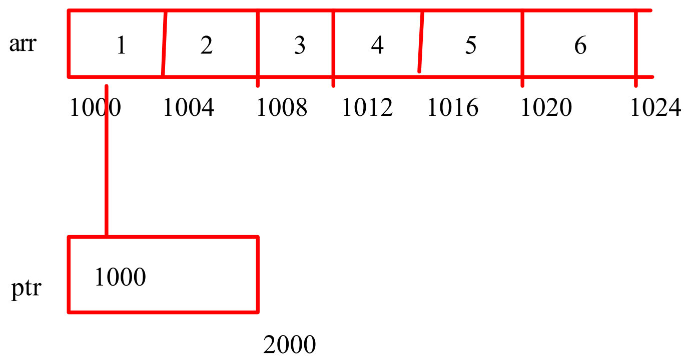
int (*ptr)[3]; --> ptr is a pointer to an array of 3 integer elements
--> its pointing to a whole row which has 3 columns

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

int *ptr = arr;

ptr++; //ptr + 1 * sizeof(int)

array of pointer ---> normal variable



** --- multilevel
[][] -- 2d array
*[] -- array of pointer
(*)[] -- pointer to an array

dynamic 2d array

How pass 2d array to a function

1. the way you declare 2d array

```
void print_array(int arr[2][3])
{
}
```

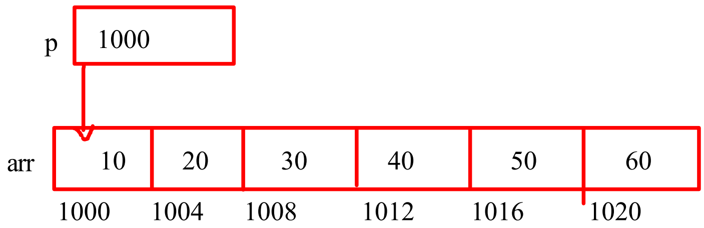
2. Passing size along with array

```
void func(int row, int col, int arr[row][col])
{
}
```

3. void func(int row, int col, int (*ptr)[col])
{}

4. normal integer pointer

indirect way




```
void print_array(int row, int col, int *p)
{
    int i, j;
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            printf("%d\n", *((p + i * col) + j));
        }
    }
}
```

1. i = 0, j = 0

*((p + 0 * 3) + 0)

*((1000 + 0 * 3 * sizeof(int)) + 0)
*((1000 + 0 * sizeof(int))
*(1000)

2. i = 0, j = 1

*((p + 0 * 3) + 1)

*(1000 + 1 * 4)
*1004

3. i = 0, j = 2

*1008

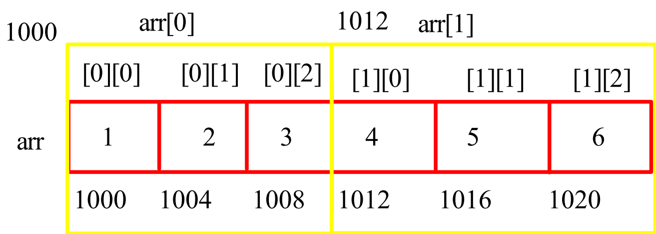
2. i = 1, j = 0

*((1000 + 1 * 3 * sizeof(int)) + 0)

*((1000 + 3 * 4) + 0)
*(1012 + 0 * 4)
*1012

5. i = 1, j=1
*(1012 + 1 * 4)
*1016

6. i = 1, j=2
*(1012 + 2 * 4)
*1020



arr[1][1]

((arr + i) + j) == i =1, j = 1

((arr + 1) + 1 * sizeof(int))
((arr+1) + 4)
((1000 + 1 * sizeof(1d array/row)) + 4)
((1000 + 1 * 3 * 4) + 4)
((1012) + 4)
*(1012 + 4)
*1016
5

dynamic 2d array

maloc, calloc

row, column

1. Both static --
row and column are fixed

int arr[2][3];
int arr[row][col];

2. FSSD -- First Static Second Dynamic

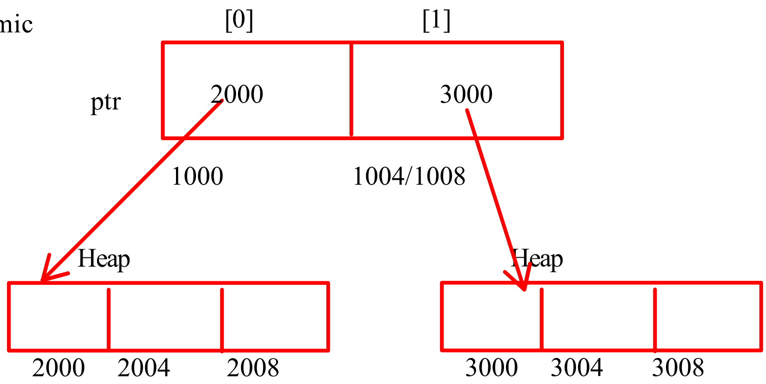
First -- row
second -- column

row will be fixed but columns are dynamic

row = 2

array of pointers

```
int *ptr[2]; //row
int col = 3;

for(i = 0; i < 2; i++)
{
    ptr[i] = malloc(col * sizeof(int));//3*4
     or
    ptr[i] = calloc(col, sizeof(int));
}
```



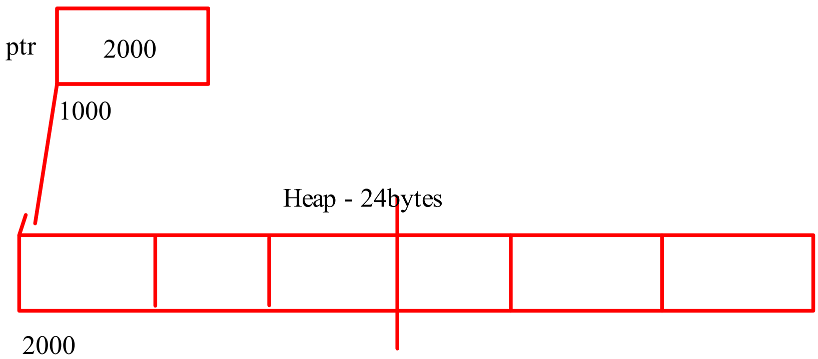
3. FDSS

First Dynamic Second Static

row is dynamic column is static

pointr to an array

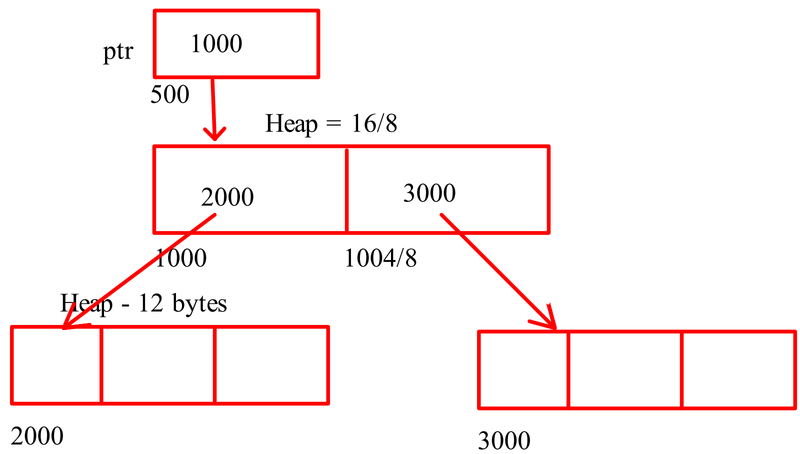
```
int (*ptr)[col];        row = 2    col = 3

ptr = malloc(sizeof(*ptr) * row)

ptr = malloc(12 * 2)
```



4. Both dynamic

row dynamic, column dynamic
row = 2   col=3

```
ptr = malloc(row * sizeof(int *));
        2 * 8

for(i = 0; i < row; i++)
{
    ptr[i] = malloc(col * sizeof(int));
}
```



multilevel pointer