⇛ Programming :-

1. Scope of a variable -

(i) Static scoping → resolved by compiler

(ii) Dynamic scoping → resolved by processor

eg:-

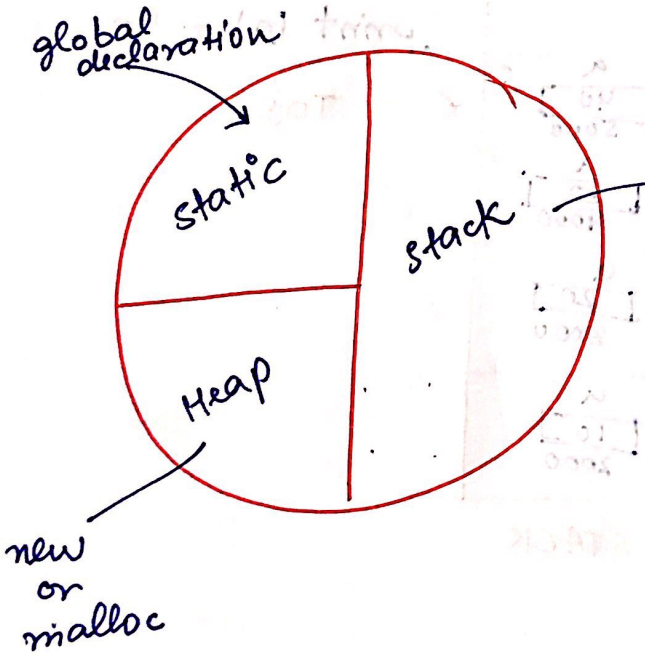
```
int a=5;  global variable
main()
{
  int a = 10;
  B();
}
```

local variable

```
B()
{
  int a=20;
  C();
}
```

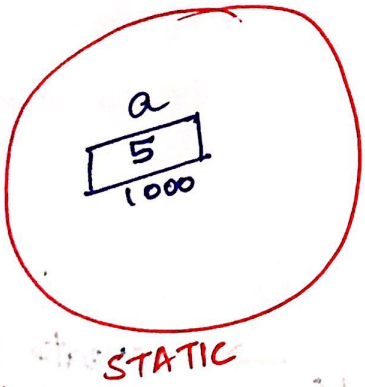
```
C()
{
  int a = 30;
  D();
}
```

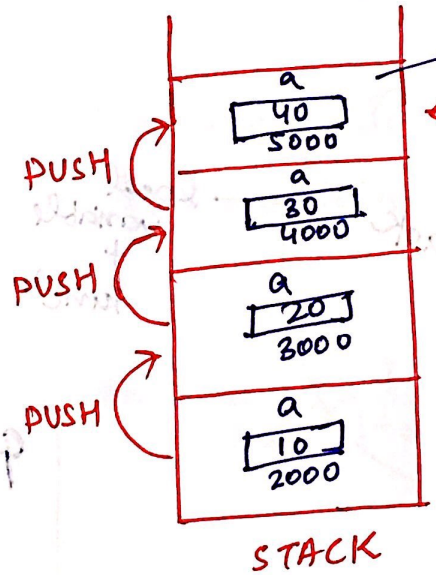
```
D()
{
  int a = 40;
  print(a);
}
```



- Global variable → create & allocated at compile time
- Local variable → create & allocate during runtime

STACK	STATIC	HEAP
- Default value = Garbage value	- Default value = 0	
- Local variable + fun^n calling	- Global variables	- variables allocated using "new" or "malloc" key word
- Large space	- small space	- small space





So, O/P = 40

But if D()
{
print(a); // no declaration for a
}

free variable

then scoping problem is occured.

- If scoping problem is handled by compiler at compile time then it is known as 'static scoping'.

Compiler checks for static variable in global scope and return it's value if exist.

- If scoping problem is handled by processor at run time then it perform POP opn to find the variable, this is known as 'Dynamic scoping'.

- So, Now O/p is:

Static scoping = 5

Dynamic Scoping = 30 from C()

eg:

```
int a;
main()
{
  int a=10;
  B()
}
B()
{
  C()
}
C()
{
  print(a);
}
```

O/p: Static scoping: 0 (Defalt value is 0)

Dynamic scoping: 10

eg:-

```
int a;
main()
{
  B()
}
```

```
B()
{
  c()
}
```

```
c()
{
  print(a)
}
```

O/p: static : 0

Dynamic : 0

eg:-

```
int a;
main()
{
  int a;
  B();
}
```

```
B()
{
  c()
}
```

```
c()
{
  print(a);
}
```

O/p static : 0

Dynamic : garbage value

eg:-

```
main()
{
  B();
}
```

```
B()
{
  c();
}
```

```
c()
{
  int a = 10;
  print(a);
}
```

O/P : 10 (No scoping problem)

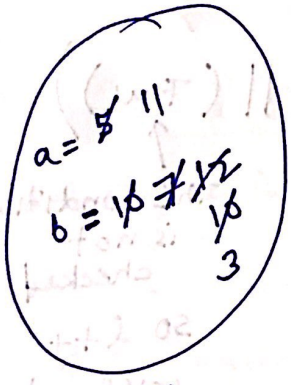
eg:- int a = 5, b = 10;

```
main()
{
    int a = 20;
    int b = 35;
    c();
    print(a,b);
    D(a);
}
```

```
C()
{
    int a = 7
    print(a,b);
    D(b);
    a=2, b=3;
}
```

```
int a
D()
{
    print(a,b);
    a=3, b=7;
    E();
    a=9; b=10;
}
```

```
E()
{
    print(a,b);
    a=11, b=12;
}
```



STATIC

a=7, b=10

a=10, b=10

a=5, b=7

a=20, b=35

a=20, b=3

a=11, b=7

DYNAMIC

a=7, b=35

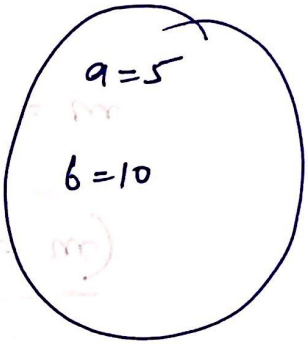
a=35, b=35

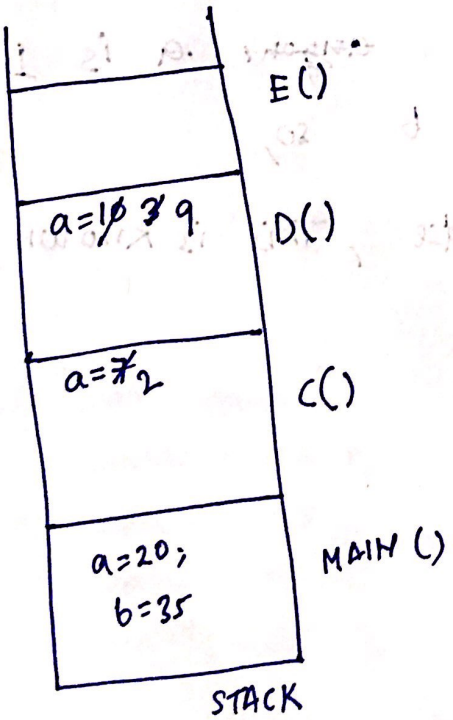
a=3, b=7

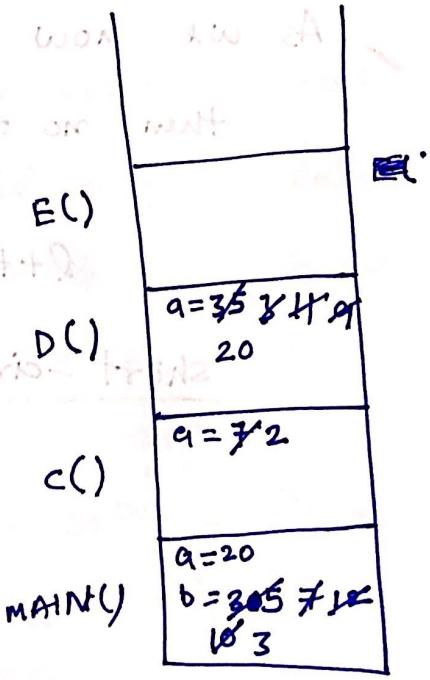
a=20, b=3

a=20, b=3

a=3, b=7







eg:-

```
main()
{
    int i=-1, j=-1, k=-1, l=2, m;
    m = ((i++ && j++ && k++) || (l++));
    print(i,j,k,l,m);
}
```

(Non-zero → TRUE, zero → FALSE)

m = ((True && True && True) || (True))

This condition is not checked so l++ will not execute l=2

(m = 1) so,

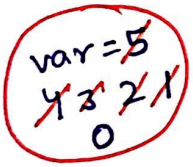
i = 0
j = 0
k = 0
l = 2
m = 1

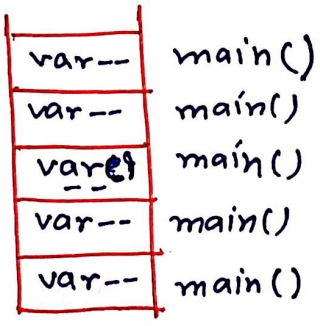
As we know in a OR b if a is 1 then no need to calculate b so, l++ will not execute, This is known as short-circuit

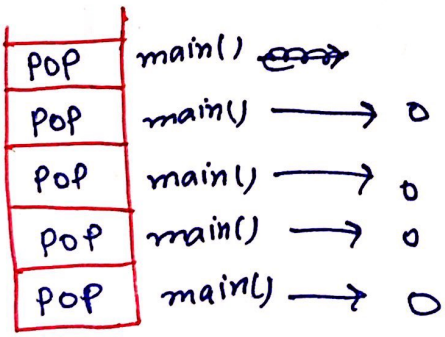
⇒ Static Variable :-

```
main()
{
   static int var = 5;
   if(--var)
   {
      main();
      print(var);
   }
}
```

"var" is a local variable, but by using static keyword we can force compile to create memory in static area at compile time.

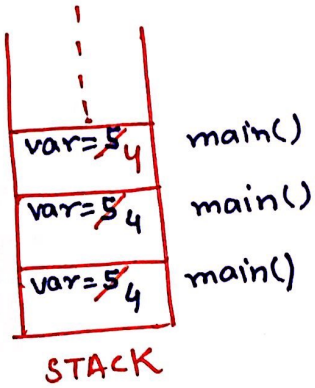






O/p :- 0000

```
main()
{
   int var = 5;
   if(--var)
   {
      main();
      print(var);
   }
}
```



O/p :- STACK overflow

Every time main() called, new local variable created with var=5, so termination condition i.e var=0 will never reached.

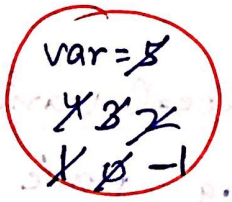
So, PUSH oph will continue infinitely until stack overflow happen.

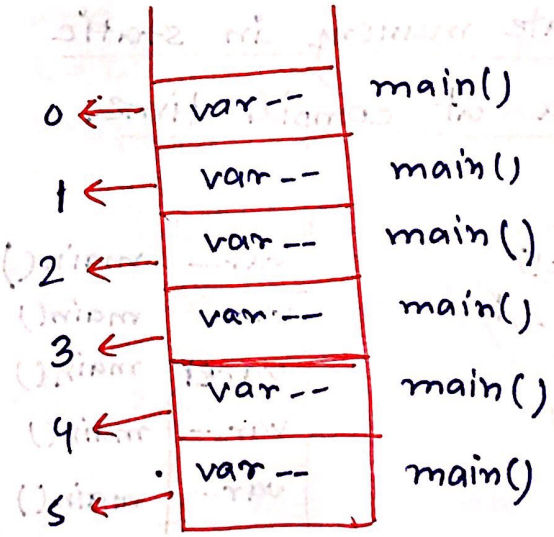
NOTE:- Memory creati only once for static variable during compile time. Procesor will not create extra stack variable during run time. It skips memory creation by seeing static token.

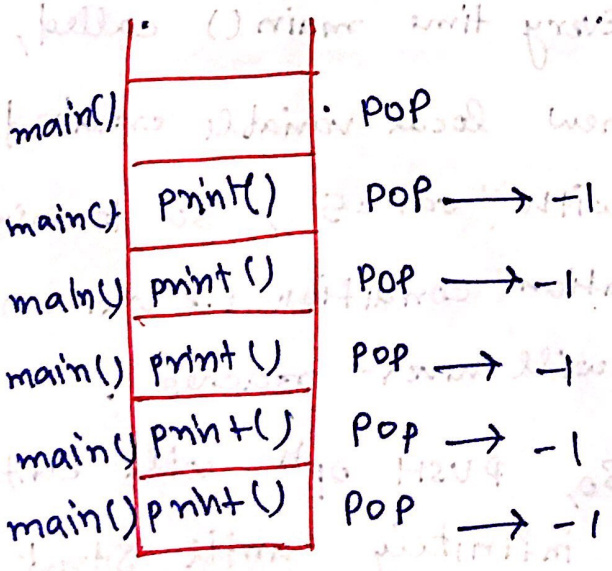
- For static variable initialization will be done only once.
- Default value for static variable = 0

eg:-

```
main()
{
  static int var = 5;
  if(var--)
  {
    main();
    printf(var);
  }
}
```







O/P:- -1 -1 -1 -1 -1

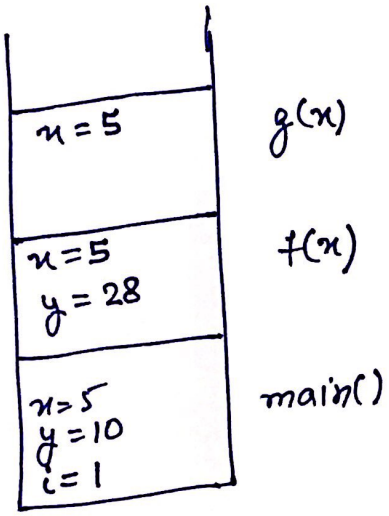
eg:-

```
main()
{
  int x=5, y=10, i;
  for( i=1; i<=2; i++)
    {
      y += f(x) + g(x);
      printf(y);
    }
}
```

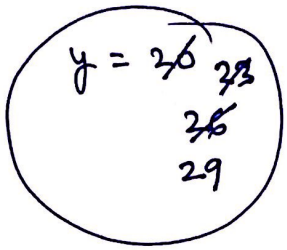
```
f(int x)
{
  int y;
  y = g(x);
  return(x+y);
}

g(int x)
{
  static int y = 20;
  y += 3
  return(x+y);
}
```

Initialization

modification



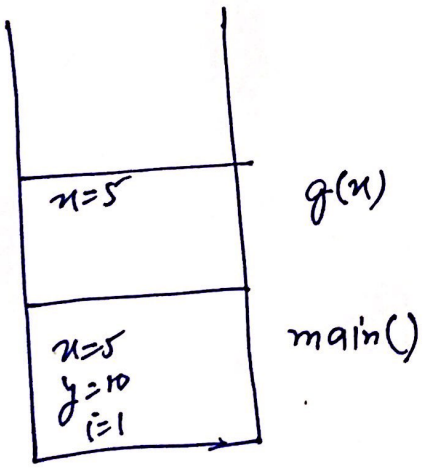
STACK

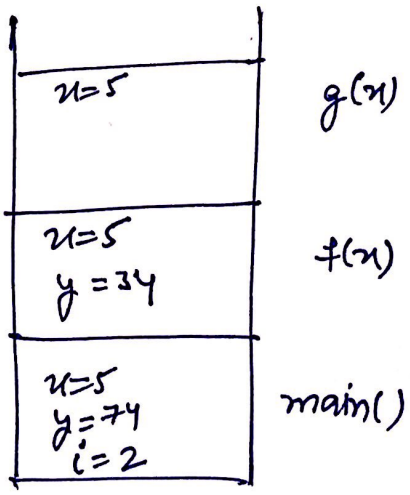


STATIC

$y += 33 + 31$

$y = 10 + 64 = 74$





O/p:- 74, 150

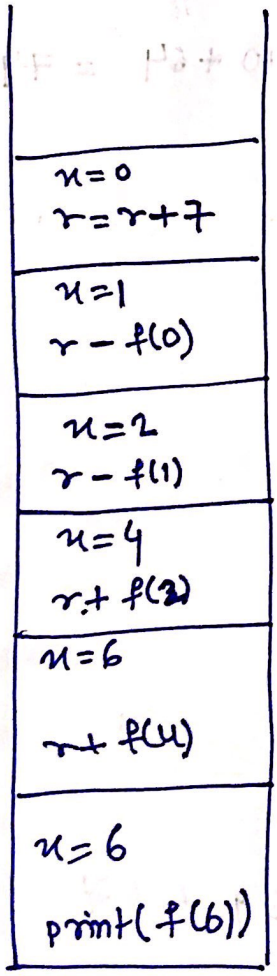
$y += 39 + 37$

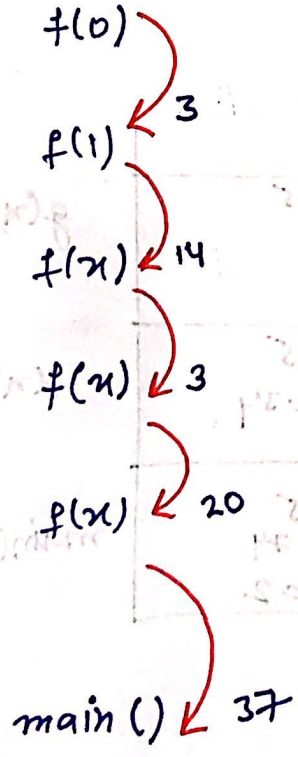
$y = 150$

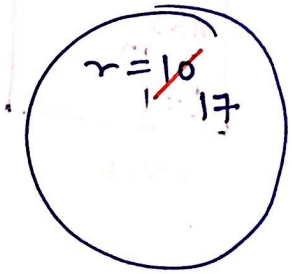
eg:-

```
f(int x)
{
  static int r = 10;
  if (x <= 0)
  {
    r += 7;
    return(3);
  }
  if(x > 3)
    return (r + f(x-2));
  else
    return (r - f(x-1));
}
```

```
main()
{
  int x = 6;
  printf("%d", f(x));
}
```



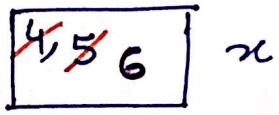


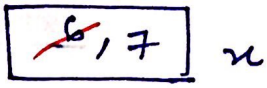


O/P:- 37

eg:

```
main()
{
    int x = 5 , y ;

    y = ++x * ++x * ++x

    printf(x,y);
}
```





O/p :- = (6 × 6) * 7

= 36 * 7 = 252

(y = ++x * ++x * ++x)

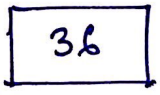
↳ y = (++x * ++x) * ++x

x=5 x=6

unary binary

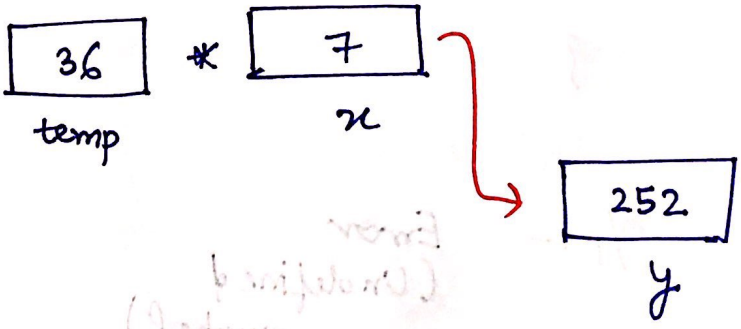
so, after unary op^n x = 6

so, x * x = 6 × 6 = 36



Then ++x ⟶ (x = 7)

so,



Extern Variable :-

eg:-
```
int a = 5;
main()
{
  extern int a;
  printf(a);
}
```

O/p :- 5

```
int a = 5;
main()
{
  int a;
  printf(a);
}
```

O/p :- Garbage value

Extern keyword tell compiler to get memory ~~outside~~ global with same name

so, no memory created ~~for~~ this extern variable.

eg:-
```
main()
{
  extern int a;
  printf(a);
}
```

O/p :- Error (Undefined symbol)

eg:-
```
int a;
main()
{
  extern int a;
  printf(a);
  a=10;
  printf(a);
}
```

O/p :- 0
10

eg:-
```
extern int a;
main()
{
  extern int a;
  printf(a);
}
```

O/p:- Error (Undefined symbol a)

eg:-
```
extern int a=10;
main()
{
  extern int a;
  printf(a);
}
```

O/p:- 10

Note:- If we declare variable with extern then no memory allocated.

If we initialize variable with extern the memory allocated.

eg:-
```
extern int a=10;
main()
{
  extern int a=6;
  printf(a);
}
```

O/p:- Error

Extern can't initialize var inside func^n body

GLOBAL

eg:-
```
extern int a=10;
main()
{
  extern int a;   // LOCAL
  static int a;   // LOCAL
  printf(a);
}
```

O/p:- Error

Ambiguity in a.

```
int a = 5 , b =10;

main()
{
   extern int a ;
   register int b = 20;
    C();
    printf( a,b);
    D();
}
```

```
D()
{  extern int b;
   static int a = 7;
   printf(a,b);
   E();
   a=4; b=5;
}
```

```
C()
{
  static int b;
  printf(a,b);        a → free variable
  D();
  a=1; b = 2;
}
```

```
E()
{
  auto int b=  23;
  printf ( a,b);      a → free variable
    a=4; b =5;
}
```

O/P:-

STATIC SCOPING	DYNAMIC SCOPING
5,0	5,0
7,10	7,10
5,23	7,23
1,20	1,20
4,5	4,5
1,23	4,23

(SWITCH)

eg:
```
main()
{
  int i;
  for(i=1; i<=25; i++)
  {
    switch(i)
    {
      case 1 : i+=3;
      case 2 : i+=4;
      case 3 : i+=5;
      default : i+=3;
                break;
    }
    printf(i);
  }
}
```

O/p:- 16
20
24
28

- Switch case statement, break is required after each case otherwise it will execute all cases sequentially.

- Default position in not fixed in switch, it can be placed anywhere and it will execute when there is no match there.

- Default (Last statement) ⟶ no break needed

  Default (middle statement) ⟶ break needed

Side Effect :-

```
int i = 4;
printf( ++i, ++i, ++i);
```

5, 6, 7 →

← 7, 6, 5

suffers side effect

(Answer different for L→R & R→L)

```
int i = 4;
printf( i+1, i+2, i+3)
```

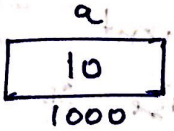
5, 6, 7 →

← 5, 6, 7

No side effects

Pointer :-

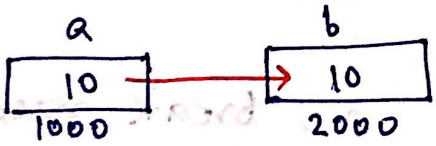
```
int a = 10;
```



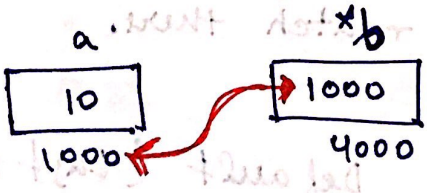
printf(a) → 10 (data)

printf(&a) → 1000 (Address)

```
int b = a;
```




```
int *b = &a;
```



int b = &a

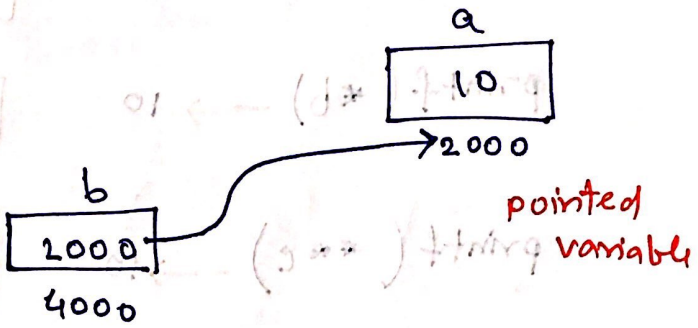
since size of b is 2 Bytes, but size of address is depends upon system memory

so, if address size > size of b it can't store in b.

so, we use pointer,

dereference operator

int *b = &a ; i.e



so, printf(b) = 2000

printf(&b) = 4000

printf(*b) = 10

So, pointer is a special variable use to store address of other variables.

'*' dereference operator can only used with variables which address of some other variables.

eg :-

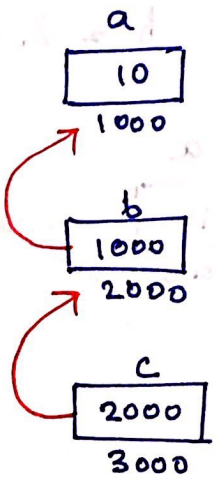
```
int a = 10;
int *b = &a;
```

✓ printf(*b);

✗ printf(*a); } a is normal variable which carry data not address

eg:-

```
int a = 10;   — var
int *b = &a;  → pointer
int **c = &b; → double pointer
```



printf(a) ⟶ 10 [Immediate mode]

printf(*b) ⟶ 10 [Direct addressing]

printf(**c) ⟶ 10 [Indirect addressing]

eg:-

```
int a = 10;
int *b = &a;
printf(a*b)
```

10 10

O/P:- INVALID op^n as there is no operation b/w a & *b

compiler checks declaration for *b, and if it present then * behaves as dereference operator otherwise multiplication.

eg:-

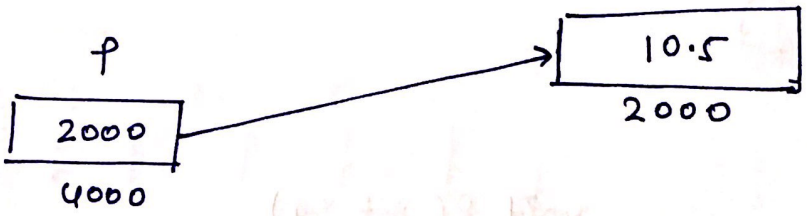
```
int a = 10;
int *b = &a;
int **c = &b;
```

size of (c) → 8B

size of (*c) → 8B

size of (**c) → 2B.

eg:- P is float pointer



size of (p) → 8B

size of (*p) → 4B

print (p+1) → 2000 + 4 = 20004

print ((*p)+1) → 10.5 + 1 = 11.5

eg:-

```
int i=5, j=10;
main()
{
  printf(i,j);
  fun(&i,&j);
  printf(i,j);
}
```

```
fun(int *j, int *i)
{
  *i = 7;
  j = *i+3;
  i = j;
  *j = 4;
}
```

O/p :- 5;10
4;7

eg:-

```
main()
{
  int a=10;
  int *b = &a;
  scanf("%d", b);
  printf(a+25);
}
```

// scanf("%d", &a);

address needed

O/p :- I+25

eg:-

```
int x;
void Q(int z)
{
  z += x;
  printf(z);
}
```

```
void P(int *y)
{
  int x = *y + 2;
  Q(x);
  *y = x-1;
  printf(x);
}
```

```
main()
{
  x=5;
  P(&x);
  Printf(x);
}
```

O/p :- 12 7 6

eg:-

```
f( int x, int *p)
{
   *p = x;
    x=10;
}
```

```
main()
{
  int a=5, b=6;
  int *p = &a, **q;
  *p = 20;
   q = *&p;
  f(a, &b);
  *q = &b;
  *p = 30;
  f(a, &b);
  print(a, b);
}
```

O/p :- 20 20

eg:-

```
main()
{ int a[] = {25, 30, 90, 55, 22, 63};
  printf("%d", f(a, 6));
}

f( int *a, int n)
{
   if( n<=0)
       return (0);
   else if ( *a % 2 == 0)
       return ( *a - f(a+1, n-1));
   else
       return ( *a + f(a+1, n-1));
}
```

O/p :- -21

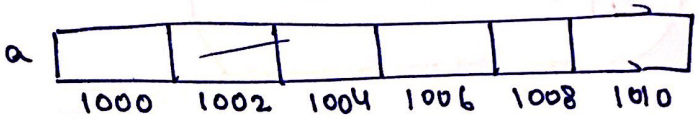
- $a[i] = *(a+i) = *(i+a) = i[a]$

- $a[i][j] = b[j] = *(b+j) = *(a[i]+j) = *(*(a+i)+j)$

  ($a[i]$ = b)

- Base address of array which is itself a pointer, can't be modified.

- int a[6];

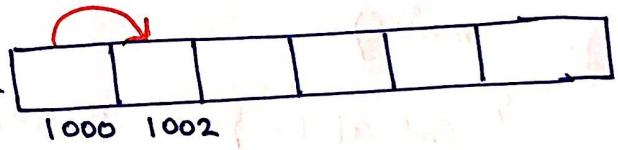


print(a) ⟶ 1000

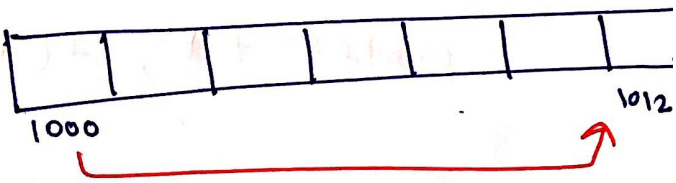
print(&a) ⟶ 1000

But,

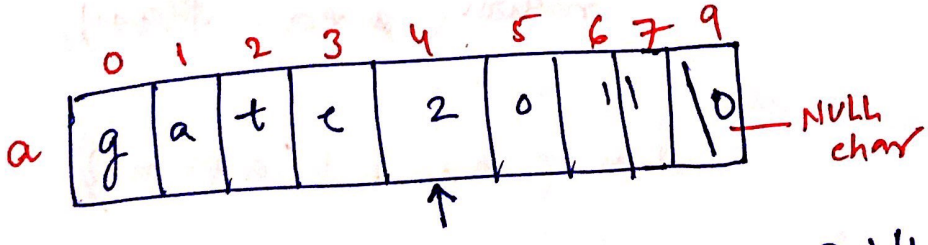
skip element: print(a+1) ⟶ 1002



skip array: print(&a+1) ⟶ 1012



eg:-
```
main()
{
  char a[] = "gate2011";
  printf("%s", a + a[3] - a[1]);
}
```



so, $a + e - a = a+4$

O/p :- 2011

s = "abcd" ⟶ sizeof(s) = 5 (Include NULL char)

strlen(s) = 4

= printf("%s", base address); print all char from BA to null char.

= printf("%c", base address); print char at BA only

so,

printf("%s", "gate2011") ⟶ gate2011

printf("%s", a) ⟶ gate2011

printf("%s", a+2) ⟶ te2011

printf("%c", a+3) ⟶ e

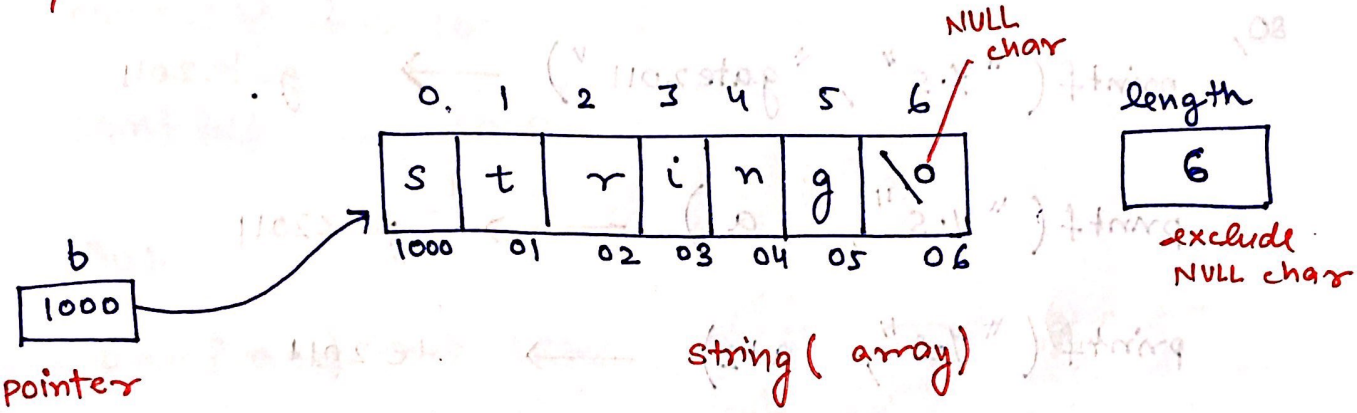
printf("%d", a+1) ⟶ 97

printf("%c", *(a+1)+1) ⟶ b

*(a+1) = a[1]

eg:-

```
main()
{
    char a[10];
    char *b = "string";
    int length = strlen(b);
    for(i=0; i<length; i++)
    {
        a[i] = b[length-i];
    }
    printf("%s", a);
}
```



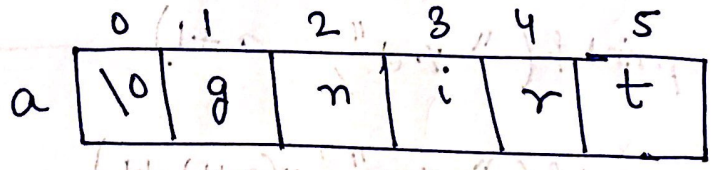
*b → s

*(b+1) → t

*(b+2) → r

⋮

*(b+5) → g



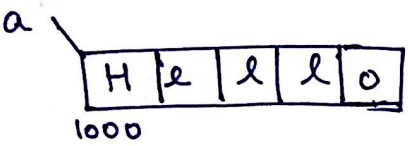
printf("%s", a)

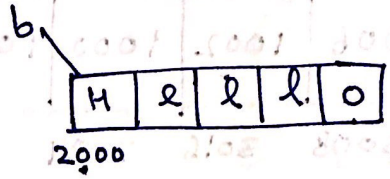
↳ *. Nothing as starting char is itself a '\0' char.

eg:-

```
main()
{ char a[] = "Hello";
  char b[] = "Hello";
  if (a == b)
  {
    printf("hi");
  else
    printf("bi");
}
```

O/p :- Error





a = b

assignment operator

since a and b carry base address which can't be change

so, (a = b) can't be performed

```
if ( a == b)
```

as a = 1000
b = 2000

O/p :- "bi"

```
if ( *a == *b)
```

*a = H

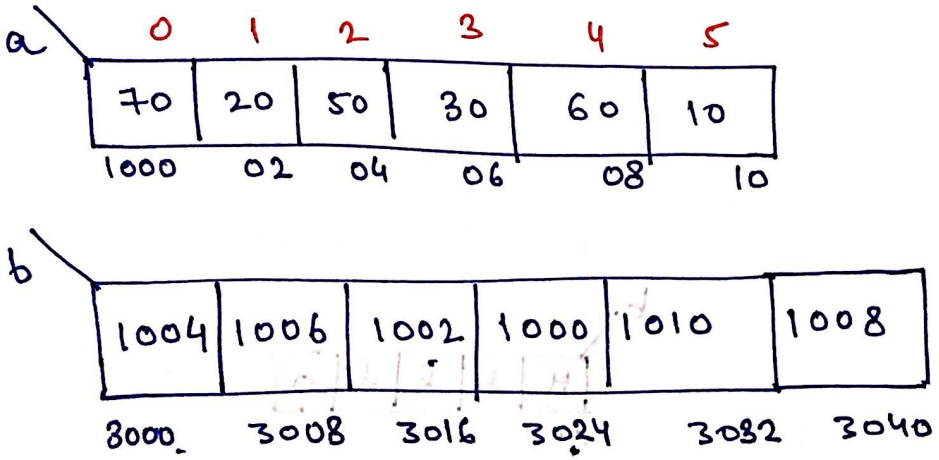
*b = H

O/p :- "Hi"

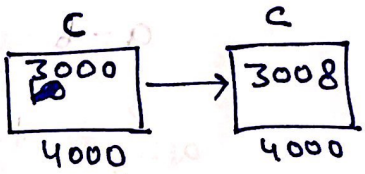
Array of pointers

eg:-

```
main()
{
    int a[] = {70, 20, 50, 30, 60, 10};
    int *b[] = {a+2, a+3, a+1, a, a+5, a+4}
```



priority:
(++ > *)




```
    int **c = b;
        *c++;  ⟶  c = 3008

    printf(c-b, *c-a, **c);  ⟶  (3008-3000)/8 , (1006-1000)/2 , 30
                                (1, 3, 30)
        **++c;

    printf(c-b, *c-a, **c);  ⟶  (2, 1, 20)
        ++**c;

    printf(c-b, *c-a, **c);  ⟶  (2, 1, 21)
        ++*++*c

    printf(c-b, *c-a, **c);  ⟶  (2, 2, 51)
}
```

eg:-

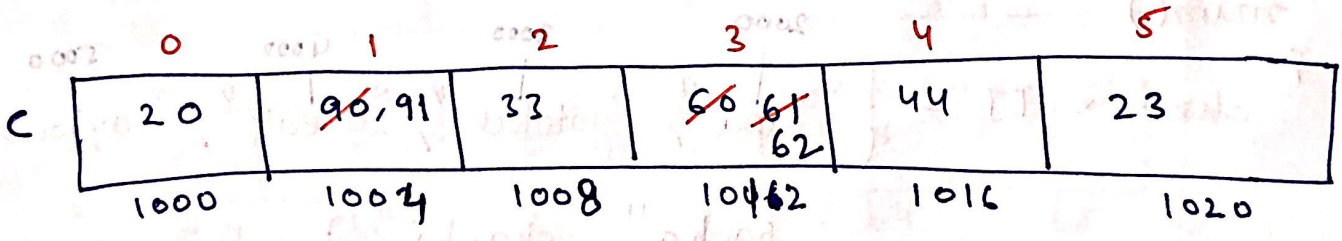
```
main()
{
    float c[] = {20, 90, 33, 60, 44, 23};
    float *a[] = {c+3, c+2, c+1, c, c+5, c+4};
    float **b = a;
    ++**(b+2);
    printf("%f, %f, %f", b-a, *b-c, **b);
    ++*++*++b;
    printf("%f, %f, %f", b-a, *b-c, **b);
    ++**b+9;
    printf("%f, %f, %f", b-a, *b-c, **b);
}
```

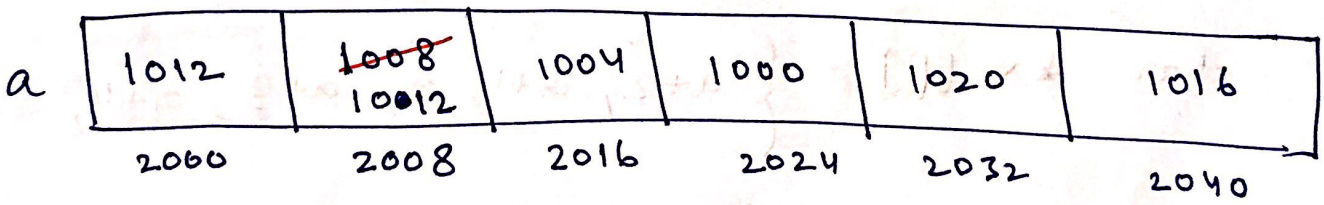
o/p:-

(0, 3, 60)

(1, 3, 61)

(1, 3, 62)





NOTE:- We can't add pointer with any other Numeric data-type.

- We can substract two pointers eg:- $Ptr_1 - Ptr_2$

  i.e no. of position Ptr_1 ahead of Ptr_2

- Adding const to pointer : Ptr + 5 (Move 5 steps forward)

  Substract const to pointer : Ptr − 5 (Move 5 steps backward)

- We can't perform Addition / Multiplication / division because they had no meaning.

eg:-

```
main()
{
char *a[] = { "Papa", "gotohell", "break", "ayyooo",
              "bacha", "chachi" };

char **b[] = { a+2, a+1, a, a+3, a+4, a+5 }

char ***c = b;

++**++*c+4;
```

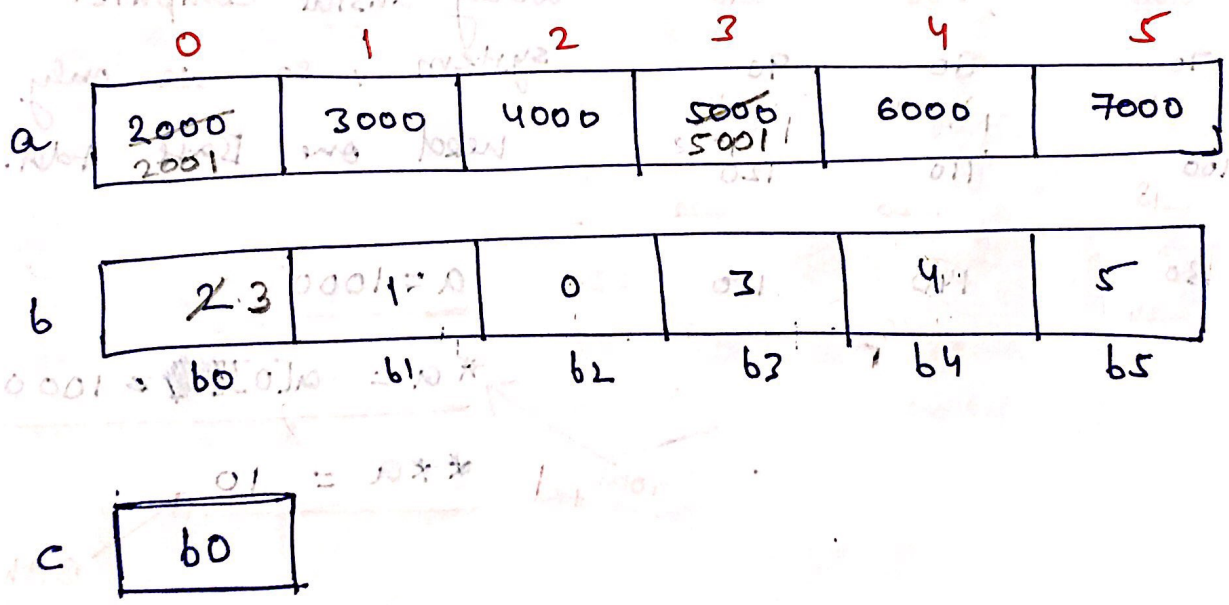
2000 3000 4000 5000 6000 7000

printf("%s", ++**(c+2)+2); → a

printf("%s", ++*++*c+3); → a

printf("%c", **++*++c+2); → d

}



2-D array pointer

eg:-

```
main()
{
  int a[5][3] = {10,20,30, ------150};
  printf("%d", ( a == *a) && ( a[0] == a));  → 1
  printf("%u", *((a+2)+3)+3);  → 1036
  printf( *a+10, a[10]+3, **a+30);  → 1020, 1066, 40
  printf( *(a+3)+10, **(a+3)+60);  → 1038, 160
}
```

a	0	1	2
0	10 (1000)	20 (02)	30 (04)
1	40 (06)	50 (08)	60 (10)
2	70 (12)	80 (14)	90 (16)
3	100 (18)	110 (20)	120 (22)
4	130 (24)	140 (26)	150 (28)
5			

row-major order

since multidimensional array stored as 1-D array inside computer system, so we only need one Base Add.

a = 1000

*a = a[0] 1000

**a = 10

row selected only

Both row & column selected

a = 1000

a+1 = 1002 (6B)
= 1000 + 2(row size) (skip-row)
= 1006

Now, $a+1 = 1006$

$*(a+1) = 40$ ~~40~~

$= 1006$

only row selected, column has to be selected

$*(a+1)+1 = 1008$

Both row & column selected

row, colm

$*(*(a+1)+1) = 50$ Data at 1008

$(a+1)+1 = 1006+1006$

$= 1012$

skip ~~Add~~ row 2 times

$*((a+1)+1) = 1012$

$*((a+1)+1)+1 = 1014$

Row, column selected

$*(*((a+1)+1)+1) = 80$

So,

$a = 1000$

$\&a = 1000$

for array both are same.

$a+1 = 1006$ — skip 1 row

$\&a+1 = 1030$ — skip whole array

$*a+1 = 1002$

```
main()
{
  int a[4][5][3];
  printf( *(a+3) +7);      ⟶ 1132
  printf( *& a[4][2] +20);  ⟶ 1160
  printf( **a +30, *(*(a+2) +2) +10 ⟶ 1060, 1092
  printf( *(*a+7) +10)      ⟶ 1062
}
```

$a := 1000$

matrix size = 15 × 2B = 30B

row size = 3 × 2B = 6B

$= *(a+3) + 7$

$= 1000 + 3(30) + 7(6)$

$= \boxed{1132}$

$= *a[4] + 20$

$= **(a+4) + 20$

$= 1000 + 4(30) + 20(2)$

$= \boxed{1160}$

$= **a + 30 \;,\; *(*(a+2)+2) + 10$

$= \boxed{1060} \;,\; 1000 + 2(30) + 2(6) + 10(2)$

$\boxed{1092}$

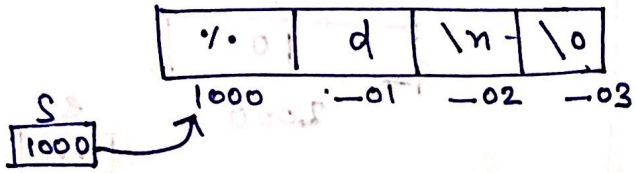
$= *(*a+7) + 10$

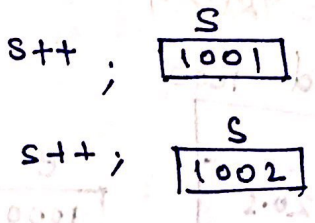
$= 1000 + 7(6) + 10(2) = \boxed{1062}$

eg:-

```
main()
{
    char *s = "%d\n";
    s++;
    s++;
    printf(s-2, 300);
}
```

1002 − 2 = 1000





↳ printf("%d\n", 300)

= 300

⇒ Structure & Union :-

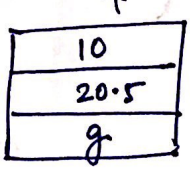
Structure

eg:-

```
struct node
{
    int a;     — 2B
    float b;   — 4B
    char c;    — 1B
}                    } 7B
```

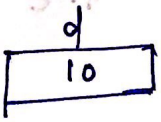
User data type is created by d (definition)

```
struct node d = {10, 20.5, 'g'};
```



User defined datatype

```
int d = 10;
```



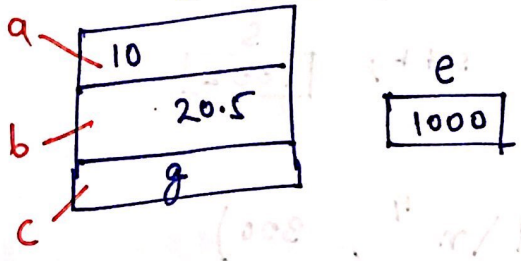
pre-defined datatype

printf("%d" , d) ⟶ 10 (int d is accessed)

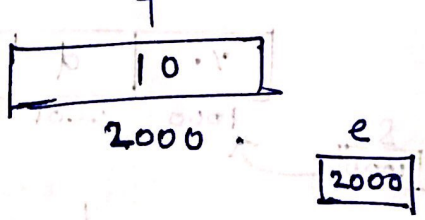
printf("%d" , d.a) ⟶ 10 (node d is accessed)

member access operator

STRUCT



INT




```
struct node *e = &d;

printf("%u", e);
    ↳ 1000

printf("%d", (*e).a);
    ↳ 10

printf("%f", (*e).b);
    ↳ 20.5

printf("%c", (*e).c);
    ↳ g

or

print("%c", e→c);
    ↳ g
```

```
int *e = &d;

printf("%u", e);
    ↳ 2000

printf("%d", *e);
    ↳ 10
```

NOTE:-

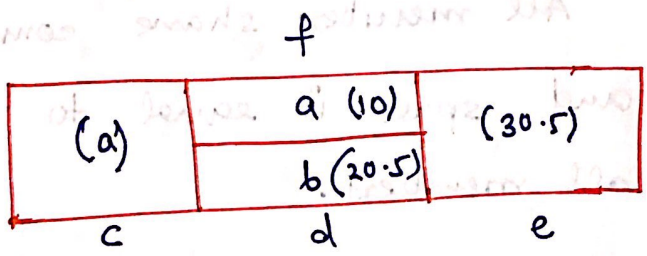
Priority:-

(. > *)

eg:-

```
struct s1
{
   int a;    2B
   float b;  4B
};
```

```
struct s2
{
   char c;        — 1B
   struct s1 d;   — 6B
   float e;       — 4B
};
```

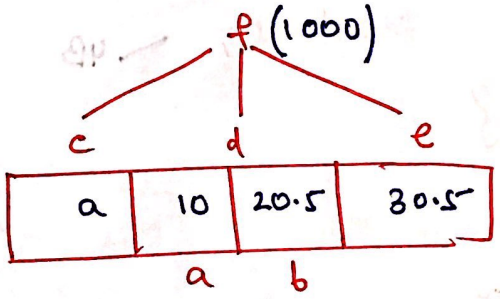


struct s_2 f = { 'a', {10, 20.5}, 30.5 };

printf(" %f ", f.d.b) ⇒ 20.5

printf(" %f ", f.e) ⇒ 30.5

struct s_2 * g = &f;



printf(" %f ", (*g).d.b) ⇒ 20.5

printf(" %f ", (g→d).b) ⇒ 20.5

printf(" %f ", g→e) ⇒ 30.5

printf (g) ⇒ 1000

printf (g+1) ⇒ 1000 + 11 = 1011 (SKIP whole struct node)

Union :- All member share common space, and space is equal to max size of all members.

In this one of the member is active at a time.

```
union u1
{
    char c;      — 1B
    struct node d; — 6B
    float e;     — 4B
}
```

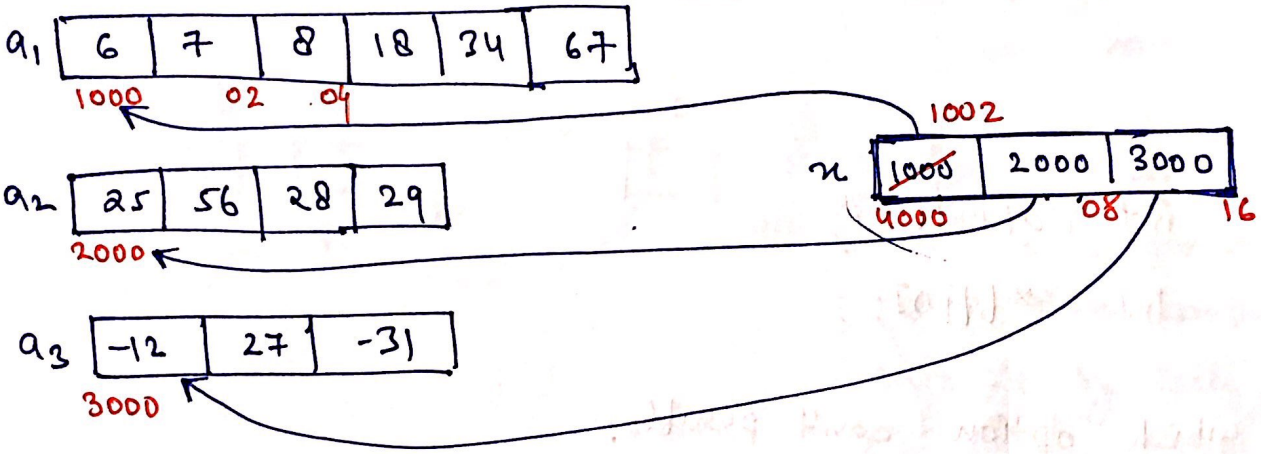
← max (1, 6, 4) ⇒ 6B

so, **size of (u1) = 6**

eg:-

```
int a1[] = {6, 7, 8, 18, 34, 67};
int a2[] = {25, 56, 28, 29};
int a3[] = {-12, 27, -31};
```

```
int *x[] = { a1 , a2 , a3 };

f(int **a)
{
    printf( a[0][2]);  ⇒ 8
    printf( *a[2]);    ⇒ -12
    printf( *++a[0]);  ⇒ 7
    printf( *(++a)[0]); ⇒ 25
    printf( a[-1][1]);
}

main()
{
    f(x);
}
```



$a[0][2] = *(*(a+0)+2) = *(*a + 2) = *1004$ (*a = 1000; 1004)

$= 8$

$*a[2] = *(*(a+2)) = -12$ (*(a+2) = 3000)

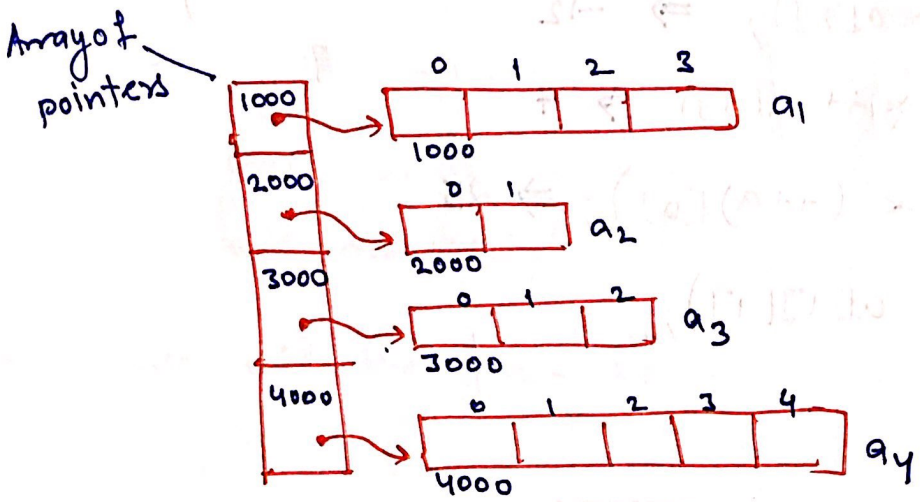
$*++a[0] = *++*a = 7$ (*a = 1000)

$*(++a)[0] = *2000 = 25$ $\quad a = 2000$

$a[-1][1] = *(*(a-1)+1) = 8$ (*(a-1) = 1002; 1004)

Note:- Array of pointers can be used as 2D array format.

- We can creati an 2-D array of non-uniform columns using array of pointers.



Q: int a[10][5];
int *b[10];

which option can't possible:

(a) a[5][3] = data

(b) a[5] = address

(c) b[5] = address

(d) b[5][2] = data

a[5] = *(a+5) → address of row of 2D array which can't be changed.

Type Casting :-

Conversion of one data-type to another during some operations.

Implicit :-

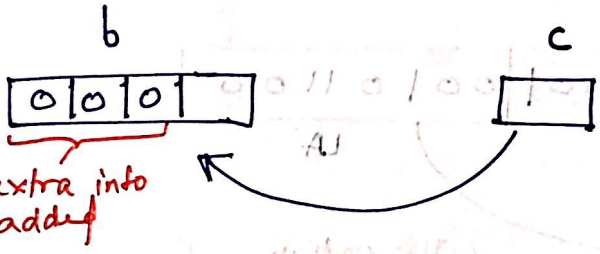
```
int a;
float b;
char c;
b=c;
```

b = c

float — 4B

char — 1B

(small size var ⟶ Large size var)



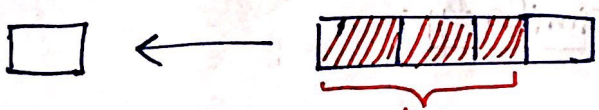
storing small size into into bigger storage space then compiler cast it by itself.

Explicit :-

```
c = b;
```

char — 1B

float — 4B



Large data ⟶ small space

in this ~~will~~ compiler generates error.

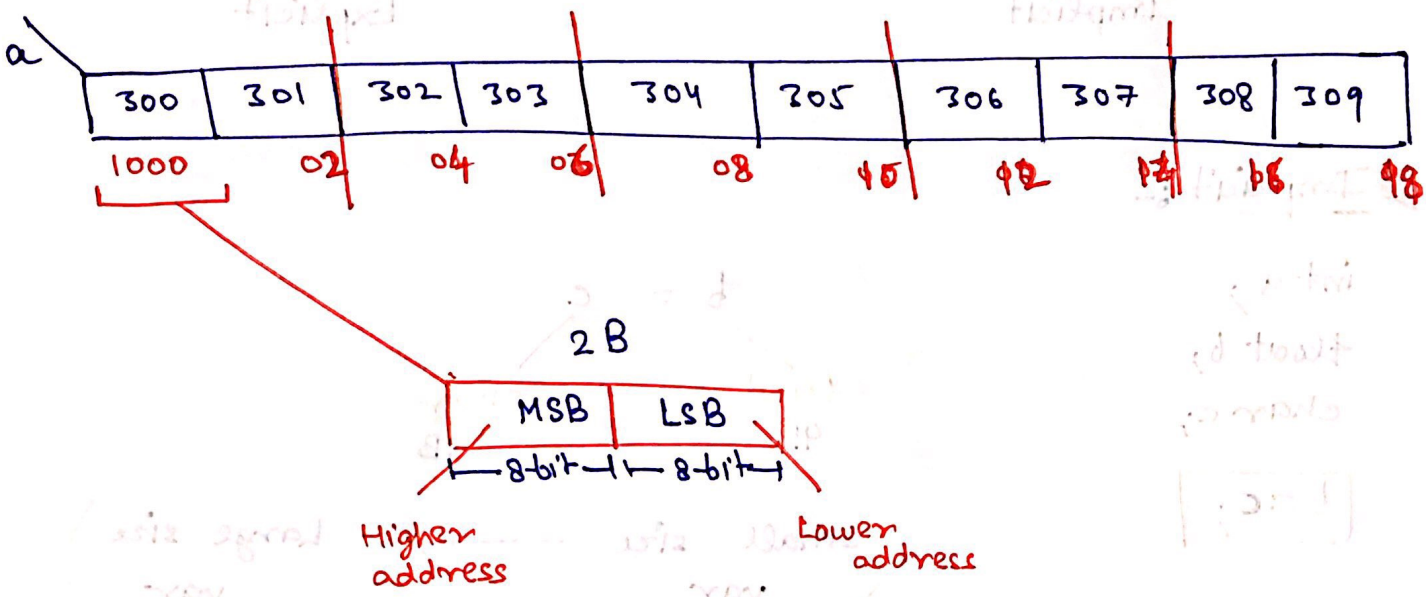
So, user has to done it explicitly.

```
c = (char) b;
```

target type

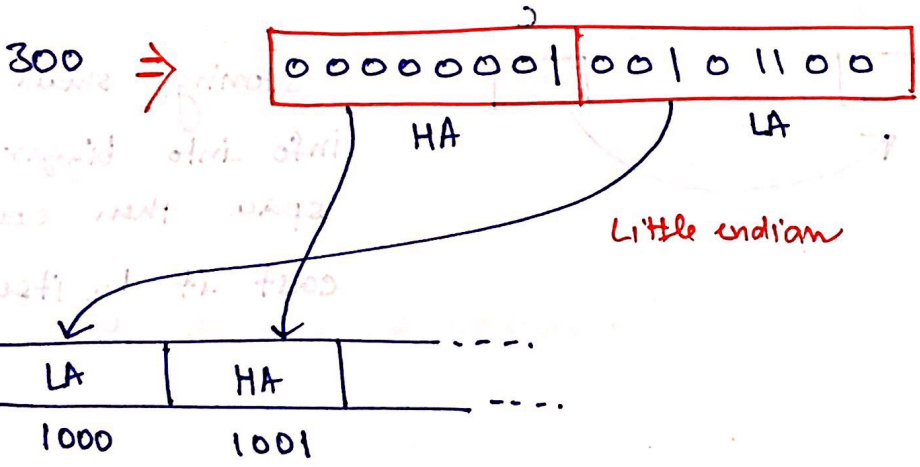
= Type casting in pointers :-

```
int a[10] = {300, 301, 302, ....., 309};
```

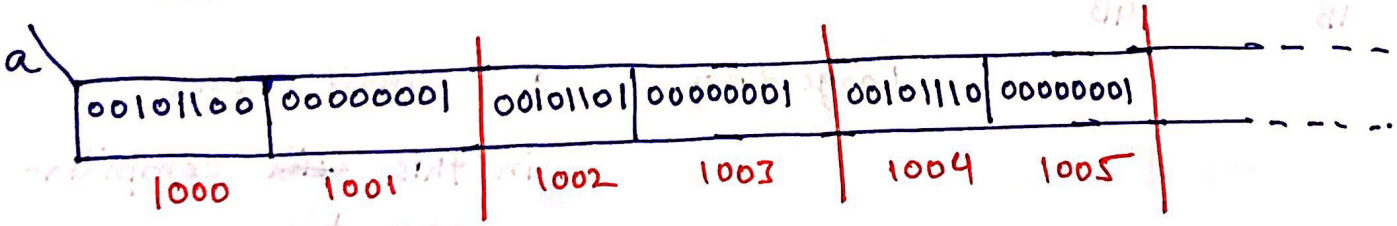


Higher address

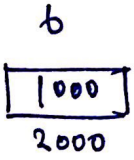
Lower address



similarly :-

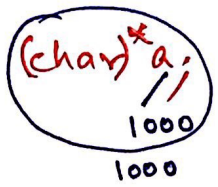


`int *b = a;`

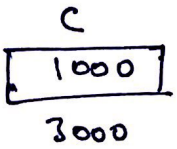


No type casting required (same type).

`char *c = (char*)a;`



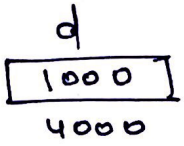
int a is address which is of 8B and after type casting it is an char address of 8B



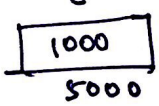
so, same size.

`float *d = (float*) a;`

INT add a → FLOAT address
1000 (8B) 1000 (8B)



`void *e = a;`



INT	CHAR	FLOAT	VOID
b ⇒ 1000 *b ⇒ 300 Read 2B	c ⇒ 1000 *c ⇒ 44 Read 1B only	d ⇒ 1000 *d ⇒ data Read 4B	e ⇒ 1000 *e ⇒ X can't read B as it don't know how many B have to Read
b+1 ⇒ 1002 MOVE 2B	c+1 ⇒ 1001 MOVE 1B	d+1 ⇒ 1004 MOVE 4B	e+1 ⇒ X can't mov c=(char*)e so a void pointer can be cast into any type