Programming Language

Topics :

- Programming In C
  - Function
  - Recursion
  - Pointers
  - Storage Classes
- Parameter Passing Technique
  - Call by Value
  - Call by Reference
  - Call by Value Result (or) Copy Result
  - Call by Name
  - Call by Text
  - Call by Need.
- Scope
  - Static Scope
  - Creating Static Link
  - Dynamic Scope
- Binding

Books:

Principles of Programming Lang
— Horowitz.

Storage Classes:

→ In C every variable processes some characteristics, like data type, storage area, scope and life time.

→ Storage Classes describe scope and life time of a variable

→ C support following storage classes.

1) Auto
2) Static
3) Register
4) Extern.

1) Auto :→ Syntax to declare auto variable is

auto datatype variable name ;

• auto int i

→ By default every variable in c is auto variable

→ If auto variable is not initialized, by default it contains garbage variable.

→ Auto variable has function life time and it has body scope (i.e auto variable is accessable within the body where the variable is declared)

→ It is recreated and re-initialized for every function call.

void xyz()
{

```
void xyz( )
{
  auto int a = 15;
    ++a;
  printf("%d", a);
}
void main ()
{
  xyz();
  xyz();
  xyz();
  printf("%d", a);
}
```

16, 16, 16

```
void xyz();  — Line ①.
void main()
{
  xyz();
  =
}
void xyz()
{
  =
  =
}
```

↳ o/p: Undefined symbol a;

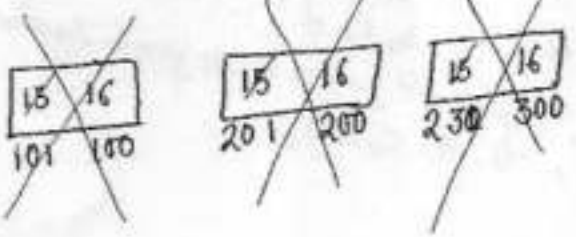
3,

Program

↓ Compilation → Top down.

Execution → main

In the above program if we remove line ①, then it generate compile time error.



→ Automatically these value will be destroyed after execution the function xyz().

```
void main()
{                                   scope: B1
    int a, b;                       B1 - B2
    {                               B2
        int b, c;                   B2
        {                           B3
            int d;
        }
    }
}
```

↳ after this parenthesis all the variables will be destroyed.

```
void main()                         B1-B2   B1-B3
{   int a=15, b=20;
    {   int a=12, c=18;
        printf("%d %d %d", a, b, c);        12  20  18
        {   int b=25, d=30;         B3  B3
            printf("%d %d %d %d", a, b, c, d);      12  25  18  30
        }
        printf("%d %d %d %d", a, b, c, d);          12  20  18  undefined symbol
    }
    printf("%d %d %d %d", a, b, c, d);              15  20  undefined symbol.
}
```

→ Static :

→ Syntax to declare static variable is :

static datatype variable_name;

• static int i ;

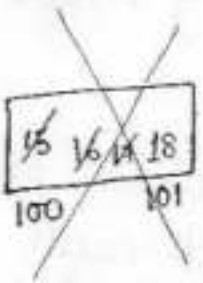
→ If static variable is not initialized, by default it contains the value 0.

→ It created only onces and persisting previous stage value from the distraction of several function calls.

→ It has program life time and function scope.

```
void xyz()
{
  static int a=15;
    ++a;
    printf("%d", a);
}
void main()
{
  xyz();
  xyz();
  xyz();
  printf("%d", a);
}
```

→ 16, 17, 18

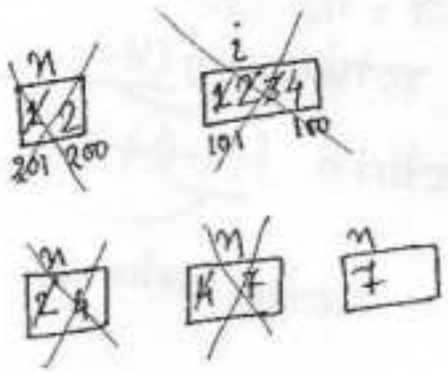


→ Undefined Symbol a ;

↳ After reaching this, the veriable a will be destroyed.

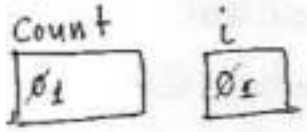
→ What is the return value of f(1) ?

```
int f(int n)
{
  static int i=1;
  if (n>>5)
    return n;
  n = n+i;
  i++;
  return f(n);
}
```



Time Complexity =

```
int incr (int i)
{
  static int count = 0;
  count = count + i;
  return (count);
}
main ()
{
  int i, j;
  for (i = 0; i <= 4; i++)
    j = incr (i);
}
```

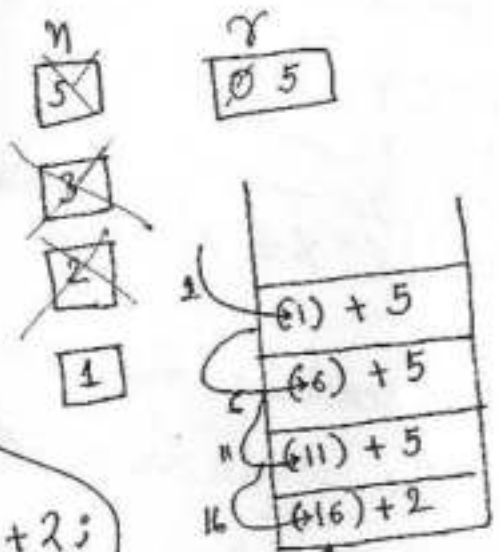


i	0	1	2	3	4
Count	∅ 0	1	3	6	10
d	0	1	3	6	(10)

∴ final value of J = 10.

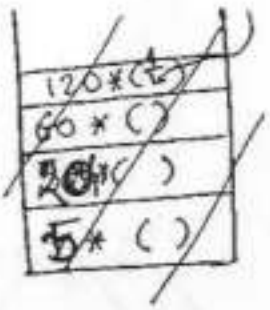
If in the above prob. if line ① then the value stored in variable j at the end of execution = 4.

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



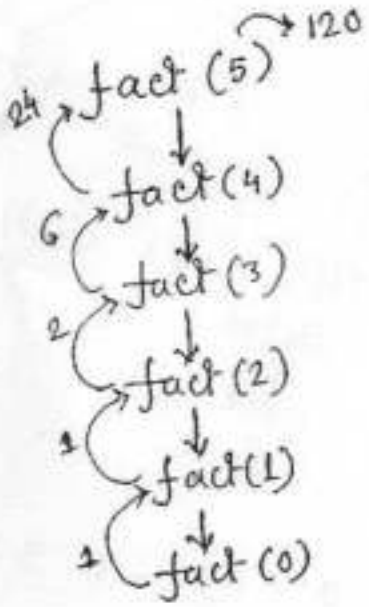
What is the return value of f(5) = ?

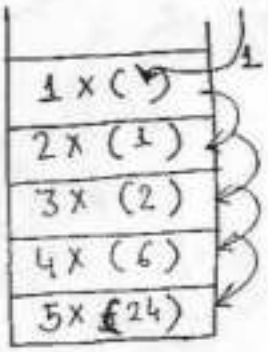
```
int fact (int n)
{
  if (n==0)
    return 1;
  else
    return n*fact(n-1);
}
```



f(5) = ?

= O(n).





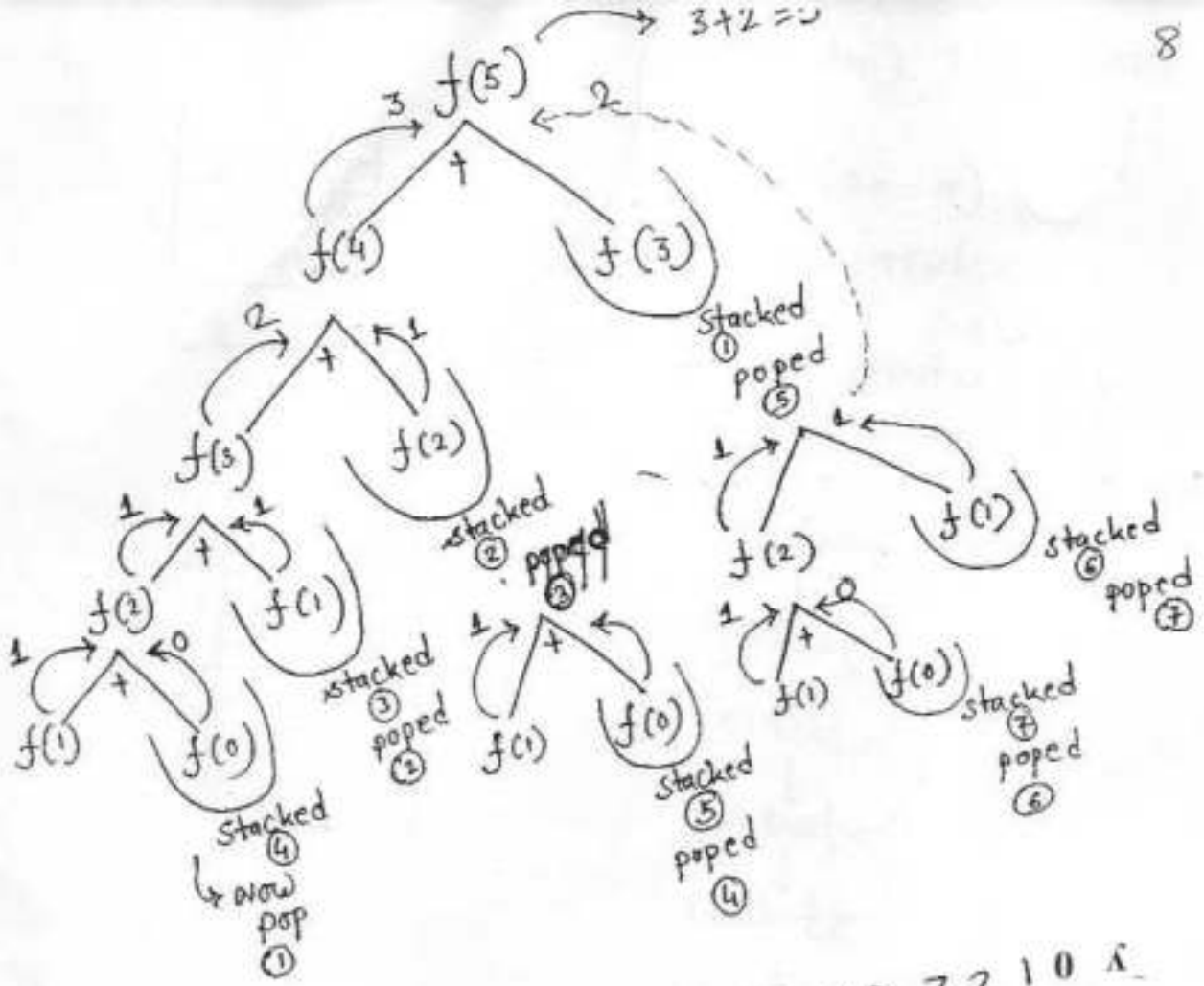
→

```
int f(int n)
{ if (0---0)
    return 0;
  if (n==1)
    return 1;
  else
    return f(n-1)+f(n-2);
}
```

i) what is the return value of f(5)

ii) What is the [X] bound to the no. of fun on any i/p size n?

a) O(n) b) O(n²) c) O(2ⁿ) d) O(



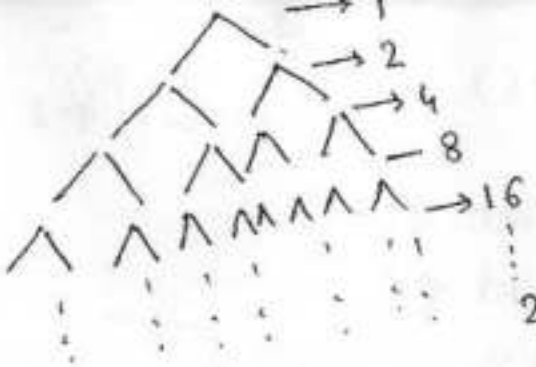
I/P seq: 5 4 3 2 1 0 1 2 1 0 3 2 1 0 ∧

n	f. calls	options
5	15	= a) $O(5)$ X
		b) $O(25)$ X
		c) $O(32)$ ✓
		d) $O(\log_2^5)$ X.

n	2^n	n^2
1	2	1
2	4	4
3	8	9
4	16	16 ←
5	32	25
6	64	32
7	128	49
⋮		

So: Ans: $O(2^n)$.

for $n = 5$



$1 = 2^0$

$2 = 2^1$

$4 = 2^2$

$8 = 2^3$

$16 = 2^4$

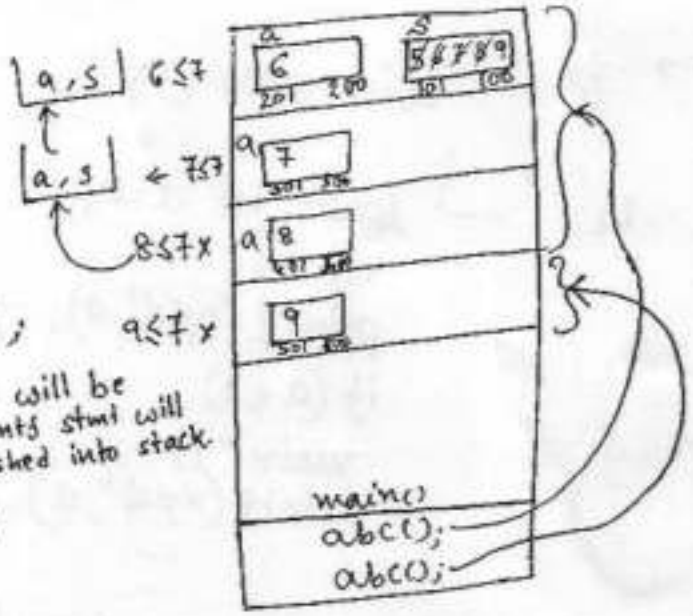
for $n = n$ $f(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}$ → for complete binary tree.

$= \frac{1(2^n - 1)}{2 - 1} = (2^n - 1) = O(2^n)$.

i.e. for large value of n, the recursive ~~tree~~ tree will become complete binary ~~tree~~ tree. Then it will give as $O(2^n)$.

```
void abc ( )
{
    auto int a;
    static int S = 5;
    a = ++S;
    printf("%d %d", a, S);
    if (a <= 7) // when this cond. will be satisfied, the printf stmt will be pushed into stack
        abc ();
    printf("%d %d", a, S);
}
void main ()
{
    abc ();
    abc ();
}
```

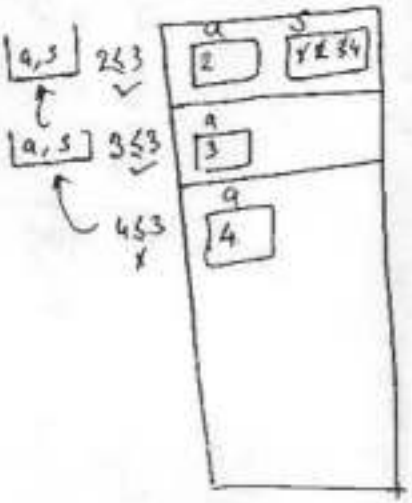
6,6 (1) 7,7 (2) 8,8 (3) 9,9 (4) 8,8 (5) 7,8 (6) 6,8 (7) 9,9 (8)



∴ 8 time printf stmt is executing.

```
void abc()
{
  static int s=1;
  auto int a;
  a = ++s;
  printf("%d %d", a, s);
  if (a<=3)
    abc();
  printf("%d %d", a, s);
}
void main()
{ abc();
}
```

2,2 ①
3,3 ②
4,4 ③
4,4 ④
3,4 ⑤
2,4 ⑥

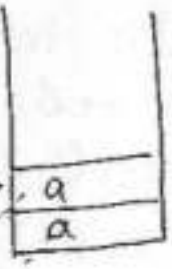



```
void MMOUM ( )
{ static int a=1;
  ++a;
  printf("%d", a);
  if (a<=3)
    main();
  printf("%d", a);
}
```

L_1:

2, 3, 4
① ② ③

4, 4, 4
④ ⑤ ⑥



∴ 2, 3, 4, 4, 4, 4. (Ans).

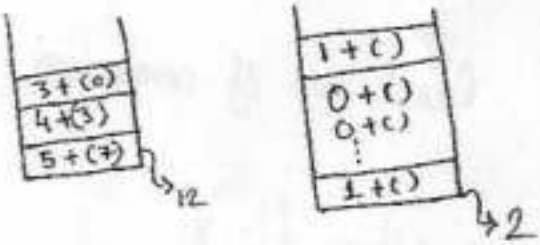
If L_1 replaced by "auto int a=1"; then it will result stack overflow error, because every time a will be initialized by 1 and 1 always ≤3.

[0,1 / 1,2 / 2,3 / 3,4 / 4,5 / 3,4 / 2,4 / 3,4 / ⊃∧/9∧!
1 2 3 4 9 5 6 7 8 10

```
void abc()
{
 auto int a;
 static int s;
 a = s++;
 printf("%d %d", a, s);
 if (a<2)
  abc();
 printf("%d %d", a, s);
}
void main()
{
  abc();
  abc();
}
```

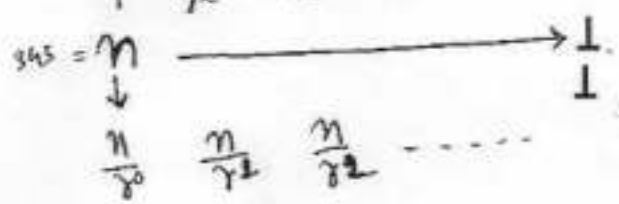
How many time printf will execute?

```
int f(int n, int r)
{ if (n>0)
   return ((n%r)+f(n/r, r));
  else
   return 0;
}
```



what is return value of f(345,10)? →12

what is return value of f(513, 2)? →2



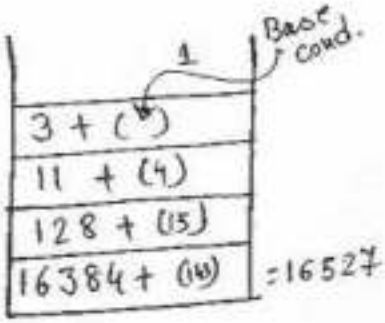
$\frac{n}{r^{k-1}} = 1 \Rightarrow k = \log_r n + 1 = O(\log_r n)$.

```
int DoSomething(int n)
{
  if(n<=2)
    return 1;
  else
    return ( DoSomething(floor( sqrt(n)) +n);
}
```

12

what is the return value of DoSomething(16384)?

$\sqrt{16384} = 128$



$n \longrightarrow 2$ (1st, 2nd, 3rd, ... k^{th})

$n \quad \sqrt{n} \quad \sqrt{\sqrt{n}} \quad \ldots$ at k^{th} iteration.

$n \quad n^{1/2} \quad n^{1/2^2} \quad \ldots \quad n^{1/2^{k-1}}$

$n^{\frac{1}{2^{k-1}}} = 2$

$\Rightarrow \frac{1}{2^{k-1}} = \log_n 2$.

$\Rightarrow 2^{k-1} = \log_2 n \Rightarrow k-1 = \log_2(\log n)$

$\Rightarrow 2^k = 2\log_2 n = 2\log n$

$k = \quad \therefore O(\log_2 \log_2 n)$

Exact no. of comparison $= \lceil \log_2(\log_2 n) \rceil$

* Why floor?

$2\log_2 \log_2 16384 + 1$

$= \log_2 \log_2 2^{14} + 1$

$= \log_2 14 \cdot \log_2 2 + 1$

$= \log_2 14 + 1$

$= \log_2 2^{3.-} + 1$

$= (3.-)\log_2 2 + 1$

$= 3.- + 1$ (take floor fun)

$= \lfloor 3.- \rfloor + 1 = 4 + 1$

$= 5$

n	fun Call
16384	5

Register :

* Not a comman, it is a request

```
void main()
{
  register int i=10;
  printf("%d", i); // i will be accessed faster.
}
```

→ Syntax to declare register variable is:

register datatype variable_name;

Ex: register int i;

→ The keyword register is not a command, it is just request to the compiler that hold the variable i into the register insted of RAM. If there are free registers are available, then i stored into the regiate Otherwise i stored into the RAM only.

→ There are limited no. of registers.

→ We can not access address of register variable. So poin related concept is not applicable for register variable

→ register variable have body scope and function lifetim

```
void main()
{
  register int i;
  printf("Enter value of i");
  scanf("%d", &i); → Error: Address of i not availe
}
```

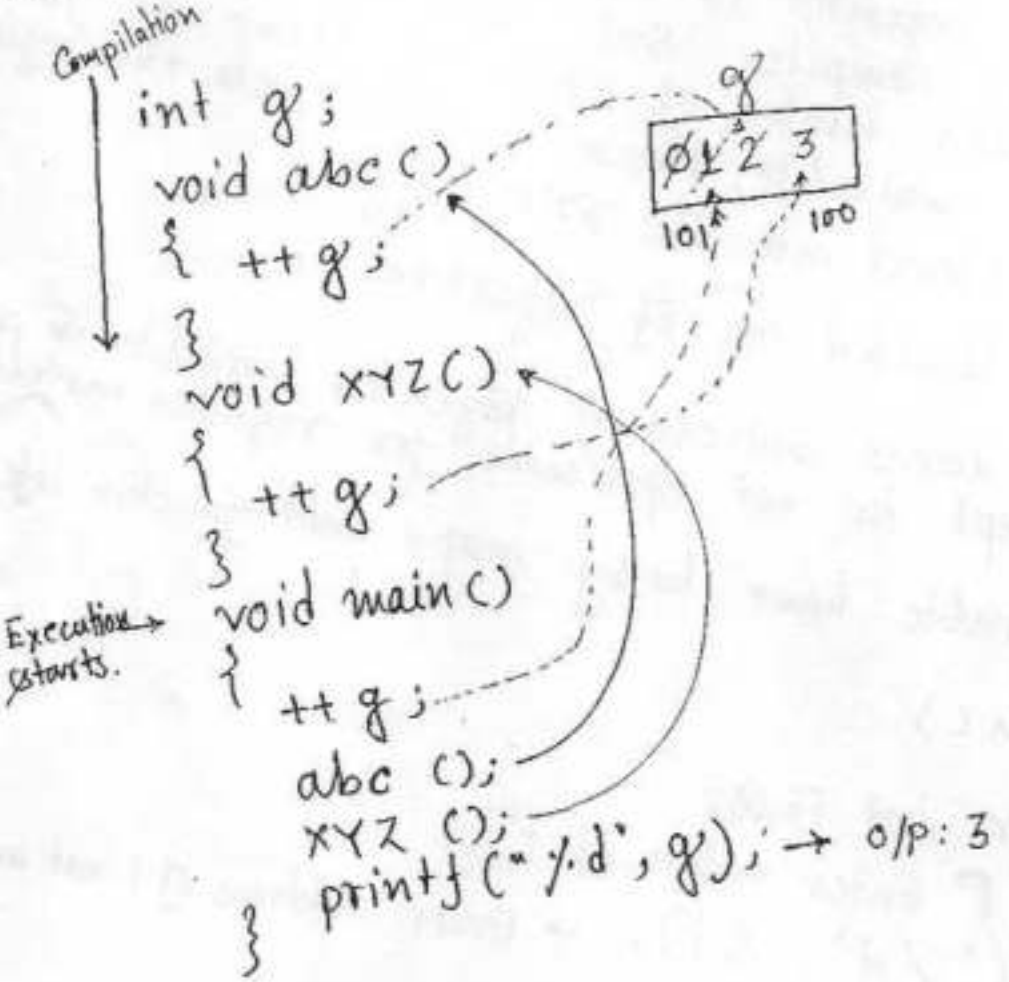
· Global Variables:

- A variable which is declared outside of all function blocks is called global variable.

If global variable is not initialized, by default it contains value 0.

- It has program lifetime.

· Global variables are accessable in all fun blocks provided that there is no local variable with the same name of local variable.



```
void xyz()
{ extern int g;
  ++g;
}
int g; // Global var.
void main()
{
  ++g;
  xyz();
  printf("%d", g);
}
```

"Undefined variable 'g'" will be given if extern is not used.

⇒ Due to top-down approach of Compilation process this program generates compile time error "Undefined variable g" in function xyz(); To overcome this we have to specify the prototype of global variable using the keyword "extern".

O/P: 2 (after including extern).

```
extern int g;
void xyz()
{ // extern int g;
  ++g;
}
void main()
{ // extern int g;
  ++g;
  xyz();
  printf("%d", g)
}
int g; // Global variable.
```

⇒ Here if we declare g as global variable at the last then 'A' will give error for xyz() as well as main();

```
*extern int g;
 void xyz()
 { ++g;
 }
 void main()
 { ++g;
   xyz();
   printf("%d", g);
 }
```

Explanation: This program will generate runtime error (linking error) because there is no physical memory location for the variable g.

NOTE: • If the prototype declaration not available it generate compile time error, if definition of variable not available it generate linking error.

• No physical memory allocated to prototype declaration and a

→ If the initialization is done at the time of prototype declaration, then physical memory will be allocated.
In the above program if we replace line * with extern int g=0, then the o/p is 2.

```
extern int g=0.
void xyz()
{
 =
}                   // Error: Multiple declaration of g.
void main()
{
 =
}
int %-..>
```

a [=0]   b [=0]   c [=0]

```
→ int a,b,c=0;
  void ptr fun(); // If we remove it, will give compile time error.
  main()
  { static int a=1; // Line 1        [1 2]
    ptrfun();
    a+=1;                            i) o/p: 4,2  6,2  2,0
    ptrfun();                        ii) o/p: 4,2, 4,2, 2,0
    printf("%d%d", a,b);
  }
  void ptrfun()
  { static int a=2; // Line 2.
    int b=1;                         a=a+ ++b = 2+2 = 4
    a+=++b;
    printf("%d %d", a,b);
  }
```

→ what is the o/p? if line 1 replaced with auto int a=1,
∧ what is the o/p if line 2 replaced with register int a=2;

Pointers:

- Basics
- Pointer to Array
- Pointer to function.
- Pointer to Pointer
- Array of Pointers
- Pointer to String
- Pointer to Structure.

→ Memory is collection of segments. (Ex: 16 segments). and every segment is collection of byte (Ex: 1 segment = 64 kB) Every byte is recognized by a unique location no. called its address. To store address of a byte we need pointer variable.

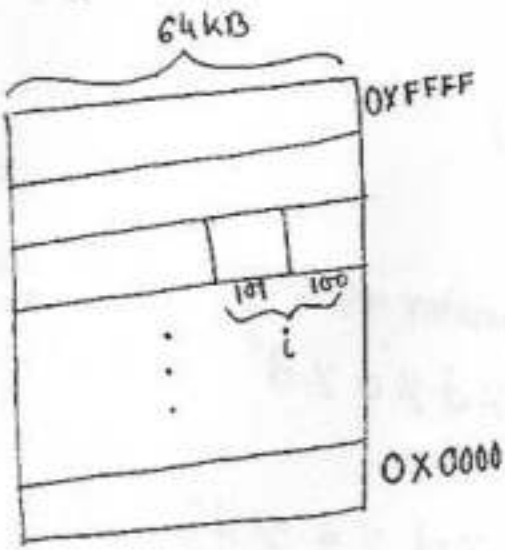
→ Pointer variable uses two operators.

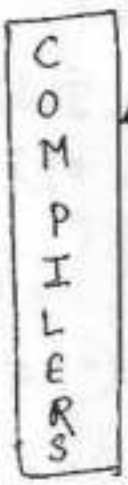
1. & - address operator (or) reference operator.
2. * - value operator (or) dereference operator. (or) indirection operator.

Terbo C 3.0
↓ operates by
8086
↓ uses
DOS
↓
16 bit

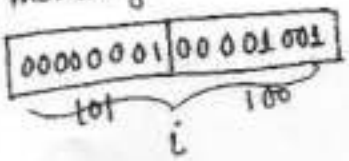
16 x 64 kB = 1 MB

∴ TC require min 1 MB memory.





int i = 265



```
void main()
{
  int i=20;
  int *p; // p is a pointer variable which accept an integer value.
  p = &i;
  printf("%d%d", i, *p);
  printf("%U %P", &i, &i);
  printf("%d", p);
  printf("%U", p);
  printf("%U", &p);
}
```

→ 200 → Hex form of 200.

→ -504 (unknown value)

→ 200

→ 300

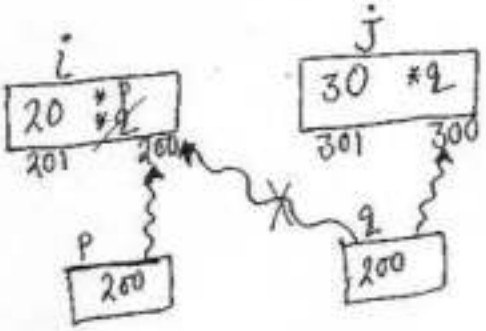
→ %P → prints address in hexadecimal form.

→ %U → prints address in decimal form.

→ Pointer Assignment :

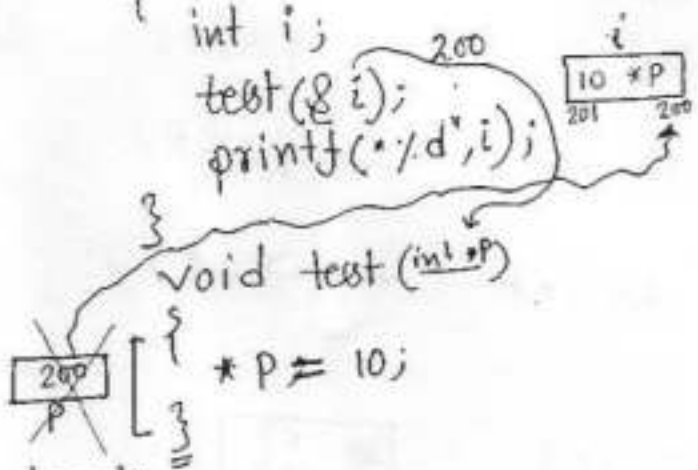
```
void main()
{ int i=20, j=30;
  int *p, *q;
  p = &i;
  q = p; // pointer assignment
  printf("%d %d %d %d", i, j, *p, *q);
  q = &j;
  printf("%d %d %d %d", i, j, *p, *q);
}
```

(20 20 20 20) (20 30 20 30)



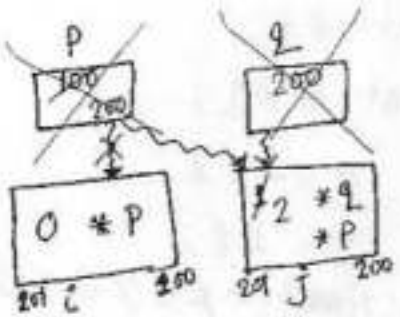
* Passing Address of a variable to a function.

```
void test(int*); // prototype.
void main()
{
  int i;
  test(&i);
  printf("%d", i);
}
void test(int *p)
{
  *p = 10;
}
```

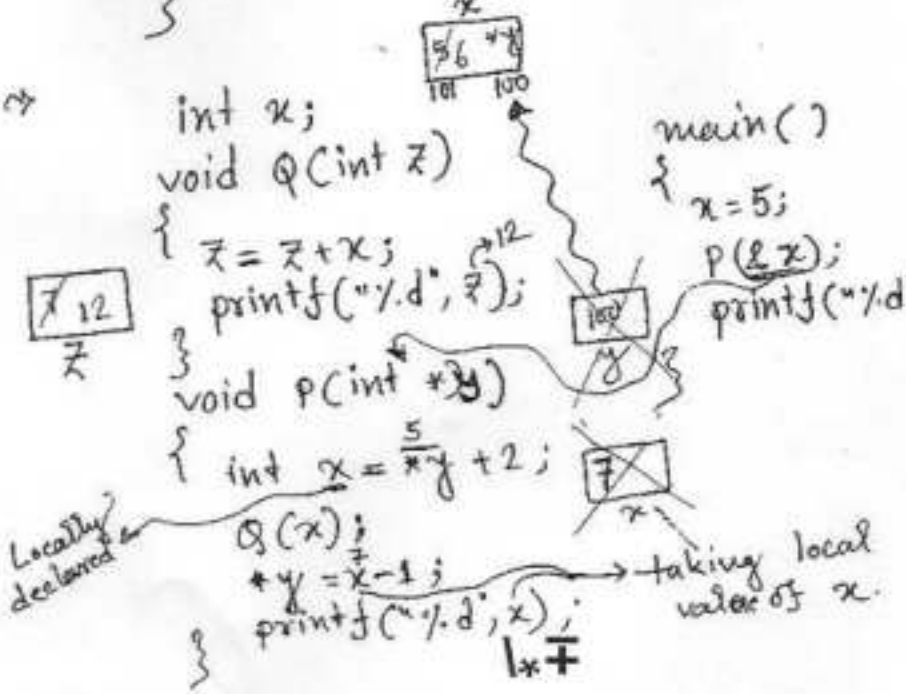


↳ auto, so Life time is function.

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




```
int x;
void Q(int z)
{
  z = z + x;
  printf("%d", z);
}
void P(int *y)
{
  int x = *y + 2;
  Q(x);
  *y = x - 1;
  printf("%d", x);
}
main()
{
  x = 5;
  P(&x);
  printf("%d", x);
}
```

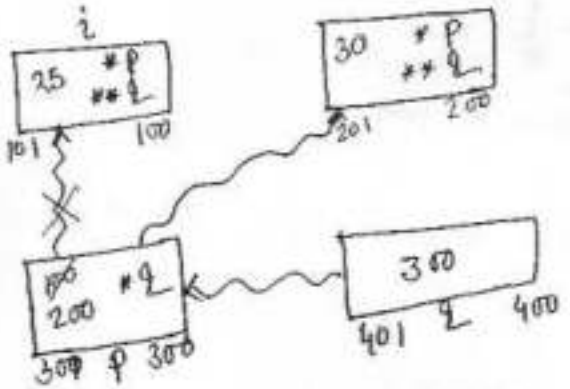


↳ 6 (it is accessing global variable x because within main() no declaration of x, only value assignment.

* Pointer to pointer:

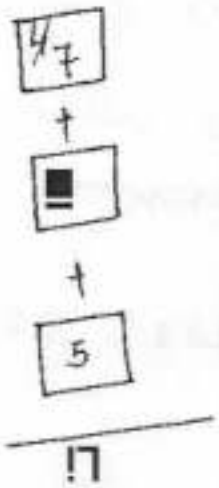
NOTE:

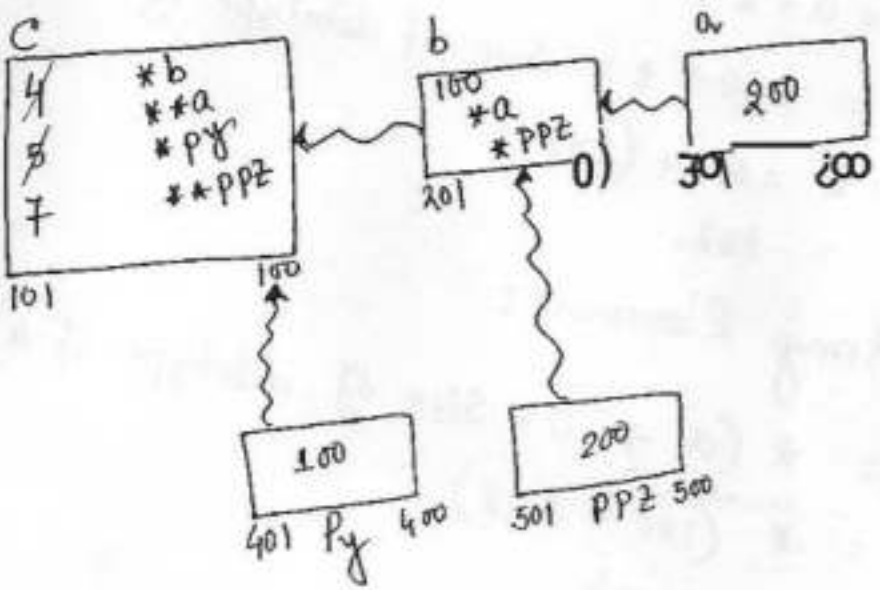
	datatype	
int i;	int	
int *p;	int *	→ p = &i;
int **q;	int **	→ q = &p;
int ***r;	int ***	→ r = &q;




```
void main()
{
    int i=25, j=30;
    int *p;
    int **q;
    p = &i;
    q = &p;
    printf("%d %d %d %d", i, j, *p, **q);
    *q = &j;
    printf("%d %d %d %d", i, j, *p, **q);
}
```

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





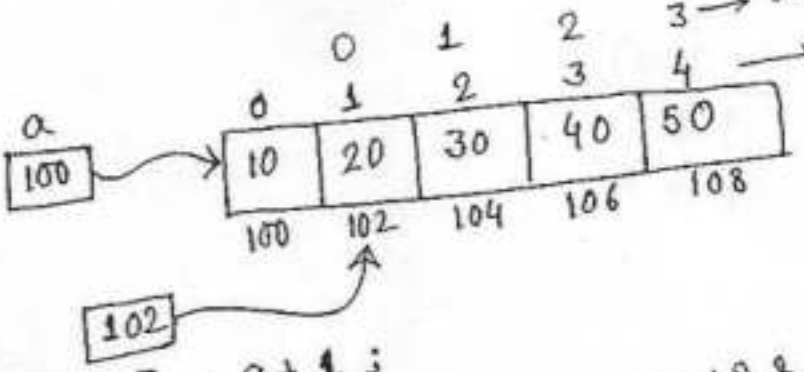
- Base Address of array is accessible using

  a; or &a[0];

- Array Name is called constant pointer. i.e. we can not increment or decrement the ∩ name.

  a = a+1; // Error: L-value required.

```
int a[5] = {10, 20, 30, 40, 50};
int *p; // → Incrementation of a is not possible. That's why p is necessary
```



P = a+1;

= 100 + 1 x Size of datatype of a

= 100 + (1x2)

= 102

Accessing Array Elements:

a[0] = *(a + 0 x Size of datatype of a)

= *(100 + 0x2)

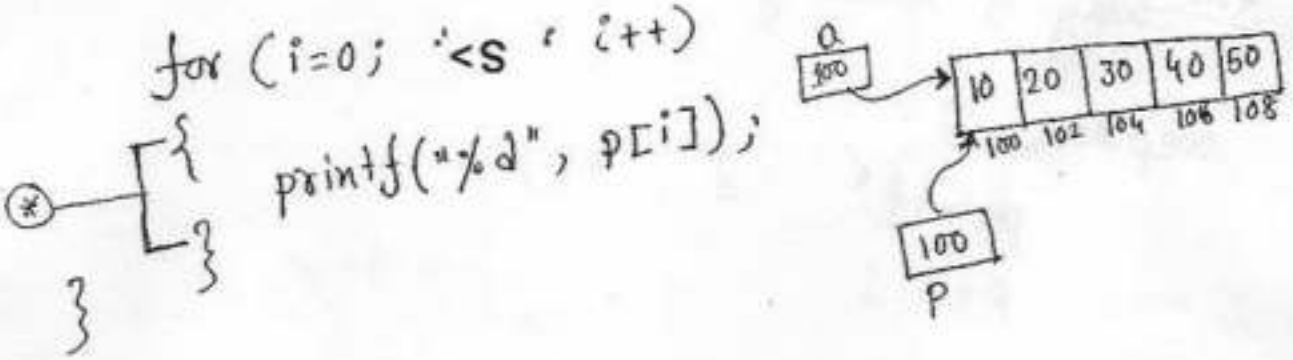
= *(100)

= 10

a[3] = *(a + 3 x 2)
= * (100 + 6)
= * (106)
= 40

a[i] = * (a + i x Size of datatype of a)

```
void main()
{
    int i;
    int a[5] = {10, 20, 30, 40, 50};
    int *p;
    p = &a[0]; // p = a : Pointer Assignment.
    for (i=0; i<5; i++)
    {
        printf("%d", p[i]);
    }
}
```



i	p[i]
0	p[0] = * (p + 0 x Size of datatype) = * (100 + 0 x 2) = * (100) = 10
1	20
:	:
4	50

2nd way of Accessing Array Element Using pointer.

* Replace Eqn ⊛ with the statement

```
printf ("%d", *(p+i));
```

i	*p[i] *(p+i)
0	= *(100+0x2) = *(100) = 10
1	= *(100+1x2) = *(102) = 20
⋮	⋮
4	= 50

3rd Way of Accessing Array Elements Using Pointer:

* Replace ⊛

```
:
printf("%d", *p)
p++;
```

i		p++
0	10	102
1	20	104
2	30	106
3	40	108
4	50	110

p = p + 1
= 100 + 1
↑ Address scalar
= 100 + 1X 2
= 102

here in for loop we should not use ('<=5'), bez the equal (=) sign will access 110 memory location which will provide unexpected value according to compiler behaviour.

→ Passing Address of Array to a function :

```
void display (int *, int);
void main()
{
    int a[5] = {10, 20, 30, 40, 50};
    display (a, 5);
    display (a+2, 5);
    printf(" %d", sizeof(a));
}
void display (int *p, int n)
{ int i;
  for (i=0; i<n; i++)
     printf (" %d", p[i]);
}
```

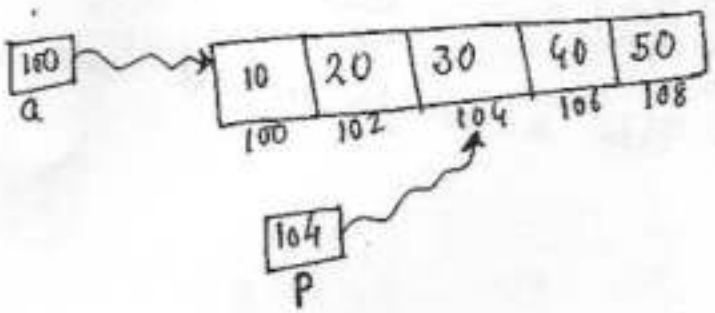
(100, 5)

(100+2*2, 5) = (104, 5)

O/P: 10 20 10 40 30

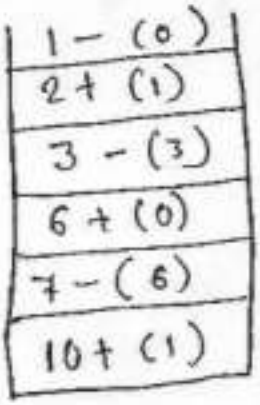
30 40 50 0 0

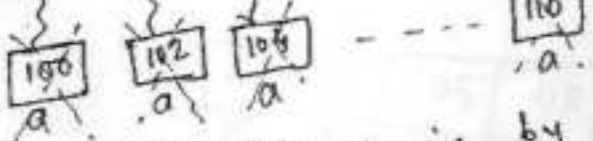
3 types of pointer:
→ Near Pointer
→ far Pointer
→ Huge Pointer



```
#d f(int *, int);
void main()
{
    int a[6] = {10, 7, 6, 3, 2, 1};   // a → name of array
    printf("%d", f(a, 6));
}
int f(int *a, int n)                  // a → pointer, scope is only within f();
{
    if (n<=0)
        return 0;
    if (*a % 2 == 0)
        return *a + f(a+1, n-1);
    else
        return *a - f(a+1, n-1);
}
```

0	1	2	3	4	5
*a	*a	*a	*a	*a	*a
10	7	6	3	2	1
100	102	104	106	108	110





↳ Here a is by default 'auto' variable.

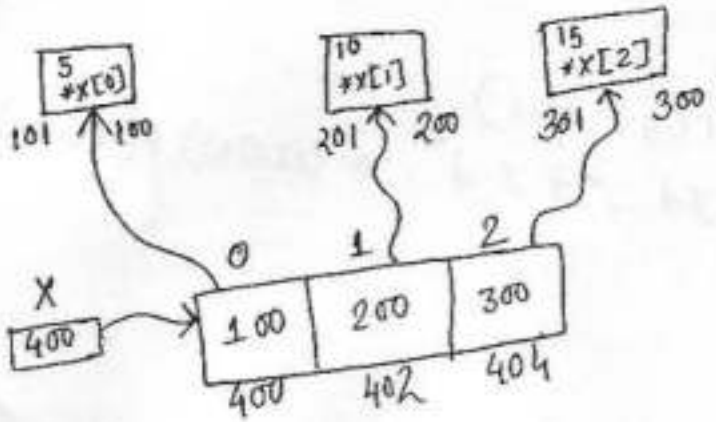
⇒ Array of Pointer:

NOTE:

float x[10]; // x is array of 10 float values and its size is 40 Byte.

float *x[10]; // x is array of 10 pointers and each pointer is waiting for 0, dd of float variable. Since the size of pointer 2-bytes, so size of array is 20 bytes.

```
void main()
{ int i;
  int a=5, b=10, c=15;
  int *x[3];
  x[0] = &a;
  x[1] = &b;
  x[2] = &c;
  for(i=0; i<3; i++)
    printf("%d", *x[i]);
}
```



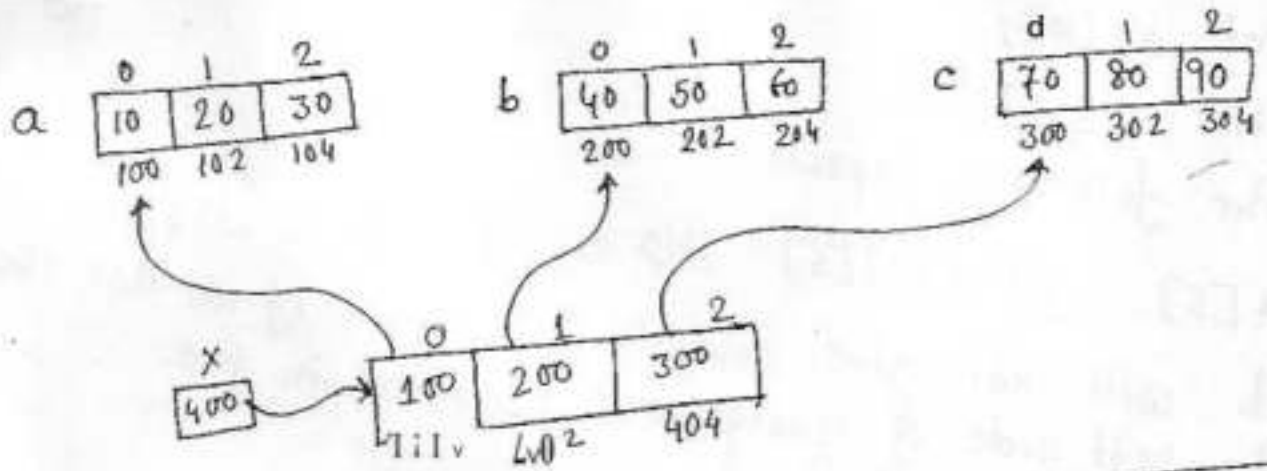
> According to compiler execution:

i	* x[i]
0	= * x[0] = * *(x + 0 x Size of Pointer) = * * (400 + 0 x 2) = * * (400) = * (100) = 5
1	= 10
2	= 15

↳ default value of pointer.

```
int *P;
printf("%d", Sizeof(P));
printf("%d", Sizeof(*P));
```
o/p: 2, 4

```
void main()
{
  int a[3] = {10, 20, 30};
  int b[3] = {40, 50, 60};
  int c[3] = {70, 80, 90};
  int i;
  int *x[3];
  x[0] = a; // pointer assignment.
  x[1] = b;
  x[2] = c;
  for(i=0; i<3; i++)
    printf("%d %d %d", *(x[0]+i), *(x[1]+i), *(x[2]+i));
}
```



i	*(X[0] + 0	*(X[i] + i)	*(X ∨ ∧0
0	= *(X[0] + 0 x size of data) = *(100 + 0 x 2) = *(100) = 10	= *(X[i] + 0 x Size of data) ∴ (300 + 0 x 2) = *(200) = 210	=*(X[i] + 0x Size of data) = *(300 + 0 x 2) = *(300) - 10
1	= 20	= 50	= 90
2	= 30	= 60	

According to previous program.

10 20 30 40 50 60

a[0] a[1] a[2] b[0] b[1] b[2]

X[0][0] X[0][1] X[1][0] X[1][1] X[1][2]

If X[2][1] = 35 then,
c[1] is updated by 35.

int *A[10];

int B[10][10];

Of the following expression

i) A[5] ii) A[5][5] iii) B[5] iv) B

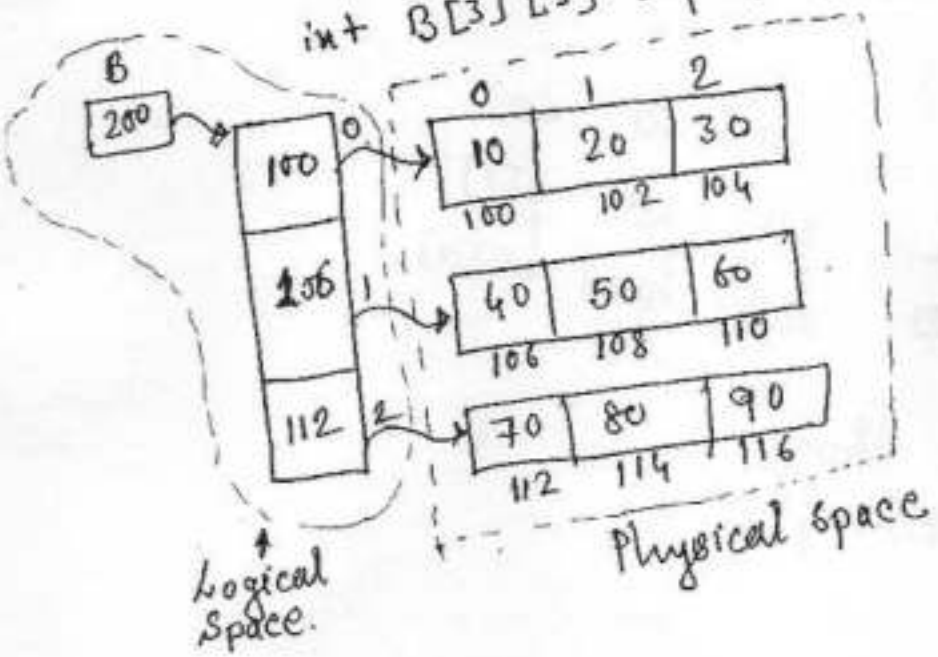
which will not give compile time error if we use them to the left side of assignment operator in the assignment statement?

- In int B[i][j],

B[i] represent ith row base address. we can not update to x as B[i] = x → returns compile time error.

- In int B[i][j],

the expression B[i][j] indicate ith row, jth column element. we update it to x using B[i][j] = x;



B[1][1] = *(6-A 1x Size of pointer)[1]

= *(200 + 1x2)[1]

= *(202)[1]

= 106[1]

= *(106 + 1x Size of data)

= *(106 + 1x2)

= *(108)

= 50

B[i][j] = *(*(B + ix Size of ptr) + j x Size of data)

```
printf("%U", B);      → 100
printf("%U", B[0]);   → 100
printf("%U", &B[0][0]); → 100
```

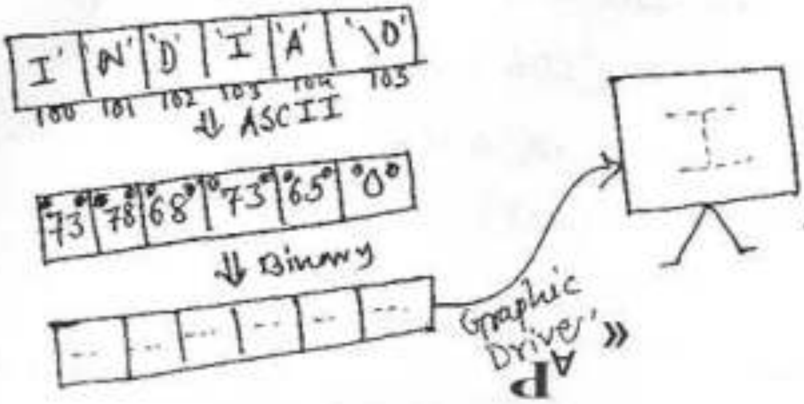
Pointer to String:

→ String Constant:

1) Group of characters enclosed in double code is string cor
2) Every character of the string constant is represented in the form of character constant.
3) All character constants are coverted into their corresponding ~~ase~~ ASCII values. Thesse ASCII value are stored into the memory sequentially in contiguous location.
4) Every string constant is terminated by \0 (null character).

5> Every string contains ... address.

Ex: "INDIA"



Initialization of String:

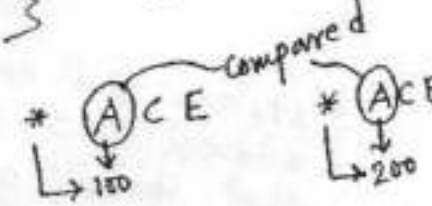
1. char = "INDIA";
2. char c[] = "INDIA";
3. char c[] = {'I', 'N', 'D', 'I', 'A', '\0'};
4. char c[] = {73, 78, 68, 73, 65, 0};
5. char *p = "INDIA";

NOTE:
char c[5] = "Hi"; 'H' 'I' '\0' '\0' '\0'
char c[2] = "Hi"; → Error: Too many initialization error.

```
void main()
{
  if("ACE" == "ACE")
  printf(" Strings are equal");
  else
  printf (" Strings are not equal");
}
```

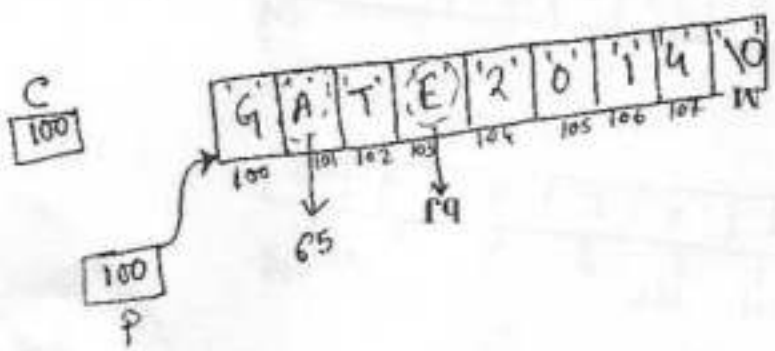
⇒ Base address of two strings get compared since no two strings have same base address, so else part will be executed.

```
void main()
{ if(*"ACE" = *"ACEHYD")
   printf (" Strings are equal);
  else
   printf (" strings are not equal");
}
```



⇒ ASCII value of the 1st character of the two string constant get compared. Since they are equal, if part will be executed.

```
char c[] = " GATE 2014";
char *p = c;
printf("%d", p+p[3]-p[1]);
```



(**) %s expecting address and it will scan untill ~~it~~ ~~not~~ null character is not encountered.

```
printf("%s", p + p[3] - p[1]);
```

↓

100 + (69×1) − (65×1)

= 104

↳ from address 104 the value will be printed untill null character is encountered.

↳ 2014 (o/p).

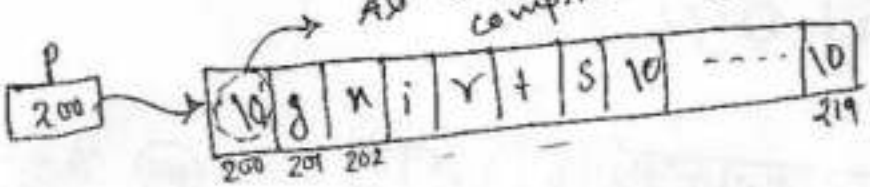
104 → 105 → 106 → 107

«.

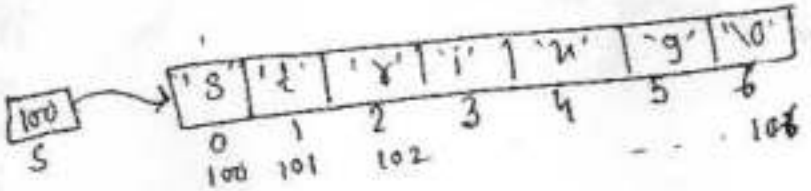
```
char p[20];
char *s = "string";
int length = strlen(s);
for (i=0; i<=length; i++)
    p[i] = s[length-i];
printf("%s", p);
```

'strlen' will expect address of s and from that it will count no. of non-null character. Here, length = 6.

After for loop no braces, so, this stmt will execute after completion of for loop.

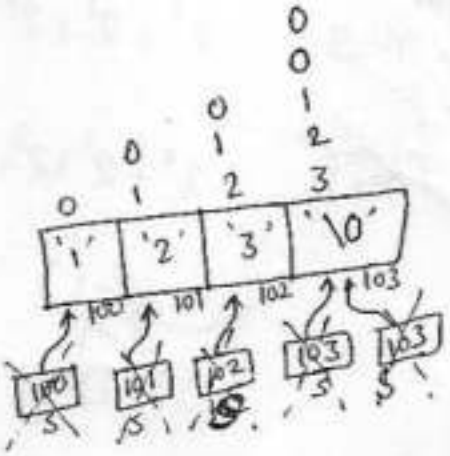


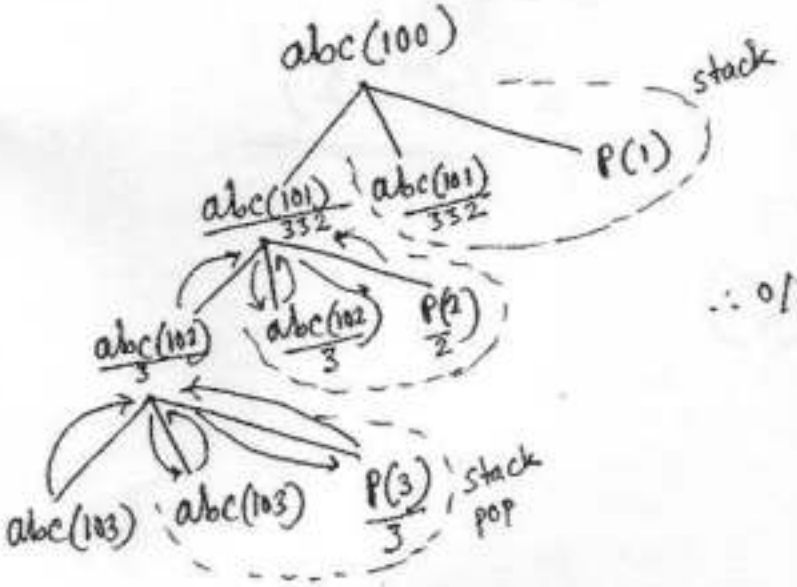
As at first null character is encountered, compiler will assume that this is the end of string. So no o/p will printed.



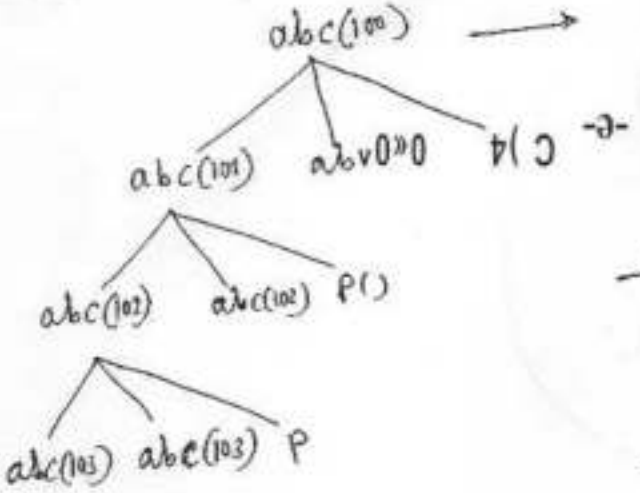
```
void abc (char *s)
{
  if (s[0] == '\0')
    return;
  else
    abc (s+1);
    abc (s+1);
    printf("%c", s[0]);
}
void main()
{
  abc ("123");
}
```

1) O/P?
2) If i/p string is of length "n" then
   a) How many character printed?
   b) How many function call?





∴ O/P: 332 332 1.



	printed	
	0 -	$1 = 2^0$
	$1 = 2^0$	$2 = 2^1$
→	$2 = 2^1$	$4 = 2^2$
→	$4 = 2^2$	$8 = 2^3$

Character printed

f- calls.

for $n=3$ $\quad 2^0 + 2^1 + 2^2$ $\qquad 2^0+2^1+2^2+2^3$

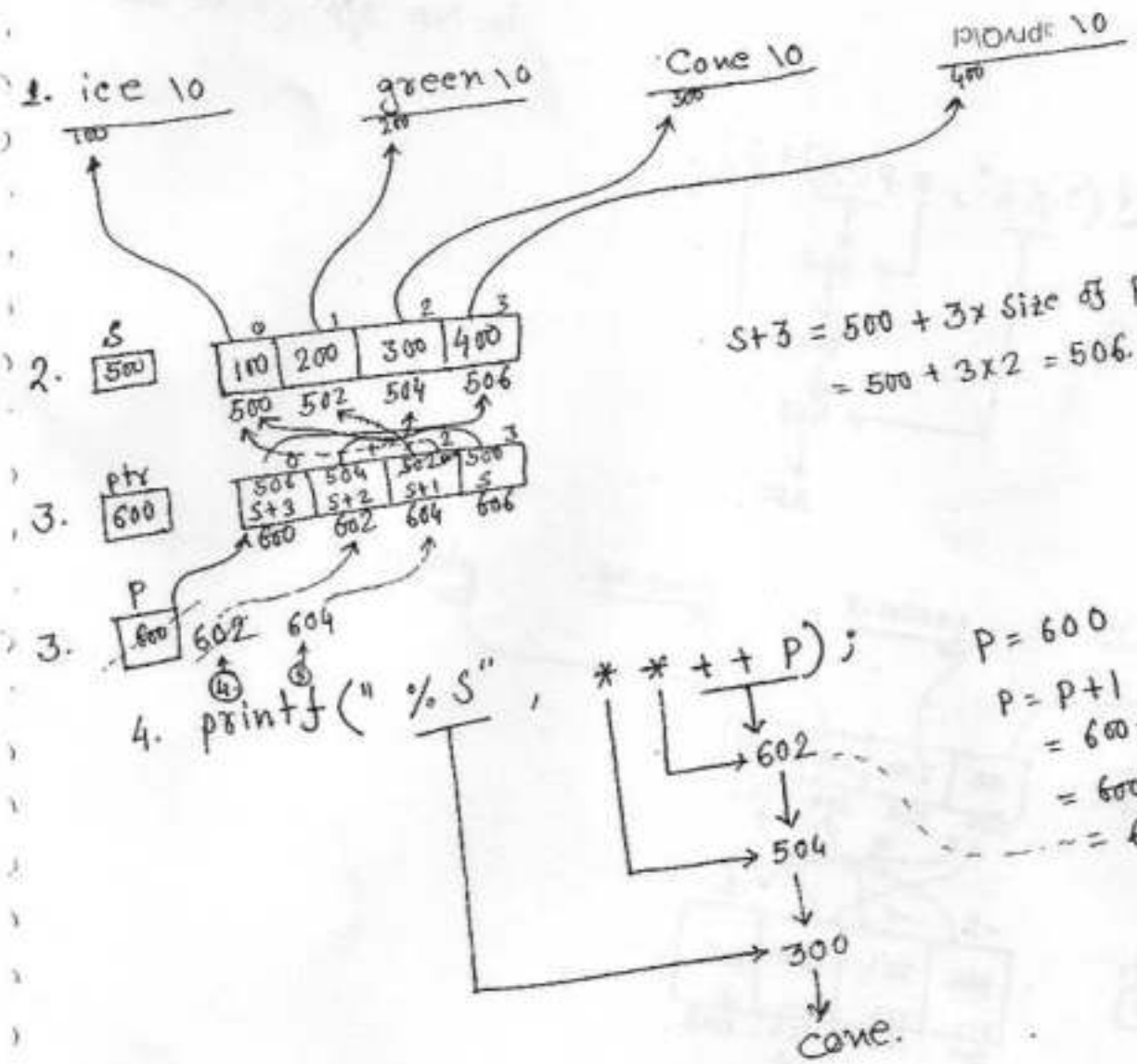
$n=n$ $\quad 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}$ $\qquad 2^0+2^1+2^2+\cdots+2^n$

$= \frac{*(r^n-1)}{2-1}$ $\qquad = \frac{1(2^{n+1}-1)}{2-1}$

" $(i<n-0$ $\qquad = (2^{n+1}-1)$

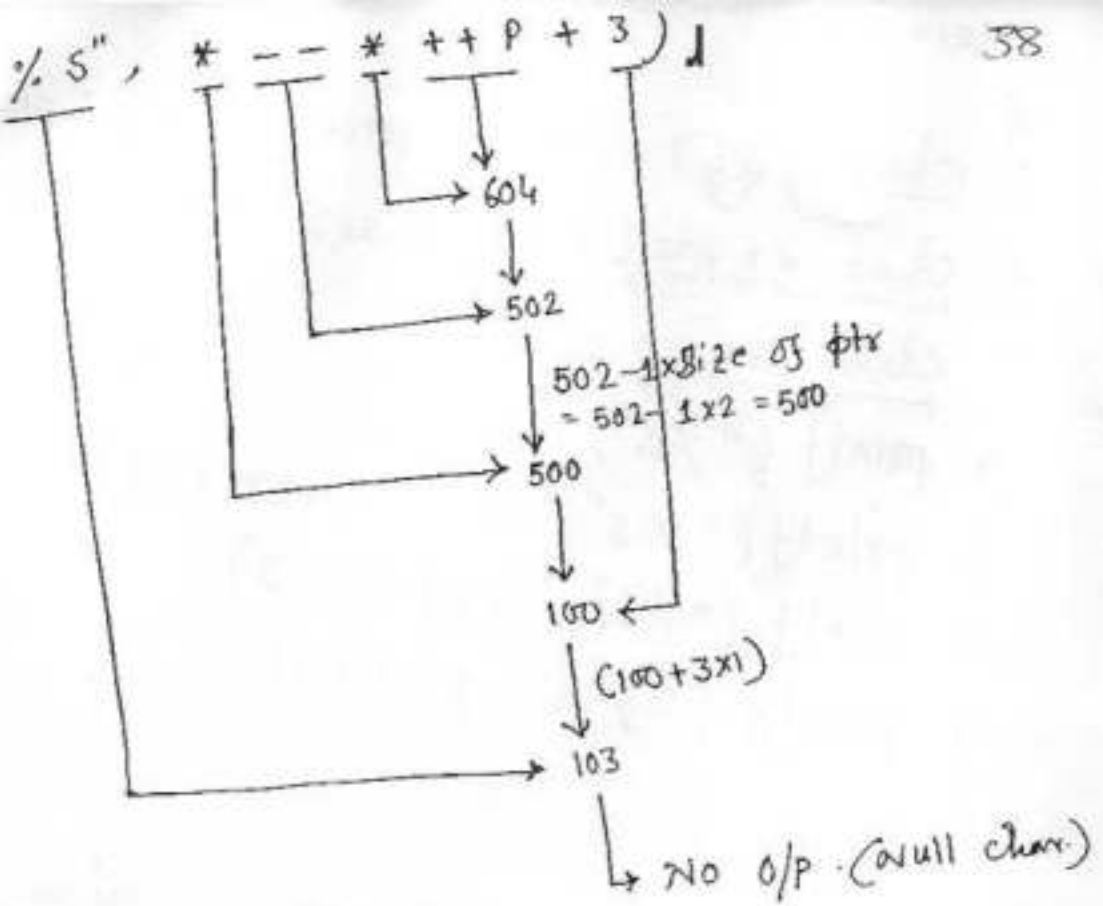
$O(2^{n+1}-1) = O(2^n)$.

```
main ()
{
1. char *s[] = {"ice", "green", "cone", "please"};
2. char **ptr[] = {s+3, s+2, s+1, s}; // ptr is an array of pointer type which is ready to accept 'character star' type of data.
3. char ***p = ptr;
4. printf ("%s", **++p);
5. printf ("%s", *--*++p+3);
6. printf ("%s", *p[-2]+3);
7. printf ("%s", p[-1][-1]+1);
}
```

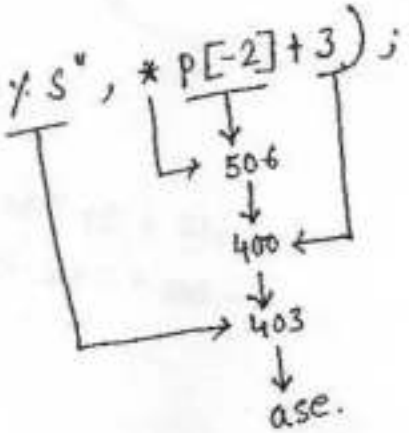


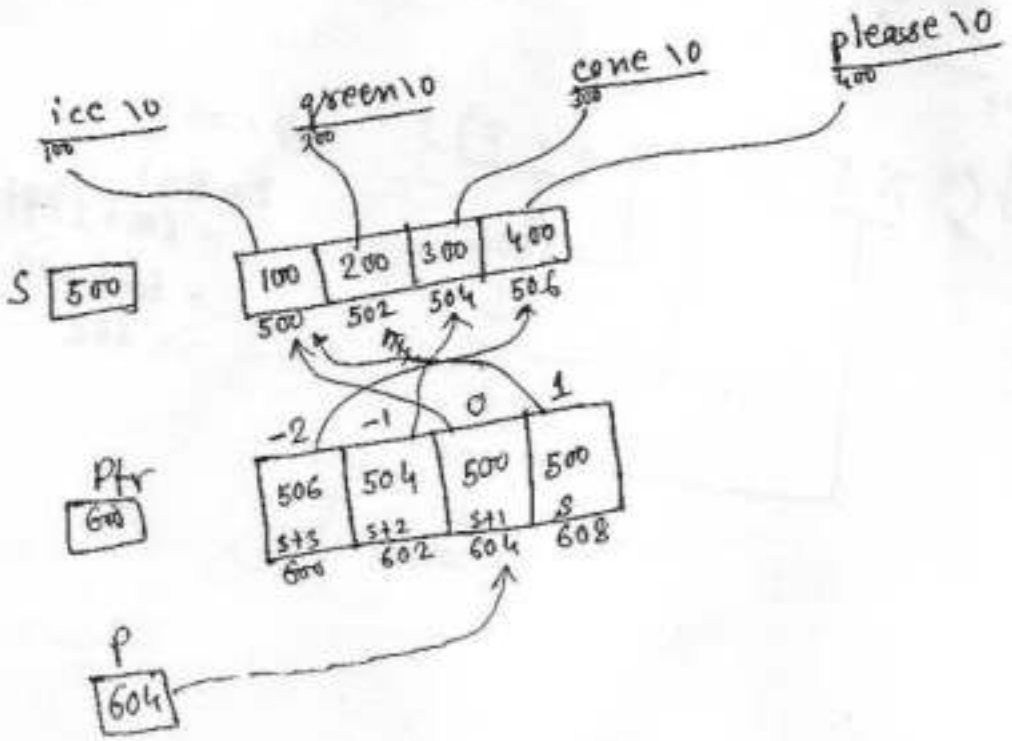
S+3 = 500 + 3x size of pointer
= 500 + 3x2 = 506.

5. printf("%s", *--*++p+3);



6- printf("%s", *p[-2]+3);





or

by subscript rule:

* p[-2] + 3

↓ ↓ *(p + (-2 * 2)) + 3

↓ ↓ *(604 - 4) + 3)

↓ ↓ *(600) + 3

↓ ↓

506

400 ← +3

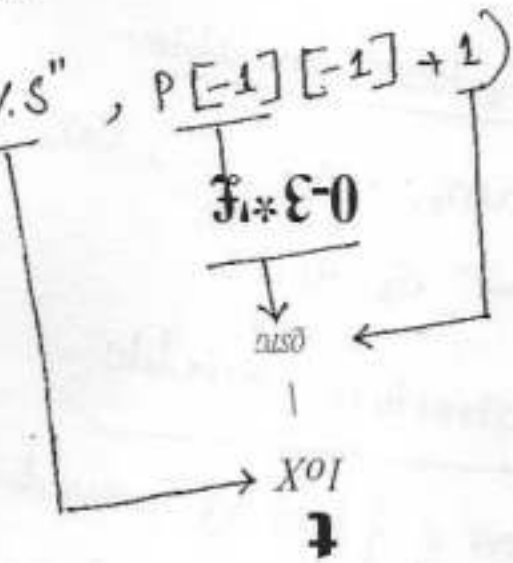
400 + 3 × 1

403

↓

ase.

5. printf("%s", P[-1][-1] + 1);



- Pointer to Structure :

Basic

- Structure Declaration

Syntax

```
Struct tagname
{
   member 1;
     :
   member n;
};
```

Ex:

```
Struct Student
{
   int dno;
   char s[5];
};
```

→ Declaration of structure variable.

Struct tagname $var_1, var_2, \ldots, var_n$;

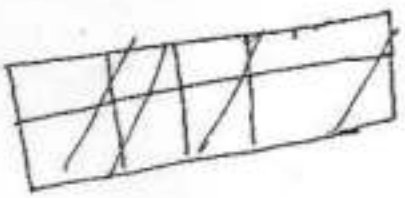
Ex: Struct student s_1, s_2 ;

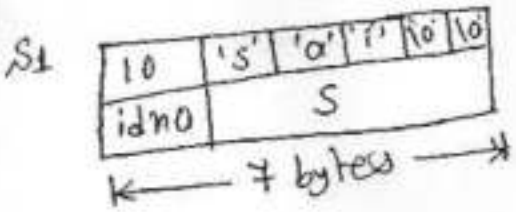
→ Initialization of structure variable

Struct tagname var = { list of members };

Ex: Struct student s1 = {"Sai", 10}; ✗ → wrong according to our declaration.

Struct student s1 = {10, "Sai"}; ✓





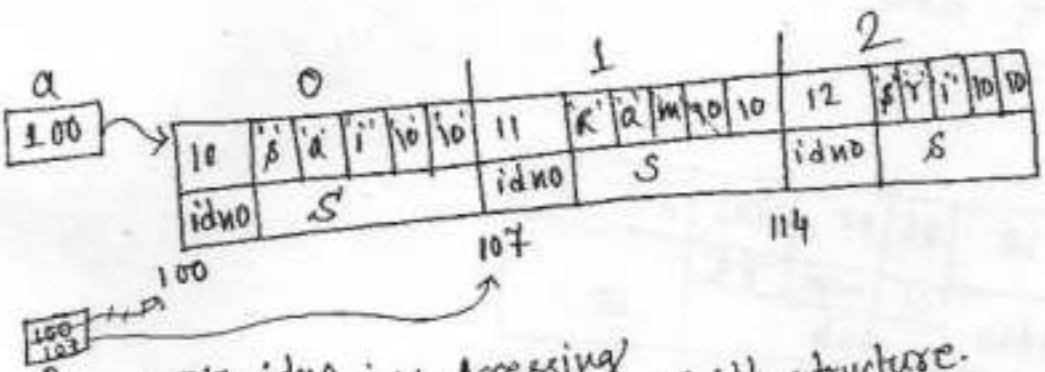
→ Another Way to Define Structure:

```
struct student
{
    int idno;
    char s[5];
} s1, s2 = {10, "sai"};
```

Syntax:

```
struct tagname array_name[size];
```

: `struct student a[3] = { 10, "sai", 11, "Ram", 12, "Sai" };`
(0, 1, 2)




```
a[i].idno;   // Accessing member of ith structure.
a[i].s;
```

`struct student *p = a;`

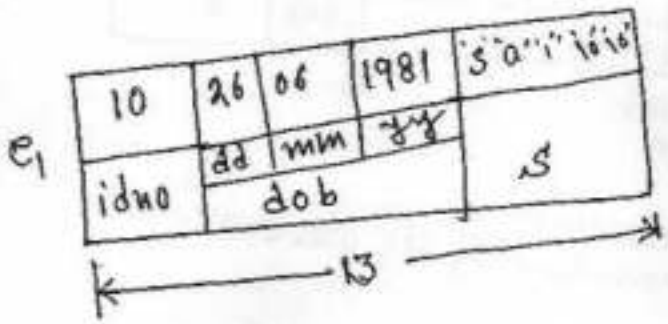
p = a + 1
= a + 1 x size of datatype
= 100 + 1 x 7 = 107.

→ The process of using one structure variable using another structure is called nested structure. 42

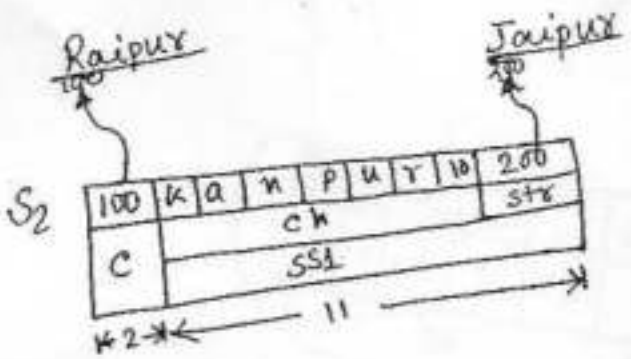
```
student Date
{
  int dd;
  int mm;
  int yy;
};
struct Emp
{
  int idno;
  struct Data dob;
  char s[5];
} e1 = {10, 26,06,1981, "sai"};
```

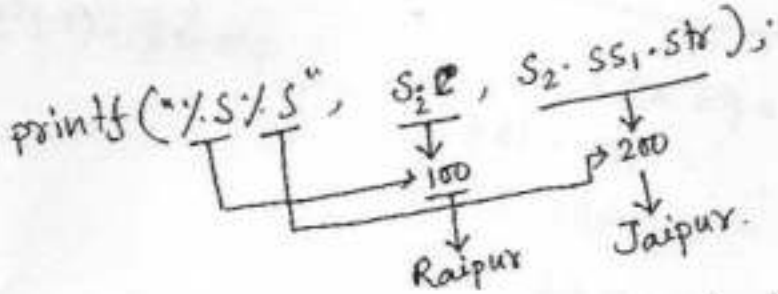
Access:

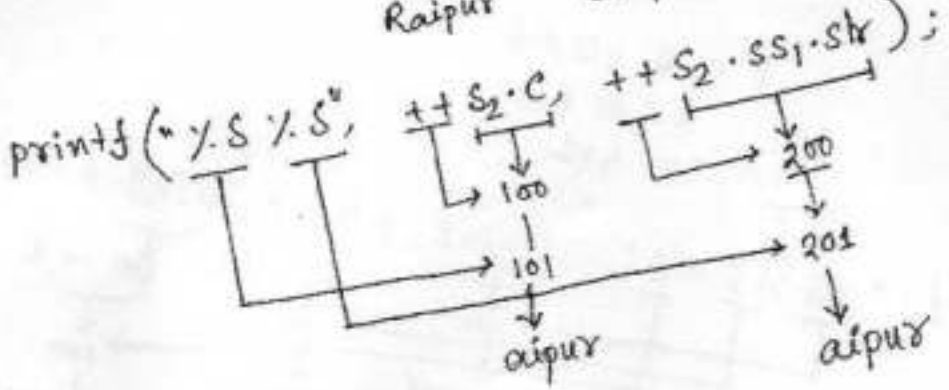
```
e1.idno;
e1.dob.dd;
e1.dod.mm;
e1.s;
```



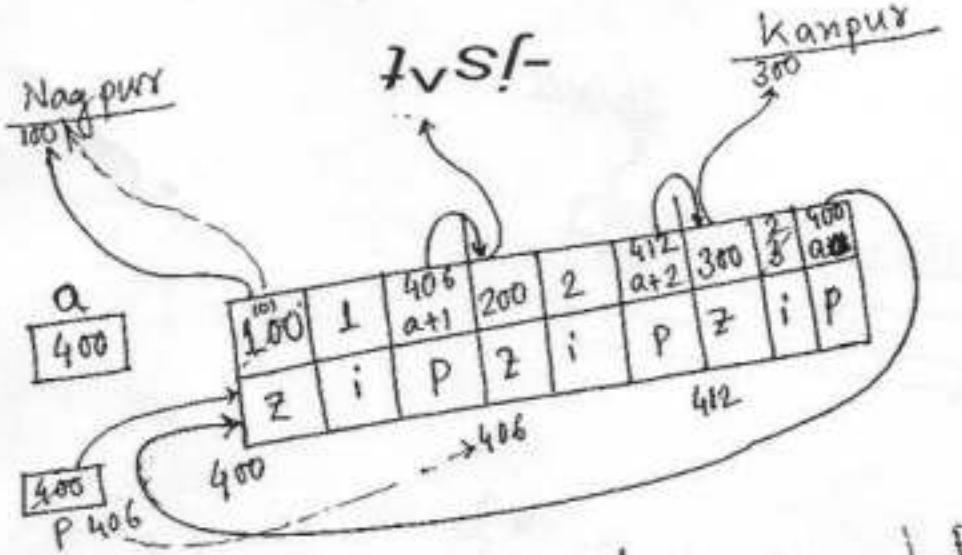
```
main()
{
  struct a;
  {
    char ch[7];  } → 9 byte.
    char *str;
  };
  struct b
  {
    char *c;  ---  } → 2 byte
    struct a ss1;  } → 9 byte  } → 11 byte.
  };
  Struct b s2 = {"Raipur", "Kanpur", "Jaipur"};
  printf("%s %s", s2.c, s2.ss1.str);
  printf("%s %s", ++s2.c, ++s2.ss1.str);
}
```







```
main()
{
  struct S1;
  {
    char *z;  → 2
    int i;    → 2
    struct S1 *P;  → 2/c byte
  };
  struct S1 a[] = {{"Nagpur", 1, a+1}, {"Raipur", 2, a+2},
                   {"Kanpur", 3, a**}};
                    0                  1                 2
  struct S1 *ptr = a;
  printf("%s %s %s", a[0]=z, ptr→z, a[2] P→z);
}
```



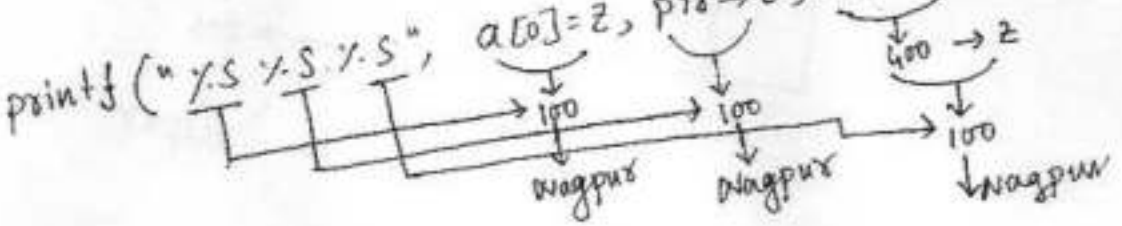
ptr→z = (*ptr).z

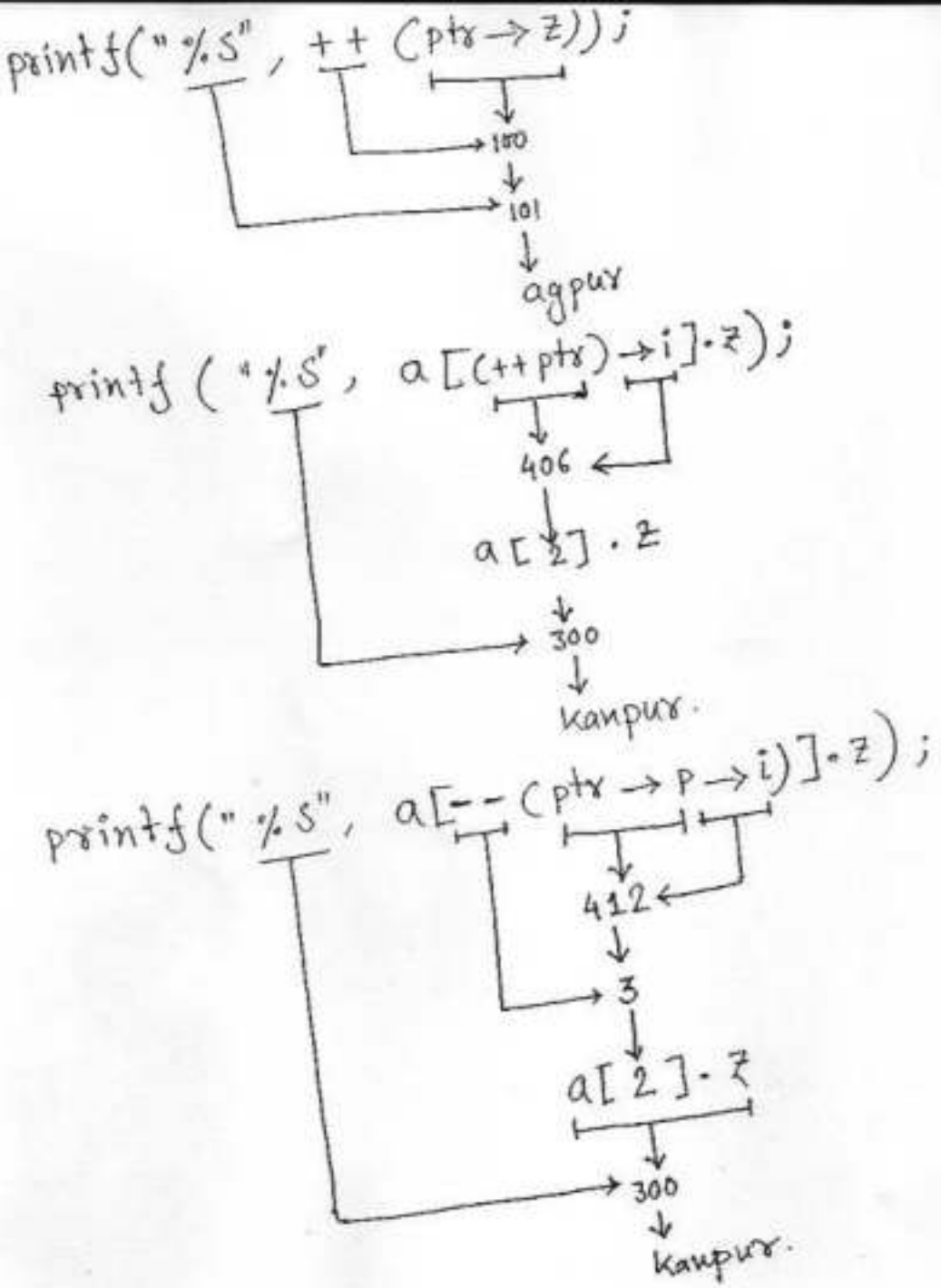
cap

struct S1 *p = a+1
= 400 + 1×6
= 406.

p = a + 2×6
= 400 + 12
= 412

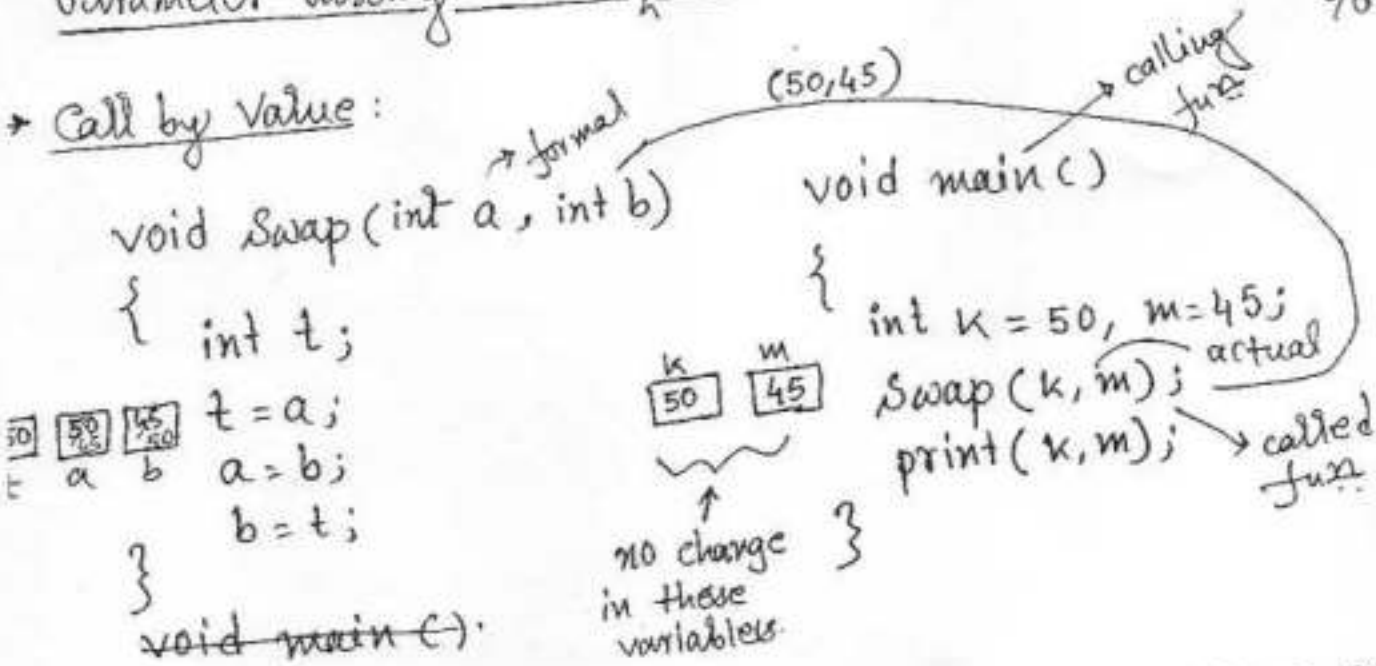
printf("%s %s %s", a[0]=z, ptr→z, a[2]·P→z);




printf("%s", ++(ptr→z));
100
101
agpur
printf("%s", a[(++ptr)→i].z);
406
a[2].z
300
kanpur.
printf("%s", a[--(ptr→p→i)].z);
412
3
a[2].z
300
kanpur.

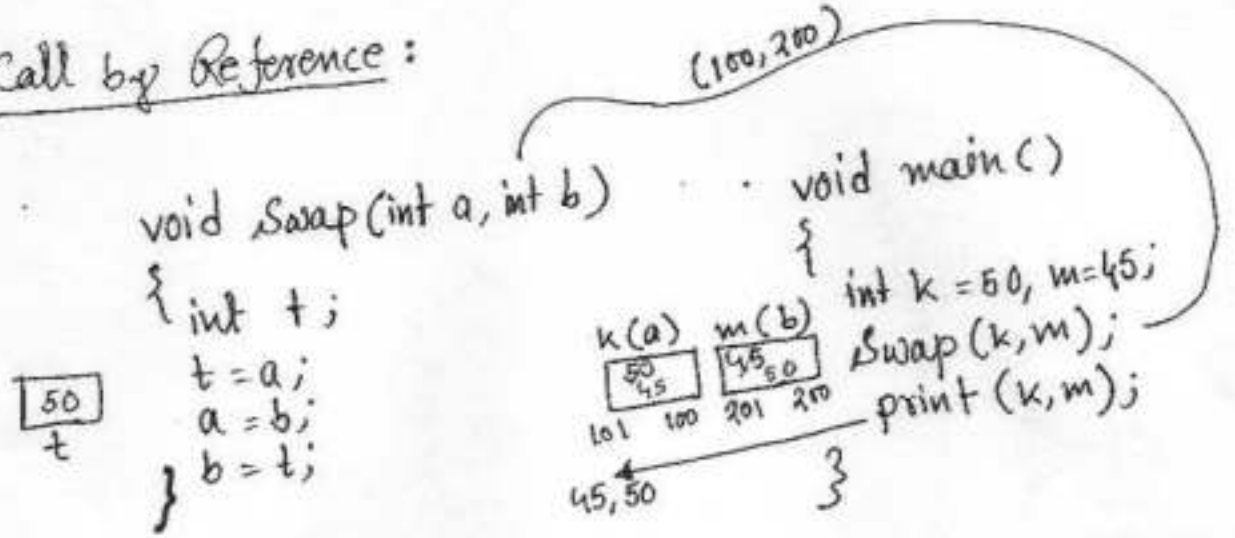
- Parameter Passing Techniques:

* Call by Value:



* Actual arguments carry their values and these values we copied into formal arguments.
* Formal arguments are modified by called function. These modifications are not effecting following function variables.
* Java, C, PASCAL support call by value mechnanism.

* Call by Reference:



* In call by Reference, actual argumen carried a and formal arguments are acting as alise to the corresponding actual arguments.
(another name)

* Whatever modification performed on formal arguments in the called fun will indirectly effecting on actual arguments.

** FORTRAN, C++, PASCAL support call by reference.

→ Call by Value Result (or) Copy Restore :

```
void Swap (int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
```

50 | 50/45 | 45/50

```
void main ()
{
    int k = 50, m = 45;
    Swap (k, m);
    print (k, m);
}
```

k: 50/45 m: 45/50 (50, 45)

45, 50 ←

↳ It is the combination of call by value and call by refere
↳ Actual arguments are carried a values and it is copies into formal arguments. Later these formal arguments are modified by call function. Upon completion of execution of called function formal argument values are restored into their corresponding actual arguments.
↳ FORTRAT, ORACLE, ALGOL ... supports call by Value Resul
↳ Is Call by Reference and Call by Value Result prints the same result always ?

```
void test (int a, int b)
{ a = a+1;
  b = b+10;
}
void main()
{ int k=2;
  test(k,k);
  print(k);
}
```

(100,100)

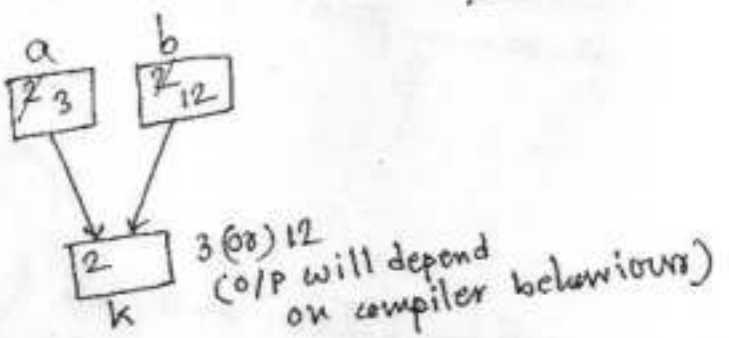
k (a, b)

2 3 13
101 100

→ 13 (call by Reference)

(call by value)

So, NO



* The results returned by the functions ~~and~~ under call by Reference and Call by Value Result —

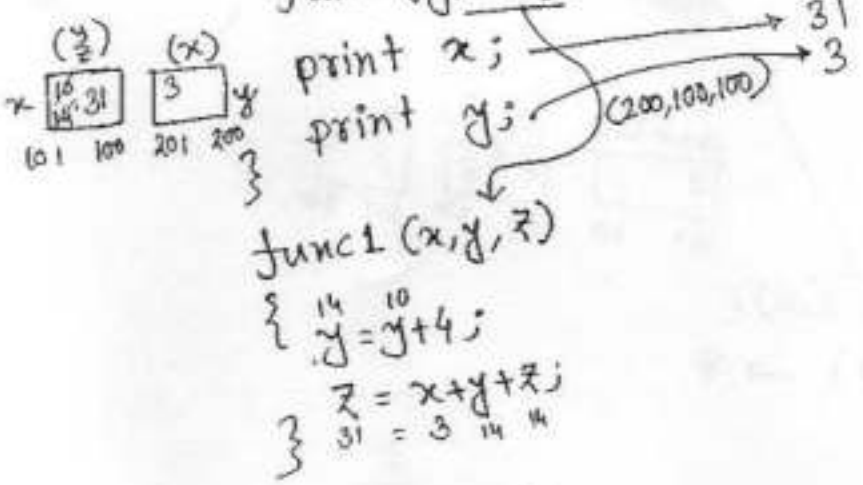
a) Do not differ b) Differ in the presence of loop

c) Differ in all cases ~~of~~ may differ in the presence of exception

* If multiple formal arguments are refering to the same variable of the calling function, then call by Reference and Call by Value Result results will differ.

Program 9 ( )

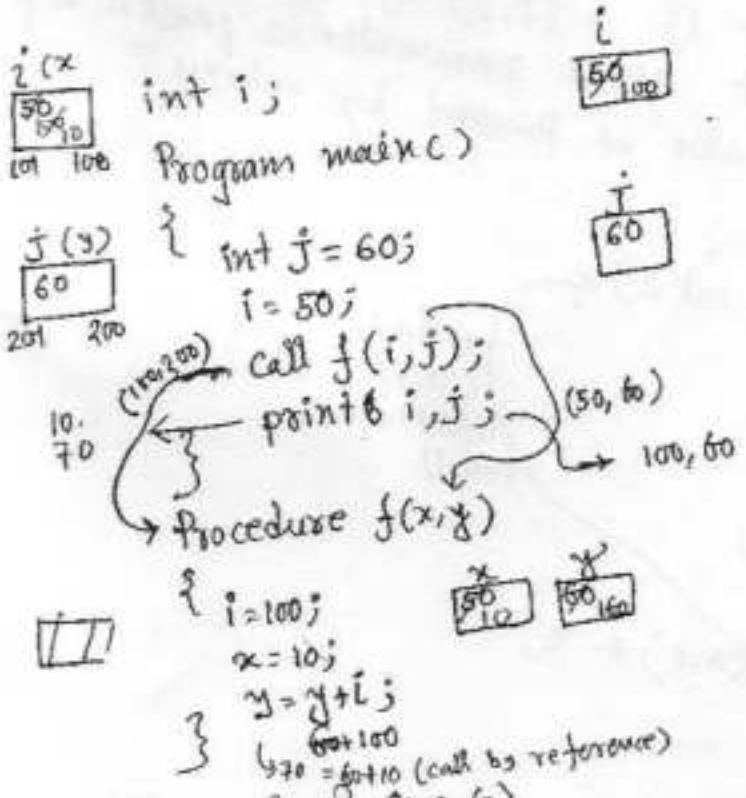
```
{ x=10;
  y=3;
  func1(y, x, x);
  print x;
  print y;
}
```



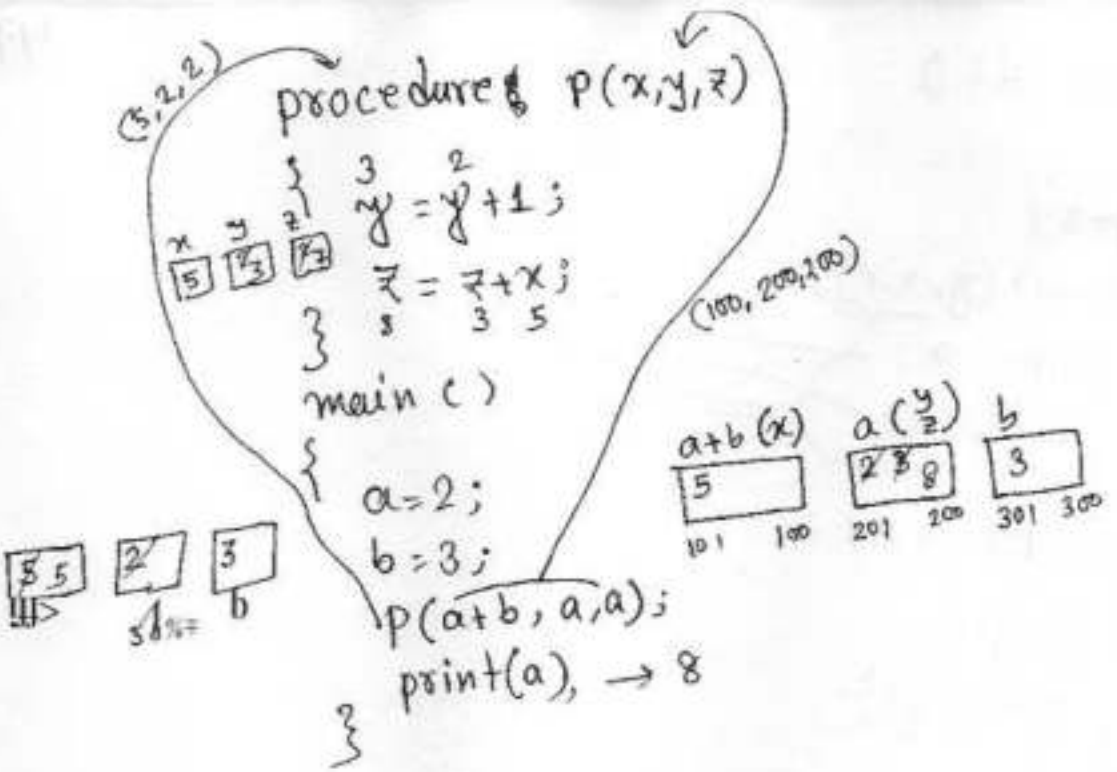

```
func1 (x, y, z)
{ y = y+4;
  z = x+y+z;
}
```

14 10
31 = 3 14 14

* Call by value? (10, 3)
* Call by reference? (31, 3).



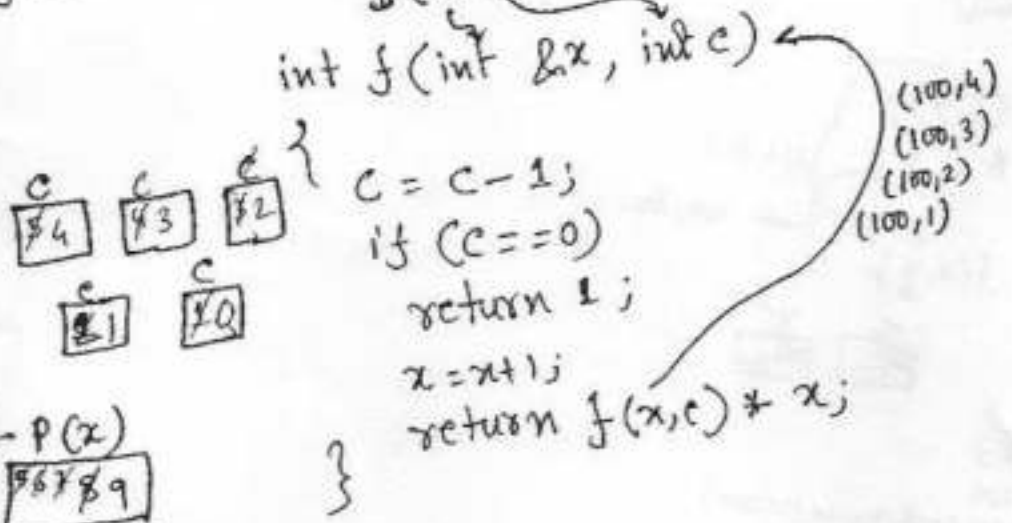
Call by value? (100, 60)
Call by Reference? (10, 70)



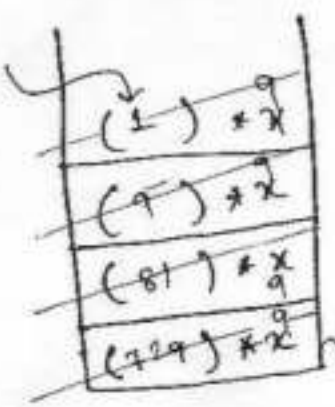
Call by ref? → 8

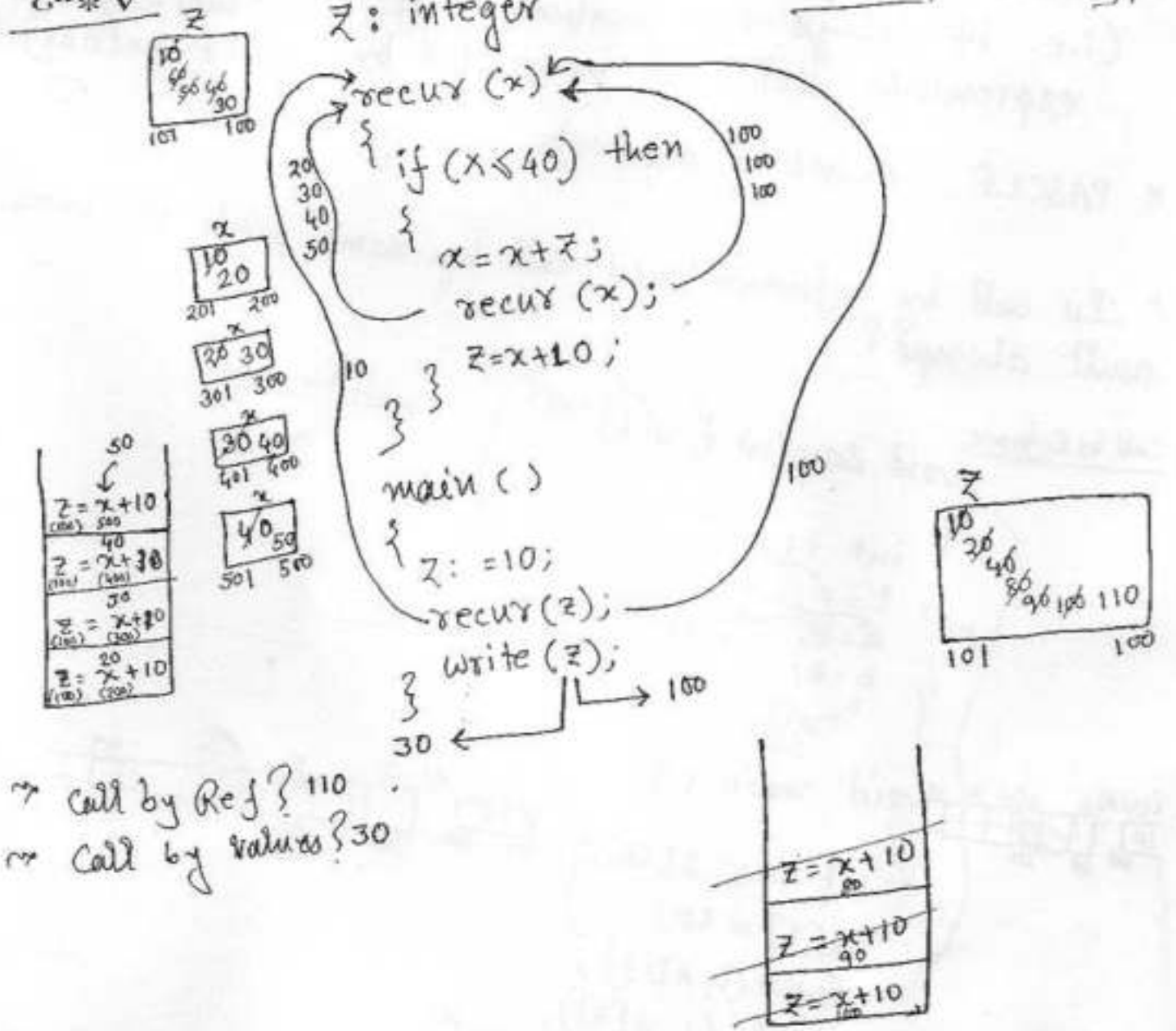
Call by Value Result? → 3 or 7.

? what is the return value of f(P,P) if P=5 initialized before the call? note that first parameter is passed by reference and 2nd parameter is passed by values?



auto variable, will create for 5 times.





* Call by Ref? 110
* Call by values? 30

* Call by Name :

```
void Swap (int a, int b)
{ int t;
  t = a;
  a = b;
  b = t;
}
```

```
void main ()
{ int k=50, m=45;
  Swap (k, m);
  print (k, m);
}
```

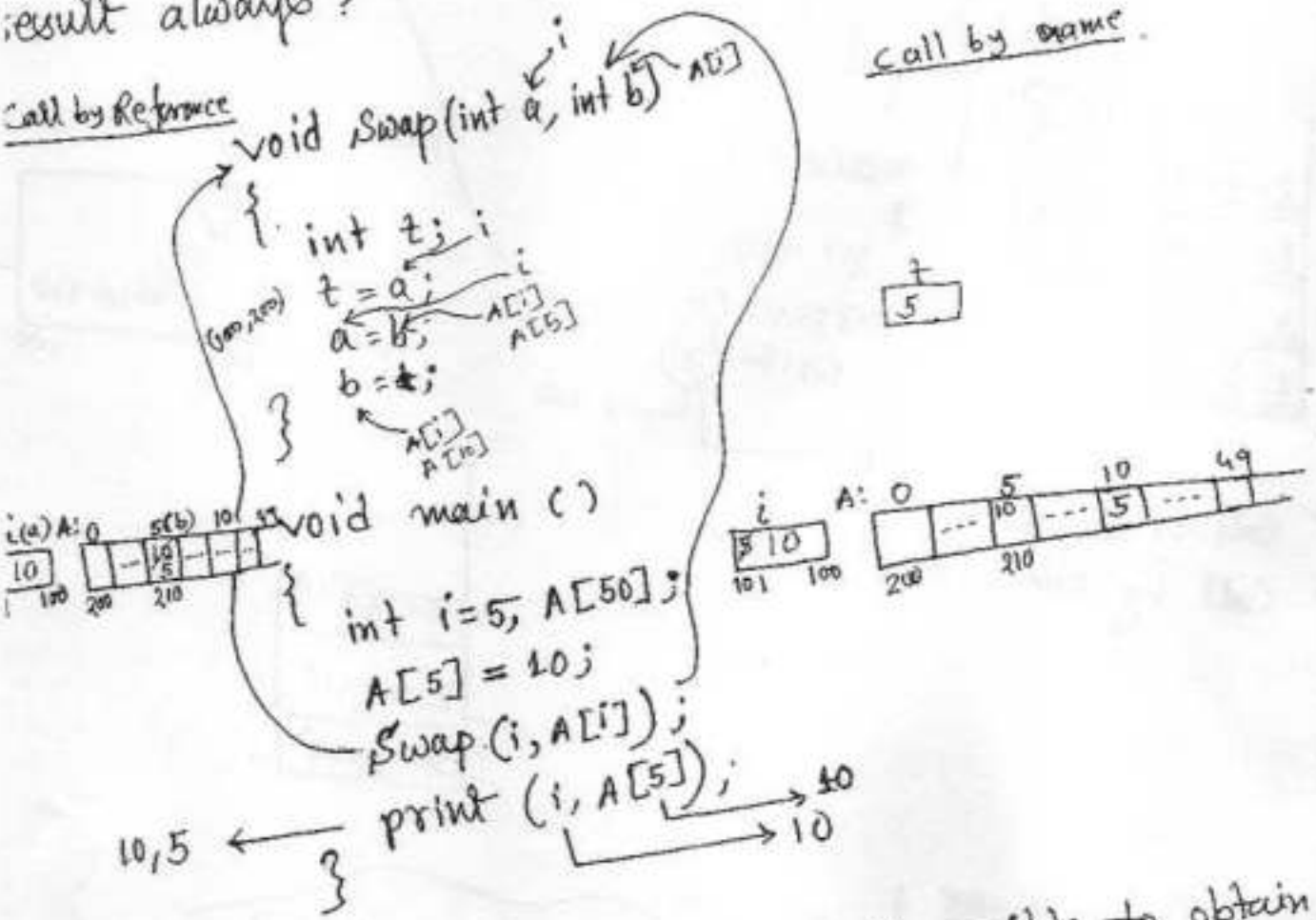
45, 50

* In call by name formal arguments are renamed or replaced with actual arguments.

* It does not pre-evaluate the functional argument (i.e. it delays the evaluation of functional argument expressions untill it is needed by called function).

* PASCLE, ALGOL 60 supports call by name.

* Is call by reference and call by name prints the same result always?



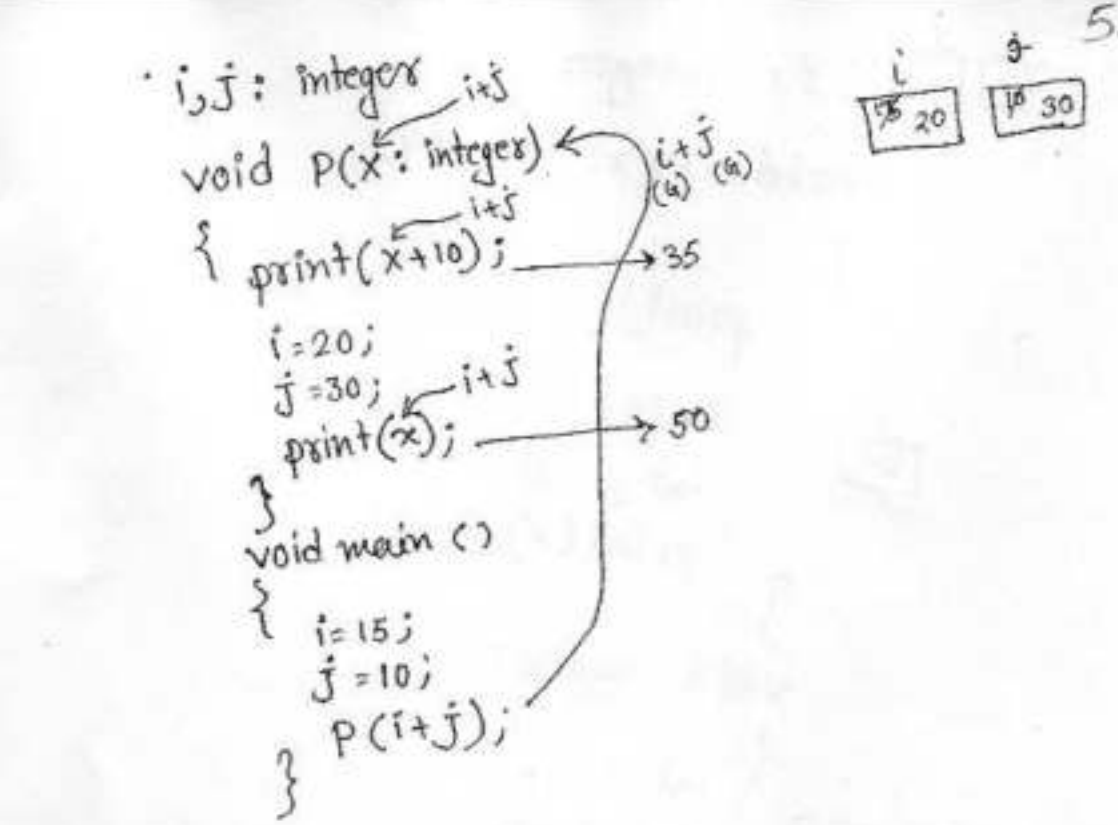
* In which of the following cases, is it possible to obtain different result for call by reference and call by name parameter passing technique?

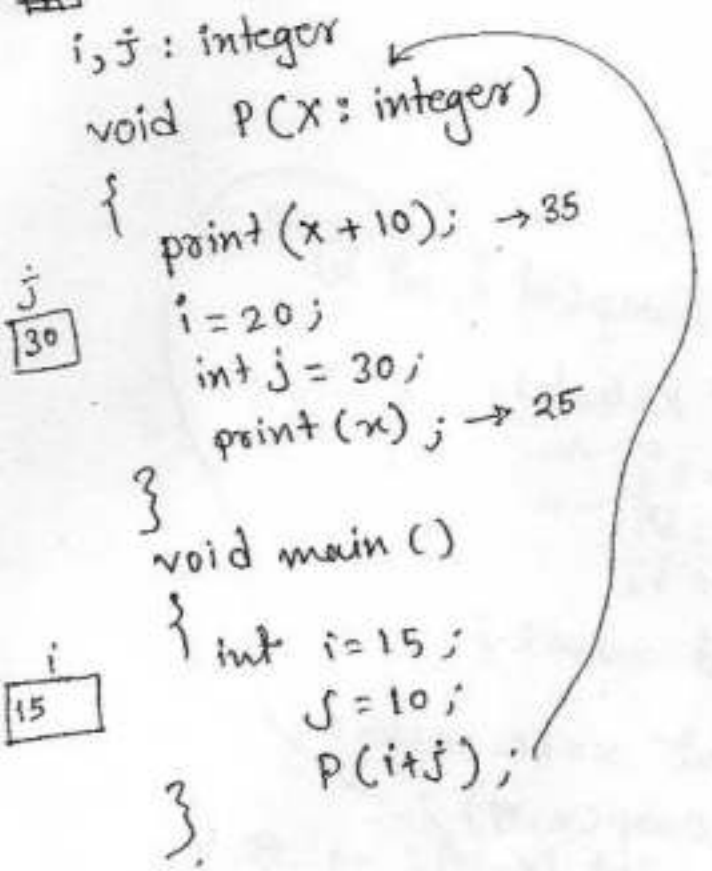
a) Passing a constant value as a parameter.

b) Passing address of array as a parameter.

c) Passing an array as a parameter.

d) Passing an array element as parameter.


i,j : integer
void P(x: integer)
{ print(x+10);
i=20;
j=30;
print(x);
}
void main()
{ i=15;
j=10;
P(i+j);
}
i+j
i+j
i+j
35
50
i
15 20
j
10 30


int
i,j : integer
void P(x: integer)
{ print(x+10); → 35
i=20;
int j=30;
print(x); → 25
}
void main()
{ int i=15;
j=10;
P(i+j);
}
j
30
i
15

```
i, j: integer
void P(x: integer)
{
    printf(x + 10); → 35
    i = 20;
    int j = 30;
    print(x); → 25
}
void main()
{ int i = 15;
  int j = 10;
  P(i+j);
}
```

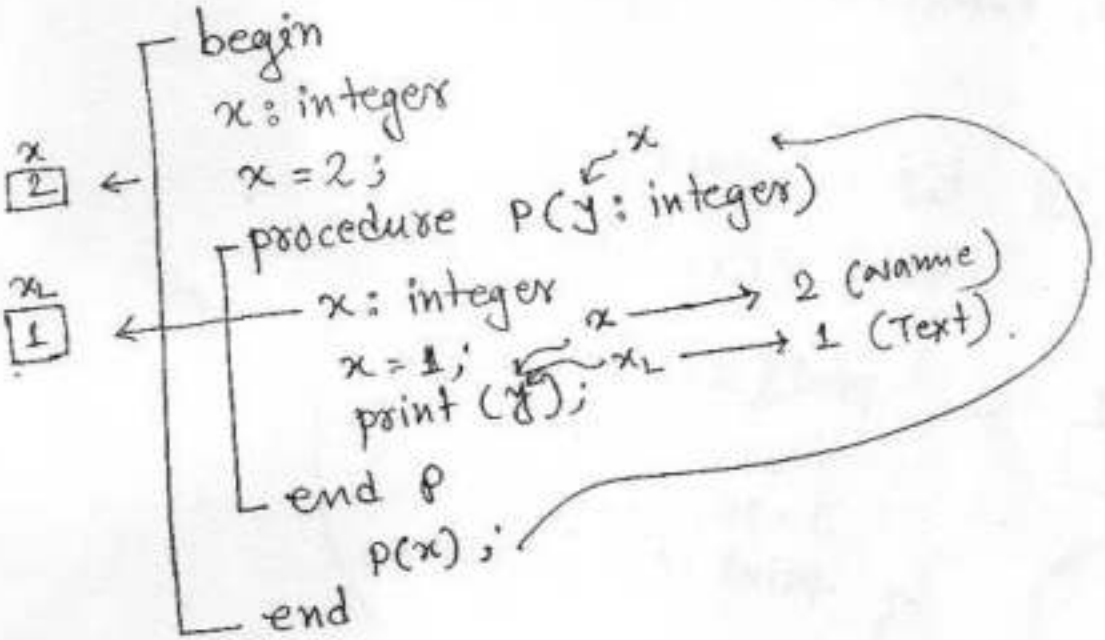
* Call by Text:

```
void Swap(int a, int b)
{ int k=5, t;
  t = a;
  a = b;
  b = t;
}
void main()
{ int k=50, m=45;
  Swap(k, m);
  print (k, m); → 50, 5.
}
```

* It is similar to call by name except that whenever replace variables are defined locally to the called function, then call by text gives preference to local variables.

NOTE

→ If there are no local variables with the same name of replace variable in the called function, then call by text and call by name prints the same result.



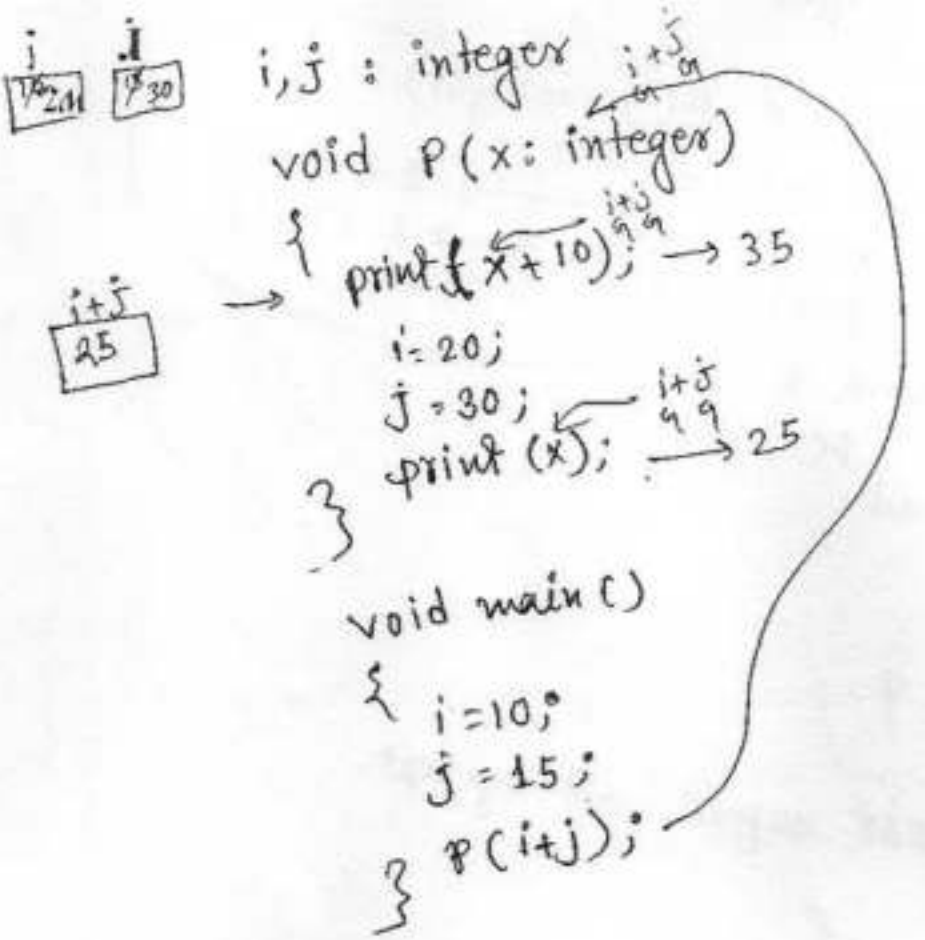
→ Call by text?

→ Call by name?

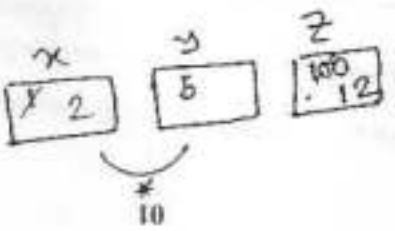
* ALGOL 60, LISP support call by text.

Call by Need.

* It is memorized version of call by name.
* Once the functional arguments are evaluated, that value will be stored for subsequent uses. (i.e. if the actual argument is an expression, then it is evaluated only once and that value is stored for subsequent uses.
* It is a kind of macro expression or lazy evaluation.
* ALGOL 60, FUNCTION PL support call by need.



```
procedure p(a, b, c)
{
  a = 2;
  c = a + b;
}
main()
{
  x = 1;
  y = 5;
  z = 100;
  p(x, x*y, z);
  write(x, z);
}
```

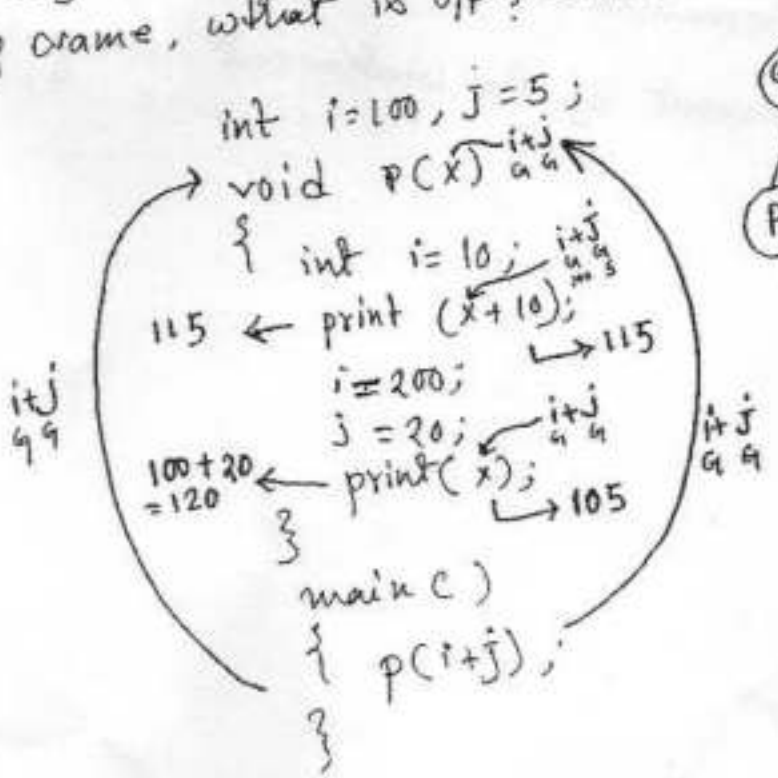


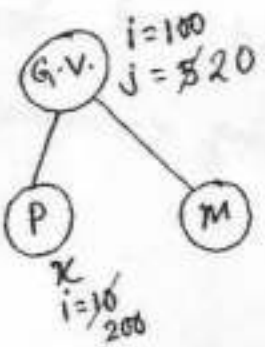
Call by Name: 2, R

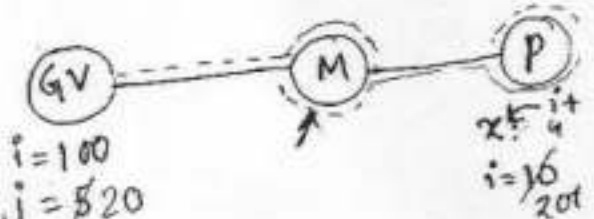
Call by Reference: 2, 7

* If the programming language uses static scope and call by need, what is the o/p? — 115, 105

* If the programming language uses dynamic scope and call by name, what is o/p? — 115, 120







Scope:

Referencing Environment:

→ R.E of a statement is ~~collection of all~~ function of variables which are ~~locally declared and~~ visible in the ~~environment~~.

Static Scope

→ R.E. of a statement in a statically scoped language is the collection of all variables which are locally declared + all ancestor variables which are visible in the same

Dynamic Scope:

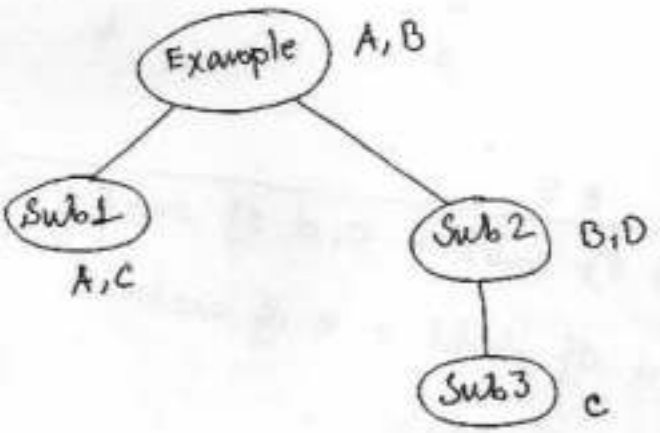
→ R.E. of a statement in a dynamically scopped language is the collection of all variables declared locally + all other active ~~subprogram~~ variable which are visible of the statement.

NOTE:

→ A subprogram is said to be active if its execution has begin and not terminated.

→ find referencing environment of the statement using static scope.

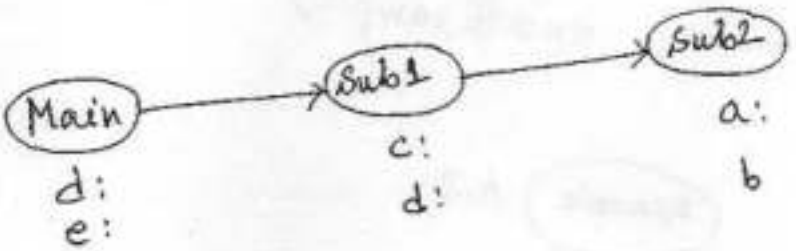
```
Program Example
    A,B : integer
    procedure Sub1
    begin
        A,C : integer ;          ①
    end Sub1 .
    procedure Sub2
        B,D: integer
        procedure Sub3
        begin
            C: integer           ②
        end Sub3
        begin                    ③
    end Sub2
    begin (Example)              ④
    end (Example)
```



Statement	RE
①	A,C of Sub1 + B of Example
②	C of Sub3 + B,D of Sub2 + A of Example
③	B,D of Sub2 + A of Example
④	A,B of Example.

using scope.

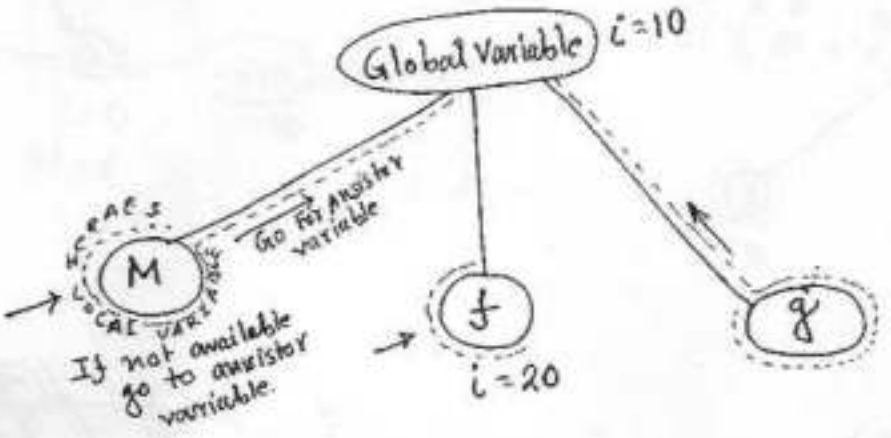
```
Procedure Sub2
{
   int x, b ____ ①
}
procedure Sub1
{  int c,d ____ ②
   Sub2();
}
void main()
{  int d,e; ____ ③
   Sub1();
}
```

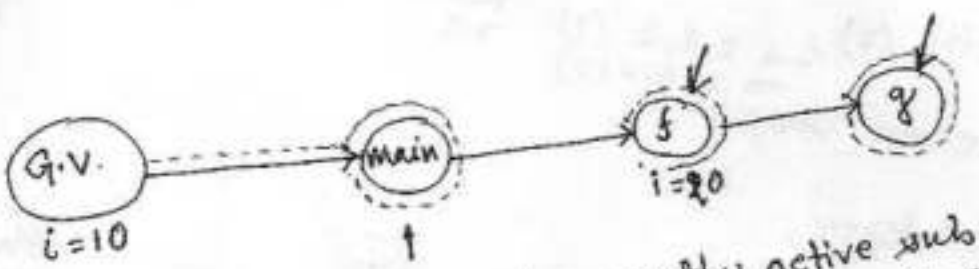


Stmt	RE
①	a, b of Sub2 + c, d of Sub1 + e of main
②	c, d of Sub1 + e of main
③	d, e of main.

```
int i;
Program main()
{ i=10;
  call f();
}
procedure f()
{
  int i=20;
   call g();
}
procedure g()
{ print i; ⟶ 10 (static scoping)
           ⟶ 20 (Dynamic scopping).
}
```

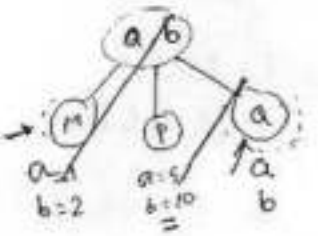
→ what is the o/p if the variables are using i) static scope, ii) dynamic scope?

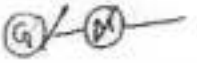




If dynamic scope most recently active sub program variable will be taken. (due to stack) so, o/p = 20.

```
a, b: integer;
procedure P()
{ a = 5;
  b = 10;
}
procedure Q()
{
  a, b: integer;
  P();
}
main
{
  b = 2
  Q();
  write(a,b);
}
```

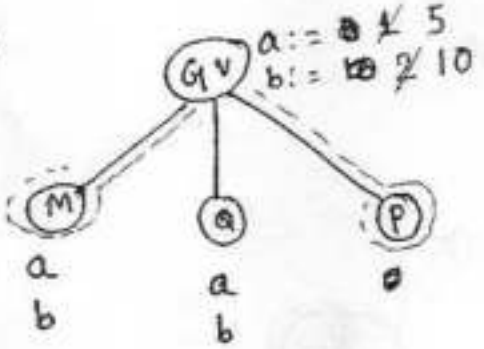




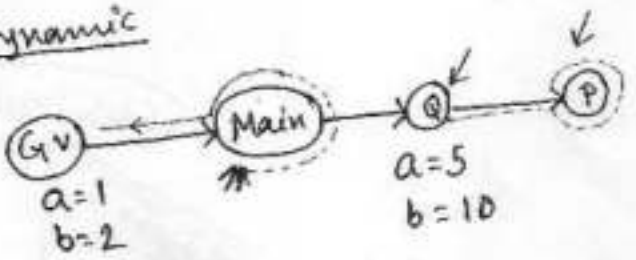
→ (5,10)

→ O,X)

static :

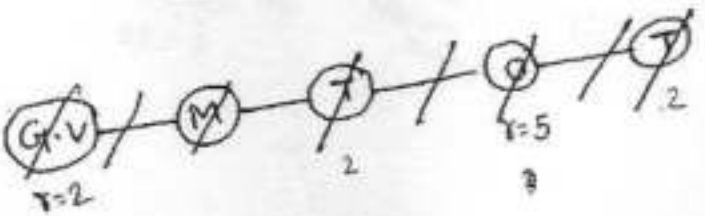


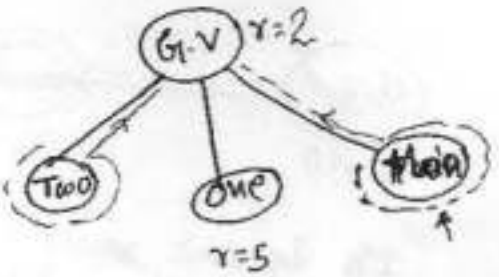
Dynamic

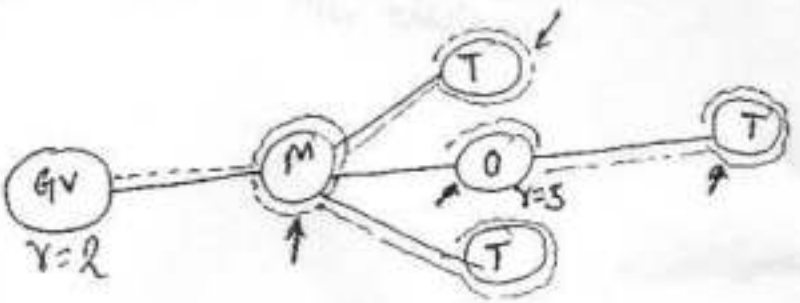



```
r: integer
procedure Two()
{ write(r);
}
procedure One()
{
  r: integer;
  r=5;
  Two();
}
main()
{ r=2;
  Two();
  One();
  Two();
}
```

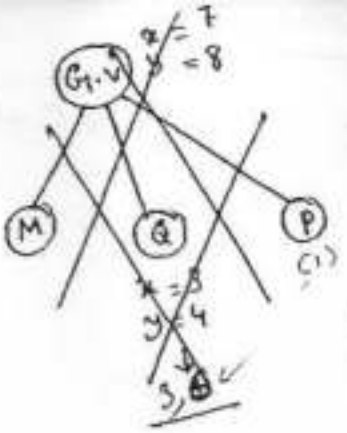
→ 2,2,2 (5)
→ 2,5,2 (0).



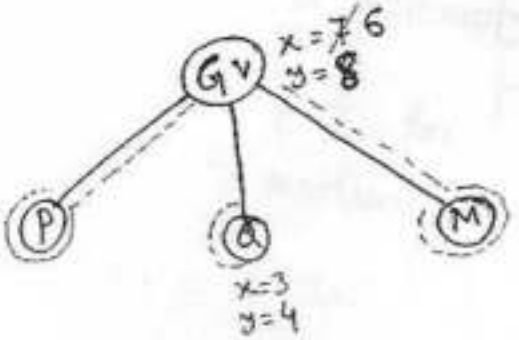


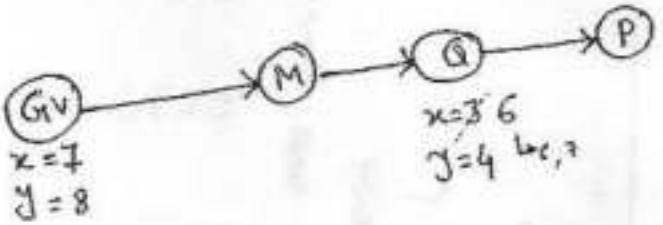


```
x, y : integer;
P (n: integer)
{
   x = (n+2)/(n+3)
}
Q()
|
   x, y: integer;
   x = 3;
   y = 4;
   P(y);          → 6
   write (x);     → 3
}
main ()
Λ  x = 7;
   y = 8;
   Q;             → 7
   write(x);      → 6
}
```



static: 3, 6.
Dynamic: 6, 7.



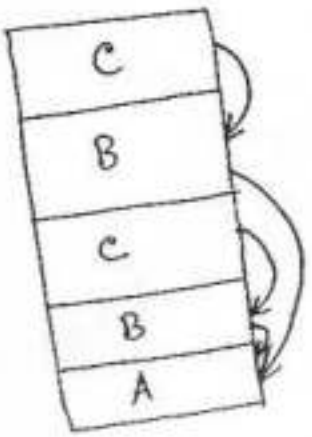


* Create

Creating Static Link :

Calling Sequence A → B → C → B → C

```
function A
{   int i;
    function B
    {   int j;
        function C
        {   int k = i+j;
            B();
        }
        C();
    }
    B()
}
```

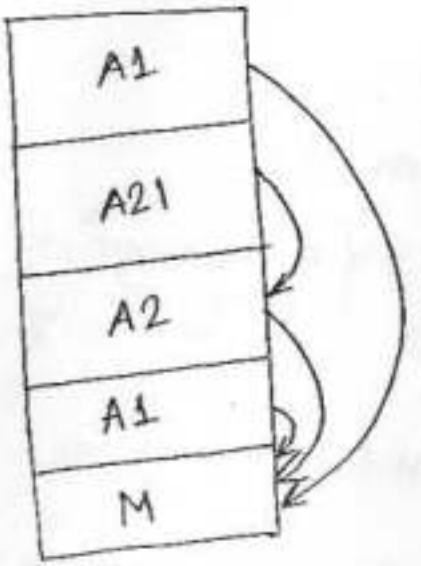


Rule for Creating int k;

Static link must have a static variable which lexically contains it.

→ In the above example A lexically contains B and C.
and B lexically contains C.

```
Program Main
  var...
  Procedure A1
  var...
    call A2
  End A1
  Procedure A2
    var...
    procedure A21
      var...
        call A1
      End A21
        call A21
      En
    End A2
    call A1
  End Main
```

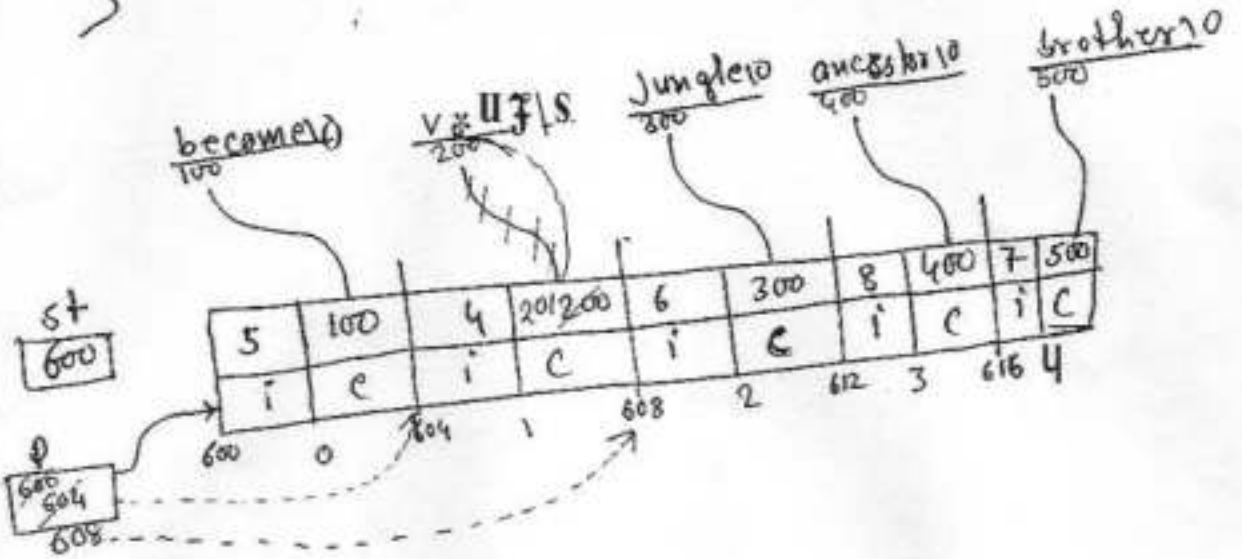


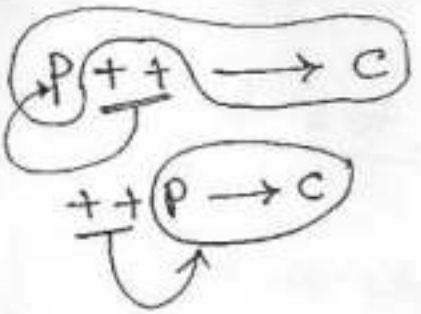
Create static link for the sequence : Main → A1 → A2 → A21 → A1

→ 1

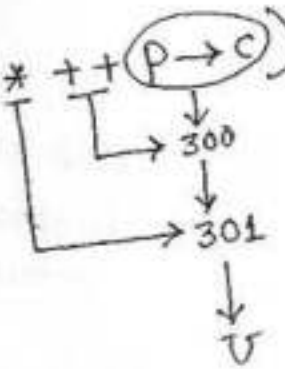
* Pointer to Structure (Continued...)

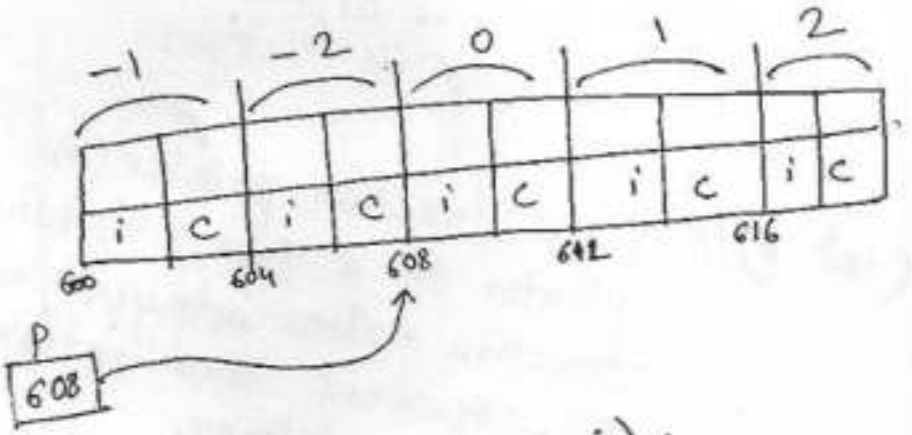
```
struct test
{ int i;
  char *c;
} st = { 5, "become", 4, "better", 6, "jungle", 8, "ancestor",
         7, "brother" };

main()
{ struct test *p = st;
  p+=1;
  ++p->c;
  printf("%s", (p++)->c);
  printf("%c", *++p->c);
  printf("%d", p[0].i);
  printf("%s", p->c);
}
```



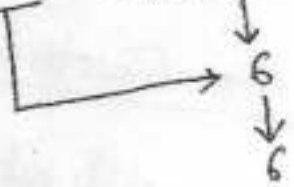


printf ("%c", *++(p→c));

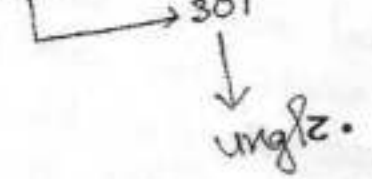




pf ("%d", p[0].i);



pf ("%s", p→c);



→ Pointer to function :

** Syntax to declar a pointer to a function:

return type of function (* pointer variable) (list of arguments of function);

ex:

```
void (*P)( );  // P is a pointer to a function where function takes no argument and its return type is void.

int (*P) (int, int);  // P is a pointer to a function where function takes two integer arguments and its return type is integer.

int (*f)(int *);  // f is function pointer (or pointer to a function) where function takes integer pointer as argument and return type is integer.
```

→ Syntax to Assign a Pointer to function:

Pointer Variable = function name with out parenthesis.

```
Ex:  P = test;  // Every function loaded into the seperate physical memory block in the form of m/c instruction code and entry point address of that block is assigned to the function name (i.e. test). That address is copied into pointer P.
```

→ Syntax to Call a function Using Pointer:

Pointer Variable (list of arguments of function);

or

(* pointer variable) (list of arguments of function);

Ex:

```
void test();
void main()
{
Declaration // void (*P)();
Making pointer // P = test;
Calling // P();
           or
           (*P)();
}
```

```
void test()
{
  pf("ACE");
}
```

→ ACE
→ ACE

→

```
int get();
void put(int);
void main()
{  int k;
D // int (*P)();
  // void (*q)(int);
M // k  P = get;
   10  q = put;
C //   k = P();
       q(k);
}
```

```
int get()
{ return 10;
}
```

10

```
void put(int i)
{
  p("%d", i);
}
```

i: 10

→ 10

10

~~void~~

```
void main()
{ int n;
  void (*P)(); // D
  printf("Enter your choice: 1,2,3");
  scanf("%d", &n);
  Switch(n)
  {
  Case 1: P=X;
          break;
  Case2:  P=Y;
          break;
  Case3:  P=Z;
          break;
  default: printf("invalid choice");
          return;
  }
  P(); // C
}
```

```
void X()
{ printf("we called X");
}
void Y( )
{ printf("we called");
}
void Z ( )
{ printf("we called");
}
```

M

↳ if we remove it, then for invalid i/p by user it will call P. i.e. remaining part will execute.