17/05/20

Programming Languages

1. Storage classes (12-1.)
2. Recursion (24.1.)
3. Pointers (33.1.)
   - Basics
   - Pointer to pointer
   - to array element
   - to one dimensional array
   - to two dimensional array
   - to string
   - to structures
   - to functions
   - to array of pointers
   - to arry of function pointers
4. Static and Dynamic scope. (12.1.)
5. Parameter passing techniques (4.1.)

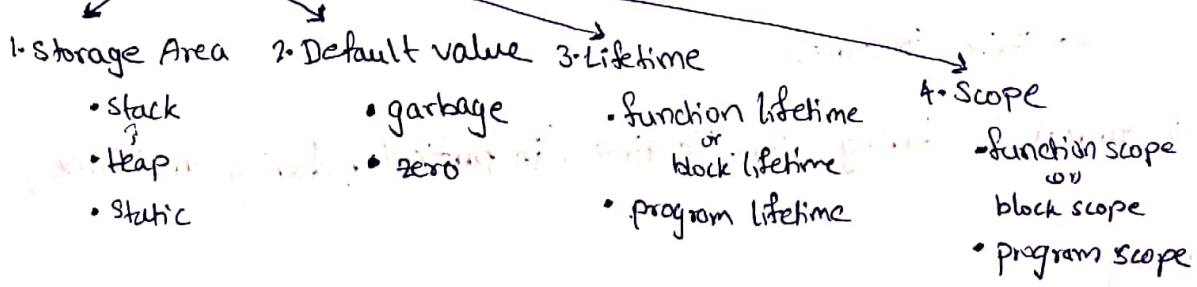
General questions (15.1.)

Ref Books: C Programming by Dennis Ritchie

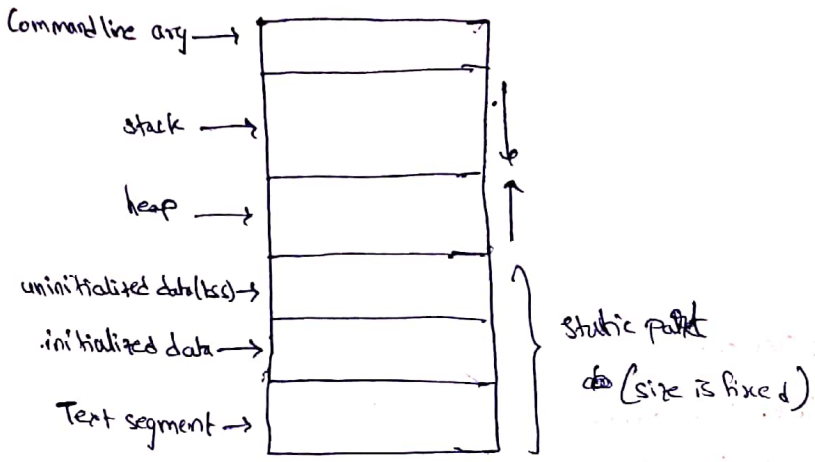
Storage Classes:

Consider

```
int i=10;
int k;
```



Memory Organization of C program:



Stack Memory:

It contains

1. Local variables (Auto variables)
2. Formal parameters
3. Return address of the function
4. Function calls (Recursive calls)

→ It is allocated during runtime of the program.

Static Memory:

It contains

1. static variables
2. global variables
3. Function code
4. String Constants

→ allocated during compile time (in particular during load time)

Heap Memory:

→ It is allocated by explicit functions like malloc(), calloc()

→ It is deallocated by free()

→ allocated during Runtime of the program

→ It is used for Dynamic Data Structures.

Storage Classes:

1. Storage Area (stack, static, heap)
2. Default value (Garbage value, 0)
3. Lifetime (Function lifetime, program lifetime)
4. Scope (Function scope, Program scope)

Types of Storage classes:

1. Auto
2. Register
3. static
4. Extern

19/05/20

1. Auto:

Syntax: auto datatype variablename;

Eg: auto int i;

By default all local variables are auto variables.

→ Storage area of auto variable is stack memory (Runtime)

→ Default value: Garbage value.

→ Lifetime: function lifetime

→ Scope: function scope

Eg:

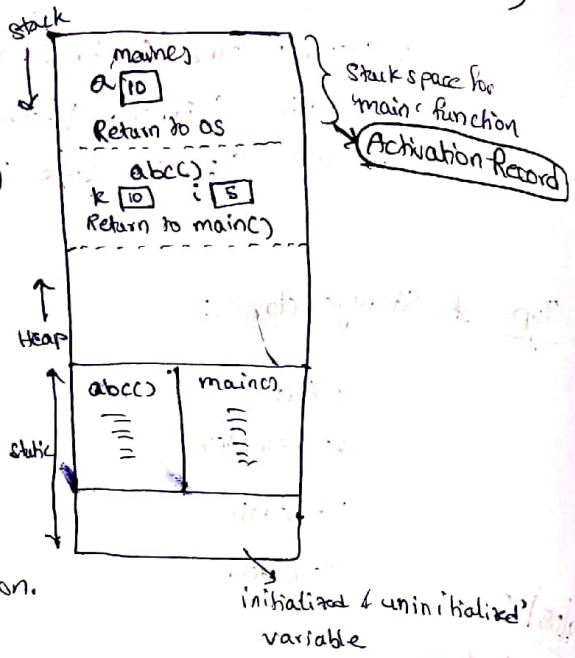
```
void abc (int k)        → formal parameters (should be stored in stack memory during runtime)
{
    auto int i=5; → stack Memory (~~runtime~~)
    ++i;
    k=k+1;
    printf("%d", k);
}
void main()
{
    int a=10; → auto variable (∴ stack memory)
    abc(a);
    abc(a);
    abc(a);
    Printf("%d", a);
}
```

Compilation:

→ During compilation choice of memory layout is finalized. But stack memory is not allocated (i.e., only choice of memory layout is finalized) (for auto variable) i.e., deciding where to store the variable

→ Compilation is done from the top instruction (i.e., top down approach)

→ After compilation of abc(), It is stored in static memory. (Machine code of abc() is stored in static memory.)



→ Now main() is also converted into machine code and it is stored in static memory

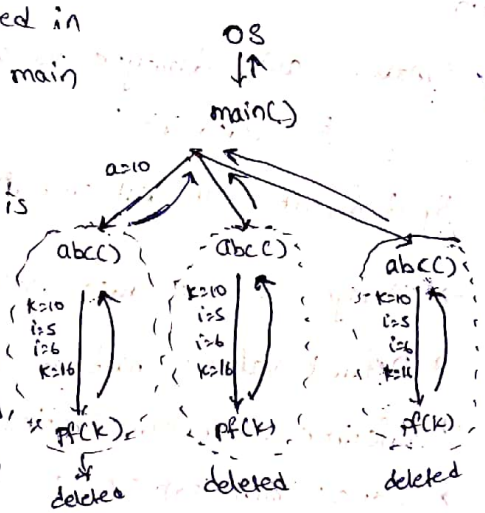
Execution: (Run time)

→ Execution of program is always started from main().

Operating system initially calls main function.

→ So the function call of 'main' is stored in the stack. Along with local variables of main function are stored in stack.

So now the memory for variable 'a' is allocated in the stack.



→ Activation Record contains auto variables, formal parameters, Return address, function calls

→ Now to call 'abc()' abc's activation record must be stored into stack

→ As the program executes, the values of auto variable & formal parameters, in the stack, are modified.

→ Once the first call of 'abc' is executed control goes back to main and the activation record of the 'abc()' call is deleted.

→ Now in main() program counter is incremented.

→ So 'abc()' is thus called again

→ So 'abc()' is called again for 3rd time.

→ After all this main()'s execution is finished.

→ Activation record of main() is deleted. Now stack becomes empty.

→ Finally, static memory of main & abc is also released.

2. Static Variables:

→ Syntax: static datatype variable_name;

Eg: static int i;

→ storage area: static memory. (compile time (load time))

default value : 0

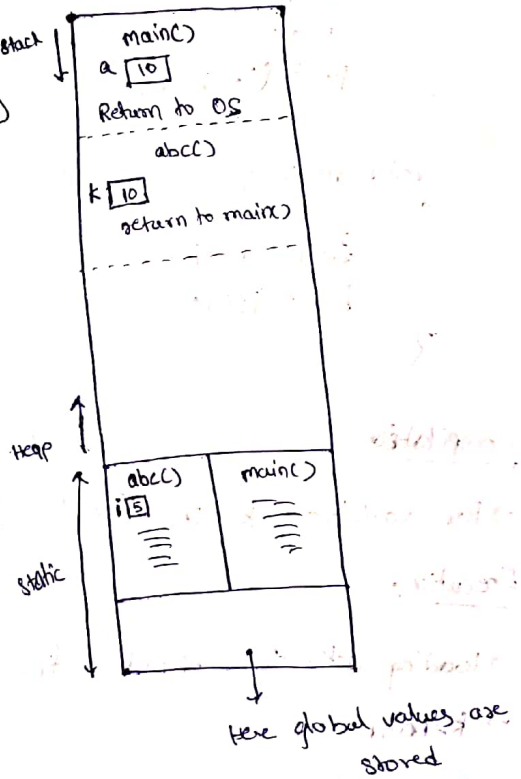
Lifetime : program lifetime

scope : function scope

Eg:

```
void abc(int k)
{
    static int i=5; (Static memory)(Compiletime)
    ++i;
    k=k+i;
    printf("%d",k);
}
void main()
{
    int a=10; (Stack memory)(Runtime)
    abc(a);
    abc(a);
    abc(a);
    printf("%d",a);
}
```

o/p: 16, 17, 18, 10



Compilation:

→ Each function is converted into its machine code and stored in static memory.

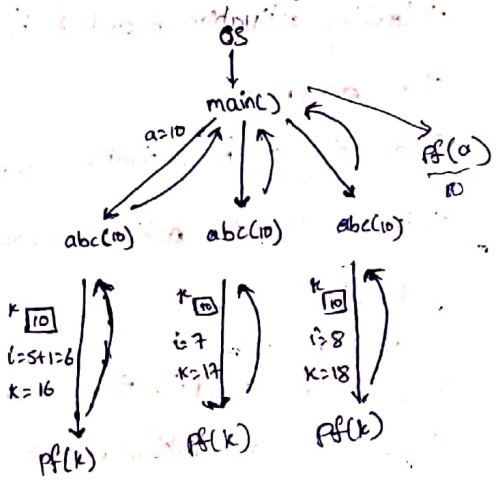
Execution:

→ During execution of 'abc()'

'static int i=5'

The above line is skipped. No action is performed for above line.

→ When activation record is deleted, the deletion doesn't affect 'i'. Because 'i' is stored in static part of memory. So value of 'i' is retained b/w function calls

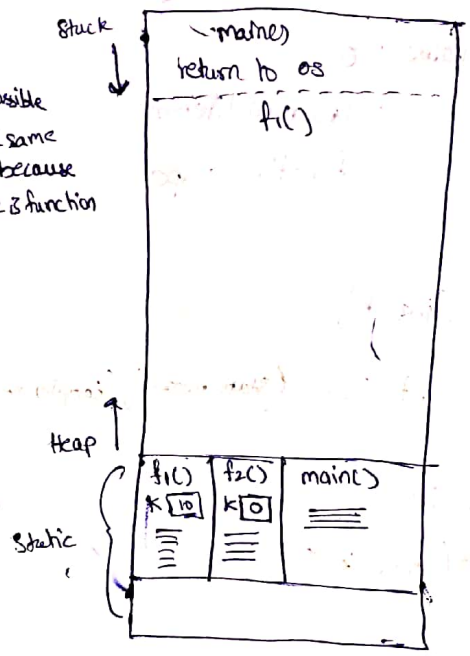


→ Static memory is deleted as soon as control returns to operating system.

→ Memory is allocated for 'i' only once.
Memory is allocated for 'k' thrice.

Eg:
```
void f1()
{
    static int k=10;
    ++k;
    printf("%d", k);
}
void f2()
{
    static int k;
    k=10;
    ++k;
    printf("%d", k);
}
void main()
{
    f1(); f1();
    f2(); f2();
}
```

It is possible to have same name because the scope is function scope.

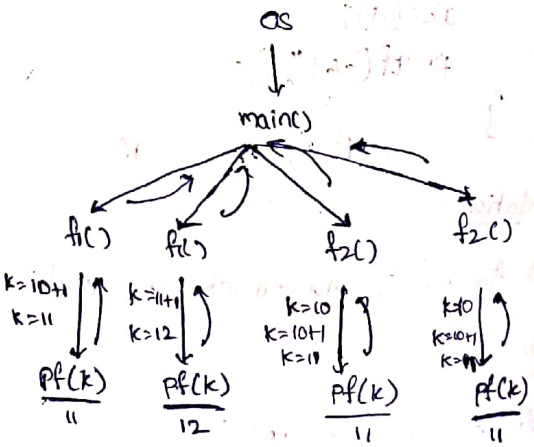


Compilation:

→ For variable 'k' in 'f2()', default value '0' is initialized.

Execution:

→ loading activation record of 'f1'.
→ k=10+1=11
→ deletion of the activation record
→ Control back to main()
→ loading activation record of 'f1'
→ k=11+1=12
→ deletion of the activation record
→ Creation & loading activation record of 'f2'
→ k=10
→ k=10+1=11
→ delete of the activation record
→ loading activation record of 'f2'
→ k=10



→ k = 10 + 1 = 11

→ deletion of activation record of f2

→ Activation record of main is deleted

→ Control is back to O.S

→ Static memory is also released.

(Q1) what is the o/p of below program

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

a) 2,3,4,4,3,2 b) 2,3,4,4,4,4

c) 2,3,4,3,2 d) 2,3,4,4,4

It is possible to call main() inside main()

Initial values assigned to static variables must be constants, but not expressions

sol:

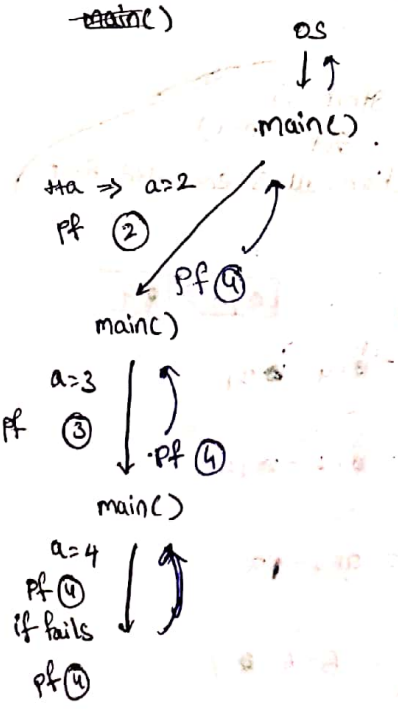
~~++a => a=2~~

~~pf~~ ——— 2

~~main()~~

Static

a=1

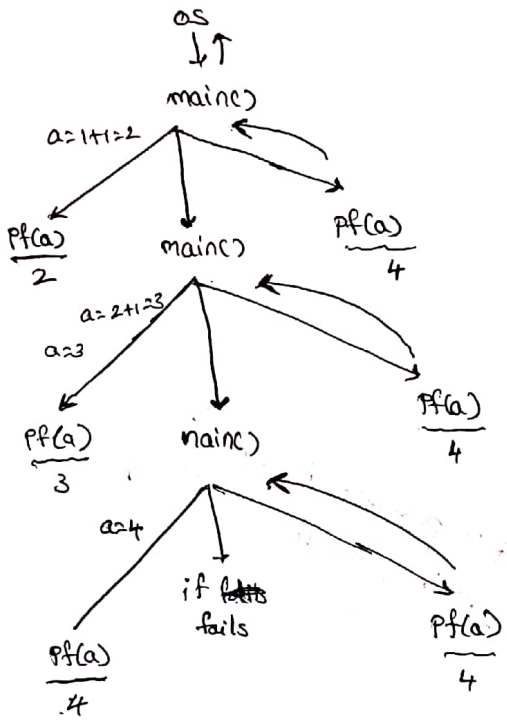


∴ From fig the sequence is

2,3,4,4,4,4

∴ opt (b)

a better diagram for above question

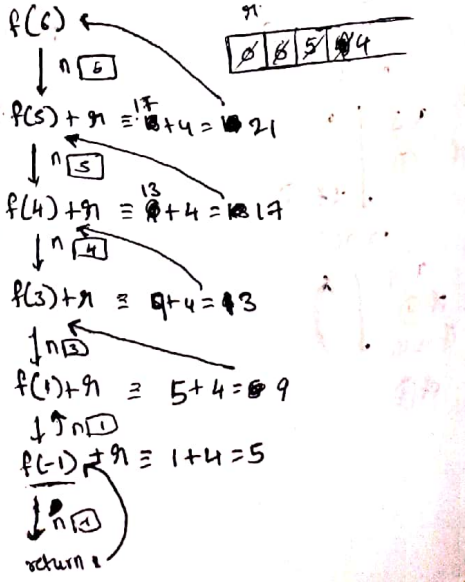


(Q2) Find the return value of f(6) for below function

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

In f(n-1)+x
f(n-1) is executed first and later 'x' value at that time is added

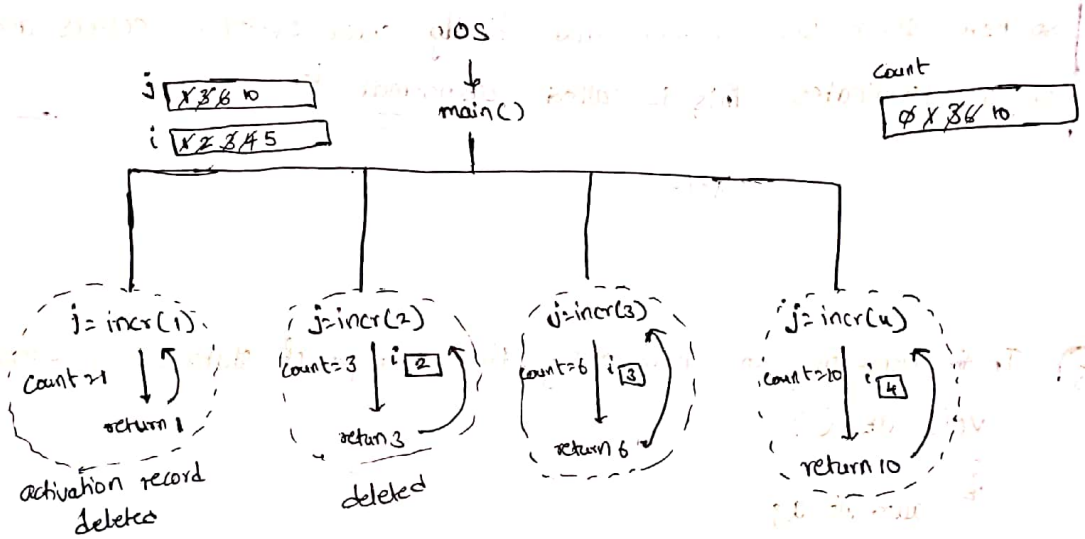
Even if stmt is
x + f(n-1)
function call is executed first



∴ f(6) = 21

(Q3) what is the value 'j' after the execution of below program.

```
int incr(int i)
{
    static int count=0;
    count = count+i;
    return count;
}
void main()
{
    int i,j;
    for(i=1; i<=4; i++)
        j=incr(i);
    printf("%d", j);
}
```



Now pf(j) is executed

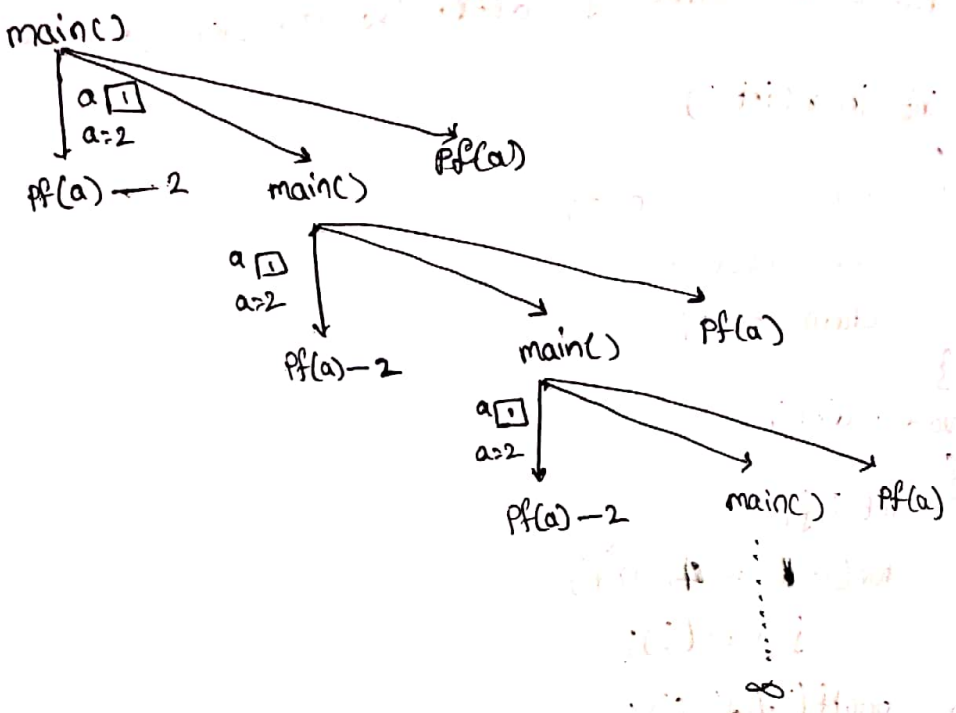
Hence value in j is 10.

(Q4) What is the o/p of below program

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

a) 2,2,2,2,2,2 b) 2,3,4,4,4,4 c) Infinite loop d) Abnormal Termination

Sol:



So here stack size increase and finally stack overflow occurs and program terminates. This is called abnormal termination.

∴ opt (d)

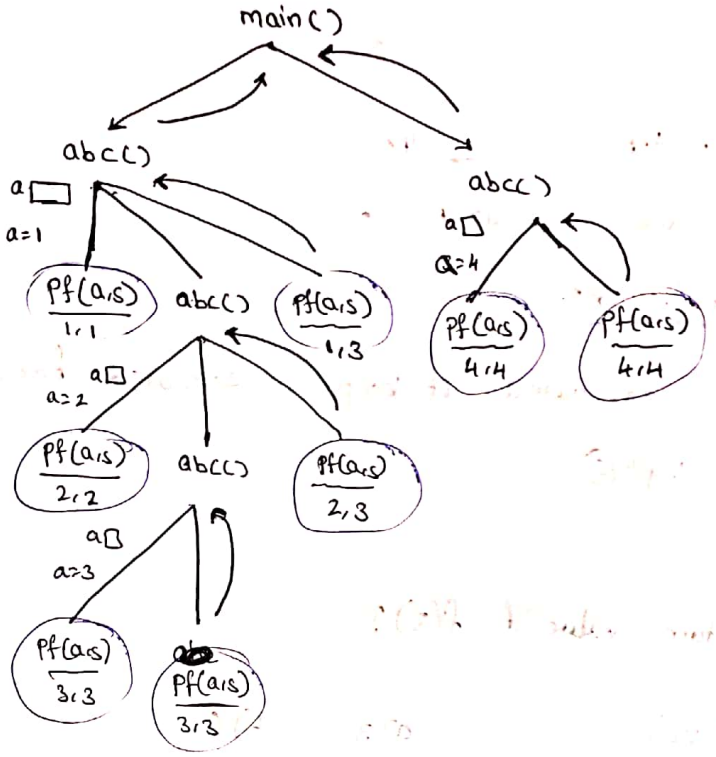
(Q5) In the below program, how many times the printf statement is executed?

```
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();
}
```

Sol:

S

0x7734



∴ printf is executed 8 times.

(Q6) what is return value of f(3)?

```
int f(int val)
{
    int x=0;
    while (val > 0)
    {
        x = x + f(val-1);
    }
    return val;
}
```

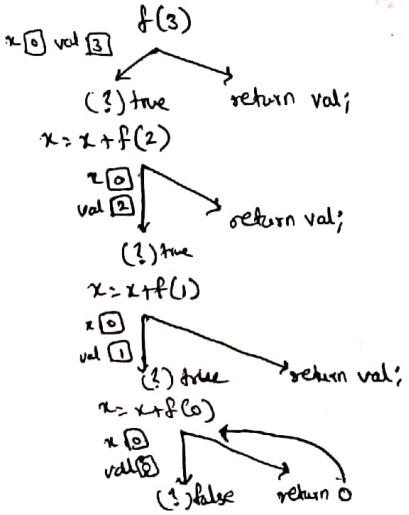
a) 3 b) 6

c) Infinite loop

d) Abnormal termination

A.R - Activation Record

Sol:



→ So for now of A.R = 4

→ Now after return 0, control move to the function f(1)

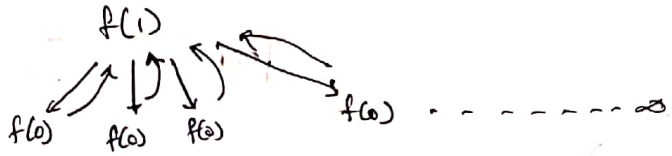
→ Now, no of A.R = 3

→ As we are in while loop, x=0 and val(1) > 0 so x=x+f(0) executes again

→ So f(0) is called again.

→ No of A.R = 4

→ Now this process continues in a loop.



Note that here no of A.R $\overset{1}{\underset{}{}}$ 2, 3, 4, 3, 4, 3, 4, ... ∞

∴ Stackoverflow doesn't occur here.

So this called infinite loop.

Here program is never terminated. It keeps on executed forever.

∴ opt (c)

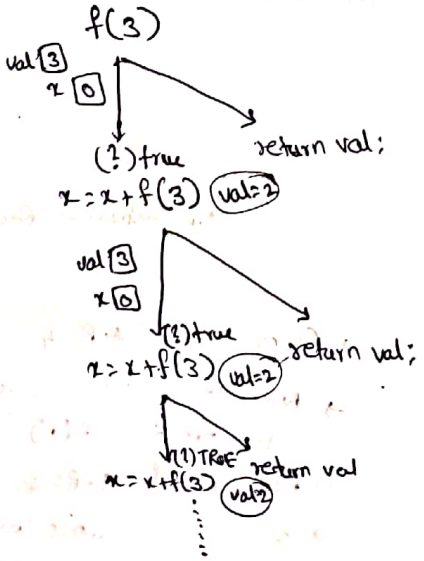
(Q7) what is the return value of f(3)?

```
int f(int val)
{
    int x=0;
    while (val>0)
    {
        x= x + f(val--);
    }
    return val;
}
```

a) 3 b) 6

c) Infinite loop

d) Abnormal Termination



given is

f(val--)

so we need decrement after calling the function.

Here stack size keeps increasing and finally stack overflow occurs

∴ Abnormal Termination occurs.

∴ opt (d)

Q8 GATE 2019

what is the o/p of the program

```
int f()
{
    static int num = 7;
    return num--;
}
void main()
{
    for(f(); f(); f())
    {
        printf("%d", f());
    }
}
```

a) 5, 2
b) 4, 1
c) 6, 3
d) 6, 3, 0

Sol:

(0) F → condition fails ∴ loop terminates

for(f(); f(); f())
(7) (6) T (4)
(3) T (1)

printf(f())
(5)
(7)

num
7 6 5 4 3 2 1 0 -1

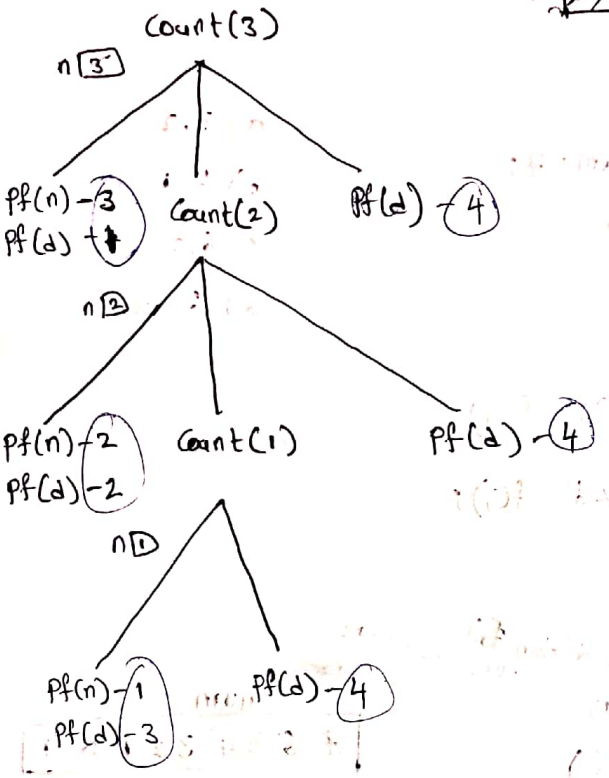
0 — false
non-zero — true

∴ option (a)

Q9 Find the o/p below program

```
void count(int n)
{
    static int d = 1;
    printf("%d", n);
    printf("%d", d);
    d++;
    if(n > 1)
        count(n-1);
    printf("%d", d);
}
void main()
{
    count(3);
}
```

a) 3, 1, 2, 2, 1, 3, 4, 4, 4
b) 3, 1, 2, 2, 1, 3, 4
c) 3, 1, 2, 1, 1, 1, 2, 2, 2
d) 3, 1, 2, 1, 1, 1, 2

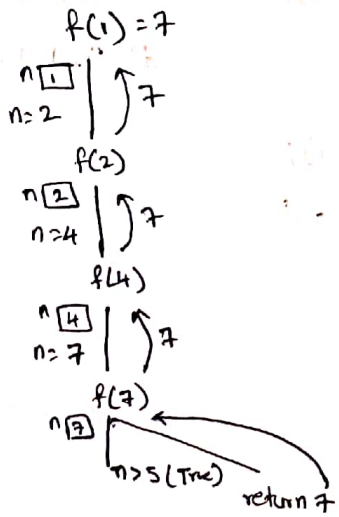


∴ O/P :- 3, 1, 2, 2, 1, 3, 4, 4, 4

(Q10) what is the return value of f(1)?

```
int f(int n)
{
    static int i=0;
    if (n>5)
        return n;
    i++;
    n=n+i;
    return f(n);
}
```

a) 5
b) 7
c) 8
d) 9



∴ f(1) = 7

∴ opt (b)

20/05/19

External Variables (Global variable)

→ It is a variable declared outside of every function.

→ Storage area: static memory (uninitialized part (bss) or initialized part (ds))

bss - block started with symbol

ds - data segment

* uninitialized global variable is stored in bss
* initialized global variable is stored in ds
* function code is stored in text segment.

Default value: 0

Lifetime: program lifetime

Scope: program scope provided that there is no local variable with same name.

If there is a local variable with same name, then preference is given to local variable.

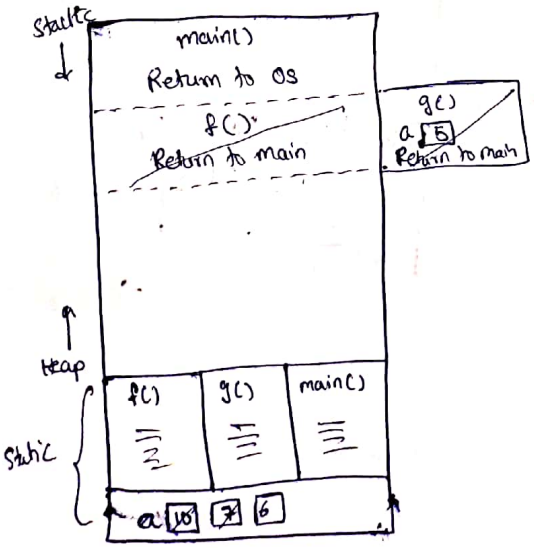
→ Global variable need not to be declared on top of function blocks.

Ex:

	Global (a)	Local (a)
f()	✓	×
g()	×	✓
main()	✓	×

```
int a=10;
void f()
{
    a=6;
    printf("%d", a)
}
void g()
{
    int a=5;
    printf("%d", a);
}
void main()
{
    a=7;
    f();
    g();
    printf("%d", a);
}
```

Compilation:



→ During compilation choice of memory layout of 'a' in 'f()' is ~~chosen as~~ 0 refered to global.

→ But for 'a' in 'g()' compiler recogonizes it as auto and no memory is allocated.

→ In 'main()', the memory layout of 'a' is refered to global.

Execution:

O/P: 6, 5, 6

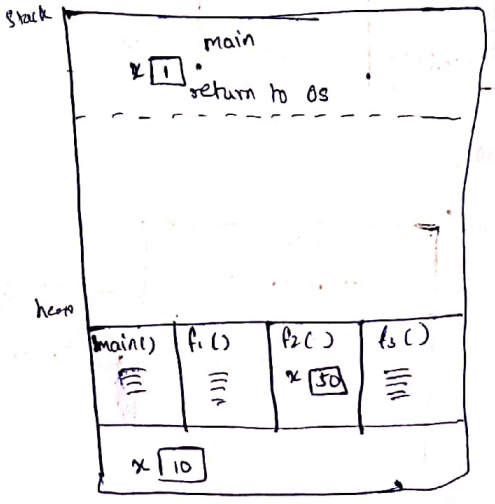
f() g() main

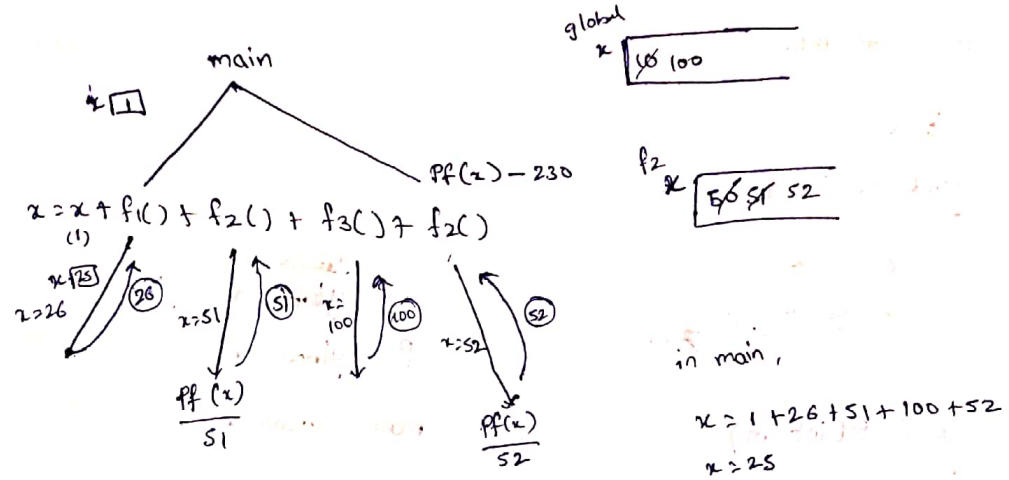
In g() preference is given for local variable 'a'

Eg: Find the O/P?

```
int x=10;
int f1();
int f2();
int f3();
void main()
{
    int x=1;
    x= x+f1()+f2()+f3()+f2();
    printf("%d", x);
}
int f1()
{
    int x=25;
    ++x;
    return x;
}
int f2()
{
    static int x=50;
    ++x;
    printf("%d", x);   return x;
}
int f3()
{
    x=x*10;
    return x;
}
```

	Global (x)	Local (x)
f_1	✗	✓
f_2	✗	✓
f_3	✓	✗
main	✗	✓





∴ O/P: 51, 52, 230

(Q11) Find the O/P of below program.

```
int x;
void Q(int z)
{
    z = z+x;
    printf("%d", z);
}
void P(int m)
{
    int x = 6;
    x = x+m;
    Q(x);
    x = x-1;
    printf("%d", x);
}
void main()
{
    x = 5;
    P(x);
    printf("%d", x);
}
```

Note:

Global variables can be assigned initial values as a part of variable definition. But initial values must be expressed as constants rather than expressions

int i = 2; → valid
int i = 2*3; → valid
int i = 2*j; → Invalid

static, x 5

main → P(5)
x = 11 — Q(11) return to P(5) → x = 11-1, x = 10 return to main → pf(x) - 5
z = 11+5
pf(z) - 16
pf(x) - 10

∴ O/P: 16, 10, 5

Q12 Find o/p below program.

```
int x;
void Q(int z)
{
    z = z + x;
    printf("%d", z);
}
void P(int m)
{
    x = 6;
    x = x + m;
    Q(x);
    x = x - 1;
    printf("%d", x);
}
void main()
{
    int x = 10;
    P(x);
    printf("%d", x);
}
```

Q - global 'x'
P - global 'x'
main - local 'x'

x: 0 → 16

Sol:

main
x [10] ⟶ P(10)
x = 6 + 10
x = 16 ⟶ Q(16)
z = 16 + 16
z = 32
Pf(z) → 32
Return to P(10)
x = 16 - 1
x = 15
main() ⟶ pf(x) → 15

∴ o/p : 32, 15, 10

Q13 GATE 2002

```
#include <stdio.h>
int a, b, c = 0;
void ptrFun(void);
int main()
{
    static int a = 1; /* line 1 */
    ptrFun();
    a += 1;
```

```
    prtFun();
    printf("\n %d %d", a, b);
}
void prtFun(void)
{
    static int a=2; /* line 2*/
    int b=1;
    a++ = 1
    a+ = ++b;
    printf("\n %d %d", a, b);
}
```

Sol:

global c [0]

global b [0]

global a [0]

main 'a' [2]

prtFun 'a' [2 4 6]

	Global			Local		
	a	b	c	a	b	c
main	x	✓	✓	✓	x	x
prtFun	x	x	✓	✓	✓	x

main() → prtFun()

b [1]

a = a + b

a = 2 + 2

a = 4

b = 2

pf(a,b) - 4,2 —return to main()→ a = 2

b [1]

a b = 2

a = 6

pf(a,b) - 6,2 —return to main()→ pf(a,b) - 2,0

∴ O/P: 4 1 2

6, 2

2, 0

Register:

→ syntax: register datatype variable_name

Eg: register int a;

→ storage area: CPU (register)

Default value: Garbage value.

Life time : function lifetime

scope : function scope

Note:

* We can't access address of register variable.

Accessing the address of register variable gives compiletime error

```
Eg: void main()
    {
        register int i;
        scanf("%d", &i);
    }
```

↳ accessing address is not possible

Compilation error

So pointers concept is not applicable for register variables.

* Generally all variables are stored in RAM. During the time of execution, the variables are moved into registers. But for a register variable this moving time is reduced. So accessing register variables is fast.

* But making all variables 'register' is not possible as no of registers available is limited.

* Declaring a variable as 'register' doesn't guarentee that variable is stored in register. It depends on availability of register. If registers are not available, then values must be stored in RAM. This process of checking of if register is available or not is done at runtime.

Find o/p of below program.

Eg:

```
int a,b,c =0;
void prtfun();
void main()
{
    static int a=1;
    prtFun ();
    a+=1;
    prtfun();
    printf("%d %d", a,b);
}
void prtFun()
{
    register int a=2;
    int b=1;
    a+= ++b
    printf("%d %d", a,b);
}
```

global a | 0

global b | 0

global c | 0

~~register~~ After a function returns, register variable is removed from registers

main()

a [1] ⟶ prtfun

a [2]

b [1]

~~b~~ b=2

a=4

pf(a,b) (4,2) return to main ⟶ a=2 ⟶ prtfun()

a [2]

b [1]

b=2

a=4

pf(a,b) (4,2) return to main ⟶ pf(a,b)

(2,0)

∴ o/p: 4,2

4,2

2,0

Extern:

Model 1:

```
void main()
{
    x=5;
    printf("%d",x);
}
int x=6;
```

Compilation error

Model 2:

```
void main()
{
    extern int x;   → no memory is allocated for x in main()
                    → This stmt has nthg to do during runtime
    x=5;
    printf("%d",x);
}
int x=6;
```

→ Global variables need not to be on top of the program.

→ In Model 1, x is a global variable. By the time compiler reaches 'x=5' line, ~~comple~~ compiler thinks that x is not declared and throws an error.

→ In Model 2, we have added a line 'extern int x'. This line tells compiler, that x is declared as global, somewhere else in the program.

→ In Model 2, after seeing 'int x=6' compiler makes compilation sucessful.

→ During runtime,

no action is performed for 'extern int x'.

This statement is used only during compile time.

→ In model 2, ~~Execution~~ is

x is stored in static part (cuz it is global)

(This is done during compilation).

∴ x | 6 |

During execution

x=5

x
5

pf(x) — (6)

→ ⑧ Extern is used to say compiler that global variables are declared elsewhere in the program.

Model 3

```
void f()
{
    extern int x;
    x=6;
    printf("%d",x);
}
void main()
{
    extern int x;
    x=7;
    f();
    printf("%d",x);
}
int x=2;
```

we need to specify 'extern' for every function.
But instead of that we can specify 'extern' on the top for once as below:

```
extern int x;
void f()
{
}
void main()
{
}
int x=2;
```

Compilation :

Memory allocated for

x 2

Execution :

x=7 → f()

x=6

pf(x) → 6

return to main()

pf(x) → 6

∴ op: 6, 6

Model 4:

```
void main()
{
    extern int x;
    x=5;
    printf("%d", x);
}
```

→ This program will be executed without any compilation error.

→ But, during linking time compiler can't find physical memory for 'x'.

This type of error is called Linker Error.

Application of extern:

→ This generally used when size of code is big.

A big code is distributed among different files.

Eg:

t1.c

```
int x=10;
void f()
{
    print("ACE");
}
```

Compilation → obj

t2.c

```
void main()
{
    extern int x;
    x=5;
    f();
    printf("%d", x);
}
```

Compilation → obj

obj + obj → Linker → .exe

→ This is generally used in file handling.

21/05/20

Pointers

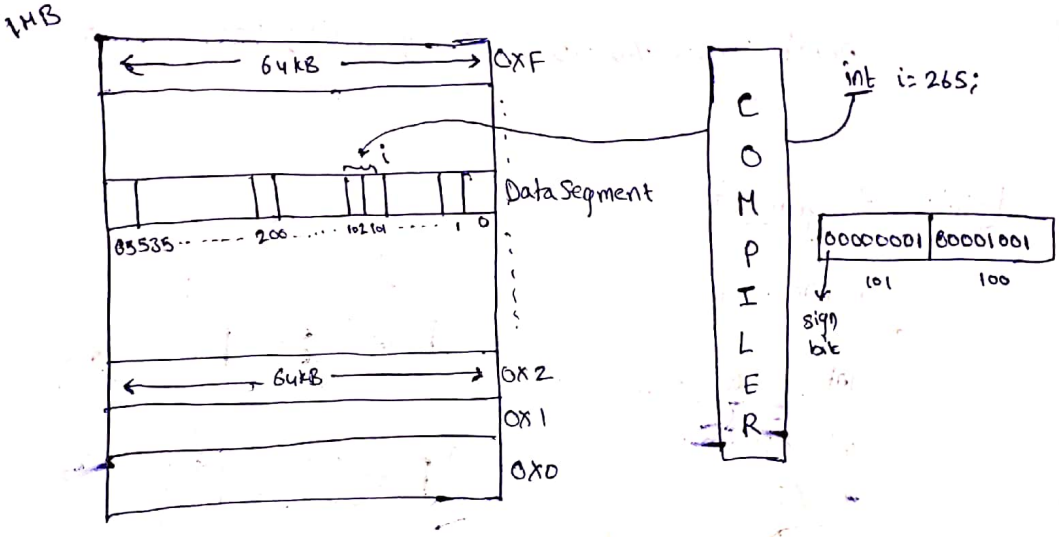
Basics of pointers:

Turbo C 3.0 is based on 8086 processor.

In this, every program has 1 MB size in RAM

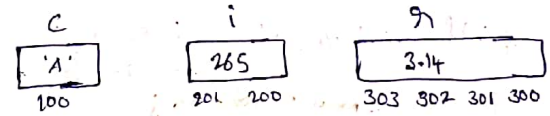
1MB ⟶ 16 segment (each of 64 kb)

One of those 16 segments is data segment



Consider

```
char c='A';
int i = 265;
float r=3.14;
```



→ To print the addresses of above variable we use format specifier %u and %p and %x also we need use &var_name

%u → address in decimal form

%x → address in hexadecimal form

Eg: printf("%u %u %u", &c, &i, &r) : 100, 200, 300

→ A pointer stores the starting byte of the address of a variable.

```
char *p; p=&c;
int *q; q=&i;
float *s; s=&r;
```

→ Now p is pointing to address of c.

Content of 'c' can be ~~acced~~ accessed, by '*p'.

Ex: printf("%c, %d, %f", *p, *q, *s); A, 265, 3.14

★

→ we can use %u and %x to print addresses of variables from pointer variables to.

printf("%u %u %u", p, q, s) : 100, 200, 300

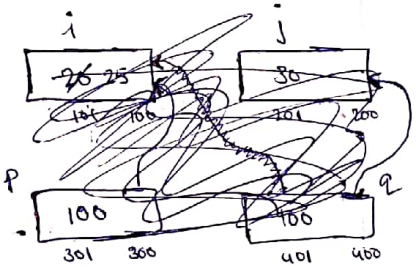
★

→ To get address of pointer variable

printf("%u %u %u", &p, &q, &s); 400, 500, 600

Pointer Assignment:

```
void main()
{
    int i=20, j=30;
    int *p, *q;
    p=&i;
    q=p;
    *q=25;
    printf("%d %d %d", i, *p, *q);
    p=&j;
    *p=35;
    printf("%d %d %d %d", i, *p, *q, j);
}
```



This is known as pointer Assignment

→ The content of p is assigned to q

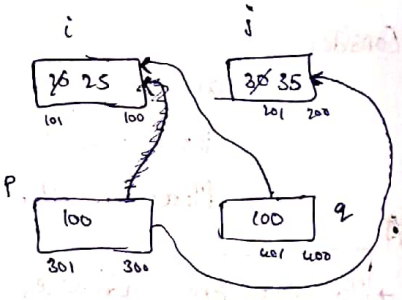
→ This modifies value in i

25 25 25

→ Now p points to j

→ modifies value of j

25 35 25 35



Passing address of a variable:

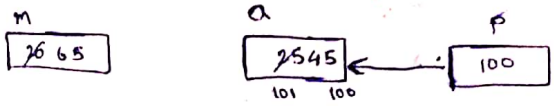
```
void f(int *p)
{
    int m=20;
    *p=*p+m;
    m=m+*p;
}
```

→ It holds address 100

*p= 25+20=45

m= 20+45=65

```
void main()
{
    int a=25;
    printf("%d", a);
    f(&a); → Here we are passing address. So to hold this address
    printf("%d", a);   function 'f' must have pointer variable as
}                      a formal parameter.
```



↓ 45

In above program, memory allocation for 'p' & 'm' is deleted as soon as "f" returns

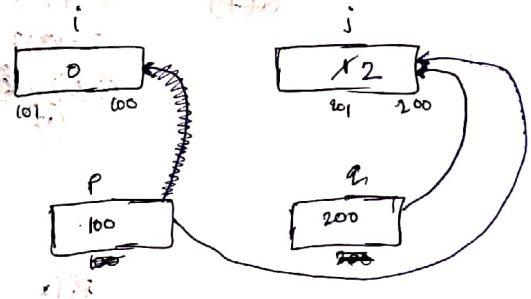
→ Note that in above program, we have modified content of variable 'a' ~~whose~~, whose scope is bound to main, in function 'f'.

(Q14) (GATE)

```
void f(int *p, int *q)
{
    p=q; → p also points to j now
    *p=2; → j=2
}
int i=0, j=1;
void main()
{
    f(&i, &j);
    printf("%d %d", i, j);
}
```

↓100 ↓200

↓ ↓
0 2



∴ o/p: 0, 2

(Q15)

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

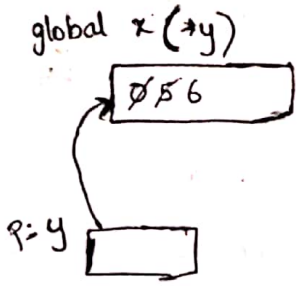
Note:

A pointer can't be assigned with an integer value.

However we can assign 0.

```
int *p=0;
int *p=NULL;
```
— same meaning

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



Sol:

main()

x=5 ⟶ P(&x)

P(int *y)

~~x=5+2~~

x

x=5+2

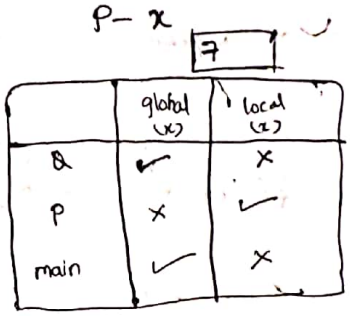
x=7 ——— Q(7)

~~Q(7)~~

z=7+5

z=12

pf(z) - (12)



return to P() ⟶ *y = x-1

*y = 7-1

= 6

pf(x) - (7)

pf(x) ⟵ return to main

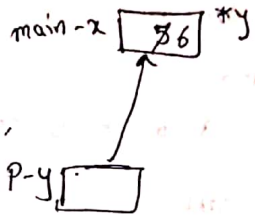
(6)

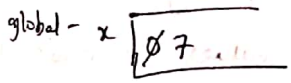
∴ O/P: 12, 7, 6

(Q16)

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

```
    printf("%d", x);
}
void main()
{
    int x=5;
    P(&x);
    printf("%d", x);
}
```





Sol:

main

x [5] ⟶ P(&5)

x = 7 ⟶ Q(7)

z = 7+7

z = 14 ⟶ return to P() ⟶ *y = 7-1 ⟶ main() ⟶ pf(x)

pf(z) (14)

=6

pf(x) (7)

(6)

O/P: 14, 7, 6

(Q17)

```
void f1(int *a, int *b)
{
    int c;
    c = *a;
    *a = *b;
    *b = c;
}
void f2(int a, int b)
{
    int c;
    c = a;
    a = b;
    b = c;
}
void main()
{
    int a=5, b=6; c=7;
    f1(&a, &b);
    f2(a, b);
    printf("%d", c-b-a);
}
```

Sol:

In 'f1' call

original values in main are swapped

∴ In main

a = 6 , b = 5 , c = 7

Now after 'f2' call

as we are passing variable

the values in main are not modified

c - b - a = 7 - 5 - 6

= -4

∴ O/P: -4

(Q18)

```
void mystery (int *ptra , int *ptrb)
{
        int *temp;
        temp = ptra;
        ptra = ptrb;
        ptrb = temp;
}
void main()
{
   int a=2020 , b=42, c=7 , d=6;
   mystery (&a,&b);
   if(b<d)
      mystery(&b,&d);
   mystery (&c, &d);
   printf("%d", a);
}
```

This is just swapping contentents of pointers. As soon as this function returns, the pointers are deleted. Hence this function has no impact on main().

∴ O/P: 2020

Q19 GATE

```
void f(int *p, int m)
{
    *p = *p + m;
}
void main()
{
    int i=20, j=30;
    f(&i, j);
    printf("%d", i+j);
}
```

observing this function we can say, the function modifies content of *p by adding value m to it.

f(a,b) means a=a+b

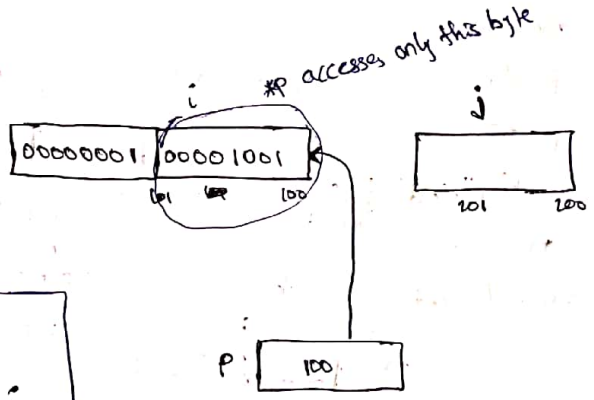
$\longrightarrow i = i+j = 50$

$\frac{i+j}{80}$

$\therefore$ O/P: 80

Q20

```
void main()
{
    int i=2650, j;
    char *p;
    p = &i;
    j = *p;
    printf("%d", j);
}
```



This type of assignment is valid.

But compiler generates a 'Suspecious pointer Conversion' warning message.

As the pointer type is character,

the pointer will access only one byte from 'i'. i.e., 100th byte

j = *p is executed as

$j = (00001001)_2 = (9)_{10}$

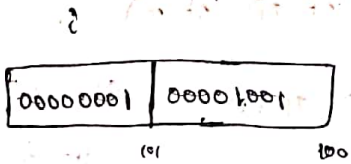
(sign bit)

so pf(j) — 9 → This has to be seen in 2's Complement form.

$\therefore$ O/P: 9

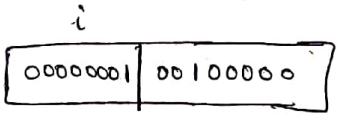
Q21

```
void main()
{
    int i=265;
    char *p;
    *p=32;
    printf("%d", i);
}
```



Now *p=32 modifies only 100th byte

∴



∴ i = 256+32 = 288

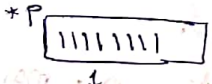
```
printf("%d", i);
        ↓
       288
```

∴ O/P: 288

Q22

```
void main()
{
    int i=255, j;
    char *p;
    p=&i;
    j=*p;
    printf("%d", j);
    *p=32;
    printf("%d", i);
}
```

p will access only 100th byte



This information is treated as character

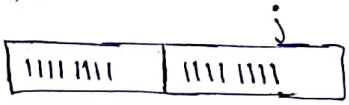
j = 11111111

sign bit (negative)

∴ consider 2's complement form

∴ j = -1

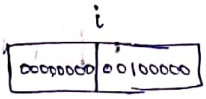
Now this value has to be stored in 2 bytes of j



(-1)

```
printf("%d", j);
        ↓
       -1
```

```
*p=32;
printf("%d", i);
        ↓
       32
```



i = 32

∴ O/P: -1, 32

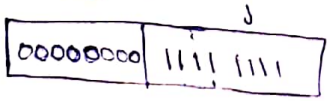
Q23

```
void main()
{
    int i=255, j;
unsigned char *p;
    p=&i;
    j=*p;
    printf("%d", j);
}
```

As it unsigned

$j = (11111111)_2$

= 255



∴ j = 255

∴ O/P: 255

Problems with pointers:

1. Dangling pointer problem.
2. Uninitialized pointer problem.
3. Memory leakage problem.
4. Null pointer dereference problem.

Q24 GATE 2017

```
int total(int v)
{
    static int count=0;
    while(v)
    {
        count += v&1;
        v>>=1;
    }
    return count;
}
void main()
{
    static int x=0;
    int i=5;
    for( ; i>0; i--)
        x = x + total(i);
    printf("%d", x);
}
```

options :

a) 23
b) 24
c) 25
d) 26

Sol:

The bitwise and of any even number with '1' = 0

" " " " " odd " " '1' = 1

Now

x = x + total(5)

count += 5&1 count = 0+1 = 1

v = 5/2 = 2

count += 2&1 count = 1+0 = 1

v = 2/2 = 1

count += 1 & 1 count = 1+1 = 2

v = 1/2 = 0

∴ return 2

In main()

x = 0+2

= 2

Now x = x + total(4)

(2)

for v = 4 count = 2+0 = 2

for v = 2 count = 2+0 = 2

for v = 1 count = 2+1 = 3

return 3

x = 2+3 = 5

x = 5 + total(3)

v = 3 – count = 3+1 = 4

v = 1 – count = 4+1 = 5

x = 5+5 = 10

x = 10 + total(2)

v = 2 – count = ~~2+0~~ 5+0 = 5

v = 1 – count = ~~5~~ 5+1 = 6

∴ x = 10+6 = 16

v = 1 – count = 6+1 = 7

x = 16+7 = 23

∴ opt (a)

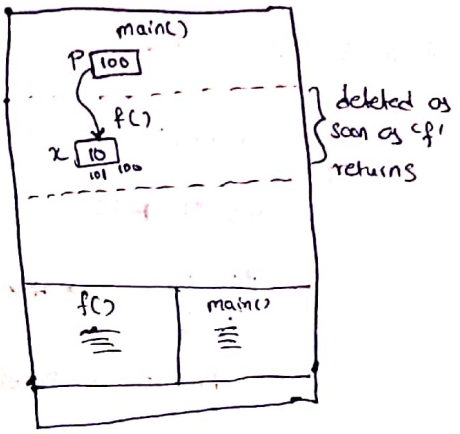
22/05/20

1. Dangling pointer problem:

Consider

```
int* f()
{
    int x=10;
    return &x;
}
```

```
void main()
{
    int *p;
    p=f();
    printf("%d",*p);
}
```



→ Now p is pointing to an object whose memory is deallocated. (This problem doesn't give any compile time error).

Overcoming the problem

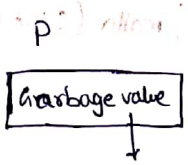
```
int* f()
{
    static int x=10;
    return &x;
}
void main()
{
    int *p;
    p=f();
    printf("%d",*p);
}
```

→ Now this memory is not deallocated when function return

↳ 10

2. Uninitialized Pointer problem:

```
void main()
{
    int *p;
    *p=10;
    printf("%d",*p);
}
```



This address may point to O.S code or processes. Updating these values creates error in system & system may crash.

i.e. Illegal Access.

3. Null Pointer Dereference:

```
void main()
{
    int *p = NULL;
    *p = 10;
}
```

p → X

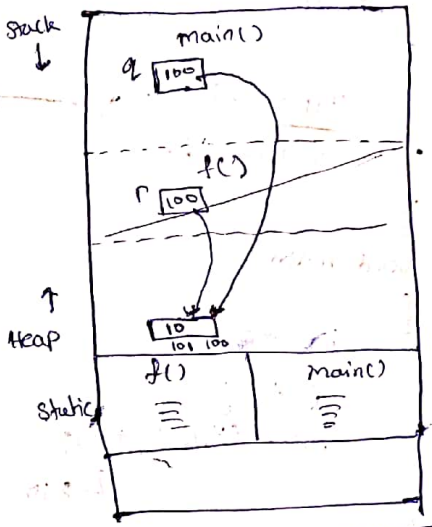
Here we are trying to store value in NULL (no memory)

Null pointer Dereference

4: Memory Leakage Problem:

```
int* f()
{
    int *p = (int*) malloc(sizeof(int));
    *p = 10;
    return p;
}
void main()
{
    int *q;
    q = f();
    printf("%d", *q);
}
```

(10)



Now consider the program

```
int* f()
{
    int *p = (int*) malloc(sizeof(int));
    *p = 10;
    return p;
}
void main()
{
    int *q, i;
    for(i=1; i<=10; i++)
    {
        q = f();
    }
    printf("%d", *q);
}
```

when malloc is used
default value is garbage value
when calloc is used
default value is 0
Heap memory has program lifetime

Here by the time i=10

9x2 bytes before have no reference, but exist in heap.

This problem is called Memory Leakage Problem.

overcoming the memory leakage problem:

```
int *f()
{
    ==
}
void main()
{
    int *q, i;
    for(i=1; i<=10; i++)
    {
        q=f();
        printf("%d", *q);
        free(q);
    }
}
```

☆ ☆

Q25 ☆ GATE 2017

```
int * assignval(int* x, int val)
{
    *x = val;
    return x;
}
int main()
{
    int *x = malloc(sizeof(int));
    if (NULL==x) return;
    x = assignvalue(x,0);
    if(x)
    {
        x= (int*) malloc(sizeof(int));
        if (NULL==x) return;
        x= assignvalue(x,10);
    }
```

```
    printf("%d\n",*x);
    free(x);
}
```

The code suffers from which one of the following problems:

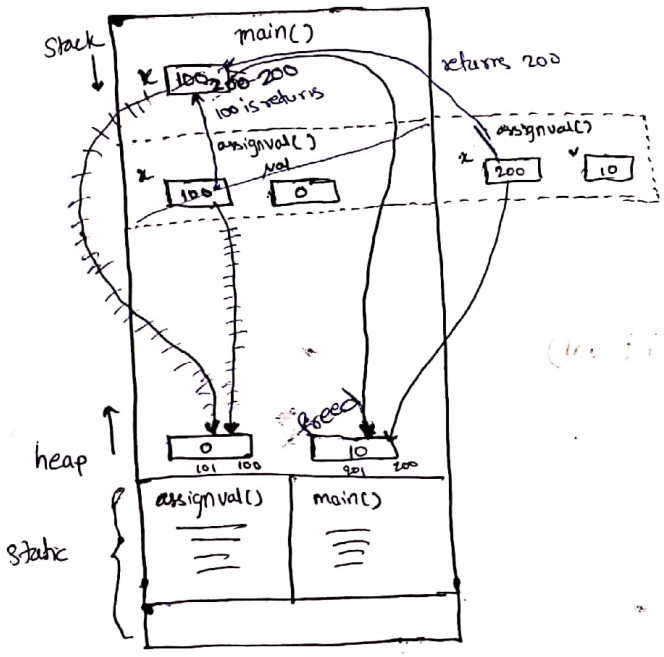
a) Compiler error as the return of malloc is not typecasted appropriately.

b) Compiler error because the comparision should be made as x=NULL and not as shown.

c) Compiles successfully but execution may result in dangling pointer.

d) Compiles successfully but execution may result in memory leak

Sol:



malloc doesn't need explicit typecasting. It performs implicit typecasting

NULL==x & x==NULL both are same

'return x' in assignvalues returns address.

This doesn't cause dangling pointer problem

In 'if(x)'

↓

x contains address which is non-zero

∴ 'if' block executes

After 2nd malloc, x now points to 2nd block created (200) and it no more points to previous one (100)

printf(*x) — 10

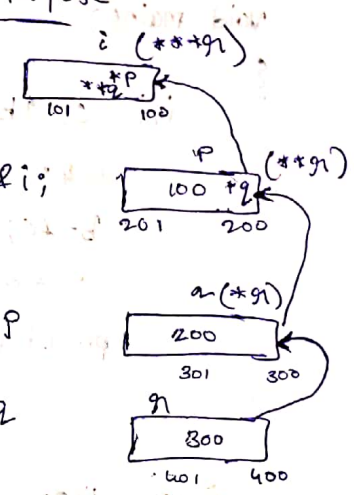
free(x) — frees 200 location

But the first create memory (100th byte) has no reference and not used in current ~~set~~ situation..

∴ opt (d)

Pointer to Pointer:

Declaration	Datatype	Purpose
int i;	int	
int *p	int*	p = &i;
int **q	int**	q = &p
int ***r	int ***	r = &q



Here ~~both~~ content of p & *q is same

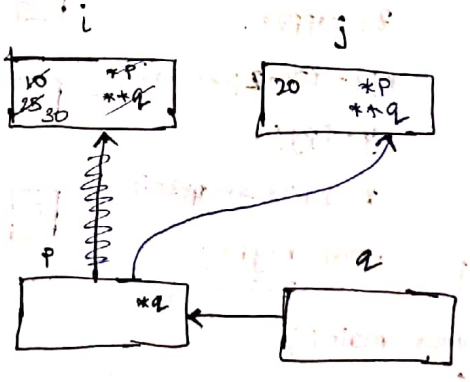
So p & *q refer to i.

∴ *p & **q ~~refer to~~ is content of i.

Same explanation goes for r, *r, **r, ***r ;

Q26

```
void main()
{
    int i=10, j=20;
    int *p, **q;
    p=&i;
    q=&p;
    **q=25;
    *p=**q+5;
    printf("%d %d", i, j);
                  30 20
    *q=&j;
    *p=50;
    Printf("%d %d", i, j);
}
```



→ This statement can be seen as p=&j;

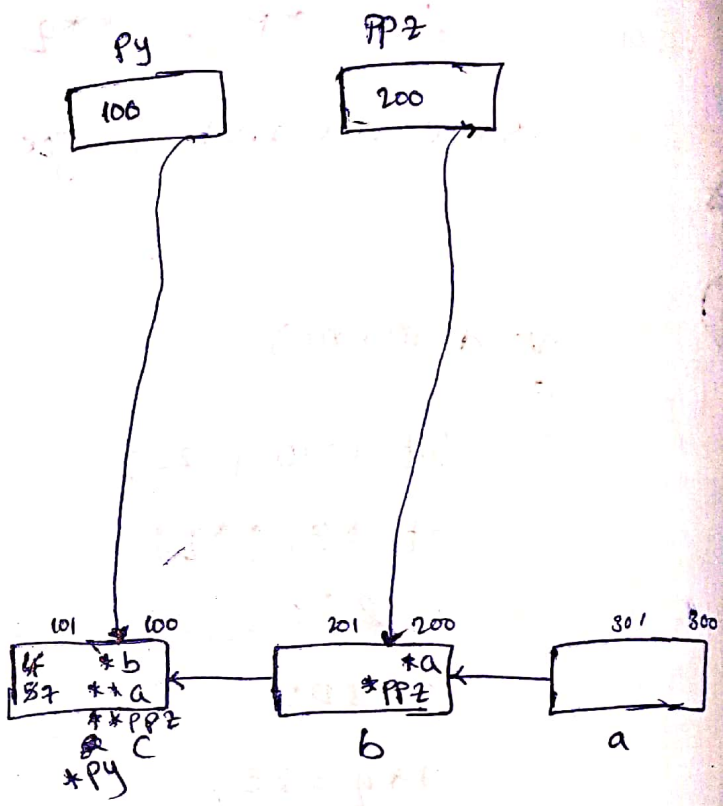
→ value of j will be modified

}

a) 19 b) 20 c) 21 d) 22

Q27) int f(int x, int *py, int **ppz)
{
int y,z;
**ppz = **ppz + 1; → c=4+1=5
z = **ppz;
*py = *py + z; → c=5+2=7
y = *py;
x = x+3; → x=4+3=7
return x+y+z;
7+7+5=19
}
void main()
{
int c, *b, **a;
c=4;
b=&c;
a=&b;
printf("%d", f(c,b,a));
}

x 7
y 7
z 5

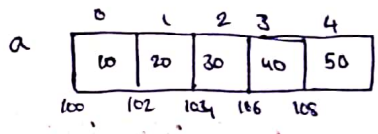


a) 19 b) 20 c) 21 d) 22

23/05/20

Pointer to array element:

int a[5] = {10, 20, 30, 40, 50}



→ Array name contains base address of the array.

printf("%u", a) : 100

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

→ Subscript Rule : a[i] = *(a+i) = *(i+a) = i[a]

Eg: a[3] = *(a+3)

= *(100 + 3*sizeof(int)) → Size of data residing in 100th address.

= *(100+6)

= *(106) = 40

★ → printf("%d %d %d %d", a[3], *(a+3), *(3+a), 3[a]);

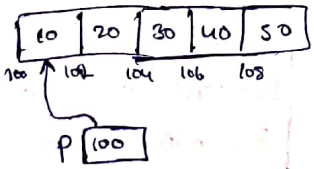
40 40 40 40

→ Declaration of pointer to array element:

int a[5] = {10,20,30,40,50};

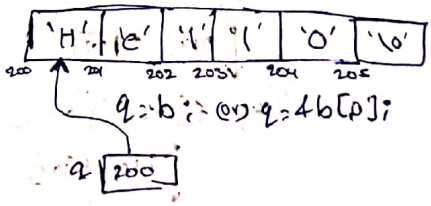
int *p;

p = a; (or) p = &a[0];



char b[6] = {'H', 'e', 'l', 'l', 'o', '\0'}

char *q;



★ ★ ★ a = a+1; (a = 100+1*2)
a = 102

↓

This is not possible

Array pointer 'a' is a constant pointer

~~Array pointer 'a'~~ is ~~always~~ a compiletime error (L-value required)

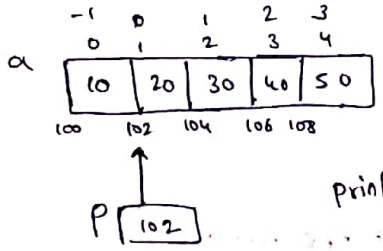
So ~~must~~ we must use another variable to modify address of a.

p = a+1; → This is value

→ Consider p = a+1

= 100+2

= 102



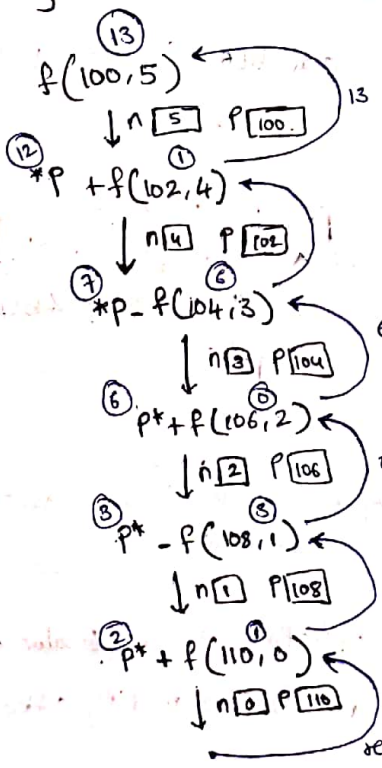
printf("%d %d %d", P[-1], P[0], P[3]);

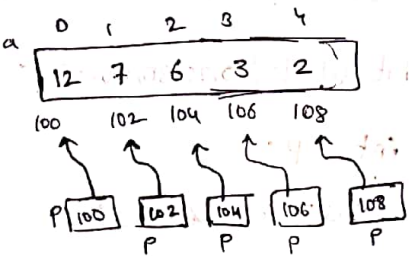
↓ ↓ ↓

10 20 50

(Q28)

```
int f(int *P, int n)
{
    if(n<=0)
        return 1;
    if(*P%2==0)
        return *P+f(P+1, n-1);
    else
        return *P-f(P+1, n-1);
}
void main()
{
    int a[5]={12,7,6,3,2};
    printf("%d", f(a,5));
}
```





Note that function call have higher precedence

∴ o/p: 13

Q29 GATE

```
void main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10,11};
    int *ip = a+5;
    printf("%d", ip[3]);
}
```

Sol:

$ip[3] = *(ip+3)$

$= *(a+5+3) = *(a+8) = a[8] = 9$

Q30 GATE 2017

Assume that base address of array is 2000 and integer occupies 4 bytes. Then what is the o/p of the following program?

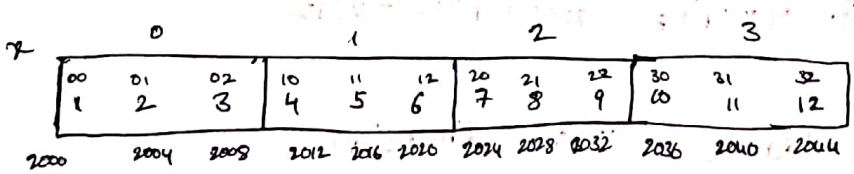
```
void main()
{
    unsigned int x[4][3] = { {1,2,3}, {4,5,6}, {7,8,9},
                             {10,11,12} };
    printf("%u %u %u", x+3, *(x+3), *(x+2)+3);
}
```

a) 2036, 2036, 2036

b) 2012, 4, 2204

c) 2036, 10, 10

d) 2012, 4, 6



Here size of each one dimension is 3*12

$x+3 = 2000 + 3*12 = 2036$

$*(x+3) = 2000 + 3*12 = 2036$

$*(x+2)+3 = (2000 + 2*12) + 3*4 = 2024 + 12 = 2036$

$\therefore$ opt (a)

2D Arrays:

Consider 1D array

```
int a[5];
```

*(a+i) ≡ value at ith index

a+i ≡ address of ith index.

In 1D array,

if it has one *, it is value

if it has no *, it is address

For 2D array

```
int x[5][4];
```

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

In 2D array

if it has two *, it is value

if it has one * it is address

if it has no * it is address

Note:

1) *(*(x+i)+j) ≡ value at i^{th} one-dimension & j^{th} index

i.e., x[i][j]

Eg: *(*(x+1)+2) ≡ goto 1st one dimension & access value at 2nd index

i.e. x[1][2]

2) *(x+i)+j ≡ Address of i^{th} one dimension & j^{th} index.

*(x+1)+2 ≡ (2000 + 1*12) + 2*4

= 2020

(size of each one dimen)

These values are from Q30

3) *(x+i) ≡ Address of i^{th} one dimension and 0th index

Eg: *(x+1) ≡ 2000 + 1*12 = 2012

4) x+i ≡ Address of ith one dimension

Eg: x+1 ≡ 2000 + 1*12 = 2012

5) x ≡ Address of array

Note:

In Q (30), consider

$*(x+1)+4 \equiv *(*2000+12)+4*4$

$= 2028 \equiv \&x[2][1]$

$*(x+1)+6 \equiv (2000+12)+6*4$

$= 2036 \equiv \&x[3][0]$

Pointer to one-dimensional array:

Here we declare a pointer which points to an array but not array element.

Syntax: `int (*p)[n];`

↓

It is pointer which points to an array with n elements.

Eg:

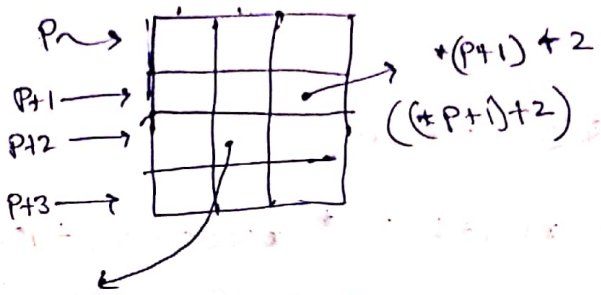
```
void main()
{
    unsigned int x[4][3] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
    int (*p)[3];
    p = x; //(or) p = &x[0]; Here 'p' is pointing to 1st one dimension
                              Now wherever there is x, we can replace it
                              with p

    printf("%u %u %u", p+3, *(p+3), *(p+2)+3);
                       ↓      ↓      ↓
                      2036   2036   2036
}
```

Here

`p = p+1` → size of one dimensional array (3*sizeof(int))

Mem 2000 + 3*4 = 2012

i.e., incrementing will make it point to next array



*(P+2)+1 — address

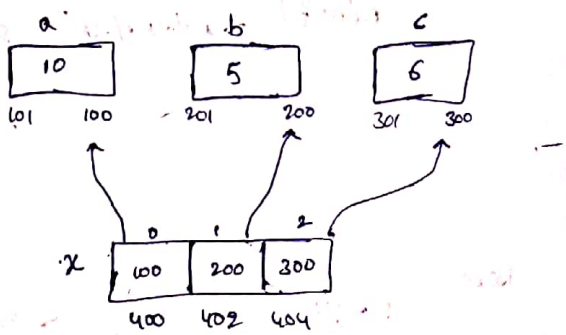
((P+2)+1) — value

	Declaration of pointer	Meaning of P=P+1
int a[i]	int *p p=a;	p=p+1 means p will be pointing to next element of that array i.e., element at index 1
int x[i][j]	int (*p)[j] p=x; ↓ here	[ ][ ]p=p+1 mean p is pointing to next one dimensional array i.e., array x[i][j] ↑ *j
int b[i][j][k]	int (*p)[j][k]; p=b; ↑ Here p is pointer to 2-dimensional array	p=p+1 means p is pointing to next two dimensional array i.e., 2D array x[i][j][k] ↑ ↑ *j *k

Array of Pointers:

Consider

```
void main()
{
    int a=10; b=5; c=60, i;
    int *x[3];
    x[0]=&a;
    x[1]=&b;
    x[2]=&c;
```



```
for(i=0; i<3; i++)
    printf("%d", *x[i]);
}
```

o/p: 10, 5, 6

→ This how *x[i] is evaluate

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

* (address) =

ex: *x[0]
↓
* *(x+0)
* (100) = 10

*x[2]
* *(x+2)
* *(400+2*2)
* *(404)
* (300) = 6

Arithematic operations on pointers:

1) Consider int *P1, *P2;

Invalid operation:

1) P1 + P2
2) P1 * P2
3) P1 / P2
4) P1 % P2

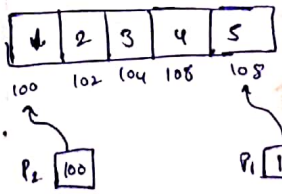
valid operations:

1) P1 + 1
2) P1 - P2 (in some cases it is valid)
3) Comparision operator like <, <=, >, >=, ==, != can be made
4) P1 = 0 (only zero can be assinged) (or) P1 = NULL

we use this when P1 & P2 point to same array. It is valid to use in other cases too. But those cases serve no purpose

Consider

```
int a[5] = {1,2,3,4,5};
int *P1, *P2;
P1 = &a[4];
P2 = &a[0];
printf("%d", P1 - P2);
```



$$P_1 - P_2 = \frac{\text{content}(P_1) - \text{content}(P_2)}{\text{sizeof(datatype)}} = \frac{108-100}{2} = 4$$

only in this context, we can find the difference.

Q31 GATE

```
int main()
{
    static int a[] = {10,20,30,40,50};
    static int *p[] = {a, a+3, a+4, a+1, a+2};
    int **ptr = p;
    ptr++;
    printf("%d %d", ptr-p, **ptr);
}
```

↳ here '*' is not req because p is array.

Sol:

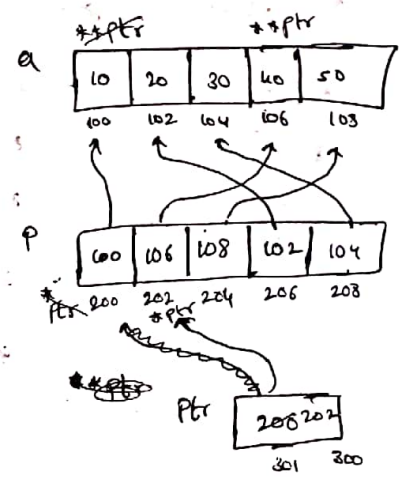
~~ptr = p~~

~~200 = 200 = 0~~

ptr++ → *ptr = 202

~~**ptr = *(*ptr) = *(106) = 10~~

~~∴ o/p: 2, 10~~



ptr - p = 202 - 200 = 2

**ptr = *(*ptr)

= *(106) = 40

∴ o/p: 2, 40

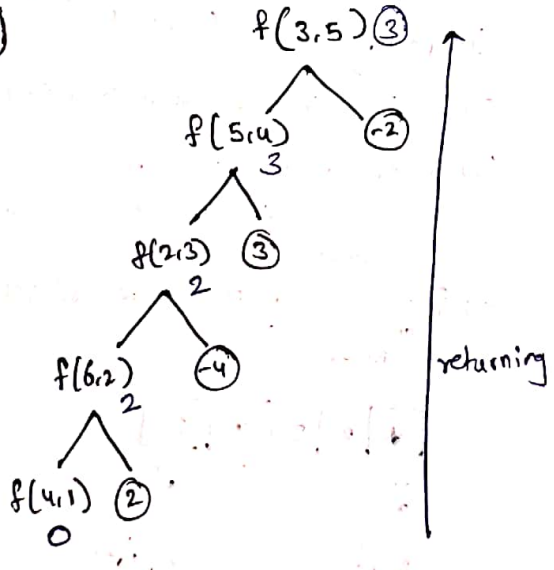
Note:

++, -- can't be applied on constants or expression.

Eg:

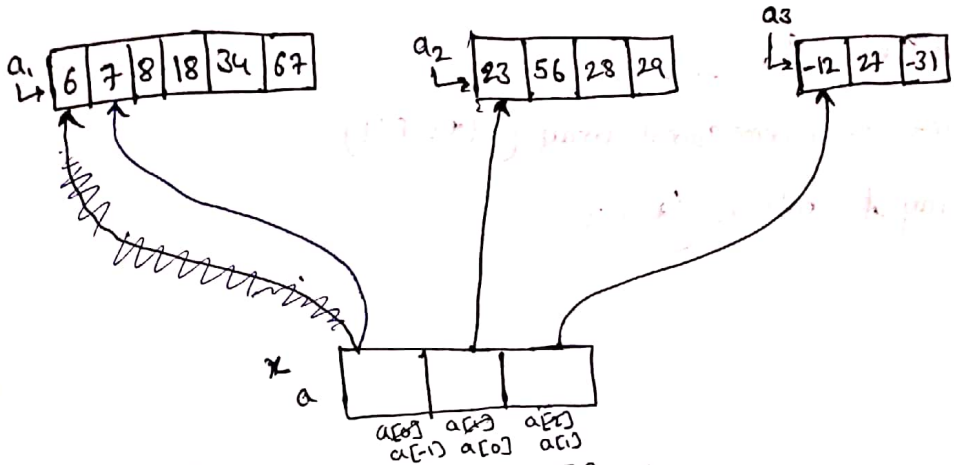
```
int i = 50, j = 6;
++i;      → valid
++(i+j);  → not valid
++5;      → not valid
```

P/49



∴ 3

P155



$x[0] = a_1 \quad x[1] = a_2 \quad x[2] = a_3$

when print is call

x is copied into int *a[]

$a[0] = a_1 \quad a[1] = a_2 \quad a[2] = a_3$

→ $a[0][2] = a_1[2] = 8$

→ $*a[2] = *a_3 = a_3[0] = -12$

→ $*++a[0]$

*, ++ have same precedence

associativity is right to left

$\therefore *(++a[0])$ (also now a[0] points to 2nd element of $a_1[]$)

$= *(a_1+1) = 7$

→ $*(++a)[0]$

Here a is increament

now a will point to previous a[1]

now $*a[0]$ (This is equal to previous $*a[1]$)

$*a_2 = 23$

→ $a[-1][1]$

$a_1[1]$

a[-1] points to 2nd element of $a_1[]$

$= 8$

$\therefore$ o/p : 8, -12, 7, 23, 8

Note:

we can declare a two dimensional array in 3 ways:

(i) direct way (a[][])

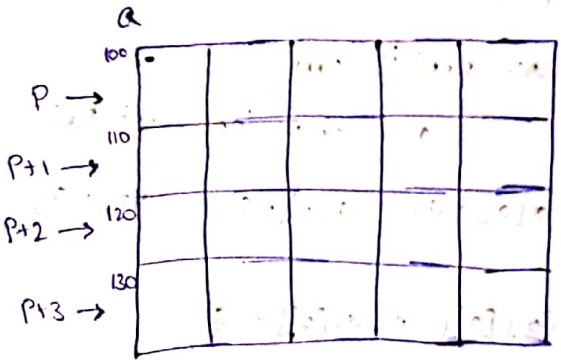
(ii) pointer to 1-dimensional array ((*a)[])

(iii) array of pointers (*a[])

Eg:

Consider

```
int a[4][5];
```



Now consider

```
int (*p)[5];
p=a;
```

if 100 is base address

$P+2 = 100 + 2*10$ $(5*sizeof(int)=10)$

$= 120.$

$a[2][4] = *((*(P+2)+4))$

Here P can be assigned as

P=a, P=a+1 ...

P=&a[0], P=&a[1] ...

Now consider

```
int *p[4];
```

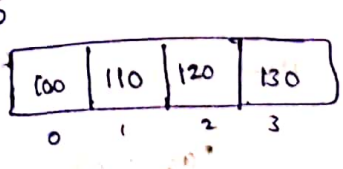
⟶ Here P[0], P[1], P[2], P[3] are treated as normal pointers like *P.

P[0] = a[0] (or) P[0] = *a

P[1] = a[1] or P[0] = *(a+1)

P[2] = a[2] or P[2] = *(a+2)

P[3] = a[3] or P[3] = *(a+3)



$a[2][4] = *(P[2]+4) = P[2][4]$

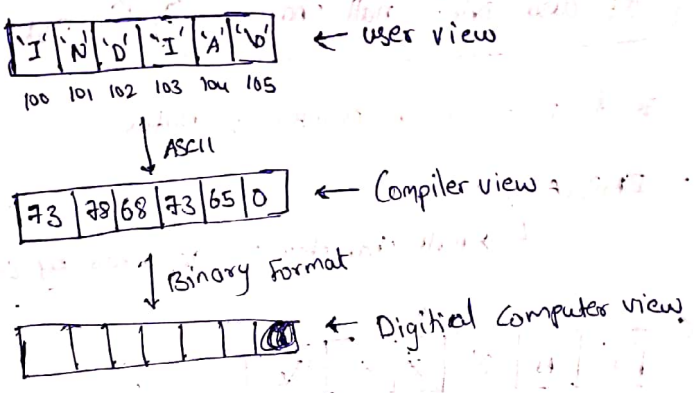
25/05/20

Pointer to string:

string constant : group of characters enclosed in double quotes

Eg: "Apple", "a", " " (space)

Consider "INDIA"



1) char c[6] = {'I', 'N', 'D', 'I', 'A', '\0'};

2) char c[6] = "INDIA";

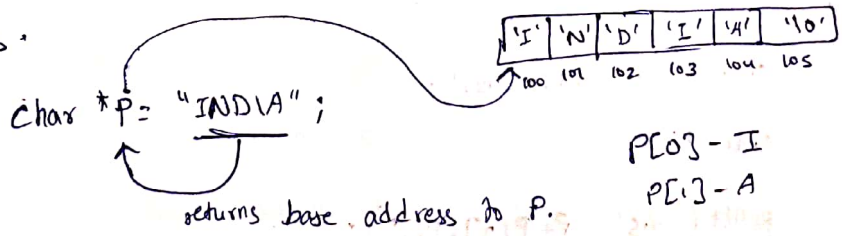
3) char c[6] = {73, 78, 68, 73, 65, 0}

4) Char *p = "INDIA";

~~In ① size of~~

Note:

→ String Constant is a pointer and it always returns its base address.



→ String constant is stored in static memory at compile time

printf("%c", P[3]) : I

→ P[3]='k'; ⟶ This is not possible.
Segmentation fault error
We can't modify content of string constant

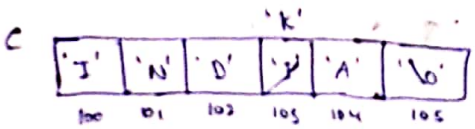
→ printf("%s", P) : INDIA
↳ I
↳ 100

from address '100' %s prints character by character until it encounters null character.

It also prints null character ★ But we can't see it.

→ Here size of P = size of pointer = 2 bytes

* char C[6] = "INDIA";
↳ null character is appended by compiler



C[0] : I
C[4] : A

→ printf("%c", C[3]) : I

→ ~~P~~ C[3] = 'k'; → Here it is valid

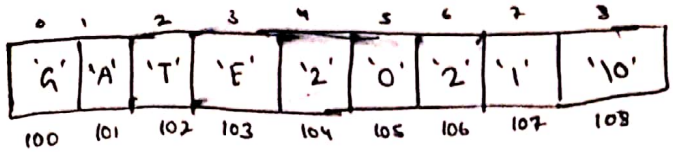
→ printf("%s", C) : INDkA

→ Its storage ~~area~~ area is like ~~Normal~~ other variables.

→ Here size of C = 6 bytes

(Q32)

```
void main()
{
    char C[] = "GATE2021";
    char *P = C;
    printf("%s", P + P[3] - P[1]);
}
```



p+ p[3] - p[1]

100 + 69 65

100 + 69 * sizeof(char)

169 - 65

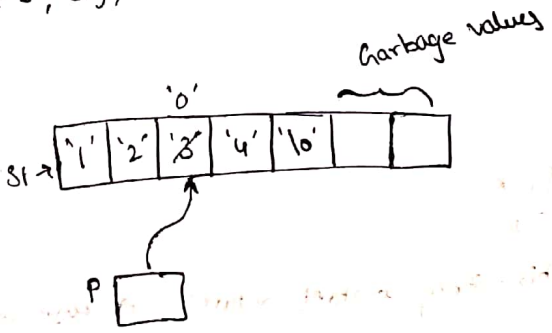
169 - 65 * 1 = 164

printf("%s", 164);

o/p: 2021

(Q33)

```
void main()
{
    char s1[7] = "1234";
    char *p = s1+2;
      *p = '0'; // line1
    printf("%s", s1);
}
```



pf(s1) → 1204

(Q34) In (Q33) if 'line1' is written as

*p = 0; or *p = '\0'

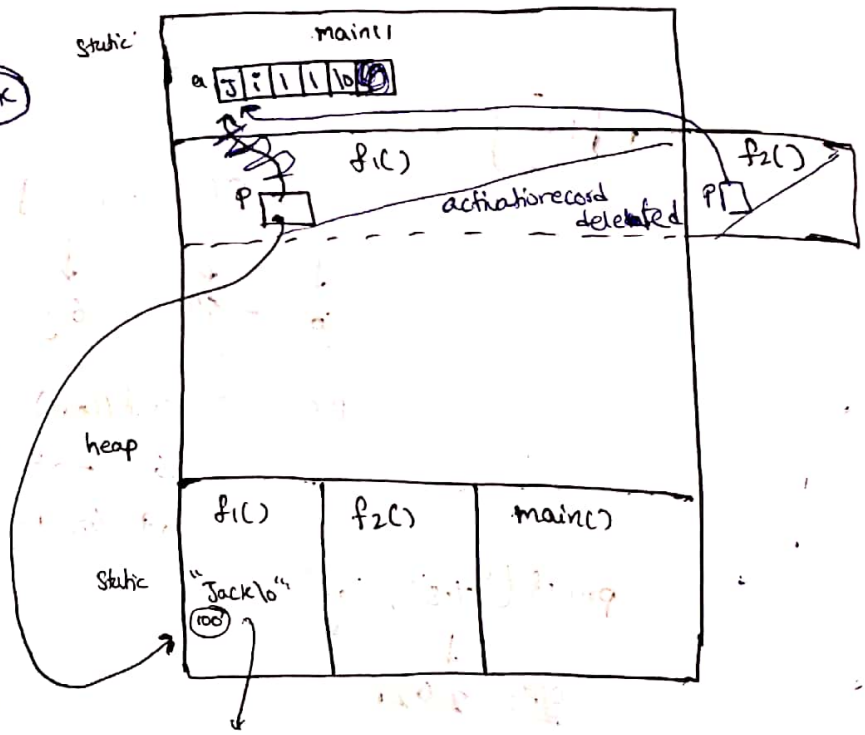
then what will the o/p be?

pf(s1) → 12

∴ o/p: 12

Q35

```
void f1(char *p)
{
    p="Jack";
    puts(p);   // 100 → Jack
}
void f2(char *p)
{
    p[0]='J';
    p[1]='i';
    p[2]='l';
    p[3]='l';
    p[4]='\0';
    puts(p);   // 200 → Jill
}
void main()
{
    char a[5];
    f1(a);
    puts(a);
    f2(a);
    puts(a);
}
```



Now this string constants has only read permision but not write permission.

garbage values

In f1

p="Jack"

This string constant returns its base address (100)

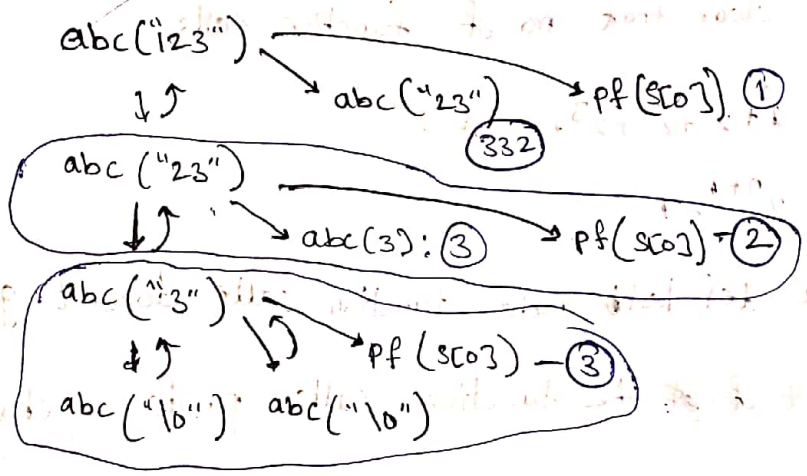
Now 100 is copied into p and now p will be pointing to "Jack\0" in static memory

∴ O/P: Jack, garbage, Jill, Jill

Q36

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

↳ base address of string constant is passed.



Here abc("3") producing o/p: 3

o/p of abc("23") : 332

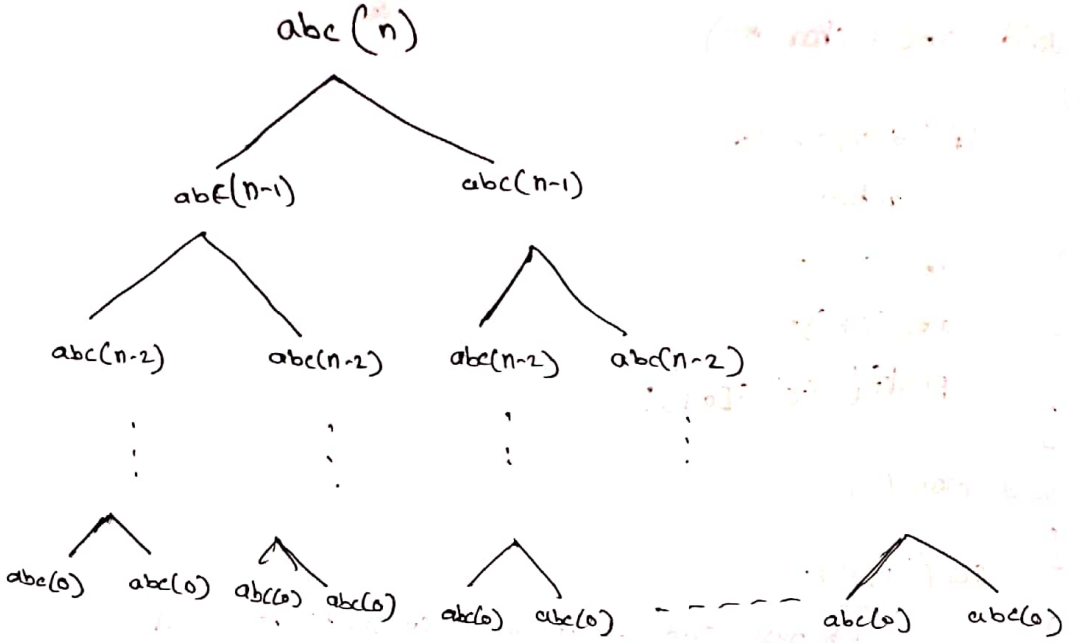
∴ o/p : 3323321

Q37 In Q36 if abc(str); is called where str is a string of length 'n'.

a) How many characters are printed?

c) How many no of time abc() is called?

Sol:



It is ~~was~~ clear that no of function calls.

$= 2^0 + 2 + 2^2 + 2^3 + \cdots + 2^n$

$= 2^{n+1} - 1$

Here the ~~left~~ leaf node function calls doesn't given any o/p

~~&~~ rest of ~~&~~ the function calls prints 1 character each.

$\therefore$ no of characters printed $= 2^{n+1} - 2^n - 1$

$= 2^n(2-1) - 1$

$= 2^n - 1$

Q.38 (GATE)

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

if the input string is "ABCD EFGH"

~~"ABCDEFGH"~~ then what is the output?

*p

P/50

3) *P[-2]

*(P-2)

506

400

405

ase

4) P[-1]

*(P-1)

*(502)

504 [-1

*(504-1)

*(502)

200 +

201

Sol:

*p && *p != ' ' is satisfied

until a null character or space is met

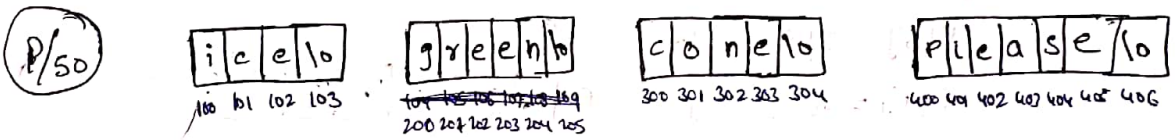
So after ABCD ↑

space will not satisfy the condition

f("ABCD EFGH") (A)
↓ ↑ ↑
f("BCD EFGH") (B)
↓ ↑ ↑
f("CD EFGH") (C)
↓ ↑ ↑
f("D EFGH") (D)
↓ ↑
f(" EFGH")

if condition not satisfied

∴ O/P: DCBA



3) *P[-2] +3

*(P-2)

506

400 + 3*1

403

case

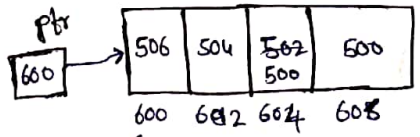
7) **++p

600

602

504

800 → Now "%s" takes 300 and prints cone



4) P[-1][-1]+1

*(P-1)

*(602)

504[-1]

*(504-1)

*(502)

200 + 1

= 201

reen

∴ opt (a)

2) * -- * ++p + 3

602

604

502

500

100 + 3

100 + (3*1)

103

∴ Null character is printed

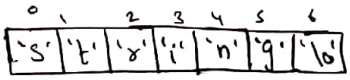
26/05/20

Q39

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

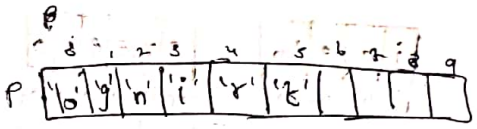
strlen() accepts a string pointer. and returns an unsigned integer.

a) gnirts b) nirts c) string d) prints nothing



length = 6

P[0] = S[6] = '\0'
P[1] = S[5] = 'g'
P[2] = S[4] = 'n'
P[3] = S[3] = 'i'
P[4] = S[2] = 'r'
P[5] = S[1] = 't'



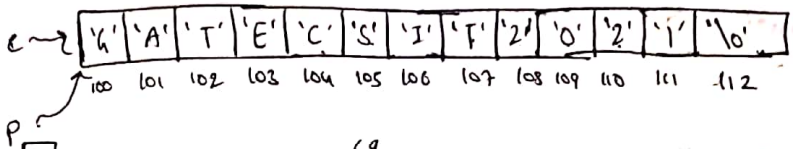
∴ pf(P) prints nothing as it starts with null character

∴ option (d)

Q40

```
void main()
{
    char C[] = "GATECSIT2021";
    char *P = C;
    printf("%d", (int) strlen(P+P[3]-P[1]));
}
```

Sol:



P → 100

P[3] = 'E' (69)

P[1] = 'A' (65)

P + P[3] - P[1]

~~100 + 101~~ 100 + 69 * 1 - 65 * 1

100 + 4 = 104

```
pf("%d", (int) strlen(104));
```

(8)

Here typecast is done because return type of strlen() is unsigned integer.

Q4)

```
void f(char *, char*)
void main()
{
    char *p = "abc";
    char *q = "1234";
    f(p,q);
}
void f(char *x, char *y)
{
    unsigned int c=0;
    int length;
    length = ((strlen(x) - strlen(y)) > c) ? strlen(x) : strlen(y);
    printf("%d", length);
}
```

Sol:

(strlen(x) − strlen(y)) > c

3 (−1) 4 (0)

When there is expression with signed and unsigned integer, signed integer is converted into unsigned integer

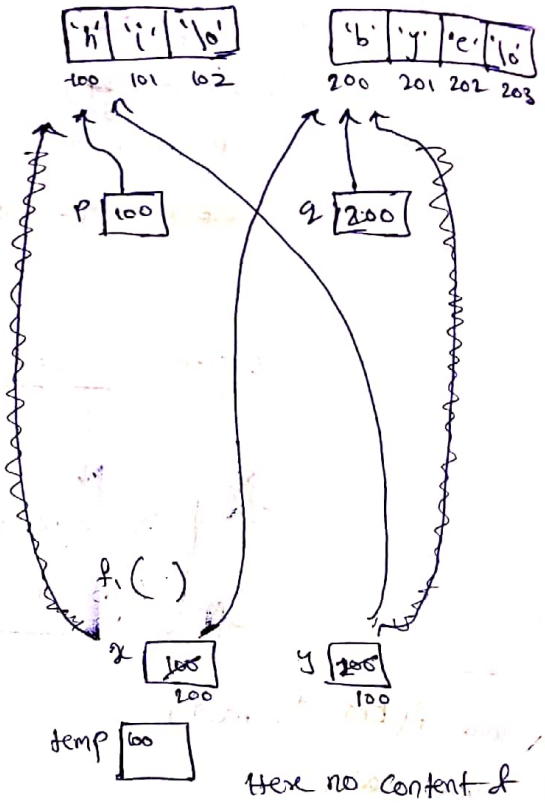
where '-1' is integer type and '0' (c) is unsigned integer

-1 when converted to unsigned will be greater than 0.

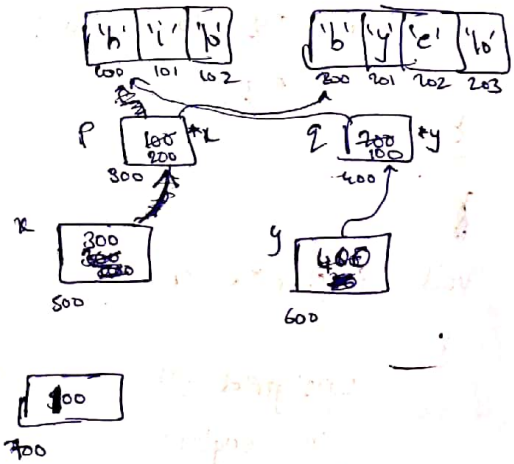
∴ length = 3 O/P: 3

Q42

```
void f1(char*, char*);
void f2(char**, char**);
void main()
{
    char *p="hi";
    char *q="bye";
    f(p,q);
    printf("%s %s", p,q);
    f1(p,q);
    printf("%s %s", p,q);
    f2(&p,&q);
    printf("%s %s", p,q);
}
void f1(char *x, char* y)
{
    char* temp;
    temp=x;
    x=y;
    y=temp;
}
void f2(char** x, char** y)
{
    char *temp;
    temp=*x;
    *x=*y;
    *y=temp;
}
```



Here no content of p & q is altered



Here in f2 content of p & q is modified

∴ op: hi bye
hi bye
bye hi

Pointer to Structures:

Basics of structures:

(i) Declaration of structure:

```
struct student
{
    int idno; (2 bytes)
    char s[5]; (5 bytes)
};
```

Here compiler knows abt layout of structure

Another way of defining:

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

(ii) Declaration of structure variable:

```
struct student s1; → allocates 7 bytes
```

(iii) Initialization of structure variable:

```
struct student s1 = {10, "sai"};
struct student s2 = {"sai", 10}; → This is wrong way of initialization
```

Invalid structure

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

s1

10	's' 'a' 'i' '\0'
idno	s

~~struct student *p;~~

~~p = &s1;~~

(iv) declaring pointer:

```
struct student *p;
p = &s1;
```

(v) Accessing member of structures:

using variable;

```
s1.idno;
s1.s;
```

using pointer;

```
p→idno;
p→s
```

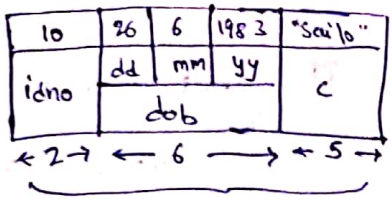
another way using pointer:

```
(*p).idno;
(*p).s;
```

p.idno; → invalid way of accessing because '.' operator has high precedence than ''

(vi) Nesting of structures:

```
Struct date
{
    int dd,mm,yy;
};
Struct student
{
    int idno;
    Struct date dob;
    Char c[5];
} sr = {10, 26, 06, 1983, "sai"};
```

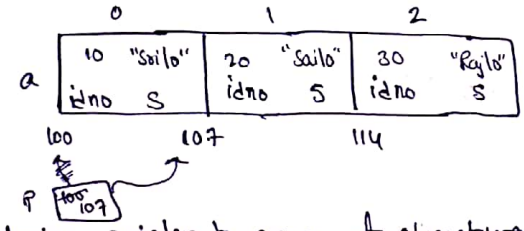


13 byte

```
Sr.idno;
Sr.dob.dd;
Sr.dob.mm;
Sr.dob.yy;
Sr.c;
```

(vii) Array of structures:

```
Struct student
{
    int idno;
    char s[5];
};
Struct student a[3] = { {10, "sri"}, {20, "sai"}, {30, "raj"} }
```



declaring pointer to array of structures

```
Struct student *p = a;
```

p = p+1 means

= p+(1* size of student structure)

= p+(1*7)

= 100+7

= 107

Now p points element a[1]

Accessing arrays:

```
a[1].idno; /*20*/
a[1].s; /*sai*/
p->idno; /*20*/
p->s; /*sai*/
p[0].idno; /*20*/
p[0].s; /*sai*/
```

Q43

```
void main()
{
    struct a
    {
        char ch[7];
        char *str;
    };
    struct b
    {
        char *c;
        struct a ss1;
    };
    struct b s1={"Raipur", "kanpur", "Jaipur"};
    printf("%s%s", s1.c, ++s1.ss1.str);
    printf("%s%s", ++s1.c, ++s1.ss1.str);
}
```

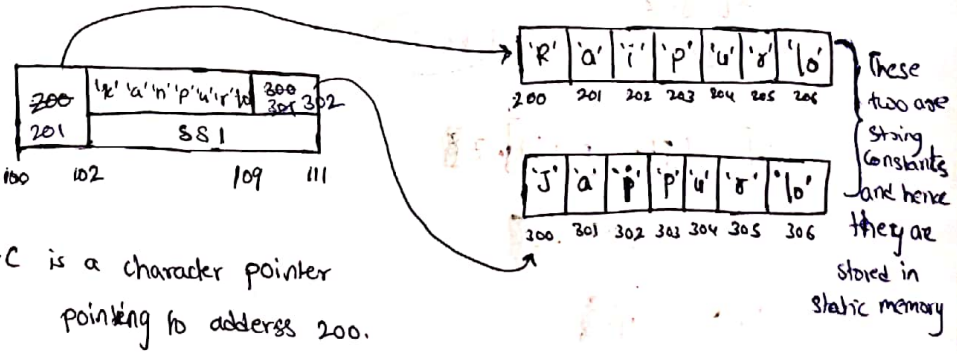
Raipur → s1.c

aipur → ++s1.ss1.str

aipur → ++s1.c

ipur → ++s1.ss1.str

Sol:



→ s1.c is a character pointer pointing to address 200.

→ ++(s1.ss1.str) here we are increamenting str, so it now points to 302

→ ++s1.c, now c points to 201

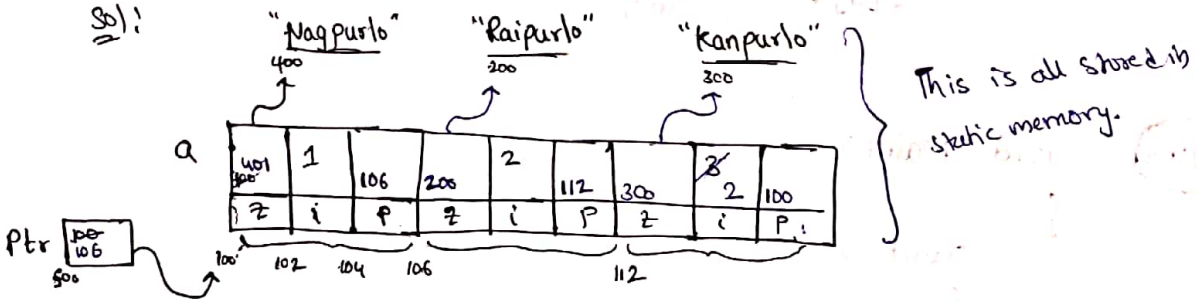
→ ++s1.ss1.str, now str points to 302

∴ o/p: Raipur
aipur
aipur
ipur.

P/52 Find o/p for

```
printf("\n %s %s %s", a[0].z, ptr->z, a[2].p->z);
printf("%s", ++(ptr->z));
printf("%s", a[(++ptr)->i].z);
printf("%s", a[--(ptr->p->i)].z);
```

Sol:



→ a[0].z = 400 pf("%s", 400) : Nagpur

→ ptr->z = 400 pf("%s", 400) : Nagpur

→ a[2].p->z = 100->z pf("%s", 400) : Nagpur

→ ++(ptr->z) = ++(100->z)
= ++(400)
= 401 : agpur

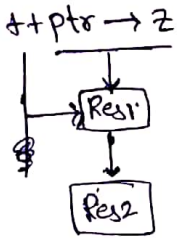
→ a[(++ptr)->i].z
a[(++100)->i].z
a[106->i].z
a[2].z
300 —— kanpur

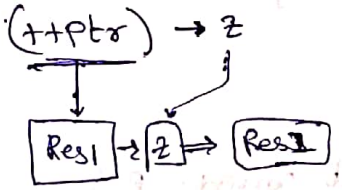
[.], (), ->, ; •
has same precedence
Associativity: Left → Right

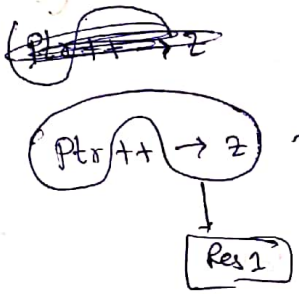
→ a[--(ptr->p->i)].z
a[--(112->i)].z
a[--(3)].z
a[2].z
300 —— kanpur

Note:

Consider



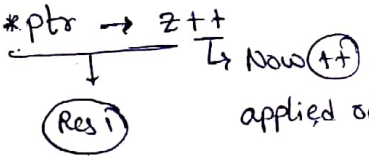




→ This is also equivalent to (ptr++) → z

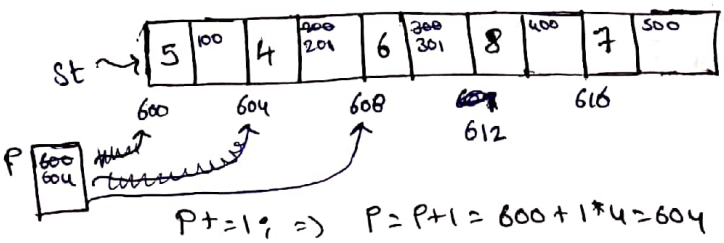
Now ++ is applied on ptr++

Now consider



P/54

became 100 | better 200 | jungle 300 | ancestor 400 | brother 500



P+=1; => P = P+1 = 600 + 1*4 = 604

++P→c → 200 → 201

printf("%s", P++→c)

Here first P→c is evaluated later P++

→ 200 → ether

=> P = 608

printf("%c", *++P→c)

→ 300 → 301 → u

printf("%d", P[0].i)

→ 6

printf("%s\n", P→c)

→ 301 → ungle

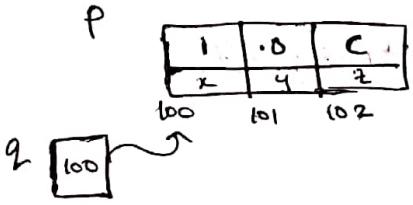
∴ opt (b)

27/05/20

Q44 GATE 2018

```
struct ournode
{
    char x,y,z;
};
int main()
{
    struct ournode P={'1', '0', 'a'+2};
    struct *q = &P;
    printf ("%c %c", *((char*)q+1), *((char*)q+2));
    return 0;
}
```

A) 0, c
B) 0, a+2
C) '0', 'a+2'
D) '0', 'c'



'a'+2
97+2
99 = 'c'

→ *((char*)q+1) Here q is casted to char* pointer
((char)100+1) so '+1' increament by size of char
*(101)
= '0'

→ Silly *((char*)q+2)
((char)100+2) = *(102) = c

∴ o/p: 0, c

∴ opt (A)

Pointer to function:

Declaration of pointer to function:

Syntax:

```
return_type   (*ptr) (arg1_type, arg2_type...);
```

Eg:
```
void (*p)(int,int);
int  (*p)(int,int);
```

Assigning a pointer to function:

```
ptr = function_name;
      (or)
ptr = &function_name;
```

No parantheses should be written

Accessing a function using pointer:

```
ptr(arguments);        (or)   (*ptr)(arguments)
```

Eg:
```
ptr(2,3);                     (*ptr)(2,3);
ptr();                        (*ptr)();
```

Q45

```
int get()
{
    int n=5;
    return n;
}
void put(int k)
{   printf("%d", k);
}
void main()
{   int m;
    int (*p)();
    void (*q)(int);
    p=get; → base address of get is copied into p
    q=put; → bases address of put is copied into q
    ;
    m=p();
```

Function name holds base address of function block.

```
    q(m); // 5
}
```

Qu6

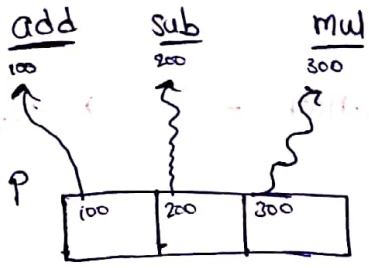
```
int add(int x, int y)
{
    return x+y;
}
int sub(int x, int y)
{
    return x-y;
}
int mul(int x, int y)
{
    return x*y;
}
void main()
{
    int (*p)(int, int);
    int n
    printf("Enter your choice: 1,2,3");
    scanf("%d", &n);
    switch(n)
    {
        case 1:
            p=add;
            break;
        case 2:
            p=sub;
            break;
        case 3:
            p= mul;
            break;
    } default: printf("Invalid Choice");
    printf("%d", p(3,2));
}
```

O/P:

n	P(3,2)
1	add(3,2) = 2
2	sub(3,2) = 1
3	mul(3,2) = 6
4	Invalid choice & Uninitialized Pointer Problem.

Array of function Pointers:

```
int add(int x, int y)
{
    return x+y)
}
int sub(int x, int y)
{
    return x-y;
}
int mul(int x, int y)
{
    return x*y;
}
void main()
{
    int (*P[3])(int,int);
    int i;
    P[0]=add;
    P[1]=sub;
    P[2]=mul;
    for(i=0; i<3; i++)
        printf("%d", P[i](3,2));
}
```



o/p : 5, 1, 6

This can also be written as

```
(*P[i])(3,2);
```

Recursion:

Questions on recursion are asked in

5-6 Marks

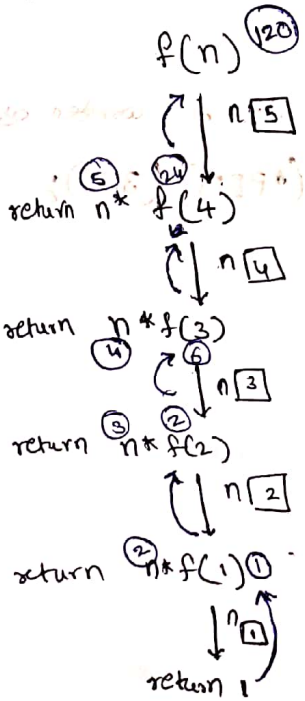
- PL → Return value
- DS → Applications of stack
- Algorithms → Solving Recurrence Equation
  - * Substitution
  - * Master Theorem
  - * Recursion tree.
- ~~D.Maths~~
- D.Maths → Solving Recurrence Equation
  - * Method of characteristic roots
  - * Method of undetermined coefficients

Qu7

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

what is the return value of f(5)?

Sol:

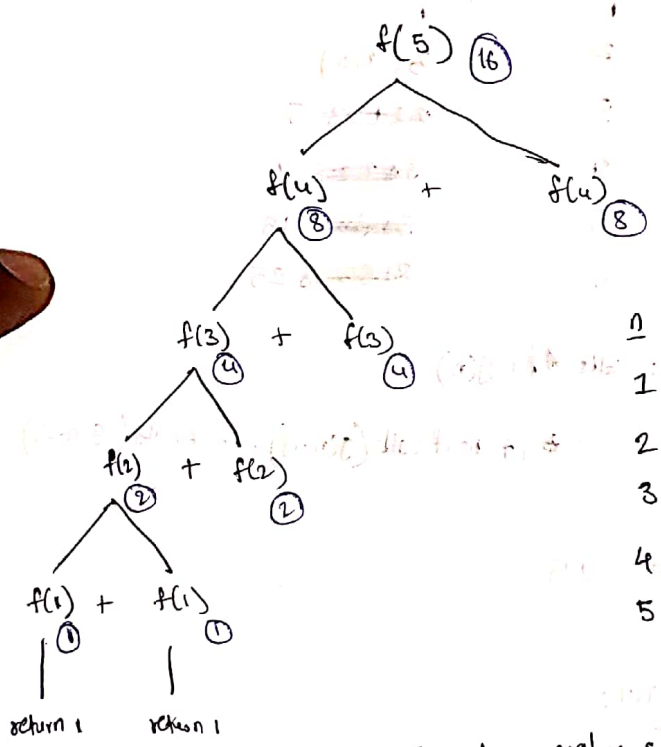


∴ 120

Qu8

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

what is the return value of f(5)?



n	no of function calls
1	1
2	3(2,1,1)
3	7(3,2,1,1,2,1,1)
4	15
5	31

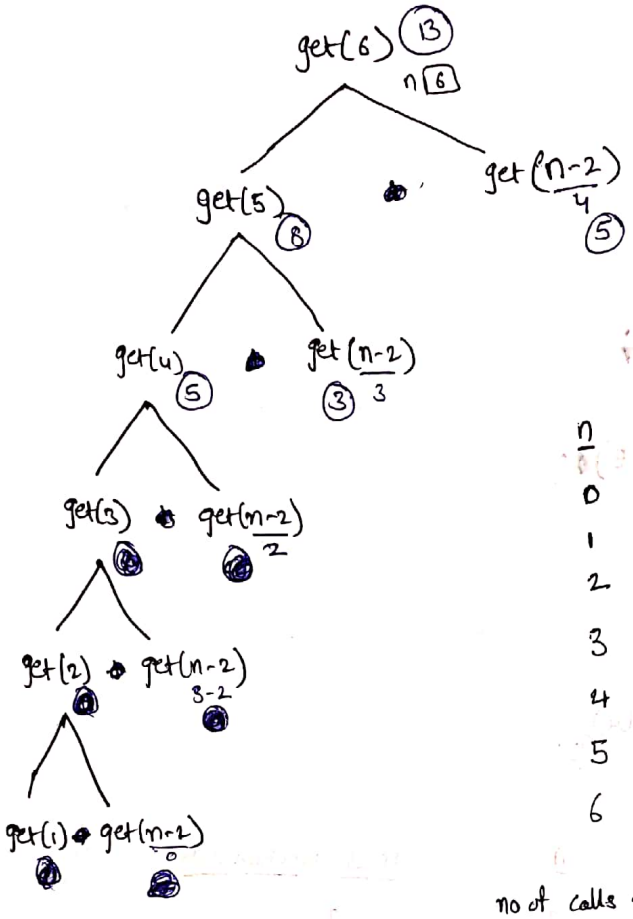
∴ return value = 16

Qu9

GATE

```
void get(int n)
{
    if(n<=1)
        return;
    get(n-1);
    get(n-2);
    printf("%d", n);
}
```

How many times get() function is called for n=6 (By including get(6))?



n	no of function calls
0	1
1	1
2	3 (1,0)
3	~~2+1+3~~ 5
4	~~3+1+5~~ 9
5	~~5+3+8~~ 15
6	~~8+5+13~~ 25

no of calls for g(n)

= 1 + no of calls(g(n-1)) + no of calls(g(n-2))

∴ no of calls = 25

Qu7 by reccurence equation :-

Let T(n) = no of function calls for f(n)

$T(n) = 1 + T(n-1)$ if $n>1$

$= 1$ if $n=1$ or $n=0$

$\Rightarrow \boxed{T(n) = n}$

∴ no of calls for f(5) = 5

Qu8 by reccurence equation :

no of calls

$T(n) = 1 + T(n-1) + T(n-1)$ if $n>1$

$= 1$ if $n=0$ (or) $n=1$

T

by recurrence eq

Q49) Let T(n) no of function calls for get(n)

$$T(n) = 1 + T(n-1) + T(n-2) \quad \text{if } n > 1$$
$$= 1 \quad \text{if } n \le 1$$

$T(6) = 1 + T(5) + T(4) = 25$ (16, 9)

$T(5) = 1 + T(4) + T(3) = 16$ (9, 5)

$T(4) = 1 + T(3) + T(2) = 9$ (5, 3)

$T(3) = 1 + T(2) + T(1) = 5$ (3, 1)

$T(2) = 1 + T(1) + T(0) = 3$

$\therefore$ no of calls = 25

Q50) unsigned int

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

Here both are parenthesis
$\therefore (n\%r)$ is evaluated first

Q1) find return value of f(345, 10)

Q2) find return value of f(513, 2)

f(345, 10) (12)

n 345, r 10

5 + f(34, 10)

n 34, r 10

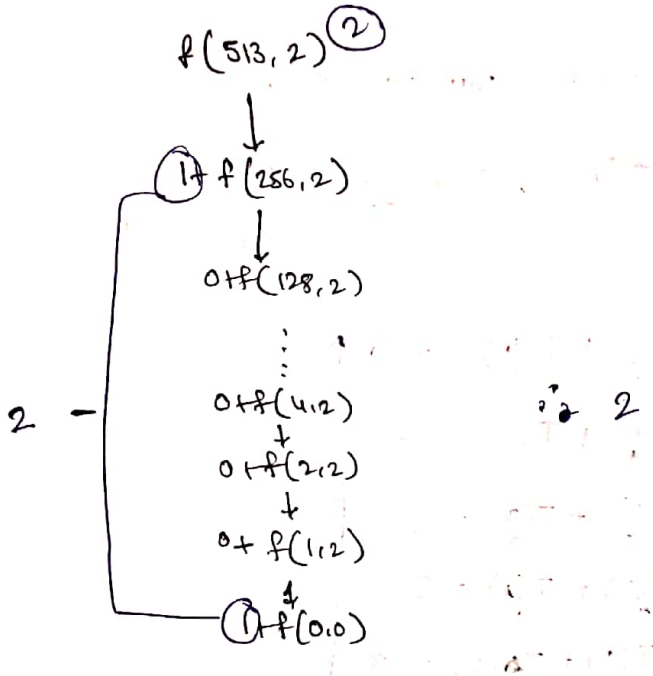
4 + f(3, 10)

n 3, r 10

3 + f(0, 10)

n 0, r 0

$\therefore$ 12



28/05/20

(Q51)

```
int f(int arg)
{
    static int num=6;
    if (num<=1)
        return num;
    return arg + f(arg--, num--);
}
void main()
{
    printf("%d", f(16));
}
```

Find o/p of above program.

f(16) (11)

arg 16 10 ↓

(10) arg + f(10) (1)

arg 10 5 ↓

5 arg + f(5) (-4)

arg 5 1 ↓

(1) arg + f(1) (-5)

arg 1 -2 ↓

-2 (0) arg + f(-2) (-3)

arg -2 -4 ↓

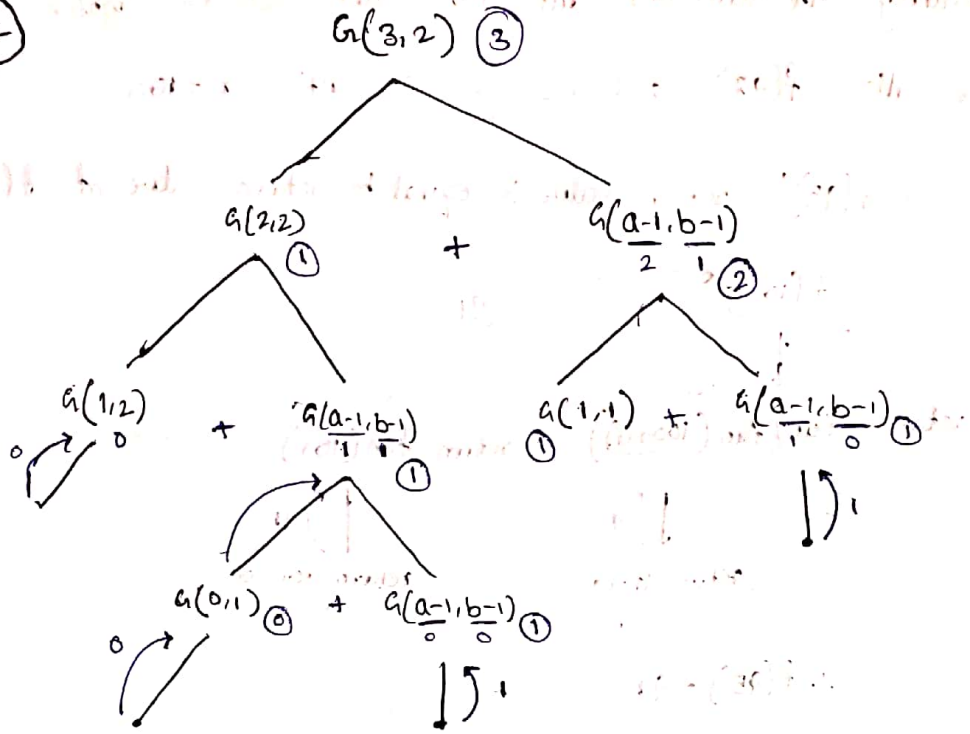
(-4) arg + f(-4)

arg -4 ↓ (1)

num 6 4 3 2 1

∴ o/p = 11

P/12



o/p : 3

Function calling sequence : (3,2) (2,2) (1,2) (1,1) (0,1) (0,0) (2,1) (1,1) (0,1) (0,0) (1,0)

No of function calls : 11

Recurrence eq for no of calls :

$$T(a,b) = 1 + T(a-1,b) + T(a-1,b-1) \quad \text{if } b!=0 \text{ \& } a \geq b$$

$$= 1 \quad \text{if } b=0 \text{ (or) } a<b$$

P/13

```
main()
x=5
result = f(x) * f(x)
```

x = 5 (6) (6) result = 36

x = 6 ↓ 6 6 ↑ x = 6 result = 36

P/14

f(95)

↓

return (fun(fun(x+11))) = return fun(96)

(96)

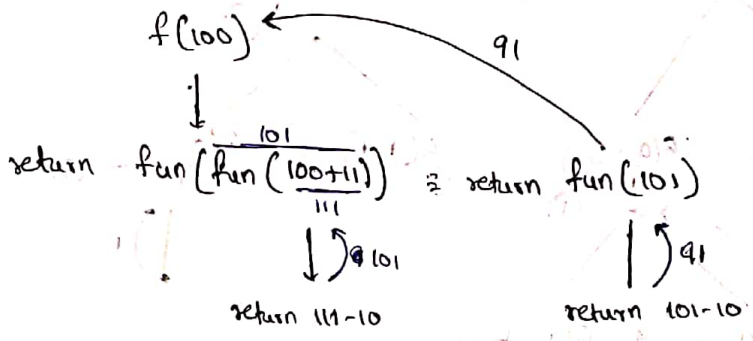
106

↓

return 106-10=96

Now similarly fun(96) ultimately calls fun(97) which further calls f(98) and this goes on until x = 100

∴ f(95)'s return value is equal to return value of f(100)



∴ f(95) = 91

P/16

main()

a 2048 sum 0 pf(sum) 0

f(2048, 0)

n 2048 sum 8 pf(k) 8

k 8 j 204

foo(204, 8)

n 204 sum 12

k 4 j 20

pf(k) 4

foo(20, 12)

k 0

pf(k) 0

foo(2, 12)

k 2

pf(k) 2

foo(0, 4)

∴ o/p: 2, 0, 4, 8, 0

P/20 fun(1) = 1

fun(2) =
x = x + fun(k) * fun(n-k)
= x + fun(1) * fun(1)
= 1 + 1 * 1
= 2 → initial x value

fun(3) = 1 + fun(1) * fun(2) + fun(2) * fun(1)
= 1 + 2 + 2 = 5

fun(4) = 1 + fun(1) * fun(3) + fun(2) * fun(2) + fun(3) * fun(1)
= 1 + 5 + 4 + 5
= 15

fun(5) = 1 + fun(1) * fun(4) + fun(2) * fun(3) + fun(3) * fun(2)
+ fun(4) * fun(1)
= 1 + 15 + 10 + 10 + 15
= 51

Q52 GATE 2019

```
void convert(int n)
{
    if (n<0)
        printf("%d", n);
    else
    {
        convert(n/2);
        printf("%d", n%2);
    }
}
```

Which one of the following will happen when the function convert is called with any positive integer n?

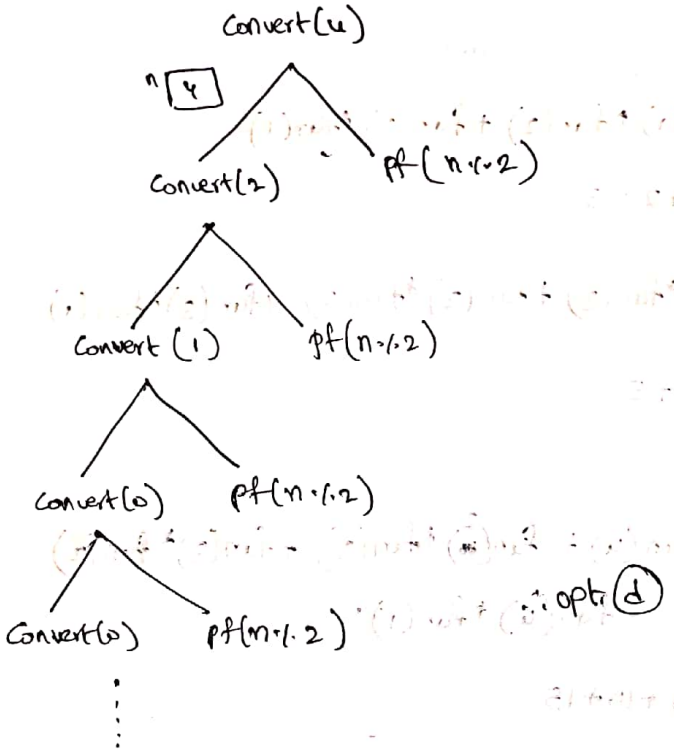
a) It will print the binary representation of n and terminates.

b) It will print the binary representation of n in reverse order and terminate.

c) It will print the binary representation of 'n' but will not terminate

d) It will not print anything and it will not terminate.

Sol:



∴ opt: (d)

Q53 GATE 2019

```
int jumble (int x, int y)
{
    x = 2*x + y;
    return x;
}
int main()
{
    int x=2, y=5;
    y = jumble(y,x); // y = 2*y+x = 12
    x = jumble(y,x); // x = 2*y+x = 26
    printf("%d", x); ⟶ 26
    return 0;
}
```

x [~~2~~ 26] y [~~5~~ 12]

∴ O/P: 26

Q54 ~~int do(int)~~

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

what is the return value of D(16384)?

Hint: $\sqrt{16384} = 128$

Sol:

f(16384): 16527.

n 16384

do(floor(sqrt(n))) + n → 16384

128 → 128 + 15 → 143

do(128)

n 128

do(floor(sqrt(128))) + n → 128

11 → 15 + 128

n 11

do(floor(sqrt(11))) + n → 11

3 → 4 + 11

n 3

do(1) + n → 3

1

∴ O/P: 16527

Q55

```
double foo(int n)
{
    int i;
    double sum;
    if(n==0)
        return 1.0;
```

```
else
{
    sum = 0.0;
    for(i=0; i<n; i++)
        sum += foo(i);
    return sum;
}
}
```

what is the return value of foo(3)

Sol:

foo(0) = 1

foo(1) = foo(0) = 1

foo(2) = foo(0) + foo(1) = 2

foo(3) = foo(0) + foo(1) + foo(2) = 1 + 1 + 2 = 4

P/11

GATE 2017

Let x=2 y=3

a=2 b=3 res=1

For this set of values all four options satisfy the condition

Now iterate loop once

res = 1*2 = 2 a=2 b=2

a) $2^3 = 2^2$ ✗

b) $(2*2)^3 = (2*2)^2$ ✗

c) $2^3 = 2 * 2^2$

$= 2^3$ ✓

d) $2^3 = (2*2)^2$

$2^3 = 4^2$ ✗

So now we can eliminate a, b, d

∴ opt (c)

29/05/20

Static scope & Dynamic scope:

Static scope:

Referencing environment of a statement in a statically scoped language is the collection of all local variables and all other ancestor variables which are visible in the statement.

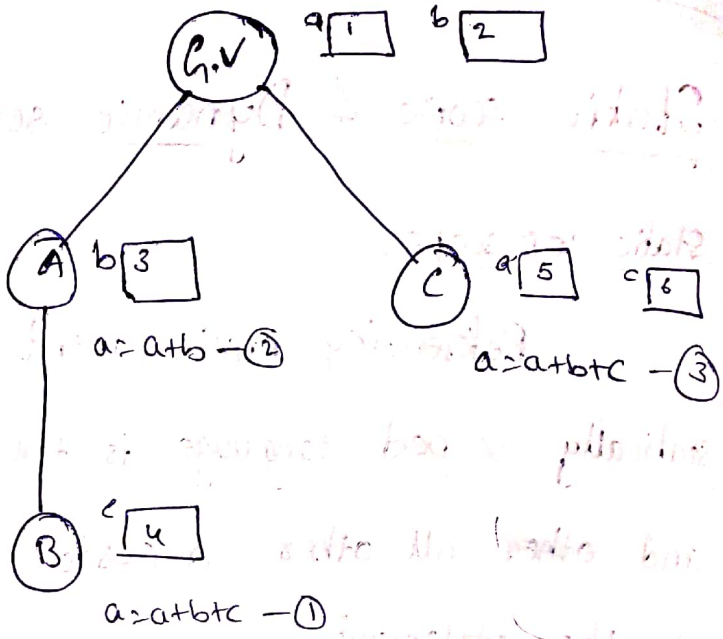
Dynamic scope:

Referencing environment of a statement in a dynamically scoped language is the collection of all local variables and all other active subprogram variables which are visible in the statement.

Eg: Write the referencing environment of the statements ①, ②, ③ for the given psuedocode using static scope

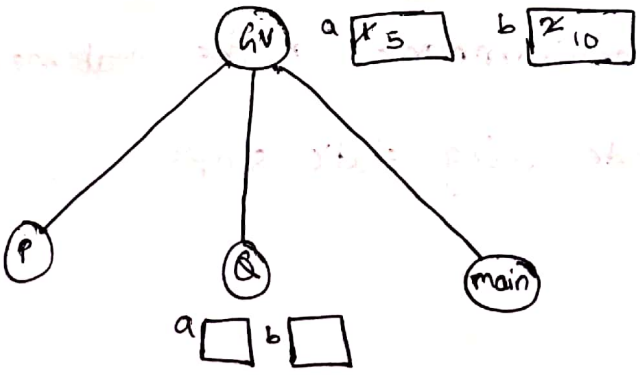
```
var a,b : integer;
a:=1;
b:=3;
procedure A
    var b: integer;
    b:=3;
    begin
        procedure B
            var c: integer;
            c:=4;
            begin
                a:=a+b+c; ——①
            end
        a:=a+b; ——②
    end A
```

```
Procedure C
    var a,c: integer
        a:=5, c:=6;
    begin
        a:= a+b+c;
    end c
```



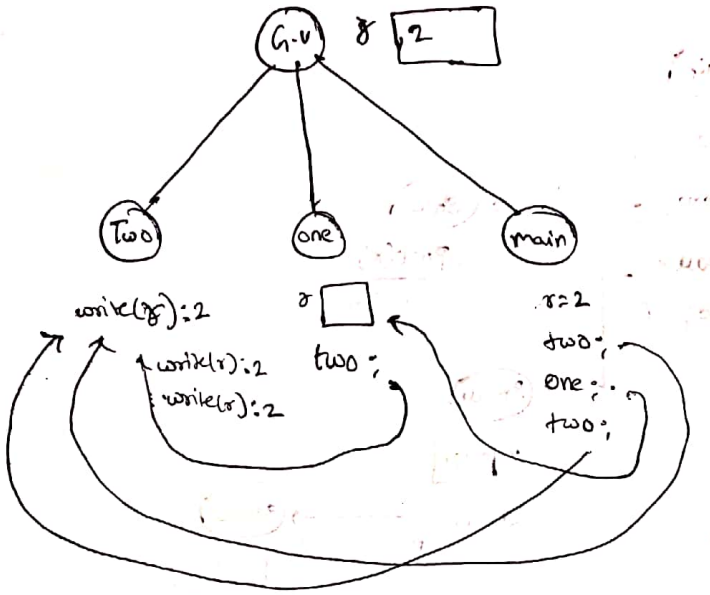
Stmt	Ref. Environment
1	'c' of (B) + 'b' of (A) + 'a' of (G.V)
2	'b' of (A) + 'a' of (G.V)
3	'a' of (G.V) + 'b' of (G.V)

P/68(i)



∴ o/p: 5, 10

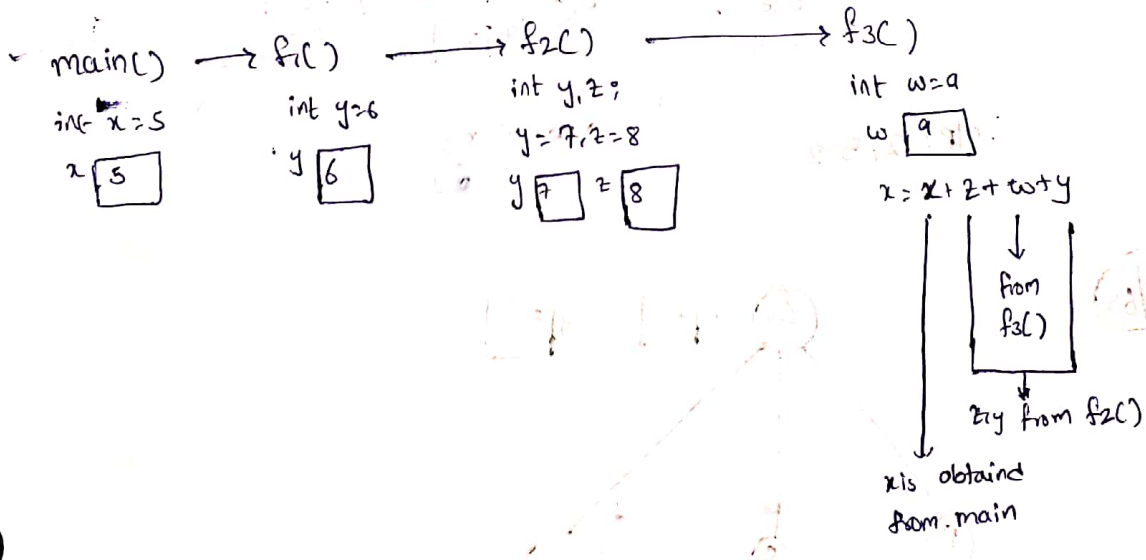
P/69(i)



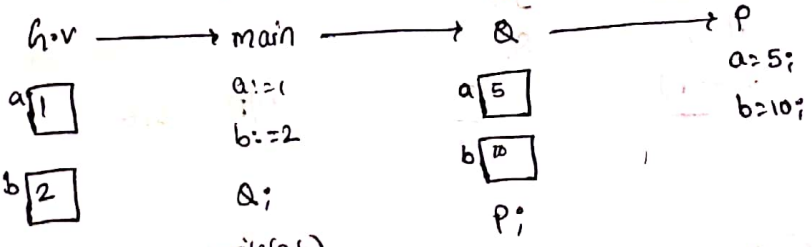
∴ O/P: 2,2,2

Dynamic scope:

Consider the function calling sequence

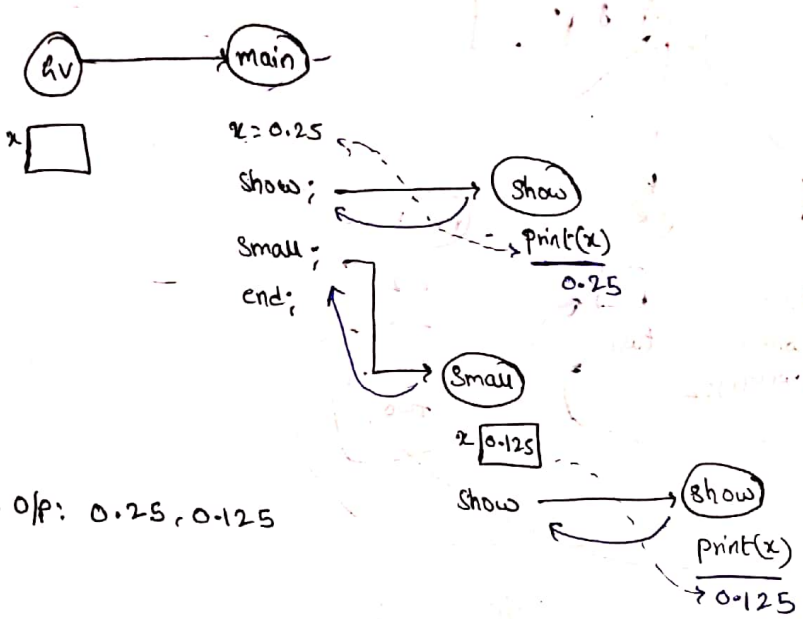


P/68(ii)



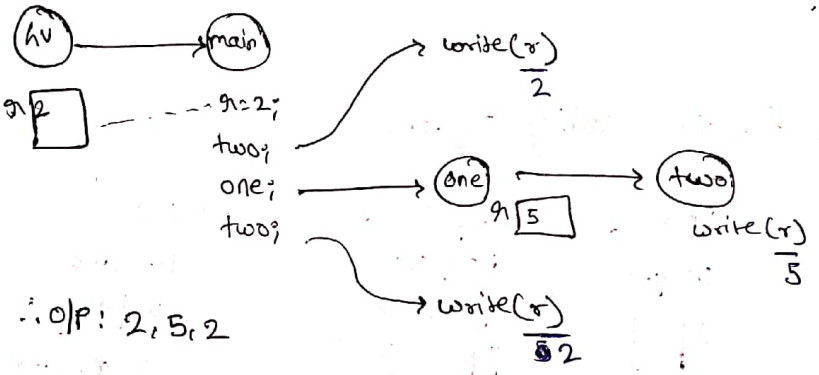
∴ O/P: 1, 2

P/70



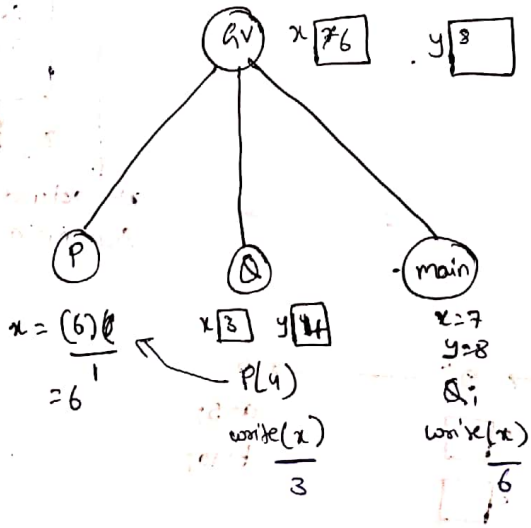
∴ O/P: 0.25, 0.125

P/69(ii)



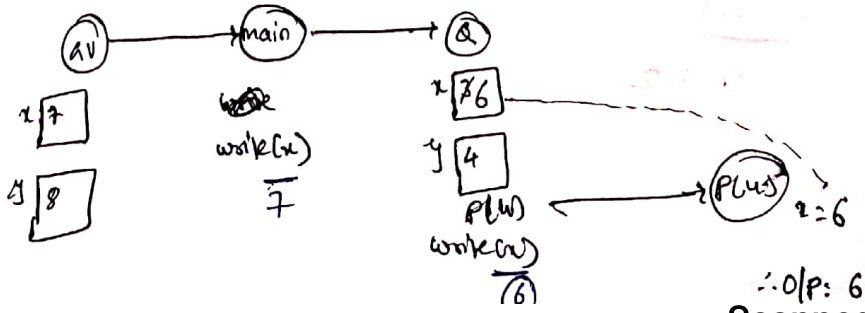
∴ O/P: 2, 5, 2

P/65



∴ O/P: 3, 6

P/66



∴ O/P: 6, 7

Parameter Passing Techniques:

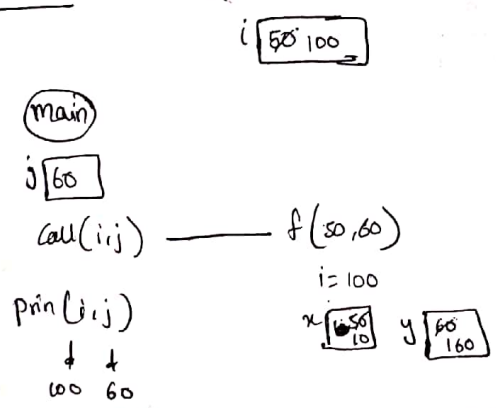
1. call by value:

In call by value actual parameters always carry their values and these values are copied formal parameters, later formal parameters may be modified, but these modifications are not reflected in the calling function variables.

2. Call by Reference:

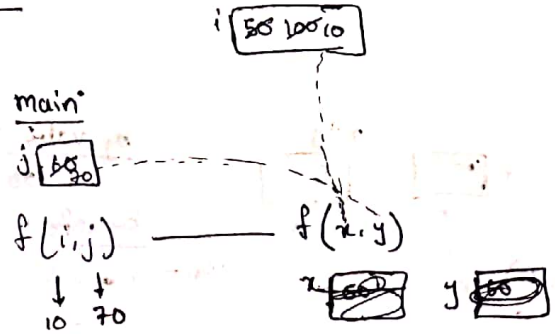
In call by reference, actual parameters are passing their address and formal parameter are acting as an alias to their corresponding actual parameters.

P/58 by value



o/p: 100, 60

by ref

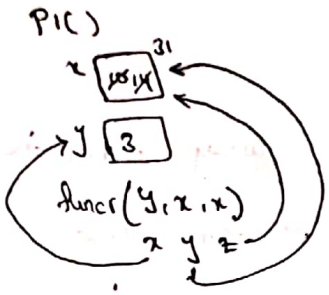


∴ o/p: 10, 70

∴ opt (4)

30/05/20

P/64



y=y+4

z=x+y+z;

3+14+14

31

∴ opt ⑥ 31,3

P/63

Procedure P(x,y,z)

begin

y=y+1;

z=z+x;

end

begin

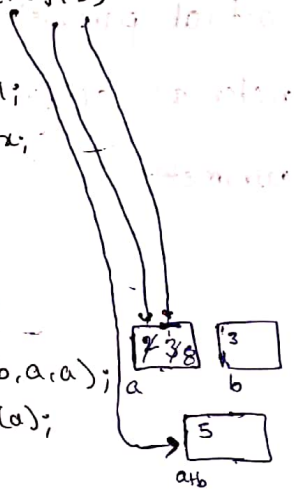
a=2;

b=3;

P(a+b,a,a);

print(a);

end



call by value:

print(a) — 2

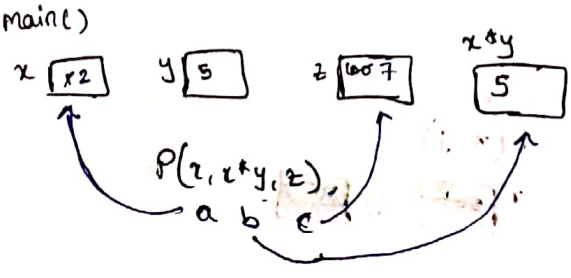
call by ref:

∴ o/p = 8

Temporaries:

- When an expression is passed as a parameter, a temporary memory location is created and the result is stored in it. This is a part of activation record.

P/61



by value:

1, 100

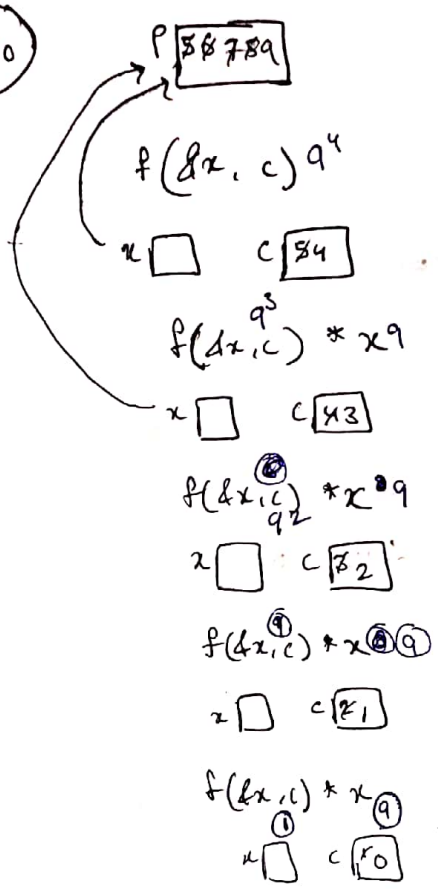
by refe:

write(x, z)

2 7

∴ 2, 7

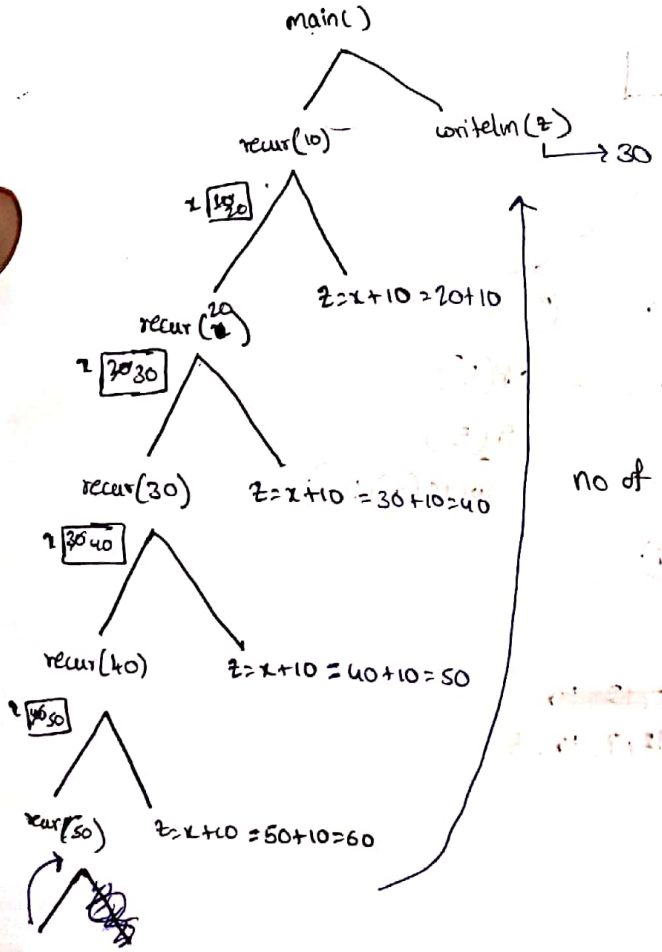
P/60



∴ o/p: $9^4 = 6561$

P/62 Call by value:

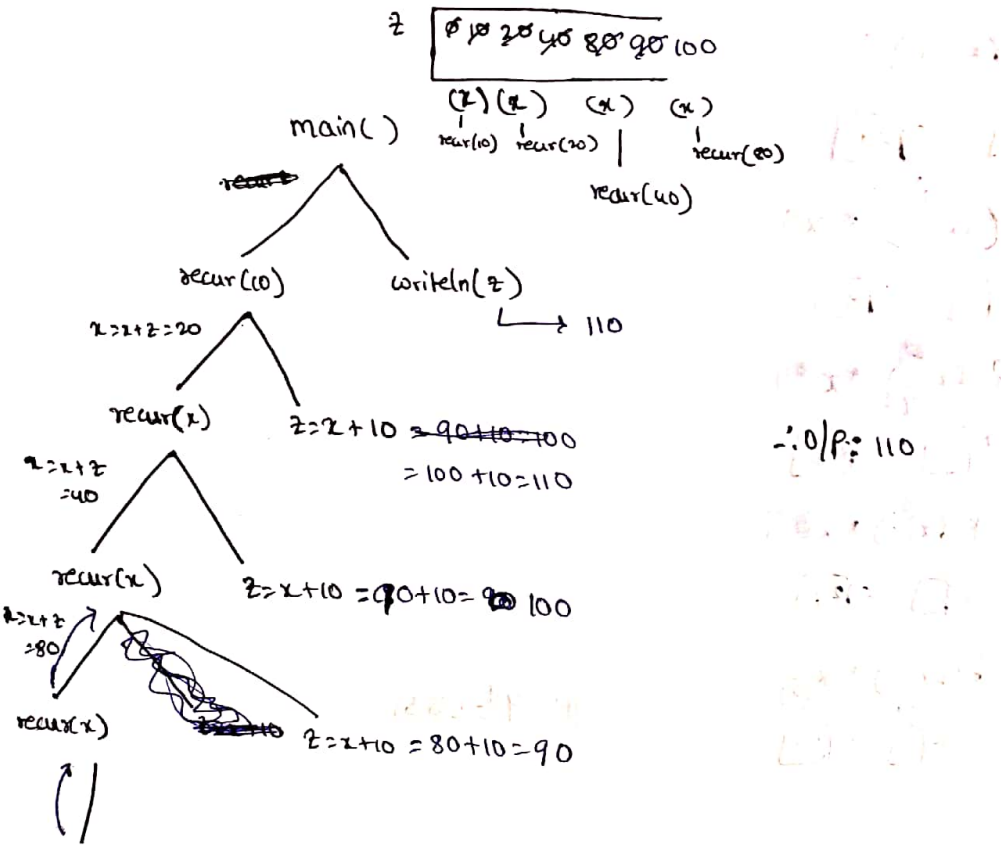
z 10 20 30 40 50 60



∴ o/p: 30

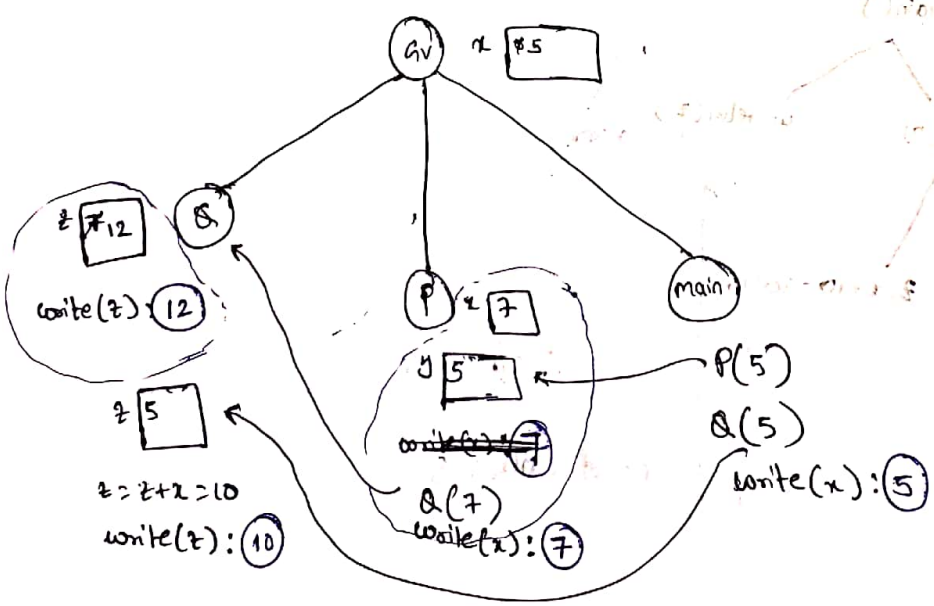
no of calls: 5

Call by reference:



∴ O/P: 110

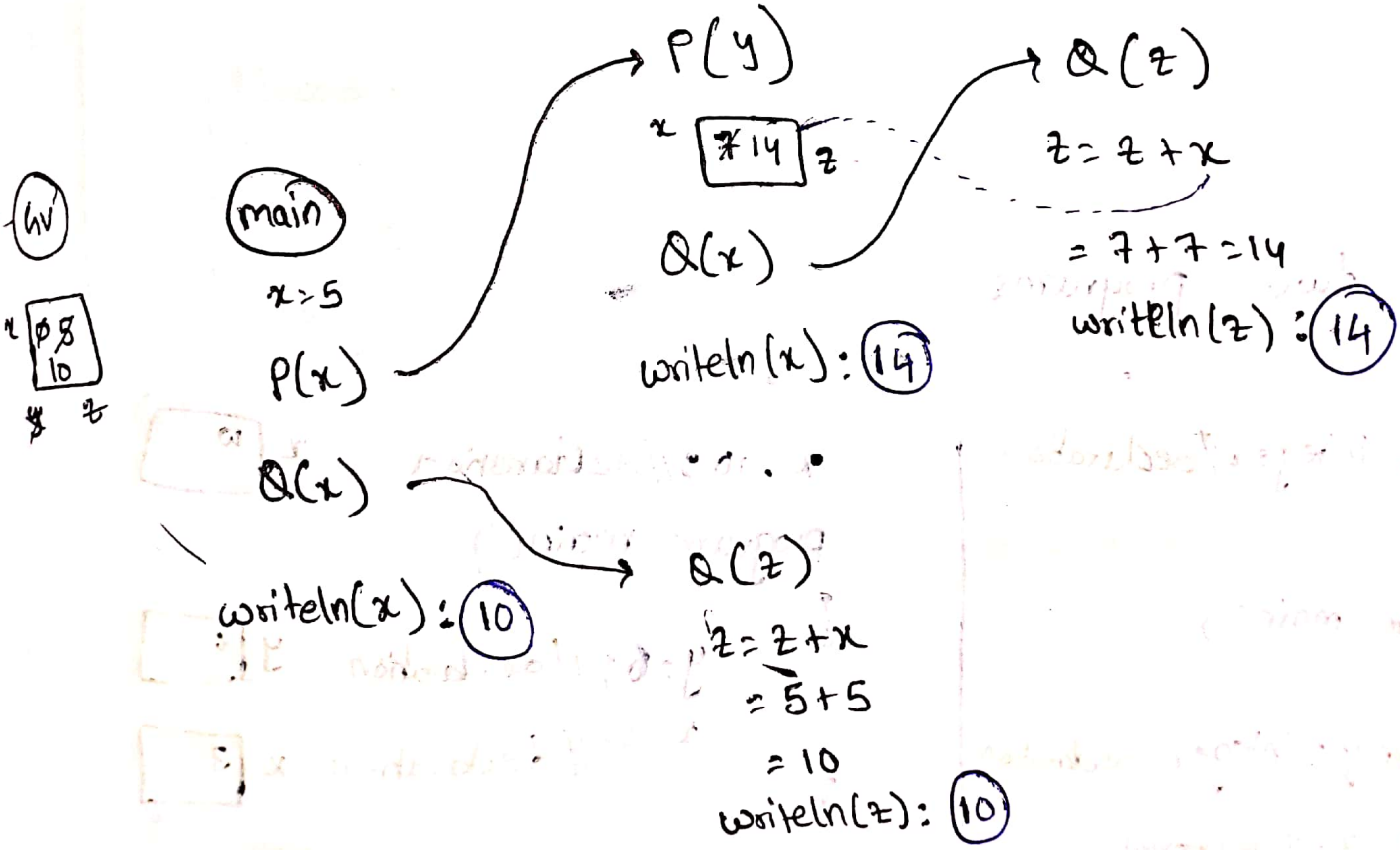
P/71 **Call by value & static scope:**



∴ O/P: ~~7 12 10~~

12, 7, 10, 5

Call by ref & Dynamic scoping :



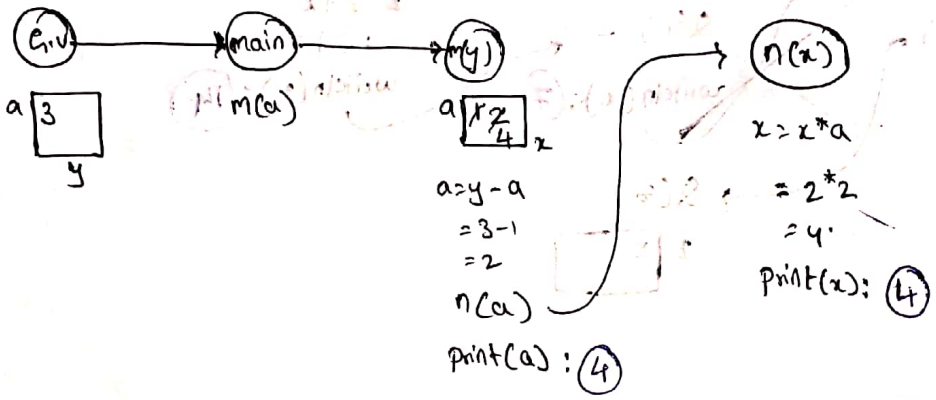
∴ O/P: 14, 14, 10, 10

P174

a=3; is a global variable declaration

Thus the stmt

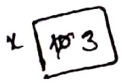
a=1 in m(y) should also be considered a declaration.



∴ O/P: 4 4

Note:

Consider the two programs




```
var x: integer//declaration
x=10;
program main()
{
var y: integer//declaration
x=3;//assignment
y=5;//assignment
}
```

In the above program the syntax for declaration is written as var x: datatype. So, the same syntax must be followed throughout the program

```
x=10;//declaration
program main()
{
y=6;//declaration
x=3;//declaration
}
```

Here the declaration syntax followed is

x = value;

Access link:

Every activation record in the stack must have access link to the most recent activation record in which lexically contains it.

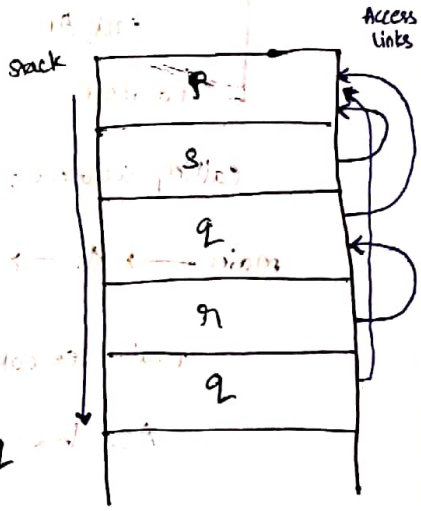
~~Access links are stored in Activation record.~~

(P/67)

```
Procedure P
    Procedure q
        Procedure r
            begin
                q; // function call
            end r
        begin
            r;
    end
    Procedure S
    begin
        q;
    end s
    s;
end P
```

Here procedure p lexically contains q, r, s. because they are defined within P.

Silly q lexically contains r



Given sequence is

~~S~~ P → S → q → r → q

→ To create access link to 'q' first check which is lexically containing 'q'. i.e, P

→ Now, To create access link to 'r' check which is lexically containing 'r' i.e., q & P.

But q is most recent

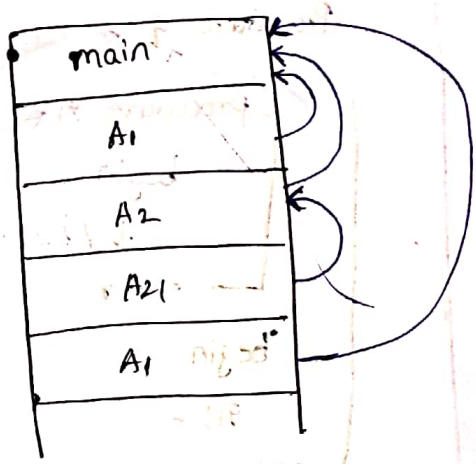
Note:

→ Access link concept is used in static scoping.

→ Using access links it checks for variables in ancestor functions.

P/73

```
Program main
    var ....
    Procedure A1
        var....
        Call A2
    end A1
    Procedure A2
        var....
        Procedure A21
            var....
            Call A1
        end A21
        call A21
    end A2
    call A1
end main
```



Calling sequence:

main ⟶ A1 ⟶ A2 ⟶ A21 ⟶ A1

main lexically contains A1, A2, A21

A2 lexically contains A21

Note:

In pascal language

procedure w(var x: int) — This is considered address variable

procedure w(x: int) — This is considered value variable.

P/57

main()

x [15x2285]

(8) fun(5, &x)

n [5] fp [ ]

t [5] f [8]

t = fun(4, fp) (5)

n [4] fp [ ]

t [3] f [5]

t = fun(3, fp) (3)

n [3] fp [ ]

t [2] f [3]

∴ o/p: 8

t = fun(2, fp) (2)

n [2] fp [ ]

t [1] f [2]

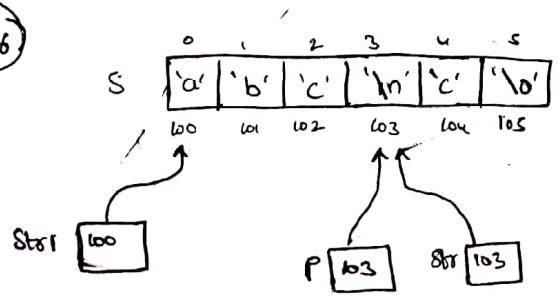
t = fun(1, fp) (1)

n [1] fp [ ]

t [ ] f [ ]

*fp = 1

P/56



'\n' = 10

++*p + ++*str1 -32

++10 + ++97 -32

11 + 98 -32 = 77 = M