C PROGRAMMING

Handwritten Notes

VIVEKANAND VERNEKAR

GateWallah Parakram C 2024

(Hinglish Weekday)

Other Notes Uploaded Below

https://t.me/pwgatenotes

Data Types and Operators

Data Types

- **Primitive Types**
  1. integer
  2. floating
  3. Character
  4. bool, void
- **Derived**
  1. array
  2. pointer
  3. String
- **User defined**
  1. Structure
  2. Union
  3. Enum
  4. typedef

Integer
- Short
- int
- long int
- long long int

all are signed by default

all have Signed & Unsigned type

Signed → (+42, -36)

Unsigned → (36, 42 ...)

default
Signed short int a = 2
or (same)
Short int a = 2

Floating
- float
- double
- long double

Char
- Signed char
- unsigned char

*Assuming - 2B
↓
Short int
- Signed
- unsigned

2 byte = 2×8 = 16 bit
(each bit stores 0 or 1. (2 possible value))

$= 2^{16}$

$= 65,536$

↓

Unsigned Short int = (0 to 65,535)

Signed short int

↳ 65536

32765.

−ve: (−32768 to −1)

+ve: 32768 (0 to 32767)

⟹ −32768 to 32767 } Range

int ⟹ 4B = 4×8 = 32 bits

$= 2^{32}$ possible values.

Unsigned int : 0 to $2^{32}-1$.

Signed int : 2^{32}

−ve: $\frac{2^{32}}{2^1}$ $(-2^{31} \dots -1)$

+ve: $\frac{2^{32}}{2^1}$ $(0 \text{ to } 2^{31}-1)$

} Range = -2^{31} to $+2^{31}-1$

General : n bits = 2^n values

Unsigned = 0 to 2^n-1.

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

```
#include <stdio.h>
void main ()
{ short int a;
signed short int b;
  short c;
signed short d;
}
```

all are signed short int type

all valid declarations.

```
#include <stdio>
void main ()
{  signed int a=10; //initialized
     int b; }
```

declared (has garbage values)

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

everything inside " " is considered as text

to print value in a → printf("%d", a)

format specifier

```c
#include <stdio.h>
void main()
{ short int a= 20;
Printf("The value is %d", a);
}
```

O/p : The value is 20

%d → format Specifier of integer type (Short int and int type not for LL)

```c
# include <stdio.h>
void main()
{ Signed short int a=20;
Printf("%d is %d", a, a);
}
```

the value are Printed accordigly

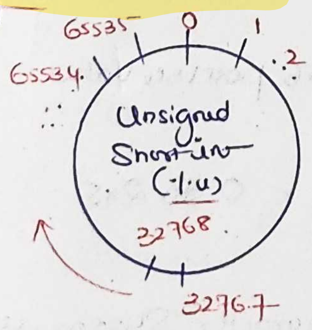
O/p ⇒ 20 is 20

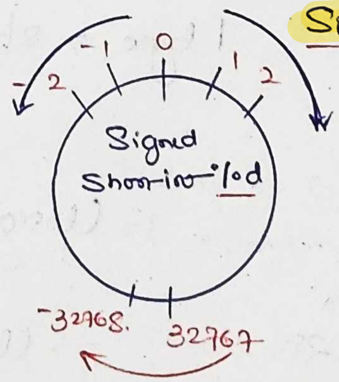
%d → Short int, int	%u → Unsigned Short int, Unsigned ~~short~~ int
% ld → long int	
% lld → long long int	
% c → Char	

Signed int → 2 Byte { Cyclic Property }

→ Unsigned 0 to 65535

Signed : -32768 to +32767





excess numbers, goes in cycle after limit

Short int i = -2 Printf("%d", i) O/p = -2.	unsigned short int i = -2; Printf("%u", i); O/p = 65534	Short int i = 32769; Printf("%d", i); O/p = -32767 because of cyclic Property

Short int i = -32770;

Printf("%u", i);

O/p = 32766

↳ goes ahead in cycle after limit.

Unsigned short int a = -3;

Printf("%u", a); // 65535

Printf("%d", a);

-3

because Printf in "%d" which prints signed value.

-3, it's not available in unsigned, so goes in reverse of cycle ∴ 65535

Taking input from the keyboard

```
#include <stdio.h>
void main()
{   int a;
  Printf("Enter a number");
    Scanf("%d", &a);
 Printf("The value of a is %d", a);
}
```

&a → "address of a".

format to store values at particular address of a variable

Char

1 byte = 8 bits = 2^8
= 256 possible values

Signed Char

256

128 128

(-128...-1) (0 to 127)

Range = -128 to 127

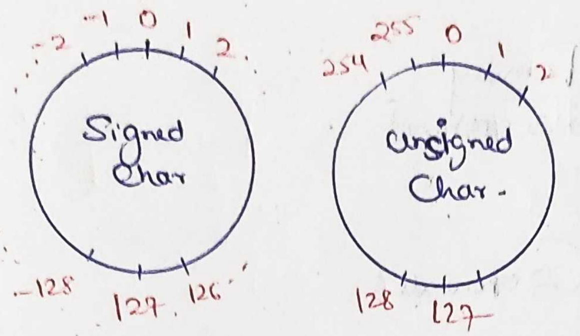
Unsigned Char = 0 to 255

→ Using a Character System, we represent Symbols of a long by the integer constants

A - 65	a - 97	'0' - 48
↓	⋮	⋮
Z - 90	z - 122	

ASCII System.

%c → Character System.



signed by default

Char c = 65;

Printf("%d", c);

Signed int value ∴ O/p = 65

Printf("%c", c);

Char ∴ O/p = 'A'

has ascii value = 65

Char c = 128;

Printf("%d", c)

signed int, but 128 cannot be stored in 'Char' data type, so goes in cycle due to cyclic property

O/p = -128

Char c = -130;

Printf("%d", c);

Signed integer, -130 not available so goes in reverse cycle

O/p = 126

declared value Char c	Stored value
-130	126 (-130 + 256)
-132	124 (-132 + 256)
-191	65 (-191 + 256)

Char c = -191

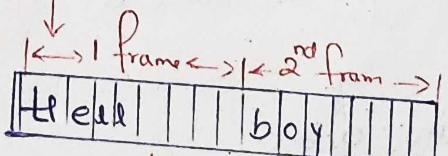
Printf("%c", c);

Char value

O/p = 'A' because value '65' is stored because of cyclic property

"\n" → Moves the cursor to the beginning of next line.

"\t" → Moves the cursor to next empty available frame.



Printf("Hell\t boy");

each frame is of 8 characters

8 space in only MS-DOS

→ A frame is group of 8 characters.

31/10/2023

Operators

Unary	Binary	Ternary
→ 1 operand	→ 2 operand	→ 3 operand

1) Binary Operators (2 operands)

→ **Assignment Operator:**

$a = \underbrace{30 \times 2 + 1}$

evaluate value & then Store

Lvalue = Rvalue

Rvalue → expression / constant / variable

Lvalue:
- Cannot be expression
- Cannot be constant
- must be some variable.

→ **Arithmetic Operator**

$\underbrace{+, -, *, /}_{\text{Math}}$ %

↳ Modulo

* Follows BODMAS Rule & Priority (precedence)

(i) %, *, / ↓ Priority high to low
(ii) +, −

eg: 8/2*4 → both operator of Same Priority

Here Associativity comes into picture. (Left → Right)

a%b : returns the remainder when a is divided by 'b'

* both operand must be of integer type
* Sign of result is same as sign of first operand

→ eg:
```
int a,b,c;
a=b=c=4;
printf("%d", a); → 4.
```

Result of an Operator ⟶ Operand

int ← $12/5 \rightarrow 2$

float ← $12.0/5 \rightarrow 2.4$

$12/5.0 \rightarrow 2.4$

$12.0/5.0 \rightarrow 2.4$

eg: int a;

a = 4.0 * 6 % 3 + 2;

printf("%d", a);

→ Compiler error

= 4.0 * 6 % 3 + 2

$= \underbrace{24.0 \% 3}_{\text{Compiler error.}}$

Relational Operator

greater >, < smaller, >=, <=, == equals to, != not equals to

The result of every relational operator is either 0 or 1.

high

Unary	+, −	
Arith.	[*, /, % +, −]	Left to Right
Relational	[<, >, <=, >=, ==, !=]	Left to Right associative.
Assignment	=	R to L

low.

Q1.) int i;

i = printf("%d", 4/2+3*2);

Printf("%d", i);

O/p : 81

Q2) int a;

a = printf("%d", printf("Gate 2023"));

printf("%d", a);

O/p: Gate 202391.

Logical Operators

(i) Logical AND (&&)
(ii) Logical OR (||) } Binary
(iii) Logical NOT (!) } Unary

→ printf("%d", 2 && 7); ⟹ 1
non zero (true), non zero (true)

All non zero values are treated as "true". And zero is treated as false.

→ printf("%d", 2.3 && -13); ⟹ 1

→ printf("%d", 1 && 0) ⟹ 0
false so result is zero.

→ int a; a = 3 && printf("Hello");
printf("%d", a); ⟹ Hello 1.
5 (True)

!(true) = false

→ printf("%d", 13.78 || 0.0) ⟹ 1.

→ int a;
a = 0 && $\boxed{}$
printf("%d", a);

→ If the first operand is 0 for logical AND operator then we need not to evaluate 2nd operand.
"Short Circuit Evaluation".

→ int a;
a = 0 && printf("Hello");
first output is zero
× not evaluated

→ int a;

a = 10 || [ ];

If first operand — not evaluated — "Short-Circuit Evaluation".

for logical OR operator is non zero, we need not to evaluate 2nd operand.

Modify Operator

- Increment (++)
  - Pre-increment: ++var
  - Post Increment: var++
- Decrement (--)
  - Pre-decrement: --var
  - Post decrement: var--

eg:
```c
int a=5, b;
b = ++a;
printf("%d %d", a,b);
```
O/p: 6 6

- first increment the value then assign/use the value.

```c
int a=5, b;
b = a++;
printf("%d %d",a,b);
```
O/p: 6 5

- first use the value of var
- then increment the value

Note: In C standard between any two sequence point, a variable can be modified using modify operator only once. More usage of modify operator results in Compiler dependent output.

①
```
int a=0, b=1, c;
c = (a++ && ++b);   Short circuit evaluation
Printf("%d %d %d",a,b,c);   O/p = 110.
```

②
```
int a=0, b=1, c=2, d;
d = (++a && --b || ++c);
printf("%d %d %d %d", a,b,c,d);   1031
```

③
```
int a=2, b=2, c=0, d=2, e;
e = ((a++ && b++) && c++) || d++;
printf("%d %d %d %d %d", a,b,c,d,e);   O/p = 33131
```

Decimal to Binary

$(67)_{10} \longrightarrow (\quad)_2$

$(1000011)_2$

2	67	Rem.
2	33	1
2	16	1
2	8	0
2	4	0
2	2	0
2	1	0
	0	1

Stop here

Binary to Decimal

$(1000011)_2 = 1\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

$= 64 + 2 + 1$

$= 67.$

10 . . . . $\boxed{1}$ = $2 \times x + 1$ ⟹ odd

. . . . . $\boxed{0}$ = $2 \times x + 0$ ⟹ even.

Bitwise Operators.

(i) Bitwise AND (&)

Check bit by bit

a 00000111

b 00001011

000000011 ⟶ 3

1&1 = 1.
else all 0.

```
int a=7, b=11, c;
c = a&b;
printf("%d",c);
```

O/p = 3.

0&0 = 0

0&1 = 0

1&0 = 0

1&1 = 1.

(ii) Bitwise OR (|)

0001 0001

0000 1001

11001

```
int a = 9, b=17, c;
c = a|b;
printf("%d",c);
```

O/p = 25.

0|0 = 0

0|1 = 1

1|0 = 1

1|1 = 1.

(iii) Bitwise XOR (^)

```
int a=5; b=9, c;
c=a^b;
printf("%d",c);
```

O/p = 12

00000101

00001001

00001100 ⟶ 12

0^0 = 0

0^1 = 1

1^0 = 1

1^1 = 0

a^0 = a

a^a = 0

a^a^a = a

* Even number of times XOR the same no. ⟹ 0

* Odd no. of times ⟹ a

(iv) Bitwise left-Shift Operator

int a=5;

b = a<<1;

10 — no. of bits to shift.

Popped out ← | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | ← Empty Space.

$$a<<1 = \frac{a}{2^1}$$

(v) Bitwise Right Shift Operator

int a=10;

int b = a>>2;

$$\frac{a}{2^2}$$

| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | → popped out

| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

extra Space

Negative Number in Binary

Sign magnitude

← 8 bit →

MSB | | | LSB

Sign bit ← 7 bit → (magnitude)

Problem with this = | 0 | 00000000 |, | 1 | 00000000 | Creates confusion.

2's Complimentation

+ve = As it is

−ve = 2's complimentation = 1's compliment + 1

eg:

```
 0101001
 1010110   — 1's compliment (flip all bits)
      +1
 -------
 1010111   — 2's compliment number.
```

eg: $-23 \Rightarrow$ 00010111 → convert the positive way
→ 11101001 → 2's compliment, starting from left keep the first set bit (1) as it is and then flip all the remaining bits

→ If the number given in 2's compliment form.

msb ← 1 1 1 0 1 0 0 1.
$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$.

direct method $= -2^4 - 2^2 - 2^1 - 1$
$= \underline{\underline{-23}}$.

Another approach: Find 2's compliment again and then put a negative sign.

(vi) <u>Bitwise Not (~)</u>

```
int a=48;
    b=~a;
printf("%d", b);
```

$\boxed{(\sim x) = -(x+1)}$

$48 \to 00110000$ ($2^5\ 2^4$)
$\sim 48 \to \underbrace{11001111}_{}$ all bits are flipped (2's compliment)

$= -2^5 - 2^4 - 1$
$= \underline{\underline{-49}}$.

<u>Ternary Operator (?:)</u>

exp1 ? exp2 : exp3

true/non zero

false/zero

* Expression 1 is evaluated first if it is true then 'exp2' is evaluated else 'exp3' is evaluated/carried out.

(1.) int a;

$a = \underbrace{12 > 10}_{true} ?\ 12 : 10;$

(2.) $a = \underbrace{10 \,!= 3 > 6}_{exp1} ?\ \underbrace{2 == 6 \,!= 3\ ?\ 10 : 20}_{exp2} : \underbrace{30}_{exp3};$

$a = 10$

true

(3.) (false) ⇒ exp3.

$a = \underbrace{15 < 3}_{exp1} ?\ \underbrace{3 \,!= 4 > 10\ ?\ 10 : 20}_{exp2} : 4 < 7 \,!= 7 > 10\ ?\ !5\ ?\ 30 : !2 \,!= 2\ ?$

$4 : 5 : 6;$

expression 3.

⇓

$a = \underbrace{4 < 7 \,!= 7 > 10}_{exp1} ?\ \underbrace{!15\ ?\ 30 : !2 \,!= 2\ ?\ 4 : 5}_{exp2} : \underbrace{6}_{exp3}$

true

⇓

$a = \underbrace{!15}_{exp1} ?\ \underbrace{30}_{exp2} : \underbrace{!2 \,!= 2\ ?\ 4 : 5}_{exp3};$ ⟹ $a = 4.$

(false)

Shorthand Property:

$x = x + 10 \longrightarrow x += 10$

$x = x \% 10 \longrightarrow x \%= 10$

$x = x \wedge 10 \longrightarrow x \wedge= 10$

$x = x \& 10 \longrightarrow x \&= 10$

$x = x << 10 \longrightarrow x <<= 10$

Similarly for all Arithmetic, bitwise operator

(4.) int a;

a = $\underbrace{12 > 2}_{exp1}$? printf("Gate") && printf("wallah") || printf("2023") (exp 2)

: printf("Sir"); (exp3.)

printf("%d", a);

O/p = Gatewallah1

(5.) int a;

a = 20 > 110 ? 100 : !2! = 3750 ? 300 : 400;

exp1 exp2 exp3 false

printf ("%d", a);

O/p - 400

Control Flow Statements

```
#include <stdio.h>
void main()
{     S1;
      S2;
      S3;
      S4; }
```

Sequential order

1> Selection Statements : if, if-else, switch

2> Iteration Statements : loops : for, while, do-while
(Repetation)

3> Jump Statements : Continue, break, return, exit

Selection Statements

```
S1;
S2;
if (condition/expression)
{  S3; S4; }
S5;
S6;
```

if true S3, S4 will execute.

all stmts in the flower bracket will execute.

```
if (condition/expression)
    S1;
    S2;
    S3;
```

S1; if true

only first statement has scope of if statement.

S2; S3; independent of if.

① void main ()

```
{ printf ("start");
  if (2<5){
        printf ("0");
          printf ("1");}
  printf ("2");
}
```

O/p : Start 012

②

```
printf ("1");
if (5<2)  false
    printf ("2"); not executed!
    printf ("3");
printf ("4");
```

O/p : 134.

③

```
int i = 2;
printf ("1");
if (i+2){ non zero (true).
   printf ("2");
   printf ("3");}
printf ("4");
}
```

O/p = 1234

④

```
printf ("He")
  if (2) non zero (true)
{ printf ("prabhu");
   printf ("bacha lo");}
printf ("Iss Rawan se");
```

O/p : He prabhu bacha lo Iss Rawan se.

if (!2)
↓
0
(false)

→ if (-2) non zero

```
{ . . . } ∴ true
```

if (printf ("GATE")) value = 4 ∴ true

```
{ . . . . }
```

```
if (! printf ("Pankaj"))   !6 = 0
{ printf ("sharma"); }      (false
```

$\underbrace{\text{printf ("Pankaj")}}_{6}$

O/p = Pankaj

```
if (condition/expression)
  { . . . . }  if non zero (true)
  else {
  . . . . }    optional!
```

if false

Associate an alternate block of code when if condition is false.

```
if (!2)          -0 (false)
      printf ("Hello");
else if (2+3-5)  -0 (false)
      printf ("Dance");
else
      printf ("ye Rawan hai");
```

false
false

Given 3 distinct integers
Get the maximum number

```
int a,b,c,max;

max = (a>b && a>c) ? a :
        b>c ? b : c;

printf ("%d", max);
```

Iterative Statements

(i) **For loop** :

```
for (initialisation; condition; Inc/dec)
{
    code; }
```

① → ② → ④

③ if true

again goes to 4th expression & condition is checked
if false, breaks out of loop

if false

```
for (expression 1; expression 2; expression 3)
{
    ..... code;

    .....  }
```

① → ② → (true) ③ → ④

(false)

→ if no ~~semicolon~~ { }, scope will be available till first statement

eg:

```
for (int i=1; i<=5; ++i)
{ printf ("%d", i); }
```

O/p : 12345.

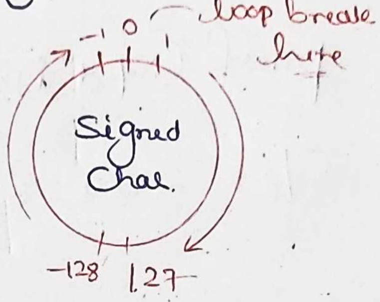
```
int i=1;

for (i=10; i<=5; 10)
{
    printf ("%d", i);
    i=i+2; }
```

increment

O/p = 135.

(1)
```
Char Ch = 1;
for (10; Ch; Ch++)
{ printf("Hello"); }
```
255 times

How many times printf will execute?



⇒ 255 times

(2)
```
Char Ch = 1;
for (10; Ch; Ch += 2)
{ printf ("Hello"); }
```
Infinite loop. as it dosen't land on 0.

(3)
```
int i = 1;
for (printf("1"); i <= 4; printf("4"))
{ printf ("2");
  i = i + 1; }
```
O/p = 124242424

(4)
```
for (1; 2; 3)
{ printf ("Hello"); }
```
∞ loop

(5)
```
for (1; 0; 1)
{ printf ("Hello"); }
```
0 → false

0 times

(6)
```
int i = 1;
for ( ; i <= 5; i++)
{ printf ("Hello"); }
```
Prints 5 times "Hello".

* All three expressions are optional, but semi colon mandatory.

(7)
```
for ( ; ; )
{ printf ("Hello"); }
```
→ non zero (true)

∞ loop

⑧ int i=1;

```
for( ; i++<6 ; )
{ printf("%d",i);}
```

post increment

O/P = 23456

Loop Analysis

1) `for(int i=0; i<n; ++i)`
`{ printf("Pankaj"); }` → loop will run "n" times.

2) `for(int i=1; i≤n; i=i+2)`
`{ ..... }` → loop will run $\lceil \frac{n}{2} \rceil$ times

3) `for(int i=1; i≤n; i=i*2)`
`{ ..... }` → $\lfloor \log_2 N \rfloor + 1$ times

4) `for(int i=1; i≤n; i=i*3)`
`{ ..... }` → $\lfloor \log_3 N \rfloor + 1$ times

5) `for(int i=1; i≤n; ++i)`

```
{ for(int j=1; j≤n; ++j)
  { printf("Hello"); }
}
```

inner loop: n times
outer loop: n times

$\Rightarrow n \times n = n^2$ times

6> for (int i=1; i≤n; ++i)

{ for (int j=1; j≤n; j=j*2)

{ pf ("pankaj"); }

}

$\lfloor \log_2 n \rfloor + 1$, n

$= n(\lfloor \log_2 n \rfloor + 1)$

7> for (int i=1; i≤n; i=i+2)

{ for (int j=1; j≤n; j=j*2;)

{ printf ("pankaj"); }

}

$\lfloor \log_2 n \rfloor + 1$, $\lceil \frac{n}{2} \rceil$

$\Longrightarrow \lceil \frac{n}{2} \rceil (1 + \lfloor \log_2 n \rfloor)$

8> for (int i=1; i≤n; ++i)

{ for (int j=1; j≤i; j++)

{ printf ("pankaj"); }

}

i=1	i=2	i=n
j=1	j=1,2	j=1,2...n

$\Rightarrow \frac{n(n+1)}{2}$

9> for (int i=1; i≤n; ++i)

{ for (int j=i; j≤3*i; ++j)

{ ...... }

}

i=1	i=2	......
j=1 to 3	j=2 to 6	
3 times	5 times	......

i=n

j = n to 3n. = 3n-n+1

(2n-1) times

$[3+5+7 ... +(2n+1)]$ = AP

$= \underline{S_n} = \frac{n}{2}(2a+(n-1)d) = \frac{n}{2}(3+(2n+1))$

$= \underline{n(n+2)}$

(ii) While Loop

while (condition/expression)
if false/0 { ....Code } if true!
.... }
exit from loop.

loops → for :- no of iterations are known in advance.
→ while :- when no of iterations are not known.

→ WAP to calculate the no. of digits in a number.

```
int c=0;
while (n!=0)
{ c++;        → count digit
  n = n/10; } → delete last digit
return c;
```

(iii) do-while loop :

```
int i=1;
do{ printf("pankaj");
  }
while (i>5);
```

* The code will execute the code in loop atleast one time irrespective of weather the condition is true or false.

O/P = pankaj.

Jump Statements

(i) Continue:

```
for ( ; ; ; )
{   S1;
    S2:
    Continue;
    S3;
    S4; }
```

Skip the remaining part of current iteration & Continue with next iteration.

```
for (int i=1; i<=10; ++i)
{  if (i%3 ==0)
      Continue;
   printf ("%d", i);
}
```

⇒ in this case below part is Skipped

O/p = 12457810

(ii) Break:

```
for (int i =1; i<=10; ++i)
{  if (i%3 ==0)
      break;
   Printf ("Hello");
}
```

⇒ when the condition is true, break is Called which breaks the loop when i=3.

(Hello, Hello.

```
for(int i=1; i<=5; ++i)
{ for (int j=1; j<=5; ++j)
  {  if ((i+j) % 4 ==0)
        break;
     printf ("%d %d", i, j):
  }
}
```

break; – breaks the inner loop

O/p = 111221414243 5152

```
for (int i=1; i<=n; i=i*3)
{
    for (int j=i; j<=n; j++)
    { printf("Hello"); }
}
```

$i=1$	$i=3^1$	... $i=3^k$
$j=1$ to n	$j=3$ to n	$j=3^k$ to n
$(n-1)+1$	$(n-3+1)$	$(n-3^k+1)$

$3^k \le \log n$

$\log 3^k \le \log n$

$k\log 3 \le \log n$

$k \le \frac{\log n}{\log 3}$

$k = \lfloor \log_3 n \rfloor$

$= (k+1)(n+1) - \frac{1(3^{k+1}-1)}{3-1}$

$= (k+1)(n+1) - \frac{(3^{k+1}-1)}{2}$

$= (\lfloor \log_3 n \rfloor + 1)(n+1) - \frac{(3^{\lfloor \log_3 n \rfloor + 1} - 1)}{2}$

```
for (int i=1; i<=n; ++i)
{   for (int j=1; j<=i*i; j++)
    {  for (k=1; k<=n; k++)
       { printf("pankaj"); }
    }
}
```

$\Rightarrow \sum_{i=1}^{n} \sum_{j=1}^{i^2} \left(\sum_{k=1}^{n} 1 \right)$

Switch Statement

* Switch is a keyword
* Used to provide selection statement with multiple choices.
* Multiple choices are provided using keyword : case

```
Switch (n)
{ Case (1) : Code to be executed when value of n is 1
             break;
  Case 2 :   Code to be executed when value of n is 2
             break;
  default:   Code to be executed when none of above
             cases match.
             break;
}
```

'A' — allowed (65) because it has a value ✓.

Switch (expression) → expression that evaluated to be int value.

```
{
   Case Constant1 :   Code1
                      break;
   Case Constant2 :   Code2;
                      break;
   Case Constant3 :   block of code
                      break;
        default:      Code;
                      break;
}
```

Case labels cannot be a variable, It can only contain constants.

break; → breaks the loop if matched. (Switch)

* Continue is only for loops not for switch.

* if break is not there then cases below will execute (default). — fall through will happen.

break; → takes out of the scope "break" is optional.

* duplicate case labels are not allowed.

eg:

```
i=2;
Switch (i+2)
{ case 2 : printf("2");
           break;
           printf("Hello"); → ignored
  Case 4 : printf("4");
}
```

Note:

* Break is optional
* Order of case label does not matter.
* Position of default does not matter.
* Default is optional.
* Switch (i); ✓
* Switch (i) { } ✓
* Switch ( ); × compiler error, expression is mandatory.
* Case labels must be constant or any expression
* Containing all constants.
* Duplicate case labels not allowed

eg: low, high code:

```
int i=3;
Switch(i)
.......
{ Case 1 ... 10 : printf("Hello");
                  break;
  Case 11 ... 20 : printf("World");
                   break;
}
```

* Not for all compilers

Any statement here is ignored.

Space

3 dots.

O/p : Hello

Functions and Storage Classes

*Compilation happens top to bottom.

```
#include<stdio.h>
int multiply (int, int);
void main()
{ int a=10, b=20, answer;
  answer = multiply(a,b); // call/use of function
  Printf("%d", answer);
}

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

int multiply (int, int); ⟶ forward declaration, to avoid C.E. as we are calling the function and defined later or below

int a=10, b=20, answer; — int type

if return type & int data not match. Error!

return type of function

int multiply (int x, int y) ⟶ function header

function definition

if not written anything, by default Compiler will set return type as "int", only in also in forward declaration!

eg:

```
void main()
{ int a= 10;
  double b;
  b= fun(a);
  Printf("%f", a);
  return 0;
}
```

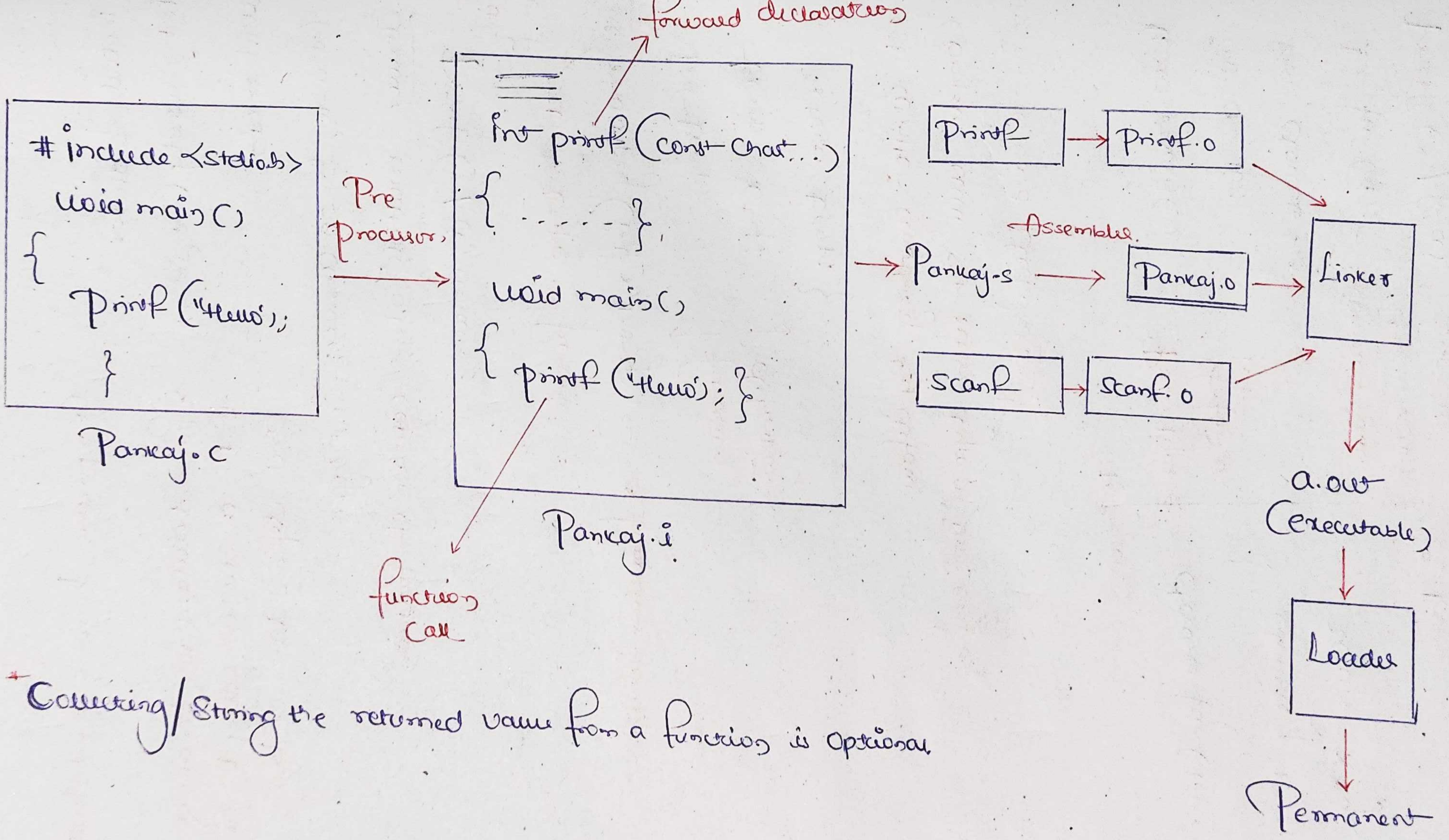
b= fun(a); ⟶ compiler saw info = return type is integer

Return type of fun is int ⟵ double fun (int x)

(mismatch)

```
double fun (int x)
{
  double y = 10.2;
  return x*y; }
```

∴ Error



*Collecting/Storing the returned value from a function is optional

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

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

→ function called but value returned by the function not stored,

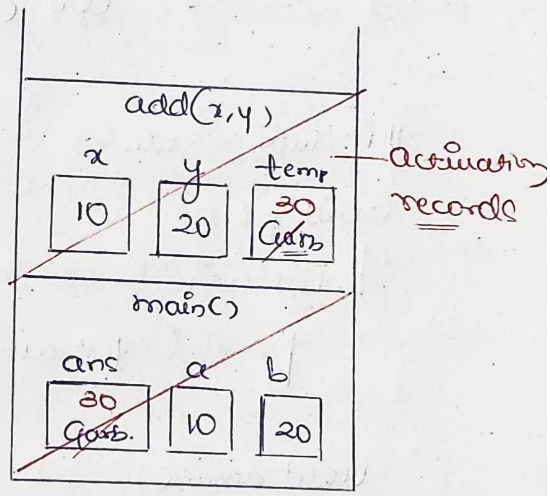
→ no Error

How a function works

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

O/P = 30

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



Function Call Stack

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

O/P = 10 20

swap(a,b); → actual arguments

O/P = 10 20

formal parameters

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

Called by values

(copy of the arguments passed will be here. So variables in main will not be changed.

Storage Class

Scope: part of the program in which a variable is visible.
It is the section of the code in which the variable can be accessed directly.

Lifetime: duration → alive

Storage area: where a variable is stored.

Default value: The value that is present in a variable when we don't initialise the variable

```
#include <stdio.h>
void f()                    → Start of scope
{ int a=10, b=20;           → local variables
  printf("%d",a+b);}        → end of scope

void main()
{  int x=10;
   f();  → function calling.
   printf("%d", x+a);}
                ↑
      Error, Cannot access local variable
      of another scope
```

Auto Variables

* By default, variable declared inside a function are auto variable
* Scope and block: block in which they are declared.
* Default value: Garbage
* Storage area: Stack

```
void main()
{ int a=0;
  ++a;
  printf("%d",a);     — O/P = 1
     { int a=10;                  → New
       ++a;
       printf("%d",a); }          → Scope (sub scope)
                         O/P : 11
  ++a;
  printf("%d",a);
           OP = 2                 Overall Output : 1112.
}
```

* Main variable Scope variable can be accessed to sub-scope but not vice versa

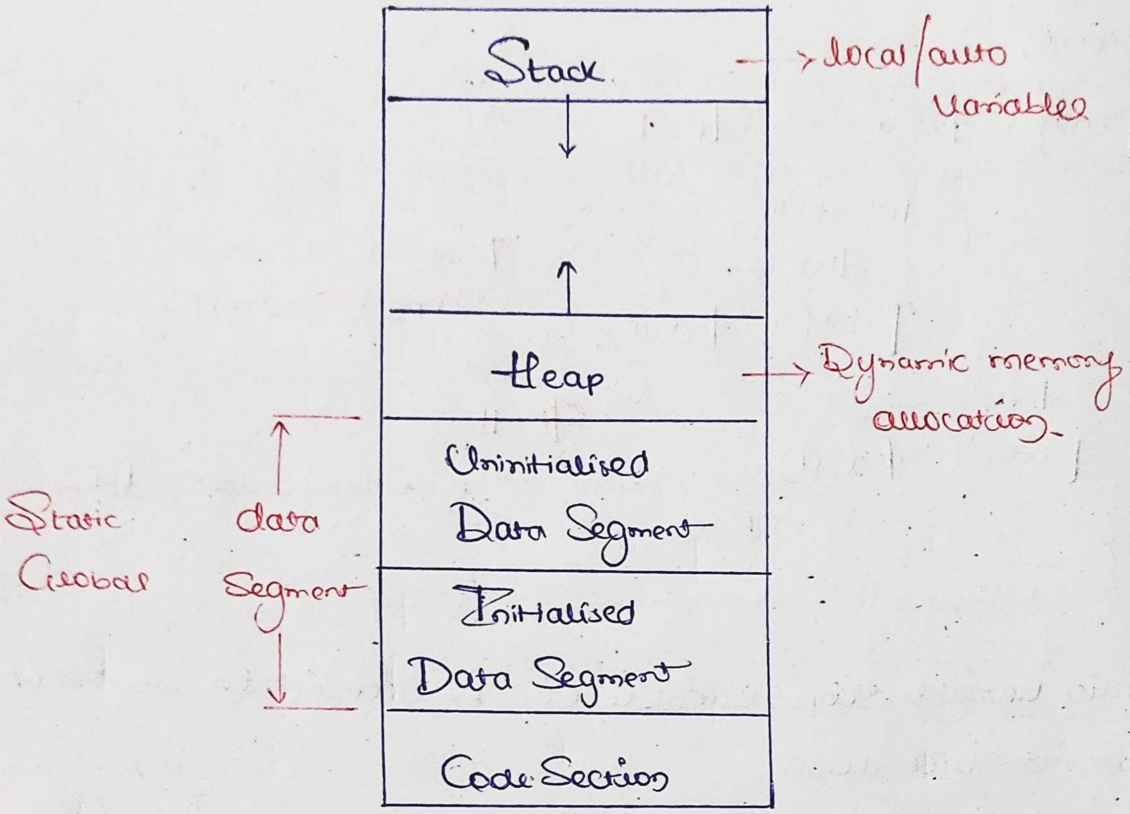
* Sub-scope variables are not accessible to main scope

* Auto variables are created automatically when we enter the block in which they are declared and destroyed automatically when we exit the block.

eg: #include<stdio.h>

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

void main()
{  f();                 Output = 111
   f();
   f(); }
```

Memory Diagram

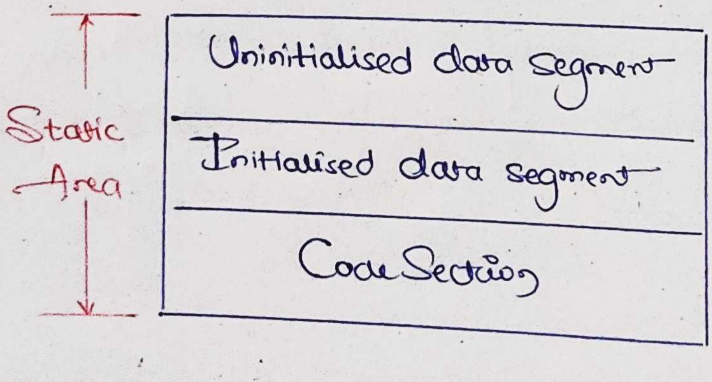


Register

* Same as auto variable
* Storage area : CPU register/stack

Static Variables

* Scope : block in which they are Created
* lifetime : lifetime of program
* default value = 0
* Storage area : Static area (data segment)



```c
#include<stdio.h>
void fun()
{ static int i=0;   // this line is executed only once
  ++i;              // updates the last updated value.
  printf("%d",i); }
void main()
{ fun(); -1
  fun(); -2
  fun(); } -3
```

O/P = 1 2 3.

* No redeclarations
* Value present b/w different function call
* They are created only once in the program.

Global Variables

* By default a variable defined outside all the functions is called as global variable.
* Global variables can be accessed inside every function if the variable is declared above the function definition.

```c
void fun()
{ ++x;                 // Compilation error.
  printf("%d",x); }
int x=0;  <- global variable
void main()
{ ++x;
  fun();
  printf("%d",x); }
```

```c
void fun()
{ extern int x;   -> local forward declaration for global variable.
  ++x;
  printf("%d",x); }
int x=0;
void main()
{ ++x;
  fun();
  printf("%d",x); }
```

* extern is only applicable on global variable

```c
extern int x;  -> global forward declaration
void fun(){...}
void main(){...}
int x=0;  -> global variable definition
```

Internal linkage

```
Static int i=10;
void f(){ }
void g(){ }
void main()
{ }
```

External linkage

```
int i=10;
void f(){ }
void g(){ }
void main()
{ }
```

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

Pankaj.c

Pre Processing →

forward declarations

```
extern int printf(...);

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

Pf-definition

Linker

Pf - f.d

Pf - call

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is a recursive function.

```
{ if(n is small)
  { - Can be answered directly.
    - no recursion needed.
    - easy problem. }
  else { - Cannot be answered directly.
         - not easy
         - recursion needed }
}
```

eg: Print "pankaj" n times using recursion.

```
void Print(int n)
{  if (n==1) printf("Pankaj");
   else {  printf("Pankaj");
           Print(n-1); }
}
```

↓ Recursively call for smaller value.

eg: Sum of digits using recursion.

```
int sum_digits (int n)
{   if (n>0 && n<=9)
        return n;
    else { return (n%10) + sum_digits (n/10); }
}
```

eg: 125

125 % 10 = 5 + Sum_digit(12) ← 3 ; 125/10 = 12 ; 8 → 8 ✓

12 % 10 = 2 + Sum_d (12/10) → return 1

eg: a^b $a>0$, $b\geq 0$

```
int Power (int a, int b)
{   if (b==0)     → base condition
        return 1;
    else { return a × Power (a, b-1); }
}
```

a = 3
b = 3.

27

3 × Power (3,2) ← 9

↓

3 × Power (3,1) ← 3

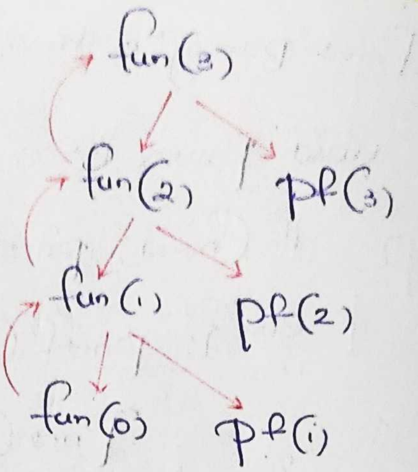
↓

3 × Power (3,0) ← 1.

↓

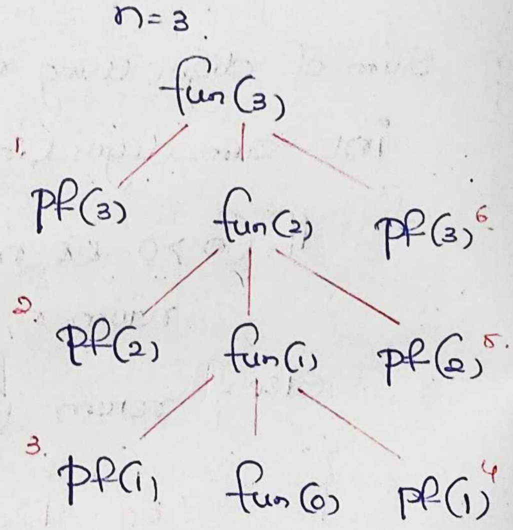
return 1.

eg:
```
void fun(int n)
{ if(n<=0) return;
  else{ fun(n-1);
        printf("%d",n);}
}
```
If they are reversed O/P = 321.

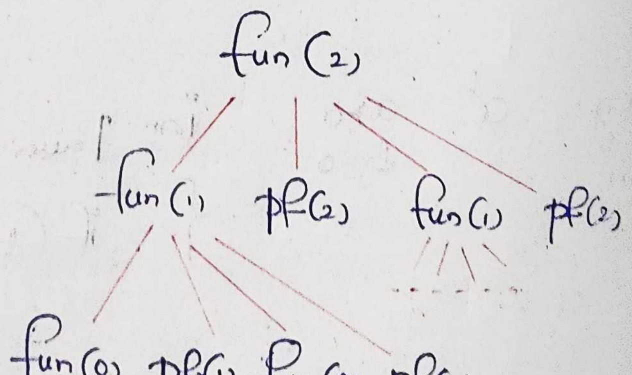


fun(3) = 123.

eg:
```
void fun(int n)
{ if (n<=0)
      return;
  else{ printf("%d",n);
        fun(n-1);
        printf("%d",n);}
}
```
O/P = 321123.



eg:
```
void fun(int n)
{ if(n<=0)
      return;
  else{ fun(n-1);
        printf("%d",n);
        fun(n-1);
        printf("%d",n);
  }
}
```
O/P = 112112



①
```c
void f(int n)
{ if (n<=0)
        return;
  printf("%d", n);
  f(n-1); }
```
What is O/p of f(5)?

→ 54321.

②
```c
void f(int n)
{ if (n<=0)
        return;
  f(n-1);
  printf("%d", n); }
```
What is the Output of f(5)?

→ 12345

③
```c
void f(int n)
{ if (n<=0) return;
  f(n-1);
  printf("%d", n);
  f(n-1); }
```
What is O/p of f(4)?

→ 121312141213121

④
```c
int f(int n)
{ if (n<=1)
        return 0;
  return f(n/2)+f(n/2)+1; }
```
What is the output of f(5)?

→ 7

⑤
```c
int f(int n)
{ if (n<=1)
        return n;
  return f(n/2)+n/2; }
```
What is O/p of f(12)?

→ 11.

⑥
```c
int f(int n)
{ if (n<=1) return n;
  if (n%2)
    return f(n/2)+n;
  return f(n/3)+n;
}
```
What is the Output of f(22)?

→

⑦ //Assume n≥0

```
void fun(int n)
{  if (n==0) return;
   fun (n/2);
   Printf ("%d", n%2);
}
```

What is O/p of f(11)?

→ 1011

⑧

```
void foo (int n, int sum)
{ int k=0, j=0;
  if (n==0) return;
  k=n%10;
  j= n/10;
  Sum= Sum+k;
  foo (j, sum);
  Printf ("%d", k); }

void main()
{ int a=2018, Sum=0;
  foo (a, sum);
  printf (" %d", sum); }
```

Output is 20480

⑨

```
void main ()
{ Static int var=5;
  Printf ("%d", var--);
  if (var)
     main(); }
```

⑩

```
void main()
{ Static int i=5;
  if (--i)
  { main();
    printf("%d", i); }
}
```

O/p - 0000

⑪

```
int fun (int a, int b)
{ if (b==0) return 0;
  if (b%2==0)
     return fun (a+a, b/2);
  return fun (a+a, b/2)+a;
}
int main ()
{ printf ("%d", fun (4,3));
  return 0; }
```

Output : 12

(12)
```
int f (int n)
{ static int r=0;
    if (n<=0) return 1;
  if (n<3)
  { r=n;
     return f(n-2)+2; }
  return f(n-1)+r; }
```
What is the value of f(5)?

→ 9

(13)
```
unsigned int foo (unsigned int n,
unsigned int r)
{ if (n>0)
      return (n%r) + foo (n/r, r);
  else
   return 0; }
```
Output of foo (513, 2)?

→ 2.

(14)
```
int fun (int x)
{ if (x>3)
  return fun (x-4) + fun (x-1)+1;
  return 1; }
```
What is the value returned by fun (12)?

→ 51

(15)
```
int fun (int n)
{ if (n==4) return n;
  else return 2*fun (n+1); }

int main()
{ printf ("%d", fun (2));
  return 0; }
```
→ 16

(16)
```
int fun (int x, int y)
{ if (x==0)
     return y;
 return fun (x-1, x+y); }
```
What is the output of fun (4,3)?

→ 13

(17)
```
int fun (int x, int y)
{ if (y==0) return 0;
  return (x + fun (x, y-1)); }
```
What does the following function do?

→ $x*y$.

(18)
```c
int fun (int x, int y)
{ if (y==0) return 0;
  return (x + fun (x, y-1));
}

int fun2 (int a, int b)
{ if (b==0) return 1;
  return fun (a, fun2(a, b-1));
}
```
What does fun2() do?

→ pow (x, y);

(19)
```c
#include <stdio.h>
void print (int n)
{ if (n>4000) return;
  printf("%d", n);
  print (2*n);
  printf ("%d", n); }

int main()
{ print (1000);
  getchar();
  return 0; }
```
What is the output?

→ 1000 2000 4000 4000 2000 1000

(20) What does the following function do?
```c
int fun (unsigned int n)
{ if (n==0 || n==1) return n;
  if (n%3 != 0) return 0;
  return fun(n/3); }
```
→ It returns 1 when n is a power of 3, otherwise returns 0.

(21) Predict the output of the following program.
```c
#include <stdio.h>
int f(int n)
{ if(n<=1) return 1;
  if (n%2==0) return f(n/2);
  return f(n/2) + f(n/2+1); }

int main()
{ printf("%d", f(11));
  return 0; }
```
→ 5

(22) int f(int n)

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

Value returned by f(1) is?

→ 7

(23) void Count(int n)

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

int main()
{ count(3); }
```

→ 312213444

(24) Consider the following recursive C function. If get(6) function is being Called in main() then how many times will the get() function be invoked before ~~running~~ returning to the main()

```
void get(int n)
{ if (n<1) return;
    get(n-1);
    get(n-3);
   printf("%d",n); }
```

⟹ 25 (Ans)

(25) What will be the output of the following C program?

```
#include <stdio.h>
int main()
{ main(); return 0;}
```

→ Runtime Error

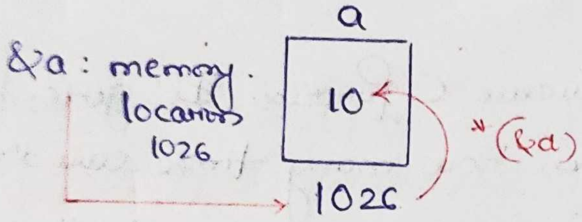
Arrays and Pointers

Address → Absolute address
Address → Relative address

→ * : value at operator

→ & : address of operator

int a = 10;

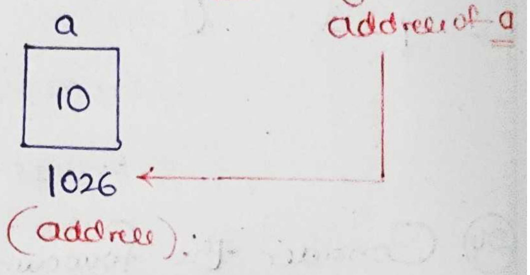


How to go address of any variable?

int a = 10;

→ & : address of operator

&a → gives the address of a



*(&a) = value at (memory location 1026) = 10

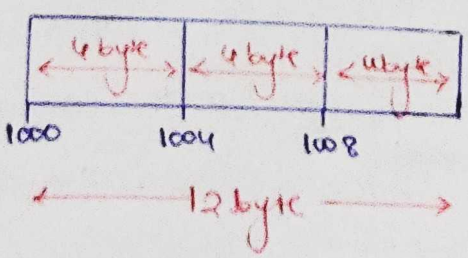
*(&a) = a

Array: Array can be defined as a group of multiple entities of similar type into a larger group. These entities can be int, float, double, char or can be user defined data types also.

eg: int a, b, c, d, e ⇒ int b[5];

if int = 4 byte

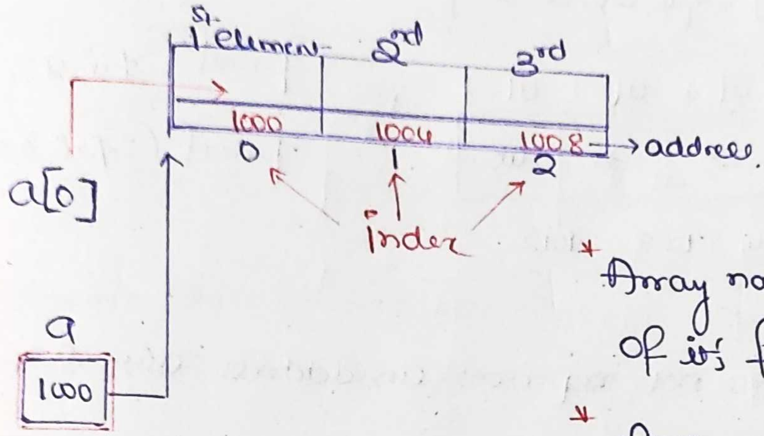
int arr[3]



* All elements are stored sequentially and represented by same name "arr".

* In C, index always starts from 0.

int a[3];



* Array name represent address of it's first element.

* Array name → Constant address.

++ array_name
-- array_name
array_name ++;
array_name --

all are invalid because array name is a constant address.

Array_name = { } ×

Invalid because l-value cannot be a constant.

```
void main()
{ int a[4];
Printf("%d", a[2]);}
```

a[0]	a[1]	a[2]	a[3]
G	G	G	G

G → Garbage value

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

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

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

10	20	0	0

int a = {10}; ✓ valid

int a[10]; ✓

int a [2×3+10]; ✓

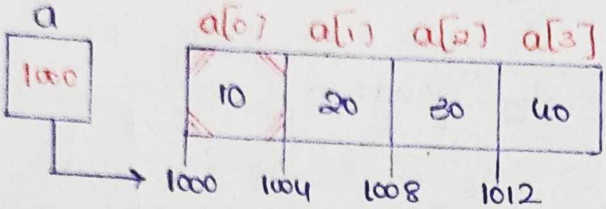
int a [2× sizeof(int)] ✓

int a[ ]; × Invalid

int a [ ] = {10, 20, 30}; ✓ valid

(1.) Array-name represents address of its first element.

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



1000 { Printf ("%u", a);
Printf ("%u", &a[0]);

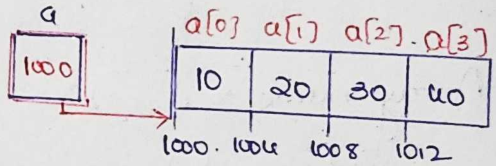
(2.) Array name does not represent an address with 2 operators

(i) & : address of Operator

(ii) Sizeof() Operator

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

address of element of 4 Byte. { Printf ("%d", a);
Printf ("%d", &a[0]);



Printf ("%d", &a);

↓

address of whole array, 16 byte ka address.

Printf ("%d", &a[0]);
Printf ("%d", &a); } Numerical value is same but logically we are taking about different elements address.

(3.) Numerical value of a is 100

$\boxed{a+1}$

a is simple var. / value

a is an address / pointer variable

int a=100;
pf ("%d", a+1);
= 101.

address arithmetic.

address + value = address ✓
address + address = invalid ✗

(4) If the declaration (Initialization) of an array has n-dimension.

int a[4]; $\longrightarrow$ 1 dimension

int a [2] [3]; $\longrightarrow$ 2 dimension

int a[2][3][4]; $\longrightarrow$ 3 dimension

* Anywhere other than declaration if we ~~are~~ provide exactly n-dimension then we are working on element.

* If we are providing less than n-dimension then we are working on addresses

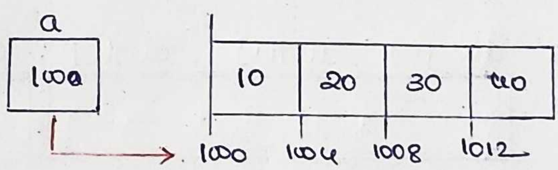
eg: int a [2] [3]; $\rightarrow$ dimension

a; $\rightarrow$ 0 dimension $\rightarrow$ address

a[0]; $\rightarrow$ 1 dimension $\rightarrow$ address

a[0] [0]; $\rightarrow$ 2 dimension $\rightarrow$ Element

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



printf ("%u", a+1);

a $\rightarrow$ address
$\downarrow$
arrayname = &a[0]
$\downarrow$
Size = 4 byte.

$\rightarrow$ a+1 = &a[0] + 1
= &a[0] + (1×4)
= $\underline{1004}$ (address of next element)

a+2 = &a[0] + (2×4)
= 1000 + 8
= $\underline{1008}$

$\rightarrow$ (a+2) = memory location of = &a[2];
1008

$\boxed{*(a+i) = a[i]}$

$\rightarrow$ *(a+2) = value at (memory loc : 1008) = ~~*&~~a [2];
*(a+2) = a[2]

→ Addition is commutative

$$\boxed{a[1] = *(a+1) = *(1+a) = 1[a]}$$

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

10	20	30	40
1000	1004	1008	1012

printf("%d", a[2]);
printf("%d", *(a+2));
printf("%d", 2[a]);
printf("%d", *(2+a));

Output = 30

Printf("%d", a+1); → 1004

Printf("%d", &a+1); → &a+1 = 1000 + (1×16) = 1016

Size of array = 16B

address of whole array

2-D Array

int a[2][3]

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

← a[0] → ← a[1] →

← a →

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

1	2	3	4	5	6
1000	1004	1008	1012	1016	1020

→ a is an array of 2 elements : a[0], a[1]

→ a : array-name, address of its first element &a[0]

→ a[0] : an array of 3 elements : a[0][0], a[0][1], a[0][2]
it is address of its first element &a[0][0]

* Assume address start from 1000.

```
void main()
eg: {
    int a[2][3] = {1, 2, 3, 4, 5, 6};
    printf("%d", a+1);     -> &a[0]+1 = 1000+(1x12) = 1012
    printf("%d", a[0]+1);  -> &a[0][0]+1 = 1000+1x4 = 1004
    printf("%d", &a+1);    -> &a+1 = 1000+1x24 = 1024
}
```

&a → address of three ele, whole array

eg:

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

int a[3][4] = {1, 2, 3, 4 | 5, 6, 7, 8 | 9, 10, 11, 12} Starting address = 100

```
printf("%d", a);
printf("%d", a[0]);     } 100
printf("%d", &a);
```

size of a[0] = 16 Byte

```
printf("%d", a+1);      -> 100+(1x16) = 116
printf("%d", a[0]+1);   -> 100+(1x4) = 104
printf("%d", &a+1);     -> &a+1 = 100+(1x48) = 148
```

→ $a[0]+1 = \underbrace{\&a[0][0]+1}_{4\text{ Byte}}$

$= 100 + (1\times4)$

$= 104$

$= a[0][1]$

$\boxed{\rightarrow a[0]+2 = \&a[0][2]}$

→ $*(a[0]+1)$ = value (memory location) $= *\&a[0][1]$

$= a[0][1]$

→ $*(a[1]+2) = a[1][2];$

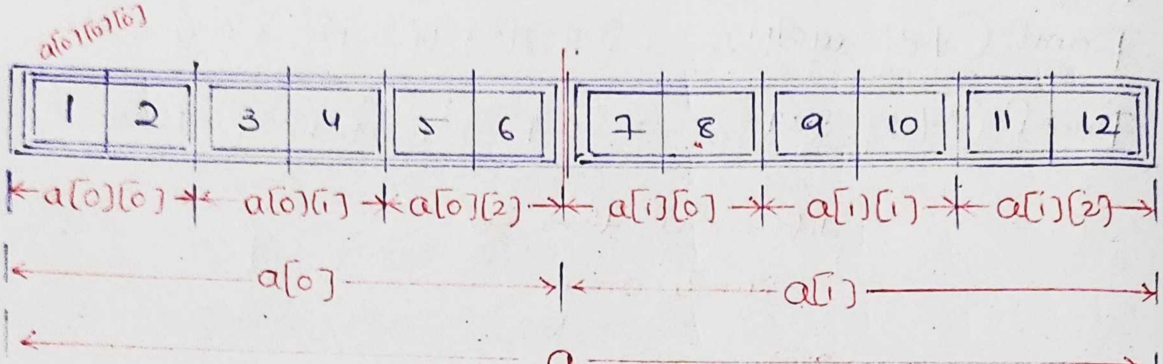
$\boxed{*(a[i]+j) = a[i][j]}$

$\boxed{\begin{array}{l} *(a[i]+j) = a[i][j] \\ *(*(a+i)+j) = a[i][j] \end{array}}$

```
a[0] <- printf("%d", *a);      -> 100
a[0][0] <- printf("%d", **a);  -> 1
printf("%d", *a+1);            -> 104
printf("%d", **a+1);           -> 2
printf("%d", *a[0]);           -> 1
```

3-D-Array

int a[2][3][2] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};



a: array a[0], a[1]

a[0] : array a[0][0], a[0][1], a[0][2]

a[0][0] : array a[0][0][0], a[0][0][1]

printf("%d", a+1); → &a[0]+1 = 1000+(1×24) = 1024

printf("%d", a[0]+1); → 1008

printf("%d", a[0][0]+1); → 1004

printf("%d", &a+1); → 1048

Q1) int a[3][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}

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

} 1000

5. a[0][0][0] — 1
6. a+1 → 1000+24 = 1024
7. a[0]+1 → 1000+12 = 1012
8. a[0][0]+1 → 1004
9. &a+1 → 1000+72 = 1072
10. a[0][0][0]+1 → 2.
11. *a → *&a[0] → 1000
12. **a → *(&a[0][0]) → 1000
13. ***a → *(&a[0][0][0]) = 1
14. *a+1 → &a[0][0]+1 = 1012
15. **a+1 → &a[0][0][0]+1 = 1004
16. *a[0]+1 → *(&a[0][0])+1 = 1004
17. **a[0]+1 → *(*a[0]+1)
= a[0][0][0]+1
= 1+1
= 2

Array declaration & initialization

- int a[]; × Invalid
- int a[] = {10, 20, 30}; ✓ valid
- int a[10]; ✓
- int a[4] = {10, 20, 30, 40}; ✓

→ If we are declaring an array without initialization, then it is mandatory to provide the size of each dimension, otherwise compiler Error.

→ In case we are initializing an array, then we have the flexibility that we can omit first dimension size but no other dimension is having such flexibility.

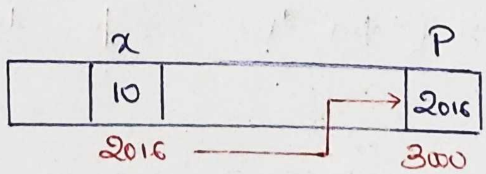
int a[] = {10, 20, 30}; ✓ valid
int a[][3] = {1, 2, 3, 4} ✓ valid
int a[][] = {1, 2, 3, 4} × invalid
int a[2][] = {1, 2, 3, 4, 5, 6} × invalid.

Pointers

A pointer variable is a special variable that is used to hold the address of another variable

int *p; → p is a pointer to integer
→ p can store address of some integer variable

eg: int x = 10;
int *p;
P = &x;



Printf("%d", &x);
Printf("%d", p); } 2016
Printf("%d", *p); → 10

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

↓
Pointer to a pointer of integer type

x		p		q
10		1016		2000
1016		2000		3000

```
printf("%d", p);   → 1016
printf("%d", *p);  → 10
printf("%d", q);   → 2000
printf("%d", *q);  → 1016
printf("%d", **q); → 10
```

Assume

int - 4byte	int *p;	Pointers are be of same size.
Char - 1byte	Char *q;	
float - 8byte	float *r;	

eg:
```
int x = 300;
Char *p;
p = (char*) &x;
printf("%d", *p);
```
O/P = -112

00000000	00000000	0000 0001	10010000

↑ P

$\overset{7}{2}\ \overset{6}{2} \ldots \overset{0}{2}$

10010000

$= -2^6 - 2^5 - 2^3 - 2^2 - 2^1 - 2^0$

$= -112$

eg:
```
int x = 130
Char *p;  p = (char*)&x;
printf("%d", *p);
```
O/p = -126 (Cyllic property)

```
int a[4] = {10, 20, 30, 40};
int *p;
p = &a[0];
p = p+1;  — 100 + (1x4) = 104
p++;  } inc & dec
--p;  } valid ✓
```

a
100

10	20	30	40
100	104	108	112

100 ↑ P → 104 ↑ p++.

Arrays and Pointers

```
int a[4] = {10, 20, 30, 40};
int *p;
p = a;  → p = &a[0];
    printf("%d", *p+0);  — 10        printf("%d", p[0]);  — 10
    printf("%d", *(p+1)); — 20       printf("%d", p[1]);  — 20
    printf("%d", *(p+2)); — 30
```

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

```
int a[4] = {10, 20, 30, 40};
  ++a;
  --a;     Invalid.
  a++;
  a--;

 a = {30, 40}; ×  Invalid
```

Array name cannot be L-value.

```
int a[4] = {10, 20, 30, 40}
int *p;
p = &a[0];
  ++p;
  --p;     valid.
  p++;
  p--;

p = &a[2]; ✓ valid
```

variable

→ pointer_var + 3 = moving 3 locations in forward direction.

```
int a[5] = {10, 20, 30, 40, 50}
int *p, *q;
p = &a[1];
q = &a[4];
```

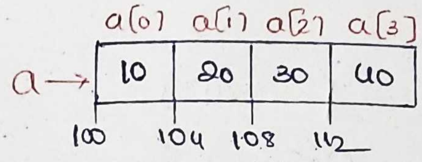
10	20	30	40	50
100	104	108	112	116

p → 104, q → 112

```
printf("%d", q-p); ✓
```

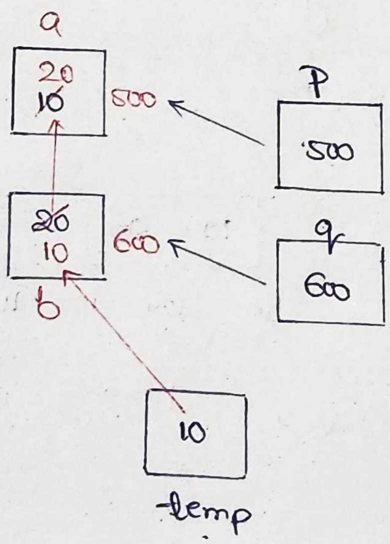
$\frac{\text{actual diff}}{\text{data size}} = \frac{116-104}{4} = \frac{12}{4} = 3.$

```
#include<stdio.h>
void main()
{ int a[4]={10,20,30,40};
  int *p;
  p=&a[0];
  printf("%d", *++p);   -> *(++p) = 20
  ++*p;   - ++(*p) = 20+1 = 21.
  ++p;   - 108 (address)
  printf("%d", *++p);}
                ↓
                30.
```



Pass by reference / call by reference

```
void main()
{ int a=10, b=20;
  printf("a=%d, b=%d",a,b);
  swap(&a,&b);      O/p : a=10, b=20
  printf("%d %d",a,b);
}                   O/p = 20,10

void swap(int *p, int *q)
{ int temp;
  temp = *p;
  *p = *q;
  *q = temp;}
```



```
void main()
{  int a[4] = {10, 20, 30, 40};
   int sum;
   sum = fun(a, 4);
   printf("%d", sum); }
```

O/p = 100

```
int fun(int *p, int n)
{  int i=0, sum=0;
   for(int i=0; i<n; ++i)
   {  sum = sum + p[i]; }
   return sum; }
```

Complex Declarations

1. ( ) ⟶ Parantheeis — function] 1 — L to R
2. [ ] ⟶ square bracket — Array]
3. Identifier ⟶ var name, fun_name] 2 — R to L
4. * ⟶ pointer]
5. datatype ⟶ int, char, float ⟶ 3.

eg: int * (p[3])   (③ int, ② *, ① p[3])

⇒ P is an array of 3 pointer to integer.

P | Add | Add | Add | ⟶ Integer.
P[0] P[1] p[2]

eg: int (*p) [4];   (③ int, ① *p, ② [4])

⇒ P is a pointer to "array of 4 integer".

```
int a[4] = {10, 20, 30, 40};
int (*p) [4];
p = &a; ✓
```

eg: int (*p) (int, int);

→ P is a pointer to a function that takes 2 integer arguments and return an integer value.

eg: int *(*p) [4];

→ P is a pointer to an array of 4 pointer to integer

Pointer to function/function pointers

```
int Add (int a, int b)
{ return a+b; }

void main()
{ int (*p) (int, int);
  P = &Add;
Printf("%d", (*P)(2,3); }
```

→ Pointer to a function which takes 2 int arguments & return integer.

→ Calling the function with pointer.

O/P = 5.

P=Add;	P=&Add;	P=Add;	P=&Add;
(P)(2,3)	(*P) (2,3)	(*P) (2,3)	(P)(2,3)

All are valid declarations/function calls.

(1.)

```
int a[4] = {10, 20, 30, 40};
int *p[4] = {a+2, a, a+1, a+3};
int **q;
q = &p[0];
printf("%d", *++*++q);
```

array of pointers

O/p 20.

++(++q)

↓

p[1]

*(p[1])++ = (&a[0]+1)

*(a[1]) = 20

②
```c
int a[4] = {10, 20, 30, 40};
int * p[4] = {a+2, a, a+1, a+3}
int **q;
q = & p[0];
*++*++q; --q;
printf("%d", *--*++q);
```
O/p = 10

③
```c
void fun (int*);
void main ()
{  int a[4] = {10, 20, 30, 40};
   fun (a);
   printf ("%d %d", a[0], a[1]);
}
```
O/p = 10 20

```c
void fun (int *p)
{  ++p;
   *p++;
   ++(*p); }
```

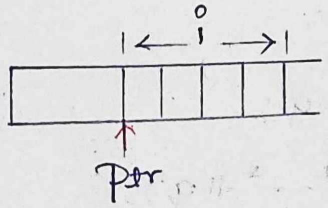
④
```c
void main ()
{  int a[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
   fun (a);
   printf ("%d %d %d", a[1][1], a[1][2], a[2],[0]); }

void fun (int (*p)[3])
{  ++p;
   (*p)[1] = (*p)[1]+1; }
```
O/p = 6 6 7

eg:
```c
void main()
{  int *p;   → declaration
   - - - -
   - - - -
   printf("%d", *p); }
```
↓ dereferencing (value fetch)

```c
char *p;
- - -
- - - -
- - - -
printf("%c", *p);
```
↓ 1 Byte info

Void Pointer

```
void *ptr;     → address
int i = 369;
char ch = 'A';
ptr = &i;
printf("%d", *ptr);  — Error.
printf("%d", *(int*)ptr);
```




```
void *p;
.....
p++;      }
p=p+1;    }  Error
```

* Don't try to dereference any void pointer without typecasting.
* Do not apply arithmatic operations on void pointer.

```
void *ptr;
char ch = 'A';
ptr = &ch;
printf("%c", *(char*)ptr);   O/p = A
```

Null Pointer

* Specially designed pointer
  → `int *p = (int*)0;`

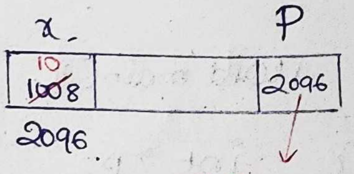
Wild Pointer

```
void main()
{  int *q;  }
```

↓ Garbage value

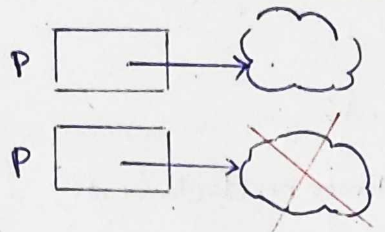
Uninitialised pointer is called as wild pointer.

```
void main()
{ int x = 1008;
  int *p;
  ====
  *p=10;
}
```



If by any chance Garbage value is 2096 then *p=10; will change the value of 'x'. (Unintentionally).

Dangling Pointer



int * f()
{ static int a=10;
return &a; } → never return address of a local variable

void main()
{ int *p;
p= f(); ← address of a local variable is returned but after the function is executed, the address is invalid
Printf("%d", *p); }
↓
error! ⇒ Solution is making 'a' static!
O/p = 10.

Dynamic Memory Allocation

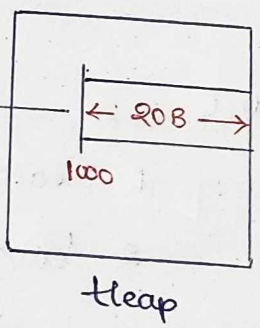
1. malloc
2. calloc
3. realloc
4. free

} Used for DMA in C.

1. malloc

malloc(20)
↑
Size in bytes.

Starting address is returned.



(void*) malloc (Size in bytes).

Syntax: (void*) malloc (size_t size)

Sizeof

int *p;
P = malloc (5 x sizeof(int));

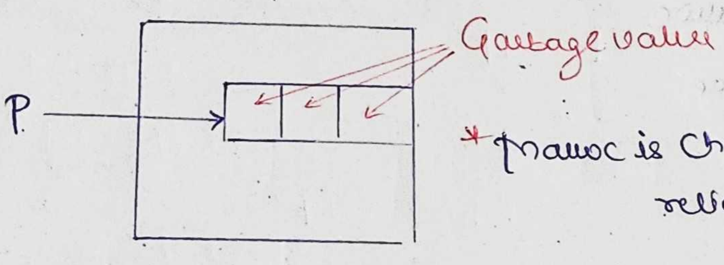
*(p+i) = p[i]

for(int i=0; i<5; ++i)
Scanf("%d", p+i); ← Storing elements
for (int i=0; i<5; ++i)
Printf("%d",*(p+i)); ← Printing value at loc.

```c
void main()
{   int N, *p, i;
    printf("Enter no of elements");
    scanf("%d", &N);
    P = malloc(N x sizeof(int));
  if(p!=NULL){
        for(int i=0; i<N; ++i)
                scanf("%d", p+i);
        for(int i=0; i<N; ++i)
                printf("%d", p[i]); }
}
```

malloc : 1. Search

ii. block is available → Starting address is returned.



* malloc is cheaper but not reliable.

② **Calloc** :

1. Search

2. block is available

→ Sets all bits = 0 (instead of garbage value)

3. Starting address is returned.

* Calloc is expensive and reliable

③ realloc

Size of the dynamically allocated memory can be changed by using realloc.

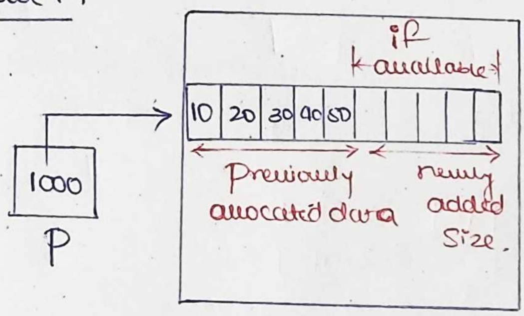
```
{ int *p;
  P = malloc (5x sizeof(int));
  ===
  P = realloc (p, 10x sizeof(int));
}
```

10x sizeof(int) → new size

```
P = malloc (5x sizeof (int));
===
P = realloc (p, 3x sizeof (int));
```

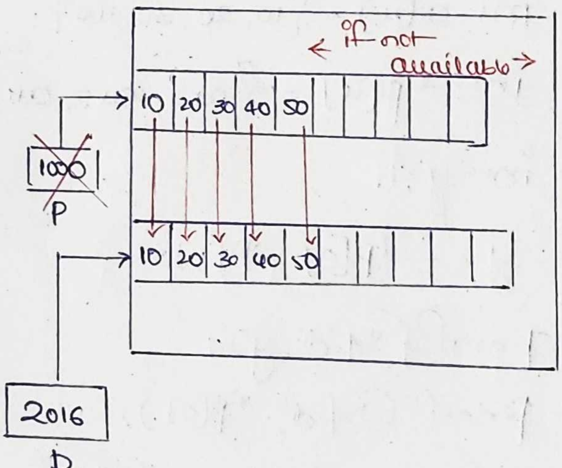
→ last 2 space is erased.

Case 1:



Heap

Case 2:



④ free

: It is used to free the dynamically allocated memory.

```
void f()
{ int *p;
P = malloc (1000 x sizeof(int));
===
} free (p);
void main()
{ f(); f(); f(); }
```

int *p → local variable

free (p); → deallocate the memory, Otherwise memory will not be available for other functions in heap.

* Memory allocated in heap will be there in heap throughout the lifetime of program.
* Not de-allocating the memory after use will cause "memory leakage problem", so we need to use "free()".

Gate PYQs

① `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);`

→ 2036 2036 2036

② BA = 1000, 4 byte

```
int a[2][3] = {1,2,3,4,5,6};
printf("%u %u %u", a, *a, **a);
printf("%u %u %u", a+1, *a+1, **a+1);
```

→ 1000 1000 1 1012 1004 5

③
```
int a[4] = {10, 20, 30, 40};
int *p[4] = {a+3, a+2, a+1, a};
int y;
y = --p[0] - p[1];
printf("%d", y);
printf("%d", *p[0]);
```

--p[0] = &a[3] - 1
= &a[2]

p[0] - p[1]
$= \frac{\&a[2] - \&a[2]}{4} = \frac{0}{4} = 0$

→ 0 30

*p[0] = *(a[2])
= 30

④ `int (*f) (int*);`

→ A pointer to a function that takes an integer pointer as argument and returns an integer.

⑤
```
int a=5, b=10, c=15;
int *p[3] = {&a, &b, &c};
printf("%d", *p[*p[1]-8]);
```

p[&b - 8]
= *p[10-8]
= *p[2]
= c

→ 15.

⑥ void 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 = &p[0]
  ptr++;   ptr = &p[1]
  printf("%d %d", ptr-p, **ptr); }
```

→ 1. 40

$\frac{\&p[1] - \&p[0]}{4}$

$= \frac{204 - 200}{4} = \frac{4}{4} = 1$

$**\&p[1]$

$*p[1] = *\&a[3]$

$= 40$

⑦

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

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

O/P → 12 22 14 7

⑧

```
int *A[10];
int B[10][10];
```

Of the following expressions which will not give CTE if used as L-value of an assignment statement.

→ A[2], A[2][2], B[2][2]

⑨

```
int a[4] = {10, 20, 30, 40};
int *p = (int*)(&a+1);
printf("%d %d", *(a+1), *(p-1));
```

→ 20 40

Strings

* String is a sequence of characters terminated by a null character
↓
\0
→ Ascii code = 0

Char arr[10] = "pankaj"; → String constant

arr = 100 → 100

p	a	n	k	a	j	\0			
0	1	2	3	4	5	6	7	8	9

printf("%s", arr); → pankaj.

* Size of the Character array should be atleast one more than length of string.

Char arr[] = {'p', 'a', 'n', 'k', 'a', 'j'};
printf("%s", arr);
} Behaviour is undefined

✓ {
Char arr[] = {'p', 'a', 'n', 'k', 'a', 'j', '\0'};
Printf("%s", arr);

1. Char name[] = "pankaj";
Printf("%s", name);
→ address.

* String gives address to printf()

2. Char name[] = {'p', 'a', 'n', 'k', 'a', 'j', '\0'};
printf("%s", name);

3. Char name[10] = "pankaj";
printf("%s", name);

Char name[] = "pankaj";

printf("%s", name+1);

→ ankaj

name = &name[0]

name+1 = &name[0]+1

= &name[1]

Char *ptr = "Neeraj";

Same { printf("%s", ptr); } Neeraj".
printf(ptr);

printf(ptr+1); — "eeraj".

Char arr[] = "Neeraj";	Char *ptr = "Neeraj"; → Read only area
arr = "pankaj". × Invalid,	ptr = "pankaj"; ✓
arr++; --arr arr-- ++arr; } Invalid	ptr++; ptr--; --ptr; ++ptr; } Valid

* We can change content or individual element of an array.

arr[1] = 'A';

As it is stored in read/write area.

→ ptr[1] = 'a'

Do not try this as pointers are stored in Read only area

Array.	Pointer
* We can change individual element.	We can't change individual element.
* Cannot assign completly new String.	Can assign completly new String
* Increment/dec not possible	Increment/decrement possible.
* Read/write area	Read only area

```
void main()
{   printf("Hello");   → address of 'H'
                          = Hello
    printf("Hello"+1); }
          ↓
     address of 'e'
        = ello
```

H	e	l	l	o	\0
100	101	....		...	105

eg:

```
if ("pankaj"[2] == "neeraj"[0])
      printf("yes"); ✓
else  printf("NO");
```

eg:

```
Char arr[] = "GATE";
  printf("%s", arr);  }  GATE
  printf("arr");      }
  printf(arr+1);  → ATE
  printf(arr+3-1); → TE
```

pf(arr + 3[arr] − arr[1])

arr + 'E' − 'A'

arr + $\cancel{x}$+4 − $\cancel{x}$

arr+4 = \0

∴ no output

eg:

```
Char arr[] = "GATE 2023";
Char *ptr;
ptr = arr+3;
ptr = '\0';  ——→  arr+3 = 0
                  ( arr[3] = 0.
printf(arr); → GAT.      = GAT \0 ....
printf(ptr); → no output        ↓
                          String ends here.
```

1. Char arr1[] = "pankaj"; } both have different address
   Char arr2[] = "pankaj"; }
   if(arr1 == arr2) printf("yes");
   else printf("NO");

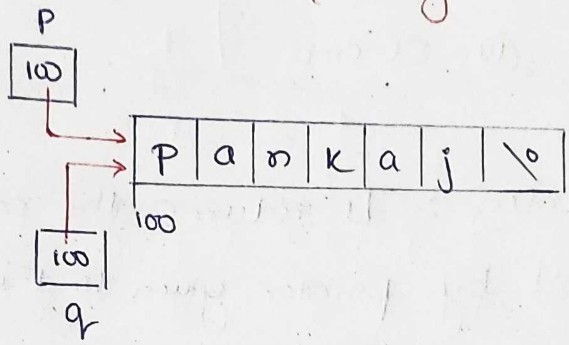
array name is address of first element

So output = NO

(2) Char arr1[] = "pankaj";
Char arr2[] = "pankaj";
if (*arr1 == *arr2) pf("yes");
else pf("NO");

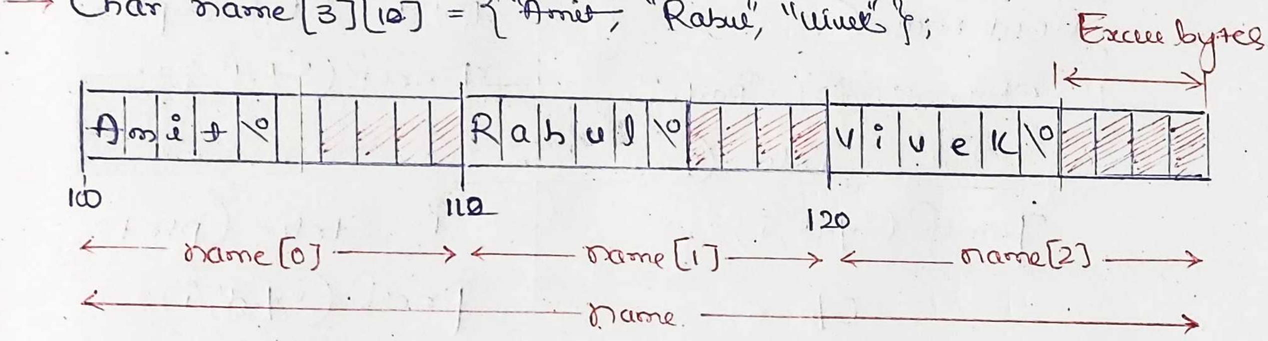
*arr1 = *&a[0]
= 'p'
*arr2 = *&arr2[0].
= 'p'
∴ Output = "yes".

(3.) Char *p = "pankaj";
Char *q = "pankaj";
if (p==q) pf("yes");
else printf("NO");



Multiple Strings

→ Char name[3][10] = {"Amit", "Rahul", "Vivek"};

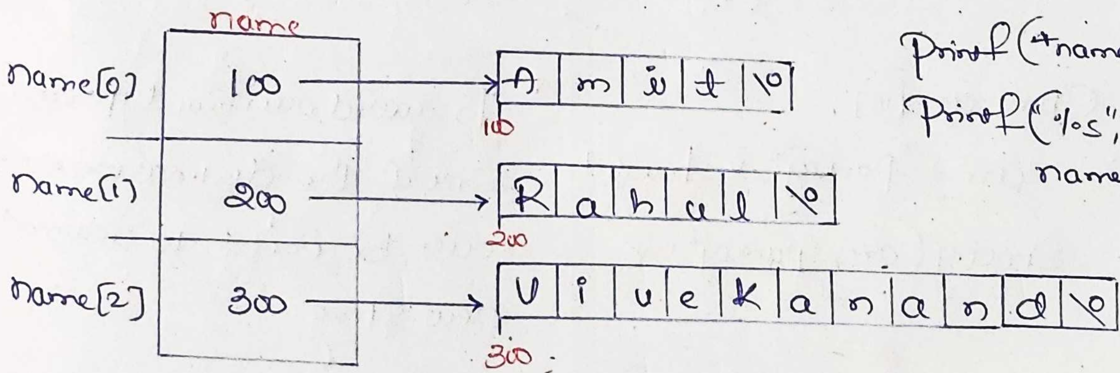


Printf("%s", name[0]); → Amit
printf("%s", name[0]+2); → it

*(name[1]+2)
↓
name[1][2] = 'h'

*Drawback with 2D array is, Some bytes will remain unused.

→ Char *name[3] = {"Amit", "Rahul", "Vivekanand"};



Printf(*name)
Printf("%s", name);
} Amit

<String.h>

(i) Strlen
(ii) Strcpy
(iii) Strcat
(iv) Strcmp

Inbuilt functions on strings

① **Strlen** : It returns the no. of symbols in the string pointed by pointer given to this function ('\0' is not counted).

→ Unsigned int strlen (char *p);

eg:
```
#include<string.h>
void main()
{ Char arr[10] ="pankaj";
  int i= Strlen (arr);
  Printf ("%d", i); }
```
O/p = 6

```
Char *ptr= "pankaj";
int i= strlen (ptr);
Printf ("%d", i);
```
→ 6

```
i= Strlen (ptr+2);
Printf ("%d", i);
```
→ 4.

② **Strcpy** : It copies the string pointed by the source pointer to the memory/buffer pointed by destination pointer. (also copy '\0').

→ Char *p strcpy (char* destination, const char* source);

eg:
```
Char arr[10];
arr= "pankaj"; × Invalid
Strcpy (arr, "pankaj"); ✓
```

* To avoid overflow type error size of the destination memory must be enough to accomodate new string.

eg: Char arr[10];

Char *ptr = "pankaj";

Strcpy (arr, ptr);

Printf ("%s", arr); → pankaj.

③ Strcat : It appends one string at the end of other string

→ Char* Strcat (Char *destination, Const Char *source);

It must be an array

It appends string pointed by source pointer at the end of string pointer by destination pointer.

eg: Char arr[20] = "pankaj";

Char *ptr = "Sharma";

Strcat (arr, ptr);

Printf ("%s", arr); → Pankaj Sharma.

④ Strcmp : It compares two strings character by character and returns 0 if both are same.

→ Strcmp (String1, String2);

Char* ptr1 = "pankaj";

Char* ptr2 = "pani";

$$\begin{cases} 0 \rightarrow \text{Same} \\ <0 \rightarrow \text{Smaller} \\ >0 \rightarrow \text{greater} \end{cases}$$

Strcmp(ptr1, ptr2))

'k' 'i'

k - i = +ve > 0.

Strcmp (ptr2, ptr1)

'i' 'k'

'i' - 'k' = -ve < 0.

Structures and Union

<u>Structure</u> : Collection of heterogeneous type / ~~different type~~ of data elements is called structure. It is a user defined data type.

Tag of structure

member of structure

Blueprint (no memory allocated)

```
Struct Student{ Char name [20];
                int roll; };
```

Struct is the keyword used to create user defined data type.

* When we create a structure, there will be new datatype like int, float, char.

→ Syntax to use : <u>Struct student _name_ ;</u>

<u>eg</u>:

```
Struct Student{ Char name[20];
                int roll; }
```

Structure blueprint/ template (Global)

```
void main()
{
     Struct Student S;
        S1.roll = 10; }.
```

. ⇒ membership Operator.

<u>Ways of making Structure</u>

(1)

```
Struct{ Char name[20];
        int roll; } S1,S2;
```

no tag.

variable to use.

```
void main()
{
```

we can't create a variable

```
}
```

(2)

```
typedef Struct Student
   { Char name [20];
     int roll; } x;
void main()
{
      x   S1;
```

or

```
      Struct Student S1;
}
```

③ Struct Student { Char name [20];
int roll; }

void f() {   └→ default value = 0 or null.

Struct Student s1;

⋮ }

void main()

{ Struct Student s2;   s2 = {10, "pankaj"}; × Invalid/Error, wrong order

⋮ };

s2 = {"pankaj", 10 }; ✓ Order matters (of datatype)

s3 = {"pankaj"}; ✓

s2. name = "pankaj"; × Invalid

↳ name is array, so L-value Can't be Constant.

Solution ↓

Struct Student s;

Strcpy (s. name, "pankaj"); ✓

s. roll = 10; ✓

eg: Struct student { Char name [20];
int roll; }

void main()

{ Struct Student s [2];

s[0]. roll = 10;   s[1]. roll = 20; ✓

s[0]. name = "pankaj"; × Invalid/Error.

Strcpy (s[0].name, "pankaj"); ✓

Strcpy (s[1].name, "Neeraj"); }

Struct Student s1 = s[0]; ✓ valid

| Structure student
s1 = {"4", 10};

Struct student
s2 = s1; ✓
valid

```
Struct date_of_birth { int day;
                       int month;
                       int year; }      } template

Struct Student { Char name[20];  → derived
                 int roll;        → primitive
                 Struct date_of_birth dob; }  → user defined data type
                        ↓
                 Structure inside Structure.
void main()
{
   Struct Student s;
   strcpy(s.name, "pankaj");
   s.roll = 20;
   s.dob.day = 2;
   s.dob.month = 5;
   s.dob.year = 1982; }
```

accessing member of dob

eg:

```
Struct Student {
    Char name[20];
    int roll; }

void main()
{  Struct Student s;
   Strcpy(s.name, "pankaj");
   s.roll = 10;
   display(s); }
        ( call by value
   display(&s);
        ( call by reference
```

```
void display(Struct Student t)
{ printf("%s", t.name);
  printf("%s", t.roll); }
```

```
void display(Struct Student *ptr)   ← pointer to structure.
{ printf("%s", (*ptr).name);
}   "Pankaj"
```

→ (*ptr).member1 → ptr is a pointer to structure & we can access member1 like this.

→ ptr: pointer to structure

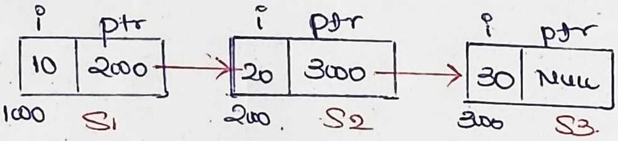
Syntax: (*ptr).member1 or ptr → member.

Self Referential Structure.

```
struct Pankaj{ int i;
       struct Pankaj *ptr; }
```

struct Pankaj *ptr → Member which is pointer to same data type.

```
void main().
{ struct Pankaj s1, s2, s3;
      s1.i = 10;
      s2.i = 20;
      s3.i = 30;
   s1.ptr = &s2;
   s2.ptr = &s3;
   s3.ptr = Null; }
```



Union: Union is a keyword used to create user defined data type. Similar to structure.

* In case of structure, all members get individual memory space but all members of a union variable share a common memory area.

Syntax: union my{ int a;
             char c; }.

```
Struct A
{ char i;      → 1 Byte
  int j; };    → 4 Byte

void main()
{ Struct A a;
  Printf("%d", sizeof(a)); }

O/P = 5
```

```
Union B
{ char i;      → 1
  int j; };    → 4          max(1,4) = 4

void main()
{ Printf("%d", sizeof(b)); }

O/P = 4.  (union B b;)
```

* In Union at a time you can work with only one of its members because memory is shared among all the members.
* Memory/size of union is equal to its largest member size.
* Application of union is "type pruning";

Miscellaneous

Comma operator

→ int x=10, y=20, z=30;

or

{ int x=10;
int y=20;
int z=30; }

* It works as a seperator
* It has the least priority as operator.
* It works as an operator.

a= (10, 20, 30); var = (Exp1, Exp2 ...)

All these expressions are evaluated from left to right and the final value of variable will be the right-most expression value. Expressions before the last one are evaluated and simply discarded.

eg:
```
int i;
i = (printf("pankaj"), 10+3);    last printf
printf("%d", i); → pankaj 13.
```
13.

during evaluation

Scoping

Static/lexical Scoping

Decision ⇒ Compile time.

Dynamic Scoping

Scope related decision ⇒ Runtime

```
→ int a;
{ { {
    printf("%d",a);
} } }
```

→ Consider the program in a hypothetical language that allows global variable and a choice of static & dynamic scoping.

```
int i;

Program main()
{     i=10;
     Call f(); }

Procedure f() { int i=20;
               Call g(); }

Procedure g() { print(i); }
```

With static : O/p = 10 (Global Variable)

With dynamic scoping : O/p = 20 (Searches 'i' value from parent function)

Let a: value under static & dynamic scoping.

Static Scoping :

main()
↓
f()
↓
g() – prints 'i' (Global variable as itf not there locally)

dynamic Scoping :

main()
↓
f()
↓
g()

Searches 'i' in parent function

Preprocessor Directive

```
#include <stdio.h>   --> File inclusion
#define MAX 10;      --> macro, every occurance of MAX is replaced by 10
void main()
{ printf("%d", MAX); }
```

↳ In pre processor this line is

⇒ printf("%d", 10);

```c
#define Square(x) x*x;
void main()
{   int i;
    i= square(5+3);
    printf("%d", i);}
```

i= 5+3*5+3
= 23

O/p = 23.

Sizeof Operator

* Compile time operator
* Unary operator ⟹ 1 Operand → variable, datatype, Expression, Constant/literal

int=4
Char=1

```c
int i=2, j=3;
pf ("%u", sizeof(i)); → 4
pf ("%u", sizeof(i+2*j)); — int+int*int = int
                                       = 4
pf ("%u", sizeof(2));
pf ("%u", sizeof(int));   } 4..
```

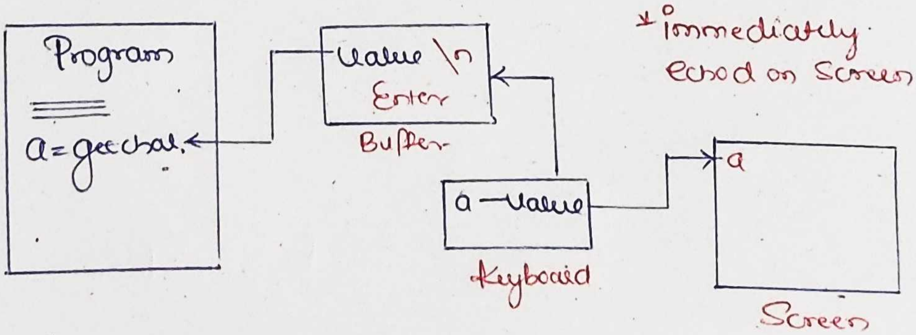
eg:
```c
int i=1, j;
j= sizeof(++i);
printf("%d %d", i, j);
```
not executed, as sizeof is compile time operator & int+1 = int

⟹ 1 4

`printf ("%d", sizeof i);` valid

no paranthesis ✓ O/p = 4.

getchar:
① buffer
② Echo.



*immediately echoed on Screen

getch:
① unbuffered
② unechoed. → nothing echoe on Screen.