

# C PROGRAMMING

Handwritten Notes

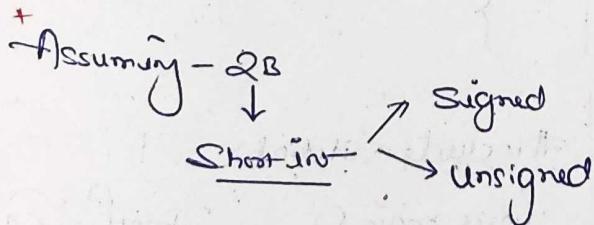
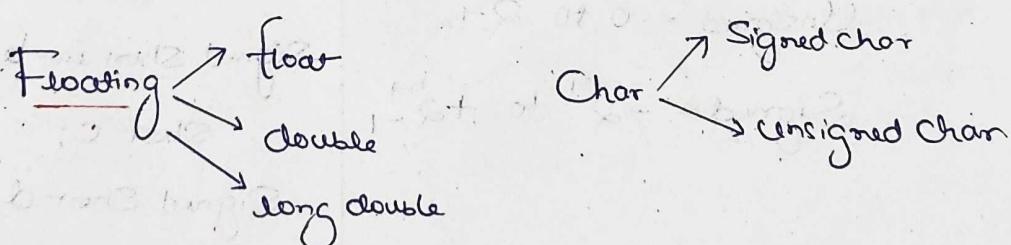
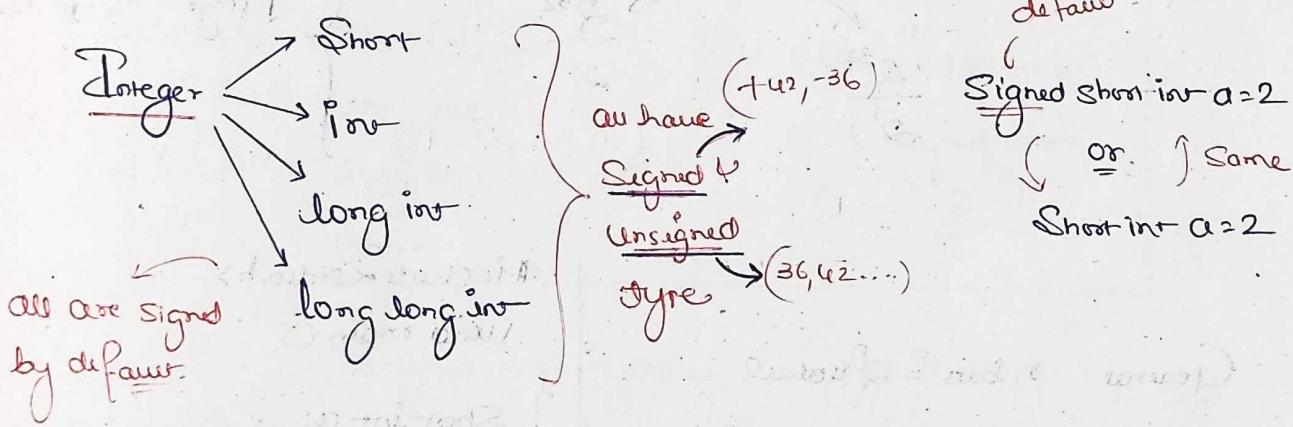
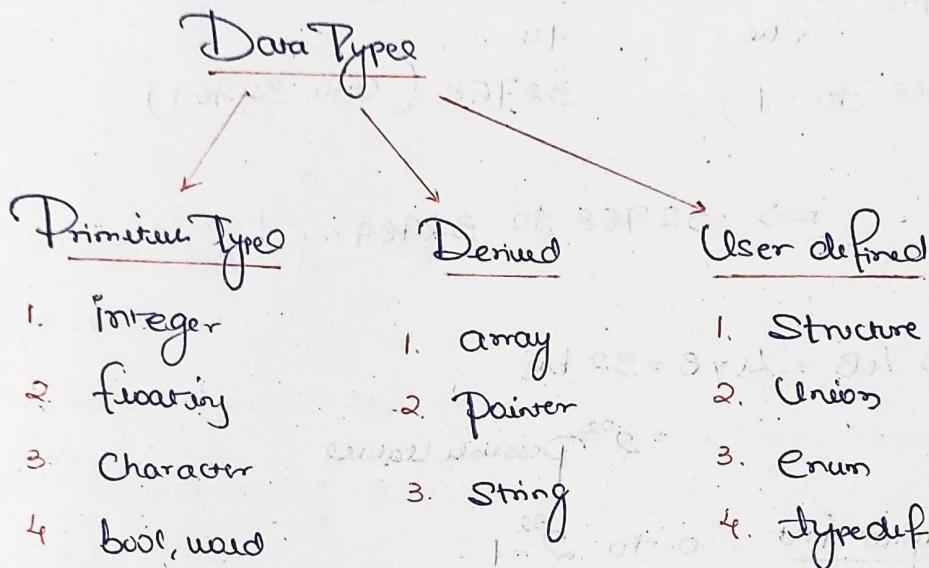
**VIVEKANAND VERNEKAR**

GateWallah Parakram C 2024  
(Hinglish Weekday)

Other Notes Uploaded Below

<https://t.me/pwgatenotes>

# Data Types and Operators



$$\begin{aligned}
 2 \text{ byte} &= 2 \times 8 = 16 \text{ bit} \\
 &= 2^{16} \\
 &= 65,536
 \end{aligned}$$

Unsigned short int = (0 to 65,535)

### Signed Short int

↳ 65536

32768

-ve

(-32768 to -1)

+ve

32768 (0 to 32767)

⇒ -32768 to 32767.

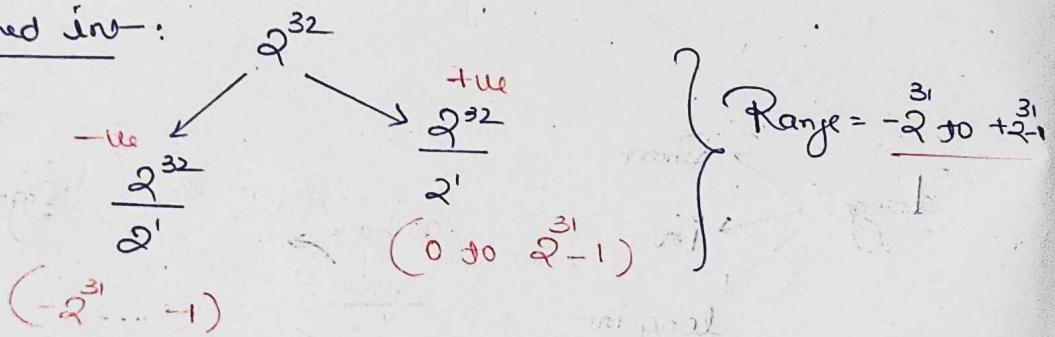
} Range

Inv ⇒ 4B = 4 × 8 = 32 bits

=  $2^{32}$  possible values.

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

Signed int:



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.h>

void main()

{ Signed int a=10; // initialised

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  
→ printf("%d", a)

Format  
Specifier

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

```
void main()
```

```
{ short int a = 20;
```

O/p:

Printf("The value is %d", a); → The value is 20

}

format Specifier of

'integer-type' (Short int and int type  
not for ll)

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

```
void main()
```

```
{ Signed short int a = 20;
```

The values are printed accordingly

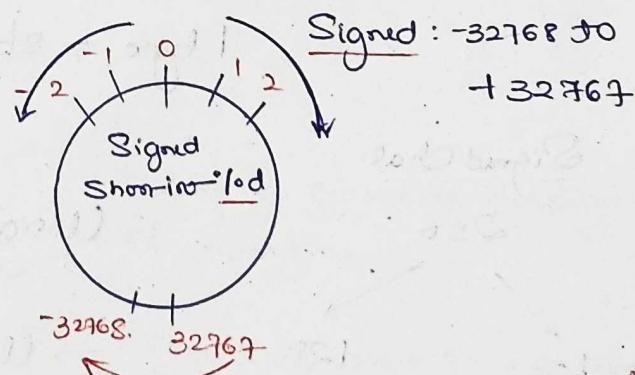
Printf("%d is %d", a, a);

} O/p → 20 is 20

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

Signed int → 2 Byte { Cyclic Property }

→ Unsigned 0 to 65535



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

excess number, goes in cycle after limit

Short int i = -32760;

printf("%d", i);

0/p = 32766

(→ goes ahead in cycle  
after limit.)

Unsigned short int a = -3;

printf("%d", a); // 65535

printf("%d", a); -3, it's not available

-3

Unsigned, so goes in  
reverse of cycle

65535

because printf is

"%d" carries print signed

Value, number then

## Taking input from the keyboard

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

```
void main()
```

```
{ int a;
```

```
printf("Enter a number");
```

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

format to store value at particular  
address of a variable

```
printf("The value of a is %d", a);
```

}

&a  
↓  
"address of a".

## Signed Char

256

128

128

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

Unsigned Char = 0 to  $2^8$

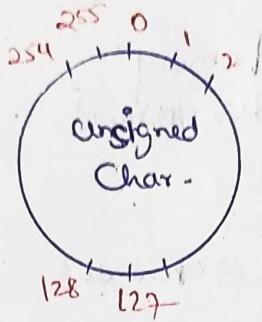
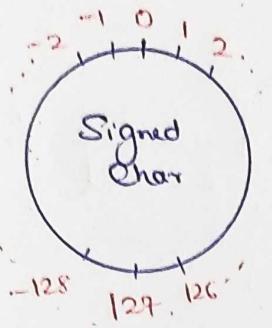
→ Using a Character System, we represent  
symbols of a long by the integer  
constants

Range = -128 to 127

A - 65	a - 97	'0' - 48
z - 90	z - 122	

ASCII  
System.

`%c` → Character System.



Char C = 65;

Printf("%d", c);

Signed int → O/p = 65  
Value

Printf("%c", c);

Char → O/p = A  
Value → hex. ASCII value = 65

Char C = -130;

Printf("%d", c);

Signed integer, -130 not available

So goes in reverse cycle

O/p = 126

Signed by default.

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

decimal value

Char → decimal value  
"C" → Stored value

-130 → 126 (-130 + 256)

-132 → 124 (-132 + 256)

-191 → 65 (-191 + 256)

Char C = -191

Printf("%c", c);

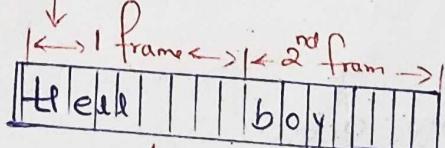
(char val)

O/p = 'A' because value '65' is

Stored because of cyclic property

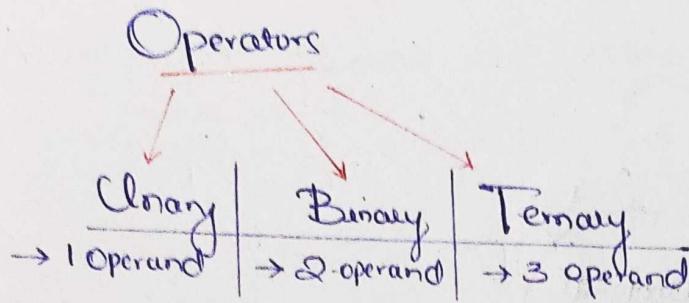
"\n" → Move the cursor to the beginning of next line.

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



Each frame is of 8 characters  
8 space in single line

→ A frame is group of 8 characters.



## 1) Binary Operators (2 Operands)

→ Assignment Operator:

$a = 30 + 2 * 1$

evaluate  
value + then  
Store

Value = Value

↓  
expression / constant  
/ variable

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

→ Arithmetic Operator

$+, -, *, /, \%$

$\frac{+}{\times}$  Math       $\frac{-}{/}$  Modulo

(i)  $\%, +, /$  Priority high  
(ii)  $+, -$  To low

\* Follows Boolean Rule

Priority (precedence)

eg:  $8 / 2 + 4$  → Both operators  
Same Priority

There Associativity coming  
prior. (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  $\leftarrow 12/5 \rightarrow 2$

float  $\leftarrow 12.0/5 \rightarrow 2.4$

$12/5.0 \rightarrow 2.4$

$12.0/5.0 \rightarrow 2.4$

eg: int a;

$$a = 4.0 * 6 \% 3 + 2;$$

printf("%.1d", a);

→ Compiler error

$$= 4.0 * 6 \% 3 + 2$$

$$= 24.0 \% 3$$

Compiler error.

## Relational Operator

greater      equals to  
 $>, <, \geq, \leq, ==, !=$   
smaller      not equal to

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

high

unary +, -

Arith.  $[*, /, \%]$  Left to Right  
 $+, -$

Relational  $[<, >, \leq, \geq, ==, !=]$  Left to Right  
associate.

low.

Assignment = R to L

Q1} int i;

i = printf("%.1d", 4/2+3%2);

printf("%.1d", i);

O/P: 8

Q2} int a;

a = printf("%.1d", printf("%d",  
2023));

printf("%.1d", a);

O/P: Gate 202391

## Logical Operators

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

→ `printf("1.d", 2 & 7);`  $\Rightarrow 1$

non zero (True)      non zero (True)

All non zero values are treated as "true"

And zero is treated as False.

→ `printf("1.d", 2.3 & -13);`  $\Rightarrow 1$

→ `printf("1.d", 1 & 0);`  $\Rightarrow 0$

False, so never is zero.

→ `int a; a = 3 && printf("Hello");`  
`printf("1.d", a);`  $\Rightarrow$  Hello 1.

$!(\text{True}) = \text{False}$

→ `printf("1.d", 13.7 & 0.0);`  $\Rightarrow 1$

→ `int a;`      → If the first operand is 0 for logical AND operator then we need not to evaluate 2nd operand.  
`a = 0 &&`   $\boxed{\quad}$   
`printf("1.d", a);`      "Short Circuit Evaluation".

→ `int a;`  
`a = 0 && printf("Hello");`  
`first operand is zero`       $\times$  not evaluated

→ int a;

a = 10 ||  $\square$ ;



not evaluated

If first operand

"Short-Circuit Evaluating"

for logical OR operator is

non zero, we need not to

Evaluate 2nd operand.

### Modify Operator

Increment (+)

Decrement (-)

Pre-increment

Post-increment

+ var

var +

Pre-decrement

Post-decrement

-- var

var --

e.g. int a=5, b;

b = ++a;

printf ("%d %d", a, b);

O/P: 6 6

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

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 points, 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;

$$C = (a++ \& b++); \text{ Short-circuit Evaluation}$$

Print ("1.d+1.d+1.a", a, b, c);  $O/p = 110.$

② int a=0, b=1, c=2, d;

$$d = (++a \& b - -b) || ++c;$$

Print ("1.d+1.d+1.d+1.d", a, b, c, d);  $O/p = 1031.$

③ int a=2, b=2, c=0, d=2, e;

$$e = ((a++ \& b b++) \&& c++) || d++;$$

Print ("1.d+1.d+1.d+1.d+1.d", a, b, c, d, e);  $O/p = 33181$

### Decimal to Binary

$$(67)_{10} \rightarrow (?)_2$$

$$(1000011)_2$$

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

$$\begin{aligned}(1000011)_2 &= 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 64 + 2 + 1 \\ &= 67.\end{aligned}$$

$$10 \dots 1 = 2x + 1 \rightarrow \text{odd}$$

$$\dots 0 = 2x + 0 \rightarrow \text{even.}$$

## Bitwise Operators

### (i) Bitwise AND (&)

Check bits by bit

a	00000111
b	00001011
<hr/>	
00000011	

$1 \& 1 = 1$ ,       $1 \& 0 = 0$   
else all 0

$$\text{int } a=7, b=11, c; \quad 0 \& 0 = 0$$

$$c = a \& b;$$

$$\text{printf}("1.\%d", c); \quad 0 \& 1 = 0$$

$$Op = 3.$$

$$1 \& 1 = 1$$

### (ii) Bitwise OR (|)

00000001
00001001
<hr/>
11001

$$\text{int } a=9, b=17, c; \quad 0 | 0 = 0$$

$$c = a | b;$$

$$\text{printf}("1.\%d", c); \quad 1 | 0 = 1$$

$$Op = 25.$$

$$1 | 1 = 1$$

### (iii) Bitwise XOR (^)

$$\text{int } a=5; b=9, c;$$

$$c = a ^ b;$$

$$\text{printf}("1.\%d", c);$$

$$Op = \underline{\underline{13}}$$

$$000000101 \quad 0 ^ 0 = 0$$

$$00001001 \quad 0 ^ 1 = 1$$

$$00001100 \quad 1 ^ 0 = 1$$

$$\underline{\underline{12}} \quad 1 ^ 1 = 0$$

$$a ^ 0 = a$$

$$a ^ a = 0$$

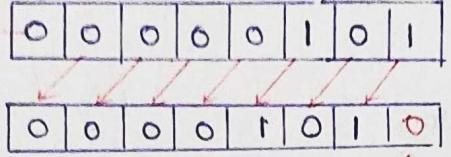
$$a ^ a ^ a = a$$

Even number of times XOR

the same no.  $\Rightarrow 0$

Odd no. of times  $\Rightarrow a$

#### (IV) Bitwise Left Shift Operator

int  $a = 5;$       *Popped out.*      

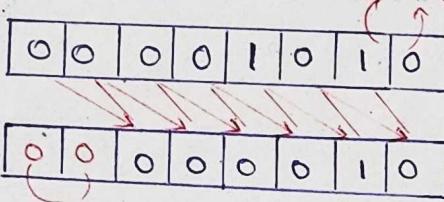
$b = a \ll 1;$       *no. of bits to shift.*

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

#### (V) Bitwise Right Shift Operator

int  $a = 10;$

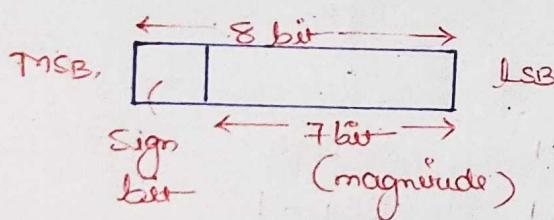
int  $b = a \gg 2;$       *a*      *2<sup>2</sup>*



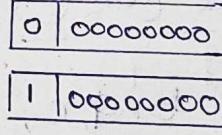
*extra Space*

#### Negative Number in Binary

##### Sign Magnitude.



##### Problem with

this = 

*Create confusion*

#### 2's Complementation

$$+ve = As it is$$

$$-ve = 2^8 \text{ complement} = 1^8 \text{ complement} + 1$$

eg:

$$\begin{array}{r}
 0101001 \\
 1010110 \\
 \hline
 \end{array}
 \quad \begin{array}{l}
 1^8 \text{ complement} \\
 (\text{flip all bits})
 \end{array}$$

$$\begin{array}{r}
 +1 \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 1010111 \\
 \hline
 \end{array}
 \quad \begin{array}{l}
 2^8 \text{ complement number}
 \end{array}$$

eg:  $-23 \Rightarrow 00010111 \rightarrow$  convert the positive way  
 $\rightarrow \underline{1110100} \rightarrow$  2's complement, starting from left keep  
 the first set-bit(1) as it is and then  
 flip all the remaining bits

$\rightarrow$  If the number given is 2's complement form.

$$m\&b \leftarrow 11101001 \\ 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$$

$$\text{Direct method} = -2^4 - 2^2 - 2^1 - 1 \\ = -\underline{\underline{23}}$$

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

### (vi) Boolean NOT ( $\neg$ )

$$\text{Int } a = 48;$$

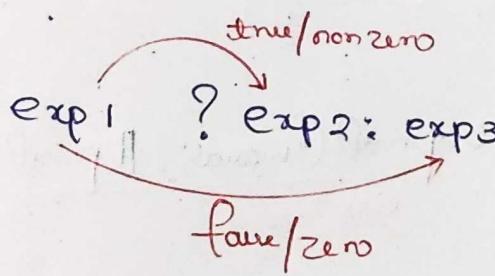
$$b = \neg a;$$

Proof ("if d", b);

$$(\neg x) = -(\bar{x} + 1)$$

$$\begin{aligned} & 48 \xrightarrow{2^7 2^6 2^5 2^4} 00110000 \\ \neg 48 \rightarrow & \underbrace{11001111}_{\substack{\text{all bits} \\ \text{are flipped}}} \rightarrow \\ & = -2^7 - 2^4 - 1 \\ & = -\underline{\underline{49}} \end{aligned}$$

### Ternary Operator ( $? :$ )



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

① int a;  
a = 12 > 10? 12:10;  
    true

$$\textcircled{2} \quad a = \frac{10!}{376?} \cdot \frac{2 = 26! \cdot 3?}{10! \cdot 20! \cdot 30!} ;$$

exp1. exp2. exp3.  
 $a = 10$  true

3.  $(\text{false}) \Rightarrow \text{exp}_3.$

$$a = 15 < 3? \quad 3! = 4 > 10? \quad 10 : 20 : \underbrace{4 < 7! = 7 > 10?}_{\text{exp}_1} \quad !5? \quad 80, !2! = ?$$

$\Downarrow \quad \text{Exp 2}$

$$a = 4 < 7! = 7 > 10? \quad !15? \quad 80 : !2! = ? \quad 4 : 5 : 6$$

$\xrightarrow{\text{Expression 3.}} \quad 4 : 5 : 6 :$

$\Downarrow \quad \text{true}$

$$a = \underbrace{!15}_{\text{exp}_1} ? \quad \underbrace{80}_{\text{exp}_2} : \underbrace{!2! = ?}_{\text{exp}_3} \quad 4 : 5 : 6 \Rightarrow a = \underline{4}.$$

$\xrightarrow{(\text{false})}$

## Shorthand Property:

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

$$x = x \% 10 \rightarrow x \% = 10$$

$$x = x^{10} \rightarrow x^1 = 10$$

$$x = x \& 10 \rightarrow x \&= 10$$

$$x = x \ll 10 \rightarrow x \ll 10$$

Similarly for all  
Arithmetic, logical operation

(4) point a;

`a = 12 ? ? ? ; printf("Gate") & & printf("wanah") || printf("2023")`  
 $\xleftarrow{\text{exp1}} \quad \xrightarrow{\text{exp2}} \quad : \text{printf("Sir");}$   
 $\xleftarrow{\text{exp3.}} \quad \text{printf("o.d", a);}$

Proof ("old", a);

O/p = Gateways,

5.  $\text{int } a;$

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

exp<sub>1</sub>      exp<sub>2</sub>      exp<sub>3</sub>

Print ("1.d", a);

$$0|p = \underline{400}$$

# Control Flow Statements

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

```
void main()
```

```
{  
    S1;  
    S2;  
    S3;  
    S4; }
```

Sequential  
order

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

2) Iterative Statements : loops : for, while, do-while  
(Repetitions)

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

## Selection Statements

```
S1;  
S2;  
if (condition/expression)  
{  
    S3; S4; } { If true  
                S3, S4 will  
                execute.  
    S5;     All statements  
    S6;     in the  
            flow bracket  
            will execute.
```

if (condition/expression)  
only  $\hookrightarrow$  S1;  $\hookrightarrow$  if true  
first statement S2; } independent  
has S3; } of if.  
Scope of  
if statement.

```

① void main()
{
    printf("Start");
    if (2<5)
        printf("0");
    printf("1");
    printf("2");
}
O/p: Start012

```

```

③ int i=2;
    printf("1");
    if (i+2) {
        non zero
        (true)
        printf("2");
        printf("3");
    }
    printf("4");
}
O/p: 1234

```

$\rightarrow \text{if}(-2)$  non zero  
 $\{ \dots \} \therefore \text{true}$

```

if (!printf("Pankaj"))
{
    printf("sharma");
}
O/p: Pankaj

```

② printf("1");
 if (s2) True
 printf("2"); not executed!
 printf("3");
 printf("4");
}
O/p: 134

④ printf("He");
 if (2) non zero (true)
 {
 printf("prabhu");
 printf("bachalo");
 }
 printf(" Iss Rawan Se");

O/p: He Prabhu Bachalo Iss Rawan Se.

$\text{if } (\text{printf("GATE"))}$  value = 4  
 $\{ \dots \} \therefore \text{true}$

if Condition / expression  
if False  $\{ \dots \}$  if non zero (true)  
else  $\{ \dots \}$  optional!

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

```

if (1<2) (True)
  printf("Hello");
else if (2+3=5) (False)
  printf("Doston");
else
  printf("Ye Rawaan hai");
  
```

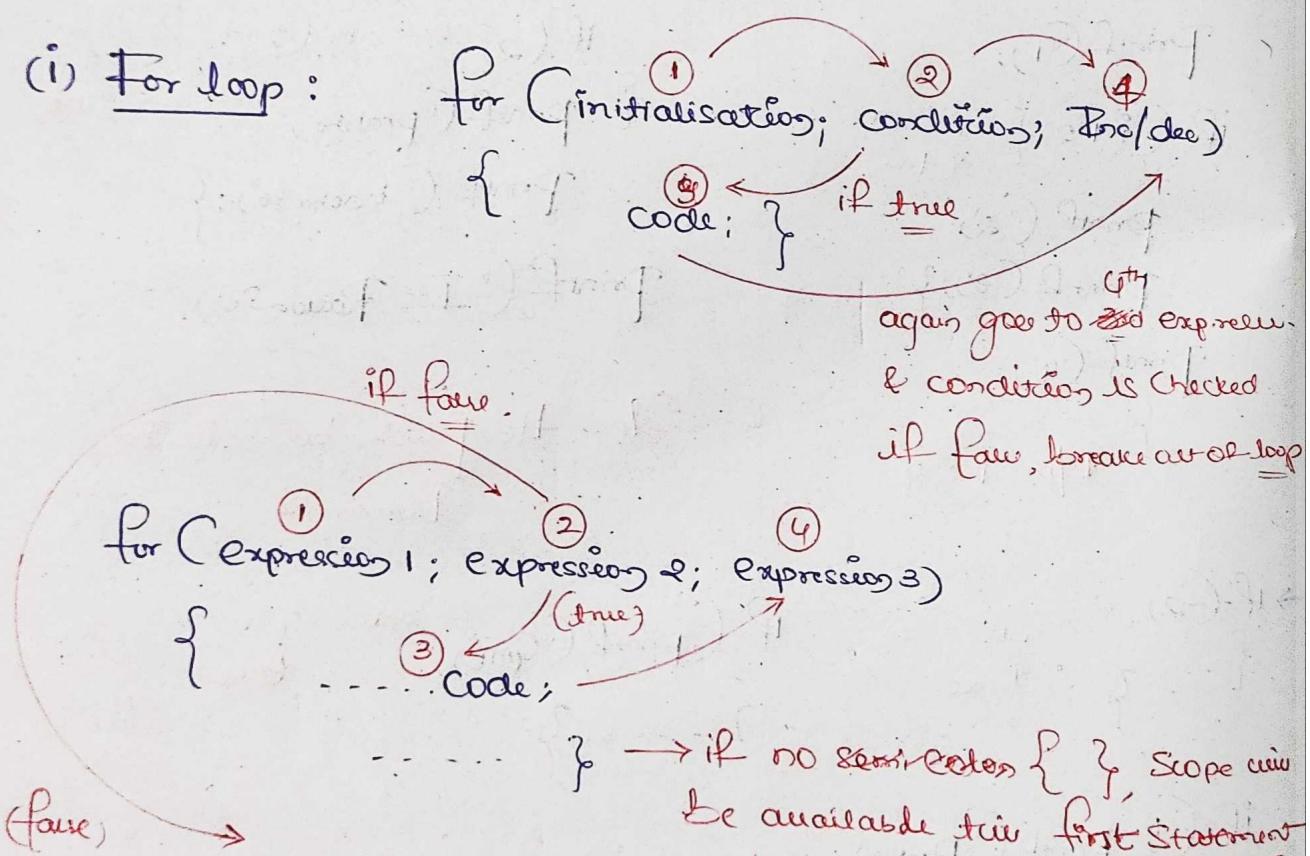
Given 3 distinct integers  
For the maximum number

```

int a,b,c,max;
max = (a>b && a>c)? a:
      b>c? b: c;
printf("%d", max);
  
```

## Iterative Statements

### (i) For loop:



eg: for (int i=1; i≤5; ++i)
 { printf("%d", i); }

O/p: 12345.

```

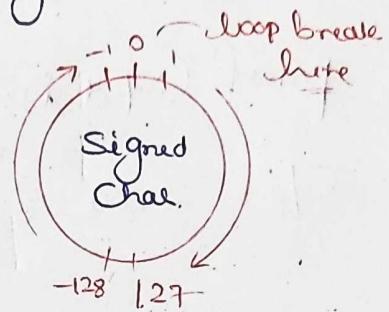
int i=1;
for (i=10; i≤5; 10)
{
    printf("%d", i);
    i=i+2;
}
  
```

O/p: 135.

① Char ch = '1';

```
for (10; ch; ch++)
{ printf("Hello"); }
    ↴ 255 times
```

How many times printf will execute?



② Char ch = '1';

```
for (10; ch; ch+2)
{ printf("Hello"); }
    ↴ Infinitely loop. As it doesn't land on 0.
```

⇒ 255 times

③ int i=1;

```
for (printf("1"); i<4; printf("4"))
{ printf("2");
    i = i + 1; }
```

Output = 124242424

face

④ for (1; 2; 3)

```
{ printf("Hello"); }
    ↴ ∞ loop
```

⑤ for (1; 0; 1)

```
{ printf("Hello"); }
    ↴ 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 ( ; ; ) (true)

```
{ printf("Hello"); }
    ↴ ∞ loop
```

⑧ int  $i=1;$

for ( $i$ ;  $i++ < 6$ ;  $i$ )

{ print ("Hello",  $i$ ); }

$$O/p = 28456$$

## Loop Analysis

1) for ( $\text{int } i=0; i \leq n; ++i$ )

{ print ("Pankaj"); }  $\rightarrow$  loop will run  $n$  times.

2) for ( $\text{int } i=1; i \leq n; i=i+2$ )

{ ... }  $\rightarrow$  loop will run  $\lceil \frac{n}{2} \rceil$  times

3) for ( $\text{int } i=1; i \leq n; i=i+3$ )

{ ... }  $\rightarrow \lceil \log_2 n \rceil + 1$  times

4) for ( $\text{int } i=1; i \leq n; i=i+3$ )

{ ... }  $\rightarrow \lceil \log_3 n \rceil + 1$  times

5) for ( $\text{int } i=1; i \leq n; ++i$ )

{ for ( $\text{int } j=1; j \leq n; ++j$ )

{ print ("Hello"); } }  $\left\{ \begin{array}{l} n \text{ times} \\ n \text{ times} \end{array} \right\}$

}  $\Rightarrow n \times n = n^2$   
 $\text{times}$

6) for (int  $i=1$ ;  $i \leq n$ ;  $++i$ )

{ for (int  $j=1$ ;  $j \leq n$ ;  $j=j+2$ )  
{ printf("pankaj"); } } }  $\left\{ \log_2 j + 1 \right\}^n$   
=  $n (\log_2 j + 1)$

7) for (int  $i=1$ ;  $i \leq n$ ;  $i=i+2$ )

{ for (int  $j=1$ ;  $j \leq n$ ;  $j=j+2$ )  
{ printf("pankaj"); } } } }  $\left\{ \log_2 j + 1 \right\} \left[ \frac{n}{2} \right]$   
 $\Rightarrow \left[ \frac{n}{2} \right] (1 + \log_2 j)$

8) for (int  $i=1$ ;  $i \leq n$ ;  $++i$ )

{ for (int  $j=1$ ;  $j \leq i$ ;  $j++$ )  
{ printf("pankaj"); } } } }  $\rightarrow n \frac{(n+1)}{2}$

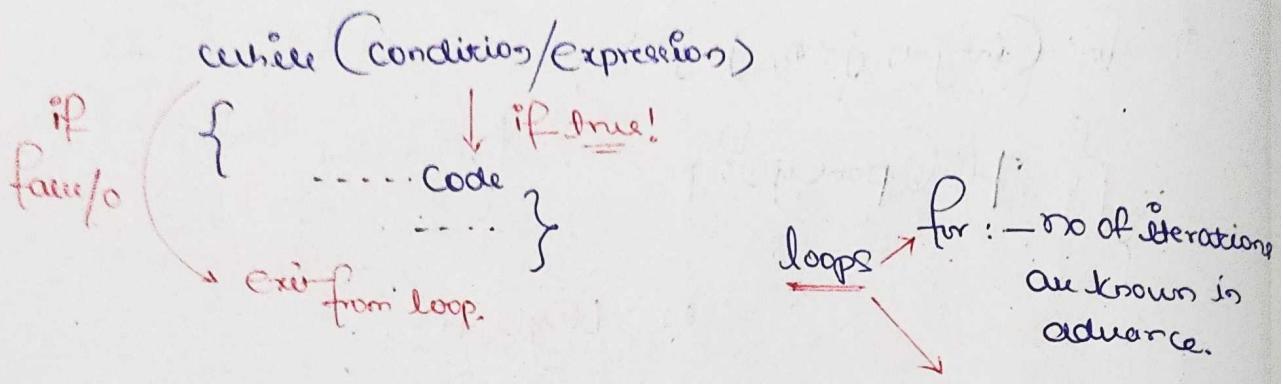
9) for (int  $i=1$ ;  $i \leq n$ ;  $++i$ )

{ for (int  $j=1$ ;  $j \leq 3^i$ ;  $++j$ )

{ ... }  $i=1$   
{ ... }  $i=2$   
{ ... }  $i=3$   
{ ... }  $i=n$   
{ ... }  $j=1 \text{ to } 3^n$   
{ ... }  $j=1 \text{ to } 3 \times 3^{n-1}$   
{ ... }  $j=1 \text{ to } 3^2 \times 3^{n-2}$   
{ ... }  $j=1 \text{ to } 3^3 \times 3^{n-3}$   
{ ... }  $j=1 \text{ to } 3^4 \times 3^{n-4}$   
{ ... }  $j=1 \text{ to } 3^5 \times 3^{n-5}$   
{ ... }  $j=1 \text{ to } 3^6 \times 3^{n-6}$   
{ ... }  $j=1 \text{ to } 3^7 \times 3^{n-7}$   
{ ... }  $j=1 \text{ to } 3^8 \times 3^{n-8}$   
{ ... }  $j=1 \text{ to } 3^9 \times 3^{n-9}$   
{ ... }  $j=1 \text{ to } 3^{10} \times 3^{n-10}$   
{ ... }  $j=1 \text{ to } 3^{11} \times 3^{n-11}$   
{ ... }  $j=1 \text{ to } 3^{12} \times 3^{n-12}$   
{ ... }  $j=1 \text{ to } 3^{13} \times 3^{n-13}$   
{ ... }  $j=1 \text{ to } 3^{14} \times 3^{n-14}$   
{ ... }  $j=1 \text{ to } 3^{15} \times 3^{n-15}$   
{ ... }  $j=1 \text{ to } 3^{16} \times 3^{n-16}$   
{ ... }  $j=1 \text{ to } 3^{17} \times 3^{n-17}$   
{ ... }  $j=1 \text{ to } 3^{18} \times 3^{n-18}$   
{ ... }  $j=1 \text{ to } 3^{19} \times 3^{n-19}$   
{ ... }  $j=1 \text{ to } 3^{20} \times 3^{n-20}$   
{ ... }  $j=1 \text{ to } 3^{21} \times 3^{n-21}$   
{ ... }  $j=1 \text{ to } 3^{22} \times 3^{n-22}$   
{ ... }  $j=1 \text{ to } 3^{23} \times 3^{n-23}$   
{ ... }  $j=1 \text{ to } 3^{24} \times 3^{n-24}$   
{ ... }  $j=1 \text{ to } 3^{25} \times 3^{n-25}$   
{ ... }  $j=1 \text{ to } 3^{26} \times 3^{n-26}$   
{ ... }  $j=1 \text{ to } 3^{27} \times 3^{n-27}$   
{ ... }  $j=1 \text{ to } 3^{28} \times 3^{n-28}$   
{ ... }  $j=1 \text{ to } 3^{29} \times 3^{n-29}$   
{ ... }  $j=1 \text{ to } 3^{30} \times 3^{n-30}$   
{ ... }  $j=1 \text{ to } 3^{31} \times 3^{n-31}$   
{ ... }  $j=1 \text{ to } 3^{32} \times 3^{n-32}$   
{ ... }  $j=1 \text{ to } 3^{33} \times 3^{n-33}$   
{ ... }  $j=1 \text{ to } 3^{34} \times 3^{n-34}$   
{ ... }  $j=1 \text{ to } 3^{35} \times 3^{n-35}$   
{ ... }  $j=1 \text{ to } 3^{36} \times 3^{n-36}$   
{ ... }  $j=1 \text{ to } 3^{37} \times 3^{n-37}$   
{ ... }  $j=1 \text{ to } 3^{38} \times 3^{n-38}$   
{ ... }  $j=1 \text{ to } 3^{39} \times 3^{n-39}$   
{ ... }  $j=1 \text{ to } 3^{40} \times 3^{n-40}$   
{ ... }  $j=1 \text{ to } 3^{41} \times 3^{n-41}$   
{ ... }  $j=1 \text{ to } 3^{42} \times 3^{n-42}$   
{ ... }  $j=1 \text{ to } 3^{43} \times 3^{n-43}$   
{ ... }  $j=1 \text{ to } 3^{44} \times 3^{n-44}$   
{ ... }  $j=1 \text{ to } 3^{45} \times 3^{n-45}$   
{ ... }  $j=1 \text{ to } 3^{46} \times 3^{n-46}$   
{ ... }  $j=1 \text{ to } 3^{47} \times 3^{n-47}$   
{ ... }  $j=1 \text{ to } 3^{48} \times 3^{n-48}$   
{ ... }  $j=1 \text{ to } 3^{49} \times 3^{n-49}$   
{ ... }  $j=1 \text{ to } 3^{50} \times 3^{n-50}$   
{ ... }  $j=1 \text{ to } 3^{51} \times 3^{n-51}$   
{ ... }  $j=1 \text{ to } 3^{52} \times 3^{n-52}$   
{ ... }  $j=1 \text{ to } 3^{53} \times 3^{n-53}$   
{ ... }  $j=1 \text{ to } 3^{54} \times 3^{n-54}$   
{ ... }  $j=1 \text{ to } 3^{55} \times 3^{n-55}$   
{ ... }  $j=1 \text{ to } 3^{56} \times 3^{n-56}$   
{ ... }  $j=1 \text{ to } 3^{57} \times 3^{n-57}$   
{ ... }  $j=1 \text{ to } 3^{58} \times 3^{n-58}$   
{ ... }  $j=1 \text{ to } 3^{59} \times 3^{n-59}$   
{ ... }  $j=1 \text{ to } 3^{60} \times 3^{n-60}$   
{ ... }  $j=1 \text{ to } 3^{61} \times 3^{n-61}$   
{ ... }  $j=1 \text{ to } 3^{62} \times 3^{n-62}$   
{ ... }  $j=1 \text{ to } 3^{63} \times 3^{n-63}$   
{ ... }  $j=1 \text{ to } 3^{64} \times 3^{n-64}$   
{ ... }  $j=1 \text{ to } 3^{65} \times 3^{n-65}$   
{ ... }  $j=1 \text{ to } 3^{66} \times 3^{n-66}$   
{ ... }  $j=1 \text{ to } 3^{67} \times 3^{n-67}$   
{ ... }  $j=1 \text{ to } 3^{68} \times 3^{n-68}$   
{ ... }  $j=1 \text{ to } 3^{69} \times 3^{n-69}$   
{ ... }  $j=1 \text{ to } 3^{70} \times 3^{n-70}$   
{ ... }  $j=1 \text{ to } 3^{71} \times 3^{n-71}$   
{ ... }  $j=1 \text{ to } 3^{72} \times 3^{n-72}$   
{ ... }  $j=1 \text{ to } 3^{73} \times 3^{n-73}$   
{ ... }  $j=1 \text{ to } 3^{74} \times 3^{n-74}$   
{ ... }  $j=1 \text{ to } 3^{75} \times 3^{n-75}$   
{ ... }  $j=1 \text{ to } 3^{76} \times 3^{n-76}$   
{ ... }  $j=1 \text{ to } 3^{77} \times 3^{n-77}$   
{ ... }  $j=1 \text{ to } 3^{78} \times 3^{n-78}$   
{ ... }  $j=1 \text{ to } 3^{79} \times 3^{n-79}$   
{ ... }  $j=1 \text{ to } 3^{80} \times 3^{n-80}$   
{ ... }  $j=1 \text{ to } 3^{81} \times 3^{n-81}$   
{ ... }  $j=1 \text{ to } 3^{82} \times 3^{n-82}$   
{ ... }  $j=1 \text{ to } 3^{83} \times 3^{n-83}$   
{ ... }  $j=1 \text{ to } 3^{84} \times 3^{n-84}$   
{ ... }  $j=1 \text{ to } 3^{85} \times 3^{n-85}$   
{ ... }  $j=1 \text{ to } 3^{86} \times 3^{n-86}$   
{ ... }  $j=1 \text{ to } 3^{87} \times 3^{n-87}$   
{ ... }  $j=1 \text{ to } 3^{88} \times 3^{n-88}$   
{ ... }  $j=1 \text{ to } 3^{89} \times 3^{n-89}$   
{ ... }  $j=1 \text{ to } 3^{90} \times 3^{n-90}$   
{ ... }  $j=1 \text{ to } 3^{91} \times 3^{n-91}$   
{ ... }  $j=1 \text{ to } 3^{92} \times 3^{n-92}$   
{ ... }  $j=1 \text{ to } 3^{93} \times 3^{n-93}$   
{ ... }  $j=1 \text{ to } 3^{94} \times 3^{n-94}$   
{ ... }  $j=1 \text{ to } 3^{95} \times 3^{n-95}$   
{ ... }  $j=1 \text{ to } 3^{96} \times 3^{n-96}$   
{ ... }  $j=1 \text{ to } 3^{97} \times 3^{n-97}$   
{ ... }  $j=1 \text{ to } 3^{98} \times 3^{n-98}$   
{ ... }  $j=1 \text{ to } 3^{99} \times 3^{n-99}$   
{ ... }  $j=1 \text{ to } 3^{100} \times 3^{n-100}$

$$[3+5+7+\dots+(2n+1)] = AP = \underline{\underline{S_n}} = \frac{n}{2} (2a + (n-1)d) = \frac{n}{2} (3 + (2n+1)) = \underline{\underline{n(n+2)}}$$

## (ii) While Loop



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

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

## (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 whether the condition is true or false.

O/P = pankaj.

## Jump Statements

(i) Continue:

```
for ( ; ; )
```

$\{ s_1, s_2 \}$

Continue;

83; ? Skip the remaining part

$S_y$ ; ? } of current iteration &

Continue doing next iteration.

for (int i=1; i<10; ++i)

{ if ( $101 \cdot 3 = 20$ )  $\Rightarrow$  in this case

Continue: ✓ below part is skipped

printf("1.%d", i);

3

① p = 12457810

(ii) Break :

for (int i = 1; i <= 10; ++i)

{ if  $(i+1 \cdot g = 0) \Rightarrow$  then the condition is

break; -tree, break is called switch

Printf ("Hello"); breaker the loop when f = 2

theo, theo

-for( int i=1; i<=5; ++i )

{ for ( $\text{inv}_j$ )<sub>j=1</sub>;  $j \leq s$ ; +j)

{ if  $(i+j) \cdot l \cdot q = 20$

break; — break the inner

Proof ( $\vdash d \dashv d$ , j): loop

?

$$\Theta|_0 = 1112214142435152$$

for (int  $i=1$ ;  $i \leq n$ ;  $i=i+3$ )

$$\left\{ \begin{array}{l} \text{for } (j=j; j \leq n; j++) \\ \quad \{ \text{printf("Hello");} \} \end{array} \right\}$$

$i=1$	$j=3^0$	$\dots j=3^k$
$i=4$	$j=3^1$	$\dots j=3^{k-1}$
$i=7$	$j=3^2$	$\dots j=3^k$
$\vdots$	$\vdots$	$\vdots$
$i=(n-1)+1$	$j=(n-3+1)$	$(n-3^k+1)$

$$3^k \leq \log_3 n$$

$$\log_3 3^k \leq \log_3 n$$

$$k \leq \frac{\log_3 n}{\log_3 3}$$

$$k = \lfloor \log_3 n \rfloor$$

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

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

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

for (int  $i=1$ ;  $i \leq n$ ;  $i++$ )

$$\left\{ \begin{array}{l} \text{for } (j=j; j \leq i+1; j++) \\ \quad \{ \text{for } (k=k; k \leq n; k++) \\ \quad \quad \{ \text{printf("Hello");} \} \} \end{array} \right\} \Rightarrow \sum_{i=1}^n \sum_{j=1}^{i+1} \left( \sum_{k=1}^n 1 \right)$$

### Select 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;

}

(65)

Switch (expression) → Expression that evaluated to be int value because it has a value.

{ Case Constant1 : Code1  
break; → breaks the loop if matched.

Case Constant2 : Code2;  
break;

\* Continue is only for loops not for switch.

Case Constant3 : block of code  
break;

\* if break is not there then lines below will execute (default).

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

Default:

Code;

break;

- fall through

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

}

\* duplicate case labels are not allowed.

i=2;

e.g.: 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    int i=3;

Code:

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

\* Not for all Compilers  
Any statement here is ignored.

O/p: Hello

# Functions and Storage Classes

#include <stdio.h>

\* Compilation happens top to bottom.

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

void main()

{ int a=10, b=20, Answer; → int type }

Answer = multiply(0,b); // call/use of function

if return printf ("%d", Answer);

type & int }

data not }

main → return type of functions,

Error! int multiply (int x, int y) → function header

{      int temp;  
        temp = x+y;  
        return temp; }

↑  
function  
definition.

If not written anything, by default

Compiler will set return type as "int", only if

also in forward declaration!

(Mismatches)

e.g.: void main()

Return type  
of fun is  
int

{ int a= 10;

double fun (int x)

    double b;

{

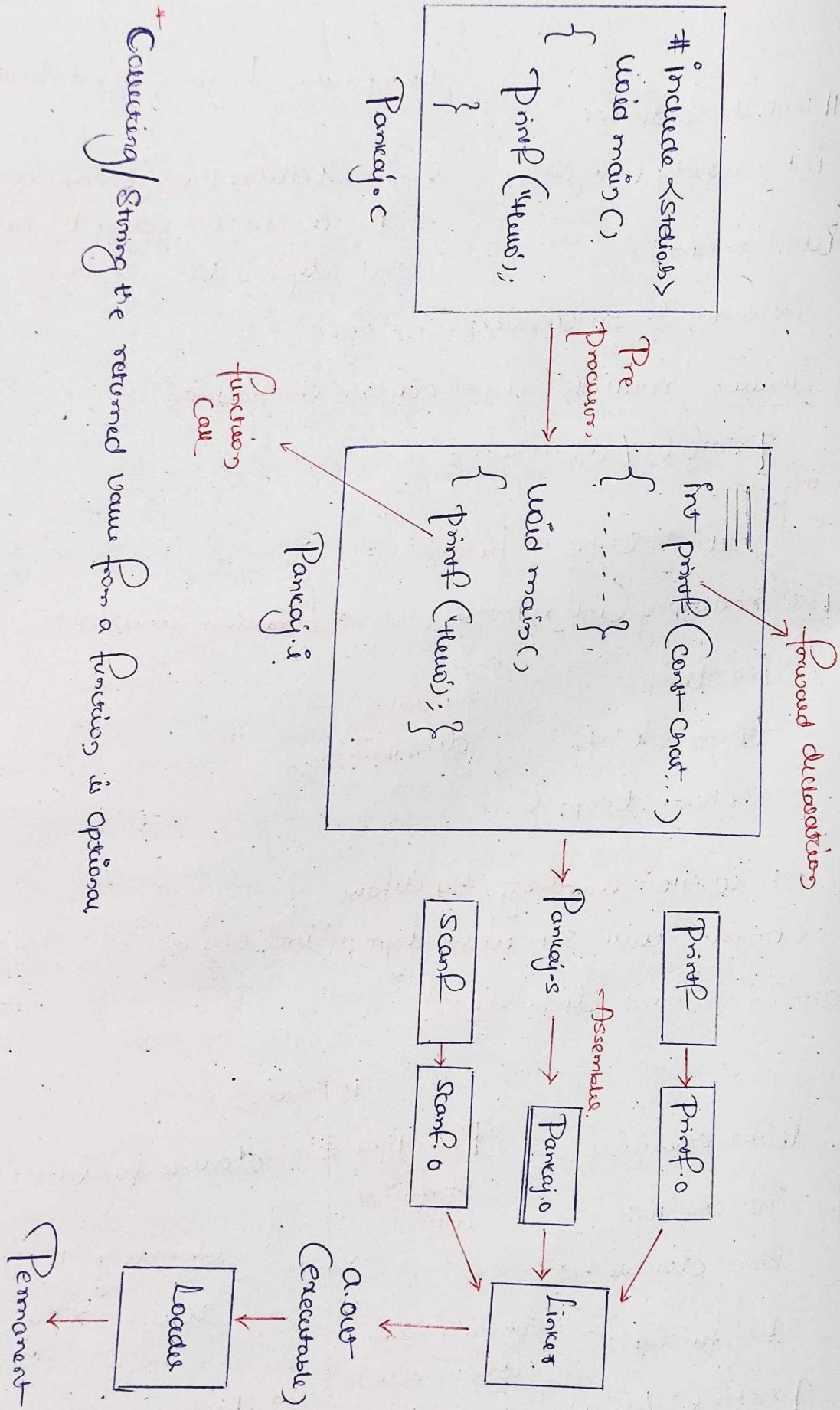
    double y = 10.2;

    b=fun(a); → Compiler gives  
    info = return

    return x+y; }

    printf ("%f", a); type is  
    return 0; integer.

∴ Error.



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

```
void main()
```

```
{ int a=10, b=20;
```

```
mul(a,b); }
```

→ function called but value returned

by the function not stored

→ no error

```
int mul(int x, int y)
```

```
{ int temp;
```

```
temp = x+y;
```

```
return temp; }
```

## How a Function works

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

```
int add(int, int);
```

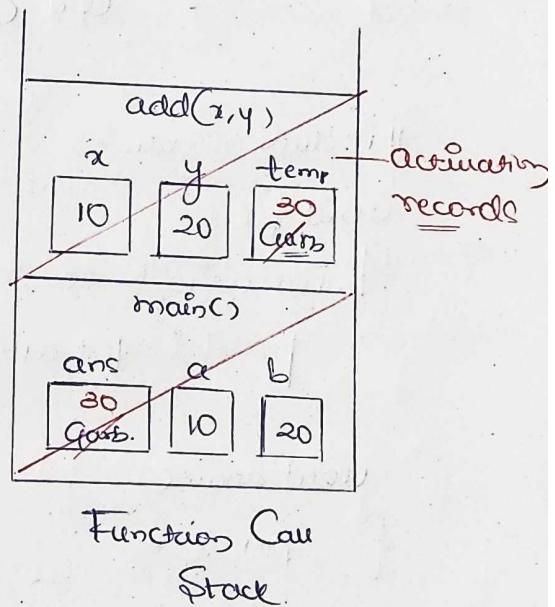
```
void main()
```

```
{ int a=10, b=20, ans;
```

→ ans = add(a, b);

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

O/P = 30



```
int add (int x, int y)
```

```
{ int temp = x+y;
```

```
return temp; }
```

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

```
void swap(int, int);
```

```
void main()
```

int a=10, b=20; O/P = 10 20

```
printf("%d %d", a, b);
```

swap(a, b); → actual arguments

```
printf("%d %d", a, b);
```

→ O/P = 10 20

formal parameters

```
void Swap(int x, int y)
```

```
{ int temp;
```

```
temp = x;
```

```
x = y;
```

```
y = temp; }
```

copy of the

Argument passed

will be here.

So changeable 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.

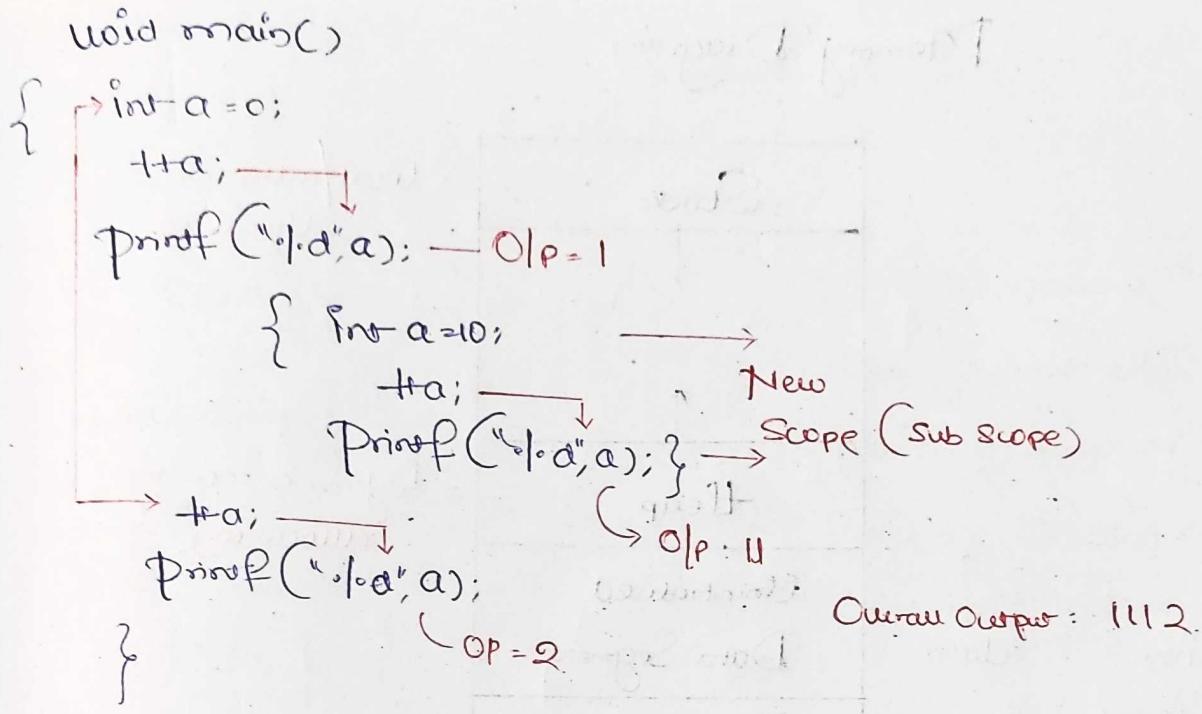
Defined value: The value that is present in a variable when we don't initialise the variable.

```
#include <stdio.h>
void f()
{
    int a=10, b=20; → start of scope
    printf("%d, %d); → local variables
} → 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, variables declared inside a function are auto variables.
- \* Scope and block: block in which they are declared.
- \* Default value: Garbage
- \* Storage area: Stack



- \* Main Variable Scope Variable Can be accessed to Sub-Scope but not vice versa
- \* Sub-Scope Variables are not Accessible to main scope
- \* Two 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 fc()
{
    int a=0;
    fa;
    printf("%d", a); }

```

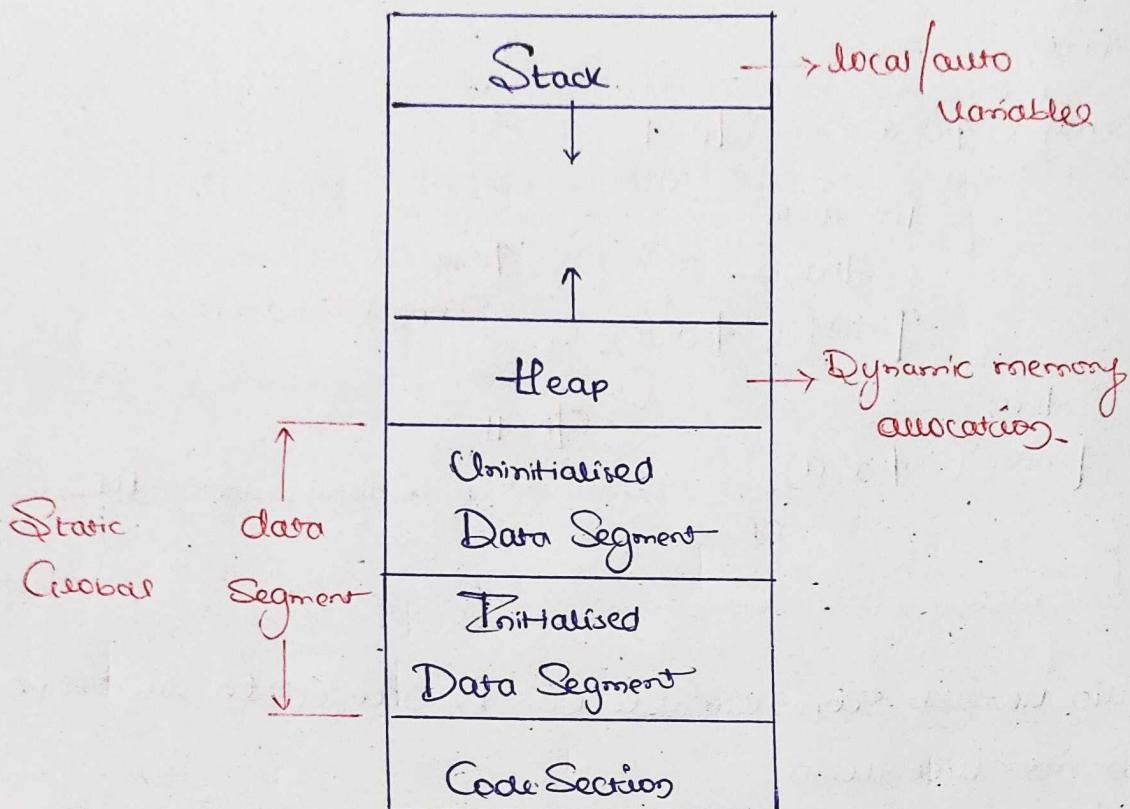
```

void main()
{
    fc();
    fc();
    fc(); }

```

Output = 111

## Memory Diagram

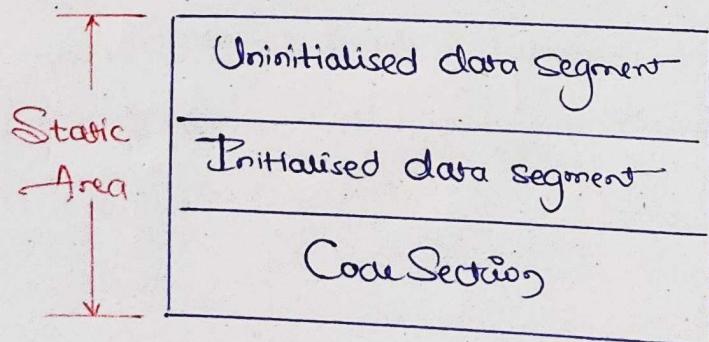


## Register

- \* 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)



```
#include <stdio.h>
void func()
{
    static int i=0; This line is
    update -- i; Executed
    the last updated value.
    printf("%d", i); } Only once
```

Used main()

```
{ func(); -1 O/p = 123.
  func(); -2
  func(); -3 }
```

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

## Global Variable

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

```
void func()
{
    #include <stdio.h>
    extern int x; → compilation error
    printf("%d", x); }
```

int x=0; ← global variable

```
void main()
{
    ++x;
    func();
    printf("%d", x); }
```

```
void func()
{
    extern int x; → local forward declaration for
    printf("%d", x); } global variable.
```

```
int x=0; * extern is only applicable on
void main()
{
    ++x;
    func();
    printf("%d", x); }
```

```
extern int x; → global forward declaration
void func() { ... }
void main() { ... }
int x=0; → global variable definition
```

Internal  
Linkage

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

int i=10;

```
void f();
void g();
void main()
{
    f();
}
```

External  
Linkage

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

forward declaration

```
extern int printf(...);
void main()
{
    printf("Hello");
}
```

PF-definition

PF-F.D  
PF-Cau

Linker

Pankaj.c

## 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 "Pancaj" n times using recursion.

void Print(Pint-n)

{ if ( $n=2$ ) printf ("Pancaj"),  
else { printf ("Pancaj"); }

Print(n-1); }

↓  
Recursively call for smaller value.

eg: Sum of digits using recursion.

int sum\_digits(Gint-n)

{ if ( $n > 0 \&& n \leq 9$ )

return n;

else { return ( $n \% 10$ ) + sum\_digits( $n / 10$ ); }

eg: 125

8 ✓

$125 / 10 = 12$

= 5 + sum\_digits(12), ← 3

$12 / 10 = 2 + \text{sum d}(12 / 10)$

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); }

$3 \times \text{Power}(3, 2) \leftarrow 9$  }

$3 \times \text{Power}(3, 1) \leftarrow 3$

$3 \times \text{Power}(3, 0) \leftarrow 1$

return 1.

eg: void fun (int n)

```
{
    if (n >= 0) return;
    else {
        fun (n-1);
        printf ("%d", n);
    }
}
```

$\text{fun}(3) = 123$

The diagram shows a recursion tree for  $n=3$ . The root node is  $\text{fun}(3)$ . It branches into  $\text{fun}(2)$  and  $\text{pf}(3)$ .  $\text{fun}(2)$  branches into  $\text{fun}(1)$  and  $\text{pf}(2)$ .  $\text{fun}(1)$  branches into  $\text{fun}(0)$  and  $\text{pf}(1)$ .  $\text{pf}(3)$  and  $\text{pf}(2)$  both have arrows pointing to the right, with handwritten notes "if they are printed" and "if they are printed". The final output is  $123$ .

eg: void fun (int n)

```
{
    if (n >= 0)
        return;
    else {
        printf ("%d", n);
        fun (n-1);
        printf ("%d", n);
    }
}
```

$O/P = 321123$

The diagram shows a recursion tree for  $n=3$  with numbered steps. The root node is  $\text{fun}(3)$ . It branches into  $\text{pf}(3)$ ,  $\text{fun}(2)$ , and  $\text{pf}(3)$ .  $\text{fun}(2)$  branches into  $\text{pf}(2)$  and  $\text{fun}(1)$ .  $\text{fun}(1)$  branches into  $\text{pf}(1)$  and  $\text{fun}(0)$ .  $\text{pf}(3)$ ,  $\text{pf}(2)$ , and  $\text{pf}(1)$  all have arrows pointing to the right. Step 1 is  $\text{pf}(3)$ , step 2 is  $\text{fun}(2)$ , step 3 is  $\text{pf}(2)$ , step 4 is  $\text{fun}(1)$ , step 5 is  $\text{pf}(1)$ , and step 6 is  $\text{fun}(0)$ . The final output is  $321123$ .

eg: void fun (int n)

```
{
    if (n >= 0)
        return;
    else {
        fun (n-1);
        printf ("%d", n);
        fun (n-1);
        printf ("%d", n);
    }
}
```

$O/P = 112112$

The diagram shows a recursion tree for  $n=3$  with numbered steps. The root node is  $\text{fun}(3)$ . It branches into  $\text{fun}(2)$  and  $\text{pf}(3)$ .  $\text{fun}(2)$  branches into  $\text{fun}(1)$  and  $\text{pf}(2)$ .  $\text{fun}(1)$  branches into  $\text{fun}(0)$  and  $\text{pf}(1)$ .  $\text{pf}(3)$ ,  $\text{pf}(2)$ , and  $\text{pf}(1)$  all have arrows pointing to the right. Step 1 is  $\text{fun}(3)$ , step 2 is  $\text{fun}(2)$ , step 3 is  $\text{fun}(1)$ , step 4 is  $\text{fun}(0)$ , step 5 is  $\text{pf}(1)$ , step 6 is  $\text{fun}(1)$ , step 7 is  $\text{pf}(2)$ , and step 8 is  $\text{fun}(2)$ . The final output is  $112112$ .

① void  $f(n)$

{ if ( $n \leq 0$ )  
return;

printf("%d", n);  
 $f(n-1)$ ;

What is O/p of  $f(5)$ ?

→ 54321

② void  $f(n)$

{ if ( $n \leq 0$ )  
return;

$f(n+1)$ ;

printf("%d", n); }

What is the Output of  $f(5)$ ?

→ 12345

③ void  $f(n)$

{ if ( $n \leq 0$ ) return;

$f(n+1)$ ;

printf("%d", n);

$f(n-1)$ ;

What is O/p of  $f(4)$ ?

→ 121312141213121

④ int  $f(n)$

{ if ( $n \leq 1$ )

return 0;

return  $f(n/2) + f(n/2) + 1$ ;

What is the Output of  $f(5)$ ?

→ 7

⑤ int  $f(n)$

{ if ( $n \leq 1$ )

return n;

return  $f(n/2) + n/2$ ;

What is O/p of  $f(12)$ ?

→ 11.

⑥ int  $f(n)$

{ if ( $n \leq 1$ ) return n;

if ( $n \geq 2$ )

return  $f(n/2) + n$ ;

return  $f(n/3) + n$ ;

}

What is the Output of  $f(22)$ ?

→

⑦ // Assume  $n \geq 0$

```
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()

```
{ 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

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

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

⑭ int fun (int x)

```
{
    if (x>3)
        return fun(x-4)+fun(x-1)+1;
    return 1;
}
```

What is the value returned  
by  $\text{fun}(12)$ ?  
→ 51

⑯ int fun (int x, int y)

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

What is the output  
of  $\text{fun}(4,3)$ ?  
→ 13:

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  $\text{foo}(53,2)$ ?

→ 2.

⑮ int fun (int n)

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

int main()

```
{
    printf("%d", fun(2));
    return 0;
}
```

⑰ int fun (int x, int y)

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

What does the following function  
do?

→  $x^y$ .

⑯ 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);

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

int main()  
 { print (1000);  
 getch();  
 return 0; }

What is the output?

→ 1000 2000 4000 4000 2000 1000

⑰ What does the following function do?

int fun (unsigned int n)  
 { if (n == 0 || n == 21) return n;  
 if (n % 3 != 0) return 0;  
 return fun (n / 3); }

→ It returns 0 if n is a power of 3, otherwise returns 0.

⑱ Predict the output of the following program.

#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.

Q2) 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

Q3) void count(int n)

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

int main()

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

→ 312213444

Q4) Consider the following recursive C function. If get() function is being called in main(), then how many times will the get() function be invoked before returning to the main().

void geo(int n)

```
{ if (n<1) return;  
geo(n-1);  
get(n-3); }  $\Rightarrow$  Q5 (Ans)
```

printf("%d.%d",n); }

Q5) What can be the output of the following C program?

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

```
int main()
```

```
{ main(); return 0; }
```

→ Runtime Error

# Arrays and Pointers

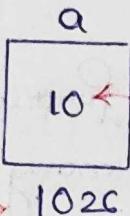
Address

- Absolute address
- Relative address

- \* : value at operator
- & : address of operator

`int a = 10;`

$\&a$ : memory location  
1026



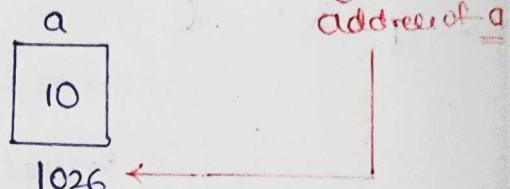
\*(&a)

How to get address of any variable?

`int a = 10;`

→ & : address of operator

$\&a \rightarrow$  give the address of a



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

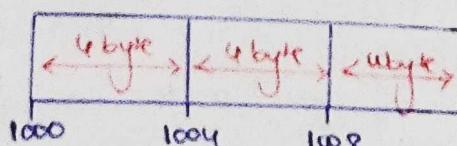
\*(&a) = a

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`  $\Rightarrow$  `int arr[5];`

if int = 4 byte

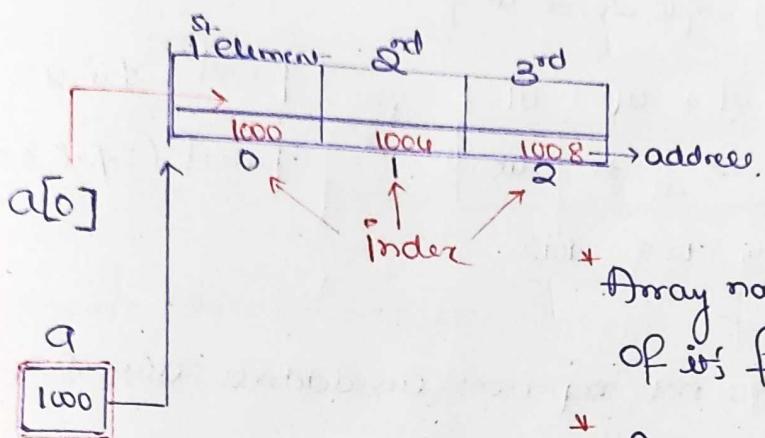
`int arr[3]`



\* 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 its first element.

Array name → Constant Address.

++array-name

--array-name

array-name ++;

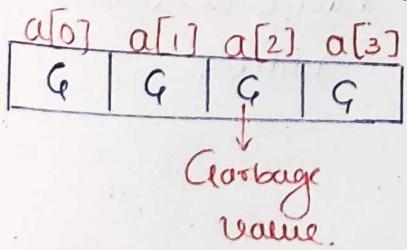
array-name -

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]);
}
```



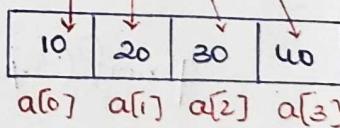
int a = {10}; ✓ valid

int a[0]; ✓

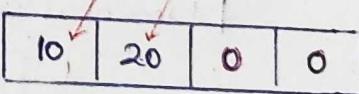
int a [2\*3+0]; ✓

int a [2 \* sizeof(int)]; ✓

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



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

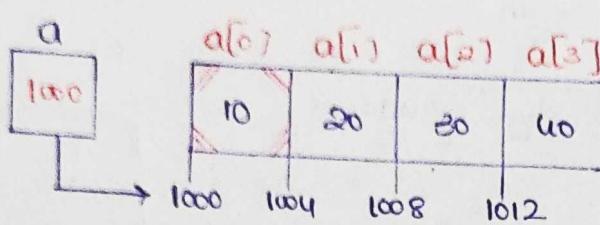


int a(); ✗ Invalid

int a[] = {10, 20, 30}; ✓ valid

① Array-name represents address of its first element.

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



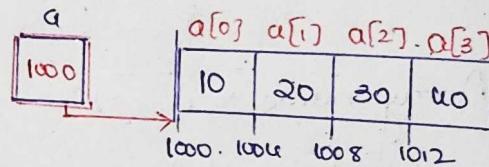
$\downarrow$        $\left\{ \begin{array}{l} \text{printf}(" \%d", a); \\ \text{printf}(" \%d", &a[0]); \end{array} \right.$

② Array name does not represent an address with & Operators

(i) & : address of Operator      (ii) sizeof : Size of Operator

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

Address of Element of 4 Byte.       $\left\{ \begin{array}{l} \text{printf}(" \%d", a); \\ \text{printf}(" \%d", &a[0]); \\ \text{printf}(" \%d", &a); \end{array} \right.$



Address of address  
Array, 16 bytes ka address.

$\text{printf}(" \%d", &a[0]);$        $\text{printf}(" \%d", &a);$       } Numerical value is same but logically we are talking about different elements address.

③

Numerical value of a is 100

a is simple user value      a is an address / pointer variable

$\text{int } a=100;$       Address Arithmetic.

$\text{pf}(" \%d", a);$        $= 101.$

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

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

int a[4]; → 1 dimension

int a[2][3]; → 2 dimensions

int a[2][3][4]; → 3 dimensions

\* Anywhere other than declarations, if we are provide exactly n-dimensions then we are working on element.

\* If we are providing less than n-dimensions then we are working on address

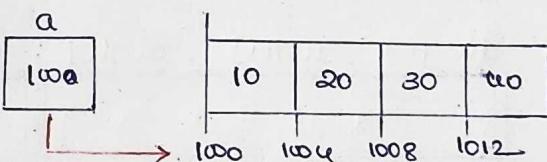
eg: int a[2][3]; → dimension

a; → 0 dimensions → address

a[0]; → 1 dimension → address

a[0][0]; → 2 dimensions → element

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



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

$a \rightarrow$  address



arrayname = &a[0]

↓  
Size = 4 byte.

$$\rightarrow a+1 = \&a[0] + 1$$

$$= \&a[0] + (1 \times 4)$$

$$= \underline{1004} \quad (\text{address of next element})$$

$$a+2 = \&a[0] + (2 \times 4)$$

$$= 1000 + 8$$

$$= \underline{1008}$$

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

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

$$\rightarrow * (a+2) = \text{Value at (memory loc: 1008)} = * \&a[2];$$

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

→ → Addition is commutative

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

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", &a[2]);

printf("%d", \*(&a + 2));

? 1000 1004 1008 1012

Output = 30

printf("%d", a[1]); → 1004

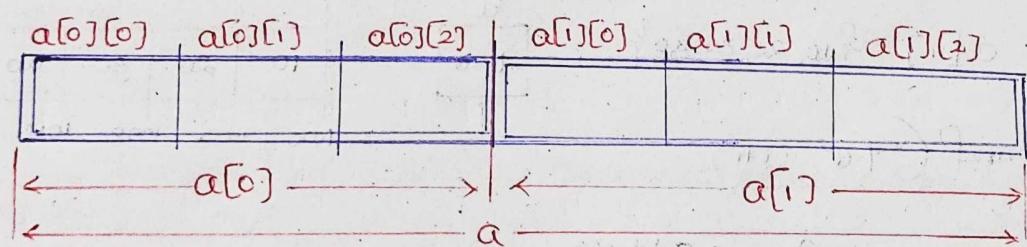
printf("%d", &a[1]); → &a[1] = 1000 + (1 × 16) =

↓  
addressee of

whole array

## 2-D Array

int a[2][3]



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]

void main()

eg: { for a[2][3] = {1, 2, 3, 4, 5, 6};

printf("%d", a+1);  $\rightarrow \&a[0]+1 = 1000 + (1 \times 12) = 1012$

printf("%d", a[0]+1);  $\rightarrow \&a[0][0]+1 = 1000 + 1 \times 4 = 1004$

printf("%d", &a+1);  $\rightarrow \&a+1 = 1000 + 1 \times 24 = 1024$

}

Address of three elements

whole array

eg: 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]);

printf("%d", &a);

Size of a[0] = 16 Byte

printf("%d", a+1);  $\rightarrow 100 + (1 \times 16) = 116$

printf("%d", a[0]+1);  $\rightarrow 100 + (1 \times 4) = 104$

printf("%d", &a+1);  $\rightarrow \&a+1 = 100 + (1 \times 48) = 148$

$$\begin{aligned} \rightarrow a[0]+1 &= \underbrace{\&a[0][0]+1}_{4 \text{ Byte}} \\ &= 100 + (1 \times 4) \\ &= 104 \\ &= \underline{a[0][1]} \end{aligned}$$

$$\rightarrow a[0]+2 = \&a[0][2]$$

$$\rightarrow *(\&a[0]+1) = \text{value(memory location)} = *(\&a[0][1])$$

$$= \underline{a[0][1]}$$

$$\rightarrow *(\&a[1][2]) = a[1][2],$$

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

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

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

a[0]  $\leftarrow$  printf("%d", \*a);  $\rightarrow 100$

a[0][0]  $\leftarrow$  printf("%d", \*\*a);  $\rightarrow 104$

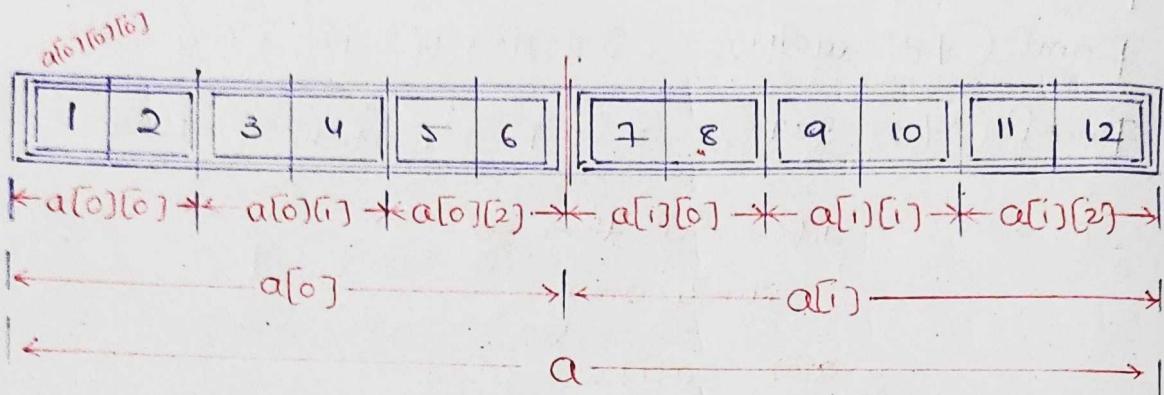
printf("%d", \*a+1);  $\rightarrow 104$

printf("%d", \*\*a+1);  $\rightarrow 2$

printf("%d", \*a[0]);  $\rightarrow 1$

## 3-D-Array

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



$a: \text{array } a[0], a[1]$

$a[0]: \text{array } a[0][0], a[0][1], a[0][2]$

$a[0][0]: \text{array } a[0][0][0], a[0][0][1]$

$\text{printf}("1.d", a+1); \rightarrow &a[0]+1 = 1000 + (1 \times 24) = 1024$

$\text{printf}("1.d", a[0]+1); \rightarrow 1008$

$\text{printf}("1.d", a[0][0]+1); \rightarrow 1004$

$\text{printf}("1.d", &a+1); \rightarrow 1048$

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

1.  $a$
2.  $a[0]$
3.  $a[0][0]$
4.  $\&a$
5.  $a[0][0][0]+1$

$$6. a+1 \rightarrow 1000 + 24 = 1024$$

$$7. a[0]+1 \rightarrow 1000 + 12 = 1012$$

$$8. a[0][0]+1 \rightarrow 1004$$

$$9. \&a+1 \rightarrow 1000 + 72 \\ = 1072$$

$$10. a[0][0][0]+1 \rightarrow 2$$

$$11. *a \rightarrow *a[0] \rightarrow 1000$$

$$12. **a \rightarrow *(&a[0][0]) \rightarrow 1000$$

$$13. ***a \rightarrow *(a[0][0][0]) = 1$$

$$14. *a+1 \rightarrow &a[0][0]+1 = 1012$$

$$15. **a+1 \rightarrow &a[0][0][0]+1 = 1004$$

$$16. *a[0]+1 \rightarrow *(&a[0][0])+1 = 1004$$

$$17. ***a[0]+1 \rightarrow *(a[0]+1)$$

$$= a[0][0][0]+1$$

$$= 1+1$$

$$= 2$$

## Array declarations & initializations

→  $\text{int } a[];$  ✗ Invalid →  $\text{int } a[0];$  ✓  
 →  $\text{int } a[] = \{10, 20, 30\};$  ✓ Valid →  $\text{int } a[4] = \{10, 20, 30, 40\};$  ✓

→ 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.

$\text{int } a[] = \{10, 20, 30\};$  ✓ Valid

$\text{int } a[][3] = \{1, 2, 3, 4\};$  ✓ Valid

$\text{int } a[][],$  ✗ Invalid

$\text{int } a[2][] = \{1, 2, 3, 4, 5, 6\};$  ✗ Invalid

## Pointers

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

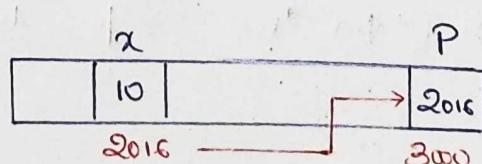
$\text{int } *p;$  → P is a pointer to integer

→ P can store address of some integer variable

e.g.:  $\text{int } x = 10;$

$\text{int } *p;$

$P = \&x;$



$\text{printf}("4.d", \&x);$

$\text{printf}("1.d", p);$

$\text{printf}("4.d", *p);$  → 10

int  $x = 10;$

int \*p;

int \*\*q;

$P = \&x;$

$q = \&p;$

x	p	q
10	1016	2000
1016	2000	3000

↓  
Pointers to a pointer  
of integer type

printf("%d", p); → 1016

printf("%d", \*p); → 10

printf("%d", q); → 2000

printf("%d", \*\*q); → 1016

printf("%d", \*\*\*q); → 10

Assume

int - 4 byte

Char - 1 byte

Float - 8 byte

int \*p;

Char \*q;

float \*r;

Pointers can be  
of same size.

eg: int  $x = 300;$

Char \*p;

$P = (\text{char}^*)\&x;$

printf("%d", \*p);

$0|P = -112$

00000000	00000000	00000001	10010000
----------	----------	----------	----------

$\begin{array}{r} 7 \\ 2 \\ 1 \\ 0010000 \end{array}$

$$\begin{aligned} &= -2^6 - 2^5 - 2^3 - 2^2 - 2^0 \\ &= -112 \end{aligned}$$

eg: int  $x = 130$

Char \*p;  $P = (\text{char}^*)\&x;$

printf("%d", \*p);  $0|P = -126$  (cyclic property)

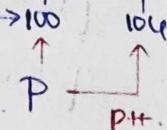
int a[4] = {10, 20, 30, 40};

int \*p;

$P = \&a[0];$

$P = P + 1; \rightarrow 100 + (184) = 104.$

a	100	10	20	30	40
	→ 100	104	108	112	



$P++;$  { inc & dec

$-P;$  { valid ✓

## 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

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

printf("%d", P[0]); → 10

printf("%d", P[1]); → 20

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

+a;  
-a;  
++a;  
--a;

} Invalid.

a = {30, 40}; X Invalid

Array name cannot  
be l-value.

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

int \*p;

P = &a[0];

+p;

--p;

P++;

P--;

} Valid.

P = &a[2]; ✓ Valid

} Variable

→ Pointer-var + 3 = moving 3 locations in forward direction.

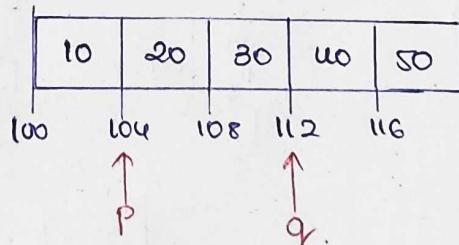
int a[5] = {10, 20, 30, 40, 50}

int \*p, \*q;

P = &a[1];

q = &a[4];

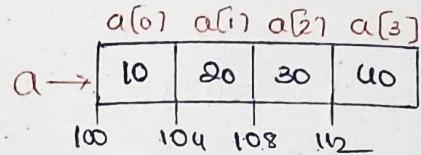
printf("%d", q-p); ✓



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

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

```
void main()
{
    int a[4] = {10, 20, 30, 40};
    int *p;
    p = &a[0];
```



printf("%d", \*(p));  $\rightarrow$  \*(p) = 20

$$+*(p) - +p = 20 + 1 - 2$$

+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);

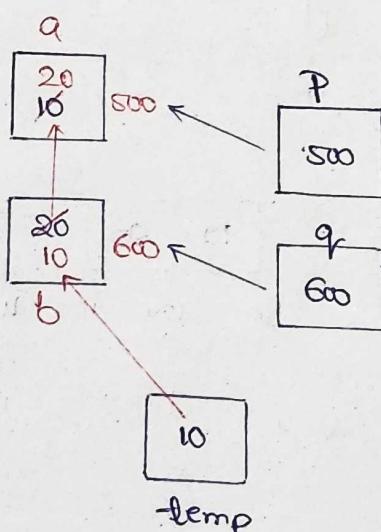
Swap(&a, &b);  $O/p: a=20, b=10$

printf("a=%d, b=%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);
}

```

$\text{int } a[4] = \{10, 20, 30, 40\};$

$\text{int } sum;$

$\text{sum} = \text{fun}(a, 4);$

$\text{printf}("%d", sum);$

0 |  $P = 100$

```

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

```

$\{ \quad \text{int } i, \text{sum} = 0;$

$\text{for } (i = 0; i < n; i++)$

$\{ \quad \text{sum} = \text{sum} + P[i]; \}$

$\text{return sum;}$

## Complex Declarations

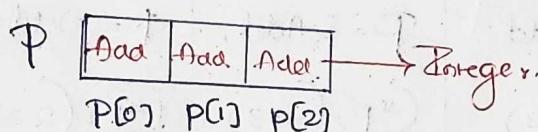
1. () → Parenthesis      functions ] , l for
2. [] → square bracket      Array ]
3. Identifiers → var name, fun-name      ] 2 Rtol
4. \* → Pointer
5. datatype → int, char, float.

eg :  $\text{int } * (P[3])$

③ ②      ①

↓      ↓      ↓

→ P is an array of 3 pointer to integer.



eg:  $\text{int } (*P)[4];$

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

$\text{int } a[4] = \{10, 20, 30, 40\};$

$\text{int } (*P)[4] \neq$

$P = \&a; \checkmark$

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

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

eg: int \*<sup>③</sup>(<sup>①</sup>\*p)[<sup>②</sup>u];

→  $\Phi$  is a pointer to an array of 4 pointers to integers.

## Pointer to function / Function pointers

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

void main()

```

{ int (*p)(int, int); → Pointers to a function which takes 2
    int arguments & returns integers.
    p = &Add;
    printf("%d", (*p)(2, 3)); } → Calling the function with
                                pointer.

```

$$O/P = 5$$

$P = \text{Add};$	$P = \&\text{Add};$	$P = \#Add;$	$P = \&\#Add;$
$(P)(2,3)$	$(\#P)(2,3)$	$(\&P)(2,3)$	$(P)(2,3)$

All are valid declarations / function calls.

- ① int  $\alpha[u] = \{10, 20, 30, u\}$ ;  
 int  $*p[u] = \{at_2, a, at_1, at_3\}$ ;  
 int  $**q;$       array of pointers  
 $q = \& p[0];$   
 printf("q.d",  $^{++} *^{++} q$ );

Op. 20.

②   
 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 void fun(int \*);      void fun(int \*p)  
 void main()  
 { int a[4] = {10, 20, 30, 40};  
 fun(a);  
 printf("%d %d", a[0], a[1]);  
 }      { \*p; \*p++; \*(\*p); }

O/p = 10 20

④ 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])      O/p = 6 6 7  
 { \*p;  
 (\*p)[1] = (\*p)[1] + 1; }

eg: void main()  
 { int \*p; } → declarations

printf("%d", \*p); }  
 ↓  
 Dereferencing  
 (value pass)

Char \*p;  
 - - -  
 - - -  
 - - -  
 printf("%c", \*p);  
 ↓  
 1 Byte info

## Void Pointer

void \*ptr; → address.

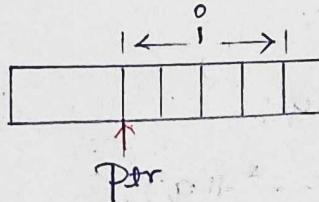
int i = 369;

char ch = 'A';

ptr = &i;

printf("%d", \*ptr); → Error.

printf("%d", \*(int\*)ptr);



void \*p;

int i = 369;

p++;

P = p + 1;

} Error

\* Don't try to dereference any void pointer without typecasting.

\* Do not apply arithmetic operations on void pointer.

void \*ptr;

char ch = 'A';

ptr = &ch;

printf("%c", \*(char\*)ptr); → 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;

}

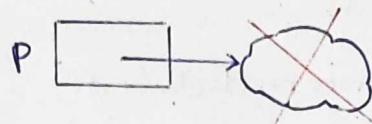
x	10	1008	2096
	2096		

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

## Dangling Pointer



int \*p();



{ static int a=10;

return &a; } → New 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!  
⇒ p = 10.

## Dynamic Memory Allocation

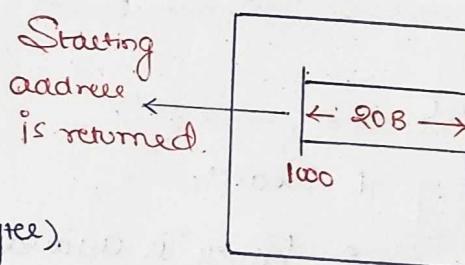
- 1. malloc
  - 2. calloc
  - 3. realloc
  - 4. free
- } Used for DMA in C.

### ① malloc

malloc(20)

↑  
Size in byte.

(void \*) malloc (Size in bytes).



Syntax: (void \*) malloc (Size -t size)

## Sizeof

int \*p;

P = malloc (5 \* sizeof(int));

\* (p+1) = P[1]

```

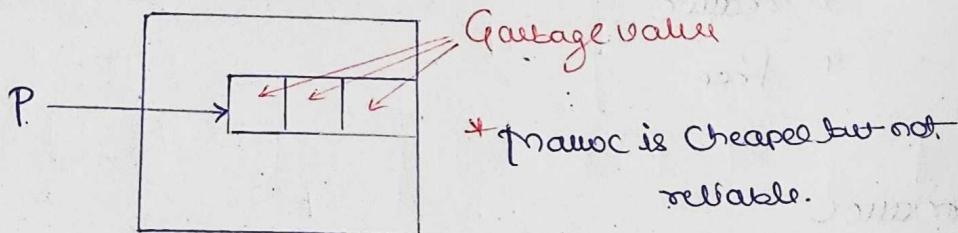
for (int i=0; i<5; ++i)
    scanf("%d", p+i); ← Storing elements
for (int i=0; i<5; ++i)
    printf("%d", *(p+i)); ← Pointing value at loc.
  
```

```

void main()
{
    int N, *p, i;
    printf("Enter no of elements");
    scanf("%d", &N);
    P = malloc(N * 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 :
- Search
  - block is available  $\rightarrow$  Starting address is returned.



## ② calloc :

- Search
  - block is available
  - Starting address is returned.
- \*  $\rightarrow$  Sets all bits = 0  
(instead of garbage value)

\*  $\text{calloc}$  is expensive and reliable

### ③ realloc

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

{ int \*p;

$P = \text{malloc}(5 \times \text{sizeof}(\text{int}))$ ;

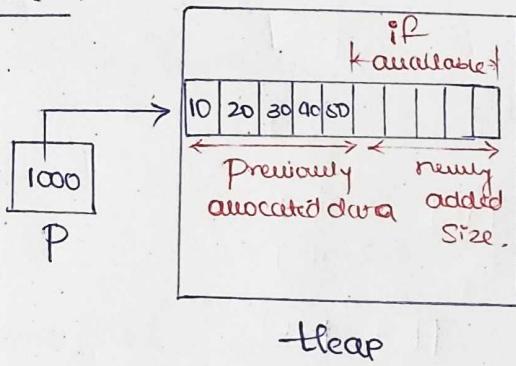
$P = \text{realloc}(P, \underbrace{10 \times \text{sizeof}(\text{int})}_{\text{New Size}});$

$P = \text{malloc}(5 \times \text{sizeof}(\text{int}))$ ;

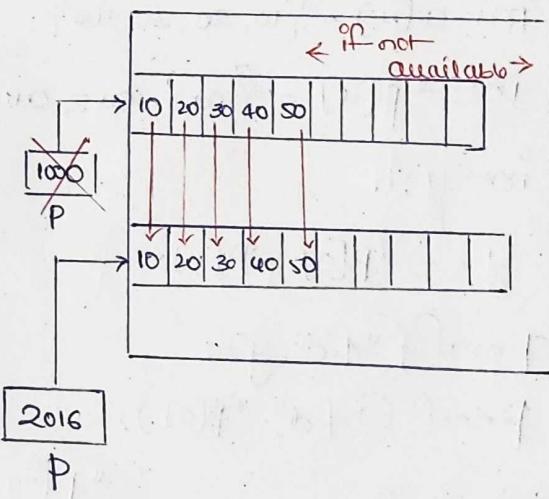
$P = \text{realloc}(P, 3 \times \text{sizeof}(\text{int}))$ ;

Last 2 spaces are erased.

#### Case 1:



#### Case 2:



### ④ free : It is used to free the dynamically allocated memory.

{ void fc();  
int \*p;  
    ↑ local variable

$P = \text{malloc}(100 \times \text{sizeof}(\text{int}))$ ;

}  $\text{free}(P)$ ; → deallocate the memory; Otherwise

{ void main()  
    fc(); fc(); fc(); }  
    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()".

## Ques P/Qs

- ① Unsigned int -  $a[4][3] = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}, \{10, 11, 12\}\}$ ;  
 $\text{printf}("1.u\1.u\1.u", a+3, *(a+3), *(a+2)+3);$   
 $\rightarrow 2036 \quad 2036 \quad 2036$

- ②  $B_A = 1000$ , 4 byte  
int  $a[2][3] = \{1, 2, 3, 4, 5, 6\};$   
 $\text{printf}("1.u\1.u\1.u", a, *a, **a);$   
 $\text{printf}("1.u\1.u\1.u", a+1, *a+1, **a+1);$   
 $\rightarrow 1000 \quad 1000 \quad 1012 \quad 1004 \quad 5$

- ③ int  $a[4] = \{10, 20, 30, 40\};$   
int  $*p[4] = \{a[3], a[2], a[1], a[0]\};$   
int  $y;$   
 $y = --p[0] - p[1];$   
 $\text{printf}("1.d", y);$   
 $\text{printf}("1.d", *p[0]);$   
 $\rightarrow 0 \quad 20$   
 $*p[0] = *(a[2])$   
 $= \underline{\underline{30}}$

- ④ int (\*f)(int\*);  
 $\rightarrow$  A pointer to a function that takes an integer pointer  
as argument and returns an integer.

- ⑤ int  $a=5, b=10, c=15;$   $*p[*p[*(p[1]-8)]]$   
int  $*p[3] = \{\&a, \&b, \&c\};$   $= *p[10-8]$   
 $\text{printf}("1.d", *p[*p[*(p[1]-8)]]);$   $= *p[2]$   
 $\rightarrow 15.$   $= c.$

⑥ void main()

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

Static int \*p[] = {a, a+1, a+2, a+3, a+4};

int \*\*ptr = p; ptr = &p[0];

ptr++; ptr = &p[1];

printf("%d.%d.%d", \*ptr, \*\*ptr); }

→ 1. 40

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

$$= \frac{20 - 10}{4} = \frac{10}{4} = 1$$

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

$$= \underline{\underline{40}}$$

⑦ #define print(x) printf("%d", x);

int x;

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

void P(int \*y){ int z = \*y + 2;

Q(z);

\*y = z-1;

print(z); }

void main()

{ x=5; O/p → 12 22 14 7

P(&x);

print(x); }

⑧ int \*A[10];

int B[10][10];

Of the following expressions which can not give L-value 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};

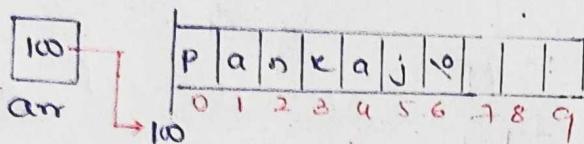
int \*p = (int \*)(&a+1);

printf("%d.%d.%d", \*(a+1), \*(p-1)); → 20 40

# Strings

\* String is a sequence of characters terminated by a null character.

Char arr[10] = "pankaj"; → String constant  
← Ascii code = 0



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'}; } Behaviour is undefined  
printf("%s", arr);

✓ { 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);  
→ pankaj

name = &name[0]

name++ = &name[0]+1  
= &name[1]

Char \*ptr = "Neeraj";

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

printf(ptr+1); → "eeraj".

Read only area

Char arr[] = "Neeraj";

arr = "pankaj"; ✗

Invalid,

arr++;

-- arr

arr--

++arr;

Invalid

Char \*ptr = "Neeraj";

ptr = "pankaj"; ✓

ptr++;

ptr--;

--ptr;

++ptr;

Valid.

→ ptr[1] = 'a'

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

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

arr[i] = 'A';

As it is stored in read/write area.

### Array.

- \* We can change individual element.
- \* Cannot assign completely new string.
- \* Increment/dec not possible.
- \* Read/write area

### Pointer

We can't change individual element.

Can assign completely new string

Increment/decrement possible.

Read only area

```

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

```

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); → TE

```

```

printf(arr + 3[arr]-arr[i])

```

```

arr + E - A

```

```

arr + X + 4 - X

```

```

arr + 4 = 10

```

∴ no output

eg: Char arr[] = "GATE 2023";

```

Char *ptr;

```

```

ptr = arr+3;

```

```

ptr = '10'; → arr+3 = 0

```

```

printf(arr); → GAT.

```

```

printf(ptr); → no output

```

( arr[3] = 0 )

= GAT 10

↓

String ends here.

① Char arr1[] = "pankaj"; }

Char arr2[] = "pankaj"; }

```

if (arr1 == arr2) printf("yes");

```

```

else printf("No");

```

both have different addresses

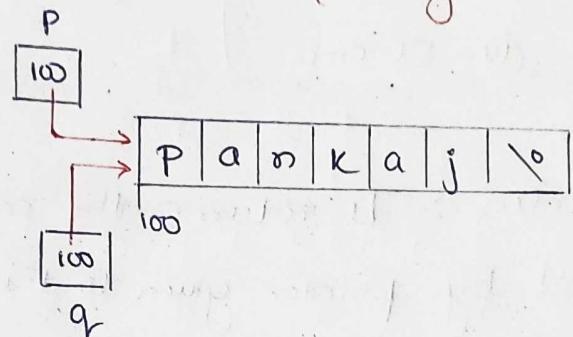
array name is address of  
first element

so Output = NO

② Char arr1[] = "pankaj";  
 Char arr2[] = "pankaj";  
 If (\*arr1 == \*arr2) pf("yes");  
 else pf("No");

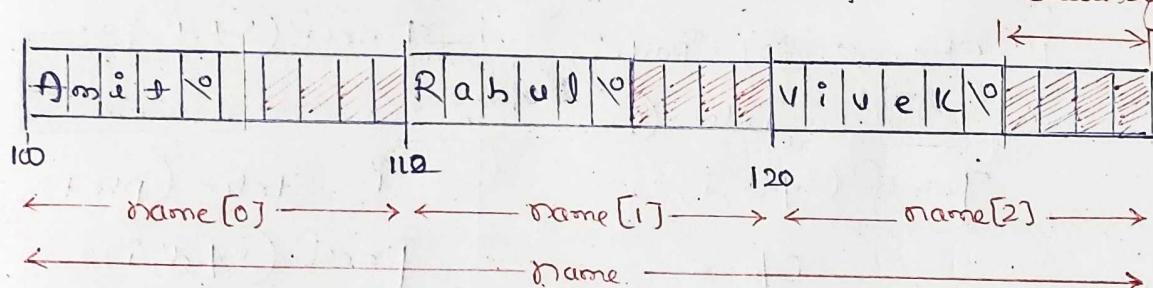
\*arr1 = &arr1[0] = 'P'  
 \*arr2 = &arr2[0] = 'P'  
 ∴ Output = "Yes".

③ Char \*p = "pankaj";  
 Char \*q = "pankaj";  
 If (p == q) pf("yes");  
 else printf("No");



### Multiple Strings

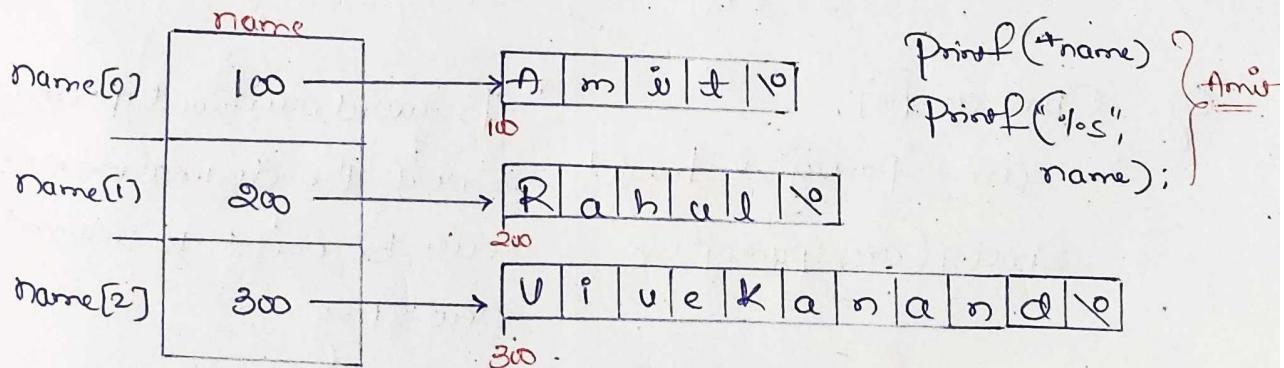
→ Char name[3][10] = {"Amit", "Rahul", "Vivek"};      Excess bytes



printf("%s", name[0]); → Amit  
 printf("%s", name[0]+2); → it  
 name[1][2] = 't'

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

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



## <String.h>

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

}      *Tutorial Functions on String*

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 \*destinations, Const Char \*source);

eg: Char arr[10];

arr = "pankaj"; ✗ Invalid

Strcpy (arr, "pankaj"); ✓

\* To avoid overflow type error

; size of the destinations memory must be enough to accommodate 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 pointed by  
destination pointer.

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

Char \*ptr = "Sharma";

Strcat (arr, ptr);

printf ("%s", arr); → Pankey Sharma.

④ strcmp: It compares two strings character by character

and returns 0 if both are same.

→ strcmp (String1, String2);

Char \*ptr1 = "pankey";

Char \*ptr2 = "pani";

{  
    0 → Same  
    <0 → Smaller  
    >0 → greater

strcmp (ptr1, ptr2)

'K'

$$k-i = true > 0$$

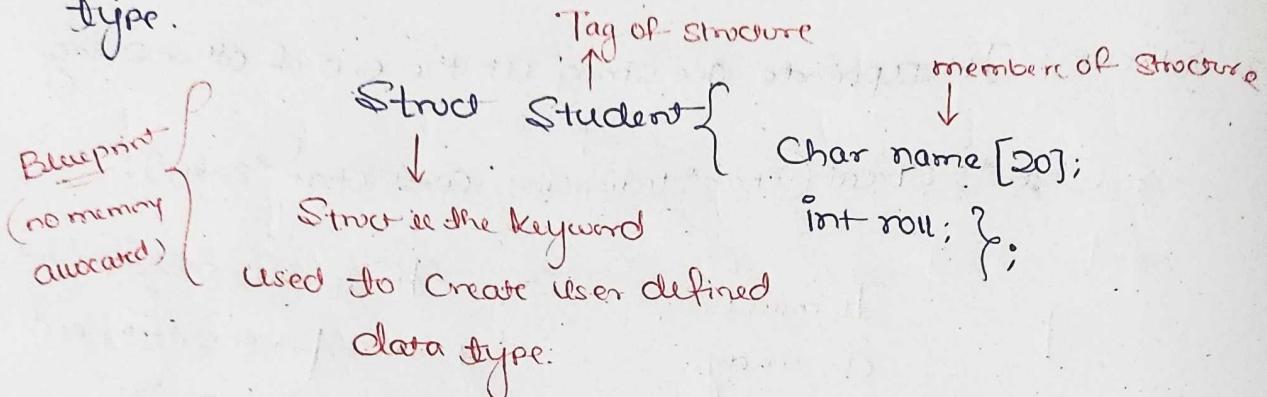
strcmp (ptr2, ptr1)

'i'

$$i-k = -ve < 0$$

## Structures and Union

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



\* When we Create a Structure, there will be new datatype like int, float, char.

→ Syntax to use : Struct student \_name ;

eg: Struct Student {  
    Char name [20];  
    Int roll; };

    } Structure blueprint template (Global)

    Void main()  
    {  
        Struct Student S;  
        S.roll = 10; };

    • => membership Operator.

### Ways of making Structure

① Struct {  
    Char name [20];  
    Int roll; } S1, S2;  
    } no tag.  
    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;      x Invalid/Error  
                                 $s2 = \{ 10, "pankaj" \};$  wrong order  
                                = { }; }

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

$s3 = \{ "pankaj" \};$  ✓

$s3.name = "pankaj";$  x Invalid

Solution ↓      ↗ name is array, so l-value can't be constant

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"; x Invalid/Error

strcpy(s[0].name, "pankaj"); ✓

strcpy(s[1].name, "Neeraj"); ✓

Struct Student s1 = s[0]; ✓ Valid

Struct Student  
 $s1 = \{ "4", 10 \};$

Struct Student  
 $s2 = s1;$  ✓

Valid

Struct date-of-birth { int day; }  
 int month; } template  
 int year; }

Struct Student { Char name[20]; → derived  
 int roll; → primitive }

Struct date-of-birth 'dob'; } → user defined  
 ↓ datatype

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)  
 { printf("%s", s.name);  
 printf("%d", s.roll); }

void display(Struct Student

pointer to structure ← → ptr

{ printf("%s", (\*ptr).name); }

"Pankaj"

→  $(\ast \text{ptr}).\text{member}$  → ptr is a pointer to structure, & we can access member like this.

→ ptr: pointer to structure

Syntax:  $(\ast \text{ptr}).\text{member}$  or  $\text{ptr} \rightarrow \text{member}$ .

### Self Referential Structure

Struct Panckaj { int i;

Struct Panckaj \*ptr; }

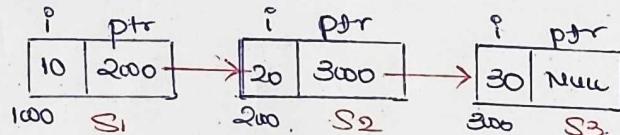
void main() {  
Member which is pointer to same data type.

{ Struct Panckaj 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 iff largest member size.
- \* Application of Union is "type pruning".

## Miscellaneous

## Comma Operator

- It works as a separator
  - It has the least priority as operator.
  - It works as an Operator.

$\rightarrow \text{int } x=10, y=20, z=30;$

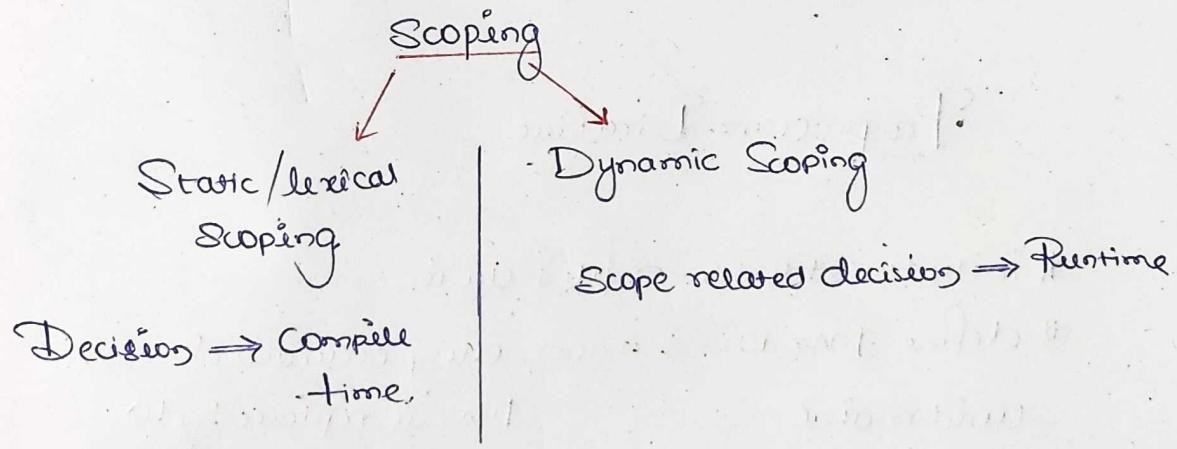
or

$\left\{ \begin{array}{l} \text{int } x=10; \\ \text{int } y=20; \\ \text{int } z=30; \end{array} \right\}$

$a = (10, 20, 30); \quad var = (\text{Exp}_1, \text{Exp}_2 \dots)$

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

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



→ 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(); }

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

With dynamic

Scoping: O/p = 20.

(Searches i; value  
from parent  
function)

Procedure f() { int i = 20;

    Call g(); }

Procedure g() { printf(i); }

Let  $\alpha$ : Value under static & dynamic Scoping.

Static main()

Scoping: ↓

f()

↓

g() - prints '1'

(Global  
variable as  
it's not there locally)

dynamic

Scoping: main(),

↓

f()

↓

g()

) Search  
(i; in

Parent  
Function

## Preprocessor Directive

#include <stdio.h> → File inclusion

#define MAX 10; → macro, every occurrence of

void main()

MAX is replaced by 10

{ printf("%d", max); }

→ In pre processor this line is

⇒ printf("%d", 10);

```

#define Square(x) x*x;
void main()
{
    int i;
    i = Square(5*3);
    printf("%d", i);
}
O/P = 25

```

$$\begin{aligned}
 i &= 5*3 * 5*3 \\
 &= 25
 \end{aligned}$$

## Size of Operator

- \* Compile-time Operator
- \* Unary Operator  $\Rightarrow$  1 Operand

variable  
 datatype  
 Expressions  
 Constant/literal

int i=4  
 Char = 1

int i=2, j=3;

pf ("%d", sizeof(i));  $\rightarrow 4$

pf ("%d", sizeof(i+2\*j)); — int+int+int = int  
 $= 4$

pf ("%d", sizeof(2));

pf ("%d", sizeof(int));

eg: int i=1, j;  $\nearrow$  not executed, as sizeof is Compile-time Operator  
 $j = \text{sizeof}(++i);$  &  $\text{int} + \text{int} = \underline{\text{int}}$

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

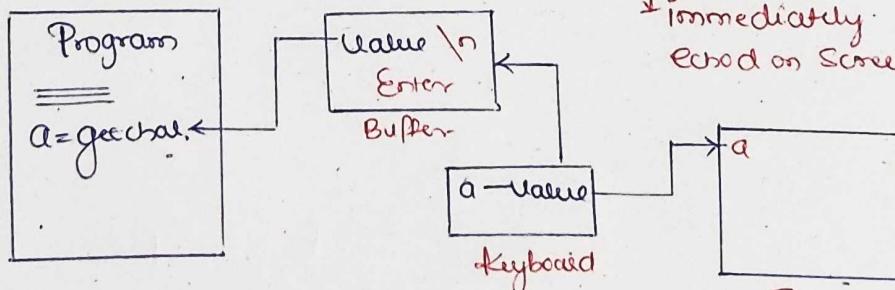
$\Rightarrow 1 4$

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

no parentheses ✓ O/P = 4.

## Getchar :

- ① buffer
- ② Echo.



## Getch :

- ① unbuffered
- ② unechoed.  $\rightarrow$  nothing echoe on screen.