
C Programming

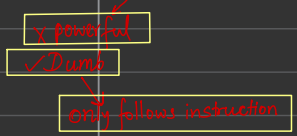
Weightage - (10-11 marks) (C & DS. Algo)

Program →
① Logic building
② Money

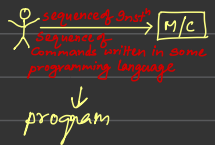
L1: Data Types and Operators:

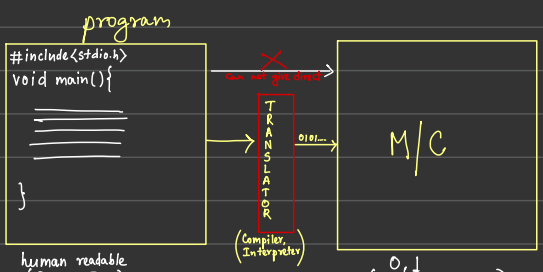
To interact with machines programming language is required.

Computer: Computer is a electronic device

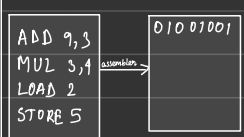


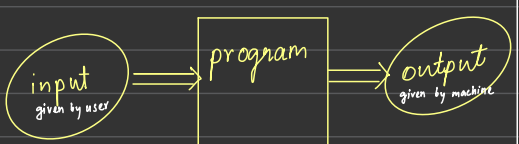
program
» Google Map
» ATM

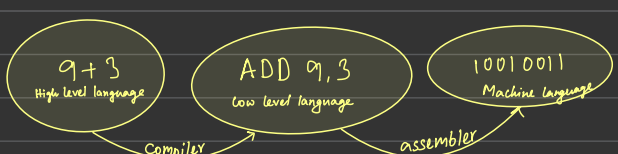




Why can't we write program in 0,1??
⇒ It is very hard to write or read a code in 0 and 1.
Human being are not good with number.







L2 25/09/23

Data Type

- **primitive**
  - a integer
  - b floating
  - c char
  - d void,
- **Derived**
  - a array
  - b pointer
  - c String
- **User Defined**
  - a Structure
  - b Union
  - c typedef

Integer

- short int → signed (by Default) / unsigned
- int → signed (D) / Unsigned
- long int → signed (D) / unsigned
- long long int → signed (D) / Unsigned

short int a = 2;
by default Signed
OR
signed short int a = 2;
} both are same

short int (2 byte) : 2^{16} combination
unsigned - 16 bit [0 to 65535]
Signed - (-32768 to 32767)

2147483648

int (4byte) 2^{32} Combination
unsigned - (0 to 4294967295) [0 to $(2^{32}-1)$]
signed - (−2147483648 to 2147483747) [-2^{31} to $2^{31}-1$]

n bits (2^n combination)
unsigned 0 to $2^n - 1$
signed -2^{n-1} to $2^{n-1}-1$

format specifier :
- %d → short int, int
- %c → char
- %f → float
- %lf → double

signed value <op> Unsigned value

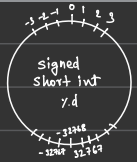
⇓

Signed value will be auto typecasted to Unsigned value

Symbol [1]	Type of operation		Associativity
`[ ]` `( )` `.` `->` `++` `--` (postfix)	Expression	P	Left to right
`sizeof` `&` `*` `+` `-` `~` `!` `++` `--` (prefix)	Unary	U	Right to left
typecasts	Unary		Right to left
`*` `/` `%`	Multiplicative	M	Left to right
`+` `-`	Additive	A	Left to right
`<<` `>>`	Bitwise shift	S	Left to right
`<` `>` `<=` `>=`	Relational	R	Left to right
`==` `!=`	Equality	E	Left to right
`&`	Bitwise-AND	B	Left to right
`^`	Bitwise-exclusive-OR		Left to right
`|`	Bitwise-inclusive-OR		Left to right
`&&`	Logical-AND	L	Left to right
`||`	Logical-OR		Left to right
`? :`	Conditional-expression	T	Right to left
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	Simple and compound assignment [2]	A	Right to left
`,`	Sequential evaluation	C	Left to right

L3

Short int (2 byte) → signed −32768 to 32767
→ unsigned 0 to 65535



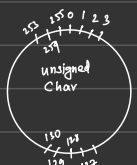
① short int x = -1;
by default printf("%d", x); (-1) o/p

② unsigned short int x = -2; 65534
printf("%u", x); (65534) o/p

③ short int x = -32769; 32767
printf("%d", x); 32767

Char datatype : (1 byte) $2^8 = 256$

unsigned char : 0 to 255
signed char : −128 to 127



ASCII : American Standard Code for Information Interchange

A → 65	a → 97	0 → 48	NULL = 0
B → 66	b → 98	1 → 49	Space = 32
C → 67	c → 99	2 → 50	
⋮	⋮	⋮	
Z → 90	z → 122	9 → 57	

④ char ch = 65;
printf("%d", ch); 65
printf("%c", ch); A

② char ch = 129; (ch = −127)
printf("%d", ch); −127
(signed)

③ char ch = 'A';
printf("%d", ch); 65
printf("%c", ch); A

⑤ char ch = −193;
printf("%c", ch); (Symbol) 63

④ char ch = −129; (ch = 127)
printf("%d", ch); 127

⑦ unsigned char ch = −129; (127)
Pf("%d", ch); 127

⑥ char ch = −191;
printf("%c", ch); A

−129 = 127
−130 = 126
−131 = 125
256 − 193 = 63

operators :

- Unary
- Binary
- Ternary

assignment operator (=) : (R → L)

a = b ; * it is binary operand

* left hand side must be variable.

Lvalue = Rvalue ;

- Lvalue → Variable having address in memory
- Rvalue → literal / constant, variable, expression

```
int a = 10;
printf("%d", a = 200);
```

o/p → 200

= operator returns the value which is assigned

a = b = c = 10 ; (① ② ③)

Q. int a, b, c ;
a = b = 10 = c ; CE:-
printf("%d %d %d", a, b, c);

o/p: CE : L value required

Arithmatic Operator :

+, − ⇒ Unary ex +7, −2, −3

binary :

precedence	associativity
/, *, %	L → R
+, −	L → R

(i) % → both operand must be int type

(ii) sign of the result in module → C standard : Undefined
→ Most Compilers : sign is same as the sign of first operand

-12%5 = -2
12%5 = 2

* Results of an operator depens on operend

① 12/5 = 2.4
= 2

② 12.0/5 = 2.4

③ 12/5.0 = 2.4

④ 12.0/5.0 = 2.4

⑤ int a;
a = 4.0 x 3 / 5 + 6 ; CE:-
printf("%d", a);

Relational Operators

* it is binary operator

i) <
ii) >
iii) <=
iv) >=
v) ==
vi) !=

* result of any relational operator is either 0 (false) or 1 (true) ;

precedence	Asso..
<, >, <=, >=	L → R
==, !=	

1] Unary
2] %, *, /
3] +, −
4] <, <=, >, >=
5] ==, !=
6] = (assignment) → R → L

(2] to 5] : L → R)

printf(" "); → printf returns no of character successfully printed on screen

Q. int a = printf("%d", printf("GATE 2023"));
printf("%d", a);

O/P
```
GATE 2023
9
```

Logical operator:

1. Logical and (&&) [Binary] → if both operand non zero then result is 1 else 0
2. Logical or (||) [Binary] → if any one operand non zero then 1 else 0
3. Logical not (!) [unary] → it complements the result (non zero → zero, zero → non zero)

Short Circuit Operation:

AND ① in && if first operand is 0 we need not to check the second operand

OR ② in || if first operand is 1 we need not to check the second operand

Q. a = printf("Gate") || printf("2023");
printf("%d");
O/P Gate 1

Q. int a;
a = 2<5 || 5>10 == 8>10;
printf("%d", a); O/P = 0

precedence:
1. Unary + - !
2. *, /, %
3. +, -
4. <, >, >=, <=
5. ==, !=
6. &&
7. ||
8. =

int a;
a = 2!=5 == 3 && 5>10 == 10>6;
printf("%d", a); Ans 0
This part not executed due to short circuit

1 != 2<5 && 8>10 == 8>10
1 != 1 && 8>10 == 8>10
1 != 1 && 0 == 0>10
1 != 1 && 0 == 0
0 && 1 == 0
0 && 1
= 0

Modify

Increment
- pre increment (++a)
- post increment (a++)

Decrement
- pre decrement (--a)
- post decrement (a--)

preincrement: first increment value then use
post increment: first use the value then increment

predecrement: first decrement value then use
postdecrement: first use the value then decrement

```
int a = 5, b;
b = ++a + a++ + ++a;
printf("%d %d", a, b);
```

Sequence points
b/w two successive sequence point we can modify the value of a variable at most one time

Ans: Compiler depedent

Bitwise Operator:

1) Bitwise AND (&)
2) Bitwise OR (|)
3) Bitwise XOR (^)
4) Bitwise NOT (~) ~a = -(a+1)
5) Bitwise Left Shift (<<)
6) Bitwise Right Shift (>>)

Q1. int a = 5, b = 17, c;
c = a & b;
Pf("%d", c); 1
decimal

```
void main()
{ 12;
  12.23;
}
```
no o/p

Ternary Operator: (?:)

exp1 ? exp2 : exp3;

True / False

```
int a;
a = 9 != 3<5 ? 1 == 8!=3 ? 10 : 20 : 30;
```
0 != 3 → 1
1 (true)
a = 10

```
int a=2, b=2, c=0, d=2, e;
e = a++ && b++ && c++ || d++;
```
a: 2 3 → 3
b: 2 3 → 3
c: 0 1 → 1
d: 2 3 → 3
e: 1

(2 && 2) && (c++) || d++
(1 && c++) || d++
(1 && 0) || d++;
0 || d++;
0 || 2
= 1

1. `int a;`

`a = (15 < 2) ? 1 != 12 > 15 ? 10 : 20 : 4 < 7 != 7 > 9 ? !5 ? 30 : !1 == 1 ? 4 : 5 : 6;`

(exp1 = 15 < 2 → false; exp2 = 1 != 12 > 15 ? 10 : 20; exp3 = 4 < 7 != 7 > 9 ? ... ; 4 < 7 → true, 7 > 9 → false, exp1 = true; !5 → exp1 = false; !1 → 0, 0 == 1 ... true)

Ans = 4

2. `int a;`

`a = 5<12 ? printf("GATE") && printf("Wallah") || printf("2023") : printf("Sir");`

(exp1 = true; exp2 = printf("GATE") && printf("Wallah") || printf("2023") → 4 && 6; exp3 = printf("Sir"))

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

O/P : GATEWALLAH1

3. `int x = 12, y;`

`y = (x&1) ? printf("Hellow") : printf("Everyone");`

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

(1100 → 12 & 0001 → 1 = 0000)

x&1 → 0 if x is even
→ 1 if x is odd

O/P : Everyone8

4. 5 ^ 5 = 0

5^5^5 = 0^5 = 5

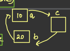
```
   000101
 ^ 000101
   000000
 ^ 000101
   000101
```

5^5^5^5 = 0

~~5~~^~~1~~^~~2~~^~~3~~^~~3~~^~~1~~^4 = 4^0 = 4

Q) `int a, b; a = 10, b = 20, c;`

// swapping 1 (using 3rd variable)
```
c = a;
a = b;
b = c;
```



// without using 3rd variable
```
a = a+b;
b = a-b;
a = a-b;
```

// using ^ (XOR) operator
```
a = a^b;
b = a^b;
a = a^b;
```

Short hand Oparator:

5.
```
x = x+10   => x += 10;
x = x-10   => x -= 10;
x = x*10   => x *= 10;
x = x/10   => x /= 10;
x = x%10   => x %= 10;
x = x&10   => x &= 10;
x = x|10   => x |= 10;
x = x<<10  => x <<= 10;
x = x>>10  => x >>= 10;
x = x^10   => x ^= 10;
```

Q) `int a = 5, b = 10;`
```
b = (a+b) - (a=b);
printf("%d", a); 10
printf("%d", b); 5
```
a: ~~5~~ 10 b: ~~10~~ 5

15 − 10

6.

Dec	bin
2 =	10
4 =	100
8 =	1000

2^x = binary values always Contain Single 1

bits addition

1 + 0 = 1
0 + 1 = 1
0 + 0 = 0
1 + 1 = 2 = 10 (carry, sum)

```
  001010      0001111     0000111
+ 000110    + 0010000   +       1
  010000      0011111     0001000
```

```
  (15)   0000 1111
+ (1)    000 0 0001
         0001 0000
```

Sum. GP = $\frac{a(r^n - 1)}{r-1}$

1 + 2 + 4 + 8 + 16   r = 2, n = 5
$= 2^0 + 2^1 + 2^2 + 2^3 + 2^4$
$= \frac{1.(2^5 - 1)}{2-1}$
= 31

```
  15 - (0001111)
& 16 - (0010000)
        0000000
```

Q.) given a number. How can you find the number can be written in power of 2 using oparator:-

```
int x ; // let the +ve no is x
result = x &(x-1)
if( !result)
{
  // Can be written in power of 2
}
else
{
```

'sizeof' oparator
↳ unary
↳ Compaile time answer;
↳ result is unsigned int.

sizeof(int) =>
sizeof(i+10) => int
↳ compile time

ex int + int => int
```
int a = 10;
sizeof(a) => 4
float b = 20.8;
sizeof(b) => 4
```

ex
```
int i=10;
printf("%u", sizeof(i=i+10)); => 4
printf("%d", i); // 10
```
(i=i+10 → not executed)

ex sizeof(10 + 5.6);
5.6 ↳ literal ↳ Considered a double
↳ // ans : sizeof (double) = 8

Flow Control

if statement:

- ++a
  - Increase a by 1
  - use the updated value
- a++
  - use the value
  - Increase a by 1

```
if (Condition/Expression)
{
    S1;
    S2;
    S3;
}
```

non-zero → S1; S2; S3;
zero → skip

scope

// if curly braces not provided then scope is upto first semi Colon(;)

Eg:

```
int i=0;
if(i==0)
printf("Pankaj");
```

o/p:- pankaj

```
int i=0;
if(i=0)
printf("Pankaj");
```

o/p:- pankaj

```
if(0)
printf("Pankaj");
```

→ no output

```
if(12-78)
printf("2");
printf("3");
```

o/p:- 23

```
if( ) no expression
printf("Pankaj");
```

Compile time Error

```
if (12+3x4)
printf("2");
printf("3");
```

o/p:- 23

Q) int i=0;
if(i++){
printf("Pankaj");
}
→ No output

Q) int i=1;
if(--i)
printf("2");
printf("%d",i);
o/p:- 0

Q) int i=3;
if(i<2);
printf("Hello");
o/p:- Hello

Q. int x=10
printf("%d", x<<1);
printf("%d", x);
o/p:- 2010

Q. WAP to read a no if it is even then print "pankaj";

```
#include<stdio.h>
main()
{ int a;
printf("Enter a number");
scanf("%d",&a);
if(a%2==0)
printf("Pankaj");
}
```

```
if(a&1==0)
printf("Pankaj");
```

or else

```
if(a==((a>>1)<<1))
printf("Pankaj");
```

if-else statement:

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

true/Non zero → S1; S2;
false or zero → S3; S4;

// else without if cannot be used
// only if can be used
//

if-elseif statement:

```
if(expression1){
}
else if(expression2){
}
else if(expression3){
}
```

expression1: true/non zero → ; false/zero → expression2
expression2: true/non zero → ; false/zero → expression3
expression3: true/non zero → ; false/zero →

if-elseif-else statement:

```
if(expression1){
}
else if(expression2){
}
else{
}
```

expression1: true/non zero → ; false/zero → expression2
expression2: true/non zero → ; false/zero → else

WAP to find largest among 4 distinct number?

Max = (a>b && a>c && a>d) ? a : (b>c && b>d) ? b : (c>d) ? c : d;

L2 loop

*No of iteration known then go for for loop

Iterative Statements:

times = (last - first + 1)

// all 3 expressions are optional

for (; ;) semicolon mendatory

```
for (exp1; exp2; exp3)
{

}
```

// if exp2 is not given then compiler will treat it as a non zero value.

```
for( ; ; )
{
    printf("Hello");
}
```

∞ times

```
for (initialization ; Condition ; incr/decr)
{
    //Code
}
```

① initialization → ②⑤ Condition → ③⑥ //Code → ④⑦ incr/decr → back to Condition; if false → exit

```
for (roll = 1 ; roll <= 5 ; roll = roll+1)
{
    printf("Pankaj");
}
```

roll	
1	True
2	True
3	True
4	True
5	True
6	false

5 times

Q.
```
char ch = 1;
for (ch = 1 ; ch ; ch++)
    printf("Pankaj");
printf("END");
```
1 2 3 . . 127 -128 - - - -1 0
255
total = 255 times

Q.
```
char ch = 1;
for (ch = 1 ; ch ; ch = ch+2)
    printf("Pankaj");
printf("END");
```
1 3 5 . . 127 -127 -125 -1 1
∞

Q.
```
int i = 1;
for (printf("1"); i <= 5; printf("3"))
{ printf("2");
  i = i++;
}
```
output: 1 2 3 2 3 2 3 2 3 2 3

Q.
```
for (10; 11; 12)
{
    printf("Hello");
}
```
// infinite times pankaj

Q.
```
int i = -1;
for (i++; i++; i++)
{ printf("Pankaj");
}
```
0 times

Q.
```
char i = -1;
for (i++; ++i; i++)
{ printf("Pankaj");
}
```
infite time

Q.
```
int i = 1;
for( ; ++i < 5; )
{ printf("%d", i);
}
```
o/p: 2 3 4

```
int i = 1;
for( ; i++ < 5; )
{
    printf("%d", i);
}
```
o/p: 2 3 4 5

1 < 5 → T
2 < 5 → T
3 < 5 → T
4 < 5 → T
5 < 5 → F

```
for (i = 1 ; i <= n ; i++)
    printf("Hi");
```
→ n times

n is positive
```
for (i = 1 ; i <= n ; i = i+2)
    printf("Hi");
```
→ $\lceil \frac{n}{2} \rceil$ times

```
for (i = 1 ; i <= n ; i = i*2)
    printf("Hi");
```
→ $\lfloor \log n \rfloor + 1$

1 → 1
2 → 2
3 → 2
4 → 3

$i = 1, 2, 2^2, 2^3, \ldots, 2^k$
$= 2^0, 2^1, 2^2, 2^3, \ldots, 2^k$

loop runs k+1 times

L9 Nested loop

```
for(i=1; i<=n; i++)
{
  = code
}
// n times
```

```
for(i=1; i<=3; i++)
{
  for(j=1; j<=4; j++)
  {
    printf("Pankaj");
  }
}
```

i=1 j=1 2 3 4
i=2 j=1 2 3 4
i=3 j=1 2 3 4

total 12 times pankaj printed

Q.
```
for(i=1; i<=3; i++)
{
  for(j=1; j<=4; j++)
  { printf("%d %d", i, j); }
}
```
op: 1 1 1 2 1 3 1 4
2 1 2 2 2 3 2 4
3 1 3 2 3 3 3 4

Q
```
for(i=1; i<=n; i++)
{
  for(j=1; j<=n; j=j*2)
  {
    printf("Pankaj");
  }
}
```
$n*(\lfloor \log_2 n \rfloor)$ times

Q.
```
for(i=2; i<=n; i=i*2)
{
  printf("Pankaj");
}
```
$\lfloor \log_2 n \rfloor$ times

Q.
```
for(i=1; i<=3; i++)
{
  for(j=1; j<=i; j++)
  { printf("Pankaj"); }
}
```
i=1 1
i=2 2
i=3 3

$\sum_{i=1}^{3} i = \frac{i(i+1)}{2}$ = 6 times

Q.
```
for(i=1; i<=n; i++)
{
  for(j=i; j<=2*i; j++)
  { print("Rudra"); }
}
```
$2i - i + 1 = i + 1$

$\sum_{i=1}^{n} i+1$

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

Q.
```
for(i=1; i<=n; i++)
{
  for(j=i; j<=2*i; j++)
  { printf("Pankaj"); }
}
```
$\sum_{i=1}^{n} 3i - i + 1$

$= \sum(2i+1)$

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

$= n(n+2)$

Q.
```
for(i=1; i<=n; i=i*3)
{
  for(j=i; j<=n; j++)
  { printf("Pankaj"); }
}
```
$\lfloor \log_3 n \rfloor + 1$

$(k+1)(n+1) - \left(\sum_{i=0}^{k} 3^i\right)$

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

$* (\lfloor \log_3 n \rfloor)$

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

i=1 n-1+1
i=3 n-3+1
i=3² n-3²+1
i=3^k n-3^k+1

check answer

$1+2+3+\ldots+n = \frac{n(n+1)}{2}$

$1^2+2^2+3^2+\ldots+n^2 = \frac{n}{6}(n+1)(2n+1)$

$1^3+2^3+3^3+\ldots+n^3 = \left(\frac{n(n+1)}{2}\right)^2$

While loop :

* expression is mendatory in while loop
* no of iteration unknown go for while loop

```
while(expression)
{
  // Code
}
```

```
int i=1
while(++i<5)
{ printf("%d", i); }
```
// op:- 2 3 4

2<5 T
3<5 T
4<5 T
5<5 F exit

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

do-while loop :

* used in menu driven program
* loop runs atleast 1 time

```
do{
  // code
}while(expression);
```

WAP to find sum of n natural no.

```
int sum=0;
for(int i=1; i<=n; i++){
  sum = sum + i;
}
printf("Sum = %d", sum);
```

L9

break :

break exist from current loop

ex
```
for(i=1; i<=10; i++)
{
  if(i%4==0)
    break;
  printf("%d",i);
}
```
op: 1 2 3

continue :

Skip remaining code of current iteration and continue with next iteration.

```
i=3
for(i=1; i<=5; i++)
{
  if(i%3==0)
    continue;
  printf("%d",i);
}
```
op:- 1 2 4 5

skip iteration for i=3

Switch statement

* used to create selection statement with multiple choice
* multiple choice are provided with another keyword : case

```
switch(expression){        // Evaluates to integer
  case constant1 :
      // block of stmt
      break;
  case constant2 :
      // block of stmt
      break;
  case constant3 :
      // block of stmt
      break;
  default :
      // block of stmt
      break;
}
```

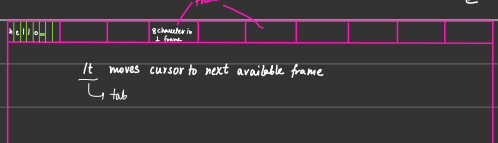
ex1:
```
switch(n){
  case 1: //Code we want to execute if the value of n is 1.
          break;
  case2 : //Code we want to execute if the value of n is 2
          break;
  default : //Code we want to execute if the value of n do not match with any case level;
}
```

* break is optional
* expression ⟶ evaluate to a int value. x, 'A', 22, allowed
  12.3, 15.5*3 not allowed
* position of default does not matter, it can be anywhere → start, end, ----
* default is optional
* Duplicate case labels are not allowed case 'A': case 65:
* Case label cannot be a variable. case a :
* range ⇒ ex case lowvalue ... highvalue (supported only on latest compiler)
* set of value ⇒ ex

```
Case 1:
Case 2:
Case 3:
    printf("hello");
    break;
Case 4:
Case 5:
Case 6:
    printf("hi");
    break;
```

```
n=2
switch(n){
  n=n+2;        // never get executed
  case 3;
  case 2:
  case 1:
}
```

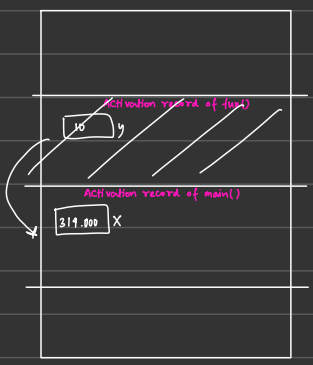
```
switch(n){ }   ✓
switch(n)
{
  i=i+1;
}              ✗
// no error
```



Function & Storage Classes

- printf
- scanf

↳ provides code reusability



Main Memory

* Copy of value is passed

```
#include<stdio.h>
void main(){
  int a =10;
  printf("a = %d");
}
```


```
#include<stdio.h>
void main(){
  int x;
  x=fun(10);
  printf("%d", x);
}
double fun(int y){
  return 3.14*y*y;
}
```

by default returntype is int; stored in symbol table

here it is double → error

Formal Parameter: fun(int x, int y) — value in caller function *in definition

Actual Argument: value in caller function — fun(10, 20);

* to avoid compilation error forward declaration is used
* function prototype of standerd function is available in header file
* body is present in .dll file
* linker has info about main() in .o file
* by default returntype is int if not provided

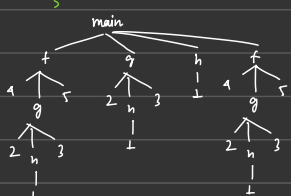
```
eg  mul(int,int);
    void main(){
    }
    int mul(int, int)
    {
      return 0;
    }
```

void → Empty/nothing

return value of a fun^n may or may not be used.

```
main(){
  f();
  g();
  h();
  f();
}
h(){ printf("1"); }
g(){ printf("2");
     h();
     printf("3");
}
f(){ printf("4");
     g();
     printf("5");
}
```

op: 4 2 1 3 5 2 1 3 1 4 2 1 3 5



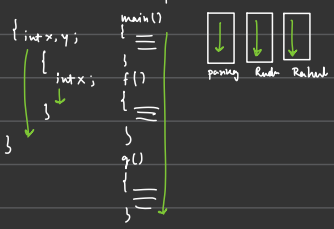
12

Storage class :

* scope : part of code in which a variable is visible
* lifetime : Duration active/alive
* default value : if we don't initialize a variable, what will be value
* Storage area : where a variable is stored

type of scope :

block scope : File Scope : multifile :



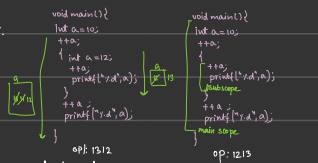
auto :

* by default variable declared inside a function are <auto>.

"auto" keyword is optional.

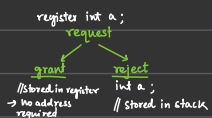
scope → block in which they are declared
lifetime → block in which they are declared
default value → garbage
Stored in → stack (activation record of the fun^n)

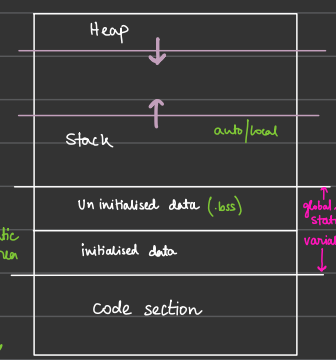
* created automatically when we enter the block in which they are declared and destructed automatically when exit the block.
* main scope variable are accessible in sub-scope
* sub-scope variable are accessible in main scope



register :

* same as auto
* Storage : cpu register / stack





static : → allocation in compile time

lifetime : program
Scope : block → internal
default value : 0
storage : static area

① value persist b/w different function call
② No re-initialization

```
void fun(){
  static int i=0;      // executed only once
  i++;
  printf("%d",i);
}
main(){
  fun();
  fun();
  fun();
}
```

o/p : 1 2 3

```
void fun(){
  static int b=0;
  int a;
  b++;
  printf("%d",a);
}
main(){
  int a=12;
  b++;
  printf("%d%d",a,b);
}
```

o/p : 1 2 3

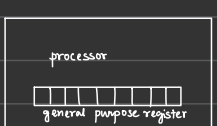
error : b unknown

```
int a ;
static int b=a;
```

* error :
can not assign a static variable with local variable
* initialize with literals.

global variable : variable which is not declared inside any function

lifetime : program
Scope : external
Storage : static area
default value : 0



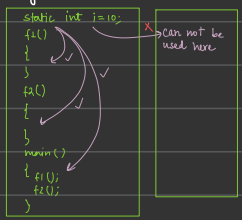
extern :

* used for forward declaration for global variable
* no memory space allocated.

```
void f1(){
  ++x;
  printf("%d",x);
}
int x=10;
void f1(){
++x;
}
main()
{
f1();
f2();
}
```

→ error : Sol^n : extern int x;

* Static global variable can be accessed from the file only



* normal global variable can be used in other file using extern.

* Can not initialize a static variable with an auto variable or other static variable.

```
int main()
{ int b=10;
  static int i=b;
}
// error
```

```
int main()
{ static int b=10;
  static int i=b;
}
// error
```

* Static variable can be initialized with constant literals.

```
int main()
{ static int b=10;
}
// no error
```

>> declaration for compiler

```
* void main(){
    static int i;
    static int i;
  } error
// redeclaration of static variable is not allowed
```

```
* static int i;
  static int i;
  void main(){
  ========
  ========
  }
// valid as it is global variable
re declaration allowed
Compiler save details to symbol table
```

```
static int i;
static int i;
static int i=12;
void main(){
========
========
}
// valid
```

```
static int i;
static int i=10;
static int i=12;
void main(){
========
========
}
// error
redefination not allowed
```

Static
- global
  - redeclare allowed
  - redefination not allowed
- local → redeclare not allowed

```
f1()
{extern int x;
}
f2(){
  extern int x;
}
```

```
int i=10;
i=200;
// warning
as not executed
redefination not allowed
```

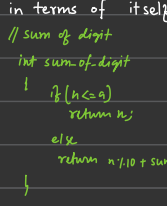
```
static int i=10
i=200;
// non static declaration follows static declaration.
```

Recursion: when an operation is defined in terms of itself

```
if(n is small)
  { we can answer directly
  }
else
  { input is large
  }
```

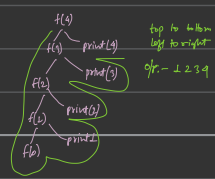
```
fun(int n)
{ if(n==1)
  {printf("Rudra");
  }
  else
    fun(n-1);
}
```

```
// sum of digit
int sum_of_digit
{
  if(n<=9)
    return n;
  else
    return n%10 + sum_of_digits(n/10);
}
```

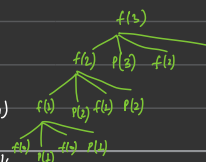


* If somecode is written after recursion call then it is done in reverse

```
eg: void f(int n)
    { if(n==0)
        return;
      else{ f(n-1);
            printf("%d",n);
          }
    }
```



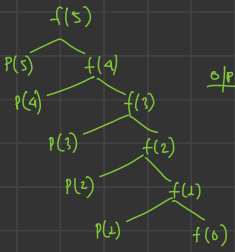

```
void f(int n){
  if(n==0)
    return;
  else{
    f(n-1);
    printf("%d",n);
    f(n-1);
    printf("%d",n);
  }
}
main()
{ f(3)
}
```



o/p: 1 1 2 1 1 2 3 1 1 2 1 1 2 3

```
void f(int n)
{ if(n<=0)
    return;
  printf("%d",n);
  f(n-1);
}
```

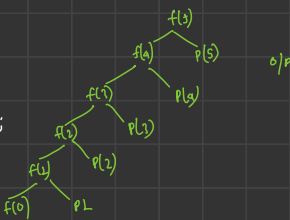
Output for f(5);



o/p: 54321

```
void f(int n)
{ if(n<=0)
    return;
  f(n-1);
  printf("%d",n);
}
```

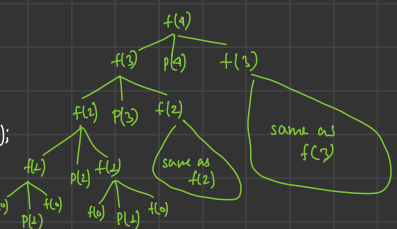
Output for f(5);



o/p: 12345

```
void f(int n)
{ if(n<=0)
    return;
  f(n-1);
  printf("%d",n);
  f(n-1);
}
```

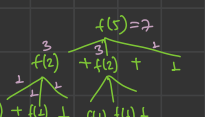
Output for f(3);



o/p:- 1213121412131213121

```
int f(int n){
  if (n<=1)
    return n;
  return f(n/2)+f(n/2)+1;
}
```

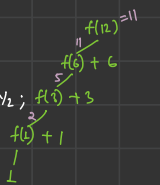
op: f(5)



o/p: 7

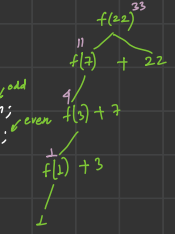
```
int f(int n)
{ if(n<=1)
    return n;
  return f(n/2)+n/2;
}
```

o/p: f(12)



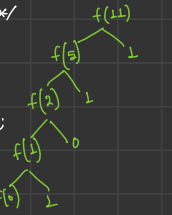

```
int f(int n)
{ if(n<=1)
    return n;
  if(n%2)
    return f(n/2)+n;
  return f(n/2)+n;
}
```

o/p: f(22)




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

op: f(11)

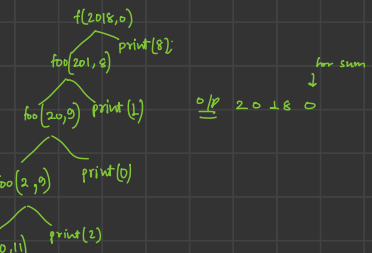


o/p: 1011

Consider the following C program:

```
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:



Q.
```
void main()
{ static int var=5;
  printf("%d",var--);
  if(var)
  main();
}
```

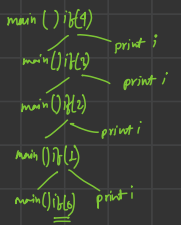
o/p 54321

var 5 4 3 2 1 0

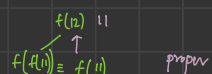
```
void main()
{ static int i=5;
  if(--i)
  { main();
    printf("%d",i);
  }
}
```

i: 4

o/p:- 0000

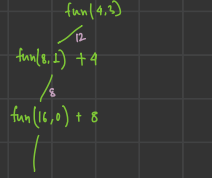


11.
```
int fun(int x)
{ if(x%2==0)
    return fun(fun(x-1));
  else
    return (x++);
}
int main()
{ printf("%d", f(12));
  return 0;
}
```



proper repeat

12.
```
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(9,3));
  getchar();
  return 0;
}
```



A) 12 B) 81 C) 64 D) 8

```
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;
}
```

f(5) 9 r=2
f(4) + r
f(3) + r
f(2) + r
f(0) + 2
1

What is value of f(5);
A) 5 B) 7 C) 9 D) 18

19. Consider the following recursive C fun"

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

output for: foo(513,2)
A) 9 B) 8 C) 5 D) 2

foo(513,2)
1 + foo(256,2)
0 + foo(128,2)
0 + foo(64,2)
0 + foo(32,2)
0 + foo(16,2)
0 + foo(8,2)
0 + foo(4,2)
0 + foo(2,2)
0 + foo(1,2)
1 + foo(0,2)

a = (x, y, z)

15. Which of the following statements is/are valid?
a) return a+b b) return a,b,c ;
c) return (a,b,c); D) all of them

16.
```
int fun(int x)
{ if(x>5)
    return fun(x-4) + fun(x-1) + 1;
  return 1;
}
```
find f(12); // 51

f(12) =13
f(8) + f(11) + 1
f(4) + f(7)+1 f(7)+f(10)+1
f(0)+f(3)+1 f(3)+f(6)+1 f(6)+f(9)+1 f(9)+f(12)+1
f(3) f(6)+1 f(6) f(9)+1 f(9) f(12)+1
f(2) f(5)+1 f(5) f(8)+1 f(8) f(11)+1
f(1) f(4)+1 f(4) f(7)+1

i	f(i)
0	1
1	1
2	1
3	2
4	3
5	5
6	7
7	10
8	13
9	19
10	28
11	37
12	51

Q. predict the output of following pgm.

```
int fun(int n)
{ if(n==0)
    return n;
  else return 2*fun(n+1)
}
int main()
{ printf("%d", fun(2));
  return 0;
}
```

A) 1 B) 8
C) 16 D) error

fun(2)
2* fun(3)
2* 2 fun(4)
1

18. Consider the following recursive fun" fun(x,y). What is the value of fun(9,3)

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

A) 13 B) 12
C) 9 D) 10

fun(9,3)
fun(3,7)
fun(2,10)
fun(1,12)
fun(0,13)
13 Ans

Q. int fun(int x, int y)

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

What does the following fun" do?
A) x+y B) x + x*y
C) x*y D) pow(x,y)

fun(x,y)
x+ fun(x,y-1)
x+x+x+ ..... y time
xy

20. What does fun2 do in general?

```
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]];
}
```

fun2(a,b) ≡ a^b
fun[a, fun2[a,b-1]]
fun[a, fun2[a,b-2]]
fun[a, fun2(a,1)] ≡ a*a
fun[a, fun2(a,0)]
≡ fun(a,1) ≡ a
fun(x,y) = x*y*y
pow(x,y)

A) x * y B) x + x*y
C) pow(x,y) D) pow(y,x)

21. output of following program:

```
void print(int n){
  if(n>4000)
    return;
  printf("%d ", n);
  print(2*n);
  print(2*n);
}
int main()
{ print(1000);
  getchar();
  return 0;
}
```

/// 1000 2000 4000 4000 2000 1000

print(1000)
P(1000) print(2000) P(1000)
P(2000) print(4000) P(2000)
P(4000) print(8000) P(4000)

22. What does the following function do?

```
int fun(unsigned int n)
{ if(n==0||n==1)
    return n;
  if(n%3!=0)
    return 0;
  return fun(n/3);
}
```

A) it returns 1 when n is a multiple of 3, otherwise returns 0
B) it returns 1 when n is a power of 3, otherwise returns 0
C) it returns 0 when n is a multiple of 3, otherwise returns 1
D) it returns 0 when n is a power of 3, otherwise returns 1

fun(6)
fun(2)
0
fun(9)
fun(3)
fun(1)
1

23. predict the output of following program

```
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;
}
```

A) stack overflow B) 3
C) 4 D) 5

f(11) 5
f(5) + f(6) 2
f(2) + f(3) 2
1 + 1
f(1)
f(1)

29.
```
int f(int n)
{ static int i=1;
  if(n>=5)
    return n;
  n = n+i;
  i++;
  return f(n);
}
```

return value of f(1) = 7

f(1) n=2
f(2)
f(4) n=4
f(7) n=7
7
i = 1 2 3 4

25.
```
int fun(int n)
{ int x=1, k;
  if(n==1) return x;
  for(k=1; k<n; ++k)
    x = x + fun(k) * fun(n-k);
  return x;
}
```

value of fun(5):

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

Ans 51

n	fun(n)
0	1
1	1
2	2
3	5
4	15
5	51

26.
```
void opt(int n)
{ if(n<1) return;
  opt(n-2);
  opt(n-3)
  printf("%d", n);
}
```

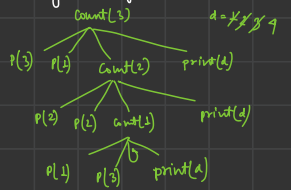
opt(6)
opt(5) opt(3)
opt(4) opt(2) opt(2) opt(0)
opt(3) opt(1) opt(1) opt(-1)
opt(2) opt(0) opt(0) opt(-2)

n	opt(n)
-2 -1 0	1
1	3
2	5
3	7
4	11
5	17
6	25

Ans: A) 25 C) 35 D) 45

Q. What will be output of the following C program?

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



o/p : 3 1 2 2 1 3 4 4 4

Q28. What will be output of the following C program?

```
#include<stdio.h>
int main()
{
  function();
  return 0;
}
void function()
{
  printf("Function C is awesome");
}
```

A) Function C is awesome B) no output

C) Runtime error D) Compilation error

Q. #include<stdio.h>

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

A) Runtime Error B) Compilation Error

C) 0 D) None of these

Array & address :

& : address

* : value at operator

Array is collection of Homogeneous data type

a: [10]

&a → 2036

a = 10
&a = 2036
*(&a) = 10
≡ value at (Memory location (2036))

a ≡ *& a

Address → absolute address
Address → relative address

int a[3]; a is a group of 3 elements each of type int

*one after another sequence
* In C, index always start from 0.

a: 1000 | 1004 | 1008 (index 0, 1, 2; 4B each)

* array name is constant, it stores the address of first element of the array.

size → int, float, -ve, +ve

a[size] → 0, 1, .., size-1

size should be unsigned integer; Standard

int a[0]; → undefined
⇒ online, ok
a[-1] ⇒ ok in online

int a[4]; [G | G | G | G]
int a[4] = {10,20,30,40}; [10 | 20 | 30 | 40]
int a[4] = {10,20}; [10 | 20 | 0 | 0]
int a[4] = {10,20,30,40,50,60}; ignored [10 | 20 | 30 | 40]
int a[] = {10,20,30,40}; [10 | 20 | 30 | 40]
int a[]; // error size mendatory

int a[3]; → declaration info given to compiler
a = {10,20,30} error
L-value = R-value

a[0] = 10;
a[1] = 20;
a[2] = 30;
update ⇒ variable

array_name = ... → Constant, invalid

int a = {10}; valid in C.

* Array name can not be Lvalue of any assignment statement.

* Array name ⇒ constant
ex 2++ invalid ; Array_Name ++ ; Array_Name -- ; ++Array_Name; --Array_Name; invalid operation

1. Collection of homo. type of element
2. Elements are stored sequentially
3. Array_Name ⇒ Constant
4. Array_Name ⇒ Address of first element
5. ++Array_Name, --Array_Name, Array_Name++, Array_Name-- invalid
6. int a[]; invalid
7. int a[]={10,20};
8. Relative addressing.

&a ⇒ address of whole array

a[0] a[1] a[2] a[3]
[10 | 20 | 30 | 40]
4B | 4B | 4B | 4B element
16 B whole array

```
void main(){
    int a[4]={10,20,30,40};
    printf("%u", a); //1000
    printf("%u", &a); //1000
    printf("%u", &a[0]); //1000
}
```

Same numerical value

a ≡ &a[0]

Name of array does not represent address with 2 operators → &, sizeof

&a → address of whole array

*If declaration of array is having n-dimensions and
(i) any where in prog. you provide exactly n-dimension ⇒ value
ii) any where in prog. you provide less than n-dimension ⇒ address

ex: int a[4] = {10,20,30,40}
a → 0 dimension → Address
a[0], a[1], a[2] → 1 dimension → value

ex int a[2][3] = {1,2,3,4,5,6}
a → 0 dim ⇒ Address
a[0], a[1] → 1 dim ⇒ Address
a[1][2] → 2 dim ⇒ value

Address arithmatic :

address + value = Address
value + address = Address
valid

address + address ⇒ invalid

1) What will be a+1?
→ if a is value the normal arithmatic
if a is address then..
→ find whose address we are talking about.
→ Find size of that object

ex: int a[4] = {10,20,30,40}
a+1
find type of a ⇒ address
whose address we are talking ⇒ &a[0]
size of a[0] = 4 byte
a+1 = &a[0]+4
= 1004

a: 1000 | 1004 | 1008 | 1012

&a+1 ⇒ whole array
size of a = 4*4 = 16 byte
&a+1 = 1000+16
= 1016 byte

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

addition is commutative
∴ a+b = b+a

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

a + i = &a[i]

*(a+1) = value at (Memory 1004)
= *&(a[1])
= a[1]

*(a+i) = a[i]

1) int a[]; invalid
2) int a[]={10,20,30}; valid

if declaration is there without initialization it is mandatory to provide size of each dimension of array.

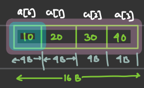
3) In case we are initializing the array, there is flexibility that u can omit the size of 1'st dimension. No other dimension has such flexibility.

int a[]; ✗
int a[2]; ✓
int a[]={10,20}; ✓
int [2]= {10, 20}; ✓
int [2] = {10}; ✓
int [2]={10,20,30,40}; ✓

int a[][]; ✗
int a[2][]; ✗
int a[][3]; ✗
int a[2][3]; ✓
} declaration

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

```
void main(){
    int a[4]={10,20,30,40};
    printf("%u",a);      &a[0]=1000
    printf("%u",&a);     &a = 1000 => whole arr.
    printf("%u",a+1);    &a[0]+1 => &a[1] => 1004
    printf("%u",&a+1);   whole arr. => 1000+16=1016
    printf("%u",*(a+1)); *(&a[1]) => 20
```



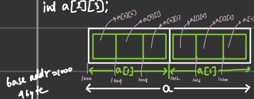
$(a[0]+2) = \& a[0][2]$

$*(a[0]+2) = a[0][2]$

$*(a[i]+j) = a[i][j]$
$\equiv *(*(a+i)+j)$

2D array:

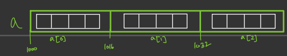
int a[2][3];



a ≡ &a[0]
a[0] ≡ & a[0][0]
&a ≡ Address of whole array

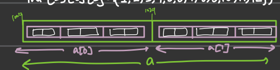
```
printf("%u",a+1);     &a[0]+1 = 1012
printf("%u",&a+1);    1024
printf("%u",a[0]+1);  &a[0][0]+1 = 1004
```

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




```
printf("%u",a);        &a[0] => 1000
printf("%u",&a);       1000
printf("%u",a[0]);     &a[0][0] => 1000
printf("%u",&a[0][0]); 1000
printf("%u",*a);       *(&a[0]) = a[0] = &a[0][0] = 1000
printf("%u",**a);      *(&a[0][0]) = a[0][0] = 1
printf("%u",a+1));     &a[0]+1 = 1000+16=1016
printf("%u",*a+1);     *a+1 = *&a[0]+1 = &a[0][0]+1 = 1004
printf("%u",a[0]+1);   &a[0][0]+1 = 1004
```

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




```
printf("%u"a);            &a[0] => 1000
printf("%u",a[0]);        &a[0][0] = 1000
printf("%u",a[0][0]);     &a[0][0][0] = 1000
printf("%u",a[0][0][0]);  => value 1
printf("%u",&a);          1000
printf("%u",a+1);         &a[0]+1 = 1000+24 = 1024
printf("%u",a[0]+1);      &a[0][0] +1 = 1008
printf("%u",a[0][0]+1);   &a[0][0][0]+1 = 1004
printf("%u",&a+1);        1000 + 2*3*2*4 = 1048
printf("%u",a[0][0][0]+1); value+1 = 2
```

*a => *(& a[0]) = &a[0][0] = 1000
**a *(&a[0][0]) = &a[0][0][0] = 1000
***a *(&a[0][0][0]) = a[0][0][0] = 1
*a+1 &a[0][0]+1 = 1000+2*4 = 1008
**a+1 &a[0][0][0]+1 = 1004

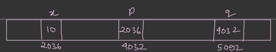
Pointer:

* special variable that are used to store address of other variables.

int a; ← a is a integer variable

int *p; ← p is a pointer to integer
p=&x;   p can hold address of some integer variable

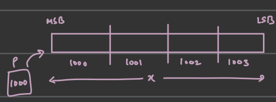
int **q; ← q is a pointer to pointer to integer
q=&p;




```
int x =10;
int *P;
P = &X;
int **q;
q =&P;
printf("%u",X); //10
printf("%u",P); //2036
printf("%u",*P); // 10
printf("%u"*q); //2036
printf("%u",**q); //10
```

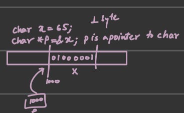
*P
*(Memory location 2036)
*P ⇒ 10

q = &P
*q = *&P
*q = P
*q = &X
**q = *&X
=X



*P ⇒ dereferencing

char x = 65;
char *P=&x; p is a pointer to char



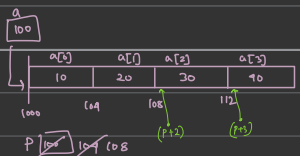
int *p; p ⇒ address of int. variable

add + value ⇒ address
address + add. ⇒ address

Address size is fixed in machine;

Addition of Two pointer is illogical & invalid:

```
int a[4]={10,20,30,40};
int *p;
p = &a[0]; valid
p++;  p=p+1;  add 1 value to 1*4
p+j;
```



Substraction of Two pointer is valid.

```
int a[4]={10,20,30,40};
int *p = a;
p+2; //108
p+3; //112
p = p+2;   108 108
```

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

printf("%d", p[0]);//10
printf("%d", p[1]);// 20
printf("%d", p[2]);// 30
printf("%d", p[3]);// 40
```

P 1000

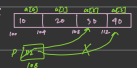
$*(x+i) \equiv x[i]$

$*(p+0) = p[0]$
$*(p+1) = p[1]$
$*(p+2) = p[2]$
$*(p+3) = p[3]$

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

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

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

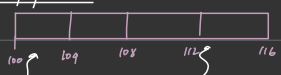


① pointer + pointer

② ++ pointer_variable;
-- pointer_variable;
pointer_variable++;
pointer_variable--;
pointer_variable + 3; moving 3 position forward
pointer_variable - 3; moving 3 position backward

Substraction of pointer:

```
int *p,*q;
p = &a[5];
q = &a[0];
```



q-p ⇒ undefined

p-q ⇒ $\frac{\text{Actual difference}}{\text{size of (int)}} = \frac{112-100}{4} = 3$

diff of Add ⇒ $\frac{\text{Actual Diff}}{\text{Size of data}}$

⑤ ++, * same priority R to L

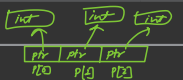
start reading from identifier

Priority

1. ( ) function
2. [ ] array
3. * → pointer
4. Identifier → var-name, fun-name
5. Data type → int, char, float

① L & R → Complex declaration
② R to L
③

`int *(P[9]);` P is an array of 3 pointer to integer

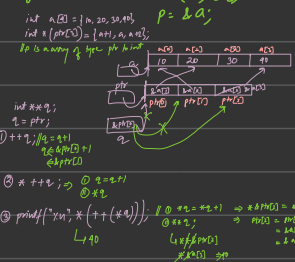


`int (*P)[9];` P is a pointer to array of 9 integer

`int (*P)(int);` P is a pointer to function which takes int as argument, returns int

```
int fun (int);
int (*f)(int);
p = &fun;
```

```
int a[3];
int (*p)[3];
p = &a;
```



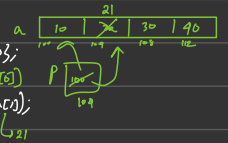
Call by reference :

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

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

priority same, right to left

```
void fun (int *);
int main()
{
  int a[3] = {10,20,30,40};
  fun(a); // fun(&a[0])
  printf("%d %d", a[0], a[1]);
  return 0;
}
void fun(int *P)
{
  ++P;      P = P+1
  ++(*P);   *P = *P + 1
}
```



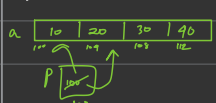
↳ 10 ↳ 21

Result of Increment/Decrement operator is treated as constant

`* ++(++P)` ⇒ not possible

`++P` → Constant → Invalid

```
void fun (int *);
int main()
{
  int a[3] = {10,20,30,40};
  fun(a);
  printf("%d %d", a[0], a[1]);
  return 0;
}
void fun(int *P)
{
  *P++;
  I) *P;
  II) P = P+1;
}
```



↳ 10 ↳ 20

```
void fun (int *);
int main()
{
  int a[3] = {10,20,30,40};
  fun(a);
  printf("%d %d", a[0], a[2]);
  return 0;
}
void fun(int *P)
{
  ++(*P++);
  I) ++(*P)
  ↳ *P = *P + 1
  II) P = P+1
}
```

P = a[0]
*P = a[0]
*P = *P+1 = a[0]+1
P = P+1 = &a[0]+1 = &a[1]

↳ 30

When an array is passed
⇒ It is call by reference
* actually call by address

```
int fun (int *P){}
int fun (int a[]){}
int fun (int a[9]){}
```

all supported

but internally act as pointer

//sum of array element

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

*(P+0) ⇒ P[0]
*(P+1) ⇒ P[1]
*(P+2) ⇒ P[2]
*(P+3) ⇒ P[3]

```
void sum(int *P, int n)
{ int S=0;
  for(i=0; i<n; i++)
    S=S+P[i];
  printf("%d", S);
}
```


```
void main(){
  int a[3][3]={10,20,30,40,50,60};
  fun(a[0]);  → &a[0][0]
  printf("%d %d %d", a[0][0], a[0][1], a[0][2]);
}
```

10 20 200

```
void fun(int *P)
{
  ++P;  P=P+1;
  *(P++);  *P; P=P+1;
  *(++P);  P=P+1; *P
  P--;  P--;
  *P=100;  *P=100
}
```


```
void main(){
  int a[3][3]={10,20,30,40,50,60};
  fun(a);  → &a[0]  ⇒ address of an array of 3 integer
  printf("%d %d %d", a[0][0], a[0][1], a[0][2]);
}
```

```
void fun(int (*P)[3])
{ ++P;                    Valid
  printf("%d", *(*P+1));
}
```

60

*P+1 ⇒ *(a[0]+1)
⇒ a[0][1]
&a[0][0]+1
= &a[0][1]

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

int *P; → 4B
char *P; → 1B
float *P; → 8B


```
int a=300;   little endian
char *P;
P=(char *)&a;   //Compiler can shown
printf("%d", *P);  → 44
```

type casting

```
int a=160;
char *P;
P=(char *)&a;
printf("%d", *P);
```

160 = 128 + 32

1's → 2's
+1

-96

char
int value

= -(2^6+2^4+2^3+2^2+2^1+2^0)-1
= -95-1
= -96

1) int (*P)[4] ⇒ p is a pointer to an array of size 4 type integer

2) int *(*P)[5] ⇒ p is a pointer to an array of size 5 of type pointer to integer

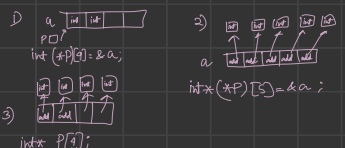
3) int *P[4] ⇒ p is an array of 4 type pointer to integer

4) int (*P)( ) ⇒ p is a pointer to function which takes no argument and returns integer

5) int (*P)(int, float) ⇒ p is a pointer to function which takes (int, float) 2 arg as argument and returns integer value

6) int (*P)(char *) ⇒ p is a pointer to function which takes (pointer to char) 1 arg as argument and returns integer value

7) int (*f)(int *) ⇒ f is a pointer to function which takes (pointer to int) 1 arg as argument and returns integer value



//Question on function pointer

```
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[3] = P[3] - 1; i.e. P[3] = &a[3] - 4 = &a[2]

y = --P[0] - P[1] = &a[3] - &a[2] = 4/4 = 1

*P[0] = *(&a[3]) = a[3] = 40

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

*P[*P[1]-8]

= *P[*(&b-8)]

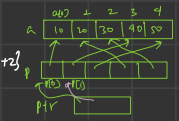
= *P[-3]

= *(&c)

= c

= 15

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



ptr = &P[1]

ptr - P = &P[1] - &P[0] = 1

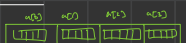
**ptr = *(*P[1]) = *(*(a+3)) = *(a[3]) = a[3] = 40

```
void f(int *)
int main(){
  int a[2][3] = {1,2,3,4,5,6};
  f(a[1]);
  printf("%d %d", a[1][1], a[1][2]);
}
void f(int *P){
  P--;
  *P = *P * *P;
  P--;
  *P = *P * *P;
}
```

P = &a[1][0];

P--; *P = *P * *P; = 3 * 3

```
int a[4][5] = {{1,2,3,4,5},
               {6,7,8,9,10},
               {11,12,13,14,15},
               {16,17,18,19,20}};
printf("%d", *(*(a+**a+2)+3));
```



a = &a[0]

**a = *(*a[0]) = *(a[0][0]) = a[0][0] = 1

a+3 = &a[3]

Ans = 19

((a+1+2)+3)

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

= *(*(&a[3]+3))

= *(a[3][3])

= a[3][3]

*(a+3) = *(a[0]+3) = a[0][3]

```
int f(int *a, int n){
  if(n<=0) return 0;
  else if(*a % 2 == 0)
    return *a + f(a+1, n-1);
  else
    return *a - f(a+1, n-1);
}
void main(){
  int a[] = {12, 7, 13, 9, 11, 6};
  printf("%d", f(a, 6));
}
```

f(&a[0], 6)

return 12 + f(&a[1], 5)

return 7 - f(a[2], 4)

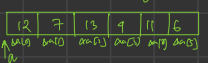
return 13 - f(a[3], 3)

return 9 + f(a[4], 2)

return 11 - f(a[5], 1)

return 6 + f(a[6], 0)

0




```
int f(int *P, int n)
{ if (n<=1) return 0;
  else return max(f(P+1, n-1), P[0]-P[1]);
}
void main()
{ int a[] = {3,5,2,6,4};
  printf("%d", f(a,5)); //3
}
```

f(&a[0], 5)

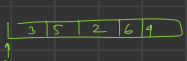
max(f(&a[1], 4), -2)

max(f(&a[2], 3), 3)

max(f(&a[3], 2), -4)

max(f(&a[4], 1), 2)

0



Function Pointer (pointer to function):

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

prog.c

```
1000  fun()
      {
      }
200   main()
      {
      }
```

prog.o

int a=10; float b=10.3f;

int *p=&a; float *q=&b;

int Add(int, int);

int (*p)(int,int); Name of function. Contain address of the function

p = Add;

Add(10,20); ≡ (*p)(10,20);

③

```
int Add(int a,int b){
  return a+b;
}
int Sub(int a,int b){
  return a-b;
}
int Mul(int a,int b){
  return a*b;
}
```

```
void main(){
  int (*p)(int,int);
  p=Add
  printf("%d", (*p)(10,20)); // 30
  p=Sub;
  printf("%d", (*p)(10,20)); // -10
  p=mul;
  printf("%d", (*p)(10,20)); // 200
}
```

2's Complement Representation:

+ve ⇒ as it is

-ve ⇒ 2's complement form

-18 ⇒ 18 = 010010

1's 101101

+1

2's Comp: 101110

direct Method: Consider 0's

Ans = -16-1 (extra) -1

= -18

Void pointer :

→ Can store address
→ Can not dereferenece directly

```
void *P ;
int a=10
float b=17.38;
P=&a;
printf("%d",*P);
printf("%d",*(int*)P);
```

→ first type cast then use
→ Do not perform any arithmatic operation on void pointer

Compiler will not shout

Compiler will give error

*P ≡ *(int*)P

```
int *P;        P=P+2
```
2 ← sizeof(int)

```
char *P;       P=P+3
```
3 ← sizeof(char)

```
void *P;       P=P+2;   //error (logically) size not told
```

Wild pointer :

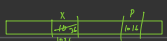
जंगली

uninitialized pointer

```
void main(){
  int a;
  printf("%d",a);//garbage
}
```

```
void main(){
  int *P;
  int x=12;
  *P=36;
}
```

wild → danger



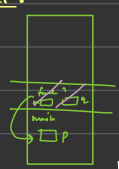
chance very less

Dangling Pointer :

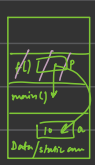
address

```
int * fun(){
  int a=10;
  int *q=&a;
  return q;
}
```

```
int main(){
  int *P;
  P=fun();
  *P=y
}
```




```
int * f(){
  static int a=10;
  int *p=&a;
  return p;
}
```

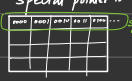



```
void main()
{ int *P;
  P=f();
  printf("%d",*P);//10
}
```

works fine as a is a static variable life time thruout program

Null Pointer :

special pointer to distinguish b/w valid & invalid pointer.



if(ptr) ≡ (value != 0)

P ⇒ NULL

Dynamic Memory Allocation: → memory allocation at runtime

A) malloc( )

B) calloc( )

C) realloc( )

D) free( )

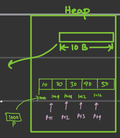
malloc:

syntax: `void * malloc(unsigned int)`

1. search for free space
2. return starting address if available

return
- NULL: memory block not avaliable
- Valid Address: memory avaible but it returns starting address of block

malloc(10);



size of data is m/c dependent, so we should use sizeof(). for allocating space.

```
int * p = malloc(sizeof(int) * 5);
// for input
if(P != NULL)
{
  for(int i=0; i<5; i++)
    scanf("%d", p+i);
  // for output
  for(int i=0; i<5; i++)
    printf("%d", *(p+i));
}   or printf("%d", p[i]);
```

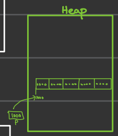
p => &P[0] ≡ &(p+0)

Calloc:

syntax: `void * Calloc(no.of_block, size_of_each_block);`

costly → more time required

1. find free space
2. if available, set all bits to zero(0) return starting address
3. else return NULL



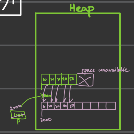
realloc:

syntax: `void * realloc(void * old_addr, unsigned int);`

again

1) find if more space available in same location contiguously if available return same addr
2) if space unavailable contiguously then it found new space and copy existing data to the new address and return new address.

```
int *P = malloc(5 * sizeof(int));
p = realloc(p, new_size);
```

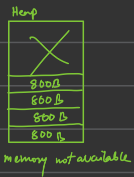


As programmer do dma programmer must do deallocation, otherwise, memory leakage problem occur.

Memory leakage:

```
void fun();
{
  int * p = malloc(200 * sizeof(int));
  =
  free(P);
  return;
}
void main()
{
  fun();
  fun();
  fun();
  fun();
  fun();
}
```

Solution: free memory after use



Memory Leak

free():

syntax: `free(void * ptr)`

deallocate memory from heap.

String :

sequence of charecters terminated by NULL character ('\0')

ascii code → 0

"Rudra"

R	u	d	r	a	\0		

char name[10] = "Pankaj";

P	a	n	k	a	j	\0	

name → name[0]; name[1] ... name[5]

int a[9] = {10,20,30,40}

a → a[0] a[1] a[2] a[3] a[4]

1	2	3	4

// In case of printing a string with printf only starting address of string is required.
It prints all charecter untill null. seeing null it stops.

printf("%s", name); // pankaj

int a[9]={10}; valid
int a = {10}; valid
int a = 10; valid

char name[10] = {"Pankaj"}; valid
char name[10] = "pankaj";

printf("%c", name[0]);

name[3] = u;

P	a	n	u	j	\0

name → name[0]; name[3] stop printing

printf("%s", name); // punkaj

& name[0]

name+2 ⇒ &name[0]+2
⇒ &name[2]

printf("%s", name+2); // nkaj

We can change content of array

This is stored in read only area
So can not change content
// compiler ignore

char *P = "Neeraj";

P → Neeraj (read only)

It is stored in data area of memory string constant pool.

P[2] = 's'; ignore

char name[10] = "Pankaj";
name = "Neeraj";
lvalue: invalid
name++; // invalid

char *name = "pankaj";
name = "Neeraj";
valid
as name is a variable
name → Pankaj / Neeraj
name++; valid

printf("%s", "Hello"); // Hello
printf("Hello"); // Hello
printf("Hello"+1); // ello

H	e	l	l	o	\0

"Hello" ⇒ addr of 'H' given
"Hello"+1 ⇒ addr of 'e'

char name[] = "Pankaj";

P	a	n	k	a	j	\0

char name[6] = "Pankaj"; \0 not stored

P	a	n	k	a	j

char name[] = {'P','a','n','k','a','j'};
char name[] = {'P','a','n','k','a','j','\0'};

P	a	n	k	a	j	\0

by ChatGPT
may error possible

In C programming, const char* p and char* const p differ in what they prevent from being modified:

1. const char* p = "Hello";
- Here, p is a pointer to a const char.
Meaning : The character data that p points to cannot be modified through p, but p itself (the pointer) can be changed to point to another string.
- Example:

```
const char* p = "Hello";
p = "World";  // This is allowed, you can change what p points to
*p = 'h';     // This is not allowed, you cannot modify the characters via p
```

2. char* const p = "Hello";
- Here, p is a constant pointer to a char.
- Meaning : The pointer p cannot be changed to point to another memory location, but the data that p points to can be modified.
- Example:

```
char* const p = "Hello";
p = "World";  // This is not allowed, you cannot change where p points
*p = 'h';     // This is allowed, you can modify the characters via p
```

Summary:
- const char* p: You cannot modify the characters, but you can change the pointer to point to another string.
- char* const p: You cannot change the pointer, but you can modify the characters.

① `printf("%c", *("Hello"+1));` // e

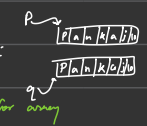
② `printf("%c", "Hello"[1]);` // e

"Hello" ⇒ addr of H
"Hello"+1 ⇒ addr of e
*("Hello"+1) ⇒ 'e'

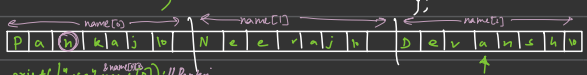
③
```
char *p = "Hello";
char *q = "Hello";
```
⇒ Read only Memory

only single copy created

```
char P[] = "Pankaj";
char Q[] = "Pankaj";
```
separate copy for array



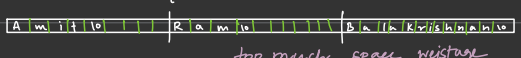
④
```
char arr1[] = "Pankaj";
char arr2[] = "Neeraj";
char arr3[] = "Devansh";
```
⇒
```
char name[3][8] = { "Pankaj",
                    "Neeraj",
                    "Devansh" };
```




```
printf("%s", &name[0]);  // Pankaj
printf("%s", name[1]);   // Neeraj
printf("%s", name[2]+3); // ansh
printf("%c", *(name[0]+2)); // 'n'
```

name[0] ⇒ &name[0][0]
name[0]+2 ⇒ &name[0][2]
*name[0]+2 ⇒ *&name[0][2]
⇒ n

⑤ `char name[3][8] = {"Amit", "Ram", "Balakrishnan"};`

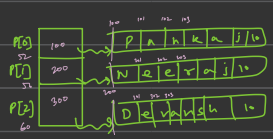


too much space wastage

stored in Read only area.

undefined behavior

sol"
```
char *p = "Pankaj";
char *q = "Neeraj";
char *r = "Devansh";
```
⇒
```
char* P[3] = { "Pankaj",
               "Neeraj",
               "Devansh" };
```



p = &P[0]
*p = *&P[0] = P[0] ⇒ addr of 'P'
printf("%s", *p); // pankaj

p+1 = &P[1]
*(p+1) = *&P[1] = addr of 'N'
printf("%s", *(p+1)); // Neeraj

P[1] ⇒ Add. of 'N'
P[1]+2 ⇒ Add. of 'e'
printf("%s", P[1]+2) ⇒ eraj
printf("%s", P[2]+1); ⇒ evansh

P[1][2] ⇒ *(P[1]+2)

Some Special Fun" ☺

Const is used so that we can't change the string it is a read-only location

```
int mystrlen(const char *str)
{
    int count = 0;
    while(*str != '\0');
    {
        count++;
        str++;
    }
    return count;
}
```

① strlen();

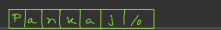
↳ it counts char begining of the string till first null character

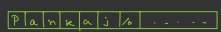
'\0' ⇒ not counted

```
int i;
char *P = "Pankaj";
i = strlen(P);
printf("%d", i);//6
```

② strcpy(char * destination , const char * source)

It must be a buffer (array)



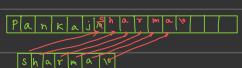



```
Char arr[10];
strcpy(arr, "Pankaj");
printf("%s", arr);
```

```
Char arr[20] = "Pankaj";
strcpy(arr, "Ram");
printf("%s", arr);//Ram
```

③ strcat(buffer str1, str2)

```
Char arr[20] = "Pankaj";
strcat(arr, "Sharma");
```



④ strcmp(str1, str2) ⇒ str1, str2 ⇒ exactly same ⇒ return 0

If not same then returns diff b/w first unmatched character

≅ strcmp("Raj", "Raj") ⇒ 0

a) strcmp("Neeraj", "Neetu");

'r' − 't'

x − (x+2)

= −ve

c) strcmp("Ram", "Rai");

'm' − 'i'

⇒ +ve

⑤

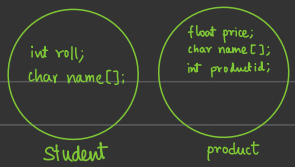
```
char arr[20];
printf("Enter Name");
scanf("%s", arr);
printf("%s", arr);//Pankaj
```

Pankaj Sharma

scanf scans input until it encounters space or new line character

Use gets()
puts()

Structure : user defined data type



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

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

⇒ it is template/blueprint to give information to Compiler.

⇒ no memory is allocated while structure declaration

it is also good practice to make structure declaration as global.

```
void main(){
    int;
    struct student;
    int x;
}
```

⇒ memory not allocated

⇒ memory allocated

```
void main()
{ struct student s1;
}
```

s1 is a group with 2 member

roll name

```
Struct Student{
    int Roll = 10;
    char name[20] = "Pankaj";
};
```

✗ Invalid

Invalid: As no memory is allocated value can not be assigned

In structure padding may be there;

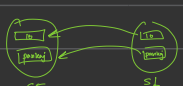
normally element are stored in structure contiguously.

roll	name

(.) dot is used as membership operator

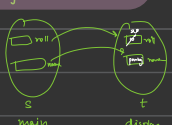
```
Struct Student{
    int Roll;
    char name[20];
};

Void main(){
    Struct student S1,S2;
    S1.Roll = 10;
    S2.Roll = 20;
    S1.name = "Pankaj";   ✗ Invalid
    // array name cannot be Lvalue
    Solⁿ: strcpy(S1.name,"Pankaj");
          strcpy(S2.name,"Neeraj");
}
```

```
Struct Student{
    int Roll;
    char name[20];
};

void main(){
    struct student s1 = {10,"Pankaj"};   ✓
    struct student s2 = {20};   ✓ like array
    struct student s3 = {,"Pankaj"};   s3.roll = "Pankaj"; not allowed
    >> to initialize during declaration we have to give value in order.
    struct student s4 = {"Pankaj",10};   invalid
    struct student s5 = s1;   ✓✓ allowed
}
```




```
void display(struct stu t){   // in array address is shared;
    printf("%d\n", t.roll);   // but in structure it is like call by value.
    printf("%s\n", t.name);
}

void main(){
    struct stu s = {10,"Pankaj"};
    display(s);
}
```

Changes in t will not affect s



Ways to declare structure :

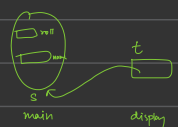
①
```
Struct stu{
    int roll;
    char name[20];
};
void main(){
    Struct stu s1,s2;
    ==
    ==
}
Void f(){
    struct stu s3,s4;
}
```

②
```
Struct{
    int roll;
    char name[20];
} s1,s2,s3;
void main()
{
    // s1 s2 s3 can be used
    // but no more variable can be created as no name provided
}
```

③
```
typedef struct stu{
    int roll;
    char name[20];
} Pankaj;
void main()
{
    Pankaj s1,s2;
    ==
    ==
}
```

-> operator is used dereference data from structure.

```
void display(struct stu *t){
  printf("%d\n",(*t).roll);
  printf("%s\n",(*t).name);
}
void main(){
  struct stu s={10,"Pankaj"};
  display(&s);
}
```



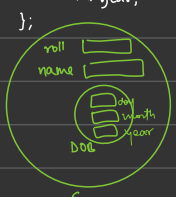

```
void display(struct stu *t){
  printf("%d\n",t->roll);
  printf("%s\n",t->name);
}
void main(){
  struct stu s={10,"Pankaj"};
  display(&s);
}
```

p → member
=
(*p).member
where p is pointer to structure

nested structure :

name ✓ 02/03/2022
Roll ✓
DOB ← Day, Month, Year

```
struct date_of_birth{
    int day;
    int month;
    int year;
};
```




```
struct date_of_birth{
    int day;
    int month;
    int year;
};
struct student{
    int Roll;
    char name[20];
    struct date_of_birth DOB;
}
void main(){
    struct student s;
    s.Roll=10;
    strcpy(s.name,"Pankaj");
    s.DOB.day=2;
    s.DOB.month=3;
    s.DOB.year=2022;
}
```

in this type of declaration we can create variable of type {struct date_of_birth}

```
struct student{
    int Roll;
    char name[20];
    struct date_of_birth{
        int day;
        int month;
        int year;
    }DOB;
}
void main(){
    struct student s;
    s.Roll=10;
    strcpy(s.name,"Pankaj");
    s.DOB.day=2;
    s.DOB.month=3;
    s.DOB.year=2022;
}
```

in this type of declaration we can not create variable of type {struct date_of_birth}

```
struct student{
    int Roll;
    char name[20];
    struct date_of_birth{
        int day;
        int month;
        int year;
    }DOB;
}
```

in this type of declaration as we will not create any variable of type date_of_birth

```
struct{
    int day;
    int month;
    int year;
}DOB;
```

also can be written