Date : / /
Page :

C++

(*) Binary Number System :->

2 Binary = {0, 1}
8 Octal = {0, 1, 2, 3, 4, 5, 6, 7}
10 Decimal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
16 Hexadecimal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

$25_{(10)} = 11001_{(2)}$

↑ Decimal ↑ Binary.

2	25
2	12 — 1
2	6 — 0
2	3 — 0
2	1 — 1
	0 — 0

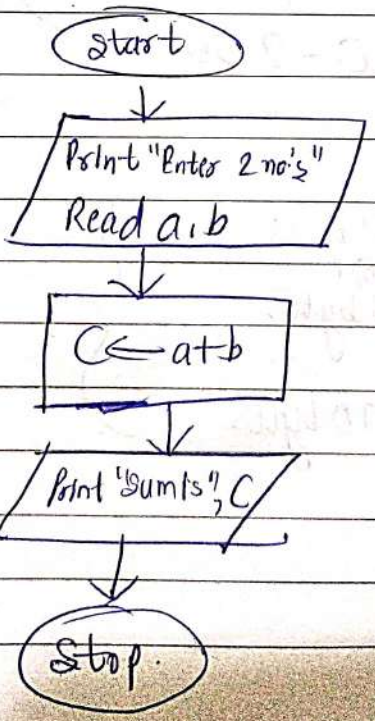
1	1	0	0	1
2^4	2^3	2^2	2^1	2^0

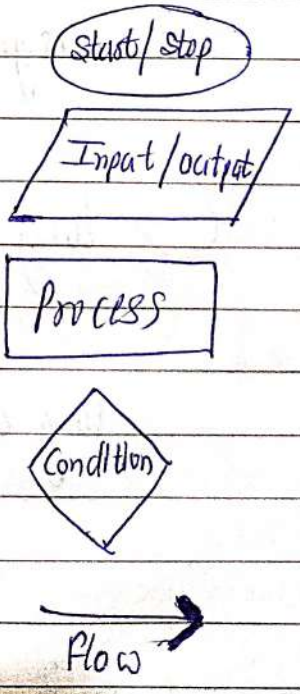
$\Rightarrow 1\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$

$\Rightarrow 16 + 8 + 0 + 0 + 1 = 25_{(10)}$.

(*) Flowchart :->

Example :->





(*) Data Types and Variables :=>

Data Type	Size (byte)	Range.
int.	2 or 4	-32768 to 32767.
float	4	-3.4×10^{-38} to 3.4×10^{38}
double	8	-1.7×10^{-308} to 1.7×10^{308}.
char	1	-128 to 127.
bool	undefined	true / false

Remember ASCII codes!

char group = 'A';
↳ only in single quotes

#=> Modifiers :=>
- ⊙ Unsigned
- ⊙ long

⊙=> Usigned int
=> 0 — 65535

Usigned char
=> 0 - 255

Only These two can be used

⊙=> long int
=> 4 bytes
(or) 8 bytes.

long double
=> 10 bytes

(*) Precedence and Expressions :->

Operator	Assumed Precendence
( )	3
*, /, %	2
+, -	1

$x = a + b * c - d / e$

3 1 4 2

L to R

#include <cmath>
(or) #include <math.h>

(1) Area of $\Delta^{(a)} = \frac{1}{2} bh \Rightarrow$ a = 1/2 ;
a = (b*h)/2 ;
(or) a = 0.5 * (b*h) ;

(2) Perimeter of □ $\Rightarrow P = 2(l+b) \Rightarrow$ P = 2*(l+b) ;

(3) Sum of n terms $S = \frac{n(n+1)}{2} \Rightarrow$ S = n*(n+1)/2 ;

(4) n^{th} term of A.P series $\Rightarrow t = a+(n-1)d \Rightarrow$ t = a+(n-1)*d ;

(5) root of Q.E $\Rightarrow r = \frac{-b \pm \sqrt{b^2-4ac}}{2a} \Rightarrow$ r = (-b + sqrt(b*b-4*a*c))/(2*a)

(6) Speed, $S = \frac{v^2 - u^2}{2a} \Rightarrow$ S = (v*v - u*u)/(2*a)
(or)
S = pow(V,2) - pow(U,2)/(2*a) ;

(*) Compound Assignment Operators :=>

```
int a = 20, b = 5, C = 25;
int Sum = 5;

Sum += a;
Sum += b;
Sum += C;

int product = 2;

prod *= a * b * C;
```

Compound Assignment
+=
-=
*=
/=
%=
&=
|=
<<=
>>=
⋮

(*) Increment and decrement :=>

```
int x = 5, y;
```

```
x++;
++x;
```

```
y = ++x;   x = 6
(2) (1)    y = 6.
```

```
y = x++;   x = 6
(1) (2)    y = 5
```

Pre inc :=>
++x;
Post inc :=>
x++;
Pre dec :=>
--x;
Post dec :=>
x--;

```
int x = 5, y = 20; z;
z = x++ * y;
    5   (2) 20
50
```

x = 6
y = 20
z = 50

```
z = ++x * y;
60   6   * 20
```

x = 6
y = 20
z = 60

(*) Bitwise Operator :=>

bit 1	bit 2	bit 1 & bit 2
0	0	0
1	0	0
0	1	0
1	1	1

&	and
|	OR
^	X-OR
~	not
<<	
>>	

bit 1 | bit 2	bit 1 ^ bit 2
0	0
1	1
1	1
1	0

```
int x=11, y=5, z;
```

x = 00001011
y = 00000101
z = x & y
00000001

z = 1

```
int x=5, y;
```

x = 00000101
00000101 0
y = x<<1 (8 4 2 1)
8 + 2 = 10

x<<i	x>>i
x * 2^i	x / 2^i

(*) Enum and Typedef :->

```
const int CS = 1
const int ECE = 2
  :
  :
```

```
enum dept {CS=1, ece, it, civil}
int main()
{
    dept d;
    d = CS;
    :
    :
```

```
enum day {mon, tue, wed, thu, fri, sat, sun}
          0    1    2    3    4    5    6
   datatype
int main()
{
    day d;
    d = mon;
    d = fri;
  X d = 0;
}
```

if (d == mon)
↳ True

(*) Typedef :->

```
int main()
{
    int m1, m2, m3, r1, r2, r3;
    :
    :
}
```

```
typedef int marks;
typedef int rollno;
int main()
{
    marks m1, m2, m3;
    rollno r1, r2, r3;
}
```

(*) Conditional Statements :=>

#:) Dynamic declaration :=> Jab ek variable bas ek particular scope ke liye data carry kre aur jab uska kam hojata hai woh memory se delete hojata hai.

```
if (int c=a+b; c>10).
{

}
```

(*) Loops :=> 1) while
2) do-while
3) for
4) for each

(1).
```
while (<cond>)
{
  Process;
}.
```

(2).
```
do
{ Process;
} while (<Condition>);
```

(3).
```
for (initialization; condition; updation)
{

}.
```

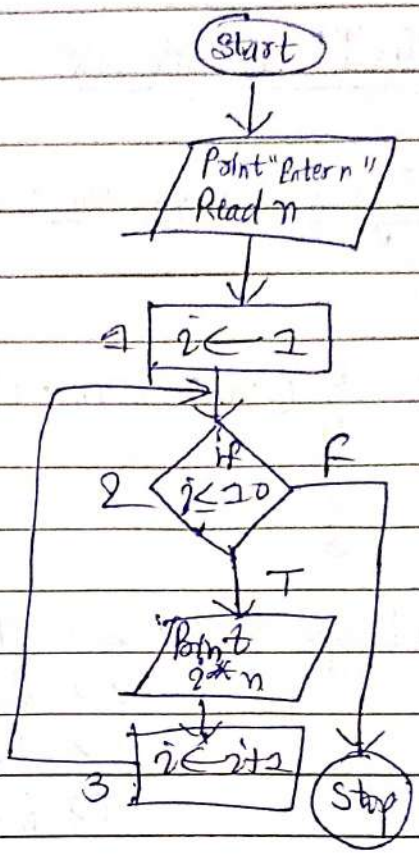
e.g. :=> for (int i=0; i<10; i++)

Date : / /
Page :

(*) Multiplication Table :=>

n = 6, i = 1

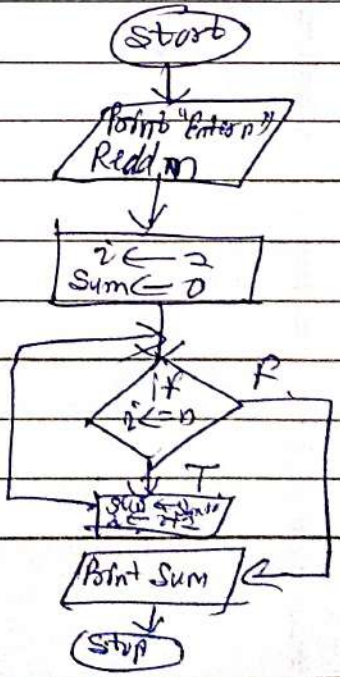
n	*	i		i*n.
6	X	1	=	6
6	X	2	=	12
6	X	3	=	18
6	X	4	=	24
6	X	5	=	30




```
=> int main()
{
    int n, i;
    cout << "Enter n";
    cin >> n;
    for (i=1; i<=10; i++)
    {
        cout << n << "X" << i << " = " << i*n << endl;
    }
    return 0;
}.
```

(*) Sum of N natural no.'s :=>

Write Programs revise Karlenge!

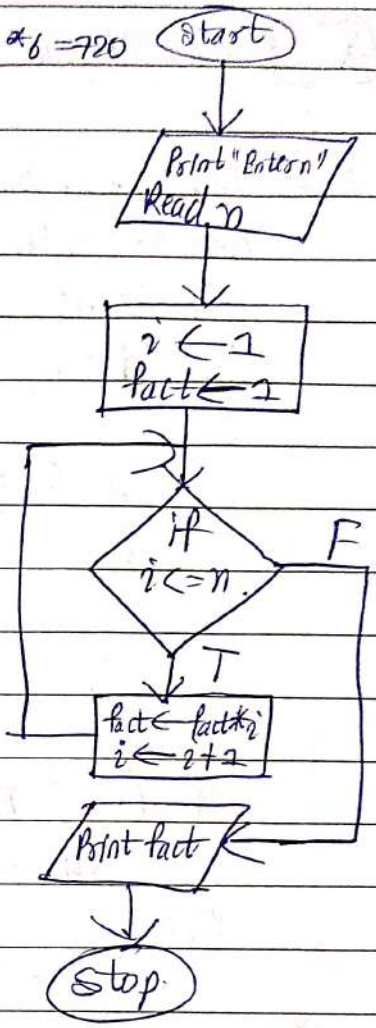



```
cout << "Enter n";
cin >> n;
for (i=1; i<=n; i++)
{
    Sum = Sum + i;
}   Sum += i;
cout << "Sum of N no is" << Sum;
return 0;
}
```

(*) Factorial of n :- $n! = 1*2*3*4*5*6 = 720$

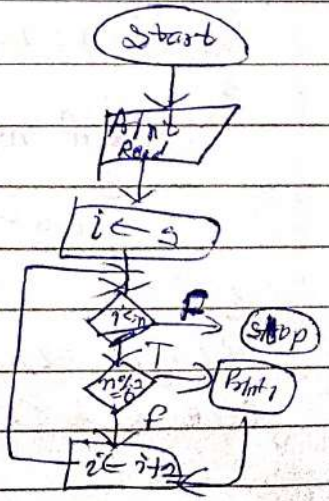
$n=6$, $i=1$ $fact=1$

i	fact = fact * i
1	1*1=1
2	1*2=2
3	2*3=6
4	6*4=24
5	24*5=120
6	120*6=720




```
=> int main()
{
    int n, i, fact = 1;
    cout << "Enter n";
    cin >> n;
    for (i=1; i<=n; i++)
    {
        fact *= i;
    }
    cout << "Factorial of " << n << " is " << fact;
    return 0;
}
```

(*) Factors of no. :-




```
int main()
{
    int n, i;
    cout << "Enter n";
    cin >> n;
    for (i=1; i<=n; i++)
    {
        if (n % i == 0)
            cout << i << endl;
    }
}
```

Date : / /
Page :

(*) Perfect Number :- If some of factors of a number is double the number, then it is perfect number.

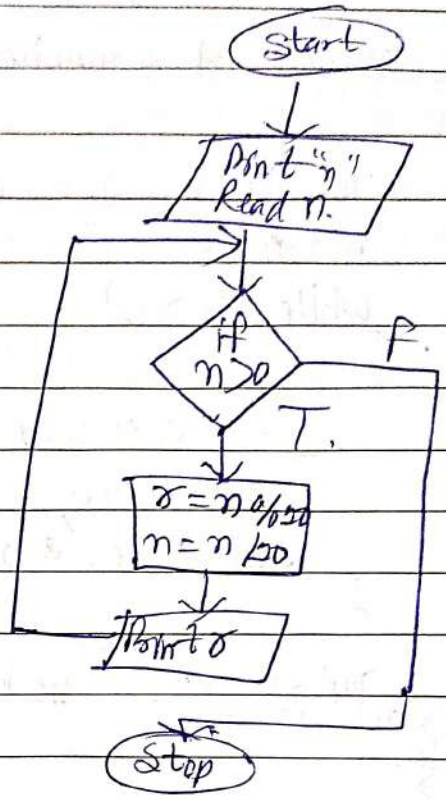
n = 6
1+2+3+6 = 12
2*6 = 12

```
=> int n, i, Sum = 0;
cout << "Enter n";
cin >> n;
for (i = 1; i <= n; i++)
{
    if (n % i == 0)
    {
        Sum = Sum + i;
    }
}
if (2 * n == Sum)
    cout << "Perfect no.";
else
    cout << "not Perfect no";
}
```

(*) Prime no. :- no. divided by 1 & itself.

```
int n, i, count = 0;
for (i = 1; i <= n; i++)
{
    if (n % i == 0)
    {
        count++;
    }
}
If (count == 2) cout << "It is a Prime";
else cout << "Not a prime";
}
```

(*) Display digits of a number :=>



=> int main()
{
int n, r;
cout << "Enter n";
cin >> n;
while (n>0)
{
r = n%10;
n = n/10;
cout << r << endl;
}
return 0;
}

(*) Armstrong Number :=>

n = 153
$1^3 + 5^3 + 3^3$
$1 + 125 + 27 = 153$.

int main()
{
int n, r, sum = 0, m;
m = n;
while (n>0)
{
r = n%10;
n = n/10;
sum = sum + r*r*r;
}
if (sum == m) cout << "Its Armstrong";
else cout << "Not Armstrong";
}

(*) reverse of a number ⇒

⇒ (**)
```
int n, r, rev=0, m;
m=n;
while (n>0)
{
    r=n%10;
    n=n/10;
    rev = rev*10+r;
}
cout<<"Reverse no is " << rev;
return 0;
}
```

To display all digits in words use switch case.
123
one two three.

(*) GCD of 2 nos ⇒

m=30 n=22

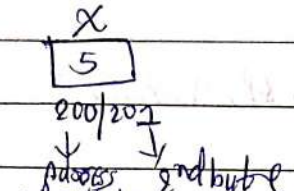
m	n
30	22
30-22=8	22
8	22-8=14
8	14-8=6
8-6=2	6
6-2=4	2
4-2=2	2

```
int main()
{
    int m, n;
    cout<<"Enter 2nos";
    cin>>m>>n;
    while (m!=n)
    {
        if (m>n)
            m=m-n;
        else if (n>m)
            n=n-m;
    }
    cout<<m;
}
```

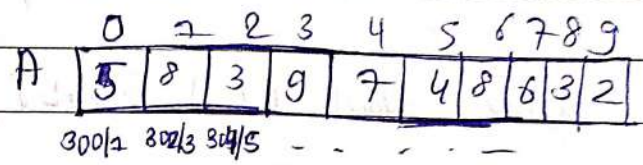
(*) Arrays :- ⇒ Scalar :- ⇒ x = 5.

Vector / List ⇒ A = (5, 8, 3, 9, 7, 4, 8, 6, 3, 2)
indices: 0 1 2 3 4 5 6 7 8 9
A0 A1 A2 A3 A4

```
int x = 5;
```




```
int A[10] = {5, 8, 3, 9, 7, 4, 8, 6, 3, 2};
```

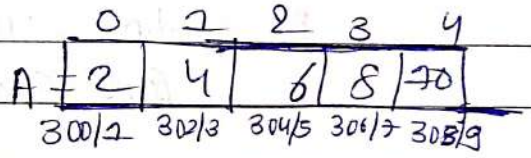


⇒ 20 bytes of memory = 2 bytes × 10 ints
= 20 bytes

```
cout << A[3];
cout << A[8];
cout << A;
```

(#) ⇒

```
int main()
{
    int A[5] = {2, 4, 6, 8, 10};

    for(int i=0; i<5; i++)
    {
        cout << A[i] << endl;
    }
}
```



(#) ⇒

```
int A[5] = {2, 4, 6, 8, 10};
float B[5] = {1.7, 2.4, 3.7, 6.2, 9.5};
char C[5] = {'A', 'B', 'C', 'D', 'E'};
int A[5] = {2, 4};
int B[] = {1, 2, 4, 6};
```

A: 2 4 0 0 0 (indices 0 1 2 3 4)

B: 1 2 4 6 (indices 0 1 2 3)

Date : / /
Page :

⇒ Works only on Collection of values.

(*) For each loop ⇒ Used in Arrays & collection type in DS.

```
int A[] = {2, 4, 6, 8, 10, 12};
for (int x : A)
    cout << x << endl;
```

(#) float A[] = {2.5f, 5.6f, 9, 8, 7};

```
for (auto or float x : A)
    cout << x << endl;
```

(#) char A[] = {'A', 66, 'C', 68};

```
for (auto x : A)
    cout << x << endl;
```

(#) Program to add all elements of an array ⇒

```
int main()
{
    int A[7] = {4, 8, 6, 9, 5, 2, 7};
    int n = 7, sum = 0;

    for (int i = 0; i < 7; i++)
    {
        sum = sum + A[i];
    }
    cout << "Sum is " << sum;
    return 0;
}
```

	0	1	2	3	4	5	6
A[]	4	8	6	9	5	2	7

n = 7, sum = 0;

i	A[i]	sum = sum + A[i]
0	4	0+4 = 4
1	8	4+8 = 12
2	6	12+6 = 18
3	9	18+9 = 27
4	5	27+5 = 32
5	2	32+2 = 34
6	7	34+7 = 42

#) Program to find Max element in an Array :=>

```
int main()
{
    int A[7]={4,8,6,9,5,2,7};
    int n=7, max;
    max = A[0];

    for(int i=0; i<7; i++)
    {
        if(A[i]>max)
        {
            max= A[i];
        }
    }
    cout<<"Max^m no. is"<<max;
    return 0;
}
```

A	4	8	6	9	5	2	7
	0	1	2	3	4	5	7.

n=7

max = ~~4~~ ~~8~~ 9

i	A[i]	if(A[i]>max) max=A[i]
0	4	4
1	8	8
2	6	8
3	9	9
4	5	9
5	2	9
6	7	9.

(*)) Linear Searching :=>

```
int main()
{   int A[20], n=20;
    cout<<"Enter the numbers";
    for(int i=0; i<n; i++)
    {
        cin>>A[i];
    }
    cout<<"Enter Key";
    cin>>Key;
```

n=20

A	6	12	25	8	15	7	22	20	9	14
	0	1	2	3	4	5	6	7	8	9.

Key = 22 Successful

&

Key = 35 unsuccessful

INT_MIN

Date : / /
Page :

Linear Search

```
for (int i=0 ; i<n ; i++)
{
    if (Key == A[i])
    {
        cout << "found at" << i;
        return 0;
    }
}
cout << "Not Found";
}
```

(*) Binary Search (Cond^n) => elements must be in Sorted order.

```
int main()
{
    int A[10] = {6,8,13,27,20,25,28,30,55};
    int l=0, h=9, mid, Key;
    cout << "Enter Key";
    cin >> Key;
    while (l<=h)
    {
        mid = (l+h)/2;
        if (Key == A[mid])
        {
            cout << "found at" << mid;
            return 0;
        }
        else if (Key < A[mid])
            h = mid-1;
        else l = mid+1;
    }
    cout << "Not found";
}
```

n=10 l=0 & h=9

A	6	8	13	27	20	22	25	28	30	55
	0	1	2	3	4	5	6	7	8	9

l h

time => O(log n)

Key = 25

mid

l h middle

0 9 $mid = \left[\frac{l+h}{2}\right]$

$\frac{0+9}{2} = 4$

5 9 $= \frac{5+9}{2} = 7$

5 6 $= \frac{5+6}{2} = 5$

6 6 $= \frac{6+6}{2} = 6.$

Key = 27.

(l) => low -> Should be on left side.

(h) => high -> Should be on right side.

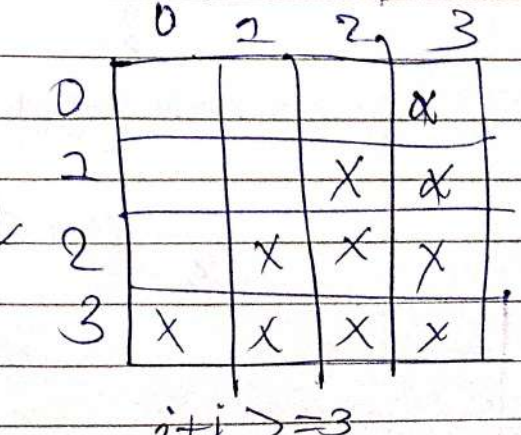
(*) Nested for loop :=>

```
for (      )
{
    for (      )
    {
        cout << i << j << endl;
    }
}
```

=> cout << "(" << i << ", " << j << ")";

#)
```
for (i=0; i<4; i++)
{
    for(j=0; j<4; j++)
    {
        if (i>=j)
            cout << "*";
    }
    cout << endl;
}
```

#)
```
for (      ;      )
{
    for (      ;      )
    {
        if (i+j>4-1)
            cout << "*";
        else cout << " ";
    }
    cout << endl;
}
```



(*) Multi dimensional Arrays :=>

```
int A[3][4];
```

A	0	1	2	3
0				
1			25	
2				

A[1][2] = 15;

=> int A[2][3] = {{2,5,9},{6,9,15}};

A	0	1	2
0	2	5	9
1	6	9	15

(OR) => int A[2][3] = {2,5,9,6,9,15};

(OR) => int A[2][3] = {2,5};

=> Accessing all elements :=>

```
for (int i=0 ; i<2; i++)
{
    for (int j=0 ; j<3; j++)
    {
        cout << A[i][j];
    }
    cout << endl;
}
```

1D Adding two matrices :=>

$$A = \begin{matrix} & 0 & 1 & 2 \\ 0 & 2 & 5 & 9 \\ 1 & 7 & 5 & 6 \end{matrix} \quad 2 \times 3$$

$$B = \begin{matrix} & 0 & 1 & 2 \\ 0 & 6 & 3 & 4 \\ 1 & 9 & 5 & 2 \end{matrix} \quad 2 \times 3$$

$$C = \begin{matrix} & 0 & 1 & 2 \\ 0 & 2+6 & 5+3 & 9+4 \\ 1 & 7+9 & 5+5 & 6+2 \end{matrix} \quad 2 \times 3.$$

Program :=>

```
int main()
{
    int A[2][3] = {{2,5,9},{7,3,6}};
    int B[2][3] = {{6,3,4},{9,5,2}};
    int C[2][3];

    for(int i=0; i<2; i++)
    {
        for(int j=0; j<3; j++)
        {
            C[i][j] = A[i][j] + B[i][j];
        }
    }

    for(int i=0; i<2; i++)
    {
        for(int j=0; j<3; j++)
        {
            cout<<C[i][j];
        }
    }
```

For Printing C matrix.

Date : / /
Page :

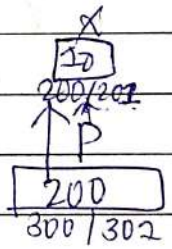
⇒ Using for each loop ⇐

```
int main()
{
    int A[2][3] = {2,4,6,3,5,7};

    for (auto& x : A)
    {
        for (auto& y : x)
        {
            cout << y << " ";
        }
        cout << endl;
    }
    return 0;
}
```

⇒ Pointers ⇐

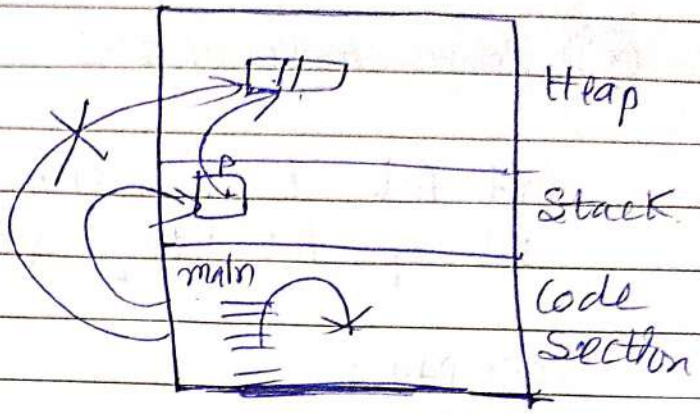
1.) data variable
2.) Address variable

```
int x = 10;
int *p;        // declaration
p = &x;        // initialization
cout << x;     // -> 10
cout << &x;    // -> 200
cout << p;     // -> 200
cout << &p;    // -> 300
cout << *p;    // -> 10   (dereferencing)
```



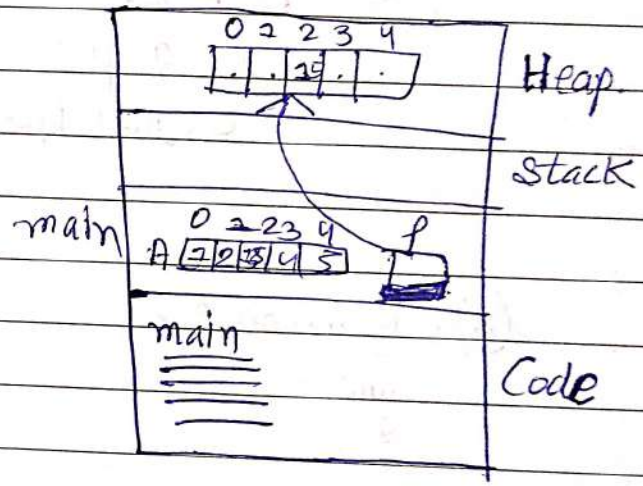
Why Pointers?

```
main()
{
  =
  =
  =
}
```



Heap Memory Allocation :

```
main()
{
  int A[5] = {1,2,3,4,5};

  int *p;
  p = new int[5];
```



⇒ `int *p = new int[5];`

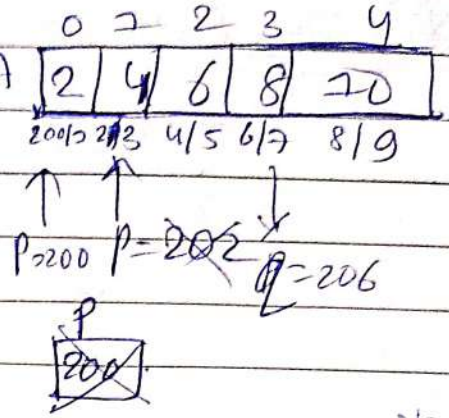
`delete []P;`

`*p = NULL;`
↳ A pointer not pointing anywhere is called null-pointer.

```
A[2] = 25;
P[2] = 25;
```

Pointer Arithmatic :=>

```
int A[5] = {2, 4, 6, 8, 10};
int *p = A; int *q = &A[3]
```



1.) p++;
2.) p--;
3.) p = p+2
4.) p = p-2
5.) d = q-p.
↳ distⁿ b/w two pointers.

1st Pointer is far away

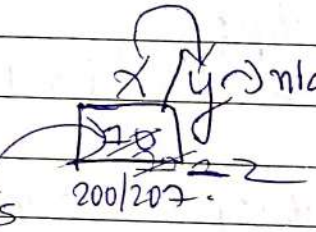
2006-200 = 6/2 = 3

200-206 = -6/2 = 3

↳ 2nd Pointer is far away

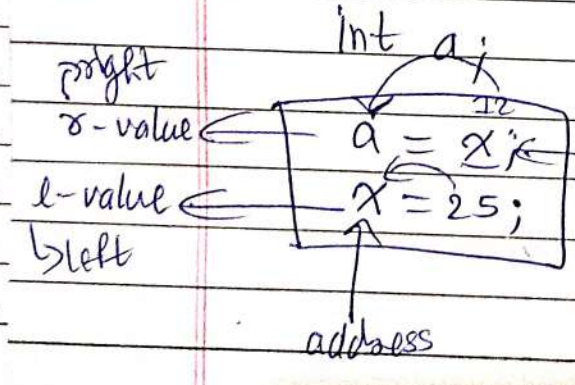
Reference :=>

```
main()
{
    int x = 10;
    int &y = x;
    x++;
    y++;
    cout << x; -> 12
    cout << y; -> 12
```



l-value.

No memory is consumed by reference..



(*) Pointer to a function :-

```
Void display()
{
    cout << "Hello";
}
int main()
{
declare ⟶ Void (*fp)();
initi   ⟶ fp = display;
Call    ⟶ (*fp)();
}
```

(#)
```
int max(int x, int y)
{
    return x>y ? x : y;
}
```

(#)
```
int min(int x, int y)
{
    return x<y ? x : y;
}
```

```
int main()
{
    int (*fp)(int, int);
    fp = max;
    (*fp)(10,5);      ⟶ max is called.
    fp = min;
    (*fp)(10,5);      ⟵ min is called.
}
```

(*) Functions :→

return-Type Function_name (Parameter List).

O/P — almost 1 value

I/P — 0 or more.

eg :→

```
void display ()
{
    cout<<"Hello";
}
void main()
{
    display();
}
```

avoid using cout & cin inside functions.

eg :→

```
int add (int x, int y)
{
    int z;
    z = x+y;
    return z;
}
void main()
{
    int a = 20, b = 25, c;
    c = add (a,b);
    cout<<"Sum is "<<c;
}
```

(*) Function Overloading :⇒ We can assign same name to multiple functions.

```
int add (int x, int y).
{
    return x+y;
}
```

```
int add (int x, int y, int z)
{
    return x+y+z;
}
```

```
float add(float x, float y).
{
    return x+y;
}
```

```
void main()
{
    int a=10, b=5, c, d;
    c= add(a,b);
    d= add(a,b,c);
    int i=2.5f, j=8.5f, K;
    K= add(i,j);
}
```

⇒ int max(int, int)
float max(float, float)
int max(int, int, int)
float max(int, int)

These two are same fxn.
⇒ There is a name conflict only otherwise they both are same.

(*) Function Template :->

```
template <class T>
T max (T x, T y)
{
    if (x>y)
        return x;
    else
        return y;
}
main()
{
    int c = max(20, 5);
    float d = max(20.5f, 6.9f);
}
```

→ T becomes "int"

→ T becomes "float"

```
int max(int x, int y)
{
    if (x>y)
        return x;
    else
        return y;
}
float max (float x, float y)
{
    if (x>y)
        return x;
    else
        return y;
}
```

↳ In the above fxns we can see that name of fxns are same. it means it is overloaded fxn. instead of writing all these, we can write this in a very particular manner. ~~Keep the lo~~ yaa the logic in fxns above is same. So we can use fxn template.

Date : / /
Page :

(*) Default Arguments :- ⇒ Assigning default values to arguments

```
int add(int x, int y, int z=0)
{
    return x+y+z;
}
```

(easy)

```
int fun(int a=0, int b, int c=0, int d=0)
```

explanation

R to L ⇒ assignment of default value is done

```
main()
{
    int c = add(2, 5);
        c = add(2, 5, 8);
        c = add(2, 5, 0);
}
```

```
int add(int x, int y)
{
    return x+y;
}
int add(int x, int y, int z)
{
    return x+y+z;
}
```

(*) Parameters Passing Method :-

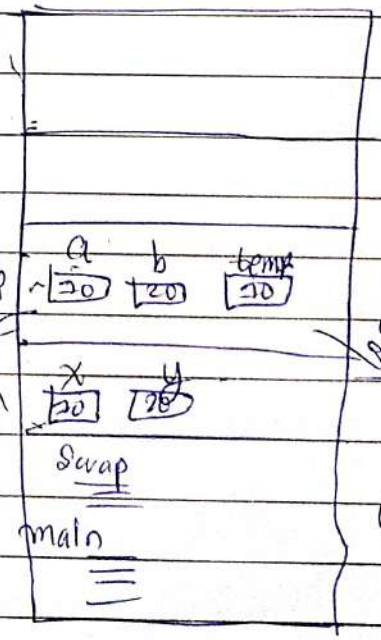
1. Pass by Value. (call)
2. Pass by Address
3. Pass by Reference.

e.g. :- (1)

```
void swap(int a, int b)        // formal parameters
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x=20, y=20;            // Actual Parameters
    swap(x, y);
    cout<< x << " " << y;
}
```

call by value.

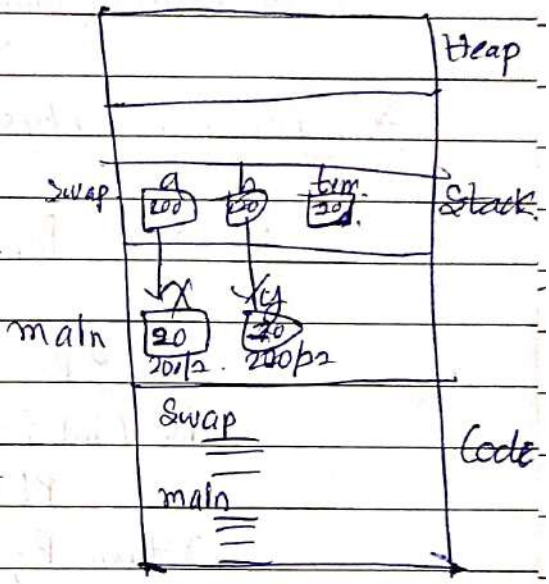
20 20 ⇒ call by value is not used for swapping values !!



⇒ In call by Value mechanism the values of the formal parameters if they are modified they are not reflected in actual parameters.

(2)) Call by Address :⇒

```
Void Swap (int *a, int *b)
{                          ↑ formal.
    temp = *a;
    *a = *b;
    *b = temp;
}
int main ()
{
    int x=20, y=20;
    Swap (&x, &y); ← actual
    cout << x << " " << y;
}
```

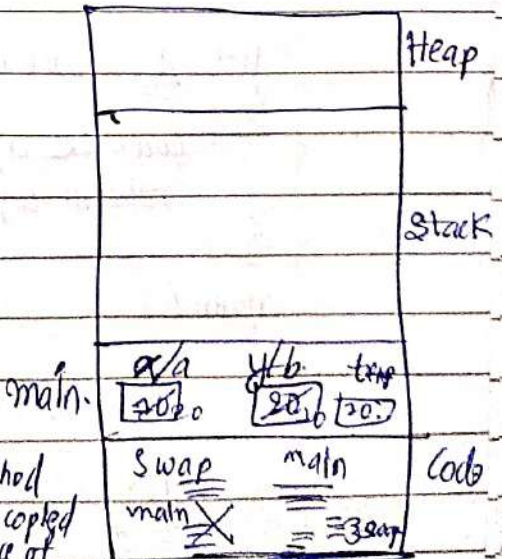


⇒ Pointers concept use hota hai iss mechanism mein.

⇒ nickname to an existing variable.

(3)) Call by "Reference" :⇒

```
Void Swap (int &a, int &b)
{                          ↑ Formal.
    Int temp;
    temp = a;
    a = b;
    b = temp;
}
Int main ()
{  Int x=20, y=20;
   Swap (x, y) ; ← actual.
   cout << x << " " << y;
}
```



⇒ In this method function is copied to the place of calling value. during runtime 😊

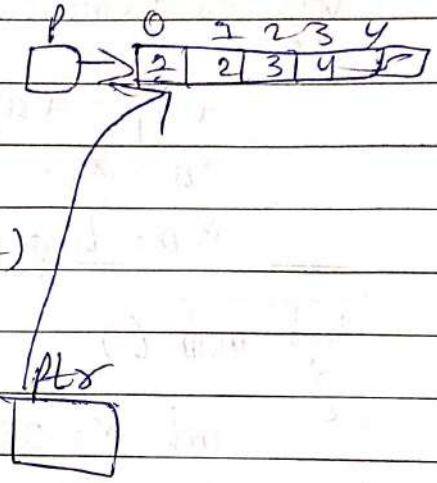
⇒ "a" alias x
"b" alias y.

=> Avoid using complex logic inside fxn while using call by reference. Avoid loops.

=> It is used only for simple fxns.

(*) Return by Address :=>

```
int * fun (int size)
{
    int *p = new int[size];
    for (int i=0; i<size; i++)
        p[i] = i+1;
    return p;
}
main ()
{
    int *ptr = fun (5);
}
```



(*) Return by Reference :=>

```
int & fun(int &a)
{
    cout<<a; => 20
    return a;
}
main ()
{
    int x=20;
    fun(x) = 25;
    cout<<x; => 25
}
```

x / a
20 / 25

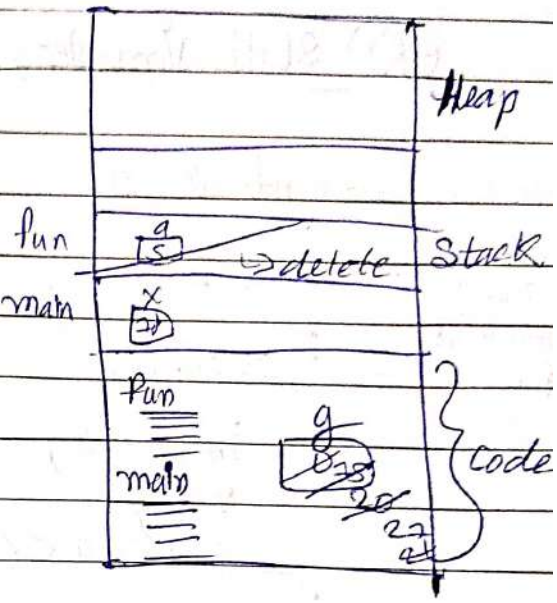
↓
nothing but "x" itself

(*) Local and Global variables :->

global :-> int g=0;

void fun ()
{
local -> int a=5;
g = g+a;
cout<<g; -> 26
}

void main ()
{
local -> int x=10;
g=15;
fun ();
g++;
cout<<g; -> 22
}



(*) Scoping method :->

```
int x=10;
int main ()
{
  int x=20;
  {
    int x=30;
    cout<<x<<endl;
  }
  cout<<x<<endl;
  cout<<::x<<endl;
}
```

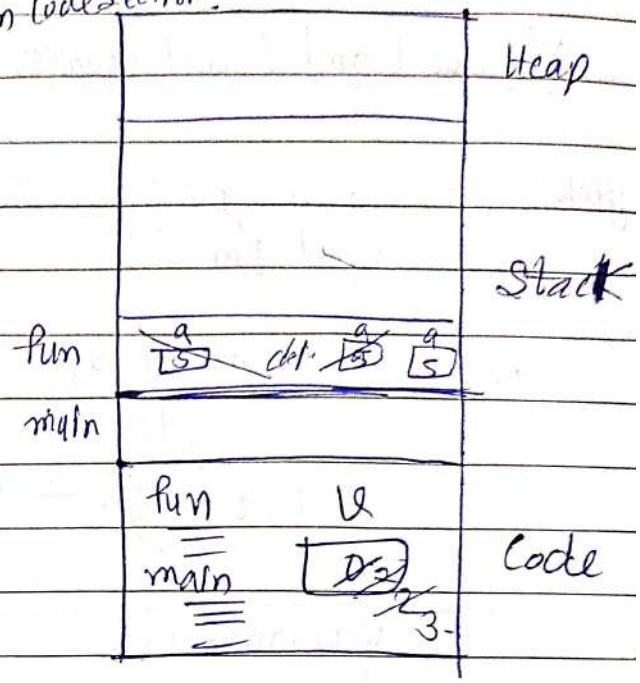
(*) Static Variables :-

→ makes variable global for that particular fxn's scope & retains code section.

```
int v=0;
void fun().
{
    static int a=5;
    v++;
    cout<<a<<" "<<v;
}
main()
{
    fun();
    fun();
    fun();
}
```

I want this int a in fun().

	a	v
fun(); →	5	2
fun(); →	~~5~~	2
fun(); →	5	3



(*) Recursive fxn :-

```
void fun(int n)
{
    if (n>0)
    {
        cout<<n<<endl;
        fun(n-1);
    }
}
int main()
{
    int x=5;
    fun(x);
}
```

Date : / /
Page :

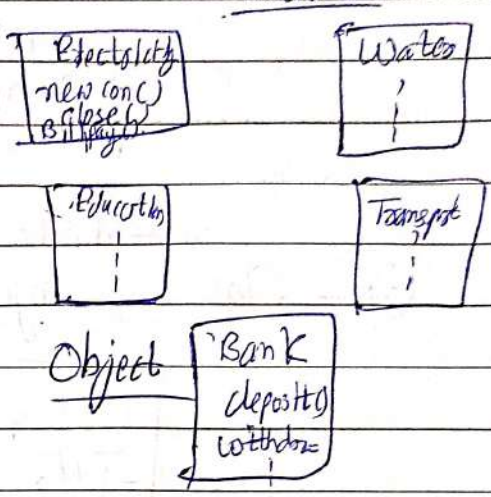
(*) Introduction OOPs :=>

Modular Programming

Bank

openAcc()
deposit()
withdraw()
checkbal()
applyloan()
:
:
:

Object-Oriented Programming



(*) Principles of Object-Orientation :=>

1.>. Abstraction
2.>. Encapsulation.
   • Data hiding.
3.>. Inheritance.
4.>. Polymorphism.

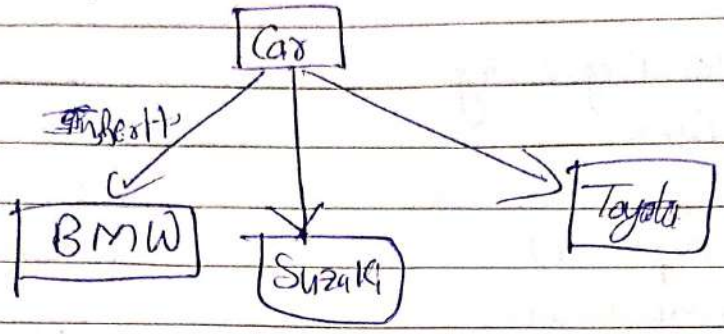
=> Abstraction & Encapsulation are interrelated.

e.g.:=>
```
class My
{
  Private:
  (1) Data.
  Public:
  (2) functions()
};
```

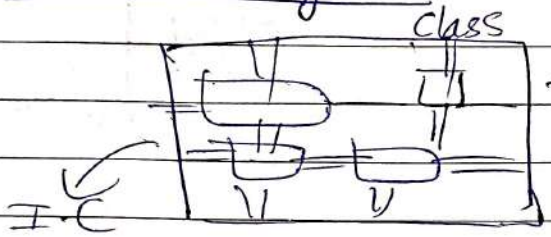
Date : / /
Page :

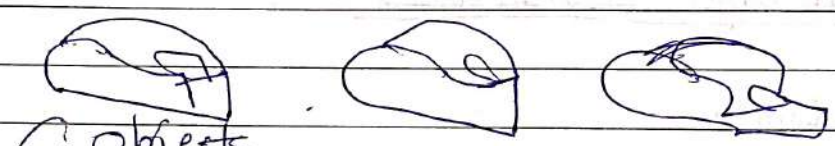
⇒ Inheritance & Polymorphism are also interrelated.

eg ⇒



(*) Class vs Object ⇒ → classification





: objects

Fingerprint Scanner.

Class Human ⇒ you, me } objects

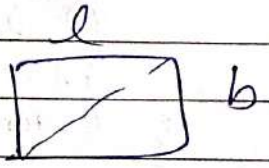
class Car ⇒ BMW, Toyota } objects

eg ⇒ class Rectangle

{

float length;
float breadth;
float area();
float perimeter()
float diagonal()

};



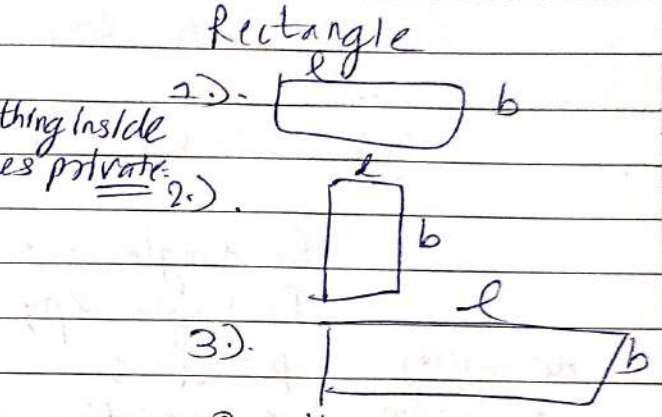
```
main()
{
    Rectangle r1, r2, r3;
}
```
objects.

(*) Writing a class :=>

```
class Rectangle
{
    public:
    int length;
    int breadth;

    int area()
    {
        return length * breadth;
    }
    int perimeter()
    {
        return 2 * (length + breadth);
    }
};
```

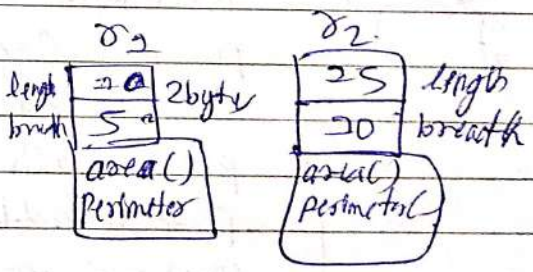
By default everything inside class becomes private.

int length; → 2 bytes
int breadth; → 2
4 bytes




```
void main()
{
    Rectangle r1, r2;
    r1.length = 10;
    r1.breadth = 5;
    cout << r1.area(); → 50
    r2.length = 25; r2.breadth = 10;
    cout << r2.area(); → 250
}
```

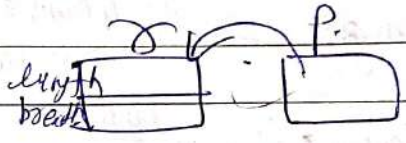
dot operator → used for accessing members.



Date : / /
Page :

(*) Pointer to Object :- (in Heap)

```
Class Rectangle
{
    Pichhe dekho;
};
int main()
{
    Rectangle r;
    Rectangle *p;
    p = &r;
    r.length = 20;
    p -> length = 10;
    p -> breadth = 5;
    cout << p -> area();
}
```

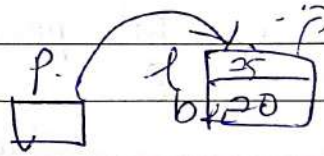


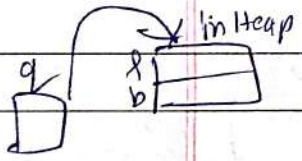
dereferencing operator instead of ->

arrow

For accessing members of an object using pointer

```
void main()
{
    Rectangle *p;
    p = new Rectangle;
    Rectangle *q = new Rectangle();
    p -> length = 15;
    p -> breadth = 20;
    cout << p -> area();
}
```





2 Methods :-

For stack →	Rectangle r;
For Heap →	Rectange *p = new Rectangle();

(*) Data Hiding :- Bas Samjhne Ki Baatein;

Property fxn

Accessor → get XXX

Mutators → set XXX

```
class Rectangle
{
    private :
        int length;
        int breadth;
    public :
        void setLength (int l)
        {
            if (l>=0)
                length = l;
            else
                length = 0;
        }
        void setBreadth (int b)
        {
            if(b>=0)
                breadth = b;
            else
                breadth = 0;
        }
        int getLength()
        {
            return length();
        }
        int getBreadth()
        {
            return breadth();
        }
};
```

} hiding data

```
void main()
{
    Rectangle r;
    r.setLength(20);
    r.setBreadth(5);
    cout << r.area(); → 20 x 0 = 0.
    cout << "Length is " << r.getLength();
}
```

r

l	20
b	5

(11) => Constructor have same name as class name.

Date : / /
Page :

(*) Constructors :=>

Types :=> (1). Default constructor => provided by compiler.

User defined { (2). Non-Parameterized "
(3). Parameterized "
(4). Copy constructor

e.g :=>

```
class Rectangle
{
private:
    int length;
    int breadth;
public:
    Rectangle()
    {
        length = 0;
        breadth = 0;
    }
    Rectangle(int l, int b)
    {
        setLength(l);
        setBreadth(b);
    }
    Rectangle(Rectangle &rect)
    {
        length = rect.length;
        breadth = rect.breadth;
    }
}
```

Non-Parameterized constr.

No need to write this constr.

Parameterized

This can be used as non-para-constr. by Rect(0,0)

Copy constr.

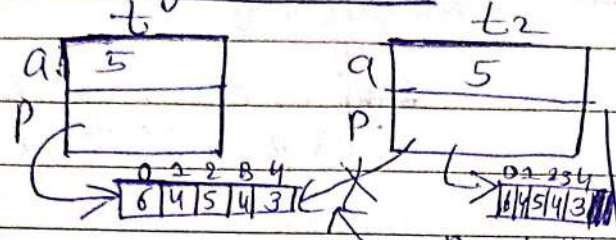
```
Rectangle r;
Rectangle r();
Rectangle r(10,5);
Rectangle r2(r);
```

r: l = 10, b = 5

r2: 10, 5

↳ It deals with the problem of "copy constructor"

Deep Copy Constructor :→ Example se samjhlo!




```
main()
{
    Test t(5);
    Test t2(t);
}
```

```
class Test
{
    int a;
    int *p;
    Test(int x)
    {
        a = x;
        p = new int[a];
    }
    Test (Test &t)
    {
        a = t.a;
        p = t.p;
        p = new int[a];
    }
};
```

Date : / /
Page :

(*) Types of Functions in a Class :-

=> Not mandatory to follow these guidelines.

↓

iss process se likhoge toh ek perfect/ideal class banega!

```
Class Rectangle
{
    private :
        int length;
        int breadth;
    public :
        Rectangle();                          // Constructors
        Rectangle(int l, int b);              // Constructors
        Rectangle(Rectangle &r);              // Constructors
        void setLength(int l);                // Mutators
        void setBreadth(int b);               // Mutators
        int getLength();                      // Accessors
        int getBreadth();                     // Accessors
        int area();                           // Facilitators
        int perimeter();                      // Facilitators
        int isSquare();                       // Enquiry
        ~Rectangle();                         // Destructor
};
```

(*) Scope Resolution Operator :->

```
class Rectangle
{
    private: int length;
    int length;
    int breadth;
    public:
        int area();
        {
            return length * breadth;
        }
        int perimeter();
};
int Rectangle :: Perimeter().
{
    return 2 * (length + breadth);
}
```

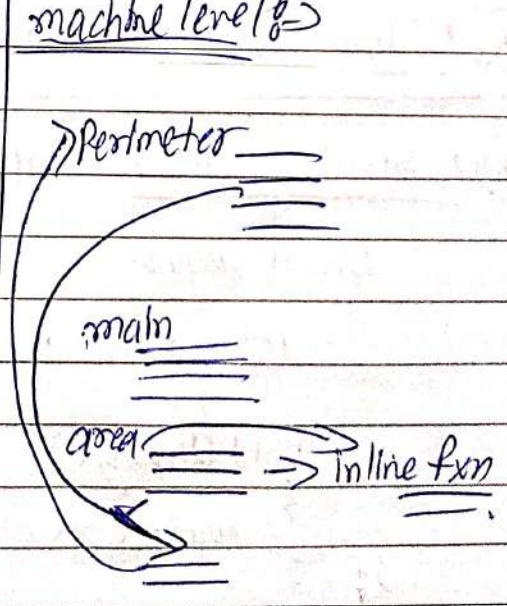
assume all fxns are written as niche dothing.

Scope resolution.

=> In C++, whenever you are using loops & conditions (logical stuff) inside then fxn., then fxn should be written outside class separately and should accessed through Scope resolution otherwise, fxn will become "inline fxn".

```
void main ()
{
    Rectangle r(10,5);
    cout << r.area();
    cout << r.perimeter();
}
```

machine level code.



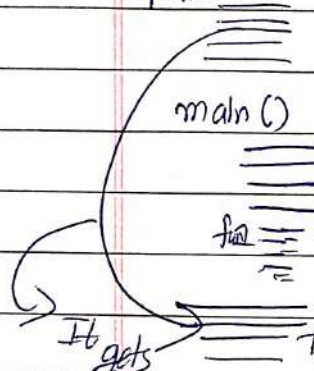
Date : / /
Page :

(*) Inline function :=>

```
int main()
{
    Test t;
    t.fun();
    t.fun2();
}
```

```
class Test
{
  public:
  -> void fun()
     {
        cout<<"In Line";
     }
inline void fun();
};
void Test::fun2()
{
   cout<<"Non-inline";
}
```

Machine Level code :=>



It gets copied inside the main fun.

(*) "This" Pointer :=> Saurabh Sir se padhna hai.

(*) Struct vs Class :=> Both are similar.

```
Struct Demo
{
   int x;
   int y;
   void display()
   {
      cout<<x<<" "<<y<<endl;
   }
};
```

```
int main()
{
   Demo d;
   d.x=10;
   d.y=20;
   d.display();
}
```

Diff:-

⇒ In Struct everything inside "struct" is "public" but in "class" by default everything is "private".

⇒ In C ⇒ Struct ⇒ Only data, No fxns.
In C++ ⇒ Struct ⇒ Both data & fxns are allowed.

(*) Operator Overloading :- C++ mein jo operators hai jaise Ki "+", "-", etc ye sb predefined data types Ke liye hote hai. Par hum Sb Inko user defined ~~class~~ datatypes Ke sath use Kr skte. Isse hi Operator overloading kehte hai!

e.g :-

Complex

$a + ib$ $\quad i = \sqrt{-1}$.

Real Img

$C_1 \rightarrow 5 + 7i$

$C_2 \rightarrow 2 + 9i$

$C_3 = C_1 + C_2 = 7 + 16i$

add.n operation

Program :-

```
main()
{
    Complex C1(3,7)
    Complex C2(5,4)
    Complex C3
    C3 = C2.add(C2);
    C3 = C1 + C2;
}
```

Operator Overloading.

```
class Complex
{
  private:
    int real;
    int img;
  public:
    Complex(int r=0, int i=0)
    {
      real=r;
      img=i;
    }
    Complex operator+ (Complex x)
    {
      Complex temp;
      temp.real= real+x.real;
      temp.img = img+x.img;
      return temp;
    }
};
```

⇒ friend Fxn.

(*) Friend Operator Overloading :⇒

```
main()
{
    Complex C1(3,7);
    Complex C2(6,4);
    Complex C3;

    C3 = C1 + C2;
```

Here, scope resolution is not used because this is a separate fxn. and also a friend fxn of above fxn.

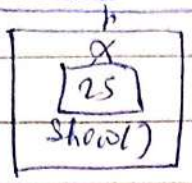
```
class Complex
{
    private :
        int real;
        int img;
    public :
    friend Complex operator+(Complex C1, Complex C2)
};
Complex operator+(Complex C1, Complex C2)
{
    Complex t;
    t.real = C1.real + C2.real;
    t.img = C1.img + C2.img;
    return t;
}.
```

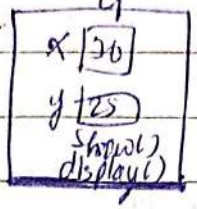
(*) Insertion Operator Overloading :⇒ / Overload Outpit Stream :⇒

```
main()
{
    Complex C1(3,7)
    C1.display();
    3+i7;
    cout << C1;
```

```
class Complex
{
    private :
        int real;
        int img;
    public :
    friend ostream & operator<<(ostream &O, Complex &C1)
};
ostream & operator<<(ostream &O, complex &C1)
{
    O << C1.real << "+i" << C1.img;
    return O;
}.
```

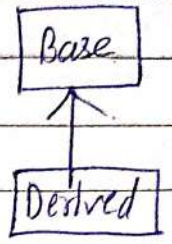
(*) Inheritance :→



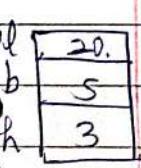



```
int main()
{
    Base b;
    b.x = 25;
    b.show();  → 25

    Derived d;
    d.x = 10;
    d.y = 15;
    d.show();  → 10
    d.display();  → 25
}
```

```
Class Base
{
  Public:
    int x;
    void show()
    {
      cout<<x;
    }
};
class Derived : public Base
  Public:
    int y;
    void display()
    {
      cout<<x<<" "<<y;
    }
};
```



Example :→




```
int main()
{
  Cubold C(20, 5, 3);
  cout<< C.getLength();
  cout<< C.volume();
  cout<< C.area();
  cout<<C;
```

```
Class Cubold : public Rectangle
{
  private:
    int height;
  public:
    Cubold(int l=0, int b=0, int h=0)
    {
      height=h;
      setLength(l);
      setBreadth(b);
    }
    int getHeight();
    void setHeight(int h);
    int Volume()
    {
      return getLength() * getBreadth() * height;
    }
};
```

```
Program :→
Class Rectangle
{
  private:
    int length;
    int breadth;
  Public:
    Rectangle(int l=0, int b=0);
    int getLength();
    int getBreadth();
    void setLength(int l);
    void setBreadth(int b);
    int area();
    int perimeter();
};
```

Date : / /
Page :

explanation :-

(*) Constructors in Inheritance :- "Pehle table ka height banta hai. Fir uske upar wala maze banta hai."

```
int main()
{
    Derived d(20, 10);
}
```

Param Default of Base 20
Param Default of derived 10.

Hidden Func

Table height and details.

```
class Base
{
public:
    Base()
    {
        cout << "Default of Base" << endl;
    }
    Base(int x)
    {
        cout << "Param of Base" << x << endl;
    }
};

class Derived : public Base
{
public:
    Derived()
    {
        cout << "Default of Derived";
    }
    Derived(int a)
    {
        cout << "Param of Derived" << a;
    }
};

Derived(int x, int a) : Base(x)
{
    cout << "Param of Derived" << a;
}
```

20 20 20

when inheriting from Base class.

when using object from base class.

(*) IsA Vs HasA :=>

class Table { HasA

Rectangle Top;
int legs;
};

class Rectangle
{
private :
Protected :
public :
}

"IsA"

class cuboid : public Rectangle
{
}

cuboid IsA Rectangle

(*) Access Specifiers :=> ① public :
② private :
③ protected :

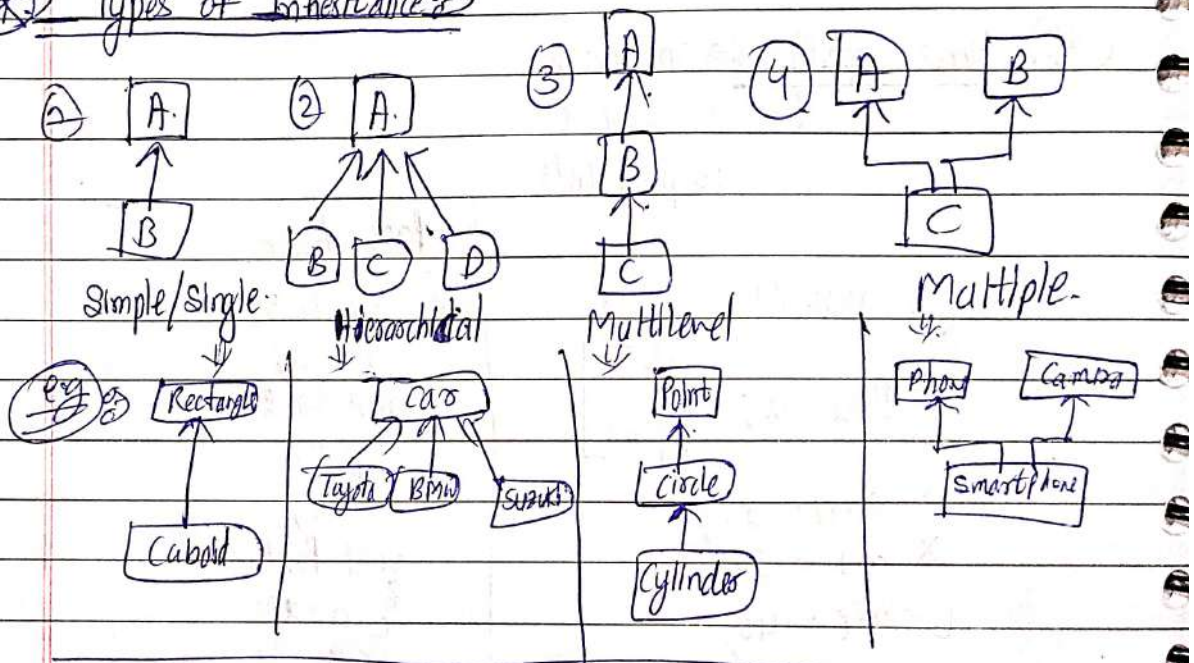
```
int main()
{
    Base x;
    x.a = 25;   ✗
    x.b = 30;   ✗
    x.c = 40;   ✓
```

x
a 25
b 30
c 40

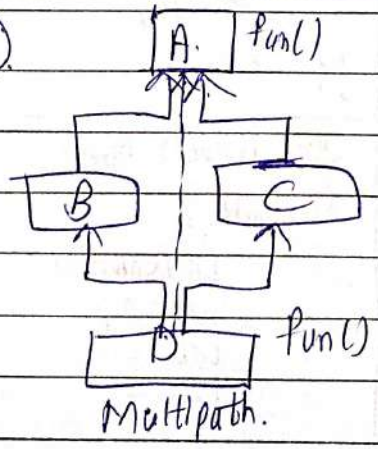
```
class Base
{
  private :
    int a;
  protected :
    int b;
  public :
    int c;
  void FunBase()
  {
    a = 10;
    b = 20;
    c = 30;
  }
};

class Derived : Base.
{
  public :
    FunDerived()
    {
      a = 1;
      b = 2;
      c = 3;
    }
};
```

	private	protected	public
inside class	✓	✓	✓
inside Derived class	✗	✓	✓
On Object	✗	✗	✓

(*) Types of Inheritance ⇒



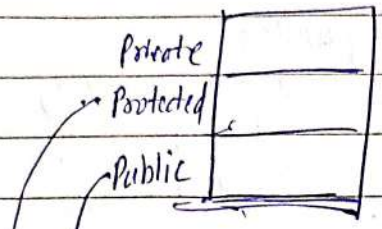
⑤



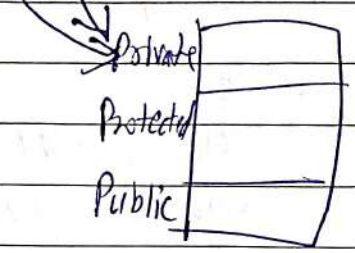

```
class A
{ =
};
class B : virtual public A.
{
};
class C : virtual public A
{
};
class D : public B, public C.
{

};
```

Ways of Inheritance :=>

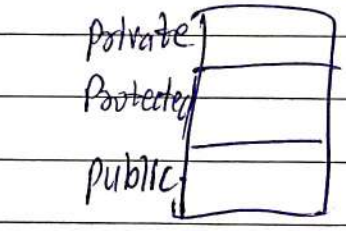
class Parent.



class child :: private Parent

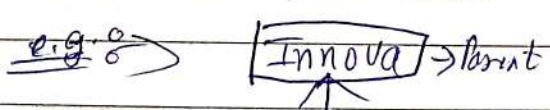


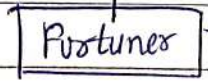
Class Grand Child :: public child.

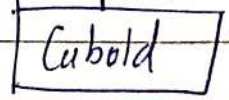


Generalization vs Specialization :=>

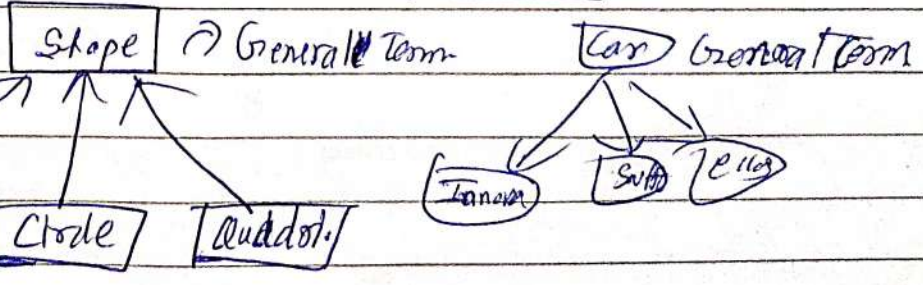
e.g :=>







e.g :=>



(*) Base class Pointer Derived Class object :⇒

```
int main()
{
    Base *p;
    p = new Base();

    p -> fun1();
    p -> fun2();
    p -> fun3();
    p -> fun4();   // call
    p -> fun5();   // =
```

```
class Base.
{
public:
    void fun1();
    void fun2();
    void fun3();
};
class Derived : public Base
{
public:
    void fun4();
    void fun5();
};
```

Polymorphism

(*) Function Overriding :-> Redefining a function of Parent class again in child class.

```
Parent P;
P.display(); -> Display of Parent

Child C;
C.display(); -> Display of child.
```

```
class Parent
{
    public:
        void display()
        {
            cout<<"Display of Parent";
        }
};
class child : public Parent.
{
    public:
        void display()
        {
            cout<<"Display of child";
        }
};
```

redefined fxn. of Parent class.

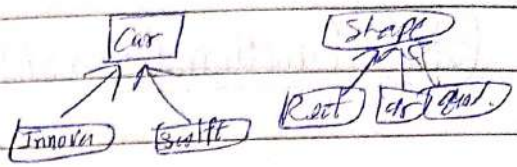
(*) Calling overrided Method :-> / Virtual Functions.

```
int main()
{
    Base *p = new Derived();
    p->fun();
}
```

```
class Base.
{
    public:
        "Virtual" void fun()
        {
            cout<<"fun of Base";
        }
};
class Derived : public Base.
{
    public:
        void fun()
        {
            cout<<"fun of Derived";
        }
};
```

If this is used then, this will happen otherwise it will point Baseclass fxn.

Polymorphism :->




```
main()
{
    Car *C = new Innova();
    C->start(); -> "Innova started"
    C = new swift();
    C-> start(); -> "Swift started"
```

```
class Car
{
  public:
  virtual void start() = 0;

  virtual void stop() = 0;
                  ↖ pure virtual function.
};

class Innova : public Car.
{
    public:
      void start()
      {
        cout<<" Innova Started";
      }
      void stop()
      {
        cout<<" Innova stopped";
      }
};

class Swift : public Car
{
    public:
      void start()
      { cout<<"Swift started";
      }
      void stop()
      { cout<<"Swift stopped";
      }
};
```

(*) Abstract Classes :=>

Abstract.

Abstract.

interface

Base :=> All Concrete fxns	Base :=> Some Concrete fxns; Some pure Virtual fxn;	Base :=> All pure Virtual fxns
↓	↓	↓
=> Reusability.	=> Reusability. => Polymorphism	=> Polymorphism.

eg :=>

```
X   Base b;
    Base *p = new Derived;
    p->fun2();
```

Abstract ⟹

```
class Base.
{
  public:
  void fun2()
  { cout<<" Base fun2";
  }
  virtual void fun2()=0;
};
class Derived: public Base.
{
  public:
    void fun2()
    { cout<<" Derived fun2".
    }
};
```

(*) Friend Functions :-

```
class Test
{
  private:
    int a;
  protected:
    int b;
  public:
    int c;
  friend void fun();
};
void fun()
{
  Test t;
  t.a = 25;
  t.b = 20;
  t.c = 5;
}
```

Due to this

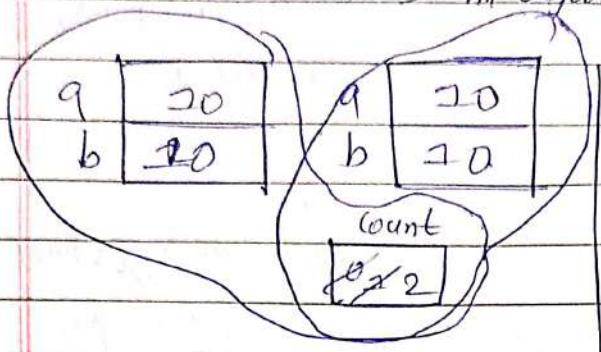
(*) Friend class :- Declare

```
class your;
class My
{
  private:
    int a=10;
  friend your;
};
class your
{
  public:
    My m;
  void fun()
  {
    cout<<m.a;
  }
};
```

Due to this

Static Members :=> They belongs to a class, not to an object.

- They acquire memory only once.
- All objects can access this member.




```
main()
{
    Test t1;
    Test t2;
    cout<<t1.count;  => 2
    cout<<t2.count;  => 2
    cout<<Test::count; => 2
}
```

```
class Test
{
    private:
        int a;
        int b;
    public:
        static int count;
        Test()
        {
            a=10;
            b=10;
            count++;
        }
};
```

Again initialization is required. => `int Test :: count=0;`

Static Member functions :=>

```
main()
{
    cout << Test::getcount() => 0
    Test t1;
    cout<<t2.getcount();
    Test t2;
}
```

```
class Test
{
    private:
        int a;
        int b;
    public:
        static int count;
        Test()
        {
            a=10;
            b=20;
            count++;
        }
        static int getcount()
        {
            a++;
            return count;
        }
};
int Test::count=0;
```

=> Static member functions can only access static data members of a class.

Q => Examples :=>

(1)

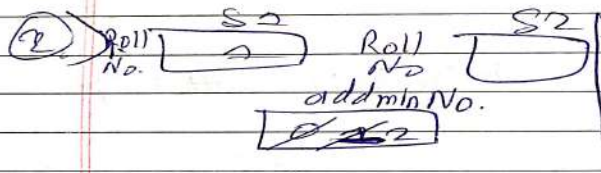
```
main()
{
    cout << Innova::getPrice();
    Innova my;
    cout << my.getPrice();
}
```

Calling upon Class → `Innova::getPrice()`

Calling upon object → `my.getPrice()`

```
class Innova
{
    public:
    static int price;
    Innova()
    {
    }
    static int getPrice()
    {
        return price;
    }
};
int Innova::price=20;
```

(2)



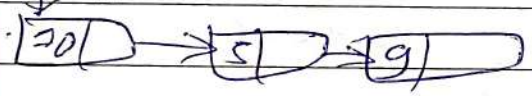

```
main()
{
    student S1;
    student S2;
}
```

```
class student
{
    public:
    int rollno;
    static int addminNo;
    student()
    {
        addminNo++;
        rollno = addminNo;
    }
};
int student::addminNo=0;
```

Inner/Nested Classes :->

Example :->

```
class LinkedList
{
    class Node
    {
        int data;
        Node *next;
    };
    Node *Head;
};
```



Explanation :->

```
class Outer
{
    public:
    ① int a = 10;
    ② static int b;
    void fun()
    {
        i.show();
        cout<<i.x;
    }
    class Inner
    {
        public:
        int x = 25;
        void show()
        {
            cout<<b;
        }
    };
    ③ Inner i;
};
int outer::b = 20;
```

(*) Exception Handling :→

User level → Runtime Error ⇒ due to
- (1). bad Input.
- (2). problem with Resources.

↳ Exception

(*) Exception Handling construct :→ example :→

Just like if-else.

```
try
{
    1. ——
    2. ——
    3. ——
}
catch ()
{
    1. ——
    2. ——
    3. ——
}
```

(1)
```
int main()
{
    int a=20, b=0, c;
    try
    {
        if (b==0)
            throw 20;
        c=a/b;
        cout<<c;
    }
    catch (int e)
    {
        cout<<"Division by zero"<<"error code"<<e;
    }
    cout<<"Bye";
}
int division (int x, int y)
{
    if (y==0)
        throw 1;
    return x/y;
}
```

(2)
```
int main()
{
    int a=20, b=0; c;
    try
    {
        (1) c=division(a,b);
        ~~xxxcout<<c;~~
    }
    catch (int e)
    {
        (2) cout<<"Division by zero"<<"error code"<<e;
    }
    cout<<"Bye";
}
```

(*) All about "throw" :=>

Built-in class of "C++".

```
class My : public exception
{
};
int division (int x, int y) throw (My Exception)
{
    if (y==0)
        throw MyException;
    return x/y;
}
```

Optional

(*) All about "Catch" :=> Can have multiple catch block.

```
try
{
    1 ——— ✓ int
    2 ——— ✓ MyException.
    3 ——— ✓ float; char
    4 ——— ✓
    5 ———
}
Catch (int e)
{
    =
}
Catch (float e)
{
    =
}
```

Catch All

```
Catch (...)
{
    =
}
```

=> This block should always be used in the last.

catch (MyExceptione)
{
    =
}

Nested try & catch :-

#1.)
```
try
{
1. —
2. —
3. —
try
{
4 —
5 —
}
catch (— —)
{
=
}
}
Catch ()
{

}
```

#2.)
```
class MyException1.
{
};
class MyException2 : public MyException2.
{
};
try
{
=
}
catch (MyException2 e)
{
}
catch (MyException1 e)
{
}
```

→ child class should be written first.

(*) Templates → functions & classes :->

(#) template<class T>

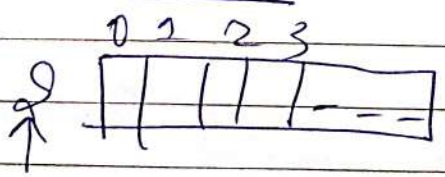
=> functions :-> T maximum(T x, T y)
{
return x>y? x; y;
}

maximum (10, 25);
maximum (12.5, 9.5);

Ex :-> (#) template < class T (int), class R (double) >
void add (T x, R y)
{
cout << x+y;
}

add (10, 12.9);

(#) Template classes :->



Top = -1.

```
template<class T>
class stack
{
private:
    int a[20];
    int top;
public:
    void push(T x);
    T pop();
};
```

outside class.

```
template <class T>
void stack<T>::push(T x)
{
    ---
}
---
template<class T>
T stack<T>::pop()
{
    ---
}
```

Objects =>

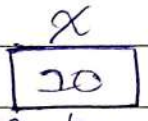
```
Stack <int> s;
Stack <float> s2;
```

(*) Constant Qualifier =>

Usage =>

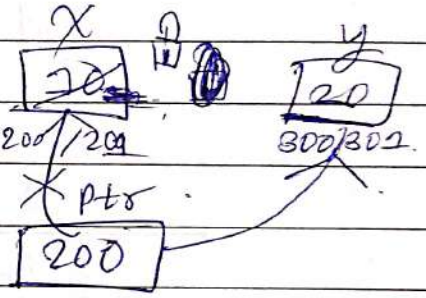
(#) (1) =>

```
int main()
{
    const int x = 10;
    x++;
    cout << x;
```



(#) (2) =>

```
int main()
{
    int x = 10;
    const int *ptr = &x;
    ++*ptr;
    cout << x;  -> 11
    cout << *ptr;  -> 10
```

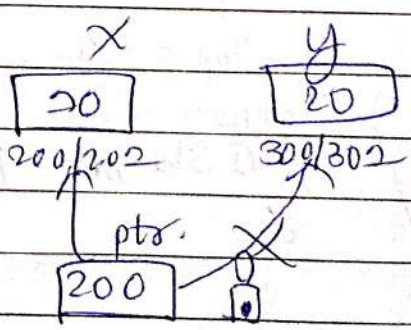


int const *ptr = &x;

```
    int y = 20;
    ptr = &y;
    ++(*ptr);
```

(#) (3) =>

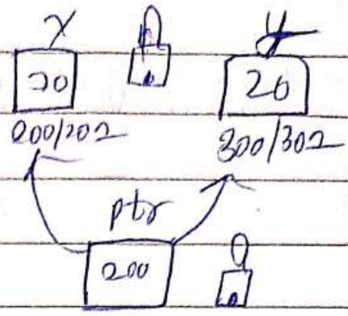
```
int main()
{
    int x = 10;
    int * const ptr = &x;
    int y = 20;
    ptr = &y;
    ++(*ptr);
```



This is read as "ptr" is constant ⇐ int * const ptr = &x;
and pointer (*) to an integer. -> ✗ ptr = &y;

#). (4).)

```
int main()
{
    int x=10;
const int * const ptr = &x;
    int y=20;
→ X ptr = &y;
  X ++(*ptr);
```

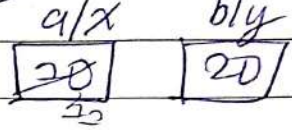


#) Constant Keyword in functions :->

example :->

```
①. class Demo
    {
      public:
        int x=10;
        int y=20;
        void Display() const
        {
          X x++;
          cout<<x<<" "<<y<<endl;
        }
    };
    int main()
    {
        Demo d;
        d.Display(); → 11,20 X
```

(2). Void fun (const int &x, int &y)
{
x++;
cout<<x<<" "<<y;
}
int main()
{
int a=10, b=20;
fun (a, b);
}



(*) Preprocessor Directives/Macros :=>

(1). #define PI 3.1425 → Symbolic constant
#define C cout.

int main()
{
cout<<PI;
3.1425
C<<20;
cout
}

(2). #define SQR(x) (x*x).
#define MSG(x) #x
"x"

int main()
{
cout<<SQR(5);
5*5
cout<<MSG(Hello);
"Hello".
}

Date / /
Page

(3) #ifndef (If not define.)
#define PI 3.2425.
[redacted]
#endif

(*) Namespaces :=> Used to resolve conflict between names;

```
namespace First
{
    void fun()
    {
        cout<<"First";
    }
}

namespace Second
{
    void fun()
    {
        cout<<"second";
    }
}

=> using namespace First;
   int main()
   {
       fun();
       Second::fun();
   }
```

(*) Destructor :=>

```
main()
{
    Test *p=new Test();
    delete p;
}

class Test
{
public:
    Test()
    {
        cout<<"Test created";
    }
    ~Test()
    {
        cout<<"Test destroyed";
    }
};
```

Dest:

=> Uses of distructor => Syntax :=>

```
class Test
{
    int *p;
    ifstream fis;

    Test()
    {
    ⟶ p = new int[20];
    ⟶ fis.open("my.txt");
    }
    ~Test()
    {
    ⟶ delete []P;
    ⟶ fis.close();
    }
};
```

(*) Virtual Destructor :=>

```
int main ()
{
    Derived d;
    .
    .
    .
}
```

Called :=>
Base Const ---
Derived Const ---

```
    .
    .
    .
}
```

=> Derived dest ---
=> Base dest ---

```
class Base
{
public:
    Base ()
    {
        cout << "Base constructor" << endl;
    }
    virtual ~Base ()
    {
        cout << "Base destructor" << endl;
    }
};

class Derived :: public Base
{
public:
    Derived ()
    {
        cout << "Derived constructor" << endl;
    }
    ~Derived ()
    {
        cout << "Derived Destructor" << endl;
    }
};
```

code :=>

```
int main ()
{
    Base *p = new Derived ();
    .
    .
    delete p;
}
```

~~Output~~ Derived Des ---
Base Destructor

(*) IO streams :- (ios) → (class)

ios
- fstream
  - ifstream → input file stream
- ostream
  - ofstream.

(*) Writting in a file :- ios::app (append)
ios::trunc → truncate → delete

```
#include<fstream>
int main()
{
    ofstream Outfile("my.txt");
    outfile<<"Hello"<<endl;
    outfile<<25<<endl;
    outfile.close();
}
```

My.txt
Hello
25.

(*) Reading from a file :- (ios::in / ios::out) → no need to mention.

```
#include<fstream>
int main()
{
    ifstream infile;
    infile.open("my.txt");
    if(!infile.is_open())
        cout<<"file cannot be opened";
```

My.txt
Hello
25

Reading

```
string str;
int x;
infile >> str;
infile >> x;
cout << str << " " << x;
```

check end of file.

```
if (infile.eof()) cout << "end of file reached";
infile.close();
```

(*) Text and Binary Files :=>

Text

00110010
00110011

↳ more space

Binary

00000000 00000010 0011

↳ faster

↳ less space.

int x = 23

ASCII 50 51.

00110010	00110011.

=> ~~ifstream~~ mode : ios :: binary

read()
write().

Date : / /
Page :

Manipulators :- Used for enhancing strings or formatting strings.

↳ endl → \n

```
cout << endl;
cout << "\n";
```

```
cout << hex << 163;
        → A3
```

integers
- hex
- oct
- dec

Float
- fixed
- scientific

```
cout << fixed << 325.731;
```

~~cout << scientific~~

Displayed in 20 spaces.

```
cout << setw(20) << "Hello";
```

other
- setw()
- left
- right

ws → whitespace

```
cout << 20 << ws << 20;
```

↳ Displayed with one space.

(*) STL → They have (1). Algorithms.
(2). Containers.
(3). Iterators

(1) Algo → fxns: search()
sort()
binary_search()
reverse()
concat()
copy()
union()
intersection()
Merge()
Heap

(2). Containers → Template classes
- vector → push_back() pop_back() insert() remove() size() empty()
- list
- forward_list
- deque
- priority_queue ← push() pop() empty() size()
- stack → LIFO
- set → unique elements
- Multiset → same as set but allows duplicate

push_front() pop_front() } list fxns
front() back() } list fxns

Map / MultiMap → In this Key is can be duplicated.
uses Hash Table.
<Key ; value> pair.

Key	value
1	"John"
2	"Ajay"
3	"Khan"

Date : / /
Page :

Using STL Classes

Examples:

① #include <vector>

```
main()
{
    vector<int> V = {10, 20, 40, 90};
    V.push_back(25);
    V.push_back(70);
    V.pop_back();
```

③ Iterations →

1st method

```
for (int x : V)
    cout << x;
```

2nd method →

```
vector<int>::iterator itr;
for (itr = v.begin(); itr != v.end(); itr++)
    cout << *itr;
```

② #include <list>

```
main()
{
    list<int> V = {10, 20, 40, 90};
    push_back(25);
    list<int>::iterator itr;
    for (itr = v.begin(); itr != v.end(); itr++)
        cout << *itr;
```

(*) C++ 03 => C++ 11 =>

(1) auto :=> See code section.

(2) final Keyword :=> || -> It restricts inheritance.

(3) Lambda Expressions :=> Useful for defining unnamed function.

Syntax :=>

[capture_list] (parameter_list) -> return type { body };

(4) main()
{
auto f = [] () { cout << "Hello"; };
[] (int x, int y) { cout << "sum: " << x+y; } (10,5);

int x = [] (int x, int y) { return x+y; } (10,5);
f();

int s = [] (int x, int y) -> int { return x+y; } (10,5);

(5) main()
{
int a = 10;
int b = 5;

[&a, &b] () { cout << a << " " << b; } ();

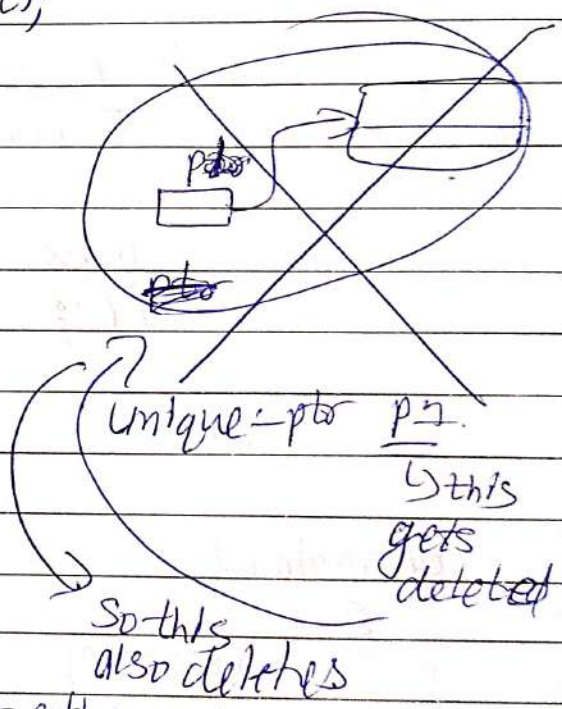
[&] capture list

}

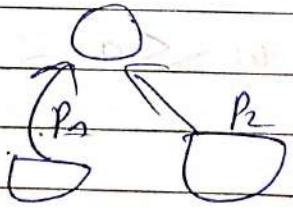
(*) Smart Pointers :→ Solves problems of memory Leak.

1. unique_ptr.
2. shared_ptr
3. weak_ptr.

(#) fun ()
{
~~point~~
unique_ptr<~~rectangle~~> p1(new Rectangle(10,5));
cout << p1 -> area();
cout << p1 -> perimeter();
}
main()
{
while (1)
{
fun();
=
}
}.



(#) 1. unique ptr :→ can't point more than one pointer.



2. shared_ptr :→ ~~Reference~~ Ref_counter = 2

3. weak_ptr :→ use_count();

Ellipsis :→ Used for taking variable number of arguments in a fxn.

```
int Sum(int n, ------)
{
    va_list list;
    va_start(list, n);
    int s=0;
    for(int i=0; i<n; i++)
        s+= va_arg(list, int)
    va_end(list);
    return 0;
}
cout<< sum(3,20,20,30)
cout<< sum(5,9,4,2,6,3,7)
```
