Operator Overloading: giving special meaning to existing operators.

```
int a, b, c;
a = b+c;
```
} → fine! (data type of function) ↓ built in

```
class dist{ }
dist d1, d2, d3;
d3 = d1 + d2
```
} normal addition can't be performed (data type → user defined)

∴ we define the addition function as well

```
class Counter {
    int c;
    public:
        Counter () : c(0) { }

        int getcount()
        { return c; }

        void operator ++ ()
        {
            ++c;
        }
};
```

void ↓ return type; operator ↓ keyword; ++ ↓ operator to be overloaded

→ prefix increament operator

UNIT-I - PROCEDURAL PROGRAMMING

L	T	P	C
3	0	0	3

LAB:

Program #1: to print value of diff data types

```
using namespace std;
#include <iostream>
int main()
{
  cout << "Welcome to CPP";
  Int a = 10;
  float b= 10.1;
  char msg[] = "Hi";
  long int Ind-POP = 10000000;
  double PI = 3.1416;
  cout << a << b << msg << PI << Ind-POP;
  return 0;
}
```

(or) #2

```
(int a; float b;)
{
  cout << "Enter int value";
  cin >> a;
  cout << "Enter float value";
  cin >> b;
  cout << a << b;
  return 0;
}
```

#3 → find area of triangle

```
#include <math.h>
{
    float a, b, c; s; area;
    cout << "Enter";
    cin >> a >> b >> c;
    s = (a+b+c) / 2;
    area = sqrt(s*(s-a)*(s-b)*(s-c));
    cout << "area of Δle is" << area;
    return 0;
}
```

#4 → calculate distance b/w 2 points

```
#include <math.h>
{
    float a, b, c, d, dist;
    cout << "Enter 1st set of points";
    cin >> a >> b;
    cout << "Enter 2nd set of points";
    cin >> c >> d;
    dist = sqrt(pow(a-c, 2) + pow(b-d, 2));
    cout << "The distance is" << dist;
    return (0);
}
```

#5 → basic arithmetic operations

\n → endl

```
{
    int a, b;
    cout << "Enter the values";
    cin >> a >> b;
    cout << a+b << a-b << a/b << a%b;
    return (0);
}
```

(don't use for float)

#6 → swap 2 variables w/o using temp

```
{
    int a, b;
    cout << "Enter the values";
    cin >> a >> b;
    a = a+b;
    b = a-b;
    a = a-b;
    cout << "The values are: " << a << endl << b;
    return (0);
}
```

isalpha(ch).
isdigit(ch)
ispunct(ch)

#7 → size of every data type

```
{
    cout << sizeof (int);
    cout << sizeof (float);
    cout << sizeof (double);
    cout << sizeof ( long int);
    return (0);
}
```

#9

```
{
    if (a >= 0) {
        a++;
    }
}
```

#10

isalpha(ch)

#14

```
{
    int n, limit, big = -1;
    cout << "Enter how many i/p";
    cin >> limit;
    cnt = limit;
    while (limit > 0)
    {
        cin >> n;
        if (n > big) {
            big = n;
        }
        limit--;
    }
    cout << "biggest no. is " << big;
```

(OR)

```
    sum = 0
    while (limit > 0)
    {
        cin >> n;
        big = (n > big) ? n : big;
        limit--;
    }
    cout << " big" << big;
    cout << " avg" << (sum / cnt);
```

```
big = -1;
while(lim>0){
cin>>n;
if (n > big)
  big = n;
lim--
```

```
big = n>big ? n:b
big = n>big ? n : big
```

Sum = Sum + n

float

~~eff:~~

#15 Factorial

```
int n;
~~fact=1~~;
cout << " Enter no. ";
cin >> n;
~~for (n>0)~~
for ( ~~fact=1~~ int fact=n; n>0; n--) {
        fact = fact * n;
}
cout << " fact of n " << fact;
return 0;
```

(or)

```
int n, f=1;
cin >> n;
for (int i=1; i<=n; i++) {
        f = f * i;
}
cout << f;
return 0;
```

```
if (n<0.)
    error

if (n=0 || n=1)
    f=1
```

eft:

(*) Finding if a gn no. is perfect:

```
{  float div; sum;
   cin >> n;
   for (int i = 1; i < n; i++) {
        div = n % i
        sum = sum + div
        if (div == 0) {
            sum = sum + i;
        }
   }
   if (n == sum) {
        cout << " It is a perfect no.";
   }
return 0;
}
```

~~Q4~~:

(*) Prime no. or not:

```
{   float div;
    int count=0;
    cin >> n;
    for (int i=1; i<=n; i++){
        if (n%.i == =0){
            count++;
        }
    if (count == 2){
        cout<< " prime";
    }
    else {
        cout << "not prime";
    }
return 0;
}.
```

>> → extraction

<< → insertion

FUNCTIONS :- a set of codes used for performing a specific function.

↳ easily re called
↳ makes debugging easy

no return, no arg

```
void fact();   → declaration of function
void perf();
int main(){
    int n;
    cin >> n;
    fact();
    return(0);
}
```

```
void fact(){
  int i, f=1, n;
  cin >>n;
  for(i=1; i<=n; i++){
      f = f*i;
  }
  cout << f;
}
```

```
void perf(){
float div, sum;=0;
cin >> n;
for(int i=1; i<n; i++)
   div = n%i
   if(div==0){
      sum=sum+i;
   }
if(n==sum){
cout<<"yes";
}
```

SA:

No return, w/ arg

```
void fact (int);

int main(){
    fact(n);        → actual arg / actual para
    return 0;
}

void fact (int n){      → formal arg / formal para

    definition;
}
```

with return, with arg

```
int fact (int);
int main() {
    int res;
    int n;
    cin >> n;
    res = fact (n);
    return 0;
}

int fact (int n) {

    ~~~~~~~~~~

    return f;
}
```

```
int perf(int);
int main(){
    int n;
    cin >> n;
    int div;
    p = perf(n);
    if (p == n){
        cout << "yes";
        }
    else {
        cout << "no";
     }
    result
    return 0;
}

int perf(int n){
    int s;
    int i = 1;
    float div;
    for(i = 1; i < n; i++) {
        div = n % i
        if (div == 0){
            sum = sum + i
            }
        return (sum);
    }
}
```

```
int square(int);
int main(){
    int n;
    cin>>n;
    cout<<"Square"<<square(n);
    return(0);
}

int square(int n){
    return(n*n);
}
```

Function advantages:

- Makes the computation logically separate
- can be used in more than one place
- easy debugging / testing

Manipulators: → endl ⇒ next line

→ setw(w) ⇒ leaves spaces ⇒ #include <iomanip>
setw(6)

Ex: cout << setw(6) << "city" << setw(16) << "Rain"
<< endl;

⇒ __city_____Rain

cout << setiosflags(ios::left) << setw(6) << -17

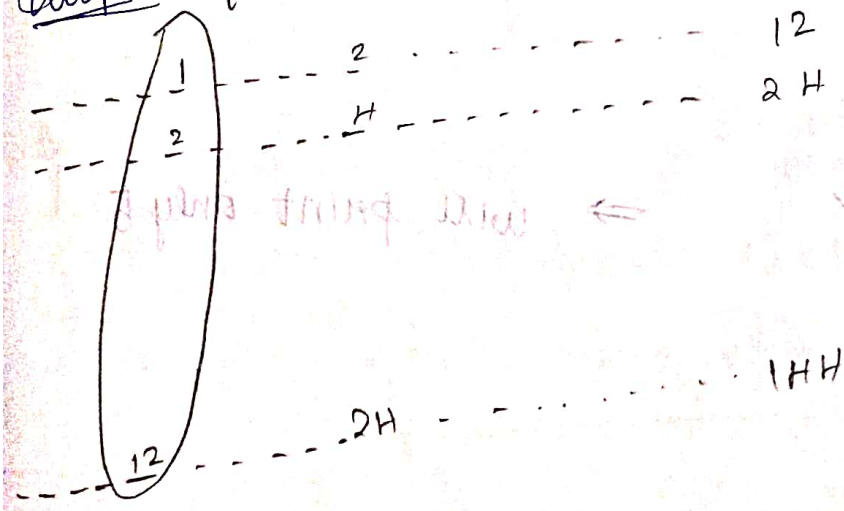
scope resolution operation.

↳ prints from left

```
#include<iomanip>
int main(){
    const int max=12;
    for(int i=1; i<=max; i++){
        for(int j=1; j<=max; j++){
            cout<<setw(5)<<i*j;
        }
        cout<<endl;
    }
    return 0;
}
```

setiosflags(ios::left) << setw(6) cout << setiosflags(ios::left) <<

Output:



```
int main()
{
  int i;
  for(i=1; i<=10; i++){
      cout << setw(4) << i;
      int cube = i*i*i;
      cout << setw(10) << cube << endl;
  }
}
```

case ↓ int / char

TYPE CONVERSION:

```
void print(int x)
{
    cout << x;
}

int main()
{
    print(5.5);
}
```

⇒ will print only 5

~~of~~

```
void print (int x)
{
    cout << x;
}

int main ()
{
    print (7/2.0)
}
```

static_cast <type>

7/2.0 ⇒ 7.0/2.0 ⇒ 3.5 ⇒ void Print (3.5)

⇓

3 //

```
int main()
{
    int v;
    v = 10000000000;
    v = (v*10)/10;
    cout << v;
}
```

→ is actually long int so inaccurate output

```
v = 10000000000;
v = (static_cast<double>(v)*10)/10;
cout << "second value of v" << v;
```

→ intermediate variable giving enough space

```
{
    float f = 3.5;
    int a = static_cast<float>(f);
    cout << a;
}
```

↳ it will still give 3

SAFE CONVERSIONS:

1. Boolean to char
2. Boolean to integer
3. Boolean to double
4. char to int
5. char to double
6. int to double

→ these conversions give accurate outputs.

Ex:

①
```
char c = 'x';
int i = c;
int j = 'x';
```
i and j get the value 120

```
char c2 = i;
```
c2 → 'x'

②
```
double d1 = 2.2;
double d2 = d1 + 2;
if (d1 < 0){
    cout << "error";
}
```

UNSAFE CONVERSIONS:

① Double to int

② Double to char

③ Double to boolean

Vectors: • Similar to arrays but can change size during compile time.

#include <bits/stdc++.h>

Syntax: vector <data type> object (size)

→ vector <int> V(6);

V[0] = 10
V[1] = 20
:
V[5] = 60

bits/stdc++.h

vector <> obj(size);

→ vector <string> stu(59);

stu[0] = " "
:
stu[31] = 91 ✗

→ every value is 0.1

→ vector<double> V(100, 0.1);

~~eft:~~

Computing mean and median of gn data:

```
int m()
{
    vector<double> temps;
    double temp; sum=0.0;
    while (cin >> temp)
        temps.push_back(temp);
```

returns true if the right value is passed here if temp is double;

~~for(i=0; i<temp~~

```
    for(i=0; i<temps.size(); i++)
        sum += temps[i];
    cout << "avg" << sum / temps.size()

    sort(temps.begin(), temps.end());
    cout << "median" << temps[temps.size()/2];
    return 0;
}
```

```
int m()
{
    vector<string> words;
    ~~double~~ char word[10];
    while(cin >> ~~words~~ word)
        words.push_back(words)
    cout << "no. of words" << words.size();
```

→ exits : ctrl + Z

sort(words.begin(), words.end());
for(i=0; i<words.size(); i++){
    if(i==0 || words[i-1] != words[i])
        cout << words[i];
}

ARRAYS :-

~~int sum(int a[], int n)~~ ?? int sum(int, int)

int a[] = {20, 10, 30, 40, 50}
int n = sizeof(a)/sizeof(int)
cout << "sum" << sum(a, n);
↳ holds the address of the 0th element.

int sum(int a[], int n){
    int i;
    int s=0;
    for(i=0; i<n; i++){
        s = s + a[i]
    }
    return(s);
}

vector<int> temp;
int num;
sum=0;
while(cin >> num){
    temp.push_back(num);
}
for(i=0; i<temp.size(); i++){
    sum = sum + temp[i]
}

Q11:

Find the smallest no. in the array:

```
int a[100], n;
cout << "Enter no. of elements";
cin >> n;
cout << "Enter numbers: ";
for (int i=0; i<n; i++){
        cin >> a[i];
   }
min = a[0];
for(i=1; i<n; i++)
 { if (a[i] < min){
        min = a[i]; }
 }
cout << "min" << min;
```

Modular:

```
int min (int a[], int n) {
  int mini = a[0];
  for (i=1; i<n; i++) {
       if (a[i] < mini) {
          min = a[i]; }
  }
 return (mini);
}
```

~~21.~~

Linear Search:

```
int found = 0;
int a[100], n;
cout << "Enter no. of ele";
cin >> n;
for (int i = 0; i < n; i++) {
        cin >> a[i];
}
```

```
void Lsearch (int a[], int n, int key) {
    cin >> key;
    for (i=0; i<n; i++) {
        if (key == a[i])
        { found = 1;
          break;
        }
    }
    if (found)
        cout << "success";
    else
        cout << " nope";
```

```
for (i=0; i<n; i++)
    if (key == a[i]){
        f = 1;
        break;
    }
if (f
```

2- dimensional array:

```
int a [3] [3];
```

row ↓ ↳ column

```
for (int i =0; i< 3; i++ ){
    for (int j=0; j< 3; j++ ){
        cin >> A[i][j];
    }
}
```

#To find sum of every row

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

~~21~~:

To find sum of every col:

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

To find if the matrix is an Identity mat

```
is_identity = 1
for(int i=0; i<3; i++) {
    for (int j=0; j<3; j++) {
        if (i==j && a[i][j] != 1) {
            is_identity=0; }
            break;
        else if (i!=j && a[i][j] != 0) {
            is_identity =0; }
            break;
        else
            is_identity =1;
    }
}
if (is_identity)
    print("Identity");
else
    nope;
```

```
void print(int a[][3], int N, int m){
                          ↳ must give 2nd dimension
    int R,C;
    for(R=0; R<N; R++){
        for(C=0; C<M; C++){
            cout << a[R][C];
        }
    }
}
int main()
{
    int A[3][3] = {{1,2,3},{3,4,5},{4,5,6}};

    print(A,3,3);
}
```

Pointers and Arrays:

```
int *ptr;
ptr = &A;
ptr = A;
ptr = &A[0];
```

↓ reference operator

} all these help store base address of an array A in ptr.

```
int main(){
    int a[5] = {10,20,12,19,21};
    int *p = a, i;
    for(i=0; i<5; i++){
        cout<<(p);        → prints the base address five times
        cout<<(p+i);      → prints the consecutive addresses
        cout<<*(p+i);     → prints the content @ p+i address
    }
```

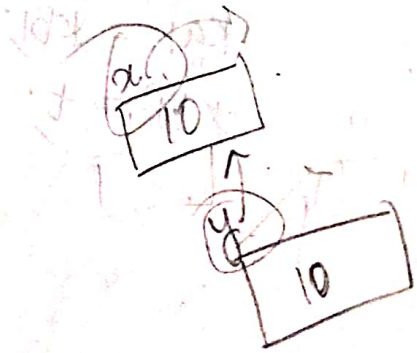
dereference operator

```
int x;
int &y = x;   → alias name for x
y = 10;
cout<<x;
x = x+20;     → any change in one var changes the other var
cout<<y;
```

int x;
int &y = x;
(y = 10)

```
int x=5;
int &x=x;,
int &y;   → error
cout<<x;  → 5
cout<<z;  → 5
z=9;
cout<<x;  → 9
cout<<z;  → 9
```

int *x = 10 also
int *y = 100



~~Q1:~~

→ Reference variable contains the address of another variable

→ No need de-referencing like pntrs

→ must be initialized when declared

```
void P_swap(int *, int *);
void R_swap(int &, int &);
int main(){
    int a, b;
    cin >> a >> b;
  P_swap(&a, &b);   R_swap(a, b)
   cout << a << b;
}

void P_swap (int *a1; int *b1){
        int t;
        t = *a1;   → de-reference
        *a1 = *b1;
        *b1 = t;
}
```

```
int main(){
    foo ob;   → now the class has a memory space
    ob.get();
    ob.put();
    return 0;
}

class std
{
    int reg_no;
    char name[20];
    char dept[5];
public:
    void get(){
      cout<<"enter details:";
      cin>>reg_no>>name>>dept;
    }
    void put(){
      cout<<reg_no<<name<<dept;
      {
};
```

Instance of class / class variable

```
ptr = new char('B');
*ptr = 'B';
```

the members functions can be defined outside the class also (after declaring it inside the class of)

```
void stu::get()
{
    cin >> reg_no >> name >> dept;
}

void stu::put()
{
    cout << reg_no << name << dept;
}

class small
{
    int somedata;
    public:
        void setdata(int d)
        {
            somedata = d;
        }
        void showdata()
        {
            cout << somedata;
        }
};
```

stu::

doing this because we can't access some data directly from m(), we can do it only thru a public member function

```
int main(){
    small s1, s2;
    s1.setdata(100);
    s2.setdata(200);
    s2.showdata();
}

class part
{   int model_no;
    int p_no;
    float price;
    public:
        void set(int m, int p, float pr)
        {   m_no = m;
            p_no = p;
            price = r;
        }
        void show()
        {   cout << m_no << p_no << price;
        }
};
```

```
int main(){
    part p1, p2;
    p1.setpart(625H, 50, 1000);
    p2.setpart(1154, 25, 20000);
    p1.show();
    p2.show();
}
```

(or)

```
{
public:
    void set()
    {
        cin >> m_no;
        cin >> p_no;
        cin >> price;
    }

int main(){
    part p[2];
    for(int i=0; i<2; i++){
        p[i].set();
    }
    for (int j=0; j<2; j++){
        p[j].show();
    }
return 0;
}
```

```
class dist
{
    int ft;
    float in;
  public: →
    void input1( int p; float q) { @ compile time
    {   ft = p;
        in = q;
    }
    void input2() { @ runtime
        cin >> ft;
        cin >> in;
    }
    void output(){
        cout << ft << in;
    }
};
int main(){
   dist P1, P2;
   P1. input1( 10, 2.3);
   P2. Input2( );
   P1. output();
   P2. output();
}
```

Function Overloading: (many func. w/ the same name but w/ diff signature.)

→ You can define more than one function w/ the same name w/ diff no. of arguments.

→ " " " " " " " "

" " " " " data type of arguments.

⇒ cant differentiate functions w/ return type.

Ex:

```
Class demo
{
    int a, b, c;
    public:
      int sum (a, b)
      {
        cout << a+b;
      }
      int sum (a, b, c)
      {
        cout << a+b+c;
      }
}
```

⇒ C doesnot support function overloading.

- void repchar();
  void repchar(char);
  void repchar(char, int);

```
int main(){
    repchar();
    repchar('=');
    repchar('+', 30);
    return(0);
}

void repchar()
{
    for(i=0; i<50; i++)
    {   cout << '*';
    }
}

void repchar(char c)
{
    for(i=0; i<50; i++){
        cout << c;
    }
}
```

void repchar()
void repchar(char
void rep

(this is compile time polymorphism)

```
void repchar(char c, int n)
{
    for(int i=0; i<n; i++){
        cout<<c; }
}
```

```
#Define PI 3.141
void area(int);
void area(float);
void area(int, int);
void main(){
    area(4);
    area(5.0);
    area(4,5);
}
void area(int a){
    cout<<a*a;
}
void area(float a){
    cout<<PI*a*a;
}
void area(int a, int b){
    cout<<a*b;
}
```

→ symbolic constants

⇒
```
float absolute (float v) {
    if (v < 0.0)
        v = -v
        return v;
}

int absolute (int v) {
    if (v < 0)
        v = -v;
        return v;
}
```

inline

Inline functions:

```
inline float lbstokg ( float p)
{
    return 0.453*p;
}

void main() {
    float lbs;
    cin >> lbs;
    cout << "wt in kg" << lbstokg(lbs);
}
```

Instead of going to the function def. processing it there & coming back, the inline function simply replaces the func. call with the func. definition

(only for single line function)

Default arguments:

→ In function prototype itself

```
void repchar( char = '*', int = 30);

void main(){

    repchar();
    repchar('=');
    repchar('+', 45);
}

void repchar(char c, int a){
    for(int i=0; i<a; i++)
        cout << c;
}
```

defining the function once is enough

⇒ Output:

→ ***...* 30 times (vertically)

→ =
=
=
=
=
.
.
.
30 times

→ +
+
+
.
.
.
45 times

Limitations:

→ You can miss only trailing arguments

Ex: mepchar(50); → // Error

→ once we provide a default value for a parameter, all subsequent parameters must also have a default value.

Ex: void do(int, int=10, int); ✗

→ If we are defining the default arg in the func. definition instead of func. prototype, then the func. must be defined before the func. call.

Constructor: → a special member function

→ name same as class → no return type → always declare under public section

→ no need to explicitly call it, the object will itself call it.

```
class line {
    double l;
public:
    void setl(double l)
    {
        l=l1;
    }
    double getl() {
        return(l);
    }
```

```
Line () {
    cout << " object is being created ";
}
};
int main(){
    Line S;  → Line() is also executed.
    S.setl(100.12);
    cout << S.getl();
}
```

(or)

```
Line (double l1)
{ l = l1;
  cout << " object created ";
}
};
int main(){
  Line S1(100.12), S2(101.51), S3(500.15);
  cout << S1.getl() << S2.getl() << S3.getl();
}
```

(or)

```
Line(double l1, double h1):l(l1), h(h1){
    cout<<
}
```

```
classname (datatype v1, datatype vn):
    datamember(v1), datamember(v2){

}
```

```
class A{
  int x;
  float y;
  public;
    A()
    { x=5;
      y=10;
    }
}

int main(){
    A ob;
}
```

```
class A{
  int x;
  float y;
  public:
    A():x(5), y(10)
    {}
}
```

↳ automatic initialisation of data members
↳ implicitly called while creating object.

~~84~~

Destructor: → de allocates the space occupied by constructor

→ same characteristics as that of const.

→ But constructor → can be overloaded

destructor cannot. → no arg / ret type to destruc

~ Line:l().

↳ destructor symbol

```
class room {
    int l;
    int b;
    int h;
public:
    room (int l, int b, int h): l(l), b(b), h(h)
    {
        int vol = l*b*h;
        cout << vol;
    }

    int vol (int l, int b, int h){
        cout << (l*b*h);
    }

    void input() {
        cin >> l >> b >> h;
    }
```

~~Q1~~:

```
int main() {
    room R1(5, 6, 7), R2;
    ~~vol(7, 2, 8);~~
    ~~input();~~
    R2.input();
    R2.vol();
} ~Room;
```

Encapsulation, polymorphism, Abstraction, Inheritance, classes & objects

Preprocessor Directives:-

→ always start with '#'
→ cannot combine 2 PP in a single line
→ should be compiled before main program

⇒ # include : to include the library files

⇒ # define : to define symbolic constants

⇒ #if #elif #endif : For conditional compilation

⇒ #ifdef #ifndef : " " definitions of symbolic constants

⇒ #error : To print an implementation dependent message.

→ #pragma : For implement dependent action

Syntax:

#define (name of const) (value of const)

#define PI 3.1416

Macros: → used to define simple expressions

→ may | maynot be used with arguments

→ data type checking is not done here.

```
#define AREA(x,y) ((x)*(y))

int main(){
    int a=3; b=4;
    cout << AREA(a+3; b+4);

}
```

#undef → undefine / remove the definitions related to the macros.

```
#define PI 3.1415
printf("Area", PI*r*r, "%d");
#undef PI
#define PI 3.14
```

```
#define num 7
int S1 = num * num;
#undef num
#define num 8
int S2 = num * num;
int main(){
    printf(" val of sq w/ 1st value of num:%d",
                                        S1);
    printf("value of sq w/ 2nd value of num:%d",
                                        S2);
}
```

```
#define studId 101
#undef studId
int main(){
#ifdef studId  ⟶ checks if its defined
                  (Here, no → false)
    printf(" stu exists %d ", studId );

#else
    printf("doesnot exists");

#endif
}
```

~~#if~~

```
#if !define (MAX)      → if its not defined
                              ↓
    #define MAX 500          #ifndef
#endif
```

```
#define FAC 5
int m() {
  #if FAC < 5
      Pf (" less than 5");

  # elif FAC = 5
      Pf(" = 5");

  # else
      Pf(" > 5");
  #endif
}
```

#if 0 .... #endif → comment statement (will not be considered)

#if 0 ~~#if~~ 1 will be considered for compilation

```
#if 0
    int x = 100;
    Pf ("%d", x);
```
} → commented

```
# endif
    (" I am here");
```

```
#define STRCAT(a,b) a##b
int m(){                                  ⇒ TWOTHREE
    STRCAT(TWO, THREE);
}
```

```
#define WELCOME(arg) printf("welcome", #arg);
int m(){
    WELCOME(SASTRA);   ⇒ Welcome SASTRA
}
```

```
#define MIN(x,y) (x<y)? x : y
```

```
#if 0
time(int h, int m, int s):
hr(h), min(m), sec(s)
#endif
```

Obj as Function arg:

```
class time
{   int hr, min, sec;
  Public:
    time(int h, int m, int s){
        hr = h;
        min = m;
        sec = s;
    }
        (or)
    time(int h, int m, int s): hr(h), min(m), sec(s){}

    time()
    {  hr=10; min = 8; sec=60; // default constructor
    }

void addtime(time t1, time t2)
{   int hours, mins, seco;
    hours= t1.hr + t2.hr;
    mins = t1.min + t2.min;
    seco = t1.sec + t2.sec;
    if(sec >=60){
      mins= mins+ seco/60;
      seco= seco%60;
```

eft.

```
if (mins >=60){
    hours = hours + min/60;
    mins = mins %.60
}

int m(){
    time t2, t1(3,49,59), t3;
    t3.addtime(t1, t2);

    t3.disp();

}
```

(or)

Return Object:

```
time addtime(time t1, time t2){
    time t3;
    t3.hr = t1.hr + t2.hr;
    t3.min = t1.min + t2.min;
    t3.sec = t1.sec + t2.sec;
    if (t3.sec >=60){
        t3.min += t3.sec/60;
        t3.sec = t3.sec %.60;
    }
    if(    ){
}
return t3;
```

int m() {

- - - , tH;

tH = t3.addtime (t1, t2);

tH.disp();

}

(or)

no need time t3; inside time addtime,

return * this

Inside int main (){

~~cout~~ ~~<<~~ ~~*~~ ~~this~~

t3.addtime(t1, t2);

~~cout << *this;~~ t3.disp();

invoking object

WAP to add 2 complex nos. and the object must be passed by reference and result must be passed by value.

class complex

{ int real, img;

public:

void getin(){

```
    cin>> real >> img;
}
void putin(){
    cout << real <<img;
}
complex add (complex &);
  data
  type
};
```

c1.add.real

```
Complex Complex :: add(complex &g1, complex &c2)
{
    complex c3;
    c3.real = real + g1.real;
    c3.img = img + g1.img;
    return c3;
}
```

×

real = real + ...

img = img + ...

return *this;

Complex

```
int m(){
    complex c1, c2, g3;
    c1.getin();
    c2.getin();
    g3=c1.add (c2);
    c1.putin();
    c2.putin();
    g3.putin();
}
```

c1.add(c2)

cout << *this;

14

```
Class dist
{
    int feet;
    float inch;
Public:
    Dist(): feet(0), inch(0.0) // default const.
    {}
    Dist(int f, float in): feet(f), inch(in){}

    void getdist()
    {
        cin >> feet >> inch;
    }

    void add(dist d1, dist d2)
    {   int tot-feet;
        float tot-inch;

        tot-feet = d1.feet + d2-feet;
        tot_inch = d1.inch + d2.inch;
        if (tot-inch >= 12){
          tot-feet = tot-inch/12 + tot-feet;
          tot_inch = tot-inch % 12;
        }
    cout << tot-feet << tot-inch; * x
}
```

* void disp(){cout << feet << inch

int main(){ → uses default constructor

Dist d1, d2, d3; feet = 0; inch = 0.0

(or)

Dist d1(6, 4), d2(10, 2), d3(10, 4);

↳ parameterised

∴ Dist d1; d3 → feet = 0; inch = 0.0

d1.getdist(); → feet = inp, inch = inp

Dist d2(5, 6.2);

d3.add(d1, d2);

→* d1.displ()
d2.displ()
d3.displ()

return 0;

}

---

return object from function:

```
dist add (dist d1, dist d2)
{
    dist d3;
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.~~feet~~inch + d2.inch;
    if(d3.inch >=12){
      d3.feet = d3.feet + d3.inch/12;
      d3.inch = d3.inch %.12;
    }
} return d3;
```

```
int m(){
    Dist d1, d2; d3;
    d1. getdist();
    d2. getdist();
    add
    cout << d3.add (d1, d2);
    return 0;
}
```

(or)

```
Dist adddist (dist d2)
{
    dist d3;
    d3. inch = inch + d2. inch;
    d3. feet = feet + d2. feet;
    if [d3. inch >=12){
      d3. feet = d3. feet + d3. inch /12;
      d3. inch = d3. inch % 12;
    }
    return d3;
}
int m(){
    dist d1, d4;
    d1. getdist();
```

eft:

```
d4.getdist();
d1.adddist(d4);
d3.disp();
        (or)
d5 = d1.adddist(d4);
d5.disp();
}
```

One arg const :-

no arg const → def
mult " " ⇒ parameterised

```
int m(){
  dist d1(6, 5.2)
  dist d2 = d1;   ⇒ default copy constructor
  dist d3(d1);    ⇒ member by member initialisation.
  d1.disp();
  d2.disp();
  d3.disp();
}
```

if in class:

```
dist (dist &a){
  feet = a.feet;
  inch = a.inch;
  cout << "copyconstructor";
}
```

Static Variables:

```
void f(){
    static int c=0; //initialised only once
           int d=0;
    cout<<"static var"<<c++<<endl;
    cout<<"Ordinary var"<<d++<<endl;
}
int main(){
    for(i=0; i<5; i++){
        f();
    }
    return 0;
}
```

OUTPUT:

c	d
1	1
2	1
3	1
4	1
5	1

⇒ For every function call, the variable d is initialised to 0 but c retains its old value as it can be initialised only once.

If we make it static the initialisation is done only once.

[struct data mem → default public
  class " " → " private]

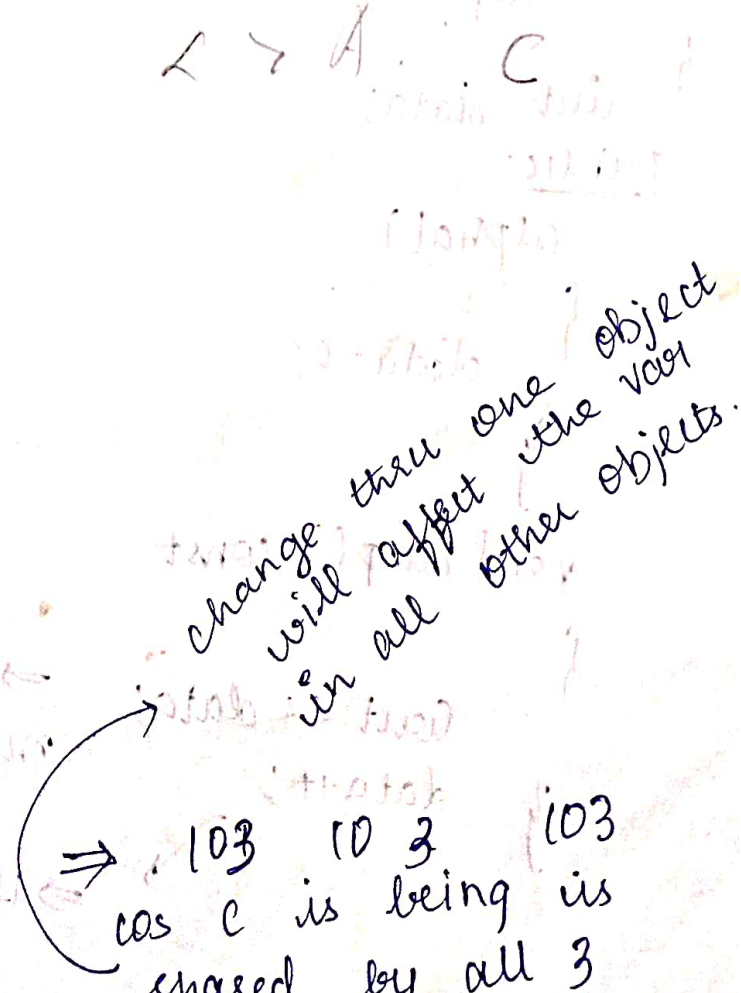
```
class foo {
    static int data;
public:
    void f();
};
```

→ all objects of this class share this data member

only declaration can be done inside class

we have to initialise static data member outside the class ⇒ data type <name> :: <n> = 10
class var name

```
class A {
    static int c;
public:
    A()
    {
        c++;
    }
    void disp() {
        cout << c;
    }
};
int A :: c = 100;
int main() {
    A o1, o2, o3;
    o1.disp();
    o2.disp();
    o3.disp();
}
```



(X) 2m:

- If data item is delared as static, only one such item is created for entire class.
- It is visible only within class, but its lifetime is entire program.
- A static member is used to share info among objects.

Constant And class:

const:

① → const object
② → const member function
③ → const " " argument

②

```
class alpha
{
    int data;
public:
    alpha()
    {
        data = 0;
    }
    void disp() const
    {
        cout << data;
        data++;
    }
};
```

→ It doesn't change/modify the value of data members

→ If tried, it will raise an error

(3) void add (const dist d1, const dist d2) {

```
int tot_feet, tot_inch;
tot_feet = d1.feet + d2.feet;
tot_inch = d1.inch + d2.inch;
if ( tot_inch >= 12) {
        inch -= 12;
        feet += 1;
    }
```

here we are not changing the value of the data member of the const. objects

(if d1.inch++ → error)

```
class demo
{   int a;
public:
    void set (int x) const
    {   a = x;
    }
    void sum (const demo o1, const demo o2)
    {   a = o1.a + o2.a;
    }
```

→ if const is used then a = x will produce error.

```
void print() const
{   cout << a;
}
};
int m(){
  demo d1, d2; d3;
  d1.set(10);
  d2.set(20);
  d3.sum(d1, d2);
  d3.print();
}
```

① We can make object as const as well, but we can call only a const member function with that.

```
class dist
{   int feet;
    float inch;
    public:
      dist(int f, float i)
      {  feet = f;
         inch = i;
      }
```

```
void getinput()
  {  cin>>inch>>feet;
  }
void disp() const
  {  cout<<feet<<inch;
  }
};
int m(){
   const dist foo (5, 6.2);
   foo.getinput(); // Error
   foo.disp(); // Okay!!
```

(b) void add (const dist d1, const dist d2){

const dist foo(

```
int m(){
  Counter c1, c2, c3;
  cout << c1.getcount(); //0
       << c2.    "   (); //0
       << c3     "   (); //0

  ++c1;
  ++c2;
  ++c3;
```

calls the operator overloading member function

c1.++(); (operator) c2.++(); (operator) c3.++(); (operator)

```
  cout << c1.getcount(); //1
       << c2.    "   (); //1
       << c3.    "   (); //1
}
```

return object:

```
Counter operator ++()
⇒ {  ++c;
     Counter t;
     t.c = c;
  }

int m(){
counter c1, c2;
cout << c1.getcount();
++c1; << c2 "   ();
c2 = ++c1;   (since c1 stored in c1 itself)
cout << c2.getcount(); cout << c1.getcount(); }
```

Nameless temporary object:

```
class counter
{
    int c;
public:
    counter(): c(0)  → default cons
    { }

    counter(int d): c(d) → one arg const
    { }

    int getcount() { return c; }

    counter operator ++() {
        ++c;
        return (counter(c));
    }
};
```

constructor is invoked (∵ object is created) but it doesn't have a name.

```
int main()
{
    counter c1;
    ++c1;
    c1.getcount();
    c2 = c1++;
};
```

c1.getcount ⇒ 1
c2.getcount ⇒ 2

Postfix:

```
counter operator ++(int)
{
    return counter(c++);
}
```

postfix operator
↓
increment
↓
a

Difference b/w prefix & postfix notation:

- Int in paranthesis is the only diff
- The 'int' is not really an argument.
- It doesn't mean integer. Its simply a signal to compiler to create postfix version of operator.

Decreament overloading:

```
class counter{
    int c;
    public:
        counter() : c(0) { }

        counter(int d) : c(d)
        { }
        int getcount()
        { return c;
        }
        counter operator -- ()
        {   --c;   → prefix
            return(counter(c));
        }
        counter operator --(int)
        {   c--;  → postfix
            return(counter(c));
};
```

```
int m(){
    counter c1, c2;
    ++c1;
    c1.getcount(); ⇒ -1
    c2 = c1++;     ⇒ -2
    c2.getcount();
};
```

```
class faculty
{
    char name[20];
    char dept[20];
    public:
    void getdata()
    {
        cin.getline(name, 20);
        cin.getline(dept, 20);
    }
    void dis(){
        cout << "name" << setw(20) << "dept" << el
```

#Allocate memory for obj @ runtime

```
int main(){
  faculty f[5]; // array of obj
  for(int i=0; i<5; i++){
        f[i].getdata();
  }
  for(int j=0; j<5; j++){
        f[j].getdata();
  }
```

(or)

```
int main(){
  faculty *f[10];
  char ch='y';
  int n=0;
  while(n<10)
  {  f[n] = new faculty();
     f[n]->getdata();
     cout<<"continue?(Y/n)";
     cin>>ch;
     if(ch=='N'){break};
     n++;
  }
```

cout<<setiosflags(ios::left)<<setw(10)<<

```
for (int i=0; i<no; i++){
    f[i]->display();
}
delete [] f;
};
```

```
class fraction
{
    int num; 4
    int deno; 8
    int fac=1;
    void simplify();
    int GCD(){
        cin>>num;
        cin>>deno;
        for(int i=0; i<num; i++){
            int div;
            div=num%i;
            if(div==0){
                if(deno%i==0){
                    fac=fac*i;
                }
            }
        }
        return fac;
    }
```

½ ??

(2)

```
int simplify(){
    int gcd;
    gcd = GCD();
    int a_num = num/gcd;
    int a_deno = deno/gcd;
    cout << a_num << "/" << a_deno;
}
public:
    void getdata(){
        cin >> num;
        cin >> deno;
    }
    void disdata(){
        simplify();
    }
};
int main(){
    fraction f;
    f.getdata();
    f.disdata();
    return 0;
}
```

```
GCD(int a, int b)
{ if(b==0)
    return a;
  else
    return gcd(b, a%b);
}
```

Binary Operator Overloading:

```
class complex
{
    int a, b;
    public:
     complex (int x, int y)
      {
          a = x;
          b = y;
      }
     complex operator + (complex);
     void disp(){
       cout << a << "+i" << b;
     }
};
complex complex :: operator + (complex c)
{
     complex t;
      t.a = a + c.a;
      t.b = b + c.b;
      return t;
}
int main(){
    complex c1(3,2), c2(2,1), c3;
     c3 = c1 + c2;
     c3.disp();
};
```

→ accept object as argument

c3 = c1 + c2; → c1.operator+(c2)

Using Nameless Object and constant member function:

```
class complex
{ int a, b;
  public:
    complex (int x, int y){
        a = x;
        b = y;
    }
    complex operator + (complex);
    void disp() const {
        cout << a << "+i" << b;
    }
};
complex complex :: operator + (complex c)
{
    a = a + c.a;
    b = b + c.b;
    return (complex (a, b));
}
int main(){
    complex c1(3,2), c2(2,1), c3;
    c3 = c1 + c2;
    c3.disp();
}
```

WAP to overload operator multiplication (*) in the class complex.

```
class complex{
    int a, b;
    public:
      complex(int x, int y){
          a=x;
          b=y;
      }
      void disp(){
        cout << a << "+i" << b;
      }
      complex operator *(complex c2){
            complex t;
            t.a = a*c2.a - b*c2.b;
            t.b = a*c2.b + b*c2.a;
            return t;
      }
int  main(){
      complex c1(3,2), c2(3,1), c3;
        c3 = c1*c2;  => c1.operator*(c2)
        c3.disp();
};
```

WAP to overload operator < .

```
class lesser {
int a;
public:
  lesser (int x, int y) : a(x); { };
  void disp() { cout << a; };
  lesser operator < (lesser p, lesser q){
      if (p.a < q.a){
          return p;}
      else {
          return q;}
  }
};

int m(){
  lesser l1(10), l2(5), l3;
  l3 = l1 < l2;
  l3.disp();
};
```

Operators that cannot be overloaded:

① :: ② sizeof ③ ?: ④ pointer to member function op

```
class dist {
    int feet;
    float inch;
    public:
      dist(): feet(0), inch(0.0) { }
      dist(int f, float i) : feet(f), inch(i) { }
      void get() {
        cout << "Enter feet & inch";
        cin >> feet >> inch;
      }
      bool operator < (Dist) const;
};

bool Dist :: operator < (dist d) const {
    float bf1 = feet + inch/12;
    float bf2 = d.feet + d.inch/12;
    if (bf1 < bf2)
        return time;
    else
        return false;
}
```

```
int m(){
  Dis d1 (5, 2), d2;
  d2.get();
  if (d1 < d2)
     cout << " dist1 is greater";
  else
     cout << "dist 2 is greater";
}

class mult {
  int num;
  int deno;
  public:
     mult(): num(0), deno(0){}
     mult(int n, int d) : num(n), deno(d){}
     void get(){
        cin >> num >> deno;
     }
     mult operator * (mult m1) {
        mult m3;
        m3.num = m1.num * num;
        m3.deno = m1.deno * deno;
        return m3;
     }
     void put(){
     cout << num << "/" << deno;
     }
};
```

eft:

```
int m(){
    mult m1(5, 4); m2; m3;
    m2.get();
    read;
m3 = m1 * m2;
    m3.put();
    return 0;
}

class assign{
    int a;
    public:
      assign(): a(0){}
      assign(int d) : a(d){}
      assign operator = (assign a1){
          assign a2.
          a2.a = a1.a;
          return a2;
      }
      void put(){
          cout<<a;
      }
};
```

```
int main(){
    assignal( 5 ); a2;
     a2 = a1;          => a2.assign(a1)
     a2. put ();
     retuen 0;
}

class notop(){
int a;
bool b;
public:
    notop(): a(0), b(True) {}
    void get(){
        cin>> a >> b;
    }
    void operator ! (){
        a = !a;
        b = !a;
    }
    void disp(){ cout << a << b; }
};
```

```
int main(){
    Nolop obj;
    !obj;
    obj.displ();
}   return 0;
```

```
#include<process.h>
#define LIMIT 100

class safearr
{   int a[LIMIT];
    public:
     int &operator[](int n){
        if(n<0||n>LIMIT)
          { cout<<"Out of Bound".
            exit(1);}
        return(a[n]);}
};
int m(){
    safearr a;
    for(int j=0; j<LIMIT; j++){
        a[j] = j*10;
```

a.operator[](j)

```
    }
```

```
for (int j=0; j< LIMIT ; j++) {
        int t = a[j];
        cout << t;
  }
return 0;
}
```

~~#include <sto~~

string concatenation:

```
string operator + (string ss)
{
    string t;
    if (strlen(str) + strlen(ss.str) < 80)
    {
        strcpy(t.str, str);
        strcat(t.str, ss.str);
    }
    else {
        cout << "Overflow";
    }
    return t;
};
```

```
#include <string.h>

class stringcmp {
        char str[100];
        public:
            string()
            { str[0] = '\0'; }

            string (char s[])

             { strcpy (str, s); }

            void disp(){
                cout << str; }

            int operator == (string ss)
            {
                if ( ! strcmp(str, ss.str))
                        return 1;

                else

                        return 0;

            }
};
int m(){
  string s1 ("CSBS"), s2("Sadhu"), s3;
  if (s1 == s2)
     cout << "strings are equal";

  else
     cout << "Nope";
  return 0;
```

TYPE CONVERSION:-

- for user defined data types

converting any built-in data type to user defined data type → we use constructor

4 types of conversion:-

① Basic to Basic → Implicit
                  → Explicit

② class type to Basic → operator to be conv ( )

③ Basic to class type → constructor (1 arg)
                                    ↓
                                  min arg basic data type

④ class type to class type

② class type to Basic:

→ constructor does not support this type of conversion

→ C++ allows us to define an overloaded casting operator, that could be used to convert a class type to basic type. It is also known as conversion function.

→ overloaded casting operator actually overloads the built in data types

Rules to be followed for operator fn:

* It must be a class member
* It must not specify return type
* It must not have any arguments.

Basic To class Type:

```
Class time
{   int hr, min;
    public:
        time() { }
        time (int dur)
            { hr = dur/60;
              min = dur % 60;
            }
        void disp() {
            cout << hr << min;
        }
};
int main() {
    int t = 80;
    time ob(t); // the int t gets converted to
                | hr 20mins. → class data type
    ob.disp();
}
```

Conversion constructor

int value is converted into a data member type

operator int() {
    return (hr * 60 + min);
}

Class type to Basic:

operator <typename> ( ).

↓ any basic data type

operator int ( ) } as a member function only

{ return (hr * 60 + min);

}

int m( ){
int t;
time ob(120); ⇒ (2 hrs 0 mins)
t = ob; ⟶ ob. operator int ( );
}

will return (2 * 60 + 00)

float / int i; 120 → int

value (float f){
i = f;
}

operator float( )

{ return (float (dm of the class))

}

int m( ){
float b;
float a = 5.0;
value v(a); v. show ( );

b = v;

Inheritance:

⇒ creating new class (derived class) from existing class (base class)

⇒ we can reuse the existing variables and ensures extensibility. ⇒ you can derive certain features and enhance them by adding extra features instead of creating everything from scratch.

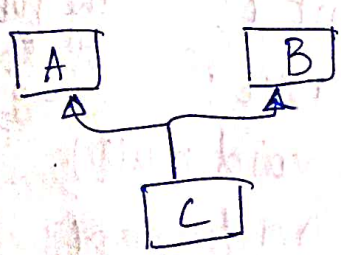
TYPES:

① Single Inheritance → 1 base class and 1 derived class

A → base class | superclass | parent class

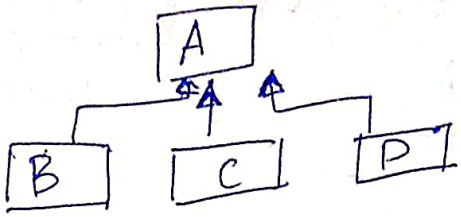
B → derived class | subclass | child class

② Multiple Inheritance - more than one parent

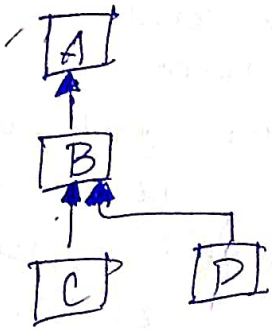


③ Multilevel Inheritance -

④ Heirarchical Inheritance:



⑤ Hybrid Inheritance: Comb. of any 2 types.



DEFINING DERIVED CLASS:

class <derived class name> :: visibility <baseclass name>

⇓

default ← Private | public | protected
visibility

Example:

```
class B
{
    int a;
public:
    int b;
    void getab();
    void geta();
    void show();
};
```

```
class D : private B
{
    int c;
public:
    void mult();
    void disp();
};
```

→ meaning: the public mem fn. of class B becomes private fn of this class

```
class D :
int c;
int b;
void getab();
void geta();
void show();
public:
   void mul();
   void disp();
```

⇒ The data members & member functions that declared under the private section of derived class are inaccessible to the object of derived class.

⇒ Public member of derived class can be accessed by its own object using • operator

```
class B {
  int a;
  public:
    int b;
    void getab();
    void geta();
    void show();
};
```

```
class D : private B
{   int c;
    public:
      void mul();
      void disp();
};
```

```
void B :: getab()
{ cin >> a >> b; }

int B :: geta()
{ return a; }

void B :: show()
{ cout << a << b; }

void D :: mull()
{ getab();
  c = b * geta();
}

void D :: disp(){
  show();
  cout << c;
}
```

Using private:

```
class B
{ int a;
  public:
    int b;
    void getab();
    int geta();
    void show();
};
```

```
class D: public B
{ int c;
  public:
    void mull();
    void disp();
};
```

Class to class type: constructor and casting operator:-

constructor : left object class


```
class product{
    int p;
  public:
    product(){}
    product(int d)
      { p=d; }
    void show()
    { return p}
};
```

```
class item{
    int i;
  public:
    item(){}
    item(product p)
    { i= p.show()
    }
    void put(){
      cout<<i;
}};
```

```
int m(){
product p2(5);
item i1 = p2;      item i1(p2);
 i1.put();
}
```

class B: public A {

SINGILE:

```
class shape {
    int x;
  public:
    void getx() {
        cin >> x;
    }
    int putx() {
        return x;
    }
};
```

```
class circle: public sha
    float area;
  public:
    void area()
    {
        area = 3.14 * putx
               * putx
    }
    void show()
    cout << area;
};
```

```
int m() {
    circle c;
    c.getx();
    c.area();
    c.show();
}
```

Overriding:

```
class B {
  public:
    void put() {
        cout << "base class";
    }
};
class D : public B {
  public:
    void put() { B::put();
        cout << "derived class";
    }
};
```

```
int main(){
  D d;
  d.put();  ⟶ "der class";
  d.B::put(); ⟶ "Base class";
```

HIERARCHICAL:


```
class Employee {
  int ID;
  char name[15];
  public
    void get(){
      cin>>ID;
      cin>>name;
    }
    void put(){
      cout<<ID;
      cout<<name;
    }
};
class Scientist: public Employee{
  int Sci_no;
  public:
    void get(){
      Employee::get();
      cin>>Sci_no; }
```

```
void put(){
    Employee :: put();
    cout<< Sci_ID;
}

int m(){
  Sci S;
}
```

Multilevel:


```
class Person{
    char name[15];
    int age;
    Public:
      void get()

      void put()
};

class Emp : Public Person{
    int ID;
    Public:
      void set_emp(){
        get()
```

Multiple Inheritance. Pg:

```
class N1{
int no1;
public;
    void get1(){
        cin>>no1;
    }
};
class N2 {
    int no2;
    public:
      void get2(){
        cin>> no2;
    }
};
class add : public N1, public N2 {
  Public:
    int addi {
      return no1 + no2;
    }
```

```
class A {
    int count;
    public:
        A() {
        cout << "def of A";
        }
        A(int i): count(i) {
            cout << "para of A";
        }
};
class B: public A {
    int count;
    public:
        B() {
        cout << "def of B";
        }
        B(int j): A(j) {
            cout << "para of B";
        }
};
int m() {
    B b;
```

FILES:

ofstream → for writing info into file
ifstream → " reading " from "
fstream → for both

For open:

open ("filename", mode)

modes:

ios::app : the output is sent to the file and appended to it.

ios::ate : opens the file for the output then moves the read and write control to file's end.

ios::in :

ios::out :

ios::trunc : BY opening, delete the content of the file.

It is possible to use 2 modes @ the same time using | (or operator)
↳ pipe symbol

ignore():

```
int main()
{
    char name[20];
    int age;
    cin >> age;
    cin.getline(name, 20);
    cout << age << name;
}
```

↳ removes input buffer.

cin.ignore()

```
#include <fstream>
int main()
{
  char text[30];
  fstream file;
  file.open("example.txt", ios::out|ios::in
  cout << "Enter the contents";
  cin.getline(text, sizeof text);
  file << text; } write
  file >> text; } read
  cout << text;
  file.close();

  return 0;
}
```

Class person has 2 data members name & age

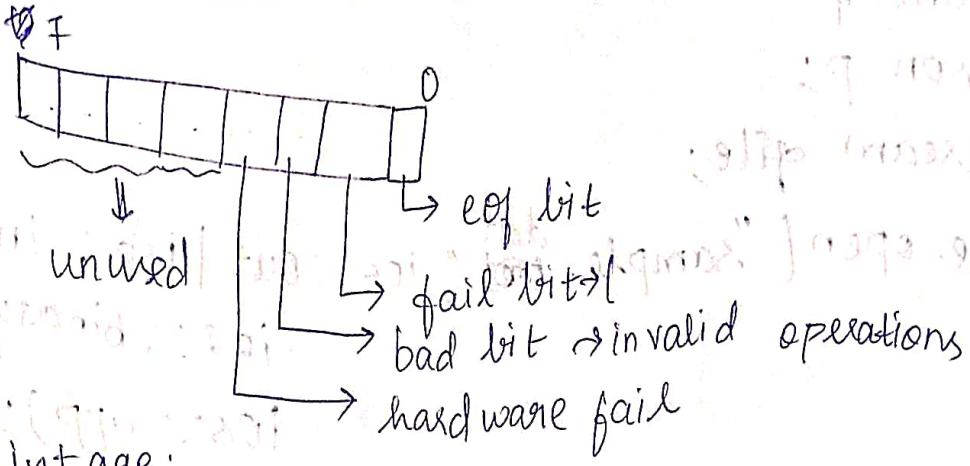
```
class person
{
  char name[30];
  int age;
  public:
    void getdata(){
      cin >> name;
      cin.ignore();
      cin >> age; }

    void disp(){
      cout << name << age; }
```

```
int main(){
    person p;
    fstream file;
    file.open("sample.dat", ios::out | ios::in |
                                ios::binary |
                                ios::app);

    do
    { p.getdata();
      file.write((char*)&p, sizeof(person));
      cout << "enter for another person";
      cin >> ch;
    }while(ch==y);

    file.seekg(0);          → file ptr moves to the beginning
    file.read((char*)&p, sizeof(person));
    while(!file.eof()){
          p.putdisp();
          file.read((char*)&p, sizeof(person));

    }
    return 0;

}
```

Casting operator:

```
file.write(reinterpret_cast<char>(&ob), sizeof
                                    (person));
```

Stream Error:




```
int age;
while(1)
{
    cout << "Enter age";
    cin >> age;
    if(cin.good())
    {   cin.ignore();
        break;
    }
    cout << "Enter age in no";
}
cout << "age is: " << age;
```

→ will test if input is correct (ie) int type
if not → false

→ flush/refresh the stream

→ and then loop (get input again)

FILE POINTERS AND THEIR MANIPULATIONS:

File pointer:
- Read ptr
- write ptr

HELLO
↑
Read ptr

ios::in

HELLO
↑
append

ios::app

ios::out

↑ write ptr

• seekg() and seekp():

→ Move the head pointer and put pointer to the specific location.

→ seekg (offset , refpos)

↳ ios::beg
↳ ios::cur
↳ ios::end

Ex:

file.seekg (10, ios::beg);

↳ move 10 Bytes from current position of the file → backward (-ve offset)

file.seekg (0, ios::end);

↳ move to eof

Example:

```
class person
{   char name[30];
    int age;
    public:
        void getdata(){
            cin>>name;
            cin>>age;
        }
        void disp(){
            cout<<name;
            cout<<age;
        }
};
int main(){
    person p;
    filestream file;
```

```
file.open("sample.txt", ios::out | ios::in | ios::binary |
                                              ios::app);
char ch[1];
do
  {
  p.getdata();
  file.write((char*) &p, sizeof(Person));
  cout << "another person";
  cin >> ch;
  } while(ch == 'y');
file.seekg(0, ios::end);
int pos = file.tellg();
int n = pos / sizeof(p);
cout << n;
```

to know the no. of records (n)

```
class complex
{
  int real, img;
  public:
    complex(int r=0; int i=0){
        real = r;
        img = i;
    }
  friend ostream& operator << (ostream & out,
                               const complex &
friend istream& operator >> (istream &in, complex
                               &c);
```

```
ostream & operator << (ostream & out, complex &c)
{
    out << c.real;
    cout << "i";
    out << c.img;
    return out;
}

istream & operator >> (istream & in, complex &c)
{
    in >> c.real;
    in >> c.img;
    return in;
}

int main(){
  complex c1;
  cin>> c1;  ⇒ operator >> (cin, c1)
  cout << c1; ⇒ operator << (cout, c1)
}
```

```
friend ostream& operator << (ostream &out,
                             complex &c)
{
    out << c.real;
    cout << "i";
    out << c.img;
    return out;
}
```

~~A~~

UNIT-A: OOAD: Object oriented analysis and de

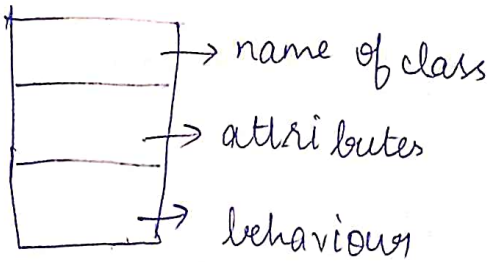
Software ~~Analysis~~ Development:

1. ~~Requirement~~ ~~Analysis~~
2.

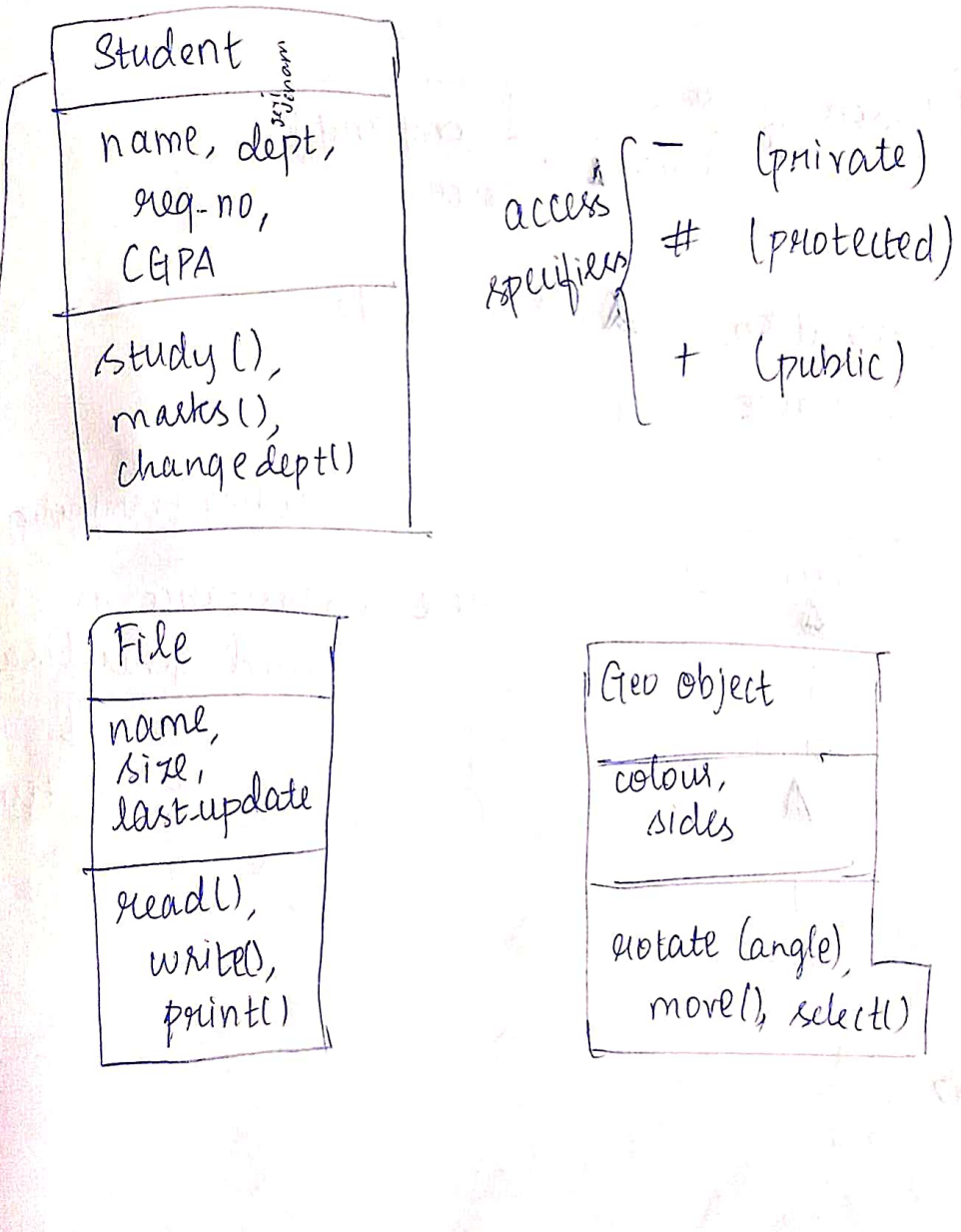
1. System ~~consumption~~ conception
2. Analysis
3. System design
4. class design
5. implementations
6. Testing
7. Training
8. Deployment ~~and~~ ~~maintenance~~
9. maintenance

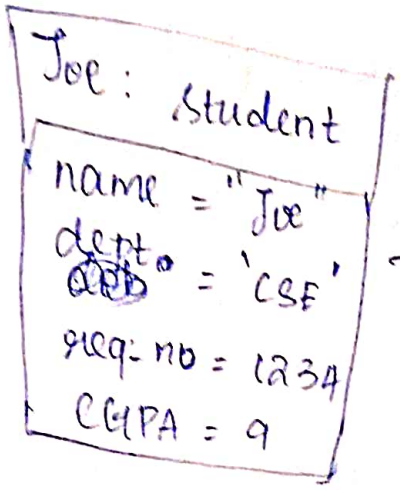
1. Gathering the requirements
   ↓
   tentative
2. Deeply understand the requirements using tools → what needs to be done
3. A high level strategy of creating diagrams, models and figures of designing policies
4. Algorithms devising
5. class design → Code
6. Code Testing
7. model training
8. Deployment → checking compatibility maintanance
9. ~~maintain~~ resolve future issues

class design:

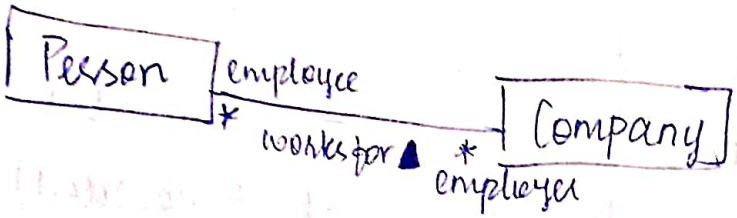


ex:



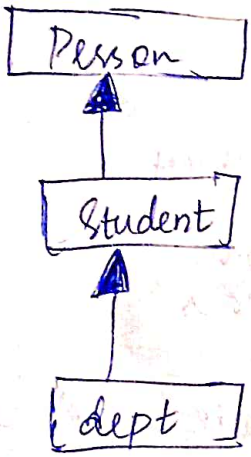


→ object diagram



Generalisation: → no shade

a) Inheritance

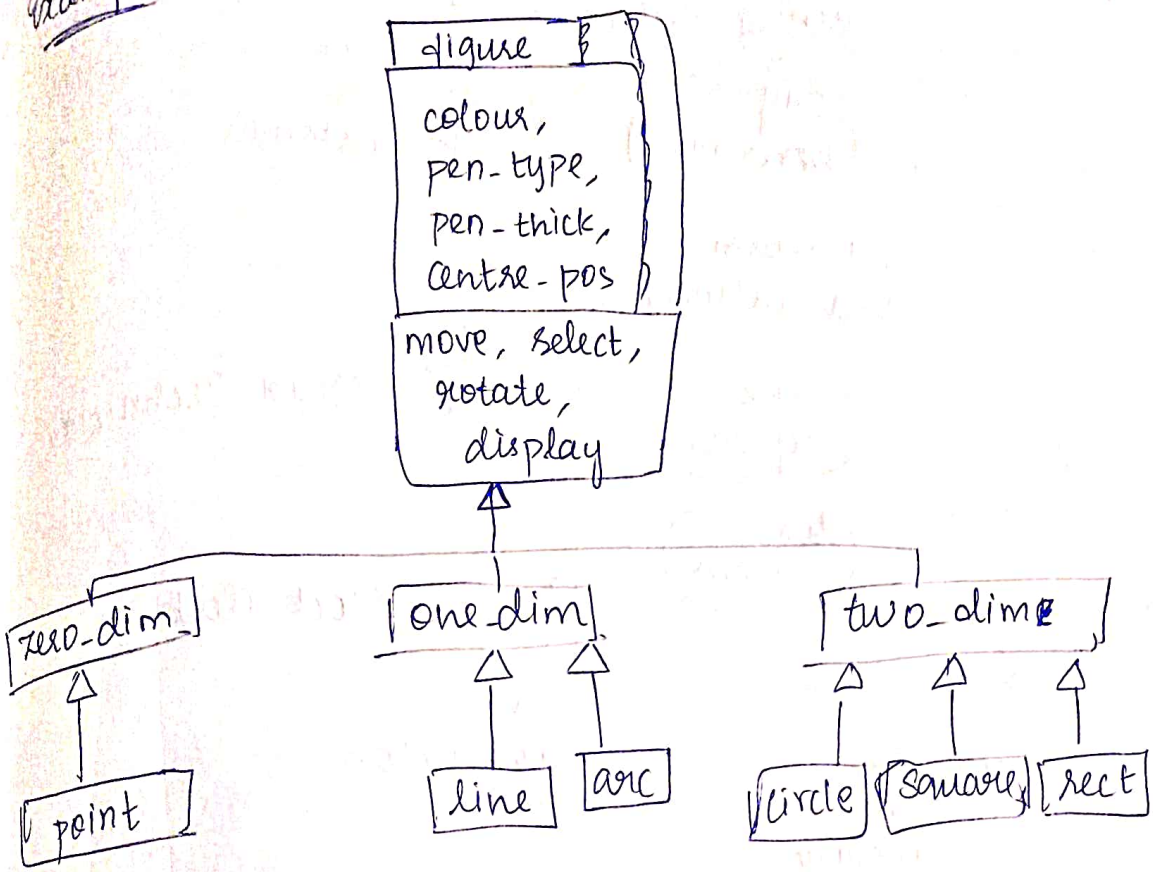


Coding terminology

Here inheritance is called generalisation

UML terminology

Example:



multiple inheritance called deligations

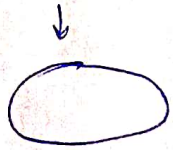
Building Blocks Of UML:

⇒ Things

⇒ Relationship

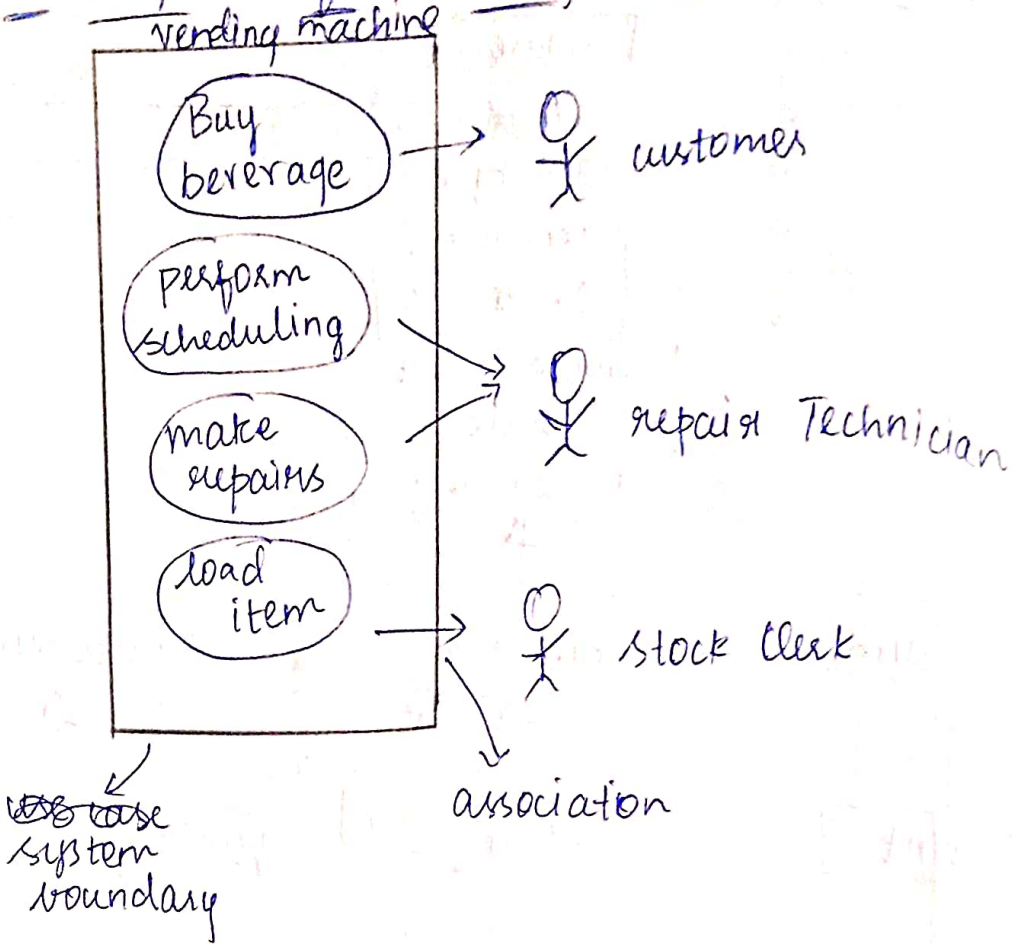
⇒ Diagram

Use case: represents set of actions performed by the system to achieve a goal.

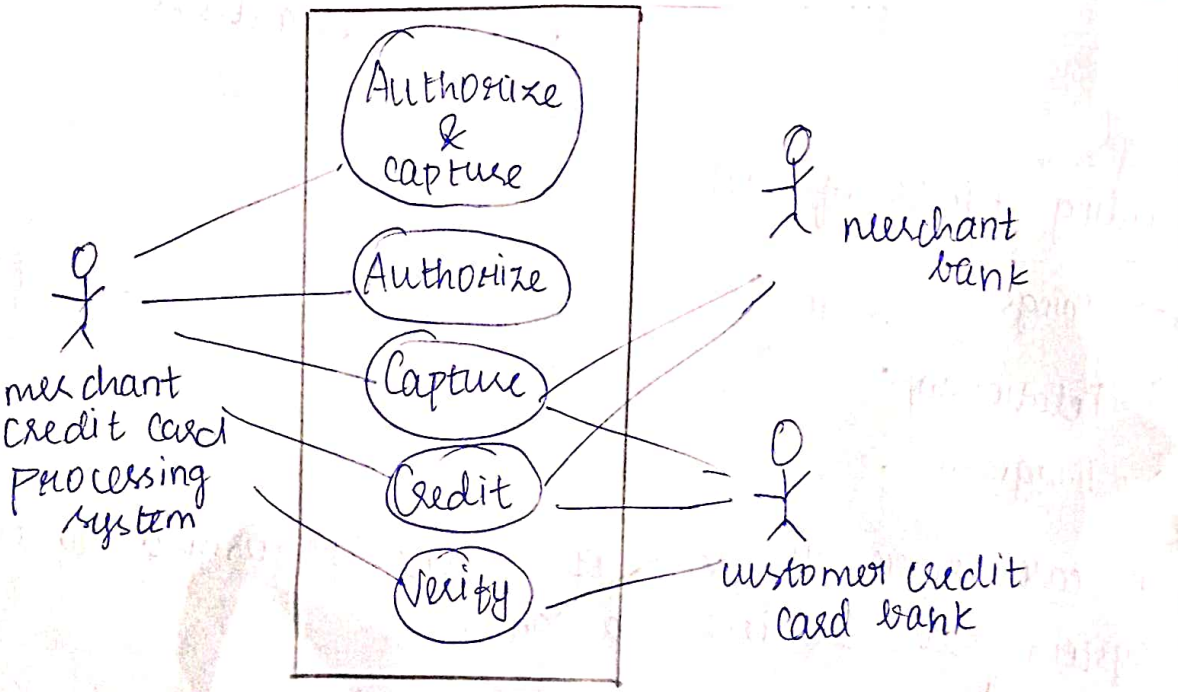


UDA: External user of the system

Use Case diagram for Vending machine:



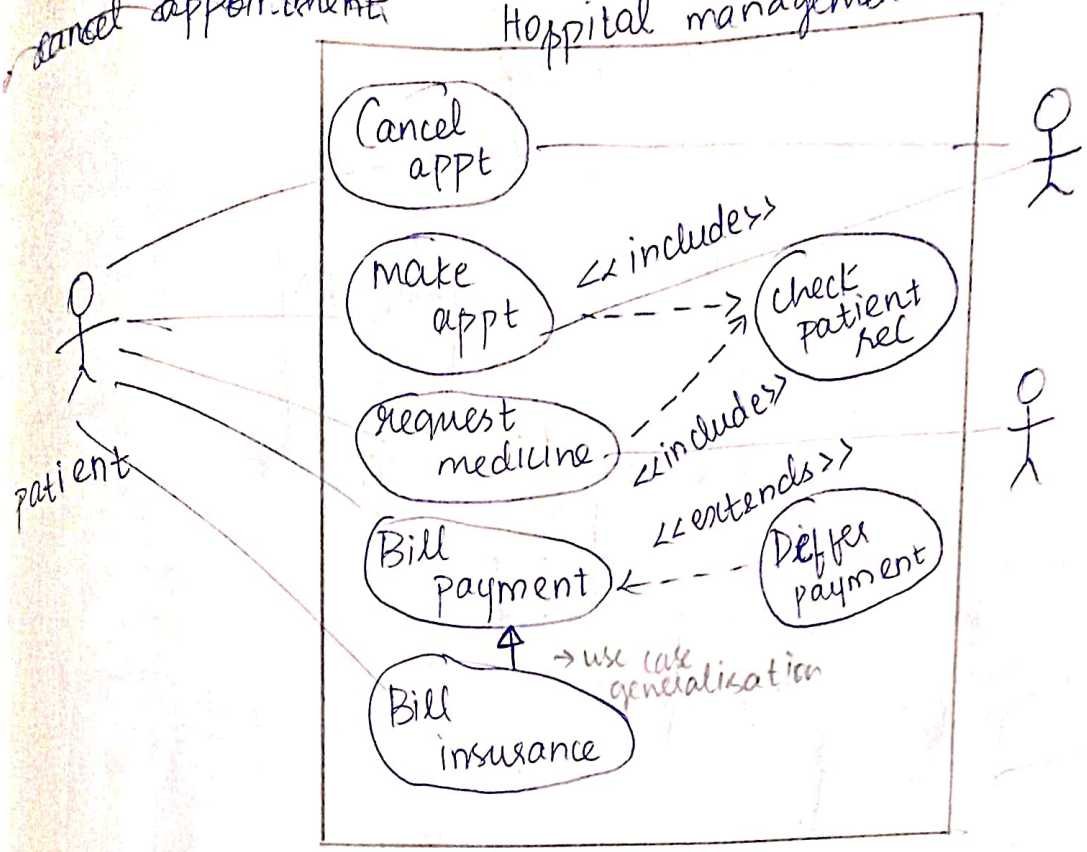
Use case diagram for credit card payment gate



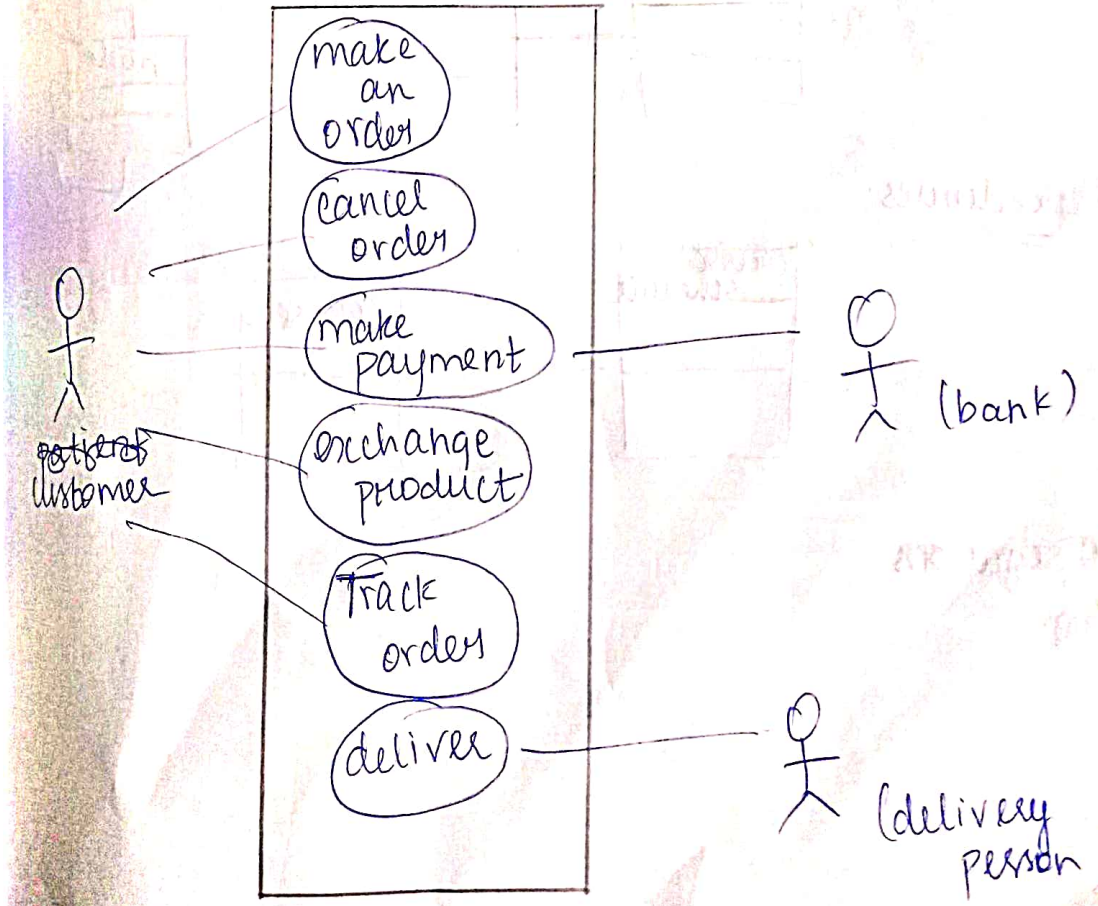
functionality

cancel appointment

Hospital management

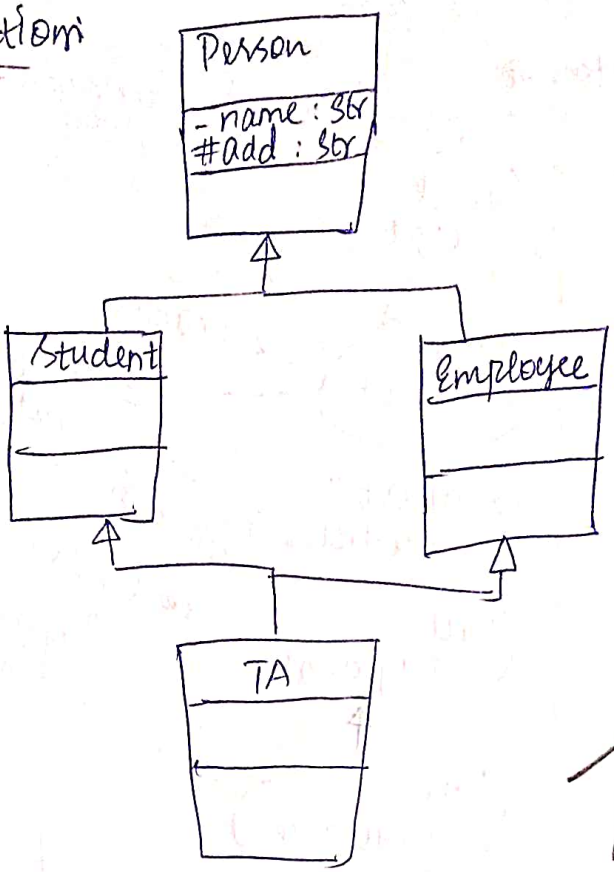


Online shopping

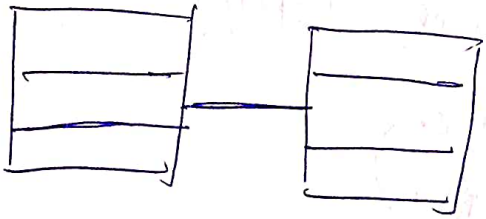


3 types of relations:

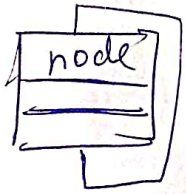
Generalization:



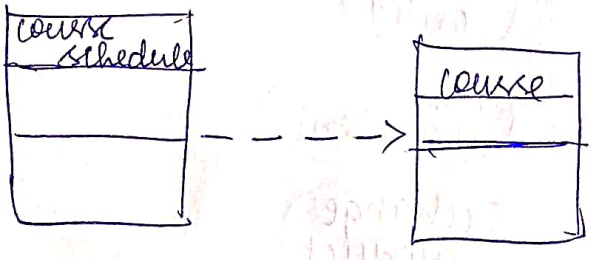
Association:



self association

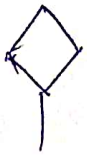


Dependencies:



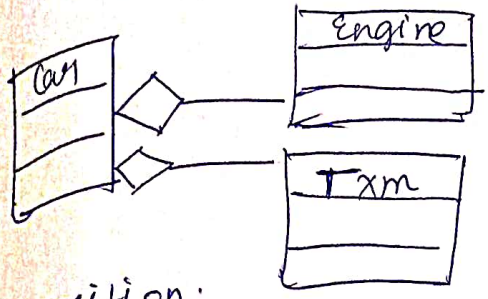
Aggregations

Aggregation: (2m)



whole part relationship

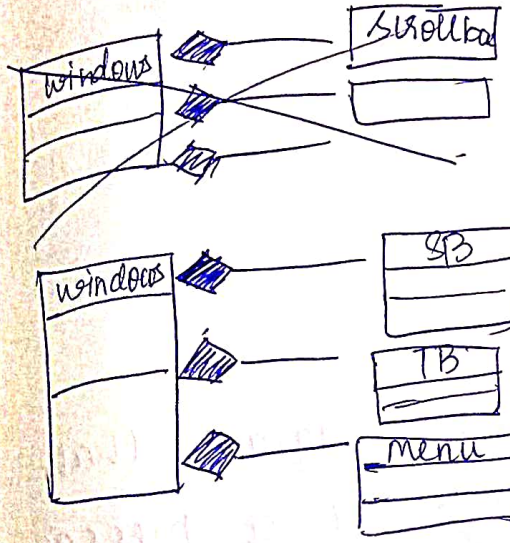
⇒ Independent lifetime



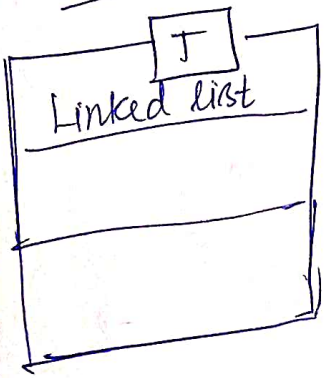
composition:

⇒ dependent lifetime

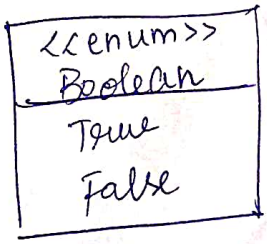
(SB, TB, menu's life depends on window class)



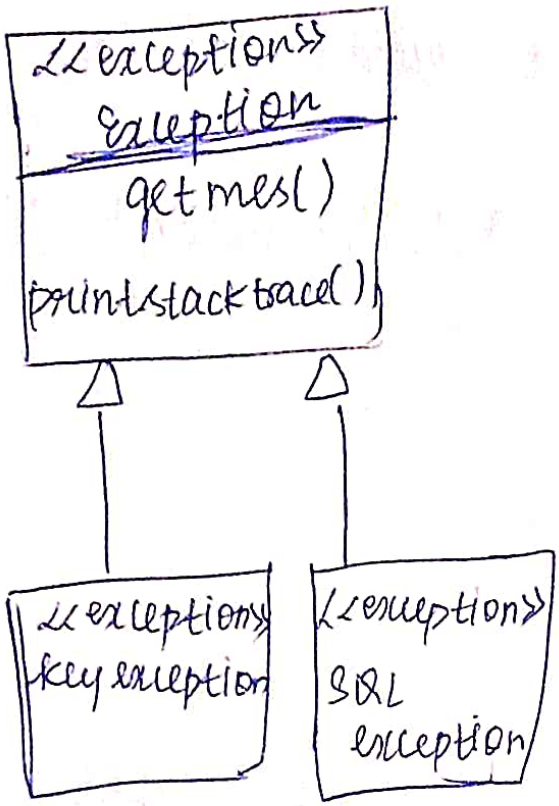
Template class:



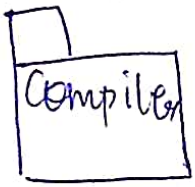
enumerations:



Exception:

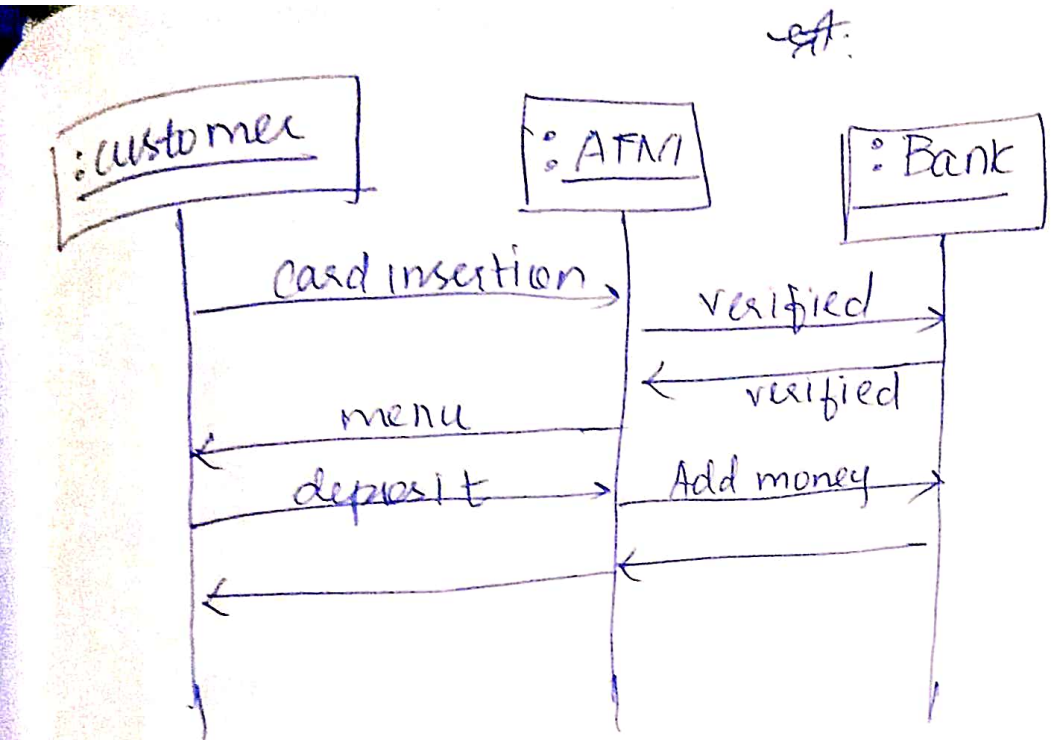


Package:



Sequence diagram:- • Interactions diagram that emphasises the time ~~modelling~~ Ordering of message.

- It shows a set of objects and the message sent and recieved by the objects.


:Customer
:ATM
:Bank
Card insertion
verified
verified
menu
deposit
Add money

class to class: using casting operator

```
class dollar {
    int dol;
  public:
    dollar() {
      dol = 0;
    }
    dollar(int d)
      { dol = d; }
    void get() {
       cin>>dol;
    }
    void put() {
       cout<<dol;
      }
};
int main() {
      dollar d(5);
      rupee r;
       r = d
    r.show();
}
```

```
class rupee {
    int rup;
  public:
      rupee() { }

    void operator
        = (dollar d)
    rup = d.dol
            * 75;
    }
    void .show()
      { cout<<rup;}
};
```