CO1: Understand and realize OOP features through C++ and Java.

CO2: Design and Implement the Object oriented solution for problems using C++ and Java.

CO3: Understand & utilize STL classes in C++ & collection classes in JAVA

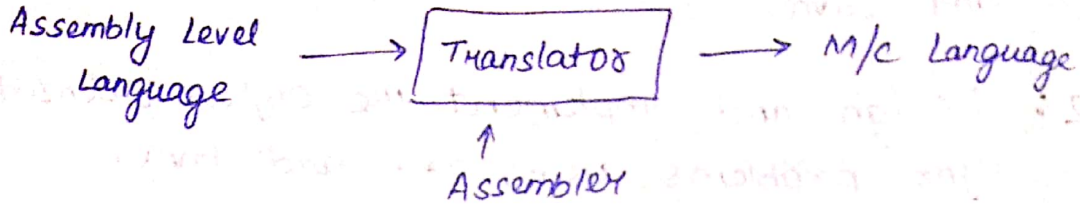
CO4: Understand and develop multithreaded programming and event driven programming in JAVA

LANGUAGES

⊙ LOW LEVEL LANG

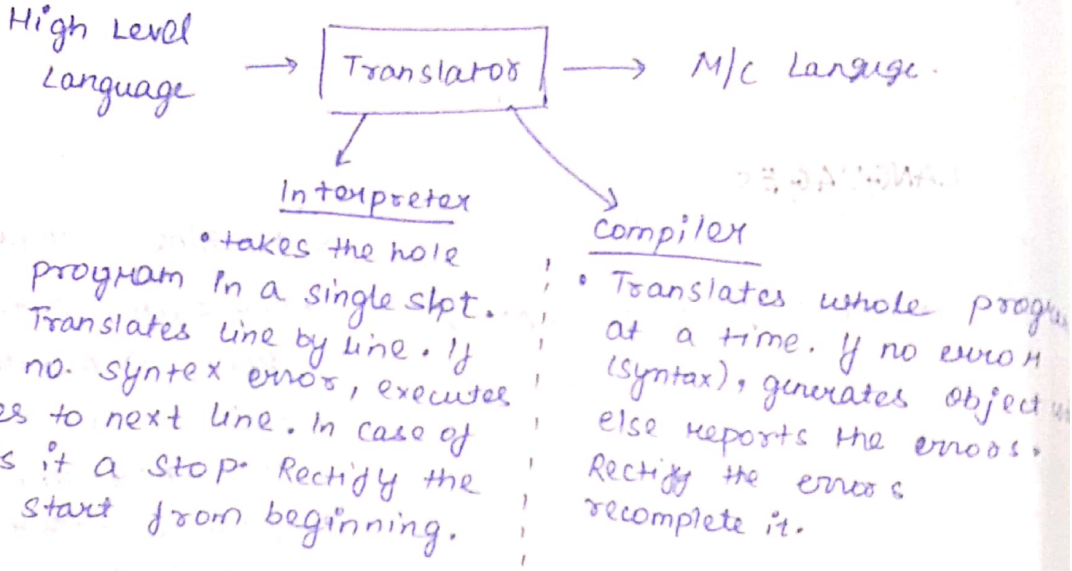
i) Machine language.
- For each operation ⇒ code consists of 0, 1.
- PROG. ⇒ A Collection of 0.1.
- Machine can directly understand as Programmer.
- Tedious error prone.
- Debugging is difficult.
- Hardware specific :-
  i) Instructions it supports register sets addressing mode.
- Machine directly understands. (Adv.)
- If hardware platform is changed may be codes to be rewritten.
- NOT Portable.

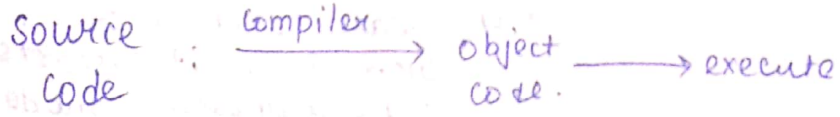
ii) Assembly Level Language:
- For every operation alphanumeric pseudo code is there.
- Easier for programmer.
- H/w specific not portable.



⊙ HIGH LEVEL LANG.

- English lines ∴ easier
- According to the rules, write the code.
- Portable and user-friendly
- Machine doesnot understand ∴ we need translator.





PROCEDURAL APPROACH

- Divide the task into number of sub tasks/procedures.
- Find the best/suitable algorithm for the procedures.
- Organisation of data is ignored.

MODULAR APPROACH / PROGRAMMING

- A set of procedures and data on which they operate.

DATA ABSTRACTION

- Hiding the complexity of internal representation of Data from end user.
- Behavioural aspects ⇒ 1) Parity with other similar types

OBJECT ORIENTED PROG.

- Major focus is on Data organisation.

Data type :- Description Part.
Behavioural Part.

① Size
② operation.
③ Range of values
④ Interpretation of Bit stream.

```
class Student
{
    int roll;
    char name[30];

    void word();
    void add();
};
Student x;
```

→ object

Object : An instance of a class.

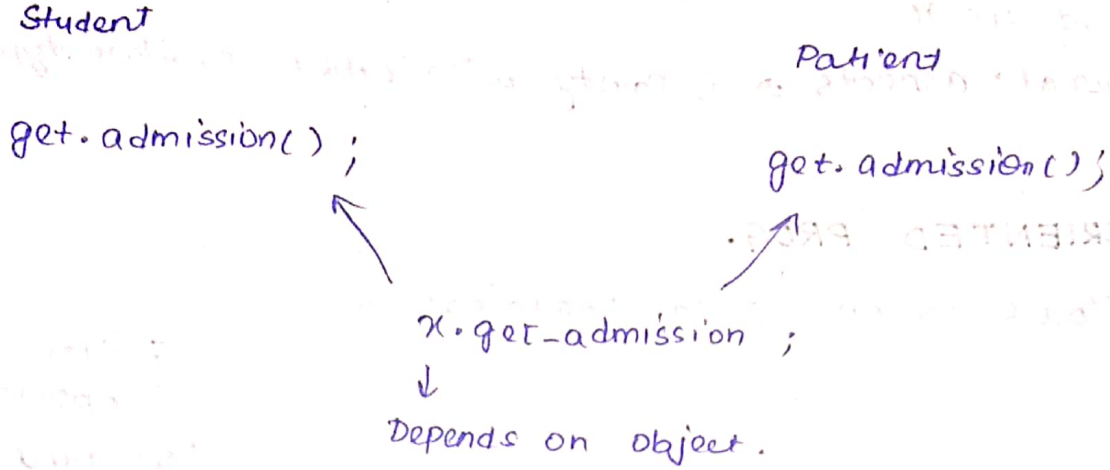
Class that defines a new datatype it combines the description & behaviour in a single unit. with access control mechanism is also there

```
Struct rec
{
    int roll;
    void get_info();
    void st_info();
};
Void rec :: get_info()

rec x;
x.get_info();
```

OO Properties

1) Class and objects
2) Polymorphism : Same method may have different Implementation depending on type of target object particular implement is choice.

Ex :-



* [Run-time polymorphism : static / Dynamic]

3) Inheritance : Sharing of attributes & behaviours between parent & child classes.
(super) (sub)
(base) (derived)
- Reuseability

C

variable declaration at the beginning of the block

C++

anywhere one can declare (before using it)

~~as a part of~~

In C++, recursive call to main() is not allowed.

In C

```
f()
{ int i;
  for(i=0; i<=10; i++)
  {
error ⟹ static int j=i;
  }
}
```

C compiler

Static variable in a function will be initialised at compile time. It remains the value at compile time.

C++ Compiler

initialize of static variable in a function is delayed till execution time.

```
char *c;
c = (char*)malloc(sizeof( ))
```

In C, void*

In C++, assigning void* to pointer of other types requires explicit casting.

—x—

* In C, every thing is passed by value.

* variable where changes required ⇒ send address of that variable.

```
void f(char *c)
{ printf("%s", c);
```

1) `const char * c = "abc";`

`*c = 'x';` ✗ Not allowed / Done

`c = "def";` ✓ Allowed.

—x—

2) `char * const c = "abc";`

`*c = 'x';` ✓

`c = "def";` ✗

3) `const char * const c = "abc";`

`*c = 'x';` ✗

`c = "def";` ✗

C

```
const int size = 5;
int array [size+1];
```

constant evaluation is done at execution time.

C++

constant evaluation is at compilation time.

basic data type, #define.

Scoping Rule & type check are preserved.

```
f(—)
{ const int x = 10;        →  Output: 10
  Printf("%d", x);
}
main(){
  f(—)
  float x = 5.0;           →  5.0
  Printf("%f", a);
}
```

Reference Variable (In C++).

```
int y;
int &x = y
```

x is reference to an int
Referring to an existing integer value y.

Here no further space is allocated to reference variable it is an alias to the variable it refers.

* x & y sharing same memory address.
* A reference variable must be initialised at the time of declaration.

```
#include <iostream.h>
{
// cout << "abc" << "\n" ;
        ↑
     Insertion
     operator
cout << x;
cin >>
    ↑
 extraction      //
  operator

int y ; int z;
int &x = y ;
cout << "Enter an int " ;
cin >> x ;
cout << "x = " << x << "y = " << y << "\n";
  x++
  &z=10 ;
  x = z ;                    // Output = 10.
 }
```

* Once * reference variable is initialized it will refer to same variable throughout its lifetime. For subsequent to it ⇒ value
⇒ assignment.

```
#include <iostream.h>
{  int x;
   x = 5;
  modify (x);
  cout << x;  // 10
}
```

```
void modify (int &p)
{ cout << p << "\n" ;  //
  p=2 * p;
}
```

```
#include <iostream.h>
{
    int x, y;
    cin >> x; cin >> y;
    int & z = larger (x, y);
}
```

```
int & larger (int &p, int &q)
{
    if p > q
        return p;
    else
        return q;
}
```

In C

```
void main (void)
{
    int max (int, int)
    int a, b;
    cin >> a; cin >> b;
    cout << max(a,b);
}
```

```
int max(int p, int q)
{
    if (p > q)
        return (p);
    else
        return (q);
}
```

```
#define max(a, b)
((a) > (b)) ? (a) : (b)
```

In C++,

Inline function

```
inline int max (int a, int b).
```

```
inline int max (int p, int q)
```

Request to the compiler to make the function inline. If granted, every call of the function is replaced by particular code.

FUNCTIONS WITH DEFAULT ARGUMENTS

```
float calc-percentage (int, int =100);
```

↳ Default value

At the time of call if actual parameter does not have the value for that, default value shall be considered. Otherwise, actual parameter passed will be taken

```
* void main (void)
  {
  cout << calc-percentage (76);
  cout << calc-percentage (38, 50);
  }
* float calc-percentage (int &, int tin)
  {
     float p;
        p = (& * 100.00) / tn;
        return (p);
  }
* float calc-percentage (int &, int tn = 100)
  {
      float p;
        p = (& * 100.00) / tn;
        return (p);
  }
```

* f (int, char, int)

f (int, char = '\0', int)

Parameters that can have default value must occupy the rightmost part in the argument int;

✱ Not possible. (cannot pass Int b/w)

```
void f (int, char = '0', int =0);
```

f (x, y); f (x); f (x, y, z)

FUNCTION OVERLOADING

* In C, for a function name, only one defination exists.
* Same function name can be used for multiple defination. signature of the functions will be different.

Argument list + (Return type)

```
int f(int, char){
    :
}
int f(int)
{
    :
}
```

```
int a, b; float f;
char c;
:
f(a, c)
f(a)
f(c)
```

No. of arguments + type in order = Signature.

N arguments : For each actual parameter P_1

$S_1 = S_2 = \cdots S_N = \phi$

{ S_1 = set of definations with which P_1 matches.

}

$S = S_1 \cap S_2 \cap S_3 \cap S_4 \cdots \cap S_N$

If $|S| = 0$ OR $|S| > 1$ then, ambignity

else, execute the defination.

Rules :-

→ Exact match OR MATCH after trivial conversion.

→ Promotion
- → char → int
- → short int → int
- → float → double.

Example of Trivial

ACTUAL	TRIVIAL
T	T&
T&	T
T[]	T*
T	const T

→ Standard Conversion.
→ User defined conversion
→ Ellipsis → f(...) matches with any no. of any argumented types.

f(int, char)
f(int)
f(char, int)

int a, b;
char c, d;
f(a, b)

Applying the rule in order;
→ Rule j° is taken (provided Rule j does not gives any mates)
→ applied on all definations
→ if matches the the def. included in the comp. set S_i
→ if $S_i = \phi$ then go for next rule.
else try with next parameter;

S_1 = f(int, char)
S_2 = f(char, int)
$S_1 \cap S_2 = \phi$

f(int, char = '\0')

CLASS Mechanism that helps to define new data type.

description ⇒ attributes
behaviour ⇒ ~~operation~~ operation.

class encapsulates the description and operations. These and bundled together.

```
class student
{   int Roll;
    char name[31];
    float score;
    void getdata(void)
    {
        cout << "---";
        cin >> Roll;
        cout << "    ";
        cin >> name;
        cout << "---";
        cin >> score;
    }
    void showdata(void);
};
```

⇒ attributes / data members.
private : cannot be access outside the member function.

- Member functions are shared by all the object.

scope resolution

```
void student :: showdata(void)
{   cout << "Roll : " << Roll << "Name : "
        << "name" << "score: " << score << "\n";
}
main()
{
    student S2;
    student S2;
    cin >> S1.Roll; // Not accessed
    S1.getdata();
// Any member function will be invoked by same object.
```

object : Instance of the class;
[Each object will have their own copy of data members]

S1: Roll name score

S2: Rollbar swore

→ Reference of invoking object is automatically passed.

```
void subscore(int v){
        score = v;
}

int main(){
    int m;
    Student s1;
    s1.getdata();
    cout << "Newe score: ";
    cin >> m;
    s1.setscore(m);
    s1.showdata();
}
```

s1 s2

Student_with_higher_score()
Object with higher value.

this :- points to invoking object.

```
Student studentwith higher score (Student t)
{
    if(t.score > score)
        return t;
    else return *this;
}
```

* data members ⇒ private (Normaly)
  member functions ⇒ public

* data members → public
  (security is compromised) Not advisable.
  member function = private
  Put a public interface as a level of reduction is sensitive task.
  private.

* No. of operations have common subtask.
  ↓
  (made a seperate function do this)

If this subtask be called always as a part of other tasks & never called independently, make it private.

```
class student{
        static int roll; int last_roll;
        int roll;
        char name[31];
        int score;
    public: void getadmission()
    {
        last_roll++;
        roll = last_roll;
        cin >> name;
    }
```

static data members : A single copy exists for the class. All the objects of the class shows the same copy. It is called instance independent variable / class variable. (not tied with any individual object).

size of an object >= sum of the size of the non static data members

```
static void show_of_students()
{
    cout << last_roll << "\n";
}
```

Student::show_of_students();

A static member function does not have any invoking object so it doesnot have "this" pointer, Cannot access the non-static members with implicit explicit this pointer.

MEMBER FUNCTION

- MANAGER FUNCTION
- ACCESSOR FUNCTION
  [can access the data members but cannot modify them]
- MUTUATOR / IMPLEMENTATION FUNCTION

```
Class Student {
    int roll ;
    char name [31];
    int score;
public:
    int add_score () ;
    {
        return (score) ;
    }
}
```

accessor ← return (score) ;

Constructor

Special type of number function automatically invoked when an object is created used to initialize the object.

[In C, int x = 5 ; → declaring + assigning.]

* Name of a constructor is same as class name.
* It has no returntype. (even not void).
  Implicitly it returns object reference
* May have arguments.
* Normally public or protected. (In a very special scenario it may be private).

⇒ Global object

On local static object ⇒ constructor is called before main.

⇒ any local object.

New == malloc. in c++.
delete == free in c++.

```
int *p
p = new int[10];
delete [] p;
```

* Using new an object is created.
* passing the values at an object to a function.
* A function is returning the value of an object (temporary object is created)
* As long as no constructor is written s, system a provides a default constructor. It has no argument.

```
Classname (void){

}
```

* Once user defines atleast one constructor system without its own constructor.

```
class student
  {  int roll;
     char name [31];
     int score;
  public:
     student (int r)
     {  cout << " student (int) " << "\n";
        roll = r;
     } student (int n, char name [ ])
     ;  {  cout << " student (int char [ ]) " << "\n";
              roll = n;
  }     }  strcpy (name, num);
```

* An Object can be declared in following the manner so that a matching constructor is found.

Destructor automatically called when the lifetime of an Object is zero.

lifetime ⇓ destroys.

→ no return type
→ no argument
→ no overloading
→ ~ classname( );

```
~student()
{
  if (name != NULL)
  delete [] name;
}
```

* Order of creation & order of destruction are Reverse.

Friend function

* Only member functions can access the private members
* We need to work with objects of more than one type
* A function which is a friend of a class can access the private members.
* Member function invoking object its reference is automatically passed. (Implicit argument).
* NO invoking object of that class. This object has to be passed explicitly (By value, by reference).
* A function can be member of one class and max act as friend of other class(s).
* A function can be member of none but acting as friend of multiple classes

✓ Global function.

```
class stock
  {
    int qty;
    !
  public:
      stock (int q=0)
      { qty=0;
      }
```

```
class sale
{
  int qty;
    :
}
public:
    sale(int q=0)
    {
      qty = q;
    }

void update (stock & t1, sale & t2)
  {
     t1.qty = t1.qty - t2.qty;
  }

friend void update ( sale ) ;
```

```
main
stock st;
sale s;

update (st, s);
st.show ();
```

member of no class acting as friend stock & sale.

Two classes

A — B

All the member functions of B may require to work with private members of A.

```
B = f1()
    f2()
    f3().
```

```
class A {
    :
public :

friend B :: f1()
friend B :: f2()
friend B :: f3()
```

```
class Node
{ int data;
  Node *next;
public :
friend class
      List;
}
```

```
class List
{ Node *head;
public :
    List()
    {
      head = NULL;
    }
addnode (int v)
{ Node *p;
p = new node;
```

OPERATOR OVERLOADING.

```
class complex
 {
   int A, im;
   public :
      complex (int i = 0, int j = 0)
         {  A = i; im = j; }
      add (complex c)
      {
        complex t;
        t.A = this.A + c.A;
        t.im = im + c.im;
        return (t)
        }
 };
```

Complex c1(2,5) , c2(7,6)

c3 = c1.add(c2); // c3 = c1 + c2;

operand on right ↓ operator on object ↓

c1.operator+(c2)

or

operator+(c1,c2)

To overload the operators,

* Either a member fn or a global friend function (non static)
* Name of the ~~same~~ function is operator operator symbol.

operator+

operator →

* One cannot define an overloading function for a new operator symbol.

```
complex operator + (complex c)
{
    complex t;
    t.A = a + c.a;
    t.im = im + c.im;
    return t;
}
```

OR

```
friend complex operator (complex, complex)

complex operator (complex t1, complex t2)
{
    complex t;
    t.A = t1.A + t2.A;
    t.im = t1.im + t2.im;
    return (t);
}
```

c1 = c1 * 5; ⟶ c1.operator *(int)

Member function

```
complex operator * (int m)
{
    complex t;
    t.A = t * A;
    t.im = t * im;
    return t;
}
```

If c1 = 5 * c1 → No member function can support it

operator * (int, complex)

Scalar operation object

↓

Not an object of type of

```
complex operator * (int k, complex c)
{
    return (c * k);
}
```

cout << x ; → Built in data type.

ostream &

↓ insertion operator

cout << student

↓

Student

```
public:
    friend void operator << (ostream &, student)

void operator << (ostream &o, student s)
{
    o << "Roll: " << s.Roll;
}
```

```
~String()
{
  if (size != 0)
  {
      *cp = *cp-1;
      if (*cp == 0)
      { delete [] c;
        delete cp;
      }
  }
}
```

—x—

Type casting

	Source	Destination
1.	Primitive Data type	Primitive Data type.
2. *	Primitive Data type	Object of a class
3. **	Object of a class	Primitive data type.
4. ***	Object of one ~~called~~ class (class A)	Object of another class (Class B)

1. ⇓ type cast operator B().

   ⇓ Write a constructor with A as argument.

* Implicit
  Explicit

```
int a;
float f;

a = f;
(destination) (source)
```

```
int a, b; float c;
c = (float)a) / b;
```

C (float) a

C++ float (a)

```
class A
  A obj;

  A (int k)
  {
     :
  }
```

Obj = a;
↑ destination ↑ source.

Makes call to a suitable constructor get object.

2. In the destination class, write a constructor with * primitive type as the argument.

3. typecast operator overloading :- operator type ( )
↓ in which casting is required

```
class length
{  int cm;
   int mm; int a;
  public:
     :
};
length l;
:
int a = l;
  ↓
length in mm

operator int()
{  return (10 * cm + mm)
```

- It is always a member function.
- It will not have any argument
- There is no return type but it returns a value of type in which casting is required. (not even void)
as indicated in the prog

** in the source class have a suitable type cast operator overloading

```
Class A
{
   int k;
   Public:
      A (int i=0)
      {
        k=i;
      }
   A operator + (A t)
   {
       A x;
       x.k = k + t.k;
       return (x);
   }
  operator int ()
   {
     return (k);
   }
}

A a1(5), a2;

a2 = a1+5;

   a1.operator + (5)
                  ⇓
                A (int)

operator int (a1) + 5

        ⇓
   int+int.
```

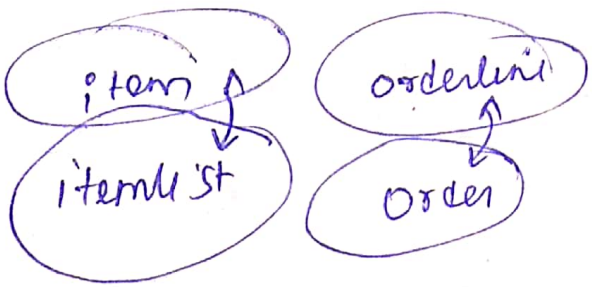
```
Class item
{
    char icode [5];
    char iname [31];
    float price;
    float qty;
    Public :
        void getitem (void);
        Void showitem(void);
        Void setprice (float p)
          {
            price = p;
          }
        float set_price ();
        float set_qty ();
        float set_qty (float q);
        char * set_icode ();
};

Class itemList
{
    int cnt;
    item list [10];
    public :
        itemlist ( )
        {
          cnt = 0;
        }
      void prepare_list ( )
        {
          item t;
          t.getitem( );
          list [cnt ]= t;
          cnt++;
        }
```

```
int fixed_icode (char * c)
{ int i ;
  for(i=0 ; i< end ; i++)
  { if (strcmp ( c, list[i].set_icode)
                    == 0)
      return i;
  }
  return (-1);
}

Void prepare_list ( )
{  item * ;
   to getitem ( )
   int k;
   if (cnt != 0)
   { k = find_icode ( t. — ).
     if (k >= 0)
     ——————
   }. list [cnt] = 1;
      cnt++ ;

Void search item (char * c.)
{
     int k ;
     k = find-icode ( c ) ;
      if ( k >= 0)
      ——————
     else
      ——————

}..
```

```
class Order
{
    int cnt;
    OrderLine list[10];
    public:
      Order ()
      { cnt = 0;
      }
     collectOrder (iList) *
     {
       list[i].getdata ();
     }
};
```

```
class OrderLine
{ char icode[5];
  float qty;
  public:
    voide getdata();
    char * set_icode();
    float set_qty();
      :
};
```

```
main ()
{
   itemList, iList;
  Order O;
O.collectOrder(iList);
```

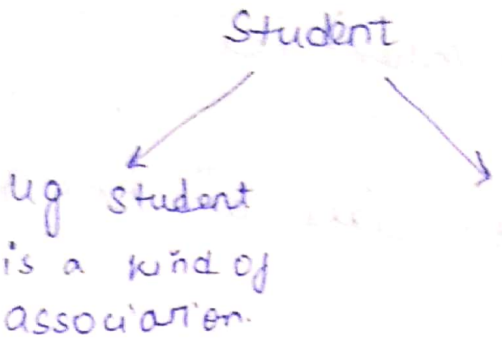


Inheritance

⇓

Mechanism for sharing attributes and behaviour from parent / base / super class / child / derived / sub class inherits.

* Parent class is the generalized version & children classes are the specialised version.




```
class student                    char Department
{  int ~~roll~~ registration;      {  int dcode;
   char name [31];                   char dname [31];
   public :                          public :
   :                                 :
};                                 };
```

1. Process of specialization.
   top down approach identified parent / generalized classes
   ⇓
   Deriving the children specialized classes.

2. Generalization
   ⇓
   identified the specialized class.
   Finding wrs of commonalities have write those to write generalized class apply Inheritance.

C++

A
↓
B

Mode of inheritance

A → public
B → Private

class B : mode of inheritance (public/ private/ protected) A

{
add attribute & add / modified behaviour
}

Irrespective of mode of inheritance

⇓

private members of base class are not accessible in derived class.

Private
⇓
Not accessible in derived class

Protected
⇓
accessible in the derived class.

Status (int terms of accessibility)

At base class members in derived class also depends on mode of inheritance.

Base class members \ Mode of inheritance	Private	Protected	public
Private	X	X	X
Protected	Private	Protected	Protected
Public	private	Protected	Public

X = Not Accessible

Scope / Accessibility of base class members in derived class.

A Protected x

Private ↓

B Private x

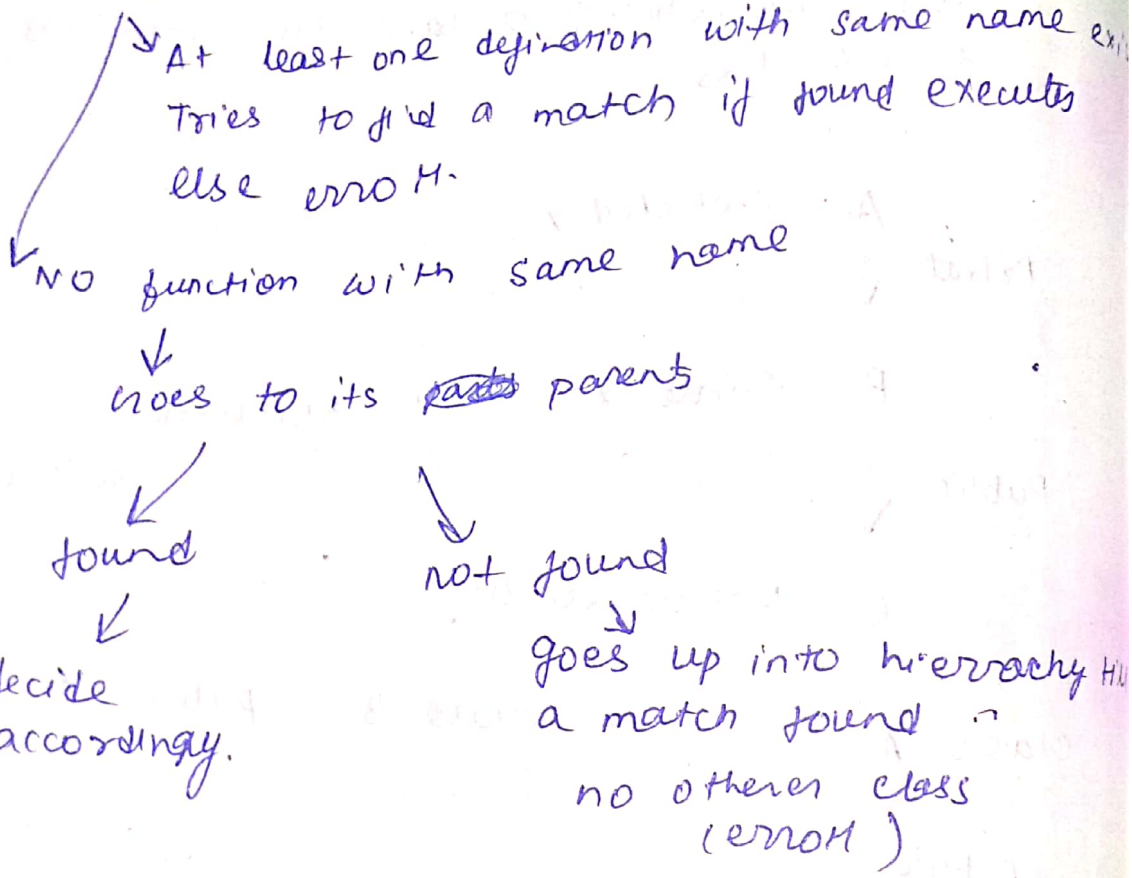
Public ↓

C Not accessible.

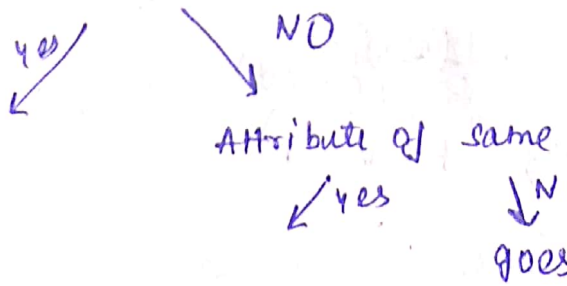
Multilevel Inheritance.

```
Class A
{
Public:
    int f1(void);
};

Class B : public A
{
:
Public:
    f1(void)
    f2(void);
};

B b;
b.f1();        // call this one.
b.f1(10);      // Error (Function overloading not allowed
b.A::f1(10)    // Not
```

* function overloading is limited to single class defination. (NOT ACROSS THE CLASS).
* Whenever a base class in 1's redefined in derived class (signature may be same or may not be same).
* Function overloading invoking a function using derived class object.
* looks for a function with same name in its own class.



* local variable



INHERITANCE

HOW TO RESOLVE METHOD TO BE INVOKED

C++
Function overloading limited to single class

C++
Function Overloading → using same function name base/derived.

* Static members in the base class.

↓
Inherited.

derived class object will also share the static class membes of base class

A static int x;

↓

B

friendship
A class can be friend of another.

A —is friendly→ B

derived ↓

C

constructor
Before executing the body of derived class constructor, base class part has to be inherited.

A

↓

B

B( ) : A(—)

{ ... → Initialize only the addl. attribute of derived class.

}

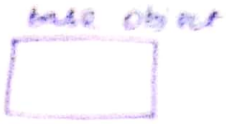
B(arguments) { — — }.

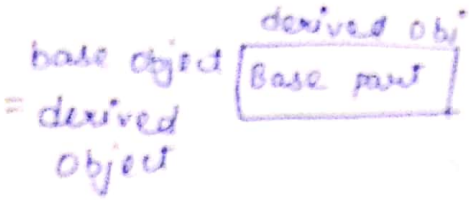
```
class student
{
    int roll;
    char name[31];
    public:
        student()
        { roll = -1;
          name[0]='\0';
        }
      student(int r, char *c)
      {
        roll=r;
        strcpy(name, c)
      }
};

class pgstudent : public student
{
    char supervisior[31];
    public:
    Pgstudent()
        {
          supervisor[0]='\0';
        }
    Pgstudent(int r, char *n, char *s): student(r,n
        {
          strcpy(supervisor, s);
    }
```

A → copy constructor

B (const B &t) : A(t)

derived class reference is automatically converted to base class





For this we need a Constructor

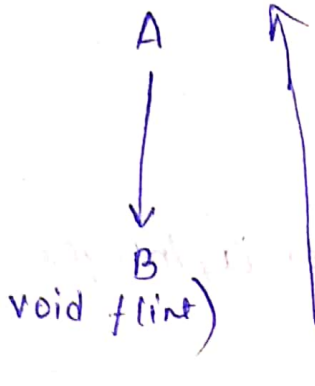
derived obj = base obj ;
(Not automatic)

```
Class Integer
{
    int x;
public :
    Integer (int);
    friend Integer operator +( Integer i1, Integer i2 )
    showdata ();
};
```

→ Integer (Integer)

```
class NewInteger : public Integer
{
  public :
    Showdata ();
    {
      Integer :: showdata ()
    }
```

```
NewInteger ni1(5), ni2(6), ni3 ;
    ni3 = ni1 + ni2;
void f(void)
```




```
//error p = f(10)
```

// base class pointer can point to both a base ~~do~~ object and a derived object. but access is limited to base class part.

* Compile time linking / static.

* Make the function virtual in base class and overwrite this in derived class with same signature.

* Base class ~~ptr~~ pointer

↓

many pt. to ~~pass~~ base object or derived object depending upon the type of the object it points, the particular version is chosen.

```
A
{
int *p
{
A( )
{
  p = new int [10];
}
```

```
A ( )
{ delete p[ ];
}
```

```
B
char *c;
B( ) ⇒ we have pointer dynamic allo. ⇒ constructor.
{
c = new ___;
}
```

Whenever a derived class object is destroyed.

⇓

derived class destructor is invoked.
then base class destructor.

virtual ~A ( )

MULTIPLE INHERITENCE

In c++, a derived class can have multiple base class. (pvt./prot.)

Attributes of A & B and

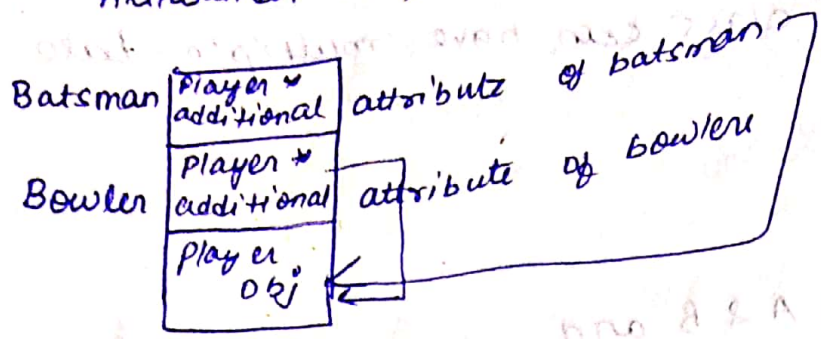
More than one base class has a fn then it has to be overwritten. Otherwise, the function with derived class Object implicitly, we are to specify which function do we need.

* VIRTUAL BASE CLASS.

A class has multiple base classes and they have a common ancestor. The lowest level class will have multiple copies of common ancestor via different base classes.

* If multiple copies are not required than the common base class to its ancestor has to be declared as virtual children.

Allrounder ar;



Abstract class.

Permanent employee
contractual employee
Employee.

```
class Perm_Emp
{ char emp[5];
};
```

```
class cont_Emp
{
};
```

Number of classes sharing lots of similarity in terms of attributes and behaviour. To avoid redefining the things, someone can utilize inheritance. By the process of generalization, one can form a base/super/general class with common characteristic other classes will be derived from it.

In reality, no direct instance of general/super/base class exists. To insure this the base class is made abstract.

Abstract class is a class for which no direct instance can be created.

⇒ Helps to utilize inheritence

C++,

A class with one or more pure virtual function is an abstract class.

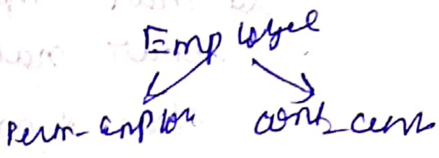
Any class derived from it must override the pure virtual functions of the abstract base class with same signature.

```
class Employee
{
    char emp-id[7];
    char name[32];
    Char dcode[5];
    Char address[91];
    char phone[15];
    public:
      void getdata(void){ ... };
      void showdata(void){ ... };
    virtual void complete(void x)=0;
};
```

Pure virtual function NO Implementation.

```
class Pcom_Emp : public employee
{ char designation[5];
  float basic-pay;
public:
  void compsalary(void)
{
```

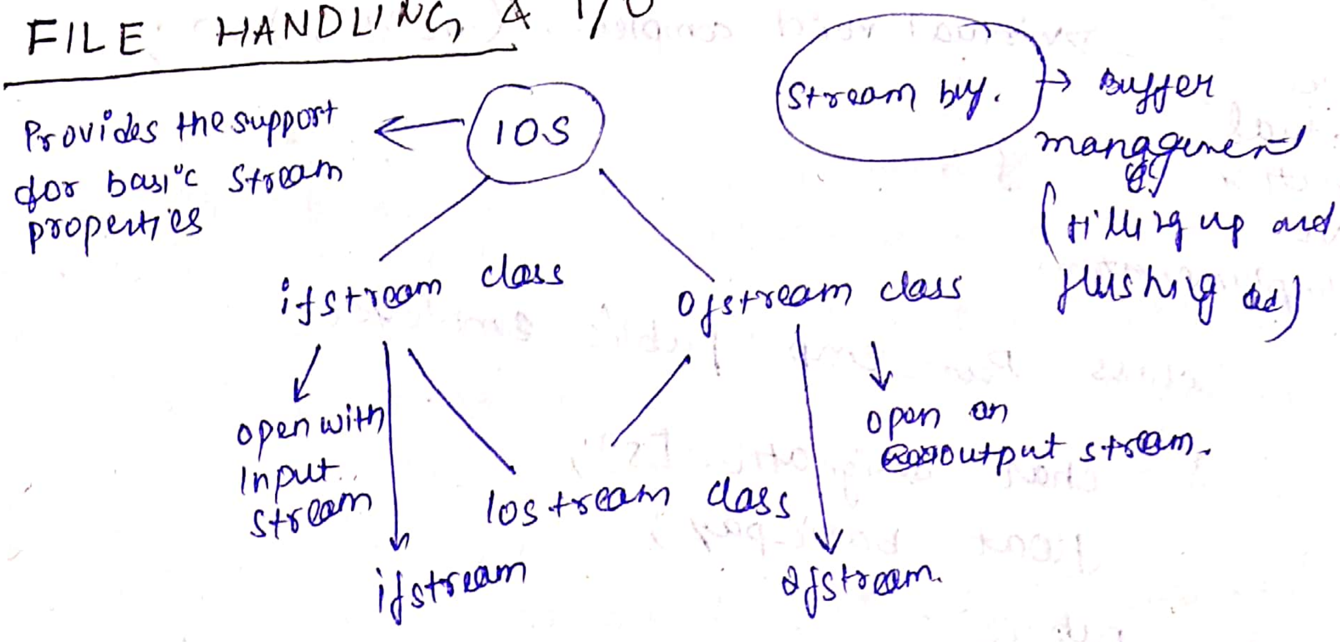
```
    float salary;
    salary = basicpay + * basicpay;
    cout <<" ---;

  }

class contemp: public employee
{
    float ---
    float ---
      :
void compute_salary_not ( )
{
    float salary;
    salary = ----

};
 void calc_p--- void
 {
 }
```



employee e; X
perm_employee pe

✓ Employee *p;
p = &pe;

FILE HANDLING & I/O



device — stream of byte — Program

istream — Extracting data / bytes

← Output stream — inserting data

cin >>
↑
Extraction operation.

cout <<
↑
Insertion operation.

command 1 | comand 2

ls -l | ws -l

```
char ch;
cin >> ch;
while (ch != '\n')
{
cout << ch;
cin >> ch;
}
```

Input = abc def

Output = abcdef (And program will not terminate)

Skips space, tab, newline → cin >> ch

cin.get(ch) → Takes space, tab, newline

```
cin.get (char *, int)
cin.get (name, 31);
```

At most 30 char and : e

→ Output = abc.

* cin.get (name, 31).get (c);

```
char name [31];
Ofstream outf ("sample.txt")
cin.getline (name, 31);
cout << name ; // Displaying in screen.
outf << name;
    outf.close ();
```

```
ifstream inf("sample.text");
inf.get(ch);          cin.get(ch);

while (inf.get(ch))
{ cout << ch;

}
inf.close();
```

```
ofstream outf;
outf.open("sample.txt", ios::out or app);

                ios::in  // ifstream.

fstream f;
f.open("  ", ________)
          opening expn.

ios::in, ios::out, ios::app, ios::binary;

ofstream f;
f.open(____, ios::out, ios::app, ios::in...)

f.write(char*(&s), sizeof(s));
f.close;
while(fread(
{ s.show
}
f.close();
```

```
class student
{
    int roll;
    char name[31];
    int score;
  public:
    student();
    student(____);
    void getdata(void);
    void showdata(void);
    int  ret_roll(void);
    void update_info(void);
};

class studentfile
{
    char fn[21];
  public:
    studentfile(char *f)
    {
      strcpy(fn, f);
    }
    addstudent(student);
    update_record(int);
};

void studentfile :: addstudent(student s)
{
    ofstream f;
    int m, k;
    m = s.set_roll();
    k = search.roll(m);
    if (k=0)
    {
      fopen(fn, ios::out, ios::app, ios::binary);
      f.write((char*)&s sizeof(s));
      f.close();
    }
}
```

roll number can be

int search_roll(int);
↓
Returns 0 if not found

```
void studentfile :: display (void)
{
    ifstream f; Student s;
    f.open (fn, ios :: in | ios :: binary);
    while (f.read ((char *) &s, sizeof(s)))
    {
        s.showdata ();
    }
    f.close ();
}

int searchch_roll (int);
int studentfile :: search_roll (int n)
{
    ifstream f; student s; int k=0; int chk=0;
    f.open (fn, ios :: in | ios :: binary);
    while (f.read ((char*) & s, sizeof (s)))
    {
        k++;
        if (s.ret_roll () == n)
        { chk = 1;
          break;
        }
    }
    if (chk == 0)
        return 0;
    else return k;
}
```

* ate · At end similar to app allows to move back & overwrite existing record.

* fseek() & ftell() → In C.

seekg() → get seekp() → put
tellg() → position of get pointer tellp() → position of put pointer } In C++

```
void StudentFile :: update_record(int n)
{
    fstream f;
    int k;
    Student s;
    k = search_roll(n);
    if (k==0)
    { cout<< "Roll does not exist";

    }
    else {
       f.open(fn, ios::ate | ios::out | ios::binary);
       fseekg((k-1) * sizeof(student), ios::beg);
       f.read((char*)&s, sizeof(s));
       s.update();
       f.close();

    }
```

```
class Studentinterface
{  public:
     static void showmenu()
     {  int opt, student s;
        StudentFile st("studentdat");
        while(1)
        {
           cout << "1. Add Student....
                !
           cout << "...      ";
           cin >> opt;
```

19/02/19

CT-1

```
if (opt == 8)
    break;
switch (opt)
{   case 1:
        s.getdata()
        st.addstudent(s);
        break;
    :
    :
}.  studentinterface :: showmenu();
```

—x—