entity

day / date:

Entity: attributes and behavior

object ≠ object → noun → verbs

class → variables → functions

1. Identify all entities
2. Identify attributes and behaviors
3. shortlist all relavent entities

entities for uni

Students	a- names, number, gender b- studying, passing
Teacher	a- name, number, email b- teaching, grading
Staff	a- name, number b- cleaning
Department	a- name, teachers b- grading
Calleteria	a- food, Price b- selling, restocking
Transport	a- name, bus no, routes b- driving,

blue print - not generalized

instance: single occurence of a class

object

```
Class car
{
    string color;
    int capacity;
    string model;
    void race()
    { }
    void brakes()
    { }
}
```

```
int main()
{
    car car1;
    car1.color = "blue";
```

encapsulation / group

Information Hidding / Abstraction

day / date:

in practice class is used to be used by someone else

```
Class car                    in defaut C++ class is Private      int main()
{ Public:                                                        {
    string color;                                                   car ob1;
    void acc ()                                                     ob1.acc();      X  if private
    {                                                               ob1.color: "red"; X
      cout << "Accelerating" ;
    }
    Private:
    void break ()
    {
    }
};
```

PRIVATE

PUBLIC

dont know it's code ← information hiding

- private → variables → information hiding
- Public → functions
- protected

encapsulation : information Hiding

Advantages of information hiding:

1. Simplicity
2. Prevent stealing
3. Preventing accidental use

Pillars of OPP

1. Encapsulation
2. Abstraction
3. Polymorphism
4. code reuse

Setter & Getter

day / date:

Struct	vs	Class
1. Members are public by default.		Members are private by default.
2. Struct are value type.		Classes are reference type.
3. cannot be abstract.		can be abstract.

abstract: incomplete classes can be completed by some one else

WONT COME IN PAPER

```
class Student
{
    float cgpa;
    Public :
    set CGPA
    void setCGPA( float par)
    {
        if (par<0 || par>4.0f)
            cout << "CGPA is not valid";
        else
            cgpa=par;
    }
    float getCGPA( )
    {
        return cgpa;
    }
```

setter/mutator function

getter function

```
int main()
{
    Student usman;
    usman.setCgpa(6.0f);
    cout << usman.getCGPA();
}
```

Constructors

day / date:

```
class BankAccount                    int main()
{
   string accountno;
   float balance;
   string pw;
   Public:
   void deposit(float a, string p)
   {                  for strings comparing
      if (P.Equals(PW))
         balance += a;
   }
   void withdraw(float a, string p)
   {
      if (P.Equals(PW))
         balance -= a;
   };
```

Problem in using setter and getter 🙅: 1. calling getter before setter.

2. extra indexing

3. dependency

constructors : special type of functions.

1. has no return type
2. name is same as class

constructors are almost always Public

```
Class A                          int main()
{A(int P)                        {
{                                   A ob(5);
   n = P;
}
}
```

constructors

day / date:

Constructor

1. constructor ~~naa~~ name is same as class name
2. NO return type
3. Has parameters just like functions.
4. They are called just once
5. They are called implicitly indirectly
6. They are almost always Public

- used for ~~ave~~ the problem when calling getter before setter.

~~- We write it instead of getter setter.~~

-used to avoid getting garbage value

- constructor for initialization
- setter for modification

```
Class A
{
→ A(int a)
  {
    n=a;
  }
};
○ A()
  {
    n=0;
  }
}
```

same name
so main will recognise it using the Parameters
1. order
2. no. of Parameter
3. type of Parameter

parameter signature

```
int main
{
  A ob(5);
  A ob2;
  A ob3(10);
```

day / date:

CONSTRUCTOR TYPES

1. Compiler Provided default

```
class A                    int main()
{                          {
                               A obj;
   int n;
   Public
   A()                     }
   {
   }
};
```

2. user defined default

```
class A                    int main()
{                          {
   int n;                      A obj;
   Public
   A()                     }
   {
      cout << "My own constructor";
   }
};
```

3. Parameterized

Can make multiple constructors

```
class A                    int main()
{                          {
   int n;                      A obj;  // 1
   public                      A obj2(10);  // 10
   A(int p)
   {
      n = p;
   }
   A()
   { n = 1; }
};
```

day / date:

Copy constructor

```
Class A                    class B                    main()
{                          {                          { B b;
  int n;                     int n;                     A a(b);
  Public                     Public                   }
  A(B nyz)                   Void f()
  {                          { }
    nyz.f(); } ;
    f();
  }
};
```

data type → B; variable name → nyz; function in B → nyz.f(); function in A → f();

This constructor

```
student (strins name)
{
  this → name = name;
}
```

name inside class → this → name

Destructer

day / date:

Polymorphism:

↳ constructer overloading : same name but different Parameters

Destructer : used for clean up purposes

```
~ MyClass()
{
    cout << "Destroying";
}
};
```

→ exactly same as constructor except sign
→ can only be created once
→ called out of scope automatically
→ used during DMA
→ destructer ~~called~~ destroys from bottom to top
→ object of destruction is in reverse order of construction
→ when using new default deconstructer useless

DMA

day / date:

```
Class Myclass
{
    int *var;   // no memory allocated at this point
    Public
    Myclass(int value)
    {
        // allocate memory dynamically
        var = new int;
        *var = value;
    }

    ~ Myclass()
    {
        delete var;
    }

    int getvar()
    {
        return *var;
    }
};

int main()
{
    Myclass 01(5);
}
```

DMA

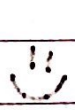
day / date:

```
Class Myclass
{
    int *var;   // no memory allocated variable at this point
    Public
    Myclass(int value)
    {
        // allocate memory dynamically
        var = new int;
        *var = value;
    }

    ~ Myclass()
    {
        delete var;
    }

    int getvar()
    {
        return *var;
    }
};

int main()
{
    Myclass o1(5);
}
```



Shallow / Deep COPY

day / date:

Shallow COPY

```
class Test
{
    int *var;
public:
    Test()
    {
        DMA:
        var = new int;
        *var = 0;
    }
};
```

For Deep COPY add this code

```
Test(const Test &n)
{ variable value cannot change
    var = new int;
    *var = *n.var
}
```

member — member (of copy constructor value)

```
int main()
{
    Test t1(5);
    Test t2 = t1;
    t2.setvar(6);
    cout << t1.getvar();
    Test t3 = t2;
}
```

Deep COPY

- does not change the original value when copying
- it is preferred over shallow copy

```
Test (int var)
{
    this -> var = new int;
    * this -> var = var;
}
```

member variable; member; constructor variable

```
void setvar (int var)
{
    * this -> var = var;
}
```

member; constructor

```
int getvar ()
{
    return *var;
}
```

member

~~Pointers with~~ constants

day / date:

Pointers with constants

```
main()
{       int b = 8;
        int a = 5;
```

1. int * PA = &a, &b, &c

2. const int * PA = &a;

```
PA = &b ✓   X   X   ✓
*PA = 15;  X   ✓   X   ✓
```

3. int * const PA = &a

4. const int * const PA = &a

1. Non constant Pointer and non constant data — everything can change
2. Non constant pointer and constant data — Pointer dereference value cannot change
3. constant Pointer and non constant data — pointer address cannot change
4. Constant Pointer and constant data — nor address nor value can change

```
class A
{
1. int y=0; X   const int y = 5;

   int y;
   Public:
2. void f(const int p , int q)
   {        value cannot change
      p=10; X
      cout << p; ✓
   }
3. void f(    ) const
   {  cannot change member variable of class
      y=10; X
      cout <<y;
   }
};
```

Classes with constants

1. const as member
2. const as parameter
3. const function
4. const objects

```
class A
{ int y;
  Public
  void f1()
  {  }
  Void f2() const
  {  }
};

int main()
{ A ob1;
  ob1.f1(); ✓
  ob1.f2(); ✓
4. const A ob2;
  ob2.f2(); ✓
  ob2.f1(); X
```

can only call const functions

MUTABLE / MEMBER INITIALIZATION LIST

```
class Employee
{
    const int emp_id;
    mutable string name;

Public:
    Employee()
    {
    }

    void f() const
    {
        name = "Ali";
    }
```

can be changed cuz mutable variable.

adv: we can change a specific variable w/o the others changing

Member initalization list: to give constants value on run time

```
Employee(string n, string e): emp_id(e), name(n)
{
}
```

- reserved for constructors
- order does not matter
- cannot be used for arrays
- recomended to initialise this way

STATIC

day / date:

```
class Employee
{
    string e-name;
    static int count-of-emp;
    float salary;

Public:

Employee (string e, float s) : e-name(e), salary(s)
{ }
void inc()
{ count-of-emp ++; }

}; int Employee :: count-of-emp = 0;

main()
{
Employee o1(" "," ");
Employee o2(" "," ");
Employee o3(" "," ");
o1.inc();
o2.inc();
o3.inc();
}
```

Static variable
- mostly used to count objects

instance member means non static

shared value for all class objects

scope resolution: ~~if you need to~~ for static, value is given out of scope
- is called once
- always used when using static

Employee:: count-of-emp = 3;
↓
preffered way for static variable so if anyone reads they will know its static

↳ count-of-emp = 3

day / date:

common data for all objects → static variable

common behavior for all objects → static function

~~static function~~

```
class Student
{
Public :   → make Private sir made it Pubic because need to make setter getter.
string name;
string ID;

static string campus;
void f1()
{

}

static void f2(&student a)
{ can not use instance variables here.
  name:"Ali";  ——X
  f1()  ——X      a.f1()  ——  compiler can recognise object
} static ~~variable~~ function only works with static variables

}; string student :: campus
```

```
int main()
{
student s1;
student s2;
s1.campus = "KHI";
student:: campus = "ISB";
s1.f1();
student::f2(s1);   ← can call w/h an object
```

Inline Functions

day / date:

```
int main()
{
    int i=5
    cout << square(i);

}

inline int square (int a)
{
    return a*a;
}
```

since code is very less we can advise compiler to replace # the ~~cout~~ code with where the function is called.

adv: stack memory overaned avoided

dis: during compilation code will be increased

boiler plate

if function has loops advised not to use loops.

~~inline~~ rejected

i - Function has loops

ii - Function is recursive

iii - It has switch/goto

iv - It has static variables

v - It has a return type other than void & does not return a value

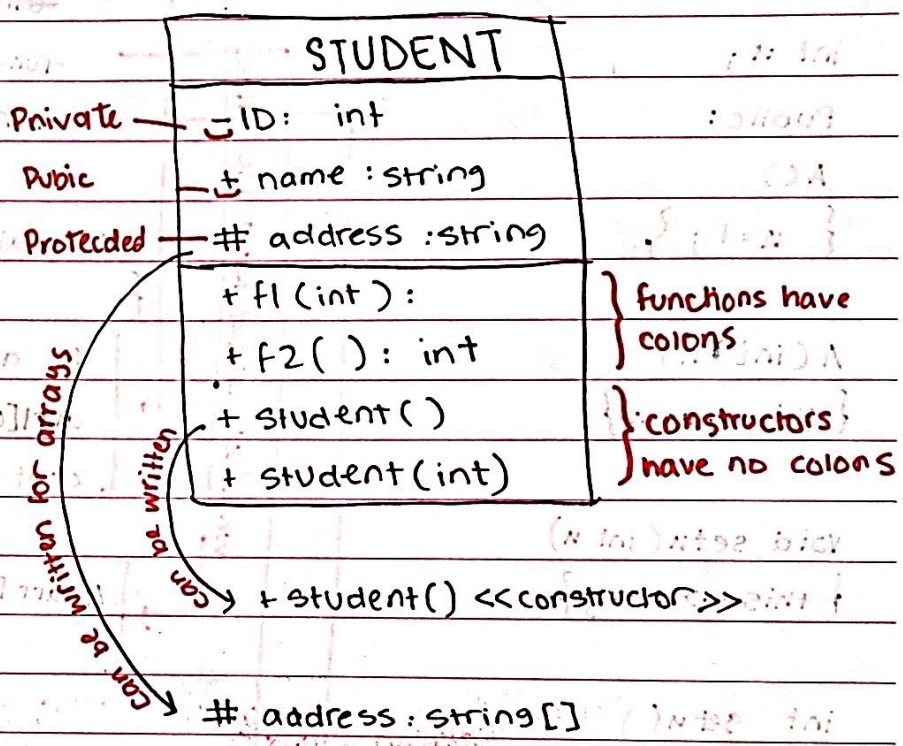
UML: universal markup language

only document overall structure

```
class Student
{
  int ID;
public:
  string name;
Protected:
    string address;
Public:
  student()
  {      }
  student (int n)
  {          }

  void f1(int a)
  {          }
  ~~void~~
  int f2 ()
  {          }

};
```



: class diigrams dont need to show if its a pointer

ARRAYS OF OBJECTS

```
class A
{
  int n;
  Public:
  A()
  { n=1; }

  A(int n)
  { this -> n = n; }

  void setn(int n)
  { this -> n=n; }

  int getn()
  { return n; }
};

void f(A *arr)
{
   arr= new A[5];
   arr[0].setn(99);
   *(arr).setn(98);
   *(arr+3).setn(97);
}
```

same

↓ address dereference

global function

when you already know the values

- compile-time array using default constructor
- compile-time array using parameterised constructor
- ~~compile-time~~
- run-time array using default constructor
- run-time array using parameterised constructor

```
int main()
{
   A arr[5];
   arr[0].setn(9);
   cout << arr[0].getn();

   A arr[] = {A(5), A(7), A(3)};
   arr[0] arr[1] arr[2]

   arr = arr1;   To copy array, deep copy

   A * arr;
   f(arr);

   A * arr;
   h(arr);
}
```

arr[i].setn(99)

```
void h(A *arr)
{
   arr= new A[5];
   for(i=0; i<5; i++)
   {
      arr[i] = A(i);   → coping in array
   }                      ↓ object
}
```

IS-A HAS-A Inheritence

day / date:

Employee IS-A person

Employee HAS-A name

Student HAS-A name

child class ↗ Parent class ↗

```
class Person
{
    string name;
public:
    Person()
    { name= " " ; }

    Person (string name)
    { this->name = name ; }

    void showname( )
    { cout << name;}
};
```

```
class Employee : public Person
{
public:
    Employee()
    {  }

    Employee(string name) : Person(n)
    { name = n;}
};
```

this class has a variable name as it is inherited from its parent class

to call parameterized constructor

Person

Employee — Student

Developer — Designer

```
int main()
{
    Person P1("Ahmed");
    Employee E1("Ali");
    P1.showname( );   Ahmed
    E1.showname( );   Ali
}
```

day / date:

TYPES OF INHERITANCE

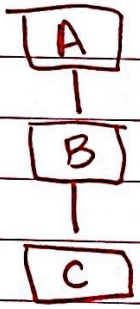
Based on relationship

- ↳ single
- ↳ multilevel
- ↳ multiple
- ↳ hierarchical (tree)
- ↳ hybrid

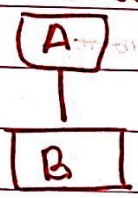
Based on access modifiers

- ↳ Public
- ↳ Private
- ↳ Protected

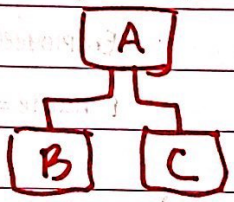
MULTILEVEL



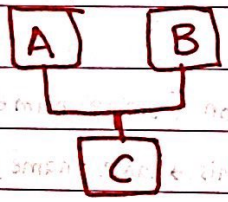
SINGLE



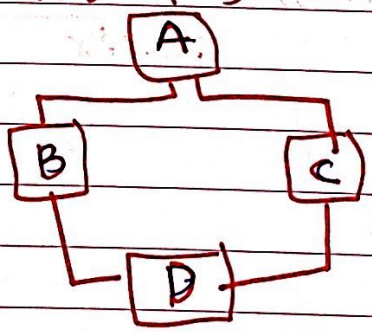
TREE



MULTIPLE

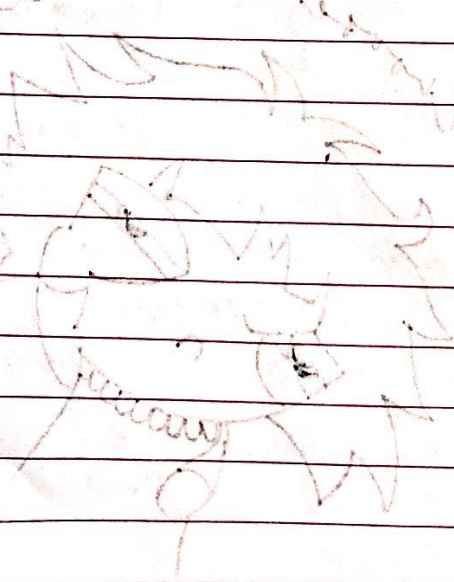


HYBRID



multiple inheritence

class A : Public B, Public C



Protected INHERITANCE

effect is shown on derived class and on current

```
class A                    class B : Public A          class C : Public B
{                          {                           {
  Public:                    Public:                     Public:
  int pubvar;                void f2()                   void f3()
  Private:                   {                           {
  int privar;                  cout << pubvar; 10          cout << pubvar; 10
  Protected                    cout << privar; error       cout << privar; error
  int protvar;                 cout << protvar; 30         cout << protvar; 30 error
  Public:                    }                           }
   A()                     };                          };
   { pubvar=10;
     Pri
     Pubvar=20;
     protvar=30;
   }
};
```

If protected then all variables here will be Protected

once protected, cannot be Public again

```
int main()
{
  C o1;
  o1.f3
  o1.pubvar=5;
  o1.protvar=5; X
}
```

protected cannot be used in main but can be in inherited classes.

POLYMORPHISM

day / date:

(same name different behavior)

- Constructor Overloading
- Operator overloading
- Function Overloading
- Function Overiding → runtime

Constructor Overloading
- name of parameter
- constructor name
- useless

Operator overloading
- ~~name of parameter~~
- ~~CONSTR~~

Function Overloading
- default Parameters need to be ~~use carefully~~ better not to use

→ compile time

Constructor overloading

```
class Myclass
{
    int n;
    Public:
    Myclass ( )
    { cout << "default"; }

    Myclass (int i)
    { cout << i; }

    Myclass (float f)
    { cout << f; }

};
```

```
int main( )
{
    Myclass O1;
    Myclass O2(5);
    Myclass O3( 2.5 );
}
```

double and float are compatible

implicit typecasting ←

function overloading

```
class Myclass
{
    int n;
    Public:
    void f(int i, float f)
    { cout << " F1"; }

    int f(char c)
    { cout << "F5"; }

    int f2
    ~~void~~ ( )
    { cout << "F2"; }

    void f(int i)
    { cout << "F6"; }

    void f(float f, int i)
    { cout << "F3"; }

    void f(int i=0, float f)
    { cout << "F7"; }

    void f( )
    { cout << "F4"; }
};
```

default Parameter

integer and char compatible

```
int main( )
{
    My class ob;
    ob.f (5,2); ✗ two integers
    ob.f ('k'); F5
    ob.f ( ); F4
    ob.f2( ); F2
    ob.f2(5); ✗
    ob.f ('j'); asci code will go in F6 (if we remove F5 function)
    ob.f ( 2.05 ); F7
```

VIRTUAL

day / date:

function overriding same name, parameter signature b/w parent/child classes

```
class Student
{
Public:
virtual void showID()
{ cout << "K.111234"; }
```

function overwrite ←

```
class CS_Student : Public Student
{
Public:
void showID()
{ cout << "CS-K11-1234"; }
```

```
int main()
{
CS_Student obj
ob.showID(); CS_K11_12
Student * O;
CS.student c;
O = &c;
O → showID(); K111234
```

virtual when written in Parent it is copied in child class

no virtual ← Early Binding

LATE Binding ← O → showID(); CS_K11_12

when using virtual

Diamond Problem: Problem of multiple inheritence : solution ~~virtual inheritence~~

```
class B : virtual Public A
{
Public:
//void show()
{ cout << "B"; }
};
```

```
class C : virtual Public A
{
Public:
void show()
{ cout << "C"; }
};
```

```
class D: Public B, Public C
{
Public:
};
```

```
int main()
{
D obj
```

no virtual ← O.show(); error

virtual ← O.show(); A

virtual and B not commented ← O show(); error

```
class A
{
Public:
void show()
{ cout << "A"; }
};
```

```
A *a = new D();
a → show();
```

Chained method Calls

day / date:

Virtual

```
class Test
{ int x,y
  Public:
  void show()
  {cout<<"A";}
  Test()
  { x=0; }

  Test& setx(int a)
  { x=a; return *this; }

  Test& sety (int b)
  { y=b;
    return *this;
  }
```

returning object

```
class B: virtual Public A
{
  Public
```

```
int main()
{
  Test obj
  ob.setx(5).sety(6);
```

- to make a chain of setter functions
- in calling in one line

ob.setx(5) returns ob

Scalars and Vectors

day / date:

$V_1 = (3, 5)$ $V_2 = (2, 4)$ $V_3 = V_1 + V_2$;

operator overloading : customize an operator behavior, called implicitly when there's atleast 1 object on the left side of the operator

```
class Vector
{
int n, y
Public:
vector( )
{ }

vector ( int n, int y)
{
this → n = n;
this → y = y;
}

void print( )
{ cout << n << y ; }

Vector operator + ( vector o)
{
Vector temp;
temp.n = n + o.n
temp.y = y + o.y
return temp;
}

Vector operator + ( int val)
{ Vector temp;
temp.n = n + val;
temp.y = y + val;
return temp;
}
```

```
int main( )
{
Vector v1 (3,5) ;
Vector v2 (2,4);
Vector v3 = v1 +v2;
```

same as: `v3 = v1.operator + (v2)`

calling object → argument → first parameter

- when adding objects compiler goes and finds if their is a behavior for this operator.
- if it's not there an error would come

```
Vector v4 (6,5);
v3= v1 + v2 + v4;
```

→ chained method call

co a → answer → co a

```
v3 = v1 + 2
```

calling object → argument

operator to be overloaded; keyword; argument

calling ob; argument

calling object; argument

Vectors and Vectors

day / date:

```
Class Vectors
{
int x, y;
public:
Vector()
{ }
Vectors(int x, int y)
{
  this -> x = x
  this -> y = y
}
Vector operator += (int val)
{
  Vector temp;
  temp.x = x + val;
  temp.y = y + val;
  return temp;
}
Vector operator ++ (xxx xxxx)        pre increment :
{
  Vector temp;
  temp.x = ++ x;
  temp.y = ++ y;
  return temp;
}
Vector operator ++ (int)             Post increment → to identify only, its not used anywhere
{ Vector temp;
  temp.x = x++;
  tempp.y = y++;
  return temp;
} friend Vector
```

```
int main()
{
Vector V1(2,5);
Vector V2(3,9)
// V2+=3;
++V1; 3, 6 → increment first
V1++; 3, 6 → used first
V2 = 3 + V1;
```

```
// Global scope
Vector operator+(int val, Vector o)
{ Vector temp;
  temp.x = o.x + val;
  temp.y = o.y + val;
  return temp;
}
```

KAGHAZ
www.kaghaz.pk

FRIEND

day / date:

Kills the point of encapsulation

```
Class Vector
{
    int x, y;
    Vector()
    { }
    Vector (int x, int y)
    {
        this -> x = x
        this -> y = y
    }
    friend void f();
        ↘ gives direct access to private variables
    friend class Test;
    friend Vector operator + (int, Vector);
        ↓ friend does not override
    friend ostream & operator<< (ostream &, Vector)
```

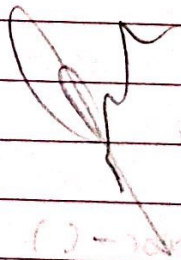
```
// Global Function
void f()
{
    Vector V;
    cout << V.x;
    cout << V.y;
}

Vector operator+(int val, Vector o)
{
    Vector temp;
    temp.x = o.x + val;
    temp.y = o.y + val;
    return temp
}
```

```
int main()
{
    Vector v1 (2,3);
    Vector v2 = 3+v1;
        first parameter, second parameter

    chain method
    cout << v2 << v1;
        ↓ insersion
    class object of class ostream
```

```
return type
ostream & operator << (ostream &o, Vector v)
    ↑ makes sure a copy isn't formed and exactly same object is sent.
{
    o << "(" << v.x << "," << v.y << ")";
    ↑ Object of ostream basically cout.
    return o;
}
```



CHAIN METHOD

day / date:

```
class Vector
{
    int x,y;
    public:
    Vector ()
    {       }
    Vector (int x, int y)
    {
        this -> x=x;
        this -> y= y;
    }
```

object reference — Pre increment

```
Vector& operator ++ ()
{
    ++x;
    ++y;
    return *this;
}
```

→ chain method

y → object values

→ Post increment

```
Vector operator & operator (int)
{
    x++;
    y++;
    return *this;
}
```

++ → chain method

```
Vector void operator - ()
    {
        x= x*-1 ;
        y= y*-1;
    }
```

```
int main()
{
    Vector v1 (2,3);
    Vector v2 (4,5);
    ++v1; (5,6)
    v2=v1++; (5,6)
    -v1; (-2,-3)
    [illegible];
```

insertion

Questions answers

day / date:

- when does an operator overload function gets called?

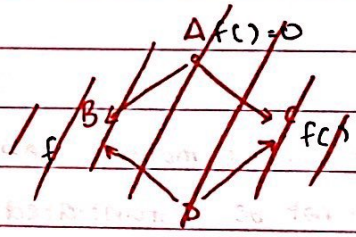
When atleast one operand is an object.

~~when operator overload~~

- when operator overload function is a member function it ~~must always~~ be non static.

- operator overload cannot change arity (no. of operands) of an operator.

- We cannot overload
  - .Dot
  - :: scope resolution
  - ?: if else
  - .*



- what happens when ~~you~~ we use abstract class with multiple inheritence

diamond problem which ~~can we solved~~ has no solution

- can dimaond problem occur when there are no functions at all

~~to~~ Due to variable overiding diamond problem can occur.

- what is data hiding

~~overriding of variables~~ variable overriding

- what is code hiding

function overriding, when parent function is overridden it is hidden.

- can constructors/destructors be virtual

no virtual constructors but there are virtual destructors

- what is factory function

PURE VIRTUAL / ABSTRACT

ABSTRACT CLASS: has atleast 1 pure virtual function, an object cannot be made of this class.

```
Class Employee
{
Public:
incomplete function/only definition
Virtual void work() = 0;
Pure Virtual
}
```

```
Class Pilot: Public: Employee
{
Public :
voicl work()
{ cout << Flies a plane"; }
```

```
Class Technitian: Public Employee
{
Public:
void work()
{ cout << "identifies faults" ; }
```

- used as a guidance for child class
- is only a blueprint
- overriding is necessary in abstract

- atleast one abstract function makes a class
- Abstract class can not be instantiated
- Abstract functions must be overriden

```
int main()
{
Employee e; X
Employee *e;
Pilot P;
e = &P;
e = new Doctor()
e -> work()
```

late binding

day / date:

```
Class Employee
{
  string empId;
  Public:
  virtual void work()=0;
Abstract (incomplete)/
Pure virtual
virtual void sign_in()
  {
  }
```

```
class Doctor: public Employee
{
Public:
void work()
{ cout<<"treats Patients"; }
```

```
int main()
{
Employee *e;
e = new Doctor();
e->work();
e->sign_in()
```

- error as ~~no~~ late binding and ~~no function available~~,
- no override done

GENERIC

day / date:

generic function keyword — keyword — generic parameter

```
template< class T> void Print(T a)
{ cout << a; }
```

stores data/parameter/placeholder — variable

```
int main()
{
Print(5)
Print2('K'),
Print3("C++");
```

```
void Point(int a)
{
  cout << a;
}
```

syntax replaced with generics

```
void Print(char a)
{ cout << a; }

void Print3(string a)
{ cout << a; }

void PrintAll(  a,   b)
{ cout<<a<<endl;
  cout<< b;
}
```

can be written instead of class

```
template <typename T> void Print(T a)
```

For more than one generic parameter

```
template < class T > void PrintAll(T a, T b)
{ cout << a << endl << b; }
```

PrintAll(3,5); ✓

PrintAll(3,"C"); ✗

data types need to be same

For more than one different generic Parameter

```
template < class T1, class T2 > void All(T1 a, T2 b)
{ cout<<a <<endl<< b; }
```

~~Print~~ All(3,"C"); ✓

For mix generic and nongeneric values

```
template <class T> void func(T a, int b)
```

- ~~function overloading is done on the basis of no. of parameters~~
- Constructor generics does not exist

day / date:

```
class MyClass
{
Public:
template<class T> void func (T par)
{ cout<<"1"; }

template <class T1, class T2> void func (T1 par1, T2 par2)
{ cout <<"2"; }
```

• function overloading done only when number of parameters are different

```
int main()
{
 func (8); 3
 func ("C++"); 1
```

Explicit Specialization: a function for a specific data type

```
void func (int p)
{ cout<<"3"; }
```

~~To sho~~ For redability that it is a function that uses explicit specilization.

```
template< > void func<int> (int p)
{ cout << "3"; }
};
template <class T> void sort (*Ta, int b)
{
 int i=0; T temp;
 for (i=0; i<b; i++)
 {
  if ( *(a+j) > *(a+j+1))
   {
    *temp = *(a+j);
    *(a+j*1) = *(a+j+1);
    *(a+j+1) = temp;
   }
 }
}
```

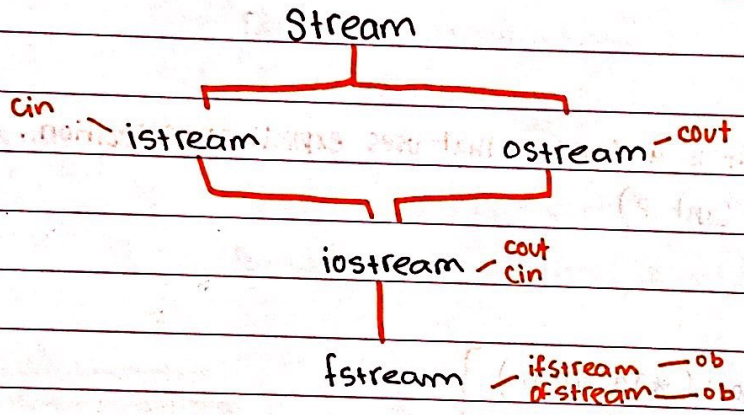
Bubble sort using Genetic

day / date:

Type of

```
int main()
{
    typedef int i;      Alias: can use i instead of int as a datatype
      i var = 5;
    typedef float f;
      f varf = 3.0f;
}
```

fstream



static_cast : to type cast a particular line

```
int main()
{
    char *word = "Hellow";
    cout << word;  Hello → dosent give address

    converts char word to void only for this line.
    cout << static_cast <void*> word;
    cout << word;  address value
```

day

Reading from file

get ( ) char single

get (c) char single

read (,) bulk text

writing to file

Put ( ) char single

write (,) bulk text

FILING

day / date:

```
int main ()
{
  string path = "D:\\OOP SPRING\\Files\\test0.txt";
```

OUTPUT BLOCK - to write to a file

```
ofstream o(Path);        // overwrite
o.put('T');
o.close;                 // always close so file doesnt get corruped
```

to write a string

```
char * inputText = "This is a string";
o.write(inputTxt, 7);    // number of char you want to write
o.close;
```

INPUT BLOCK - to read from a file

```
ifstream i(Path);
char c;
c= i.get();
cout << c;
i.close;
```

to read and display string

```
char *outputTxt = new char[10];
i.read(outputTxt, 7);
for (int i =0; i<7; i++)
  { cout << outputTxt[i]; }      // can also write cout << outputTxt;
i.close;
```

if x comes dont read it

```
char c,t;
ifstream i(Path);
while(!i.eof())
{
  i.get(t);
  if(!t=='x')
    { i.get(c);}
}
i.close;
```

To read using get

```
char c;
ifstream i(Path);
while(!i.eof())
{
  i.get();
  if(!i.eof())
  { cout << c;}      // buffer / flush in C
}
i.close;
```

day / date:

Random Acess : in write/read function a char ptr is always used as 1st argument

seek

Peek

ignore

default behavior
- write ptr at beg
- read ptr at beg
- overwrite mode
- Random Acess
  - seekg
  - seekp

```
class Employee
{
char * name;
int age;
Public:
Employee (){}
Employee ( char *n, int a)
{ name = n; age = a; }
```

spaces length ← sizeof (e1)

string length ← strlen(e1)

os.write ((char*) &e1, sizeof(e1));

type casting to convert obj to a char * ptr.

to append / enumerate }

~~os~~ ofstream os("file.txt", ios::app);

day / date:

* use getline to skip lines

ios :: beg

* cur ptr resets if we start reading after writing and vice versa

ios :: cur

ios :: end

i.seekg (4, ios::beg); skips first 4 char then reads

i.seekg (4, ios::cur); skips from where the current position of the pointer

i.seekg(-4, ios::end); reads the las 4 characters

o.seekp (2, ios::beg); skips first 2 char then outputs

KAGHAZ