RAYYAN SHABBIR

# OOP

Register

# ARRAY OF OBJECT

★ Same like we created array of any other data type.

Array object example:-

```
Algebra ar[5];
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| a= | a= | a= | a= | a= |
| b= | b= | b= | b= | b= |

To access any data member (a or b) at any index:

```
ar[0].a; // It will access 'a' member at '0' index.
```

- `0` → index number which we want to access
- `a` → data member which we want to access.

★ Similarly same for rest of these.

We use **"for loop"** for setting and printing data of an array.

∴ When no. of iterations are known we use "for loop".

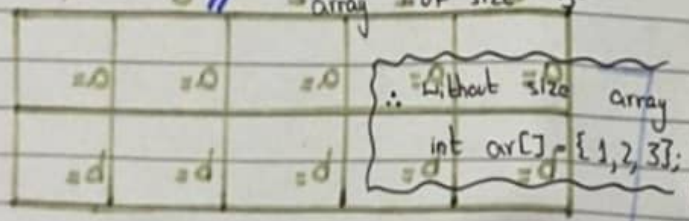
## WITHOUT SIZE OBJECT ARRAY

(We initialize the object array at creation time (with other objects) like this).

### ★INITIALIZING THE ARRAY OBJECT:

```
Algebra ar[] = {Algebra(), Algebra(1, 2), Algebra(3,4),
                Algebra(a[1]), Algebra()};
```

// It will created an object array of size 5.




**NOTE:**

★ integer array is initialized with integers i.e 1, 2, 3 etc.

★ object array is initialized with objects.
i.e Algebra(1, 2), Algebra(), Algebra(a[1]) etc.

∴

"good not"

### ↳ SETTING AND PRINTING DATA OF OBJECT ARRAY:

```
Algebra ar[5]; }// Here we are making ~~composite~~ thing.

for(int i=0; i<5; i++){

        ar[i].setData(i, i+1);

}

for(int i=0; i<5; i++){

        ar[i].print();
}
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| a=0 | a=1 | a=2 | a=3 | a=4 |
| b=1 | b=2 | b=3 | b=4 | b=5 |

## SEEING, PRINTING AND PASSING ARRAY TO MEMEBER FUNCTION.

**EXAMPLE:-** (To compare data of one object with whole array).

```
class Algebra {
    int a, b;

    public:
        Algebra () {
            a = b = 0;
        }

        Algebra (int a1, int b1) {
            a = a1;
            b = b1;
        }

        // Setting Data
        void setData (int a1, int b1) {
            a = a1;
            b = b1;
        }
        // Printing Data
        void print () {
            cout << a << b << endl;
        }
};

int main () {
    Algebra ar[5], obj2(3,4);

    for (int i=0; i<5; i++){
        ar[i].setData (i, i+1);
    }
    for (int i=0; i<5; i++) {
        ar[i].print();
    }

    if (obj2.testArray (ar, 5)) {
        cout << "Exist" << endl;
    } else {
        cout << "Does not exists" << endl;
    }
}
```

// comparing member function:

```
bool testArray (Algebra ar[], int size) {

    for(int i=0; i<size; i++){
        if ((a = ar[i].a) && (b = ar[i].b)) {
            return true;
        }
    }
    return false;
}
```

∴ '&' it checks bit-wise
'&' can also be used here.
'&&' can also be used here
'&&' is used for variables.

LECTURE #07

# POINTER OF CLASS

* Pointer of class is same as pointer of any other data type.

'&' ⟶ reference operator. (used for adress)
∴ '*' ⟶ dereference operator. (used for value)

* pointer will points to same type of memory that the pointer is itself.

```
Algebra obj1;
Algebra *p = &obj1;      // It will points to memory type Algebra.
   OR Algebra *p;
p = &obj1;
```

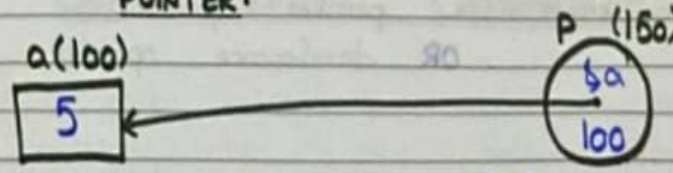
same
```
cout << &obj1 << endl;   //Obj1 adress
cout << p << endl;
```

```
cout << &p << endl;      // pointers adress
```

* pointer is not an object. pointer is only a reference.

∴
```
int a = 5;
int *p = &a;
```

↳ TO ACCESS DATA MEMBERS FROM POINTER.



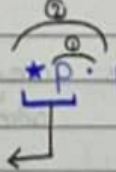

```
cout << *p ;   // 5
```

NOTE:

Precedence:
1) . (dot)
2) * (steric)

To access data members from pointer:

↳(error)




∴
*it will go to place of object.
*object is made *p

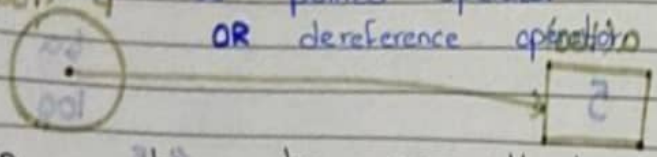
```
// This gives an error
/* Because '.' has higher precedence than '*'. so, here first from '.' print() func is access than it will be derefer by '*'. Means 'p' first access print() then derefer it which makes no sense. So, it is an error */
```

↳TO SOLVE ABOVE ERROR: (Two methods)

(1) (use braces)
```
(*p).print();      // 1st Method
```

(2) (use pointer operator)
```
p->print();        // 2nd Method
```
This one is best

→ it known as pointer operator
OR dereference operation



* Pointer can also be pass through functions.

NOTE:-

Algebra ar[]; // '[]' it holds only array adress.

Algebra *p; // '*' it holds array adress as well as adress of object.

see file → pointer_of_class.cpp

# CLASS WITH CONSTANT DATA MEMBER

* We make constant variables or data members when we further don't want to make any changes in our data.

* Const variable should be initialize at creation time.

const int S; // error.

const int S=5; // valid

∴ Convention for const data members:
↳ 1st letter of name should be capital
example:
const int Sam = 5;
const float You = 6.6;

* Constant (const) data member of class should /must be initialize in the constructors (at the time of creation of the object).

* Constant data members are very important in any (our) class.

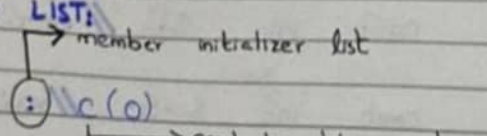
* They are used /made for fix things that we want to make i.e value of 'pi', 'e' etc.

## ↳ HOW TO INITIALIZE CONST DATA MEMBERS:

* const data members must (always) initialize in constructers only
* Constant data members can only be initialized in member **initializer list** of constructor.

like:

**MEMBER INITIALIZER LIST:**




* Member initializer list can initialize both constant and non-constant data member.
* Const must be initialize in member initializer list only.
* Const data member cannot be initialize inside body of constructor.

**EXAMPLE:-** (How to initialize const data member)

```
class Algebra {
    int a, b;
    const int c; //This need to be forcefully initialize in the constructer.
public:
                    → member initializer list.
    Algebra () :c(0)
    {
        a=b=0;
    }

    Algebra (int a1, int b1, int c1) : c(c1)
    {
        a= a1;
        b= b1;
    }

    Algebra (const Algebra &obj) : c(obj.c)
    {
        a=obj.a;
        b=obj.b;
    }

    void print () {
        cout << a << b << c << endl;
    }
};
```

```
int main () {
    Algebra obj1, obj2(1, 2, 3);
    Algebra obj3 (obj2);

    obj1.print();
    obj2.print();
    obj3.print();

    return 0;
}
```

**OUTPUT:-**

```
0   0   0
1   2   3
1   2   3
```

see File → constant_datamember.cpp

* Seters cannot be used for constant members. (because they cannot set/initialize again)

* geters can be used for constant members. (Here there value only get not setting)

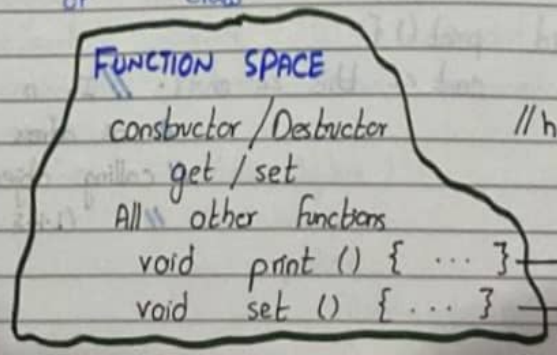
```
cout << sizeof(obj1) << endl;    // 12
cout << sizeof(Algebra) << endl; // 12
```
} same

:. It means Algebra class g sizeof any on of its objec

## FUNCTION SPACE

↳ member functions are not part of object. They exist only in memory.

* Member functions exists in separate space. This space is known as **function space** of class.




// here reference of objec is passed.

* Objects and functions exists in separate memory. This, objects go to memory of function by going through address (reference).

# * this pointer (impt)

"This pointer" exists in non-static member function of class.
It holds the adress / reference of calling object (L.H.S object)

This, type of "this pointer" is same as data type of calling object. (L.H.S object)

**EXAMPLE :-** (To print this pointer)

```
class Algebra {
    int a, b;
    public:
        void print() {
            cout << this << endl; // It is same
                                  // as adress of
                                  // calling object
                                  //    (L.H.S object)
        }
};

main() {
    Algebra obj1, obj2;
    obj1.print();
    cout << &obj1 << endl;      // Adress of obj1
    return 0;
}
```

**OUTPUT** will be same for 'this' as well as '&obj1'.

see file → this pointer_basic.cpp

**NOTE:-**

* 'this' pointer can access data members as well as data functions.

## ↳ USE OF "this pointer":

* Basically "this pointer" is used when we pass arguments in any member function of the class to set data, and these arguments have same name as name of the data members of the class. So, then while (initializing) assigning them with each other, then it will became difficult for us and comiler to distinguish between these arguments and data members.

i.e

```
class Algebra {
    int a, b;
    public:
        Algebra(int a, int b) {
            a = a;      // This gives an
            b = b;      // error.
        }
};
```

next page solution of this error

So, here comes job of "this pointer". If we use "this pointer" then it will not give any error.

```
class Algebra {
    int a, b;
public:
    Algebra(int a, int b) {
        this->a = a;
    }
};
```

By using this pointer it became easy to distinguish.

The data members with this pointer is written tells that these data members are data members of the calling object (L.H.S), and 'a' and 'b' are values which are assigning to them.

**EXAMPLE:-** (Use of "this pointer")

```
#include "iostream"
using namespace std;

class Algebra {
    int a, b;
    const int c;

public:
    Algebra () : c(0)
    {
        a = b = 0;
    }

    Algebra(int a, int b, int c1) : c(c1)
    {
        this->a = a;
        this->b = b;
    }
    void print () {
        cout << a << b << endl;
    }
};
int main() {
    Algebra obj1, obj2( 2, 4, 5);
    obj2.print();
    return 0;
}
```

file → thispointer_class.cpp

## CONSTANT OBJECT:

* Constant objects are used for securing the data of our objects.
* When we don't want to change the data of the objects, we make them constant.

↳ Constructors can simply be called for constant as well as non-constant objects. We don't need to make any changes in constructor.

syntax for const objects:

```
const Algebra obj1(1, 2, 3);   // Now we cannot change anything of the object.
```

* Now L.H.S object is secured.

```
/*
obj1.print();
obj1.getA();
*/
```

these gives an error bcz print and get function can also change obj data. So, these cannot simply call for const objects.

NOTE:-

- Constructor can simply be called for constant as well as non-constant objects.
- No, other function can cal constant objects simply.

## ↳MAKING OTHER MEMBER FUNCTIONS VALID TO CALL CONSTANT OBJECTS:

* If we want to make other functions valid to call constant objects. Then we have to make type of function same as type of the object. (means making function constant).
↳ But these functions can also not change the data of the object.
↳ they are used simply for printing, geting purposes etc.

Syntax:-

```
data type of function   name of function   ()  const  {
                                                    ↳ const keyword
```

* Now, if we want to use print() function to print the data of the constant object. We have to make the type of print function same as type of the object.

* Constant function are used for this purpose.

// Constant function only exists in class.
// Not globally.

```
void print() const
{
    a = 5;   // error bcz data of const object cannot be changed in const function also.
    cout << a << b << endl;
}
```

* Constructor cannot be constant.

**EXAMPLE:-** (How member function can access data [not changing] of const object).

```
class Algebra {
    int a, b;
public:
    Algebra(int a1, int b1) {
        a = a1;
        b = b1;
    }

    // constant print function
    void print() const
    {
        // a = 5 → ERROR
        cout << a << b << endl;
    }
};
int main() {
    const Algebra obj1(1, 2);
    obj1.print();
    return 0;
}
```

see file → constant_object.cpp

**NOTE:-**

* Constant objects cannot be accessed by non-constant function.
* Constant objects can only be accessed by constant functions.

---

LECTURE#08

## OPERATOR OVERLOADING

We have multiple operators on which we work.

(i) +, -, *, /, % } binary operators (work for two operands)

(ii) +=, -=, *=, ...

(iii) =, ≥, ≤, ==, !=, ... } assignment operator / comparison operator

(iv) &&, ||, !, ...

∴ * If we code
- ✓ int + int
- ✓ Float + Float
- ✓ char + char

} These are valid. It will run.

As, '+' operator works same for these different (int, float, char) data types. Thus, '+' operator is called operator overloading for these data types.

## "BINARY '+' OVERLOADED OPERATOR:"

* But these binary operators i.e (+, -, *, /, %) will not simply work for abstract data type (that we made by ourselves) (class).

```
Algebra obj1, obj2;

obj1 + obj2; // currently this is ERROR
```

* We cannot do this here.
* We have to do some steps.
* We have to make operator overloading for '+'.

↳ Because '+' operator is not overloading for our data type. (Algebra)

Now, if we want that '+' operator do same work/operation for our data type. It can be done by operator overloading.

** Operator overloading is basically a function that we make in our class for any specific operator i.e (+, -, ...)

∴ Basically, +, - etc are functions having fix symbols

**SYNTAX:** FOR MAKING '+' OVERLOADING OPERATOR.

Return type operator + ( . . . ) {

Return type must have to be written here (compulsory) while making operator overloading for any operator.

operator + → symbol of which operator we wanted to perform.

operator → function name (we cannot give this name of our choice)

}

→ **FOR CALLING THIS FUNCTION:**

(i) obj1 + obj2; ✓ works

OR

(ii) obj1 . operator + (obj2); ✓ works

∴ Now it will add two objects

**EXAMPLE:-**

Algebra operator + ( const Algebra &obj ) const

{

bcz we don't want to change R.H.S object.

bcz we don't want L.H.S object to be changed.

∴ Algebra temp;

temp.a = a + obj.a;
temp.b = b + obj.b;
return temp;

} ALGO 1

∴ We only want changes to make in temp object

OR

Algebra temp ( a + obj.a , b + obj.b );
return temp;

} ALGO 2

OR

*return Algebra ( a + obj.a , b + obj.b );

} ALGO 3 (It will make un-name object itself)

}

int main () {

(obj1 + obj2) . print ();

OR

Algebra r = obj1 + obj2;
r. print ();

OR

(obj1 . operator + (obj2)) . print ();

return 0;

}

---

## * CAS CADDING FOR '+' Operator.

When more than two operands (variables) are adding '+' operator adds them separately 2 by 2 from Left → to → right. First, add two left most operands and save their data (result) in temp object, then add this temp object with the left & 3rd most operator and save their result in temp object, and then repeat this adition process untill all the operator are added.

```
int a, b, c, d;
a + b + c + d;          // It will add 2-by-2
(a+b) + c + d;          // Associativity of '+' is Left → right
(temp + c) + d;
    (temp + d);
    temp;
```

* This concept is known as Cas cadding.

* Cas cadding is not possible unbill nothing returns (i.e temp in above case).

* Cas cadding will be done /possible only when something returns.

↳ THUS, FOR ADDING '3' OBJECTS:

```
Algebra r = obj1 + obj2 + obj3;
```

∴ (obj1 + obj2) + obj3
(temp + obj3)
temp;
Here '+' operator called for 2 times

↳ For '3' ~~times~~ objects by other method.

```
(obj1 . operator + (obj2)) . operator + (obj3);
```

**EXAMPLE:- (FOR '+' OVERLOADING OPERATOR)**

```
class Algebra {
    int a, b;
public:
    Algebra () {
        a = b = 0;
    }

    Algebra (int a1, int b1) {
        a = a1;
        b = b1;
    }

//OVERLOADED binary '+' operator

    Algebra operator + (const Algebra &obj) cons
    {
        Algebra temp(a + obj.a, b + obj.b);
        return temp;
    }

    void print() {
        cout << "a: " << a << "\nb: " << b <
    }
};
```

```
int main () {
    Algebra obj1(1, 2), obj2(4, 5), obj3(7, 9);
    Algebra r = obj1 + obj2 + obj3;
    r.print ();
    return 0;
}
```

**OUTPUT:**

a: 12
b: 16

see file → +_operator_overloading.cpp

## "BINARY '=' OVERLOADED OPERATOR"

obj1 = obj2;

**SYNTAX:** FOR MAKING '=' OVERLOADING OPERATOR.

```
Return type operator= ( . . . ) {
}
```

↳ FOR CALLING THIS FUNCTION:

(i) obj1 = obj2; ✓ works

OR

(ii) obj1.operator=(obj2); ✓ works

∴ Now, it will assign data of obj2 into obj1.

EXAMPLE:-

```
class Algebra {
    int a, b;
    const int c;
public:
```

**EXAMPLE:-**

```
class Algebra {
    int a, b;
    const int c;
public:
    Algebra operator = ( const Algebra &obj )
    {
        if( this != &obj) {
            a = obj.a
            b = obj.b;
            // c = obj.c; //ERROR
        }
        return *this;
    }
};

int main () {
    Algebra obj1(1, 2, 3), obj2(4, 5, 6);
    obj1 = obj2;
    obj1.print();
```

∴ bcz we don't want R.H.S object to be changed.

∴ L.H.S object can be changed. Bcz we are (copy) assigning data of R.H.S to L.H.S object. Thus, L.H.S object will be updated (changed).

∴ We use this condition bcz we want to avoid/(protect) from self assignment. i.e; obj1 = obj1

∴ 'C' is a constant. So, 'C' cannot be updated

bcz here we want to return L.H.O. As, 'this' is a pointer of L.H.O. So, it returns L.H.O adress. So, if we want to convert it to an object we derefer it by using *. Now, by '*this' it will return value of L.H.O (which is updated).

**OR**

```
    obj1.operator = (obj2)
    obj1.print();

    return 0;
}
```

## * CASCADING FOR '=' OPERATOR.

```
a = b = (c = d);
a = (b = temp);
    (a = temp);
```

// Associativity of = is Right → Left.

```
obj1 = (obj2 = obj3);

(obj1 = temp);
    (temp);
```

* It means we must have to return something.

↳ If we make the type of our "operator= function" to void:

then, `obj1 = obj2 = obj3 = obj4;` //ERROR ∴ bcz we use data type void which returns not

OPERATOR)

```
    int c;

Algebra(int a1, int b1, int c1) : c(c1)
{
    a = a1;
    b = b1;
}

//OVERLOADING = OPERATOR
Algebra operator= (const Algebra &obj) {
    if (this != &obj) {
        a = obj.a;
        b = obj.b;
    }
    return *this;
}

void print () {
    cout << a << b << c << endl;
}
```

```
int main () {
    Algebra obj1(1, 2, 3), obj(4,5,6), obj3(7,8,9);

    obj1 = obj2 = obj3;

    obj1.print();
    obj2.print();
    obj3.print();

    return 0;
}
```

**OUTPUT:-**

obj1 : 7, 8, 3
obj2 : 7, 8, 6
obj3 : 7, 8, 9

;' 'c' values cannot be changed bcz c is a constant
c remains same (not updated)

=_operator_overloading.cpp ← See file

**CONCEPT:-**

Compiler gives us assignment operator by itself.
But, if we use
obj1 = obj2;
without making any operator= overloaded then it gives us an error.

because 'c' data member is constant.

★ When we have any constant data member then we have to use overloaded assignment operator.

In for performing assignment

obj1 = obj2;

In overloaded assignment operator 'c' constant don't need to be initialize.

## "ARITHMETIC ASSIGNMENT '+=' OPERATOR"

obj1 += obj2

∴ It is not an assignment. It is an addition. i.e

obj1 = obj1 + obj2.

**SYNTAX:** FOR MAKING '+=' OVERLOADED OPERATOR

Returntype operator+= ( ... ) {

}

↳ FOR CALLING THIS FUNCTION:

(i) obj1 += obj2; ✓ works

(ii) obj1.operator(obj2); ✓ works

∴ Now it will perf
obj1 = obj1 + ob

## ★ CAS CADDING FOR '+=' OPERATOR:

```
a += b += c;

Algebra obj(1, 2, 3), obj2(4, 5, 6), obj3(7, 8, 9);
obj1 += obj2 += obj3;
obj1.print;
```

**OUTPUT:-**

obj1: 12, 15, (3) → 'c' constant cannot be changed.

## EXAMPLE:- FOR '+=' OVERLOADING OPERATOR

```
class Algebra {
    int a, b;
    const int c;

public:
    Algebra (int a1, int b1, int c1) : c(c1)
    {
        a = a1;
        b = b1;
    }

//OVERLOADED += OPERATOR
// R.H.S object must not be changed here
// L.H.S object should be changed here bcz we are making addition in it.
Algebra operator+= (const Algebra &obj) {

        a = a + obj.a;
        b = b + obj.b;
        OR
        a += obj.a;
        b += obj.b;

        return *this;
}
void print() {
        cout << a << b << c << endl;
```

```
int main () {
    Algebra obj1(1, 2, 3), obj2(4, 5, 6);
    obj1 += obj2;
    obj1.print();
    return 0;
}
```

**OUTPUT:**

obj1: 5, 7, 3 → constant cannot be changed.

see file → `+=_operator_overloading.cpp`

LECTURE #09

## → OVERLOADING COMPARISION OPERATOR:

==, !=, <, >, <=, >= } b/w two operators

* These are comparision operators, so, they ('==') return true or false. Thus, there return type must be bool.

* **OPERATOR OVERLOADING:** Operator overloading is a mechanism which makes already existing operator capable to perform those specific operations for our data type.

## "BINARY COMPARISION '==' OPERATOR"

```
obj1 == obj2;
```

**SYNTAX:** FOR MAKING '==' Overloading operator.

```
returntype operator == ( ... ) {
```
↑ must be bool:

```
}
```

↳ FOR CALLING THIS FUNCTION:

(i) (obj1 == obj2); ✓ works

(ii) (obj1.operator==(obj2)); ✓ works

∴ It will compare obj1 and obj2

EXAMPLE: ('==' Overloading operator function)

```
bool operator == ( const Algebra &obj ) const
```

R.H.S object must not be changed. ← bcz we are making only comparison here not updation → L.H.S object must not be changed

```
{
    if ((a == obj.a) && (b == obj.b) && (c == obj.c))
        return true;
    return false;
}
```

L.H.S objects data member → a; R.H.S object data member → obj.a

OR

```
    return ((a == obj.a) && (b == obj.b) && (c == obj.c));
}
```

↳ TO CALL '==' operator in main:

```
cout << (obj1 == obj2) << endl;
```

∴ OUTPUT
0 → False
1 → True

★ Can Cadding is also possible for '==' overloading operator.

# "BINARY '!=' OVERLOADING OPERATOR"

★ We can make / convert '==' to '!=' operator.
By simply calling '==' function in '!=' function and put a '!' sign with it.

EXAMPLE: ('!=' Overloading operator function)

```
bool operator != ( const Algebra &obj ) const
{
    return !(*this == obj);
}
```

↳ here we also not changing L.H.S and R.H.S object. We simply comparing them.

pointer of L.H.S object. We de refer it by '*' to compare its value with with R.H.S object.

↳ R.H.S object.

↳ TO CALL '!=' operator in main:

```
cout << (obj1 != obj2) << endl;
```

OR

```
cout << (obj1.operator!=(obj2)) << endl;
```

## EXAMPLE (OF '==' and '!=' overloaded operators)

```
#include "iostream"
using namespace std;

class Algebra
{
    int a, b;
    const int c;
public:
    Algebra (int a1, int b1, int c1) : c(c1)
    {
        a = a1;
        b = b1;
    }
    // OVERLOADED == OPERATOR
    bool operator == (const Algebra &obj) const
    {
        return ((a == obj.a) && (b == obj.b) && (c == obj.c));
    }
    // OVERLOADED != OPERATOR
    bool operator != (const Algebra &obj) const
    {
        return !(*this == obj);
    }
int main () {
    Algebra obj1(1, 2, 3), obj2(4, 5, 6);
    cout << (obj1 == obj2) << endl;
    cout << (obj1 != obj2) << endl;
    return 0;
}
```

**OUTPUT:**

0 (False)
1 (True)

see file → == and != _overloading_operator.cpp

## Unary Overloaded Operators

- , + , + + , - - , ! , ~

pre increment, post increment, pre decrement, post decrement

∴ '-' and '+' works for both binary as well as unary operator.

As binary:

```
a = a+b;
b = b-a;
```

As unary:

```
a = +b;
b = -a;
```

## Unary Overloaded Operator

- , + , + + , - - , ! , ~

pre increment, post Increment, pre decrement, post decrement

∴ '-' and '+' works for both binary as well as unary operator.

As binary:

```
b = b + a;
b = b - a;
```

As unary:

```
b = +a;
b = -a;
```

# '-' As Unary Operator Overloading

Algebra r = -obj1;
↳ L·H·S obj

**SYNTAX:** FOR MAKING '-' UNARY OVERLOADING OPERATOR

- As there is creating (temp) unknown object for saving '-a' value which then be stored in 'b'. As, temp is creating thus, return type must exists and it will be same as type of the object (bcz temp is also same as type of obj).

∴ int a = 2, b = 0;
temp
b = -a;
b = temp;
b = -2
a = 2
temp;

of class
↓ type
returntype operator - ( )
empty bcz we don't want any R·H·S object We only want L·H·S object and work for L·H·S object.

- **FOR CALLING THIS FUNCTION:**

Algebra r = -obj1;
↳ L·H·S object.

**OR** Algebra r = obj1 . operator - ();
↳ L·H·S object.

(i) Algebra obj1(1, 2, 3), obj2(4, 5, 6);

obj2 = -obj1;
↳ L·H·S object.

∴ obj2 = -obj1
obj2 = temp;
temp;

**OR**

obj2 = obj1 . operator - ();
↳ L·H·S object.

(returns behaviour same '+')

- As here value of '-obj1' is also assigning to obj2 thus, '=' overloading operator is also needed to write.

**EXAMPLE:** ('-' unary overloaded operator).

empty bcz we only want to use 1 object: obj1 (L·H·S object)
Here we don't want to change L·H·S object, we only want to return its value with a negative signs.

```
Algebra operator- ( ) const
{
    ∴ return Algebra(-a, -b, -c);
}
int main () {
    Algebra obj1(1, 2, 3);
    Algebra r = -obj1;

    obj1.print(); // 1, 2, 3
```

∴ L·H·S object i.e obj1 did not changed. Only its value with '-' sign

## '+' AS UNARY OPERATOR OVERLOADING

```
Algebra r = +obj1;
```
→ L·H·S object.

**SYNTAX:** Will be same as '-' unary overloaded operator. Only name of Function will be changed (i.e operator+).

**EXAMPLE:** ('+' unary overloaded operator)

```
Algebra operator + () const
{
    return Algebra( +a, +b, +c);
}

int main () {
    Algebra obj1 (1, 2, 3) , obj2(4, 5, 6);

    obj1 = +obj2;
```
→ L·H·S object

∴ L·H·S object did not change.

```
    obj1.print(); // 1 , 2, 3
    obj2.print(); // 1, 2, 6

}
```

**EXAMPLE:-** (OF '+' and '-' unary overloaded operator)

```
class Algebra {
    int a, b;
    const int c;

public:
    Algebra (int a1, int b1 , int c1) : c(c1)
    {
        a= a1;
        b= b1;
    }

//OVERLOADED '-' UNARY OPERATOR
Algebra operator-() const
{
    return Algebra (-a, -b, -c);
}

//OVERLOADED '+' UNARY OPERATOR
Algebra operator+() const
{
    return Algebra (+a, +b, +c);
}
```

```
//OVERLOADED = OPERATOR
Algebra operator=(Algebra &obj)
{
    if ( this != &obj) {
        a = obj.a;
        b = obj.b;
    }
    return *this;
}

void print () {
    cout << a << b << c<<endl;
}
};

int main () {

    Algebra obj1(1, 2, 3), obj2(4, 5, 6);

    Algebra r= +obj1;

    obj2 = -obj1;
    //OR  obj2 = obj1.operator-();

    obj1.print();    // 1, 2, 3
    r.print();       // 1, 2, 3
    obj2.print();    // -1, -2, 6
    return 0;
```

∴ '=' overloaded operator is necessary here bez when -obj1 or +obj1 is assigning to other object. Then, this function will be called.
obj2 = -obj1;
Its '=' (function) presence is necessary.

∴ L-H-S object did not change.

constant (c) did not change.

see file → + _ - _ unary_overloading_operator.cpp

## PRE INCREMENT OR POST INCREMENT:

(method) ++a;
a++;
} works same when not assign, to any other variable/object.

different
b = ++a; } pre increment 'a' and 'b' both values will be incremented.
∴
b = a++; } post increment 'a' value will be incremented but 'b' value will be same as orginal 'a'.

★ post/pre increment and post/pre decrement must have a return type.
The return type will be same as type of object.

b = ++a;  (temp)
b = temp;  (temp)
∴ As value is returning to temp (unnamed) object.
This, these function have return type same as type of object.
temp

★ In post/pre increment and decrement value of L-H-S object must be changed. So, we

# PRE INCREMENT

obj2 = ++obj;
↳ L.H.S object.

**SYNTAX:** (FOR making pre increment function)

returntype operator ++ ( ) ← empty bcz we only deal with L.H.S object.
{
}
↑ of class type

**FOR CALLING THIS FUNCTION:**

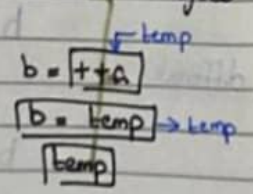
obj2 = ++obj1;
↳ L.H.S object 1st incremented and then assign to obj2.

**OR**

obj2 = obj1.operator++();

means both obj1 and obj2 will be incremented.

∴ b = ++a




# EXAMPLE:- ( PRE INCREMENTED FUNCTION)

bcz L.H.S object will be returned.

```
Algebra operator ++ ()
{
    ++a;
    ++b;
    return *this;
}
```

L.H.S object is not constant. Bcz we have to update its value (increment).

L.H.S objects 'a' and 'b' are updating.

We are returning L.H.S object so, that it will assign to other object.

```
int main () {
    Algebra obj1(1, 2, 3), obj2(4, 5, 6);
    obj2 = ++obj1;
                ↳ L.H.S object
    obj1.print(); // 2, 3, 3
    obj2.print(); // 2, 3, 6
}
```

★ In pre increment both L.H.S object and R.H.S object will be updated (changed).

# POST INCREMENT

obj2 = obj1++;

obj1 → L·H·S object.

**SYNTAX: (FOR MAKING POST INCREMENTED FUNCTION)**

★ To make difference b/w pre and post incremented function. We must have to pass (int) in the parameter while writing post incremented function.

same as type of object.

returntype operator++ ( int )

- known as "dummy integer".
- Its name is not necessary.

{

}

temp

∴ b = a++;

b = temp → temp

temp;

★ Thus, it must have a return type. same as type of object.

→ **FOR CALLING THIS FUNCTION:**

obj2 = obj1++;

OR

obj2 = obj1.operator++(0);

→ ∴ L·H·S obj assigned to obj2 then incremented.

means obj2 will not change (increment) only obj1 will be incremented.

**EXAMPLE: (POST INCREMENTED FUNCTION)**

An object is returning

L·H·S object is not constant bcz it will be updated.

```
Algebra operator++ ( int )
{
    Algebra oldState ( *this);
    • OR
    Algebra oldState (a, b, c);

    ++a;
    ++b;

    return oldState;
}
```

object

∴ oldState is object that is created to store the original / old value of L·H·S object.

L·H·S objects 'a' and 'b' are updating

but returning the old state of L·H·S object.

```
int main (){

    Algebra obj1(1, 2, 3), obj2(4, 5, 6);

    obj2 = obj1++;

    obj1.print();   // 2, 3, 3
    obj2.print();   // 1, 2, 6

}
```

L·H·S object

★ In post increment value, of L·H·S objects will be incremented only. Value of R·H·S object will be assigne (remain same) the old value of L·H·S obje

**EXAMPLE:-** (PRE AND POST INCREMENTATION)

```
class Algebra {
    int a, b;
    const int c;

public:
    Algebra (int a1, int b1, int c1) : c(c1)
    {
        a = a1;
        b = b1;
    }

//OVERLOADED PRE INCREMENT

Algebra operator ++ ()
{
    ++a;
    ++b;
    return *this;
}
//OVERLOADED POST INCREMENT
Algebra operator ++ (int)
    Algebra oldState(*this);
    ++a;
    ++b;
    return oldState;
}
```

```
// OVERLOADED = OPERATOR
Algebra operator = (const Algebra &obj)
{
    if (*this != &obj) {
        a = obj.a;
        b = obj.b;
    }

    return *this;
}
void print () {
    cout << a << b << c << endl;
}
};
int main () {
    Algebra obj1 (1, 2, 3) , obj2(4, 5, 6);

    obj2 = ++obj1;

// PRE INCREMENT
obj1.print ();    // 2, 3, 3
obj2.print ();    // 2, 3, 6

obj2 = obj1++
// POST INCREMENT
obj2.print();     // 3, 4, 3
obj2.print();     // 2, 3, 6
}
```

FILE → post_and_pre_increment_operator_overloading.cpp

## "NOT OVERLOADED OPERATOR"

(i) .

(ii) ::

(iii) .*

(iv) Type

Size

?:

* In C++ 6 operators cannot be overloaded.

## "this pointer"

* "this" pointer does not exists in global functions.

* If "this" pointer does not exists in our compiler than we must have to pass objects in parameter list.

* If "this" pointer does not exists then there exist no L-H-S object. All the parameters objects can only pass through parameters.

# "friend" METHOD / FUNCTION

- We use "friend" method to make a global function which holds data of data membess of our objects.

→ IF WE NOT USE friend ~~function~~ method. (And create a global function)

```
void test () {
    Algebra obj(1, 2, 3);

    cout << obj.a << endl;  // In-accessible
                            // This gives an error.
}
```

- TO SOLVE THIS PROBLEM: (USE OF friend)

We must have to write decleration of this global function (void test) with friend keyword in our class.
Then this global function became friend of our class and access all data memba and date function of our class.

Syntax → next Page

SYNTAX: (for decleraning friend method)

```
friend return_type_of_global_function global_function_name ( parameters data type if any with name. (But name is not necessary) );
```

friend keyword

EXAMPLE::

```
class Algebra {
    int a, b;

public:
    Algebra(int a1, int b1) {
        a = a1;
        b = b1;
    }

    friend void test();
};

void test() {
    Algebra obj(1, 2);

    cout << obj.a << obj.b << endl;
}

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

★ Now, void test (global function) can access all the data members and can run without any error

★ But this breaks encapsulation and data hiding

## ★ MAIN PURPOSE OF "friend" method :

★ The main purpose of friend method is that when we perform any operation (i.e +, -, >>, << etc) b/w two different data types ; like

★ We use friend method when L.H.S object has other than class object.
i.e

int, other than class type ← int + Algebra; → class object.  ✗ ERROR

then we cannot access them through operator overloaded function of our class.

L.H.S → int.operator+ (Algebra); ✗

So, we create a global function to perform this operation, and gives this global function access to our class data members / functions by writing its (global function) decleration in our class with friend keyword.

## ★ REAL USE OF "friend" :

★ We use friend method when we want to print or take data about our data members of objects (of our class) outside the class (not in class), in main or any global function.

cout << obj1;

## ★ MAKING "Stream insertion" operator friend OF our class

stream insertion operator ↓

cout << obj1;

ostream type ← cout ↓ global

↳ of our class type (Algebra).

★ Here '<<' operation is performing b/w ostream (L.H.S) and our class object.
So, we cannot perform this '<<' operation by operator overloading i.e.

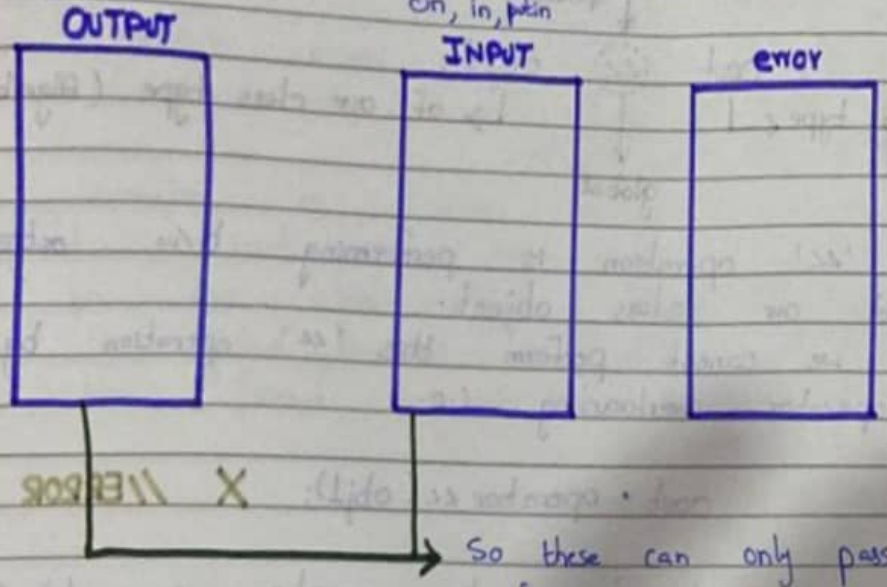
cout.operator<<(obj1); ✗ //ERROR

This, we must have to create a global function (and declare it in our class with friend keyword) and pass these cout and obj1 through its parameter to that global function.
Then this '<<' operation will be performed.

★ **NOTE:**

- "ostream" is a complete library. So, we cannot pass it through value. We pass it reference so, nothing in this library could be change.
- "object" is complex data type. So, we cannot pass it through value. We have to pass it through by reference.

**THREE STREAM OF EVERY PROGRAM**




**EXAMPLE** (Stream - insertion operator)

```
class Algebra {
    int a, b;
public:
    Algebra (int a1, int b1) {
        a = a1;
        b = b1;
    }
    friend ostream& operator<< (ostream& c, const Algebra& f);
};

ostream& operator<< (ostream& out, const Algebra &obj) {
    out << obj.a << obj.b << endl;
    return out;
}
int main () {
    Algebra obj1 (1, 2), obj2 (3, 4);
    cout << obj1 << obj2;
}
```

Same { `cout << obj.a << obj.b << endl;` OR `out << obj.a << obj.b << endl;` }

- name not necessary
- 'obj' and ostream ∴ must be pass through reference
- We make it constant bcz we don't want to change Algebra object.
- We only want to print its value
- ∴ return type of the fuction is ostream bcz (b/c we are returning a ostream object i.e (out).
- (ii) Cas cadding will be possible by this. Cas cadding requires a return value always.

return cout << obj1 << obj2;

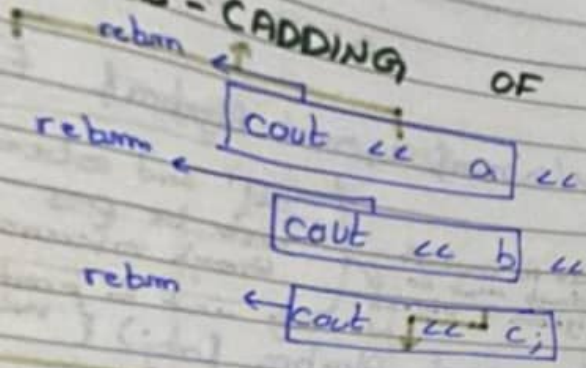
return cout << obj2;

operator << ( cout , obj1);

like:

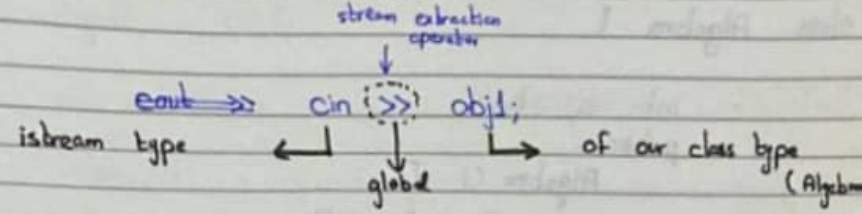
cout << obj1 ;

## CASE - CADDING OF STREAM - INSERTION OPERATOR




see file → friend_stream_insertion_operator.cpp

## * MAKING "Stream Extraction" friend of our class:




* Here '>>' is performing b/w istream (L.H.S) and our class object.
So, we cannot use operator overloading.
cin.operator >> (obj1); X //ERROR

* So, we create a global function and make it friend of our class.

**NOTE:**

* "istream" is a complete library. So, we pass it by reference only.
* "object" is also pass by reference.

**EXAMPLE:** ("friend" "istream" extraction operator)

```
class Algebra {

    int a, b;
    public:
        Algebra () {
            a = b = 0;
        }

    //making istream extraction >> friend.
    friend istream& operator >> (istream& c, Algebra& f);
};
```

**//OVERLOADING STREAM-EXTRACTION OPERATOR**

```
istream& operator >> (istream& in, Algebra &obj) {

    cout << "Enter a: ";
    in >> obj.a;

    cout << "Enter b: ";
    in >> obj.b;

    return in;
}
```

It is not constant bcz we want to change it (get data).

∴ This func has return type istream bcz we are returning 'in' which is object of istream

```
int main () {

    Algebra obj1;

    cin >> obj1;

    return 0;
}
```

* BACK-END CALL (COMPILER) OF THIS GLOBAL FUNC:

```
operator >> ( cin , obj1);
```

* We CALL LIKE:

```
cin >> obj1 >> obj2 >> obj3;
```

See file → friend_istream_extraction_operator.cpp

**NOTE:**

* Every operator can be globally overload by using friend (operator) keyword.

LECTURE #10

# STATIC DATA MEMBERS

## ↳ STATIC VARIABLES:

- static variables maintain its state.
- There life is global life, Their scope is local.
- Static variable is automatically initialize to zero (default), if we don't initialize it.

EXAMPLE:-

```
void testStatic()
{
    static int s=0;
    s++;
    cout << s;
}

testStatic();   // 1
testStatic();   // 2
testStatic();   // 3
```

## ↳ STATIC (DATA) MEMBERS: (part of class)

* Static have same context in OOP as well.
* These are called data member of class. They are not data member of objects.
* It is not became part of our object (don't link with our object)
* All access identifiers apply on this.
* Only one copy of static data member exists in memory, and exists in our class.
* Static are not part of objects, only objects can access them.

  like: in class → static int c;

  object access it in main → obj.c;

* Every object of class can access static (share b/w every object)

**EXAMPLE:** (static data members are not part of our object).

```
class Algebra {
        int a, b;
        static int c;

    public:
        Algebra() {
            a = 0;
            b = 0;
        /*
            Algebra::c = 0;
                OR
            c = 0;
        */
        }
};
int main() {
        Algebra obj1;
        cout << sizeof(obj1) << endl;
}
```

These both gives error bcz static data member is not part of object. So, it cannot be initialize by the object in constructor. like others data membos

**OUTPUT:-**

8 → bcz static does not part of object. So, it is not include in size of object.

## → TO ACCESS / USE STATIC:

Two methods:

**(1) BY CLASS NAME**

syntax:-

```
Algebra :: s ;
```

Algebra → class name; :: → scope resolution operator; s → static data member name

**(2) BY OBJECT**

syntax:-

```
obj . s ;
```

obj → object name; . → dot operator; s → static data member name

∴ But 's' static is not part of object. It only access by object

## * DECLARING STATIC MEMBER IN 'PUBLIC' OF CLASS AND ACCESS IN MAIN:

*When we declare static data member in public part of class then we can easily access them in our main by object or by class name bcz it (static member) is public.

(But we also must make sure, that we write defination of these static members).

```
class Algebra : {
    public:
        static int s; // declaration of static data member
};

int main () {
    cout << Algebra :: s << endl; → access by class name
    OR
    cout << obj.s << endl; → access by object.
}
```

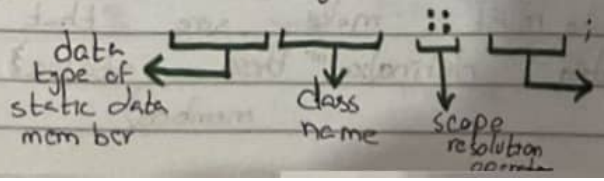
∴ It gives linker error, bcz only declaration of static data member exists. We did not define it. We must need to write its definition.

**NOTE:-**

- Decleration of static data member is inside the class.
- Defination of static data members is outside (global) the class.

**** STATIC DATA MEMBERS DEFINES IN GLOBAL ****

syntax for defining:




example:-

```
int Algebra :: s;
```

**EXAMPLE:-** (Declraring and defining static data member in public it outside the class and access it in main)

```
class Algebra {
    int a, b;
    public:
        static int c; // Declaration of static
};

int Algebra :: c;
OR
int Algebra :: c(0);   } same // Defination of static.

int main () {
    Algebra obj1;
    cout << Algebra :: c << endl; // Access by class name
    OR
    cout << obj1.c << endl; // Access by object.
    return 0;
}
```

∴ as 'c' is declared publicly, So, it is easily accessible in ma

OUTPUT: 0
0

see file → accessing_static_data_member.cpp

## ★ UPDATE VALUE OF STATIC IN MAIN :

↳ We can also update its (static data member) value in main function.

like:

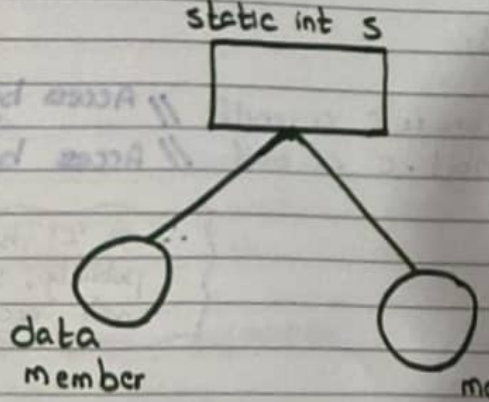
```
Algebra :: c = 99;
```

OR

```
obj .c = 99; → by object
```

★ Every object of class can access static.

★ IF one object change its value, it means it is change(update) for all the objects bcz static is shared b/w every object.




∴ if we change the value of static data mamber Then this value is hold by it. bcz static mantain their state.

**EXAMPLE:-** (static maintain their state)

```
class Algebra {
        int a, b;

    public:
        static int c;
};

int Algebra :: c(0);

int main() {

    Algebra obj1, obj2;

    Algebra :: c = 99;
    obj1.c = 88;

    cout << Algebra :: c << endl;
    cout << obj3.c << endl;
}
```

**OUTPUT:**

```
88
88
```

see file → static_maintain_state.cpp

## *DECLARING STATIC DATA MEMBER IN 'PRIVATE' (PART) OF CLASS:

↳ When we declare static data member in private part of the class then we cannot acess them in our main function. Our main function cannot print, date (access) them because they are pivately declared.

```
class Algebra {
    static int s; //PRIVATE DECLARATION {∴ class is private by default}
};
int Algebra::s; //DEFINATION
int main () {
    Algebra obj;

    Algebra::s = 9;
    obj.s = 9;

    cout << Algebra::s;
    cout << obj.s;
}
```

THESE ALL GIVES "ERROR" BCZ STATIC DATA MEMBER IS PRIVATE. SO, IT CANNOT ACCESS IN MAIN.

↳ So, we need to print, update private static data member by accessing them through any member function, or by making there setters and getters (inside class).

* Every member function (of class) can access static data members.

## * "ACCESSING STATIC MEMEBER THROUGH MEMBER FUNCTION":

EXAMPLE:

```
class Algebra {
    int a;
    static int s;    //DECLERATION
public:
    void setA(int a1, int s1) {
        a = a1;
        Algebra::s = s1; OR s=s1;
    }
    void print () {
        cout << "a: " << a << "Its:" << s << endl;
    }
};
int Algebra::s;      //DEFINATION
int main() {
    Algebra obj1;
```

```
    obj1.setA(5, 2);
    obj1.print();
};
```

**OUTPUT:**

```
a:  5
s:  2
```

see File

→ private_static_member_accessing_by_member_function.cpp

## "SETTER AND GETTER FOR STATIC MEMBER"

* The setter and getter function for static is also static.

**EXAMPLE:** (static member's setter and getter)

```
class Algebra {
    static int s;          // DECLERATION
public:
    static void setS(int s1) {          //SETTER
        this->s = s;       //ERROR  ∴ bez 'this' pointer exists only for non-static functions
        Algebra::s = s;
    }
    static int getS() {
        return s;
    }
};

int Algebra::s;            // DEFINATION

int main() {
    Algebra obj1;
```

same

obj1.setS(5); OR Algebra::setS(5);

cout << obj1.getS() << endl; OR cout << Algebra::getS() << endl; } same

}

**OUTPUT:**

5

see File → setter_getter_For_static.cpp

**NOTE:**

- 'this' pointer does not exists in static member function.
- 'this' pointer only exists in non-static member function of the class.

★ **STATIC** data members can update, print access in member fuctios, for setter, getter or any other by simply through their name.

like: s = 5; OR s = s1;

No need of this: Algebra::s = 5; , Unless we have a situation in which any argument/variable which is assigning to static data have same name: as static

i.e s = s;

Here must use Algebra::s = s;

class name

## ★ TO COUNT NO. OF OBJECTS IN OUR PROGRAM.

We have to:

- increment static data member in constructor.
- decrement static data member in desbuctor.

**EXAMPLE:** (To count no. of objects in our program)

```
class Algebra {
        static int count;
        public:
            Algebra () {
                    count++;   OR  Algebra::count++;
            }
            Algebra (int a, int b) {
                    count++;
            }
           ~Algebra () {
                    count--;
            }

           static int getCount() {
                    return count;
           }
```

```
int Algebra::count;

int main() {
        Algebra obj1, obj2(5, 6), obj3;

        cout << Algebra::getCount() << endl;
}
```

**OUTPUT:**

3

see File → count_number_of_objects.cpp

IMPT

**EXAMPLE:**

```
class Algebra {
        static int count;
        public:
            Algebra () {
                count++;
            }
            ~Algebra () {
                count--;
            }
            static int getCount () {
                return count;
            }
};

int Algebra :: count ;

void test () {
    cout << Algebra :: getCount () << endl;
    Algebra ar[5];
    cout << Algebra :: getCount () << endl;
}
int main() {
    cout << Algebra :: getCount () << endl;
    Algebra obj1, obj2, obj3;
    test();
    cout << Algebra :: getCount () << endl;
}
```

**OUTPUT:**

0
3
8
3

see file → `static_basic_program.cpp`

# "BEST METHOD OF CODING IN OOP"

* The best method of writing code in OOP is to make your class a interface.

**INTERFACE** means only write declaration of member functions inside the class. And write all the definations of member functions (i.e that is declared inside class) outside the class.

- This makes your code look better / appropriate.

**SYNTAX:** (For writing defination of all member functions which are declared in class)

```
return type of member functions if any   class name   ::   function name   ( . . . . . )   {
                                                                                  parametrize list if any
```

implementation

}

* For constructor, destructor we don't write return type.

* For constructors and destructors return type bez they don't return type.

```
class name   ::   constructor / destructor name   ( . . . . . )   {
                                                      parameters.
```

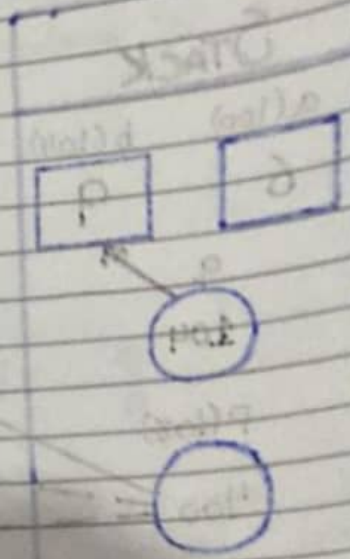
}

**EXAMPLE:** (How to write a good code)

```
class Algebra {
        int a;
        static int s;
        public:
            Algebra();
            Algebra(int , int);
           ~Algebra();
            void setA(int);
            int getA();
            static void setObjc(int);
            static int getObjc();
            void print();
};

//DEFINATION OF STATIC
int Algebra :: s;

// DEFINATION OF CONSTRUCTORS/DESTRUCTORS
Algebra :: Algebra () {
        a = 0;
}

Algebra :: Algebra(int a, int b) {
        this -> a = a;
}
```

INTERFACE HAVING ONLY DECLRATIONS.

```
Algebra :: ~Algebra () {
        cout << "Distructor executed" << endl;
}

//DEFINATION OF MEMBER FUNCTIONS
void Algebra :: setA (int a){
        this -> a = a;
}

int Algebra :: getA() {
        return a;
}

void Algebra :: setObjc (int s1) {
        Algebra :: s = s1;
}

int Algebra :: getObjc () {
        return s;
}

void Algebra :: print () {
        cout << "a: " << a;
}

int main () {
        Algebra obj1;
        cout << Algebra :: getObjc () << endl;
}
```

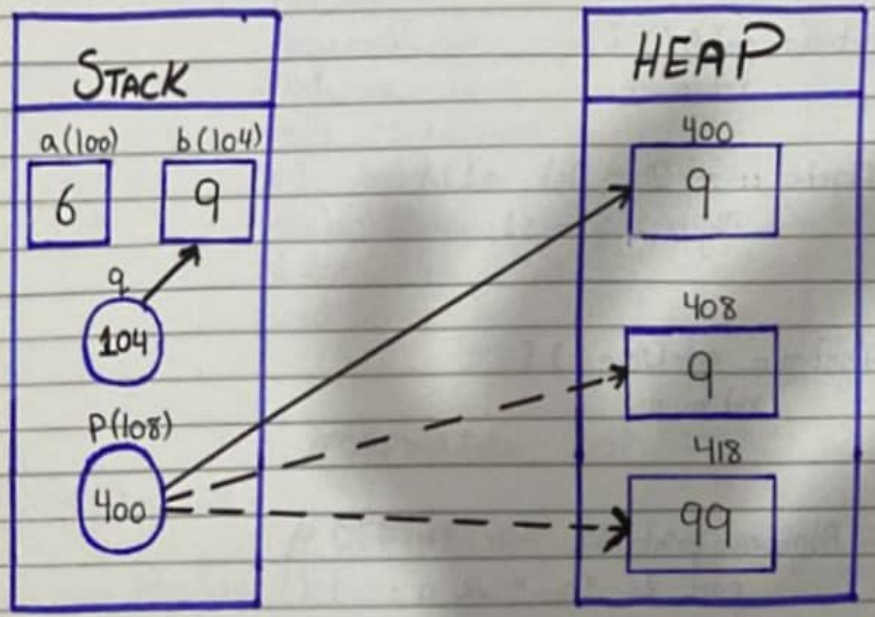


see file → best_coding_practice.cpp

LECTURE#11

# DYNAMIC MEMORY ALLOCATION

## ★ STACK AND HEAP ★




**STACK** (STATIC MEMORY ALLOCATION)

- Stack is limited.
- When stack is full, it means ~~stack~~ "stack overflow".
- stack allocates at compile time.

- The variables which created on stack will automatically destroy when program ends.

```
int main() {
    int a=6;
    int b=9;
    int *q = &a;     (Here q contains 'a' adress)
    q = &b;          (Here q contains 'b' adress)

    cout << q;  }
    cout << &b; }  same

    cout << *q; }
    cout << b;  }  same
```

OUTPUT : 9
9

(∴But now if we initialize *q = 88; it means the variable i.e whose adress is saved in pointer 'q' is also changed to 88.)

```
    *q = 88;

    cout << *q; }
    cout << b;  }  same
}
```

OUTPUT : 88
88

see file → stack_basic.cpp

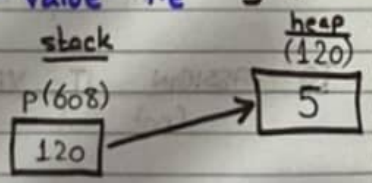
# HEAP (DYNAMIC MEMORY ALLOCATION)

- Dynamic memory allocates at run time.
- All the run time memory allocation is maintain or created in heap called dynamic memory allocation.
- It is virtually unlimited.
- It is vast memory area.
- It is unnamed memory. (means name is not available). But it has adress.
- It is adress based memory.
- They store in pointers, so they are pointer based memory.

**EXAMPLE:** (Dynamically allocating memory)

```
int main() {
    int *p = new int(5);
    cout << &p << endl; // pointer's adress; 608
    cout << p << endl; // Adress of heap; 120 (which is created)
    cout << *p << endl; // Value i.e '5'.
}
// HERE MEMORY IS LEAK
```

This memory will never destroy.




**NOTE:**

- In this above example, there is "memory leak." Bcz here we allocate a memory, then it is our responsibility to de-allocated/remove it. But in the above example we didn't remove it.

**MEMORY LEAK:** If the memory is created/allocated on heap and if we lost the adress/reference of that memory, then we cannot deallocate/remove it. So, it is "memory leake."

**MEMORY EXHAUSTED:**
If the memory of heap is due to not deallocation of then we got a message of "heap exhausted error."

★ **MEMORY LEAK:**

```
int *p = new int(9);
cout << *p << endl;
p = new int(100);
```

∴ This is also a memory leak. Bcz pointer 'p' points new created me on heap whose v is '9'. Then af printing it, 'p' again creates new memory an whose value is So, by this pointer lost adress of pre created mem So, it is leak.

## ↳ DEALLOCATION OF MEMORY:

- So, to avoid all the memory leaks, we have to destroy/remove all the memories created on heap, by ourselves.
- For removing/deallocating memory created on heap we use: 'delete' keyword.

- **Syntax:**

```
delete ┬─┐ ;
       └─→ pointer name.
```

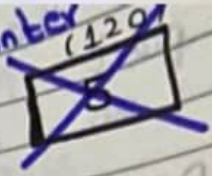
- **EXAMPLE:**

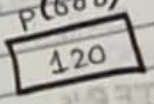
```
int *p = new int(9);
cout << *p << endl;
delete p; // Don't give memory leak
```

**NOTE:**

↳ Now, it don't give memory leak.

As, here the memory allocated by newly created 'int' on heap is removed/delete. But still ~~out~~ our pointer 'p' in 'stack memory' holds the adress deallocated/removed memory. So, pointer acts as 'Dangling pointer'.




## DANGLING POINTER:

A memory is created and then removed, so the adress of that removed memory (pointer still holds the adress of removed memory) is still in the pointer ∴ that type of pointer is called dangling pointer.

**OR**

The pointer contain a memory adress that removed is called 'Dangling pointer'.

↳ It is dangerous.

## ACCESSING UNINITIALIZE MEMORY:

IF you are trying to access to removed memory, then it means the pointer is tying to access the memory which is not part of our program, compiler gives an error called accessing uninitialize memory.

## ★ DANGLING POINTER'S EXAMPLE:

```
int *p = new int(9);
cout << *p;

delete p;

*p = 78;  // DANGLING POINTER

cout << *p;
```

∴ This makes a dangling pointer. bcz 'p' holds adress of that memory which is removed. So, now pointer 'p' cannot initialize that memory again.

**WRONG ✗**

```
int *p;

*p = 5;
```

pointer 'p' must points to some memory before initializing

bcz pointer 'p' is initializing without pointing to any variable or memory.

## EXAMPLE:

```
int main() {
    int *p = new int(5);  /* MEMORY CREATED ON HE
    *p = 99;              // INITIALIZED TO '99'

    cout << &p << endl;   // POINTER ADRESS
    cout << p << endl;    // HEAP ADRESS
    cout << *p << endl;   // VALUE i.e 99.

    delete p;             // AVOIDING MEMORY LEAK
    return 0;
}
```

see File → heap_basic.cpp

## ADVANTAGE OF HEAP:

(1) Life of heap is global.

(2) They have local scope.

**EXAMPLE:** (Returning adress of 'newly created int' (in heap) to main memory)

as we want to return adress of pointer type

```
int*  test () {
        int *p = new int (9);   // Here 'p' will finish as its scope is in this function only, but new int will not finish.
        return p; // returning adress of heap.
}

int main () {
        int *q = test();
        cout << *q << endl;  // 9
        delete q;
}
```

**OUTPUT:**

9

see file

heap_memory_adress_returning_to_main.cpp

# DYNAMICALLY CREATED ARRAY

It has starting adress.

```
int arr[5];
```

→ It holds adress of arr[0].
So, it is a pointer. As, it stores adress.

**NOTE:**

```
int arr[5];
cout << arr << endl;
cout << &arr[0] << endl;
```

} same

## ↳ DYNAMICALLY CREATED ARRAY

Dynamically created array gives benefit to a constant or a variable name (which constant) in the size of array

like:

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

→ where 'a' holds constant like

• **SYNTAX:** (For dynamically created array)

(1)

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

↳ any constant

OR

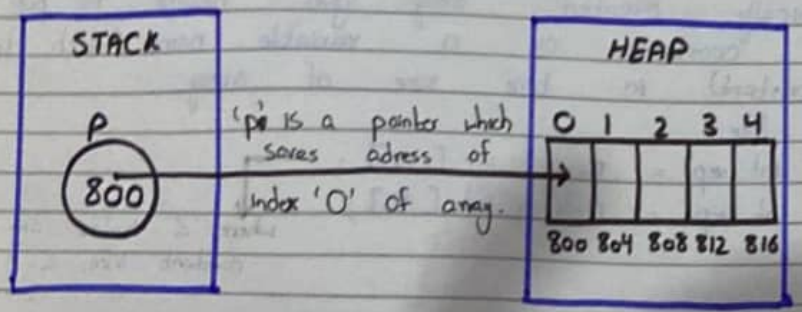
(2)

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

↳ a variable (name) which holds any integer (constant) i.e size = 5 or a = 6 etc

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

★ This dynamically created array will not destroy untill/unless we will not destroy it by ourself.




**NOTE:-**

```
int main () {
    int *p = new int[5];
    cout << p << endl;
    cout << &p[0] << endl;
}
```

} same

p stores adress of index '0'

## • TO ACCESS INDEX OF DYNAMICALLY CREATED ARRAY

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

Two methods:

(1) `p[index];` ↳ index no. which you want to access.

(2) `*(p + index);` ↳ index which we want to access.

starting adress of array i.e '0'.

• **EXAMPLE:**

```
p[2];
*(p+2);
```

} same These both will access index '2'.

- **TO DELETE DYNAMIC ARRAY :**

```
delete[] p;
```

delete keyword ← delete [ ] → square brackets; p → pointer name

*It will delete whole created array.

**EXAMPLE:** (Dynamically created array)

```
int main () {
    int size = 4;
    int *p = new int[size];
    p[0] = 5;
    *(p+1) = 6;
    p[2] = 7;
    *(p+3) = 8;

    for (int i=0; i< size; i++) {
        cout << p[i] << endl;
    }

    delete [] p;
    return 0;
}
```

} Using both two methods of accessing / initializing.

**OUTPUT:**
5
6
7
8

see file → dynamically_created_array_basic.cpp

**EXAMPLE:** (A program in which a function accepts random numbers (+ or -) array and its size, and return the +ve numbers array from it)

(∴ as it will return adress of dynamically created array)

```
int *getPositive (int arr[], int size, int &newSize) {
    int countPositive = 0;

    for(int i=0; i<size; i++) {
        if(arr[i] >= 0)
            countPositive++;
    }

    newSize = countPositive;

    int *p = new int[newSize];

    int j=0;

    for(int i=0; i<size; i++){
        if(arr[i] >=0) {
            p[j++] = arr[i];
            OR
            P[j]= arr[i];
            j++;
        }
    return p;
}

int main() {
    int arr[10] = {0, -1, -2, 3, 4, 5, -6, -7, -8, -9};

    int s= 0;

    int *q = getPositive(arr, 10, s);

    for(int i=0; i<s; i++) {

        cout << q[i] << endl;
    }

    delete [] q;
}
```

↳ we accept newSize by reference bcz when we update its value in this function, then we want to permanently store that value in it.

/* TO CHECK COUNT OF +VE Nums */

// Value of newSize updated.

// DYNAMIC ARRAY
// This array holds +ve numbers.

/* HERE +ve Numbers are assigning to dynamic array */

// Returning dynamic array (that holds +ve numbers

**OUTPUT:**

0
3
4
5

**NOTE:** if we this:
delete q; // It will remov
only starti
index of ar

*When we created/allocated memory on
Its all our responsibility to remove/free that m
Here we allocated dynamic array. So, we are resp
to free it.

see file → return_positive_number_array_from_random_array.cpp

# DYNAMICALLY ALLOCATED MEMORY FOR CLASS

- **DYNAMICALLY CREATED ONE OBJECT:**

```
Algebra *p = new Algebra;
        OR
Algebra *p = new Algebra();
```
} same — Here object is dynamically created by default constructor.

```
Algebra *p = new Algebra(1, 2);
```
} Here object is dynamically created by parametrize constructor.

- **TO ACCESS:**

We use:

```
p->
```

- **TO DEALLOCATE / REMOVE DYNAMIC OBJECT:**

```
delete p;
```
delete keyword ← | → pointer name

**EXAMPLE:** (Dynamically created one single object)

```
class Algebra {
    int a, b;

    public:
        Algebra() {
            a = b = 1;
        }

        Algebra(int a, int b) {
            this->a = a;
            this->b = b;
        }

        friend ostream &operator<<(ostream &, const Algebra&
};

ostream &operator<<(ostream &out, const Algebra &obj) {
    out << obj.a << "\t" << obj.b << endl;

    return out;
}

int main() {
    Algebra *p = new Algebra; // By Default Constru
    Algebra *q = new Algebra(3,4); // By Parametri
                                   constru
```

```
cout << *p << endl; // Default Constructor
cout << *q << endl; // Parametrize Constructor.

delete p;   /* IT IS NECESSARY
delete q;      WHILE DEALING WITH
                  OBJECT */
return 0;      OTHERWISE COMPILER
}              GIVES ERROR.
```

∴ stream-extraction operator is necessary here. bcz '<<' operator is working b/w two different data types
(i) ostream :- cout
(ii) p : Algebra

**OUTPUT:-**

| Default Constructor | |
|---|---|
| 1 | 1 |

| Parametrize Constructor | |
|---|---|
| 3 | 4 |

see file → dynamically_created_object.cpp

# DYNAMICALLY CREATED ARRAY OF OBJECTS (FOR CLASS)

**• DYNAMICALLY CREATED ARRAY OF OBJECT:**

(1) `Algebra *p = new Algebra[3];`

OR

(2)
```
int n = 3;
Algebra *p = new Algebra[3*n]; // ARRAY
```

3 Defau
Constructor objects a
created dy

**• TO REMOVE/FREE DYNAMICALLY CREATED ARRA OBJECT:**

`delete [] p;`

- delete keyword
- square
- pointer name.

**EXAMPLE:** (Dynamically created array of objects)

```
class Algebra {
    int a, b;

    public:
        Algebra () {
            a = b = 0;
        }

        friend ostream &operator << (ostream &, const Algebra &);
};

ostream& operator << (ostream & out , const Algebra & obj ) {
    out << obj.a << "it" << obj.b << endl;
    return out;
}

int main () {
    int n=3;
    Algebra *p = new Algebra[n];   /* ARRAY OF 3 Default Constructor objects is Dynamically created */.

    for (int i=0; i<3; i++) {
        cout << p[i];

    delete [] p;   // IT IS NECESSARY IN CASE OF ORTEST
```

**OUTPUT:**

0 0
0 0
0 0

See file
↳ dynamically_created_array_of_objects.cpp

**NOTE:-**

* If we dynamically created array of objects then we delete like

✗ delete p; // It gives an error bcz here we deal with object we must need to delete full array.

✓ delete[] p;

reason:-
As objects ha different dat
i.e int a, int

LECTURE #01

# POINTER AS A DATA MEMBER OF CLASS:

(∴ pointer points to an adress)

```
class Integer {
    int *val;   // It will work only when we allocate/give a memory to it
                /* Here it is uninitialized or dangling pointer

public:
    //Default Constructor
    Integer() {

        *val = 0;   // ✗ it is wrong.
                    /* bcz we didn't assign a memory to it and we are directly trying to assign it initial value. */

    }
};
```

∴

NOTE: int *p;
*p = 5; //ERR
p = new int;
*p = 5; //VAL

# MEMORY ALLOCATING FOR POINTER:

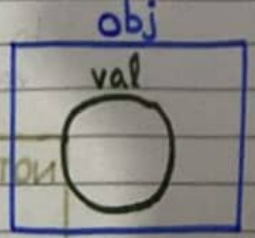
* We assign a memory to a data member pointer of class through object.

* We only assign/allocate memory to pointer data membas inside constructors.

* When we create an object then we automatically call constructors (which have been created by ourself or created by compiler itself) Thus, we only assign a memory to pointer data member of class inside constructors not anywhere else like setters etc.

Thus, in main: ✗

Integer obj;




(∴ We want/will to initialize the memory on heap bcz we want its life long, as long as object exists. Thus, we allocate a memory

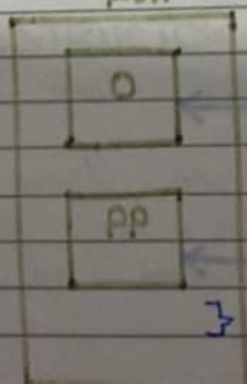
**NOTE:**

* We must allocate memory for data pointer inside constructors.

* We must create memory for pointers in every constructor. The syntax that we will write for this purpose will remain same.

## ➡ MEMORY ALLOCATING FOR POINTERS:

```
class Integer {
    int *val;

public:
    //DEFAULT CONSTRUCTOR
    Integer() {
        val = new int(0);      (∴ here a
                                on heap
        OR                      assigning
        this->val = new int(0);  pointer.)
        //Initialize value
          'zero'.
    }
```



//PARAMETRIZE CONSTRUCTOR

```
Integer (int number)
{
```

↳ bcz user always gives value not a memory to be assigned in pointer.

same

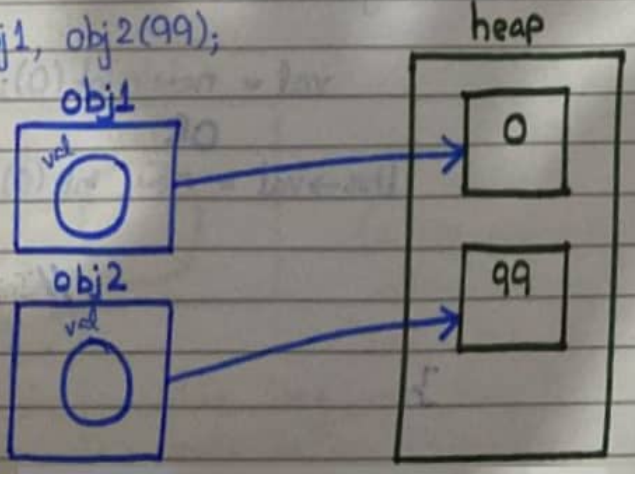
```
val = new int (number);
```

OR

```
val = new int;
*val = number;
```

// here we first assign a memory on heap to pointer then we initialize it with a 'number' that we accept in parameter. */

```
main () {
    Integer obj1, obj2(99);
}
```




// We also must delete these memories allocated on heap.

# BIG-3

↳ '3' things must have to be written by ourselfs when we have data member of pointer in our class.

↳ We must write these 'Big-3' when we have pointer in class

(1) DESTRUCTOR
(2) COPY CONSTRUCTOR
(3) ASSIGNMENT OPERATOR

(These three things we must write in our code when we have data members of pointers).

## ① DESTRUCTOR :

We must have to write destructor by ourselves

* We must also delete all the memories which we allocated (dynamically) on heap by ourself.

* We delete these memories on destructor bcz in destructor our objects will destroy so, we also want to delete heap memory which we allocated.

* Compiler will not destroy these memory itself.

* The destructor is necessary here otherwise we get an error.

* We must/necessary write destructor for this.

* When object go out of scope we must remove it (them) in destructor.

➡ SYNTAX OF WRITING DESTRUCTOR FOR POINTER DATA MEMBER:

```
//NECESSARY DESTRUCTOR
~Integer () {

        delete val;     // Here we are deleting
                        an adress which is
                        allocated on heap.
}
```

RAYYAN - SHABBIR

IT-F20

OOP

Big-3

## ② COPY CONSTRUCTOR:

There are two types of copy constructors

(i) SHALLOW COPY CONSTRUCTOR.

(ii) DEEP COPY CONSTRUCTOR.

* Compiler gives us shallow copy constructor. The copy constructor we write previously is shallow copy constructor. i.e.

```
Algebra (const Algebra &obj) {
        a = obj.a;
        b = obj.b;
}
```

∴ shallow copy constructor is write/used when we don't have pointer data members in our class

* Whenever we have pointer data member in our class, we write a deep copy constructor.

// We must write a deep copy constructor when we have a pointer data member.

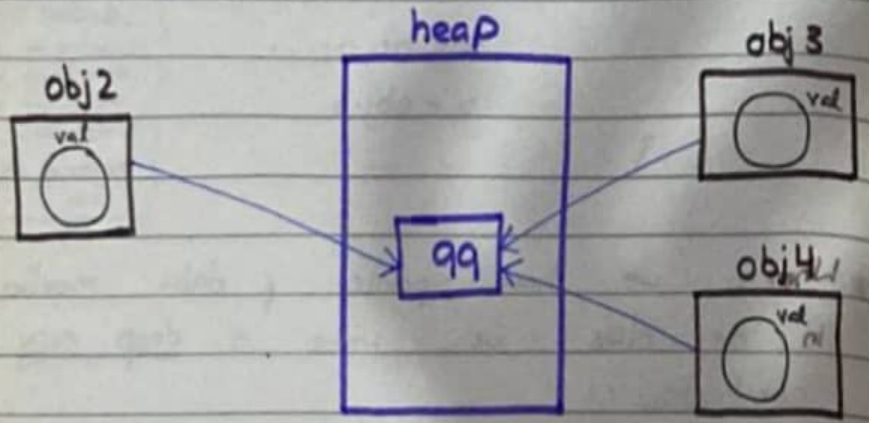
// The copy constructor which we write previously or which compiler gives us

is wrong in this scenerio (when we have pointer data member).

**NOTE:**

★ If we not write a deep copy constructor Then, if compiler makes / uses its own shallow copy constructor, then:

Integer obj2, obj3(obj2), obj4(obj3);




This is wrong. ✗

(∴ When 'obj2' desbroys than 'obj3' and 'obj4' points to a memory which is free.
Thus, these are dangling pointers.)

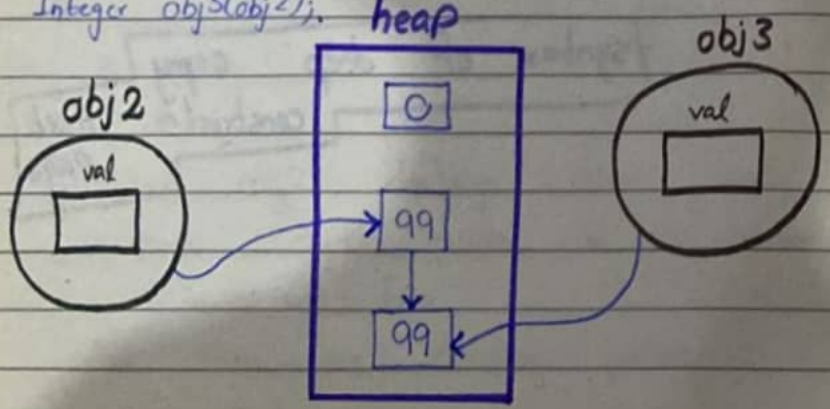
★ The copy constructor which we write precisely or which compiler gives us is shallow copy constructor.
Usually, there is no need to write a shallow copy constructor when we have pointer data member in our class.

★ So the valid copy constructor which we must have to write when we have pointer data member in our class is deep copy constructor.

syntax of deep copy constructor next page

## ★ SYNTAX FOR DEEP COPY CONSTRUCTOR:

```
Integer (const Integer &obj) {

    val = new int;      /* We must have to create a new memory for L.H.S object;
    *val = obj.val; // ERROR X on heap.  */
                    ↳ bcz obj.val shows/have adress which (cannot) is storing in value i.e *val. So, it is error.

    *val = *(obj.val); // VALID ✓

SAME          OR

    val = new int(*(obj.val));
}
;
Integer obj3(obj2);.
```




**NOTE:**

★ Compiler gives us shallow copy constructor.

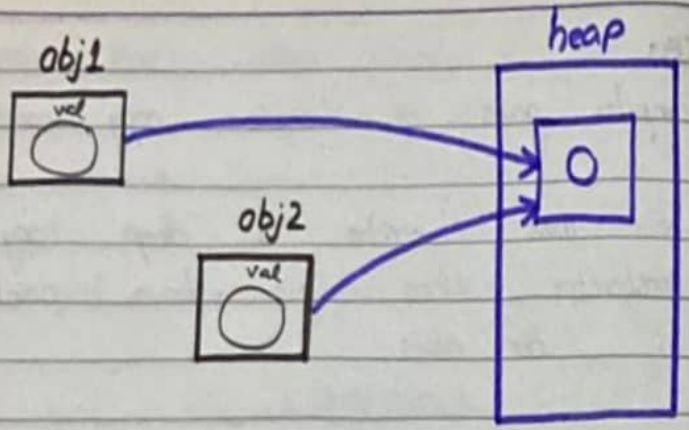
★ We must write a deep copy constructor when we have pointer in our class.

## ③ ASSIGNMENT OPERATOR:

★ The assignment operator we write previously.

```
Integer operator=(const Integer &obj)
{
        val = obj.val      // Here adresses are assigning. So, it is invalid.
        return *this;
}

int main () {
        obj2 = obj1;
}
```

★ This ~~Copy constructor~~ Assignment Operator is invalid when we have pointer data member in

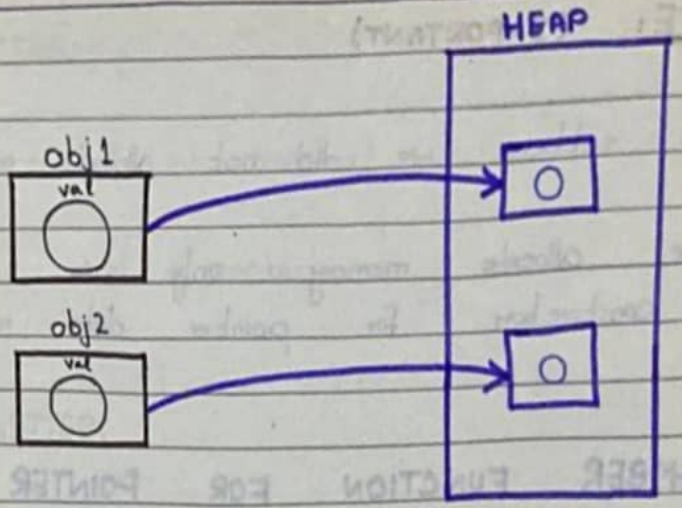



* Here by this obj2 consists the value (i.e adress) same as in heap for obj1. So, this is invalid.

*** SYNTAX FOR VALID ASSIGNMENT OPERATOR:**

So, syntax for writing valid assignment operator, when we have pointer in our class is:

```
Integer operator=(const Integer& obj) {
        //Avoiding self-assignment
        if(this != &obj) {
                *val = *(obj.val); //In assignment operator we only want to assign a value, Thus, we dereference.

        return *this;
}
```




Now, this is valid bcz here obj1 and obj2 have both ↑/have different memory adresses on heap. Only they consist the same value.

## CONCLUSION:

* We must (necessary) have to write (Big-3) deep copy constructor, destructor and assignment operator whenever we have pointer data member in our class.

**NOTE: (IMPORTANT)**

// In setters we did not allocate memory.

/* We allocate memory only in constructors for pointer data members. */

## → MEMBER FUNCTION FOR POINTER DATA MEMBER:

### ⇒ INPUT FUNCTION:

```
void input () {
        cout << "Enter Name: ";
        cin >> *val;
}
```

↳ Here we write '*' bcz we assign a value not an adress.

### ⇒ SETTER:

```
void setVal (int number)
{
        *val = number;
}
```

### ⇒ GETTER:

```
int getVal (  ) {
        return *val;
}
```

↳ here we are returning a value.

**EXAMPLE: (POINTER DATA MEMBER BASIC)**

```
class Integer {
    int *val;

  public:
    //DEFAULT CONSTRUCTOR
    Integer () {
        val = new int (0);
    }
    //PARAMETRIZE CONSTRUCTOR
    Integer (int num) {
        val = new int (num);
    }
    //COPY CONSTRUCTOR (DEEP)
    Integer (const Integer &obj) {
        val = new int;
        *val = *(obj.val);
    }SAME
        OR
        val = new int (*(obj.val));
    }
    //DESTRUCTOR
    ~Integer () {
        delete val;
    }
```

```
    //ASSIGNMENT OPERATOR
    Integer operator = (const Integer & obj) {
        //AVOIDING SELF-ASSIGNMENT
        if (this != &obj) {
            *val = *(obj.val);
        }

        return *this;
    }

    //MEMBER INPUT FUNCTION
    void inputVal () {
        cout << "Enter Number: ";
        cin >> *val;
    }

    //SETTER
    void setVal (int num) {
        *val = num;
    }

    //GETTER
    int getVal () {
        return *val;
    }
```

∴ When we write assignment operator (we must write) copy constructor must have to be written in our program. Bcz when we return something from assignment operator then compiler saves it in un-named object. As returning 'obj' is saving in newly created 'obj' so copy constructor is necessary.

```
//MEMBER DISPLAY FUNCTION
void display () {
    cout << "Value: " << *val << endl;
}
};

int main () {
    Integer obj1, obj2(93);
    // obj2 = obj1;

    obj2.setVal (62);
    obj2.inputVal ();

    obj2.display ();

    cout << obj1.getVal() << endl;
    return 0;
}
```

see file →

pointer_data_member_basic.cpp



# ARRAY IN CLASS:

**★DYNAMICALLY CREATED ARRAY IN OUR CLASS**

We will do/pronde following constructors/functions in our class:

- Default Constructor for array (dynamic created)
- Constructor receives size for array (give user right to give its own size for array).
- A constructor takes an integer array and its size and assign the value/data of that array to our dynamically created array
- Copy constructor → It is necessary.
- Destructor → It is necessary
- Assignment operator → It is necessary

} BIG-3

(Assignment operator 1st make size of L.H.S obj array equal to size of R.H.S obj array, then assign data/value of R.H.S obj array to L.H.S obj array.)

- Input data for array
- adding two array overloaded operator (+).

- printing array
- overloaded index operator [].

(Here we also implement bound checking, if user enters a (wrong/out of range) index which is not exist in our array then he must gets an error. This is what a bound checking will do).

**NOTE:**

* Memory allocation for pointer or array only do in constructors nothing no anywhere else.

# ARRAY IN CLASS:

## (1) CREATION OF ARRAY IN CLASS AND A DEFAULT CONSTRUCTOR FOR THIS ARRAY:

```
class Array {
    int * ar;  // holds reference of array, created on heap.
    int size;  // holds max size of array.

public:

    //DEFAULT CONSTRUCTOR
    Array () {
        size = 5;  // Default size of array.
        ar = new int[size];  // allocating array on heap.

        for(int i=0; i<size; i++)
            ar[i] = 0;  // assigning a default value '0' for every index of array
    }
};
```

(points to an array.)

∴ every array must have its size written with it, but for class, array and its size both exists in object.

∴ Here default size is '5' for dynamical created array, when user don't give any size, this size will be assign as default.

(2) A CONSTRUCTOR WHICH TAKES ONLY SIZE FOR ARRAY FROM USER:

```
Array ( int size)
{
    this -> size = size; // initializing size of array
    ar = new int [size]; // allocating array on heap
        OR
    ar = new int [this -> size];

    for (int i=0; i<size; i++)
        ar[i] = 0; // Here also assigning a default value '0' to every index.
}
```

(3) We can also write A CONSTRUCTOR WHICH TAKES ARRAY OF INTEGER AND ITS SIZE AS ARGUMENT AND ASSIGN THAT INTEGER ARRAY'S VALUE/DATA TO OUR DYNAMICALLY CREATED ARRAY.
(copying data of int array to our dynamic array)

Array

"We can also write"

(3) A CONSTRUCTOR THAT TAKES ARRAY OF INTEGER AND ITS SIZE IN PARAMETER AND ASSIGN THAT INT ARRAY'S VALUE/DATA TO OUR DYNAMICALLY CREATED ARRAY:
(copying data of int array to our dynamic array)

```
Array (int data[], int size)   // This is not array of object. This is array of int
{
    this->size = size;  // first making size of dynamic array equal to size of int array
    ar = new int [size];  // allocating array on heap.
        OR
    ar = new int [this -> size];

    for(int i=0; i<size; i++)
        ar[i] = data[i]; // initializing object's array with the data of array passed as argument.
}
```

# BIG-3

Big-3 have 3 things:

(i) Destructor

(ii) Copy constructor

(iii) Assignment operator

* We must (necessary) have to write these three (3) things (big-3) whenever we have a pointer or array in our class.

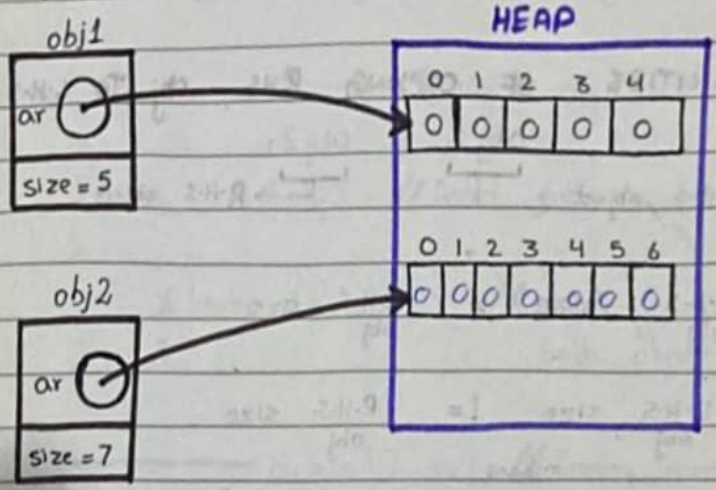
## (4) (i) DESTRUCTOR:

As when we create dynamically array. So, we immediately write a destructor for it. Otherwise it will be a big memory leak. Bcz dynamic array consist of more memory on heap than a single int/float etc.

```
~Array () {
    delete [] ar; // here we destroy/free the dynamic array not the size (bcz 'size' is allocated on stack). We are destroying adress which we allocated on heap for an array.
}
```

**IN MAIN:**

```
Array obj1, obj2(7);
```




## (5) (ii) COPY CONSTRUCTOR:

* Syntax of copy constructor for dynamic array of class.

```
Array (const Array & obj) {
```

This is object of class 'Array'. So, it consists of array as a data member also.

```
    size = obj.size; // initializing (set) size of array.
    ar = new int [size]; // allocating memory for array on heap

// NOW COPY DATA
    for (int i=0; i<size; i++)
        ar[i] = obj.ar[i]; // initializing object's array with the data of right-hand side object.
}
```

## (6) (iii) ASSIGNMENT OPERATOR:

* We don't need to allocate memory in assignment operator.

### → POSSIBILITIES OF COPYING RHS obj TO L·H·S obj

obj1 = obj2;

obj1 ← L·H·S object

obj2 → R·H·S object

① L·H·S obj size = R·H·S obj size

② L·H·S obj size != R·H·S obj size

means

L·H·S size < R·H·S size

L·H·S size > R·H·S size

* If L·H·S obj size != R·H·S obj size then first we equal both of sizes than assign data/value of R·H·S obj to L·H·S obj.

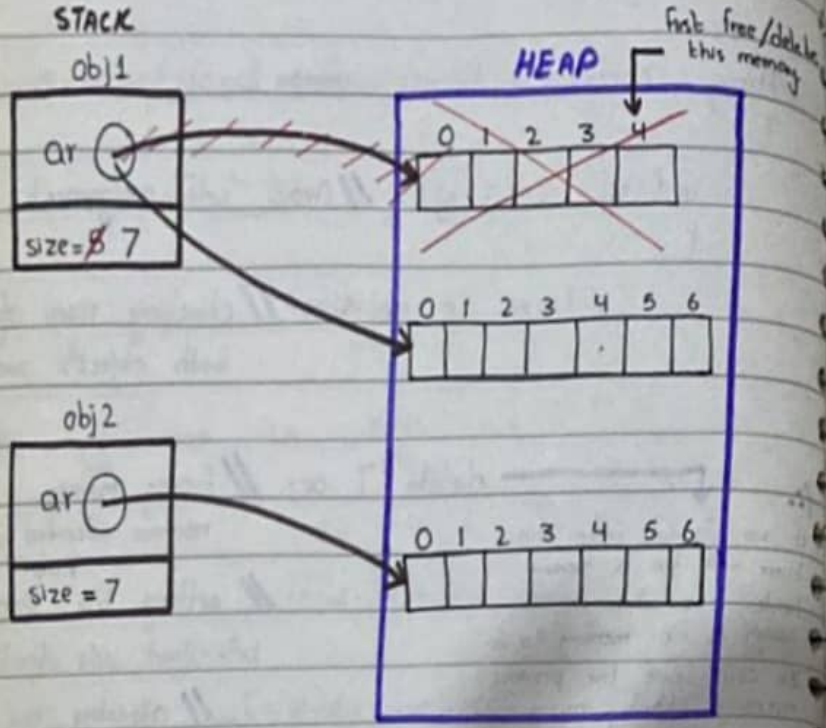
### → SYNTAX OF ASSIGNMENT OPERATOR FOR ARRAY IN CLASS:

```
Array operator= (const Array & obj)
{
    if (this != &obj) // Avoid self-assignment
    {
        if (size != obj.size) // checking sizes of both object's array
        {
            delete [] ar; // Freeing memory resources allocated on heap.
            size = obj.size; // setting the size of Left-Hand side object.
            ar = new int[size]; // allocating new array on heap for L·H·S object.
        }
        for (int i=0; i<size; i++)
            this->ar[i] = obj.ar[i]; // initializing object's array with data of right-hand-side object
    }
    return *this;
}
```

(∴ if we do not write this then there will be a memory leak bcz when we assign a new memory to it. It will leave the previous memory that memory is a memory leak.)

(∴ here copy constructor is necessary bcz when obj returns compiler makes its o unnamed object for storing the returned ob As returning obj is storing in newly created obj. So, it necessary requires a copy

obj 1 = obj 2
(5) (7)




## ➡ MEMBER FUNCTION FOR ARRAY IN CLASS:

(7)

### ⇒ INPUT FUNCTION:

★ Input function that takes values for dynamic array.

```
// Input member function to take input from user for array.

void input()
{
    for (int i=0; i < size ; i++)
    {
        cout << "Enter value " << i+1 << " = ";
        cin >> ar[i]; // Taking value/data in any index.
    }
}
```

(8) ⇒ PRINT MEMBER FUNCTION:
(To display array)

```
void print() {
    for (int i=0; i<size; i++)
        cout << ar[i] << "\t";

    cout << endl;
}
```

(9) ⇒ ADITION(+) OVERLOADED OPERATOR:

(for adding two arrays)

We will take '1' obj as L·H·S object and '1' obj as R·H·S object.

As we returning an object which is also type of 'Array'. So, its return type is Array.

```
Array operator+ ( const Array &obj) const
{
```

↳ we don't want to change R·H·S object. We only want to make addition.

```
Array temp;  // creating an object for storing addition.

if(this->size == obj.size) {
        temp.size = size;
        for(int i=0; i<size; i++)
            temp.ar[i] = ar[i] + obj.ar[i];
}

return temp;
}
```

obj1 + obj2

Array obj;  obj1+obj2  obj[0]

(10) OVERLOADED INDEX [] OPERATOR:

obj[0];
↳ index numbe

'[]' → known as index operator.

∴ int ar(5);

'()' → known as function operator

★We can write function overloaded oper

★ We can also write index '[]' overloaded operator.

★ If in main we/user do this obj[0] (↳ any index numbe) and we want he/we won't get any error (deals obj itself as an array), then we must have to write an overloaded index '[]' operator in our class.

Bcz here obj[0]; an operation '[]' is performing b/w class and int types. So, it is necessary to write overloaded operator [] in class.

## ★ SPECIALITY OF OVERLOADED '[]' OPERATOR:

Overloaded index '[]' operator can be used to set as well as get value of specific index of an array.

impt

ar[0]; → GET VALUE → get value mechanism can be used for both constant as well as non-constant objects. Bcz it is only used for getting the value of specific index.

ar[0] = 5; → SET VALUE

impt

set value mechanism can only be used for non-constant objects. Bcz we can change/update value of non-constant objects. Set value mechanism can not be used for constant objects bcz we cannot set/update/change value of constant objects.

**NOTE:**

★ As we know when user/we write this obj[0];, then he/we wanted to get a value (not anything else like adress etc). If we have array of integers, then the return type of our overloaded '[]' operator will be 'int'.

obj[0];
↳ it returns int

## ➡ SYNTAX OF OVERLOADED INDEX '[]' OPERATOR FOR NON-CONSTANT OBJECTS:

obj[0];
L·H·S object ← obj → bcz we want this index value
R·H·S object

while writing overloaded index operator function of non-constant obj we write '&' sign with the return type bcz we want to get/return an adress. So, that we can easily change (set) or get its value. As, for non-constant obj we want to handle/deal both l-value and r-value. Thus, we are returning an adress (makes it return type &).

We must pass it an index number that we want to get

```
int &  operator[](int number) {
```

↳ we want to return int value.

// We also apply bound checking here to check if the user go to (out of scope) an index which is not exists in our array.

}

(10.1)

# OVERLOADED INDEX OPERATOR FOR NON-CONSTANT OBJECTS: TO HANDLE 'l-value' AND 'r-value':

```
int &  operator [] (int  index)
{
    // Bound checking of array's index.
    if ( index >=0  &&  index <size) // We check
                                     if the user try to
                                     access any index which
                                     is not part of
                                     our array, then we
                                     simply gave him
                                     message of out-of
                                     bond (error) and
                                     exit (return).
        return ar[index];

    cout << "Array index is out of bound";

    exit (0); // Same as return.
              ↳ '0' means normal termination.
                Other than '0' means abnormal
                termination.
SAME {
    OR
    return -1;
}
```

## * ANOTHER METHOD OF OVERLOADED INDEX OPERATOR FOR NON-CONSTANT OBJECT

```
int &  operator [] (int  index)
{
    // BOUND CHECKING
    if ( index < 0  ||  index >= size)

        exit(0);

    return  ar[index];
}
```

Now, in main: Array ar1; // Non-constant object.

```
cout << ar1[0] ; // getting value.
                 // or  set value:
ar1[0] = 5;
```

both will work. ✓

## SYNTAX OF OVERLOADED INDEX '[]' OPERATOR FOR CONSTANT OBJECTS:

* In constant objects we only want to get value. We cannot set value of constant objects.

→ While writing overloaded index [] operator function of constant objects we do not write '&' sign with return type bcz we want to get/return a value (not an adress). As, for constant obj we want to handle/deal only r-value. So, we do not write '&' sign.

We must pass in an index number that we want to get.

```
int operator [] (int index) const
{
```

→ We want to return int/value.

We make it const bcz we don't return adress. And const objects can access it easily.

```
    // We also apply bound checking here, to check if the user go to (out of scope) an index which is not exists in our array.
}
```

## (10.2) OVERLOADED INDEX OPERATOR FOR CONSTANT OBJECTS TO HANDLE 'r-value' only:

```
int operator [] (int index) const
```

→ We make it const so, that const objects can access it.

```
{
    // Bound checking of array's index.
    if( index >=0 && index < size)
        return ar[index]; // returning the legal memory location.
    cout << "Array index is Out-of-Bound";
    exit(0); // cleaning the program (if required)
}
```

In main:

```
const Array obj;
cout << obj[3]; // We can only get value not set for const objects
```

* Another method/logic is also available here like non-constant objects.

## EXAMPLE: ARRAY OF CLASS BASIC

```
class Array {
    int * arr;
    int size;

public:
    //DEFAULT CONSTRUCTOR
    Array () {
        size = 5;
        arr = new int [5];

        for(int i=0; i<size; i++)
            arr[i] = 0;
    }
    //CONSTRUCTOR THAT ACCEPT SIZE
      OF ARRAY:

    Array () {
        this->size = size;
        arr = new int [size];
        for(int i=0; i<size; i++)
            arr [i]=0;
    }
```

```
    //CONSTRUCTOR THAT ACCEPT AN
      ARRAY WITH ITS SIZE AND INITIALIZE
      OBJECT'S DATA WITH THAT ARRAY.

    Array(int arr[], int size) {
        this->size = size;
        this->arr = new int[size];
        for(int i=0; i<size; i++)
            this->arr[i] = arr[i];
    }

    //COPY CONSTRUCTOR (NECESSARY)->BIG-3
    Array (const Array & obj) {
        size = obj.size;
        arr = new int[size];
        for (int i=0; i<size; i++)
            arr[i] = obj.arr[i];
    }

    //DESTRUCTOR (NECESSARY) -> BIG-3
    ~Array ( ) {
        delete [] arr;
    }
```

## //OVERLOADED ASSIGNMENT OPERATOR (NECESSARY) → BIG-3

```
Array operator= (const Array& obj)

    if(this != &obj) {
        if(size != obj.size)
            delete[] arr;

        size = obj.size;
        arr = new int[size];
    }
    for(int i=0; i<size; i++)
        arr[i] = obj.arr[i];

    return *this;
}
```

## //OVERLOADED INDEX OPERATOR[] FOR NON-CONSTANT OBJECT

```
int& operator[] (int index) {

    if(index >= 0 && index <size)
        return arr[index];

    cout<< "You enter a wrong index";
    exit(0);
}
```

## //OVERLOADED INDEX OPERATOR[] FOR CONSTANT OBJECT

```
int operator[] (int index) const
{
    if(index < 0 && index >= size)
        exit(0);

    return arr[index];
}
```

## //OVERLOADED ADITION OPERATOR (+)

```
Array operator+ ( const Array& obj ) {
    Array temp;

    if ( this->size == obj.size) {
        temp.size = size;

        for(int i=0; i<size; i++ ) {
            temp.arr[i] = arr[i] + obj.arr[i]
        }
    }
    return temp;
}
```

```
//INPUT FUNCTION
void getInput() {
    for(int i=0; i<size; i++) {
        cout<<"Enter value at index "<<i<<endl;
        cin >> arr[i];
    }
}

//PRINT FUNCTION
void print() {
    for(int i=0; i<size; i++)
        cout << arr[i];
    cout << endl;
}
};

int main() {
    int arr[4] = {1, 3, 5, 7};
    Array arr1, arr2(4), arr3(arr, 4), arr4(arr2);

    arr2.getInput();
    Array temp = arr2 + arr3;
    temp.print();

    return 0;
}
```

see file
array_of_class_basic.cpp

**CONCLUSION: IMPT NOTE**

* 'l-value' means left value. It can be on (both) right or left side of assignment operator. 'l-value' can be read as well as writeable.

* 'r-value' means right value. It can only be on right side of assignment operator. 'r-value' can only be read and cannot be write.

* Non-const objects can be read as-well as writeable. We can modify them.

* Const objects are only readable. We cannot modify them.

* If we return a function by reference '&' then if the R.H.S variable in which we are getting its return value is modified then it means the L.H.S variable which we pass will also be modified.

* Thats why we use '&' with ([] operator) functions of non-const and don't use '&' with ([] operator) functions of const objects. Bez if we use '&' with ([] operator) function of const obj then it will return a reference, which is stored in a R.H.S l-value. Now, if we modified l-value then the const object on R.H.S also be modified. But as we know const object are read only they cannot be modified. So, here then this concept will be wrong. That's why we don't return it & by reference.



# TEMPLATES

* Templates are powerful features of C++ which allows us to write generic (general /pls) programs.

## TEMPLATE TYPES:-

Templates have two types:

(ii) Function templates (these templates make functions)

(iii) Class templates (these templates make class)

* We can create/write a single function or Class to work with different data types by using a template.

* If template as well as simple (exact) function are availbe and both are valid for t be called then simple (exact) function gets higher priority.

## (1) FUNCTION TEMPLATE:

* We can write a single function to work with different data types (i.e int, float, char, double etc) by using a function template.

**IF WE HAVE A SCENERIO:**

In which we want to write a
- Function which add two ~~numbers~~ 'integer' numbers and return their result.
- Function which add two 'float' numbers and return their result.
- Function which add two 'char' numbers and return their result.

↳ By simple technique we will write these functions separately one-by-one.

see next page

**LIKE:**

```
//INTEGER FUNCTION WHICH ADD TWO INTEGE
int add (int a, int b) {
        return a+b;
}
//FLOAT FUNCTION WHICH ADDS TWO FLOAT
float add (float a, float b) {
        return a+b;
}
//DOUBLE FUNCTION WHICH ADDS TWO DOUBLE
double add (double a, double b) {
        return a+b;
}
```

(∴ These functions are know as template functions)

* This is function overloading.

* By this technique we have to write many times different functions for different data types. So, here chances of error increases.

➡ Thus, for this we prefer/use:

* A concept in which we write one time function which works for all types of data types (i.e int, float, double, char ... etc).

This concept is known as "template".

## FUNCTION TEMPLATE:

(template of functions)

* Function template is a template which make different functions according different data types.

* Function template is a template of different functions.

FUNCTION TEMPLATE

∴ create many instances

TEMPLATE FUNCTIONS

## DEFINING A FUNCTION TEMPLATE:

### (1) FOR ONE (1) DATA TYPE AT A TIME:

**SYNTAX:-**

A function template start with the keyword 'template' followed by template parameter(s) inside '<>' which is fllowed by function defination.

keyword → template <typename t> ← un-named type (that can accepts different data types)

↳ it tells have many data types

```
t functionName ( t parameter1 , t parameter2, ... )
{
    // CODE
}
```

* In above code, 't' is a template argument that accepts different data types (int, float ... etc).
* 'typename' is a keyword.
* We can also use 'class' keyword insted of 'typename'.

**NOTE:-**

We can also use class keyword.

```
template < class t>
```
OR → We can use class instead of typename keyword.
```
template < typename t>
```
} SAME

* It is template of type 't'.

* It is basically used to create a same function which for all types of data types.

* In last page, when an argument of a data type is passed to "functionName()", the compiler generates a new version of 'functionName()' for the given data type.

* Template function don't have any data type itself.

* There type will be set by user when he pass parameter or define any specific data type for it.

* They are un-named basically.

# CALLING A FUNCTION TEMPLATE

## (1) FOR ONE (1) DATA TYPE AT A TIME:

* Once we've declared and defined a function template, we can call it in other functions or templates (such as main()) with following syntax:-

**SYNTAX:- (FOR CALLING)**

```
functionName <dataType> (parameter1, parameter2, ...
```
→ data type is not necessary if our parameters are of same data types.

**FOR EXAMPLE:-**

* Parameter of same data type:

```
functionName <int> ( 1 , 2);
```
OR
```
function Name ( 1 , 2);
```
} SAME

→ here we don't need to write data type bcz our parameters are of same data types (i.e integers).

**EXAMPLE:** "LET US CONSIDER A TEMPLATE THAT ADD TWO NUMBERS"

```
template <typename t>

t add (t a, t b) {
        return a+b;
}
```

t → typename i.e (int, float etc)

FUNCTION TEMPLATE

/* This template is used for all data types i.e int, float etc.

* Compiler will simply see what we pass int, float etc. Then write functions according to passed parameters.

* It will work at compile time not at run time. */

* Before giving/passing parameter from main(), the type (return type) of this add function is unknown.

We can then call it in the main() function to add 'int', 'double', 'float', 'char'.

```
int main() {

//CALLING WITH 'INT' PARAMETER
int re1 = add (2, 3);        → makes integer add function.
    OR  add<int>(2,3);

//CALLING WITH 'FLOAT' PARAMETER
float re2 = add (1.1f, 2.2f);   → makes float add function
                ↳ we write 'f' for float otherwise, it will consider as double
    OR  add<float>(1.1f, 2.2f);

//CALLING WITH 'DOUBLE' PARAMETER
double re3 = add (1.12, 2.39);   → makes doub add functio
    OR  add<double>(1.12, 2.39);

//CALLING WITH 'CHAR' PARAMETER
char re4 = add ('R', {'S');   → makes ch add funct
    OR  add<char>('R','S');

}
```

**NOTE:**

* Templates only works at compile time.
* One function will be made against one call.

```
add (1, 2);
add (5, 6);
```
} both make one time add function for integers.

* Template function is also known as generic (general) in Java.
* This concept is also called polymorphism.

**↳ ACTUAL CALL FOR TEMPLATE FUNCTIONS:**
**(BACK END CALL / COMPILER CALL)**

```
t add<t> (t v1, t v2);
```
↳ data type

**FOR 'INT':**

```
int add<int> (int v1, int v2);
```

same for float, double etc.

## GLITCH/FOUL OF TEMPLATE:

* In templates flaws come, when parameter (passing from main()) have two or more than two different data types at a time.

```
add (1, 2.3);
```
1 → integer, 2.3 → float

* Because one template hold one data type at a single time.

**TO MAKE THIS WORK:-**

* We write a template that work for more than one data type at a time.

For this we have two methods:

### METHOD #01
### EXPLICITING CALL (CONVERTRY)

* We have to tell by ourself what will be the type of template

like:

```
cout << add<float> (1, 2.2f) << endl;
```

* By this only float type function template will be defined.

EXAMPLE:-

```
template < typename t>
t add(t a, t b) {
        return a+b;
}

int main () {

    int re1 = add<int>(2, 3.99);
    float re2 = add<float>(1, 6.6f);
    double re3 = add<double>(9, 3.2);

}
```

## METHOD # 02
## TEMPLATE FOR TWO DATA TYPES:

* We can write a template for more than two data types at a time.

## SYNTAX: (FOR TWO DATA TYPES TEMPLATE AT A SINGLE TIME)

```
template < typename t1, typename t2>
(∴ 't1' and 't2' are two different data types.)
t1 functionName (t1 a, t2 b) {
    // CODE
}
```

(return type is of our choice)

## * CALLING THIS FUNCTION TEMPLATE:

~~int main () {~~

~~functionName < data type1, data type2>();~~

```
int main () {

    functionName < data type1, data type2 > (para1, para2);

}
```

(data type 1, data type 2 → now here data types are necessary)

## EXPLICITING CALL: (OF TYPECASTING)

* Typecasting is an other example of explicit calling of function template.

```
static_cast <datatype> (variable);
```

↳ static_cast is a function template.

## EXAMPLE:- (FOR ONE(1) DATA TYPE)

```
template <typename t>
t add (t a, t b) {
        return a+b;
}

int main () {

    cout << add<int>(5, 7);
    cout << add<float>(6.3, 7.1);

    cout << add<int>(8.3, 2);
}
```

**OUTPUT:**

```
12
13.4
5
```

see file

↓

function_template_1_datatype_basic.cpp

**EXAMPLE:- (TEMPLATE FOR TWO (2) DATA TYPES)**

```
template < typename t1 , typename t2 >
t1 add ( t1 a, t2 b) {
        return a+b;
}

int main () {

    cout << add<int , float>(5, 3.3) << endl;

    cout << add< float , int>(1.6, 2) << endl;

}
```

**OUTPUT:**

8

3.6

see file

`function_template_2_datatype_basic.cpp`

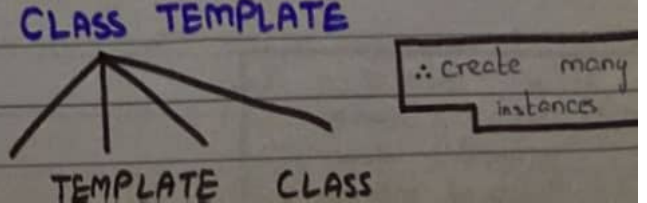
## (2) CLASS TEMPLATE:

* We can write a single class to work with different data types (i.e int, float, double, char etc) by using a class template.

* A concept in which we write one time class which works for all types of data types (i.e int, float, double, char ... etc). This concept is known as "template".

* Class template is a template which make different classes according to different data types.

* Class template is a template of different functions.



**NOTE:**

* Similar to function templates, we can use class templates to create a single class to work with different data types. (int, float, double etc)

* Class templates come in handy as they can make our code shorter and more manageable.

# CLASS TEMPLATE DECLARATION:

## (1) FOR ONE (1) DATA TYPE AT A TIME:

**SYNTAX:-**

* Class template starts with the keyword 'template' followed by template parameter(s) inside '<>' which is followed by class declaration.

keyword → template

un-named type (that can accepts different data types) →

```
template <class t> //OR template <typename t>
                   (same)
class className {

    t var;  // Members variable of type 't'.

public:
    //MEMBER FUNCTION OF TYPE 't'
    t functionName (t arg) {
        //
    }

};
```

- It is a class template.
- It is not a normal class.
- It works differently according to different data types.

* In above declaration, 't' is the template argument which is placeholder for the data type used, and 'class' is a keyword.

* Inside the class body, a member variable 'var' and member function 'functionName()' are both of type 't'.

## CREATING A CLASS TEMPLATE OBJECT:

* Once we've declared and defined a class template, we can create its object in other classes or functions (such as main()) with following syntax:

```
className <dataType> classObject;
```

**FOR EXAMPLE:**

(IF WE HAVE Class name 'Algeb

```
Algebra <int> obj1;   // IT CREATE A CLASS
                         OF TYPE 'INT'.

Algebra <float> obj2; // IT CREATES A CLA
                         OF TYPE 'float'

Algebra <string> obj3; // IT CREATES A CLA
                          OF TYPE 'string'
```

**NOTE:-**

* In case of class, if we have some data types in parameters even then we cannot write like this:

Algebra obj1; ✗ WRONG.

It is necessary to write data type whenever we create an object of class template.

like:

Algebra <int> obj1; ✓ RIGHT

* It is compulsory to specify the type when declaring objects of class templates.

Otherwise, the compiler gives/produce an error.

Algebra obj1; //ERROR ✗

Algebra <int> obj1; //VALID ✓

**EXAMPLE: CLASS TEMPLATE** [For different data types ie int, float etc

```
template < class t >
class Algebra {
    t a, b;   //MEMBER VARIABLE OF TYPE 't'.
public:  //DEFAULT CONSTRUCTOR
    Algebra () {
        a = b = 0;
    }
    //MEMBER INPUT FUNCTION
    void input () {
        cout << "Enter a:";
        cin >> a;
        cout << "Enter b:";
        cin >> b;
    }
    //MEMBER GET FUNCTION
    t getData () {
        return a;
    }
};
```

unknown type (which will be defined by user)

unknown type (which will be defined by user)

```
int main () {

    Algebra <int> obj1;  //TEMPLATE CLASS
                            FOR 'INT'.

    obj1.input();

    cout << "a = " << obj1.getData() << endl;


    Algebra <float> obj2; //TEMPLATE CLASS
                            FOR 'FLOAT'.
    obj2.input();
    cout << "a= " << obj2.getData();

    Algebra <char> obj3;  //TEMPLATE CLASS
                            FOR 'CHAR'.
    obj3.input();
    cout << "a= " << obj3.getData();

    (:. Here three classes are made
        of different data types
        i.e int, float, char.)

}
```

* This is known as polymorphic behaviour.

**NOTE:**

* **FOR MAKING CLASS TEMPLATE OF ARRAY.**

↳ size must be / always be integer.

↳ Array can be of type 't' (either int, float etc)

see file ↓

**class_template_basic.cpp**

# Defining a Class Member Outside The Class Template

* Suppose we need to define a member function outside of the class template.

```
template <class t>
class Algebra {
    //
    public:
        void output(); // DECLARATION INSIDE CLASS
};

//ERROR
void Algebra :: output() { X
    // CODE
};

//ERROR
void Algebra <t> :: output() { X
    // CODE
}
```

THESE BOTH ARE ERROR. THIS IS NOT A RIGHT WAY TO DEFINE A MEMBER FUNCTION OUTSIDE CLASS TEMPLATE.

* So, whenever we define a member function outside the class template then it is compulsory / necessary to define the template with it again.

We can do this with the following code

→ **CODE:**

```
template <class t>
class Algebra {
        t a;

    public:
        void output(); // DECLARATION INSIDE CLASS

        t getA(); // DECLARATION INSIDE CLASS
};

//TEMPLATE FOR MEMBER FUNCTION 'OUTPUT':
template <class t>
void Algebra <t> :: output() {
        cout << "a = " << a;
}

//TEMPLATE FOR MEMBER FUNCTION 'getA':
template <class t>
t Algebra <t> :: getA() {
        return a;
}
```

```
int main () {
    Algebra<int> obj1;

    obj1.output ();
    cout << obj1.getA() << endl;
}
```

* Notice that the code 'template <class t>' is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

**NOTE:**

* We have to write/define template again and again with the member function each time we define member function outside the class.

* There is no need to define/create template inside the class. (with the member functions)

# GLOBAL FUNCTION FRIEND OF CLASS TEMPLATE

* Global functions are not member of our class they can only be friend of our class. So, when we write global function and make them friend of our class template, then we must have to write/define template with their defination (outside the class) and write/define template inside the class, where we make them friend of our class.

## ↳ IF WE DO THIS:

### (FOR STREAM INSERTION OPERATOR)

### LINKEN ERROR

```
template <class t>
class Algebra {
    t a, b;

public:

    // CODE

    // LINKEN ERROR X

    friend ostream& operator<< (ostream &, const Algebra<t2>&);
};

template <class t2>
ostream& operator<< (ostream& out, const Algebra <t2>&obj){

    out << a << b;
    return out;
}
```

**NOTE:-**

* It gives linken error bcz stream insertion operator is a global function. So, we must also need to define template inside class before writing friend

* Whenever we have a global function which is friend of class template, we must write / define template for it before writing friend (inside the class) and ~~after~~ before defining the global function (outside the class). Otherwise we get an linken error.

## ↳ SO TO MAKE IT VALID:

### CODE (SYNTAX)

```
template <class t>
class Algebra {
    t a, b;
public:
    // CODE

    template <class t2>
    friend ostream& operator<< (ostream &, const Algebra<t2>&);
};

template <class t2>
ostream& operator<< (ostream& out, const Algebra<t2>&){

    out << a << b;
    return out;
}
```

This data type name must be different from data type name of template of global function. If we write its name as 't2' it gives an error.

This data type name must be differen from data type name of class template. If we make its name 't' it gives an error

**NOTE:-**

* Global function ↑ template and class template are different.

* Global functions have no link with the class.

* Global function is not member of our class.

* We must have to re-write the templates of global functions inside the class.

**IMPORTANCE / BENEFITS OF TEMPLATES:**

* Templates avoids / eliminates overloading.

**EXAMPLE: (OF CLASS TEMPLATE)**

```
//CLASS TEMPLATE FOR DIFFERENT DATA TYPES
template <class t>
class Algebra {
        t a, b; //DATA MEMBER OF TYPE 't'
                ('t' type will be difened by user)
public:
    //DEFAULT CONSTRUCTOR
    Algebra(){
        a=b=0;
    }
    //PARAMETRIZE CONSTRUCTOR
    Algebra(t a, t b){
        this->a = a;
        this->b = b;
    }
    //MEMBER INPUT FUNCTION
    void input(){
        cout << "Enter a: ";
        cin >> a;

        cout << "Enter b; ";
        cin >> b;
    }
```

```
//MEMBER FUNCTION TO DISPLAY DATA
//ONLY DECLERATION OF MEMBER
                OUTPUT FUNCTION
    void output ();


//MUST WRITE THE TEMPLATE INSIDE
    CLASS FOR FRIEND FUNCTION
    template < typename t2> --> must be different from class template
    friend ostream& operator<< (ostream &, const Algebra<t2>& );
};

//DEFINE TEMPLATE AGAIN, WHEN DEFINING
  ANY MEMBER FUNCTION OUTSIDE OF
    CLASS
template < typename t>
void Algebra<t> :: output () {
        cout << a << b;
}
//TEMPLATE FOR FRIEND (NON-MEMBER OF CLASS)
      STREAM INSERTION OPERATOR
template <typename t2>
ostream& operator<<(ostream& , out, const Algebra<t2> obj){
        out << obj.a << obj.b;
        return out;
}
```

```
int main () {

    Algebra <int> obj1;  //TEMPLATE CLASS
    cout << obj1;          FOR 'int'.

    Algebra <float> obj2; //TEMPLATE CLASS FOR
    cout << obj2;           ('FLOAT')

    Algebra <char> obj3; //TEMPLATE CLASS FOR
    cout << obj3;          'char'

    return 0;
}
```

see file

↓

class_template.cpp

# CLASS TEMPLATE WITH MULTIPLE PARAMETER

(2) FOR MORE THAN ONE DATA TYPE AT A TIME:

* In C++, we can use multiple template parameters and even use default arguments for those parameters.

**FOR EXAMPLE:-**

here means 'V' is a default parameter whose default type is 'int'.

```
template < class T, class U, class (V= int)>
class ClassName {
        T mem1;
        U mem2;
        V mem3;
            // CODE
    public;
            // CODE
};
```

* In this program, we have created a class template, named 'ClassTemplate', with three parameters, with one of them being a default parameter.

```
template <class T, class U, class V = char>
class ClassTemplate {
        // CODE
};
```

* Notice the code "class V = char". This means that 'V' is a default parameter whose default type is 'char'.

* Inside "ClassTemplate", we declare 3 variables 'var1', 'var2' and 'var3', each corresponding to one of template parameters.

```
class ClassTemplate {

    T var1;
    U var2;
    V var3;
    . . . . .
```

* In 'main()' we create two objects of 'ClassTemplate' with the code:

```
//CREATE OBJECT WITH INT, DOUBLE AND
                        CHAR TYPES
ClassTemplate <int, double> obj1(7, 7.7, 'c');

//CREATE OBJECT WITH DOUBLE, CHAR AND BOOL TYPE

ClassTemplate <double, char, bool> obj2(8, 8.8, false);
```

# COMPOSITION

"Inheritence" means reusability of code.

* Inheritence can also be done by "Association."

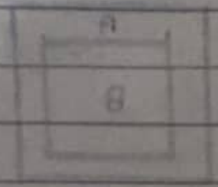
## ASSOCIATION:

If there exists / have any relationship between two objects (class), then it is called association.

- **Association types:**

Association have two types / categories:

- Composition
- Aggregation



# ASSOCIATION

COMPOSITION — AGGREGATION

both means 'reusability' of code

reusability means "has/have-a" relationship

| COMPOSITION | AGGREGATION |
|---|---|
| (1) Composition means reusability. | (1) Aggregation means reusability. |
| (2) Composition means 'has/have-a' relationship. | (2) Aggregation means 'has/have-a' relationship. |
| (3) Composition means something will be composed in another object. | |


'B' is composed in class 'A'

'A' is container.

'B' is composed / contained.

## ↳ DIFFERENCE B/W COMPOSITION AND AGGREGATION:

### • COMPOSITION:

* In composition the life cycle of contained/ composed object depend on container object.

* Composition has a "one - to - one" relationship.

* As life cycle of composed object depend on container object.
So, if container exists then composed object exists only.
And if container destroy's then composed object also destroyed.

## REAL LIFE :-

Complex objects is composed / made up of many many simpler objects.

**(1) EXAMPLE :-**

CAR ← complex object

- → Engine (itself a complete object)
- → Gear (itself a complete object.)
- → Electric (complete object)
- → Mirror (complete object)
- ⋮

∴ So, Car is a complex object made up of many simple objects (i.e gear, engine, electric, mirror etc)

**(2) EXAMPLE :-** COMPUTER

- → CPU
- → RAM
- → ROM
- → HD/SSD

} are all itself complete objects.

- ⋮

∴ So, computer is a complex object made up of many simple objects.

∴ CPU, RAM, ROM, HD/SSD are composed in computer.

★ These (example-1 and example-2) relationships are known as **"Has-a"** relation.

★ Composition is defined / called **"Has-a"** relationship.

**NOTE:**

★ When we have pointer as data member than it is necessary to write copy constructor. (bcz in each constructor we allocate heap memory for pointer objects.)

★ When we have simple (not pointer) data member in our class then there is no need to write copy constructor.

★ HOW TO MAKE COMPOSED OBJECTS:

EXAMPLE:-

```
class CPU {
    int code;
    string name;

public:
    //PARAMETRIZE CONSTRUCTOR OF CPU
    CPU(int code, string name) {
        this->code = code;
        this->name = name;
    }
        :
        :
    void print() {
        cout << code << endl;

        cout << name << endl;
    }
};
```

∴ string is a whole class. String is composed in class CPU.

//CODE CONTINUE NEXT PAGE

```
class Computer {
    int year;
    string manName;
    //DEVELOPING A RELATIONSHIP B/W
    CPU AND COMPUTER CLASS.
    CPU mCpu;
```

return type CPU ← CPU mCpu; → object as data member.

object name of return type CPU.

//NOW CPU OBJECT IS COMPOSED IN CLASS COMPUTER.

```
public:
    //PARAMETRIZE CONSTRUCTOR OF COMPUTER CLASS
    Computer(int year, string manName) {
        :
        :
    }
    //PRINT MEMBER FUNCTION
    void print() {
        cout << year << manName;
        cout << mCpu.code << mCpu.name; ✗ ERROR
```

∴ we want this print function, to print both data members of Computer as well as CPU class.

bcz both code and name are private. So, we cannot access them like this

//SO IF WE NEED TO ACCESS THEM THEN WE NEED TO CALL PRINT FUNCTION OF CPU CLASS.

ABSTRACTION: here also inner details of CPU are hidden.

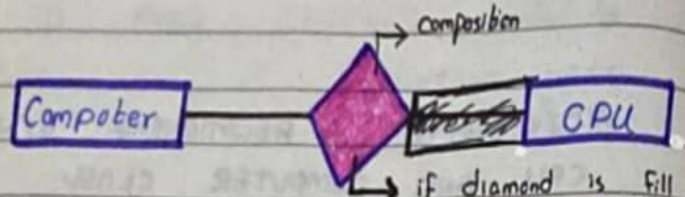
```
        mCpu.print(); ✓ VALID
```

bcz print is a public function. So, we can access easily

```
        //NOW IT PRINTS CONTENTS OF CLASS CPU. THIS IS
    }
```

REUSABILITY

DIAGRAM IN FORM OF UML:




IN MAIN:

```
int main () {
    Compoter obj;    // HERE WE MAKE OBJECT OF COMPUTER CLASS.

    obj.print();
        ↳ here compiler 1st print year, then manName the CPU detail (by calling CPU print function).

    return 0;
}
```

**NOTE:**

*** IN COMPOSITION**

* In container class we can access the public member function or data member of composed object by using '.' operator.

Like: Syntax:

```
object.publicMemberFunction/dataMember;
name
```

In previous Example: here print is a public member function.

```
mCpu.print();
```

(But keep in mind only public data member and member functions can be access like this, not private).

* We cannot access private data members or member function of composed objects in Container class.

Like: previous example: bcz here both name and code are private data member of composed class.

```
mCpu.name;
mCpu.code;
```

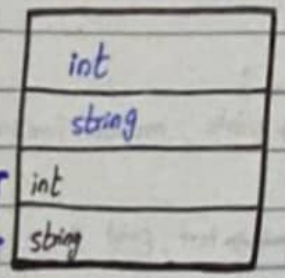
} ERROR X

(It gives an error because here 'ABSTRACTION' also exists.
Here also inner details of CPU are hidden).

**IF IN MAIN:**

```
Computer C;
C.print();
```

**COMPUTER C**




∴ object is itself nothing it is only made up of data/functions members.

***WE CAN WRITE MULTIPLE COMPOSED OBJECTS IN CONTAINER OBJECT (CLASS)!**

```
class CPU {          //CPU CLASS
    :
};

class Computer {     //COMPUTER CLASS
    int year;
    CPU obj1;
    CPU obj2;
    CPU obj3;
    :
    etc
}
```

} COMPOSED OBJECTS OF CLASS CPU IN CONTAINER COMPUTER CLASS.

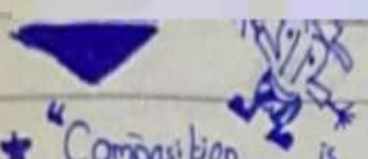
**IF WE HAVE:**

```
class CPU {          //CPU CLASS
    :
};

class RAM {          //RAM CLASS
    :
};

class Computer {     //COMPUTER CLASS
{   int year;
    string manName;

    CPU mCpu;
    RAM ram;
    CPU pCpu;
public:
    Then run its (computers) default, parameter
    constructor for its own data members.
};
```

① Compiler 1st go to CPU class.

② Then go to RAM class.

③ Then again go to CPU class.



// 3.60

★ "Composition is a strong binding among the objects."

★ Composition achieved through normal as-well-as pointer variables but not achieve from reference variable.

★ In composition life of compose object depends on life of container objects. if we use pointer variable in container class to make object of composed/contained class. And we use "new" to allocate memory for it in constructors; And use "delete pointer name" to (deallocate this memory) delete contained class object inside the destructor of container objects, then it is example of composition through pointer variables.

> Composition can also be done through pointers (when life cycle of pointers ~~depend~~ (composed) object depends on life cycle of container class.

**EXAMPLE: COMPOSITION THROUGH POINTER DATA MEMBER.**

```
class Teacher {
        int eCode;
        string tName;
    public:
        Teacher(int eCode, string tName) {
            this->eCode = eCode;
            this->tName = tName;
        }
        .
        .
        .
};

class Department {
        int dCode;
        string dName;
        Teacher* teacher;   // NOT IDEA AT THIS TIME IS IT COMPOSITION OR AGGREGATION
    public:
        Department(int dCode, string dName) {
            this->dCode = dCode;
            this->dName = dName;

            teacher = new Teacher(0, " ");   // HERE NEW OBJECT OF COMPOSED CLASS IS CREATED.
        }
```

```
~Department () {
    delete teacher; // Here new object (composed) is destroyed when container class is destroyed. (inside container class destructor)
}
};
```

// THUS, IT IS A COMPOSITION NOT AGGREGATION.

**NOTE:**

(1) Whenever we contained an object in container class, then contained class objects is execute 1st then container class itself data members.

(2) We can compose one or more (many) objects of one class (composed) into another class (container). Simply by using return type of composed class with the object name.

like:

class(return type) name

previous example:

```
CPU mCpu; // COMPOSED CLASS CPU OBJECT.
```

(3) When the constructor of container class (in which the object of another class is composed) are called, then we can also call the constructors of composed objects in member initializer list of constructor of container class. If we don't call any constructor of composed class in member initializer list, then it would simply call default constructor of composed object.

(4) If we want to call the constructor (of our choice) of composed/contained class, then we call them in member initializer list of constructor of containg class.
If we don't call any constructor in the member initializer list for composed objects, then the default constructor will be called automatically (of composed object).
No matter whether the default, parametrize or copy constructor have been called / has executed of container object (class).
If we haven't mentioned the constructor of composed object in member initializer list (constructor of containg object). Then, it would simply called the **DEFAULT CONSTRUCTOR** of composed / contained object.

(5) If we write any constructor in member initializer list (of containg object) i.e Copy or Parametrize or Default Constructor, then it would simply called that Constructor of composed object.

(6) We cannot call the constructors of composed objects inside body of constructor of container object.
It will give an error.

```
Computer (int year, string Name) {
    mCpu (0, " ");   X ERROR
}
```

We can only call/write constructor of composed object in member initializer list of constructor of container object.

a=3

(i) a = ++b;
(ii) a = b++;
a = 2
b = 3

obj1 = ++obj2;
obj1 = obj2++;

(7) We can also give / call constructor of composed object of our choice in member initializer list of container object.

If in member initializer list of parametrize constructor of container class (object) we write / call copy constructor of composed class (object) Then it would simply call copy constructor of composed object and parametrize constructor of container obj.

If in member initializer list of copy constructor of container class (object) we call / write parametrize constructor of composed class object Then it would simply call parametrize constructor of composed object, and copy constructor of container.

(8) We can change calls of any constructor of composed objects in member initializer list of other constructor of container object.

## ↳CALLING CONSTRUCTORS OF OUR CHOI

(i) IF WE WANT TO CALL "DEFAULT CONSTRUCTOR" OF CONTAINER OBJECT AND "DEFAULT CONSTRUCTOR" OF COMPOSED OBJEC

So, in container class:

```
Computer ()
{
    makeYear = -1;
    name = " ";
}
```

OR

```
Computer () : m_Cpu()
{
    makeYear = -1;
    name = " ";
}
```

SAME

```
~CPU (){
    cout << "CPU destructor" << endl;
}

void input() {
    cout << "Enter CPU model: ";
    cin >> model;

    cin.ignore();

    cout << "Enter CPU name: ";
    getline (cin, name);
}

void print () {
    cout << model << "\n" << name << endl;
}
};
```

// COMPUTER (CONTAINER CLASS)

```
class Computer {
    int makeYear;
    string name;

    CPU m_Cpu; // CPU's object is composed
               in Computer.
```

// THE LIFE CYCLE OF m_CPU depends on Computer's object.

```
public:
    Computer ()
    {
        makeYear = -1;
        name = "";
    }
```

// As life cycle of (composed) CPU depends on (container) Computer. CPU is build when Computer is already builded. Thus, this is composition.

```
Computer(int comYear, string comName, int cModel, string cName)
  : m_Cpu(cModel, cName)
{
    this->makeYear = comYear;
    this->name = comName;
}

Computer (const Computer &obj) : m_Cpu (obj.m_Cpu)
{
    makeYear = obj.makeYear;
    name = obj.name;
}
~Computer (){
    cout << "Computer's Destructor" << endl;
}
```

## EXAMPLE: COMPOSITION BASIC

```
// CPU (COMPOSED) CLASS
class CPU {
    int model;
    string name;

    public :
        CPU () {
            model = -1;
            name = " ";
        }

        CPU ( int model, string name ){
            this -> model = model;
            this -> name = name;
        }

        CPU (const CPU &obj) {
            model = obj.model;
            name = obj.name;
        }
```

```
void input () {

    cout << "Enter Computer Make Year:";
    cin >> makeYear;

    cin.ignore();

    cout << "Enter Computer Name: ";
    getline(cin, name);

    m_Cpu.input();

}

void print() {
    cout << makeYear << "\n" << name;

    m_Cpu.print();
}
};
```

```
int main() {

    Computer obj1, obj2(2002, "DELL", 543, "Gen");

    Computer obj3(obj2);

    obj1.input();
    obj1.print();

    return 0;
}
```

see FILE

(1) Composition_basic.cpp

(2) Composition_change_Calling_sequence01.cpp

(3) Composition_change_calling_sequence02.cpp

NO

## COPY CONSTRUCTOR:

* If we don't write any copy constructor in both classes i.e (container class and composed class)

then if in main function we call a copy constructor for container object. like:

```
Computer obj1, obj2(obj1);
```

Then, compiler will call simply (its own) copy constructors for both container as well as composed object.

**AS WE KNOW:**

* When no copy constructor written/available, compiler makes its own copy constructor.

## IF WE WANT TO WRITE 'INPUT FUNCTION':

In container class:

```
void input() {

    cin >> makeYear;

    cin.ignore();

    getline(cin, name);

    cin >> m_Cpu.code; ✗ // ERROR bcz code is a private data member of CPU class.

/* THUS WE MUST HAVE TO WRITE INPUT MEMBER FUNCTION IN CLASS CPU AISO. AND ACCESS THE INPUT FUNCTION IN THIS COMPUTER CLASS */

    m_Cpu.input(); ✓ // VALID bcz input is public member function of class CPU.
}
```

(vi) IF WE WANT TO CALL "COPY CONSTRUCTOR" OF CONTAINER CLASS AND "PARAMETRIZE CONSTRUCTOR" OF COMPOSED CLASS:
So, in container class:

```
//COPY CONSTRUCTOR OF CONTAINER (COMPUTER)
//MAKING A CALL TO 'PARAMETRIZE CONSTRUCTOR' OF CPU. 'DEFAULT CONSTRUCTOR' WILL BE EXECUTED OTERWISE:

Computer(const Computer &obj, int cMod, string cName)
  : m_Cpu(cMod, cName)
{
        makeYear = obj.makeYear;
        name = obj.name;
}
```

(vii) IF WE WANT TO CALL "COPY CONSTRUCTOR" OF CONTAINER CLASS AND "COPY CONSTRUCTOR" OF COMPOSED CLASS:
So, in container class:

```
//COPY CONSTRUCTOR OF COMPUTER
//MAKING CALL TO 'COPY CONSTRUCTOR' OF CPU. OTHERWISE 'DEFAULT' WILL EXEXUTE:

Computer(const Computer &obj) : m_Cpu(obj.m_Cpu)
{
        makeYear = obj.makeYear;
        name = obj.name;
}
```

(v) IF WE WANT TO CALL "COPY CONSTRUCTOR" OF CONTAINER CLASS AND "DEFAULT CONSTRUCTOR" OF COMPOSED OBJECT (CLASS)

So, in Container class:

```
//COPY CONSTRUCTOR OF COMPUTER
//MAKING A CALL TO DEFAULT CONSTRUCTOR OF CPU.

Computer(const Computer &obj)   //bcz if we don't give any call in member initializer list then it will simply call default constructor of composed class itself.
{
    makeYear = obj.makeYear;
    name = obj.name;

}
```

OR

```
Computer(const Computer &obj) : m_Cpu()
{
    makeYear = obj.makeYear;
    name = obj.name;
}
```

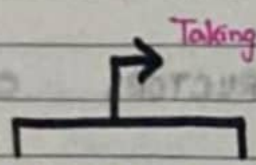
★ If we write copy constructor in container class but don't write any copy constructor in composed class and then in main we call copy for container class (Computer):

Computer obj2(obj1);

then compiler will simply call copy for container class and default for composed class.

(iii) IF WE WANT TO CALL "DEFAULT CONSTRUCTOR" OF CONTAINER OBJECT AND "PARAMETRIZE CONSTRUCTOR" OF COMPOSED OBJECT:

So, in container class.




```
Computer(int cModel, string cName): m_Cpu(cModel, cName)
{
    makeYear = -1;
    name = " ";
}
```

Data members of container (computer) class.

(iii) IF WE WANT TO CALL "PARAMETRIZE CONSTRUCTOR" OF CONTAINER OBJECT AND "DEFAULT CONSTRUCTOR" OF COMPOSED OBJECT.

So, in container class.

```
Computer(int Computer_m_Year, string computer_Name)
{
    this->makeYear = computer_m_Year;
    this->name = computer_Name;
}
```

(iv) IF WE WANT TO CALL "PARAMETRIZE CONSTRUCTOR" OF CONTAINER OBJECT AND "PARAMETRIZE CONSTRUCTOR" OF COMPOSED OBJECT:

So, in container class.

```
Computer(int comYear, string comName, int cModel, string cN
: m_Cpu( cModel , cName)
{

    this->makeYear = comYear;
    this->name = comName;
}
```

Lecture #05 {MAKE-UP LECTURE}

# AGGREGATION

★ "Aggregation is a weak binding."

★ In aggregation, Lifecycle of contained/composed object does not depends on container class.

★ In aggregation, container exists independently.

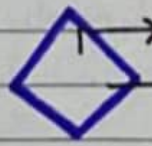
★ No matter whether the container build/exists or destroys, but contained object always exists.

★ Only services (MEMBER FUNCTIONS ETC) of contained object can be used in container class, but the life cycle of contained/composed class does not depends on container class.

★ In aggregation every class/thing exists

## UML DIAGRAM:

★ If the diamond is not filled then this "has-a relationship" is of aggregation.



empty diamond means aggregation "has-a relationship".

## ↳ HOW TO BUILD AGGREGATION:

★ Aggregation acheived/done through pointer or refrence variables only.

★ Aggregation can not acheived through normal variable.

```
    int eCode;
    string tName;
public:
    .
    .
    .
};

class Department {
    int dCode;
    string dName;

    Teacher *teacher; // NOT IDEA AT THIS TIME. IS IT COMPOSITION OR AGGREGATION.
public:
    Department(int dCode, string dName) {
        this->dCode = dCode;
        this->dName = dName;

        teacher = new Teacher(0, " "); // Here object teacher of composed class get a new memory on heap (allocated)
    }
    ~Department() {
        delete teacher; // Here new object (composed) is destroyed when containing class is destroyed. (inside container class destructor).
    }
```

★ This above (previous page) code is done through composition. As we have allocated new memory to it (pointer) and so, after function, it is destroyed with the destruction of container class. (So, its life cycle is dependant)

★ For aggregation, composed / contained class must already build / already exists before container class. Otherwise, aggregation not possible.

**EXAMPLE:** **AGGREGATION CODE:**

```
class Teacher {
    int eCode;
    string tName;

public:
    Teacher(int eCode, string tName) {
        this->eCode = eCode;
        this->tName = tName;
    }

    void teacherInfo() {
        cout << "E-Code:" << eCode <<
        "T-Name:" << tName << endl;
    }
};
```

```
class Department {
    int dNo;
    string dName;

    Teacher *teacher; // Holds a reference t
                      Teacher : Aggregation
                      (weak binding
    //OR
    //Teacher &teacher;

public:
//WE WILL TAKE TEACHER'S OBJECT IN ARGUMENT
Department(int dNo, string dName, Teacher *teacher)
{
```

here only address of teacher type came nothing else. We will only declare its adress in arguments (It only holds adress of object)

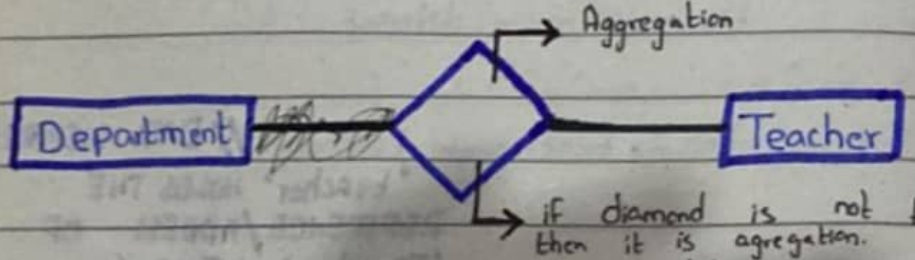
```
    this->dNo = dNo;
    this->dName = dName;

    this->teacher = teacher; // DATA MEMBER
```

'teacher' HOLDS THE REFERENCE/ADRESS OF 'Teacher' TYPE. (Here we are assigning the adress of Teacher To an object of Teacher class declared in department class.)

```
}
```

```
cout << "d-No: " << dNo << "d-Name: "
     << dName << endl;

if (teacher != NULL) //IF teacher exists
    //WE CAN ONLY CALL PUBLIC MEMBER FUNCTION OF COMPOSED CLASS
    LIKE:
    teacher -> teacherInfo(); //DISPLAY teacher INFORMATION
    //As teacher is a pointer object. So, we can only access its member functions by '->' operator, not by '.' operator.
}
};
```

**DIAGRAM IN FORM OF UML:**




**IN MAIN:**

```
int main() {
    Teacher t2(456, "Shahid");
    Teacher t1(123, "Umair"); //Teacher's class object.
        //&t1 refcrence of teacher.
    Department d1(1, "DIT", &t1);

    d1.departmentInfo();
}
```

- only pass adress of Teacher class object.
- We can also pass 'array' here. like:
  d1(1, "A", array);
  bcz 'array' also always pass by reference.

**//OR**

- We can also pass heap allocated memory like:
  new int
  new float
  etc.

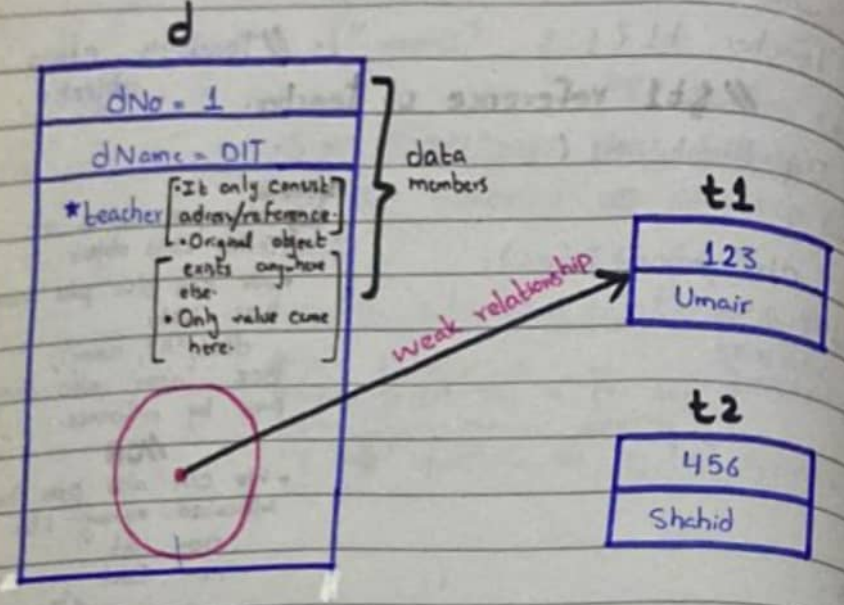
★ In aggregation object is always pass by reference

★ This, relationship b/w array and function is also a weak binding.

see File → aggregation_basic.cpp

**IN DIAGRAM:**

Department d ( . . . )




★ Here, objects 'd' and 't1' have a/ has a relationship of Aggregation. IF any one of them destroys, then the life cycle of other one does not get affected.

★ keep in mind Life-cycle.

**NOTE:**

**IN AGGREGATION:**

★ In container class we only access the public member function or data members of composed class (object) by using '→' operator. Because in aggregation we have a pointer object. And a pointer can access functions/variable by arrow '→' operator only.

Like: **SYNTAX:**

pointer object name → public member functions/ data members ;

**PREVIOUS EXAMPLE:**

teacher → teacherInfo(); teacherInfo() is a public member function.

(But keep in mind only **public** data members and member functions can be accessed like this, not **private**).

★ We cannot access **private** data members or member function of composed pointer objects in container class.

**LIKE: PREVIOUS EXAMPLE:**

teacher → eCode; } **ERROR X**
teacher → tName; } bcz both eCode and tName are private data members of composed class.

(It gives an error bcz here **ABSTRACTION** also exists).

∴ Here also inner details of teacher class are hidden.

**IF WE DO:**

```
class Teacher {
    int eCode;
    string tName;

    public:
        :
        :
        :
};

class Department {
    public: //EVERY DATA MEMBER AND MEMBER FUNCTION PUBLIC NOW.

        int dNo;
        string dName;

        Teacher *teacher;

        Department (int dNo, string dName, Teacher *teacher);
            this->dNo = dNo;
            this->dName = dName;
            this->teacher = teacher;
        }
        :
        :
};
```

**AND IF IN MAIN:**

```
int main() {
    Teacher t1 (100, "Umair");
    Teacher t2 (101, "Shahid");

    Department d (1, "DIT", &t1);
    d.print();

    //AND NOW
    //ACCESSIBLE IN MAIN BCZ teacher is PUBLIC NO
    d.teacher = &t2;

    d.print(); //'t1' does not destroy here only d teacher object in department now consist 't2' as a teacher. From here we get Life cycle of composed (t1 teacher) does not depend on container department. So, it is aggregation
    t1.print();

    //IF WE DO
    d.teacher = NULL; //Here also 't1' and 't2' does not destroys. Only teacher object in department class now hold value of Teacher as NULL. means no teacher exists in department class now.
    d.print();
    t1.print();
    t2.print();
}
```

see File → aggregation_concept.cpp

# DEFAULT PARAMETERS

## DEFAULT PARAMETERS / DEFAULT ARGUMENT LIST:

★ We can also give default parameters to arguments of functions.

★ This concept is also known as polymorphism (function overloading).

**EXAMPLE:**

```
int add (int a=0, int b=0, int c=0) {
        return a+b+c;
}
```

These are default arguments or default parameters.

// THIS FUNCTION IS CAPABLE TO DO:

```
add();          0
// only 'a' overwrite → add(5);      5
// 'a' and 'b' overwrite → add(1,2); 3
// 'a' and 'b' and 'c' overwrite → add(1,2,3); 6
```

★ We can write default as well as parametrize constructor one time by using DEFAULT PARAMETER / DEFAULT ARGUMENT LIST (Function Overloading)

★ Now we don't need to write default constructor separately.

★ If we use default parameters then constructor will act both as default or parametrize (overloaded)

**LIKE:**

default parameters

```
Algebra (int a=0, int b=0)
{
    this → a = a;
    this → b = b;
}
```

// THIS CONSTRUCTOR WORS FOR BOTH DEFAULT AS WELL AS PARAMETRIZED CONSTRUCTOR (overloaded

*IN AGGREGATION DEFAULT PARAMETER

↳IN CONTAINER CLASS CONSTRUCTOR

Department(int dNo, string dName, Teacher *teacher = NULL);
↳ default parameter.

//
//

}

# HEADER FILES

★ Header file does not compile.
Header file only consists declarations.

★ Extension of header file is ".h".

★ Only Cpp file can compile.
Cpp file consist ~~code~~ whole code (definations)

★ If we make header file separately and Cpp file separately. Then our header file must be included in Cpp file.
LIKE:

#include "____.h"
↳ header file name.

•) Algebra.h → header file

••) Algebra.cpp → Cpp file

★ Cpp file exists on std.

** We can include #include "iostream" in header file as well as in Cpp file.

* If we include "iostream" in header file, then we don't need to add/include "iostream" in Cpp file and vice versa.

**RECOMMENDATION:** → To include "iostream" only in Cpp file.

**HEADER FILE:** (How to create header files.)

#progma once → means only '1' time included if user add that header file multiple times.

```
Class Algebra {
    int a, b;
public:
    Algebra();
    ~Algebra();
    input();
    print();
};
```

→ Only declarations.

* Now, by creating header file and Cpp files separately, We give only our header file to user for his work.

Lecture#06

Lecture:- 21-0x-2x

# INHERITENCE

★ Inheritence is also a concept of reusability of code.

★ In composition or aggregation we cannot change/enhance anything of composed object class.
We can also not typecast anything in composed object class.
We can only use features of composed/contained class, not inherit that features.

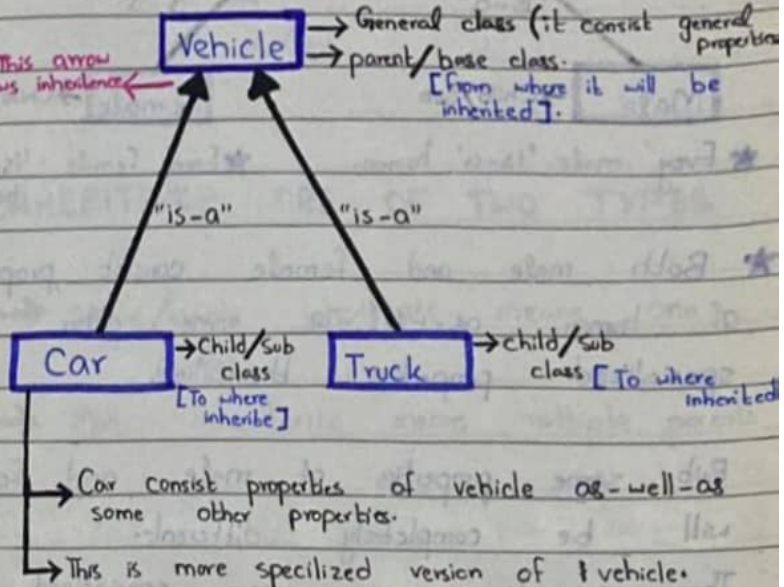
★ In inheritence, the features of main class (PARENT/BASE) will be inherited to the lower class (CHILD).

★ In inheritence we can use or enhance feature of main (parent) class in inherited (child) class.

★ Inheritence is "is-a" relationship.

IN UML CONTEXT:

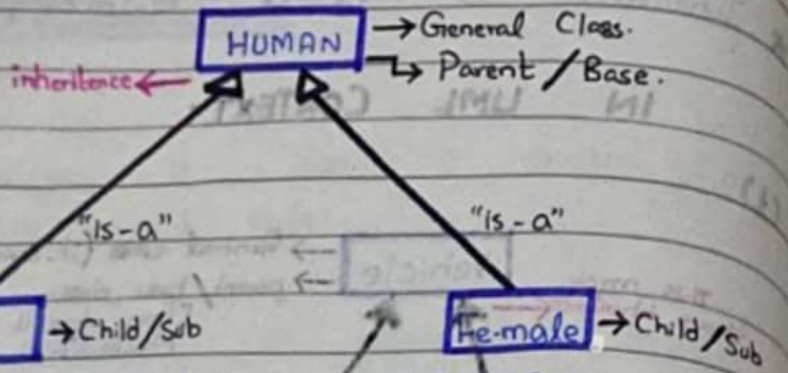
(1)




★ Every car "is-a" vehicle.
★ Truck "is-a" vehicle.

NOTE:
- '↑' this arrow shows association.
- '△' or '▲' this arrow shows inheritence.

(2)




★ Every male 'is-a' human.

★ Every female 'is-a' human.

★ Both male and female consist properties of human as-well-as some other specialized properties themselves.

But some properties of male and female will be completely different.
Those properties are more specialized properties for themselfs.

★ Only those properties will be same (common) that comes (inherit) from human.

★ In inheritence we cannot make any composed object like in association.

Teacher *teacher;

OR

CPU m_Cpu; etc.

↳ INHERITENCE ARE OF TWO TYPES:

→ single / singly inheritence means one parent.

→ Multi inheritence means multiple parents.

## ★ SYNTAX / METHOD TO MAKE ONE CLASS INHERIT OF OTHER.

→ In Inherit / Child class.

child class : Access Identifier Parent Class Name
(private / public / protected)

↳ Real inheritence is this But Cpp give us right to do private or protected inheritence as-well.

★ The class within which to be inherit (child class) has to be colonized after the declaration (of child class) and then the access identifiers (public/private/protected) has to be written and then the name of parent class has ~~be~~ to be written next to it.

**LIKE:**

```
//PARNT CLASS: 'A'.
class A {
        :
        :
};
```

```
//CHILD CLASS 'B'.
class B : public A
{
        :
        :
        :
};
```

(child class declaration; parent class name; access identifier)

// Now features of 'A' parent class inherits in 'B' child class. //

**NOTE:**

★ If no access identifier is/will be written after colon then the inheritence will be consider as private.

**LIKE:**

no access identifier means private inheritence.

```
class B : A
{
//
};
```

# INHERITENCE TABLE (which type of inheritence will be done)

| PARENT CLASS | INHERIT/CHILD CLASS | | |
|---|---|---|---|
| | PRIVATE | PROTECTED | PUBLIC |
| PRIVATE | private | private | private |
| PROTECTED | private | protected | protected |
| PUBLIC | private | protected | public |

★ Protected members of parent class are directly accessible to child class.

★ In inheritence, most restricted apply.

★ In inheritence, the one that is more restricted will occur/exists.

**RESTRICTNESS BEHAVIOUR**

private > protected > public.

**NOTE:** In inheritence:

★ Every member function/data member can be inherited except.

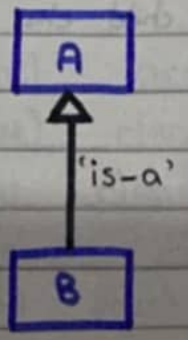
(1) Constructors
(2) Destructor
(3) Assignment operator

} Cannot be inherited.

★ Every feature of parent class can be inherited in child class except constructors, destructor, assigment operators. These three cannot be inherited.

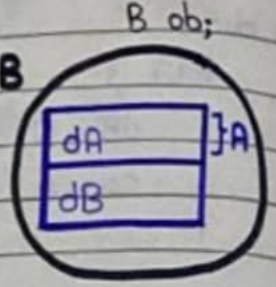
## EXAMPLE: INHERITENCE BASIC

```
// PARENT / BASE CLASS 'A'
Class A {
    int dA;

public:
    A() {
        dA = 0;
        cout << "A's default Constructor";
    }

    ~A() {
        cout << "A's Destructor";
    }

    void printA() {
        cout << "dA = " << dA << endl;
    }
};
// CHILD CLASS 'B'
class B : public A          // 'B' inherits all features of 'A' except Constructor, Destructor and Assignment operator.
{
    int dB;

public:
    B() {
        dB = 0;
        cout << "B's default Constructor";
    }

    ~B() {
        cout << "B's destructor";
    }

    void printB() {
        printA();           // In inheritence, we don't need to write "obj.print" like association.
                            // We can simply call the public functions of parent class. Because now those features inherited in child class.
        cout << "dB = " << dB << endl;
    }
};
```




- 'B' is inherited in 'A'.
- 'B' is a 'A'.

**IN MAIN:**




```
int main () {

    B ob;

    ob.printB();

}
```

see file → inheritence_basic.cpp

**NOTE:**

* Most base class (parent class)'s Constructor will be call first (1st). Then child class constructors will be called one-by-one.

**OUTPUT:**

```
A's constructor.   // Most base class constructor called 1st.
B's constructor.   // Then child class constructor will be called.
  dA=0
  dB=0
B's destructor.
A's destructor.
```

**IF IN MAIN:**

```
int main() {
    A oA;
    B oB;

X   cout << oB.dA;
```

→ It is private data member of parent class. So, it is inaccessible.

→ inaccessible

✓ `oB.printA();` → Accessible. Because 'B' is a child of 'A'. So, 'B' can use/access all public data members or member functions of parent class 'A'. 'B' is inherited ie 'A'.

X `oA.printB();` → Inaccessible.
(Compiler gives an error. It will ask there exists no printB() function in class 'A'.

Bcz 'A' is not inherit in 'B'. 'A' is only pare of 'B'. So 'A' cann access any data member or member function of

```
}
```

* We cannot access private members of parent (base) class in main or in child class. We can only access public members of parent class in main through child object or in child class.

★ We can access public member functions or data members of parent class in child class.

**LIKE:**

```
void printB () {
    printA();
}
```

In inheritence, we don't need to write "obj.printA()" like association.
We can simply call the public member function of parent class. Bez now these features inherited in child class.

```
//If we make data member of parent class public:
    dA = 5;
```

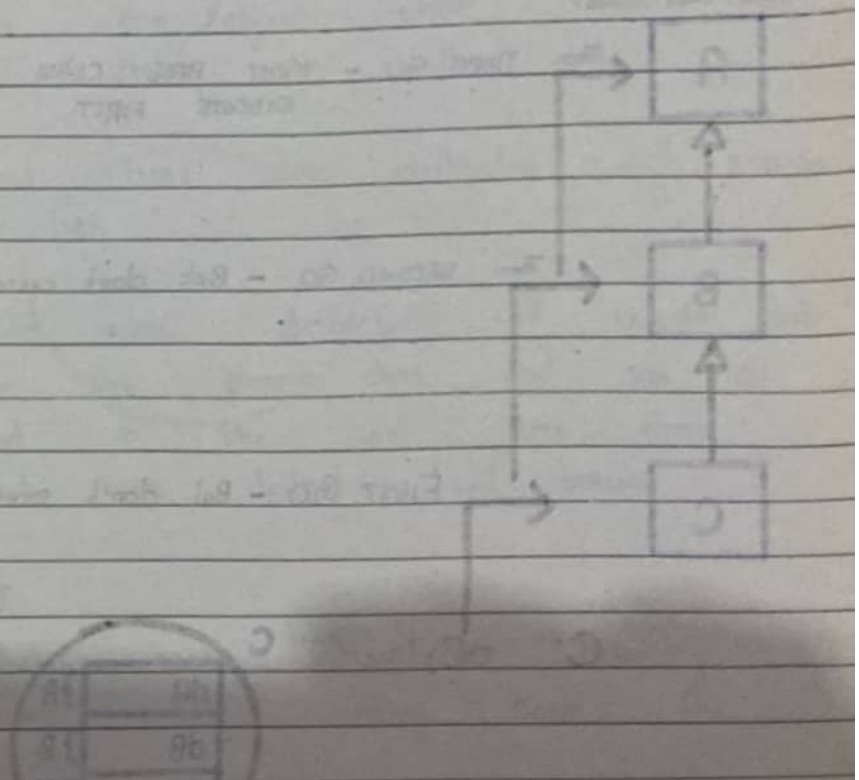
Now data member's value of parent class 'A' is changed to '5'.
We can easily access and change public data members value of parent class.
This changing is not allowed in association.

```
}
```

**NOTE:**

Every member function / data member can be inherited except:

(1) Constructor
(2) Destructor
(3) Assignment operator.



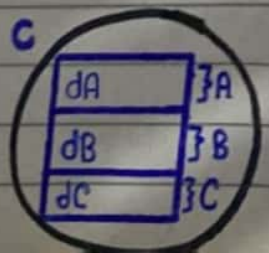
## IF WE HAVE THREE (3) CLASSES IN INHERITENCE: (one parent of other)

★ If we have '3' classes A, B and C. And 'B' is parent of 'C', and 'A' is parent of 'B'. Then, 'A' will be the most base/parent class. And 'C' is a child class.

## INHERITENCE HERARCHY CHART:

Most Base class.


C oC;




## CONSTRUCTOR

★ Most base class constructor will execute first, then 2nd base class then so on. In the last, the constructor of child class will execute.

## BUT IN CASE OF DESTRUCTOR

★ Desbructor always execute reverse.

★ The thing/object which created first will desboy in the last.

★ Most base class desbructor will execute in last.

★ Child class desbructor will execute first. then the parent class and so on. And in the last the most parent class desbructor will execute.

**NOTE:**

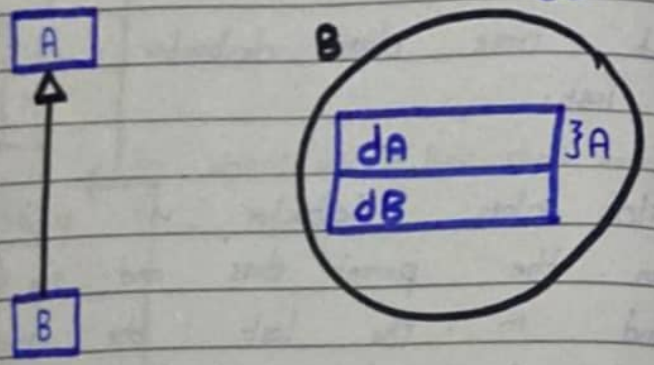
★ Always make inheritence hierarchy chart while writing code for inheritence.

# INHERITENCE

★ Inheritance gives / provides us reusability.

★ Construction done according to hierarchy.

★ If we try to get private members of parent class in child class or in main then we get inaccessible error.




★ PARENT DEFAULT AND CHILD DEFAULT:

```
B ob;

class B : public A
{
//
public: // Here parent's default as-well-as child's default is calling.
B() {
//
}
};
```

★ ACTUAL CALLING / GOING FROM CHILD DEFAULT CONSTRUCTOR TO PARENT DEFAULT CONSTRUCTOR.

```
B() : A()        ← parents and child's default constructor is calling.
{
    dB = 0;
    cout << " ";
}
```

★ IF WE WANT TO CALL 'PARAMETRIZE CONSTRUCTOR' OF CHILD AND 'DEFAULT CONSTRUCTOR' OF PARENT CLASS.

```
B ob(1);

B(int dB)
{
    //
}
```

A → default
B → parametrize
constructor is calling

OR

```
B(int dB) : A()
{
    //
}
```

A → default
B → parametrize
constructor is calling

★ IF WE WANT TO CALL: 'PARAMETRIZE CONSTRUCTOR' OF CHILD AS - WELL - AS 'PARAMETRIZE CONSTRUCTOR' OF PARENT CLASS:

IN MAIN:

```
B ob(1, 2);
```

In main, we pass two values. So, that (child) 'B' parametrize data member as - well - as (parent) 'A' parametrize data member should be assigned.

IN CHILD CLASS (B):

```
// Now here, 1st parent class parametrize constructor is called then child class parametrized constructor will be called.
B(int dA, int dB) : A(dA)
{
    this->dB = dB;

    cout << " // ";

    this->dA = dA; X //ERROR bcz 'dA' is a private data member of parent class. So, is inaccessible in child class.
}
```

A → parametrized
B → parametrized constructor is calling

## ★ PRINT FUNCTION:

```
void printB() {

    printA(); // No need to write 'obj.print' etc. here.
    cout << "\n";
}
```

## ★ COPY CONSTRUCTOR

IF WE WANT TO CALL 'COPY CONSTRUCTOR' OF CHILD CLASS AND 'DEFAULT CONSTRUCTOR' OF PARENT CLASS:

```
B obj2;
B ob(obj2);
```

```
B(const B &ob)
{
    dB = ob.dB;
}
```

A's → Default
B's → Copy constructor is calling.

OR

```
B(const B &ob) : A() {
    dB = ob.dB;
}
```

A's → Default
B's → Copy Constructor is calling.

## ★ IF WE WANT TO CALL 'COPY CONSTRUCTOR' OF PARENT AND 'COPY CONSTRUCTOR' OF CHILD CLASS.

```
B(const B &ob) : A(ob.dA)
{
    dA = ob.dA; // ✗ ERROR bcz 'dA' is a private data member of parent class 'A'. So it cannot access here in child class.
}
```

✗ ERROR bcz 'dA' is a private data member of parent class 'A'. So it cannot access here in child class.

VALID WAY

NEXT PAGE

SO, VALID WAY TO CALL 'COPY CONSTRUCTOR' OF CHILD AND PARENT CLASS AT A T SAME TIME IN INHERITENCE IS:

NOTE:

* We can pass every child class (B) object to parent class (A).

* Every object of child class (B) is also a part of parent class (A).

* Every object of child class (B) can be assigned to its parent (A). Bcz every object of B(child) class is also an object of A(parent) class. But here parent class only deals/see with its own part/data member i.e dA etc.

VALID ✓

```
B(const B &ob) : A(ob)
{
    dB = ob.dB;
    cout << " " << "\n";
}
```

Its type is 'B' but in 'A' (parent) class it will only deal its own data member i.e dA.

# SLICING CONCEPT

## ASSIGNING CHILD CLASS OBJECT TO PARENT CLASS

* In inheritence, typecasting is allowed.

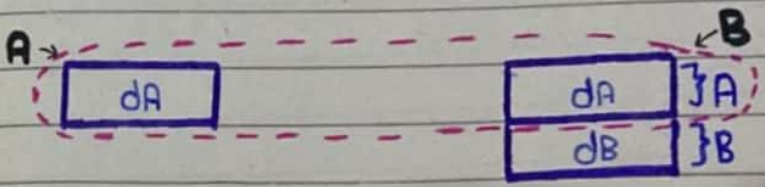
* We can assign every child class object to parent class.

* Every object of child class (B) is also a part of parent class (A).

```
A oa;
B ob;
oa = ob;
```

child class object 'ob' is assigning to parent class object 'oa'.

* It is an example of SLICE.




• SLICE

★ Every object of child class (B) is also part of parent class (A).

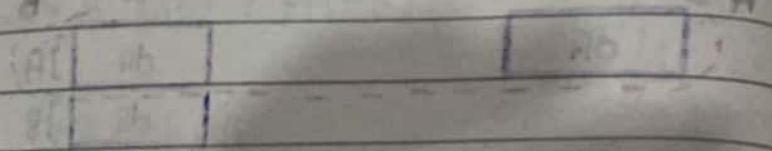
★ Every object of child class 'B' can be assigned to its parent 'A'. Bcz every object of B (child) class is also an object of A (parent) class. But here parent class only deals with its own data member i.e dA etc.

IF,

```
A oa(5);        →  dA = 5
B ob(1, 2);
```

oa = ob; // This is allowed in inheritence but not allowed in composition (association)

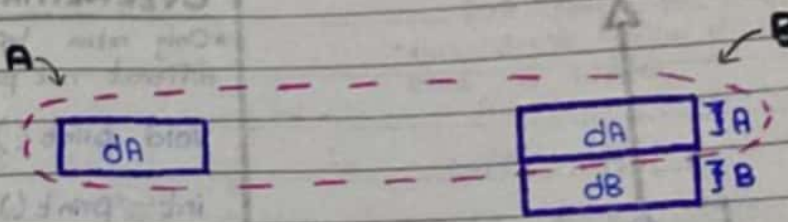
oa.printA(); → dA = 1



type memory

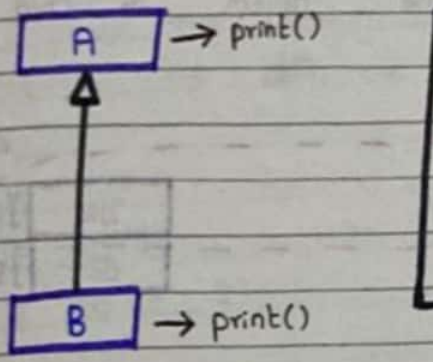
oa = ob;

★ Here compiler only deals/see object of parent class (dA). It will only see type. It will not see memory. It will not see/deals the object of child class (dB)



# SAME NAMES OF MEMBER FUNCTIONS OF CHILD AND PARENT CLASS.

★ If we make member functions name same.




**OVERWRITING :**

★Only return type will different not parameter

```
void print()
int print()
```

★ We make the same name of print member function of both parent and child class. i.e print().

★ If we make the same names of members functions of child and parent class. i.e print() in both child and parent class.

**IN MAIN:**

```
B ob(1, 2);
ob.print();
```

Here 'B' print function is calling. Bcz 'ob' is object of class 'B'. So, it simply always call print function of class 'B'.

• Here print function of base (parent) class is hidden.

★ So, best way to call 'A' print function from 'B' object is by using scope resolution operator.

**LIKE:** 'B' object, 'A' class print function

both are calling independently

```
ob.A::print(); // Here 'A' print function is being called, by 'B' class object
ob.print(); // Here 'B' print function is being called, By own object (type B)
```

```
A oa(1);
oa.print(); // Here 'A' print function is being called. Bcz 'oa' is object of class 'A'. So, it simply call print function of class 'A'.
```

IF, type memory

```
oa = ob;

oa.print();
```

★ 'A' (parent) class' print member function will be called, because the call will be made on the basis of object type not on memory's type.

---

**COMPILER ACTUAL CALLING :** (of print function)

IN MAIN:

```
ob.printA();
```
→ we called

```
ob.A::printA();
```
→ actual compiler call.

IN PRINT MEMBER FUNCTION (inside child class)

```
void printB() {
    printA();
```
→ We call

```
    this->A::printA();
```
→ Actual compiler call.

```
}
```

---

★SAME NAME OF 'PRINT FUNCTION' IN BOTH CHILD AND PARENT CLASS:

SO, IN B (CHILD) CLASS:

★ If we want to call 'A' print function inside 'B' class' print function.

```
void print() {

    print();  // ✗ ERROR: This is recursion, Because 'B' is calling itself print member function again and again.
    cout << " // ";
}
```

★VALID WAY TO CALL DEPENDENTLY ONE TIME

★ So, valid way to call 'A' print function inside 'B' class' print function, is by using scope resolution operator.

→ IN 'B' Class:

```
void print() {

    this->A::print();  // Thus, to make call to 'A' print function we use scope resolution operator. Here 'A' print function is calling.
    cout << dB;
}
```

★ IF WE MAKE 'A' CLASS DATA MEMBERS AND MEMBER FUNCTIONS TO 'PROTECTED':

```
Class A {
    protected:
        int dA;
        //
        //
};

class B : public A {
    int dB;
    public:
        //DEFAULT CONSTRUCTOR
        B() {
✓ VALID     dA = 0;
            dB = 0;
        }
        //PARAMETRIZE CONSTRUCTOR
        B(int dA, int dB) {
✓ VALID     this->dA = dA;
            this->dB = dB;
        }
```

Now, everything will be Accessible in child class directly.

// Now it is accessible here, it don't give an error.

```
        //COPY CONSTRUCTOR
        B(const B &ob) {
✓ VALID     this->dA = ob.dA;
            this->dB = ob.dB;
        }
        //PRINT MEMBER FUNCTION
        void print() {
✓ VALID     cout << dA;
            cout << dB;
        }

IN MAIN:

    ob.dA;
```

// Now it is accessible here, It don't give a error.

// Now, we can directly access data members of parent (A) class and print them in child (B) class.

// ERROR X: Not accessible parent protected data member in main but can be accessible directly in child class.

So, in child class we can deal parent's protected data members as its (childs) own data members.

★ RECOMMENDATION: Is to not make your class data members protected. Always make data member private.

## EXAMPLE: INHERITEN CE

```
// Parent / Base class 'A'
class A {
    int dA;

public: // Default Consbucter 'A'.
    A() {
        dA = 0;
    }
// Parametrize Consbucter 'A'
    A (int dA) {
        this -> dA = dA;
    }
// Copy Consbucter 'A'
    A(const A &oa) {
        this -> dA = oa.dA;
    }
// Desbuctor 'A'
    ~A() {
        cout << "Desbuctor";
    }
// Print Member Function 'A'
    void print () {
        cout << dA;
    }
};

// Child Class 'B'
class B : public A
{
    int dB;

public:
// Default Consbuctor 'B'
    B() {
        dB = 0;
    }
// Parambrize Consbuctor. 'B'
    B( int dA , int dB) : A(dA)
    {
        this -> dB = dB
    }
```

```
//Copy Constructor 'B'
B(const B &ob) : A(ob)
{
    this->dB = ob.dB;
}
//Destructor 'B'
~B() {
    cout << "Destructor B";
}
//Print Member Function 'B'
void print() {
    this->A::print();
    cout << dB;
}
};
int main() {
    A oa(5);
    B ob(1,2);

    ob.A::print();
    ob.print();
    oa = ob;
    oa.print();
}
```

see file

**inheritence_updation.cpp**

# CONSTRUCTORS IN DERIVED CLASSES:

* We can use constructors of base class in derived classes in C++.
* If base class constructor does not have any arguments (DEFAULT CONSTRUCTOR), then there is no need of any constructor in derived class.
* But if there are one or more arguments in the base class constructor then derived class need to pass arguments to the base class constructor.
* If both base and derived classes have constructors, base class constructor is executed first.

## SLICING CONCEPT:

oa = ob;

* We can also assign address of child class to the pointer (object) of parent class.

like:

```
A  *pa;
B  ob(3, 4);

pa = &ob;   ✓ VALID
```



## VIRTUAL FUNCTIONS:

* We can also assign adress of child class to pointer of base/parent class.

IF IN MAIN:

```
A  *pa
B  ob(3,4);
```

assigning memory { pa = &ob; } on stack

OR ∴ We can also assign adress by using "new". That will be on heap.

{ pa → print(); }

This decision is made on compile time. Not at run time.

∴ Calling 'A' class's print function, bcz compiler only see its own (pointer) 'pa' type. Not object 'ob' type

## (OTHER THAN VIRTUAL)

```
class A {
    :
    :
    :
    void print() {
        cout << "dA= " << dA << endl;
    }
};

class B : public A {
    :
    :
    :
    void print() {
        A :: print();
        cout << "dB = " << dB << endl;
    }
};

int main() {
    A *pa;
    B ob(1, 2);
    pa = &ob;

    pa -> print();
}
```

} base class's (A) print will be called.

**OUTPUT:** dA = 1 } only base class's print will be executed.

**NOTE:**

★ "Once virtual always a virtual."

- If we make methods / member functions virtual one time. then it will remain virtual in overall herarchy of inheritence.

- No need to write virtual again and again with child classes.

- We write "virtual" keyword mostly with parent class, because (we know that) parent's class's member function's to be used / overwrite in child classes.

**SYNTAX:** (MAKING BASE CLASS'S PRINT FUNCTION VIRTUAL.)

```
class A {
    .
    .
    .
    virtual void print() {
        cout << "dA = " << dA << endl;
    }
};
class B : public A {
    .
    .
    .
    void print() {  //Overriden function. It is already virtual here
        A::print();
        cout << "dB = " << dB << endl;
    }
};
int main() {
    A *pa;
    B ob(1,2);
    pa = &ob;
    pa->print();
}
```

here seeing from according to objects type not according to pointer type.

Now, this is not decide at compile time. It will decide at run time.

**OUTPUTS:**

dA = 1
dB = 2

Thus, B class's print function will be executed.

## IF WE DO IN MAIN:

```
int main() {
    A *pa;
    A oa(1);
    B ob(3,4);

    pa = &ob;
    pa -> print();
```

→ print will be of class 'B'. bcz he only seeing objects type, not pointers type.

```
    pa = &oa;
    pa -> print();
```

→ Exactly same call, but behaviour is different.

→ print will be of class 'A' bcz here only seeing objects type, not pointers type.

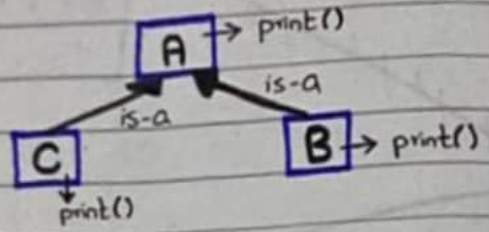
## OUTPUT:

A class { dA = 1

B class { dA = 3
        { dB = 4

★ This concept is known as POLYMORPHISM. (same call multiple functionalities.)

★ IF WE INHERITS / ADDS ONE MORE CLASS. (C)




**CODE:**

```
class A {
    :
    :
    virtual void print() {
        cout << "dA =" << dA << endl;
    }
};
class B : public A {
    :
    void print() {  // Here print is also an virtual.
        A::print();
        cout << "dB = " << dB << endl;
    }
};
class C: public A {
    :
    void print() { // Here print is also an virtual
        A::print();
        cout << "dC = " << dC << endl;
```

**IN MAIN:**

```
int main() {
    A oa(0);
    B ob(1, 2);
    C oc(3, 4);

    A *pa[3] = {&oa, &ob, &oc};   } Array of point type. Its h adresses.

    for(int i=0; i<3; i++)
    { pa[i]->print(); }   } here call is same behaviour is differe

    this is dynamic polymorphism
```

i = 0 : FIRST PRINT → oa class 'A'
i = 1 : 2nd PRINT → oa class 'B'
i = 2 : 3rd PRINT → oa class 'C'

★ Here system is automatically calling different functions of same name

**OUTPUTS :**

class A { dA = 0
class B { dA = 1
          dB = 2
class C { dA = 3
          dC = 4

# ★ RULES OF VIRTUAL FUNCTIONS ★

(1) They cannot be static.

(2) They are accessed by objects pointers.

(3) "Virtual functions" can be a friend of another class.

(4) A "virtual function" in base class might not be used.

(5) If a "virtual function" is defined in a base class, there is no necessity of redefinning it in the derived class.

(When we write virtual function in base class)

(6) For virtual functions, compiler only see objects type. Not the pointer type.

pa → print(); } object type not the pointer type.

→ IF WE DON'T WRITE/DO VIRTUAL PRINT FUNCTION OF CLASS 'A'.

```
In main:  A oa(1);
          B ob(2,3);

  cout << sizeof(oa); // 4
  cout << size of (ob); // 8
```

↳ IF WE DO/WRITE VIRTUAL PRINT FUNCTION OF CLASS 'A'.

```
In main:
    A oa(1);
    B ob(2,3);

    cout << sizeof(oa); // 8
    cout << size of (ob); // 12
```

• Here size object inc

★ Whenever atleast one virtual function exists in a class, then an additi pointer data member automatically a in the size of that class. T pointer is known as "vTable pointer

★ V Table means 'Virtual Table'.

★ V Table consists addresses of its own data member of the class.

★ V Table cannot be inherit in any of the class.

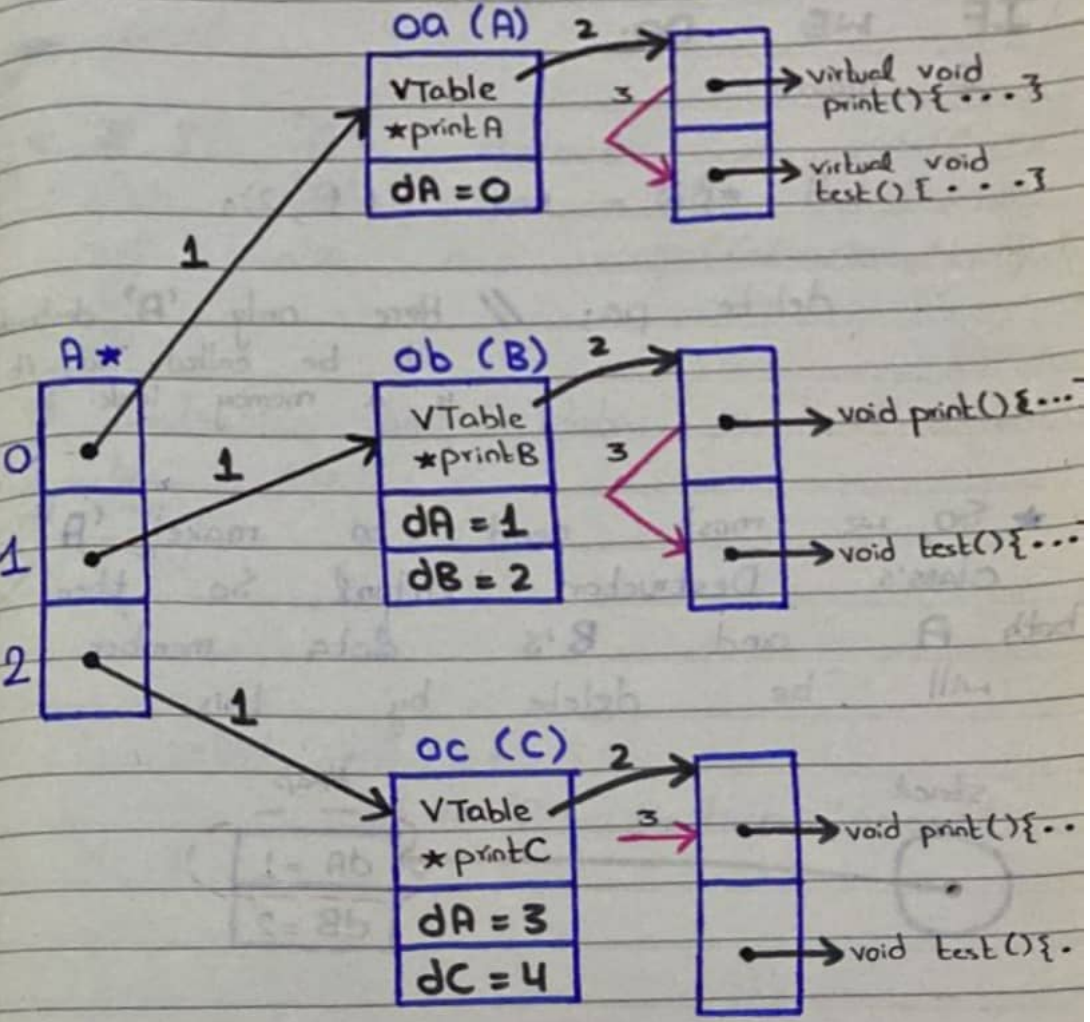
★ Virtual table only consist adresses of virtual member functions of class.

★ Virtual Table (V Table) only consist entities of that specific class in which that V Table pointer data member exists (V Table pointer data member Belongs to)

★ Virtual table does not consist (itself) virtual member functions, it only consist adresses of virtual ~~virtual~~ member functions.

p->[0]->print()

★ Every class has its own V Table.




★ V Table not consist any entities of other member function than virtual.
It only consists virtual member functions adresse

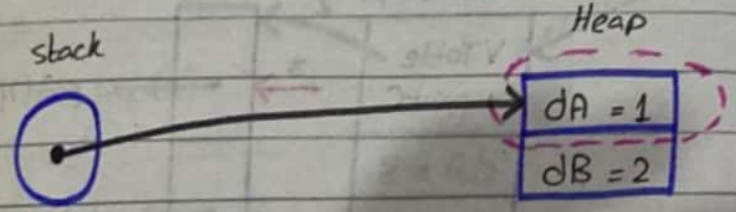
# Virtual Destructor

**IF WE DO:**

In main:

```
A *pa = new B(1, 2);

delete pa; // Here only 'A' destructor will be called. Thus, it is a memory leak.
```

★ So, we must need to make 'A' class's Destructor virtual, So then both A and B's data member will be delete by this.




## Rule of Thumb:

★ Base class destructor must be virtual.

★ If 'A's destructor is virtual then 'B's and 'C's destructor will also be virtual automatically.

★ Constructor and data members cannot be virtual.

```
virtual ~A () {
    cout << "A's Destructor" << endl;
}
```

## EXAMPLE: (VIRTUAL FUNCTIONS)

```
class A {
    int dA;
public:
    A(int dA) {
        this->dA = dA;
        cout << "A's Parametrize Constructor";
    }
//Always make base class destructor virtual to avoid memory leaks.
    virtual ~A() {
        cout << "A's Destructor";
    }

//VIRTUAL DISPLAY FUNCTION
//display() would be overriden in child class(s).
virtual void display() {
        cout << "dA = " << dA;
}

//VIRTUAL TEST FUNCTION
//test() would be overriden in child classes.
virtual void test() {
        cout << "Class A's test function";
}
};
```

```
class B : public A {
    int dB;
public:
    B(int dA, int dB) : A(dA) {
        this->dB = dB;
        cout << "B's Parametrize Constructor";
    }

//DESTRUCTOR
//B's destructor is also virtual
~B() {
        cout << "B's Destructor";
}

//VIRTUAL DISPLAY FUNCTION
//Overriden function. display() is also virtual here.
void display() {
        A::display();
        cout << "dB = " << dB;
}

//VIRTUAL TEST FUNCTION
//Overriden function. test() is also virtual here
void test() {
        cout << "Class B's test function";
}
};
```

```
class C : public A {
    int dC;
public:
    C(int dA, int dC) : A(dA) {
        this->dC = dC;
        cout << "C's parametrize constructor";
    }

//DESTRUCTOR
//C's destructor is also virtual
~C() {
    cout << "C's Destructor";
}

//VIRTUAL DISPLAY FUNCTION
//Overriden function. display() is also virtual here.
void display() {
    A::display();
    cout << "dC=" << dC << endl;
}

//VIRTUAL TEST FUNCTION
//Overriden fuction. test() is also virtual here.
void test() {
    cout << "C's test";
}
};
```

```
int main() {
    A oa(1);
    B ob(2, 3);
    C oc(4, 5);
//Pointer of base class can hold reference of child class.
    A *pa[3] = {&oa, &ob, &oc};

for(int i=0; i<3; i++)
    pa[i]->test();   //Decision will be made on run-time based on object type, not on pointer's type.

A *p = new B(1, 2);

delete p;   // Destructor of class 'B' followed by class 'A' will be executed. Because the base class (A's) destructor was made virtual.
}
```

**OUTPUT:**

```
CLASS A's test function.
Class B's test function.
class C's test function.
```

see files

↓

(i) virtual_functions.cpp

(ii) virtual_functions_basic.cpp

NOTE:

★ Virtual function of base class does not force the child class to implement it.
Because if we don't override the virtual function of base class in child class then child class will simply always use/call base class's virtual function as its own. No need to write that virtual function again and again in child classes.

NOTE:

★ Virtual function of base class does not force the child class to implement it.
Because if we don't override the virtual function of base class in child class then child class will simply always use/call base class's virtual function as its own. No need to write that virtual function again and again in child classes.

★ We make always function virtual when we know that function will have (to) be override in the child classes. (then we necessarily makes it virtual).

★ Here decision will be made on object type, not pointers type.

Lecture #08

# PURE VIRTUAL FUNCTIONS

★ Virtual function according to case

Shape ← general class

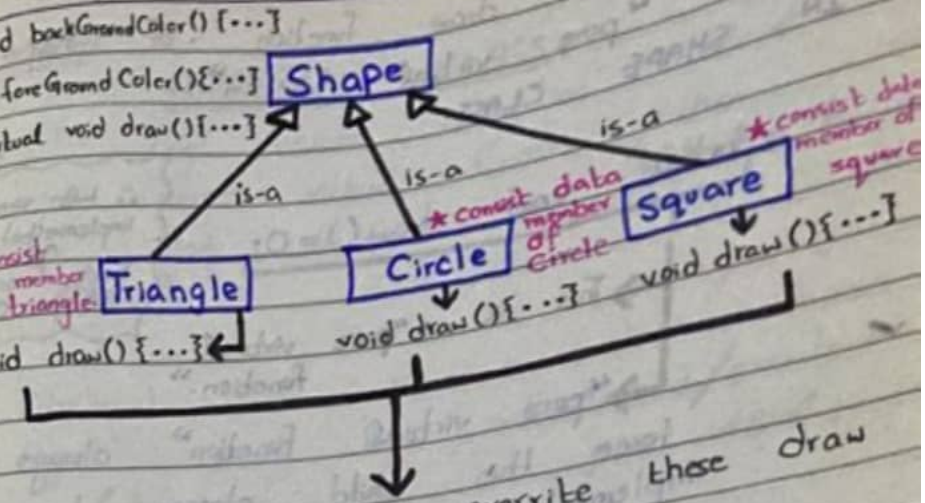
★ We must need '2' member functions for every shape.

```
void backGroundColor(...){...}
void foreGroundColor(...){...}
```

these two will be made in base class.

```
virtual void draw(){...}
```

★ For triangle we need '4' data members.
★ For square we need '8' data members.
★ For circle we need '4' data members.
Total = 16 like this it will keep increasing.




We need to overrite these draw member functions.

## FLAW/ERROR IN THIS (ABOVE) CODE:

★ We cannot see any implementation of 'draw function' in Shape Class. So, there is no need to write any implementation of 'draw function' in Shape Class.

★ Thus, To solve this problem, we eliminate/remove the implementation of draw function from Shape Class. (We make draw function pure virtual)

LIKE IN SHAPE CLASS:

```
virtual void draw() = 0;
```

→ pure virtual function

NOTE:

## ABSTRACT CLASS:

* If any class contains atleast one pure virtual function, then that class becames an "Abstract Class"

* Abstract class is a class whose object cannot be instantiated/created. (bcz it has a pure virtual function.)

* But pointer can be created/declared of Abstract class (bcz pointer only consists memory.)
  - Also reference variables can be declare/created.

## CONCRETE CLASS

* Concrete class is a class whose object can be created.
  (Other than pure virtual function all the classes we created are concrete classes.)

# CODE (PURE VIRTUAL FUNCTION)

```
class A {                    // Abstract Class
    :
    :
    :
    virtual void test() = 0;     // pure virtual function
};
```

Now, 'A' is abstract class, whoose object cannot be created in main.

```
class B : public A {
    :
    :
    :
    void test() {                // *Here test() function is virtual. It is not pure virtual.
        cout << "Test of B";
    }
};
```

No matter if we write "virtual" with the function or not, it still remains a virtual function.

```
class C : public A {
    :
    :
    :
    void test() {                // * Here test() function is also virtual, not pure virtual.
        cout << "Test of C";
    }
};
```

No matter, if we write "virtual" with the function or not, it still remains a virtual function.

main next page

## IN MAIN:

```
int main() {
    A aa;        // ✗ ERROR // Bcz 'A' is an "Abstract Class" whoose object cannot be created As, it has a pure virtual function.

    A *aa;       // ✓ VALID // Bcz pointer data member can be created of "abstract class". As, pointer holds only memory.

    B ob(1, 2);
    C oc(3, 4);

    A *pa[2] = { &ob, &oc };     // Array of pointers which holds memory

    pa[0] -> test();     // test() of class 'B'

    pa[1] -> test();     // test() of class 'C' bcz here only seeing object type, not pointer

}
```

★ IF WE REMOVE OVERRITE.
IF WE REMOVE Void test()
FUNCTION FROM CLASS 'B'.

```
class A {  //Abstract Class. 'A'
    .
    .
    .
    virtual void test() = 0;  //Pure Virtual
                                 test() function.
};
//'B' is Concrete Class.
class B : public A {
    .
    .
    .
    //Here we remove the
    void test() function. from Class
                                 'B'
};
//'C' is Concrete Class.
class C : public C {
    .
    .
    .
    .
```

Here void test() is virtual. Not pure virtual.

```
    void test() {
        cout << "C's test";
    }
```

IN MAIN:

B obj; ✕ ERROR Bcz now 'B' Class accos all properties of parent Class 'A' Thus, it also access pure virtual functions. So, this this pure virtual function will force child class to implement it (pure virtual function) OR Overrite it (pure virtual function) in (itself) Child Class 'B'.

see file

(i) pure_virtual_functions_and_abstract_class.cpp

(ii) pure_virtual_functions_and_abstract_class_basic.cpp

## NOTE: (IMPORTANT POINTS ABOUT "PURE VIRTUAL FUNCTION")

* Pure virtual functions only use in inheritence.

* When a class (BASE CLASS) contains a pure virtual function then the object of that class (BASE CLASS) can be made / created in child class, but cannot be made / created in main. It is because the child class can access all the properties of parent class.

* If a "pure virtual function" is defined in a base class, then there is necessity (it is compulsory) of redefining it in the derived classes, Otherwise we get an error. Bcz the "pure virtual fuction" forces the child classes to implement it necessarily.

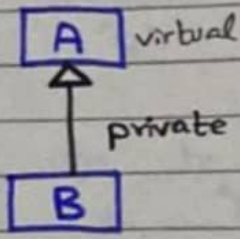


## NOTE:

* Generally inheritence is usually singly

* For private and protected inheritence object of child class cannot be assign to parent class object.

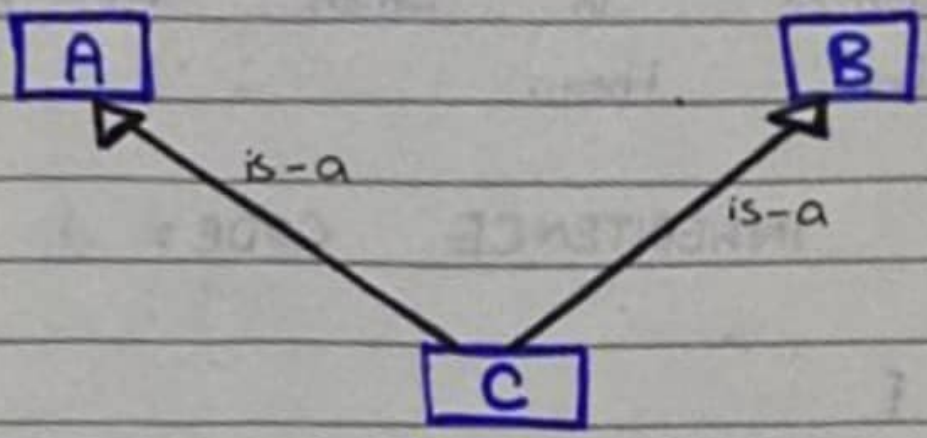
* Private inheritence is basically a composition.




A *pt = &ob; ERROR X bcz inheritence private. So, child object cannot be parent's po

pt → p(); ERROR X bcz inherit is private. So, all member functions are pri here which cannot a in main.

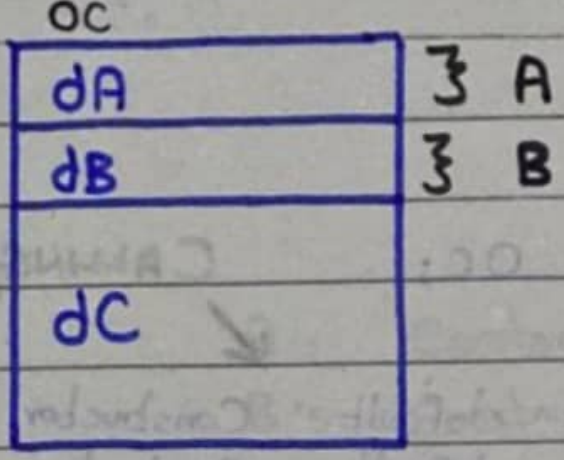
# MULTIPLE INHERITENCE




★ 'C' contains features of both 'A' and 'B'

★ 'C' object can be assign to 'A' as well as 'B' class objects.

C oc;



# NOTE: IMPORTANT CONCEPT

```
class A {
        :
};
class B {
        :
};
class C : public A , public B
{
    int dC;
public:
    //C's CONSTRUCTOR
    C() : B() , A()
    {
        dC = 0;
    }
};
```

Here also the constructor of class 'A' will be called first then the constructor class 'B' will be called bcz compiler will only see the sequence of inheritance/declaration which we have done in class 'C'. It will not see the constructor calling sequence which we do in member initializer list of a constructor but only see the calling seccquence of declaration/inheritance

**IN MAIN:**

```
C oc;
```

A's Constructor
B's Constructor
C's Constructor
C's Destructor
B's Destructor
A's Destructor

**NOTE:**

★ Compiler will only see the sequence of inheritance/declaration.

If the compiler doesn't see the sequence of inheritance and see the sequence of constructor calling only, then the objects of same class will start to be making/run from different constructors (of same class), which will be an error.

## ⇒ TO SEE POLYMORPHIC BEHAVIOUR CODE

```
class A {
    // No, virtual functions.
    'A' class does not contain any
    virtual function.
};
class B {
    // Make 'B's print() function
    virtual.
    virtual void print() {
        cout << dB;
    }
};
```

```
// Class 'C' contains
class C : public A , public B
{
    int dC;
public:
    //PARAMETRIZE CONSTRUCTOR
    C(int dA, int dB, int dC): A(dA), B(dB)
    {
        this → dC = dC;
    }
    C(const C& obj): A(obj), B(obj)
                     'C' child class
    {                object can be assigned
        dC = obj.dC; to both 'A' and 'B' parent class
    }                objects.

    // This print() function is virtual accordance
    void print() { to 'B' parent class and
        A::print(); not virtual accordance
        B::print(); to 'A' parent class
        cout << dC;
    }
};
```

**MAIN() NEXT PAGE**

**IN MAIN:**

```
C oc(1,2,3);
oc.print(); // 'C' print() function will be print here.

A* pa = &oc;
pa -> print(); // 'A' pointers type display function will be print here.

B* pb = &oc;
pb -> print(); // 'C' objects type print() function will be print here.
```

**OUTPUT:**

dA = 1
dB = 2
dC = 3 } 'C' print()

dA = 1 } 'A' print()

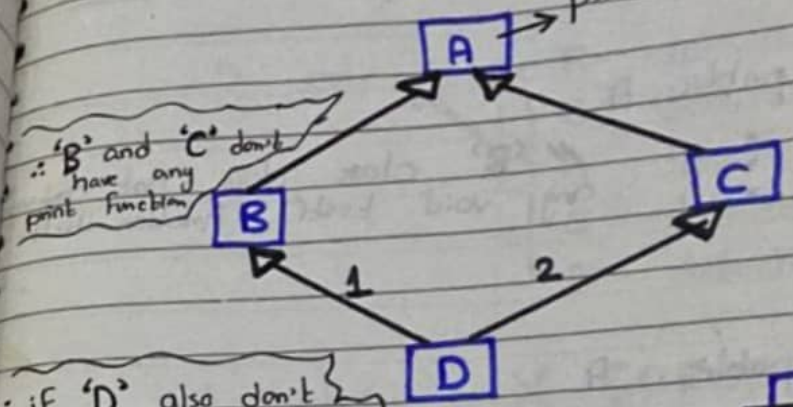
dA = 1
dB = 2
dC = 3 } 'C' print()

see File ↓ multiple_inheritence

# PROBLEM IN MULTIPLE INHERITENCE

**PROBLEM: "DIAMOND PROBLEM"**

* New comers remove / close. multi-inheritence bcz of "DIAMOND PROBLEM"




∴ only 'A' class print contains function

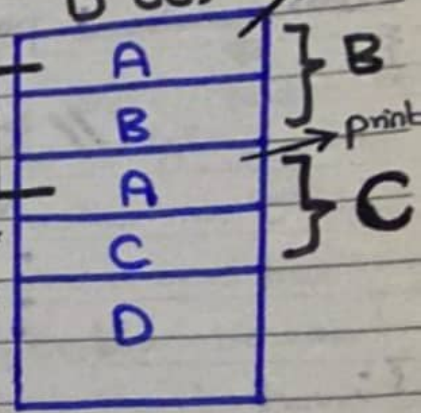
∴ 'B' and 'C' don't have any print function

∴ Looks like diamond

∴ if 'D' also don't have any print() itself

Compiler will become ambiguous bcz it will have two paths to go to print() function of 'A'.
One path by 'B'
One path by 'C'
Because 'B' and 'C' both contains features of 'A'.

D od;
od.print(); ←




One class cannot occur more than once (i.e twice) in a single space. It is a 'PROBLEM'.

## BY CODE (DIAMOND PROBLEM)

```
class A {
        :
        :
    void test() {
        cout << "Test of A";
    }
};

class B : public A
{
        :
        :
    // 'B' class does not contain any void test() function itself.
};

class C : public A
{
    // 'C' class does not contain any void test() function itself.
        :
        :
};

class D: public B, public C
{
    // 'D' class also does not contain any void test() function itself.
        :
        :
};
```

IN MAIN:

```
D od;
```

- A default Constructor
- B default Constructor
- A default Constructor
- C default Constructor
- D default Constructor

```
od.test();
```

X ERROR → bcz test() function exists in 'A' not in 'B' neither in 'C'. So, it gives error bcz D::test() became ambiguous As, it have two paths to go to test of 'A' → By 'B' or By 'C'

This is a "DIAMOND PROBLEM"

See File

diamond_problem_multiple_inheritence.cpp

# VIRTUAL INHERITENCE

★ Virtual Inheritance is the solution of "DIAMOND PROBLEM"

★ We will virtually inherit 'B' and 'C' classes from class 'A'-

↳ **CODE:**

```
class A {
        :  // Only class 'A' will contain test() funct
};
class B : virtual public A }→ Here 'A' class is virtually inherit in class 'B'
{
        :
};
class C : virtual public A }→ Here 'A' class is virtually inherit in class 'C'
{
        :
};
class D : public B, public C
{
}
```

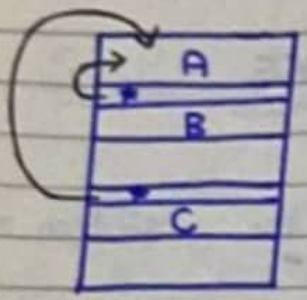
Here 'B' and 'C' classes will automatically be "virtually" inherited. There is no need to write "virtual" here.

**IN MAIN:**

```
D od;
od.test();
```

✓ VALID

∴ Now, D's object will access 'A' class's member function only by one path.

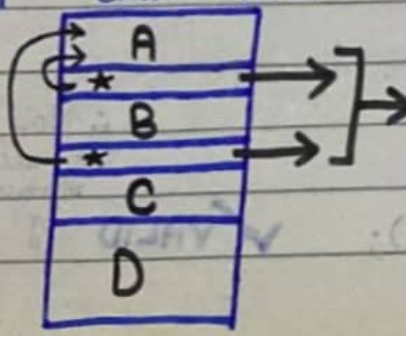



★ Now, 'A' object/class is shared b/w 'B' and 'C' classes.

★ Now, 'D' object will go to pointer of 'B' or 'C' to access 'A'

"DIAMOND PROBLEM" → Solution: "VIRTUAL INHERITENCE"

★ Now, one one additional pointer each will exist in the space of 'B' and 'C' classes. This additional pointer will points towards class 'A'. Bcz, now class 'A' will be shared b/w classes 'B' and 'C'. And 'D' object will access the functions of class 'A' through any of these pointers means there will be only one path to access 'A' class functions.




These pointers are of type 'A'. bcz these are pointing towards 'A'.

NOTE:

★ IF WE REMOVE "VIRTUAL INHERITENCE"
→ ALSO REMOVE ALL THE VIRTUAL FUNCTIONS

IN MAIN:

```
D od;
cout << sizeof(od);   // 20
```

Ambiguous Call

A = 4
B = 4
A = 4
C = 4
D = 4
20

★ IF WE MAKE "VIRTUAL INHERITENCE".
→ MEANS REMOVE "AMBIGUOUS CALL".
→ BUT ALSO REMOVE ALL VIRTUAL FUNCTIONS.

```
D od;
cout << sizeof(od);   // 24
```

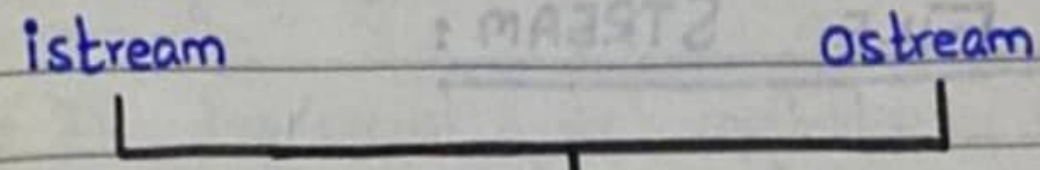
A = 4
B = 4 + 4
C = 4 + 4
D = 4

→ additional pointers

} bcz now an additional pointer adds thats why its size increase

# FILE I/O STREAM

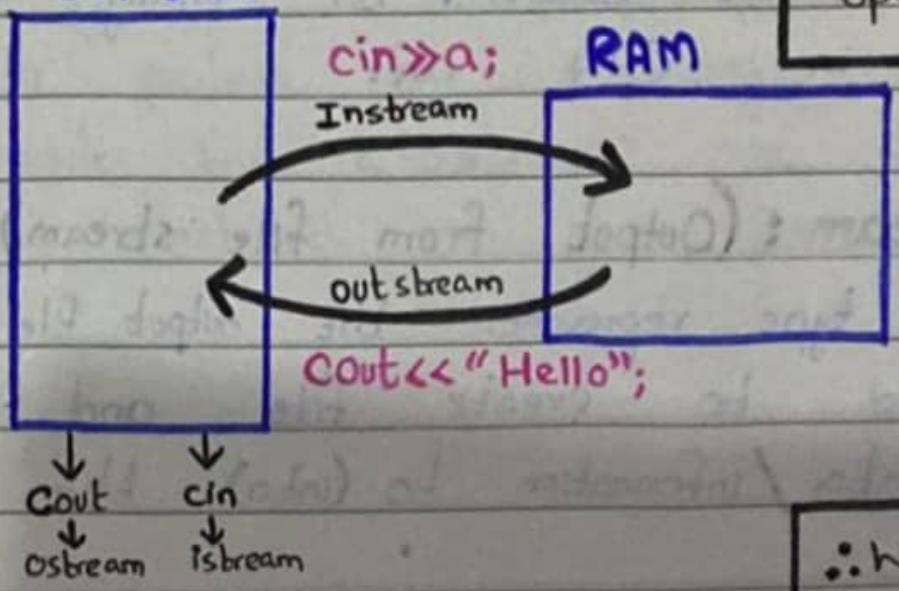
★ So, far we have been using the "iostream" standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.




```
int a=5;
cin >> a;
```

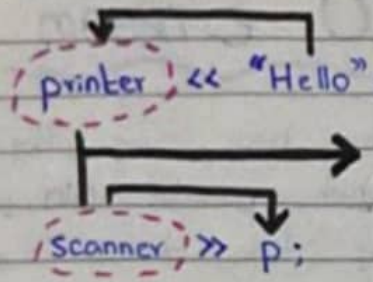
```
cout << a;
```

∴ stream insertion and extraction (<<, >>) operators are independent.




∴ We cannot create object of "ostream" and "istream". We only use their references

IF EXAMPLE:




# IN FILE STREAM:

★ For to read and write to a file, we necessarily requires another standard C++ library: **#include "fstream"**

★ The '3' useful classes (data types) for working with the files in C++ are:

**(1) ofstream:** (Output from file stream)
→ This data type represents the output file stream
→ it is used to create file and to write data / information to (into) the files.

**(2) ifstream:** (Input from file stream)
→ This data type represents the input file stream
→ It is used to read information from files.

**(3) fstream:** (Output and Input from file stream)
→ This data type represents the file stream generally.
→ It has/consist the capabilities of both 'ofstream' and 'ifstream' which means:
- It can create files.
- Write information / data to files.
- And, Read information from Files.

★ To perform file processing in C++, header files **<iostream>** and **<fstream>** must be included in your C++ source file.

# OPENING A FILE

* In order to work with the files in C++ (to read or to write), you will must have to open it.
Either **'ofstream'** or **'fstream'** object may be used to open a file for writing. And **'ifstream'** object is used to open a file for reading purpose only.

There are two(2) ways to open a file:

## (1) USING THE CONSTRUCTOR:

Syntax:

* **Without path:**

```
ofstream outf("test.txt");
```

object name which can be any

→ If we don't specify path. It will automatically create file at the default path where we are working. If the file not already exists

**OR**

```
fstream outf("test.txt");
```

* **With Path:**

```
ofstream outf("C:\\test.txt");
```

→ It will create file in 'C' folder and open it. (If file already don't exists).

**NOTE:**

```
ofstream outf ("z:\\test.txt");
if (!outf) {
        cout << "ERROR!";
        exit(0);
}
```

**ERROR! ✗**

∴ bcz folder 'z' does not exists. So, it will not capable to create any file.

## (2) USING MEMBER FUNCTION open() OF THE CLASS:

* Following is the standard syntax for **'open()'** function, which is a member of **fstream**, **ifstream** and **ofstream** objects

```
void open(const char *filename, ios::openmode mode);
```

Here the first argument specifies the name and location of the file to be opened and the second argument of the **open** member function defines the mode in which the file should opened.

IN MAIN:

* SO, IN MAIN TO OPEN A FILE BY open() MEMBER FUNCTION:

Syntax:

```
ofstream outf;
outf.open("test.txt");
```
} Open in form of write

OR

```
fstream outf;
outf.open("test.txt");
```
} Open in form of read as well as write.

OR

```
ifstream inf;
inf.open("test.txt");
```
} Open in form of read.

# CLOSING A FILE:

* When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the files, But it is a good practice that a programmer should close all the opened files by himself.

Syntax: (of closing a file):

```
outf.close();
```
} This function is use to close the file bcz operation is done b/w two memories.

File name which can be any. ←

→ member function of fstream, ifstream and ofstream objects.

# File Modes

## ios::out

→ Open a File in output mode.

→ Open a file for writing.

```
ofstream outf("test.txt", ios::out);
```

By default ofstream behaviour is "ios::out".

**OR**

```
ofstream outf;
outf.open("test.txt", ios::out);
```

→ It will overwrite the data.

→ **NOTE:** "ofstream" can never be instream / ifstream.

(ii) ios:: in

→ Open a file for reading
→ Read data from file
(Input data from file)

ifstream inf("test.txt", ios::in);
By default "ifstream" behaves is "ios:: in"

→ **NOTE:** "ifstream" can never be ofstream/ outstream.

(iii) ios:: app

→ It will place the pointer to end means where we stop our working of writing in file.

→ If the file is opened in append mode, You can reposition the file pointer but you can only write data at the (current) end of the file.

→ In append mode, Even if we reposition (move) the file pointer (means we change the position of file pointer), we cannot write there. We can only write at the end.

→ Connection/Link → ofstream

ofstream outf("test.txt", ios::app);

(iv) ios:: binary

→ Data will be write in form of binary.

→ Connection/Link → ofstream

(v)

## ios:: ate

→ Open a file for output and move the read/write control to the end of the file.

→ When a file is opened in ate mode, initially the file pointer is at the end of file.

→ We can reposition (move) the file pointer before the end of file (means change position of file pointer according to our own choice).

→ And, then if we want to write data into the file, then it will write only at position where the file pointer is currently existing.

→ Link/Connection → ofstream.

(vi)

## ios:: trunc

→ If a file already exists and we open that already existing file for writing by Trun mode, then it will delete/overwrite (replace) the contents of file.

→ (Means the data already written in the file will be removed/replaced by the new data that we enter when we open a file in Trunc mode)

**NOTE:** The file pointer is at the begioning of the file in this case.

→ Connection/link → ofstream.

~~WRITING TO A F~~

## ★CHECKING IF THE FILE IS CREATED BY CONDITION

```
ofstream outf("test.txt", ios::at);
if (!outf) {
    cout << "ERROR";
    exit(0);
}
```

We use always this condition in our program to check whether the file is created or not.

## ★USING getline AND eof:

```
string st;
ifstream inf("test.txt", ios::in);

while (inf.eof() == 0)
    getline(inf, st);
    cout << st << endl;
}
```

★ **getline ( , ):** We use when we want to read any number of lines with spaces from a file.

★ **eof ():**
eof() is a function in file iosbeam which remains '0' until the time there exists some data content in the file. When your file ends (means content in the file is discard / removed), then eof() contains some character in it whose value is other than '0'.
And, whenever content remains in your file, the value of eof() always remains '0' on every line.

**CODE:**

```
int main () {

    ofstream outf;
    outf.open ("test.txt");
    outf << "Hello World!";
    outf.close ();

    reburn 0;
}
```

See file

↓

file_stream_basic-01.cpp

## CODE: (WRITING TO A FILE)

```
int main() {
    int id;
    string str;
    //Open file using Constructor and
                    writing to it.
    ofstream outf("test.txt", ios::out);

    if(!outf) {
        cout << "ERROR!";
        exit(0);
    }

    while(cin >> id >> str){
        outf << id << " " << str << endl;
    }
```

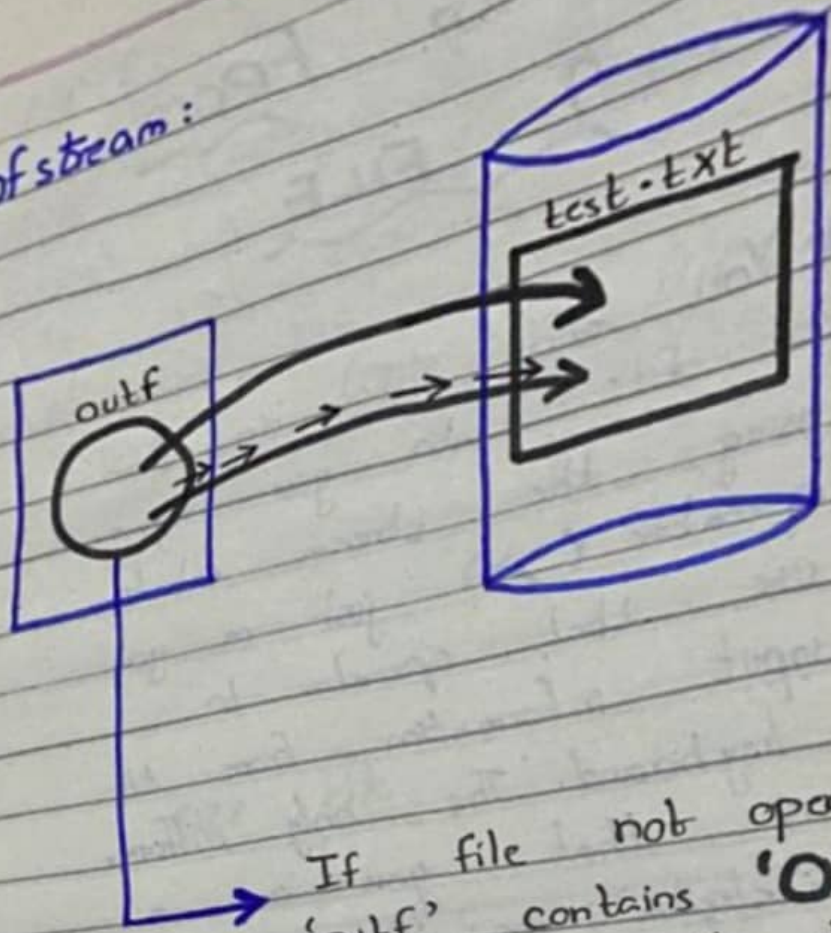
Press To end stream
Ctrl+Z → ENTER

```
    outf.close();

    return 0;
```

see file ↓

file_stream_out_write-02.cpp

ofstream:




If file not open then 'outf' contains '0'.
If file created then it has value other than zero (0).

∴ We use '!outf' bcz when file is not open: outf = 0, then !0 = 1 and condition TRUE. if condition runs / true at '1'. Thus, it print ERROR!.

# READING FROM A FILE

★ You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an 'ifstream' or 'fstream' object instead of the "cin" object.

NOTE:-

★ While reading a file format is very important. We must know format of a file before reading from a file:)

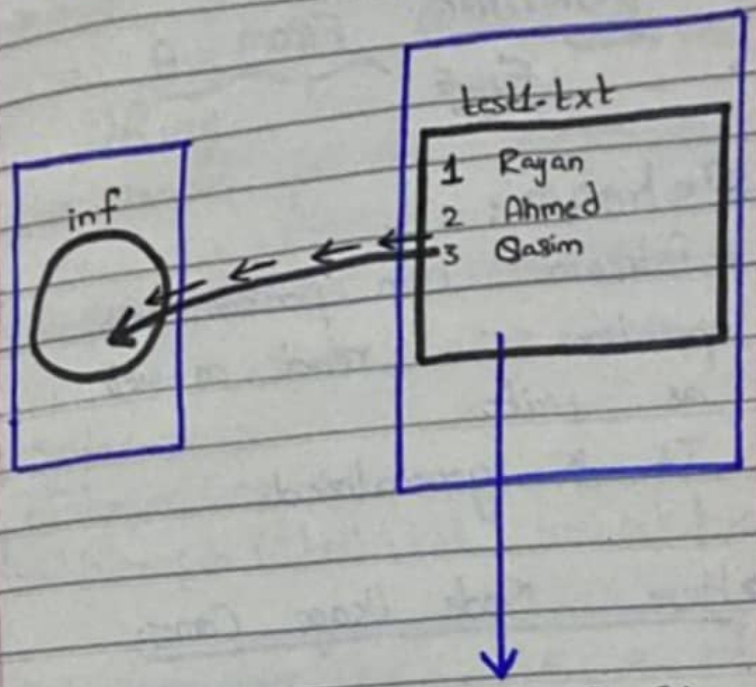
see file ↓

File_stream_in_read-03.cpp

## CODE: (READING FROM A FILE)

```
int main() {
    int id;
    string str;
// Opening file using member open() function
                and READING from it.
    ifstream inf1;
    inf1.open("test1.txt", ios::in);
    getline(inf1, str);
    cout << str;

    inf1.close();

// Opening file using Constructor
                and READING from it.
ifstream inf2("test.txt", ios::in);

while(inf2>> id >> str)
    cout << id << " " << str << endl;

inf2.close();
}
```

ifstream:




Format of file is very important.
(We must know the format of file before reading from a file).

Thus, we use

```
inf >> id >> str;
```

# READING AS-WELL-AS WRITING FROM A FILE

## fstream:

* fstream can perform both operations i.e read as well as write.
It is generalized.

⇒ Here Mode Usage Comes:

* By using fstream we can open file for read as well as write at a same time.
* if we want to open both read and write mode:

**SYNTAX:**

```
fstream iof("test.txt", ios::out | ios::in);
```

**NOTE:**

↳ We can combine two or more of the modes by 'ORing them together.
For example: If we want to open a file in write mode and want to truncate it in case that already exsts:

Syntax:-

```
ofstream outfile;
outfile.open("test.txt", ios::out | ios::trunc);
```

↳ Similarly, we can open a file for reading and writing purpose as follow:

Syntax:-

```
fstream inf;
inf.open("test.txt", ios::out | ios::in);
```

## CODE: (READING AND WRITING FROM A FILE)

```
int main() {

    int id;
    string sbr;

    fstream oif("test3.txt", ios::out | ios::in);

    if(!oif) {
        cout << "ERROR!";
        exit(0);
    }

    while(cin >> id >> sbr)
        oif << id << " " << sbr << endl;

    oif.close();

    oif.open("test3.txt");

    while(oif >> id >> sbr) {
        cout << id << " " << sbr << endl;
    }

    oif.close();

    return 0;
}
```

It will take data from user.

Data will be written in file.

Ctrl+Z → ENTER → will end stream.

We can also use getline(,) here.

It will read File. It will ignore

(It will end automatically when compiler gets eof → END OF FILE

No, need to press Ctrl+Z bcz

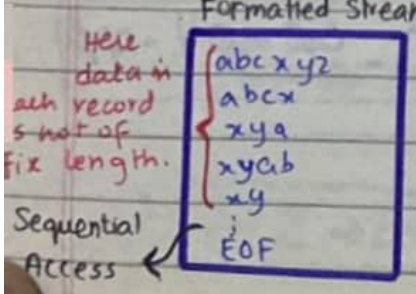
see File

file_stream_in_out-04.cpp

# FORMATTED AND UNFORMATED STREAMS:

## ⇒ FORMATTED STREAMS:

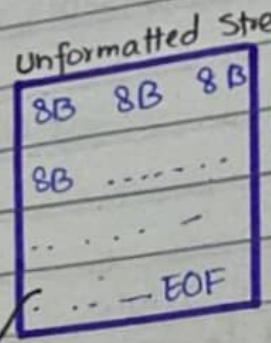
* There must exist some format in text (data).
* Formatted streams also known as **Text Streams.**
* Formatted streams are slow (means the data records in these streams do not have fixed length (variable length)). Variable length makes data search sequential (means the access of data is sequential)




**NOTE:** Stream insertion and Extraction operator only work for formatted files

## ⇒ UNFORMATED STREAMS (FILES):

* In these, you work with binary numbers (means you are working with data in bytes). It is known as **Bytes oriented Data.**
* These files gives us random (direct) access to data records. (bc2 record length fixed).
* These files are also known as Binary Files (streams).




**NOTE:** If each record of size = 8B and we want to go to record # 3 then 3 x 8 = 24 Bytes.

• Not necessary to read one (1) record full, we can read bits of our choice i-e 3, 4, etc.

* To write data in Binary files, we move our scope towards "char" because characters are Bytes Oriented Operators.
* For this purpose, we typecast by using syntax:

**reinterpret_cast<char *>(&a)**

↓

* It will convert the datatype of array to "char", by type casting.

**CODE:**

```
#include "iostream"
#include "fstream"
using namespace std;
int main(){
ofstream outf("test.txt", ios::binary);
if (!outf){ exit(0);}
int a = 5;
outf.write(reinterpret_cast<char*>(&a), sizeof(int));
outf.close();
return 0;
}
```

it converts int type data into "char"

It gives no. of bytes (size) of the data we want to write.

Here we want to write data in binary format (bytes format). So, we move to character type here and we pass reference. Thus, we use conversion by TYPECasting.

//If we want to write binary data on code.

```
cout.write(      );
```
✓VALID

## STORING CLASS OBJECT IN A FILE:

```
class A {
public:
  int a;
  float b;
A() {                    //default constructor
  a = b = 0;
}
A(int a1, float b1) : a(a1), b(b1)
{
      //parametrized constructor
}
void print(){
   cout << a << b;
}     //print function
```

```
int main(){
  A o(1, 1.1f);          //To write
outf.write(reinterpret_cast<char*>(&o),
          sizeof(A));
```

class size OR we can also write "sizeof(o);" here.

**NOTE:** class size & object size is same

//TO READ BINARY DATA FROM FILE:

* ifstream is used.

```
inf("test.txt", ios::binary);
if(!inf){
      cout << "ERROR!" << endl;
      exit(0);
    }
inf.read(reinterpret_cast<char*>(&x), 4);
```

make size of read 4 bytes.

//TO WRITE ARRAY OF OBJECT

```
A o[3] = {A(1, 1.1f), A(2, 2.2f),
          A(3, 3.3f)};
```

→ READING ... OF AN ARRAY AND TILL THE END OF FILE:

```
int main (){
   A r;
inf. read (reinterpret_cast<char*>(&r),
            sizeof (A));
while (!inf.eof ())
{
    r.print ();
  inf.read (reinterpret_cast<char*>(&r),
            sizeof (A));
}
inf.close ();
return 0;
}
```

this is a Bool type function which is true when end of file occurs.

★ This code will read the objects of an array one by one till the End of FILE (EOF).

## SEARCH IN FILE:

★ To search in file, we have two functions:

(1) seekg → This function is used for Reading file. Seekg means "to get" from if stream.

(2) seekp → This function is used for writing file (or overwriting). Seekp means "to put" in ofstream.

```
//using seekg.
A r;
inf.seekg(4);
```

move from current position to 4 bytes.

// For telling the position to read from, we use following:

⇒ ios:: beg

This read bytes from start

⇒ ios:: cur

This read bytes from current state

⇒ ios:: end

If eof not occur then it will read bytes in reverse process from end.

//example; using seekg
ios:: beg

inf.seekg(4, ios::beg);

**SEE CODE UPLOADED BY SIR UMAIR FOR MORE EXAMPLES**

★ TO DELETE / REMOVE A FILE:

★ We include a library:
#include "cstdio"

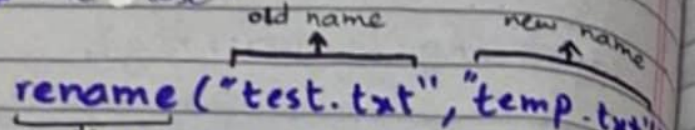
★ In main():

- To successfully remove, return '0'
- Unsuccessfully remove, return other than '0', using:

remove("test.txt");

## ★ TO RENAME FILE NAME:

★ We use:




It can also remove files.

★ rename function returns '0' when name changes