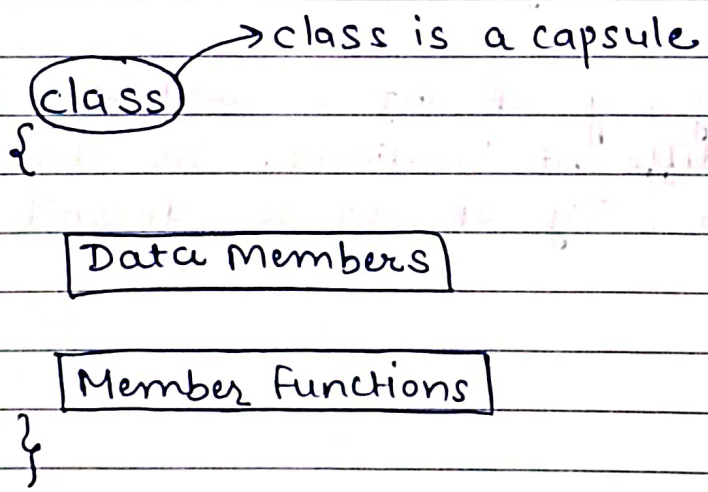
04/11/2023
Saturday

* **Four pillar of OOPS –**

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

1. Encapsulation – Data Hiding

Wrapping up of data members and member functions in a single entity is known as encapsulation.



Perfect Encapsulation – Marking all the data members private.

2. **Inheritance –**

Child class inherit properties of parent class.

Syntax →

```
class parent { };

class child : inheritance mode parent { };
```

= **Inheritance mode can be** – public
private
protected

* **Access Modifier – Protected**

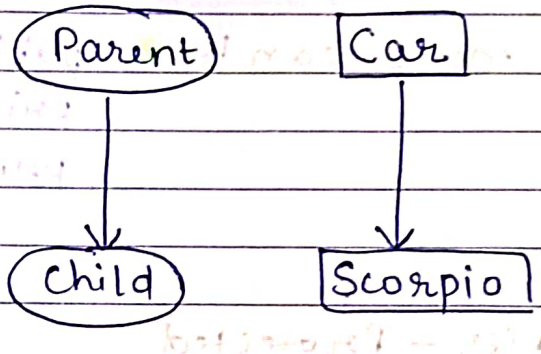
It is same as private. The difference is that we can access an entity which is marked protected inside child class which is not possible in case of private.
Simply, protected inherit ho jata hai, private member ko inherit nahi kar sakte.

Base class ka Access modifier	Mode of inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible	Not accessible	Not accessible

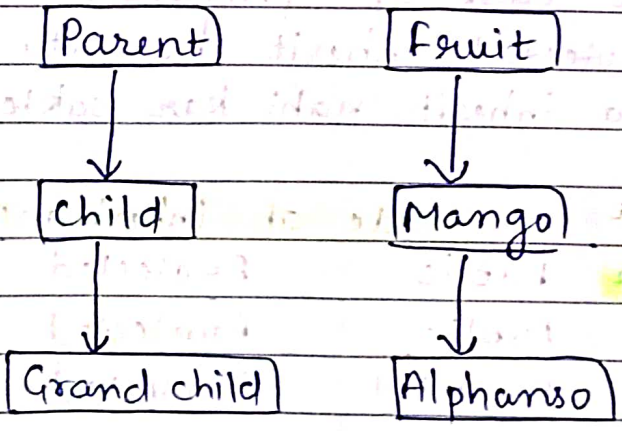
* Types of Inheritance -

1. Single
2. Multi - level
3. Multiple
4. Hierarchical
5. Hybrid

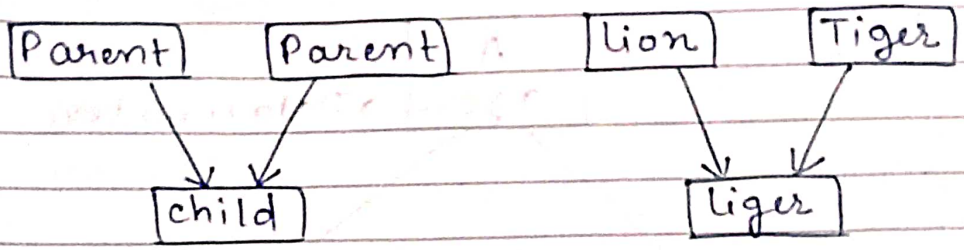
1. Single Inheritance -



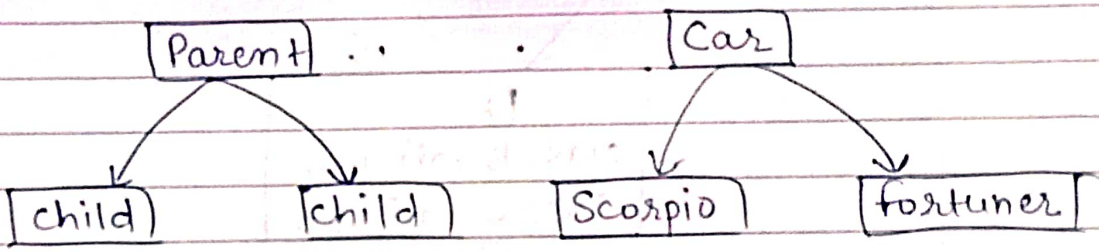
2. Multi-level Inheritance -



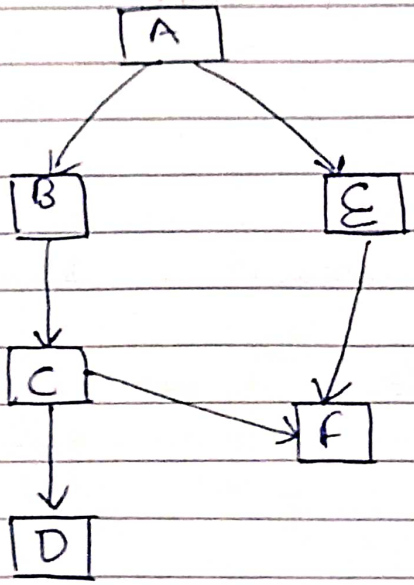
3. **Multiple Inheritance –**



4. **Hierarchical Inheritance –**

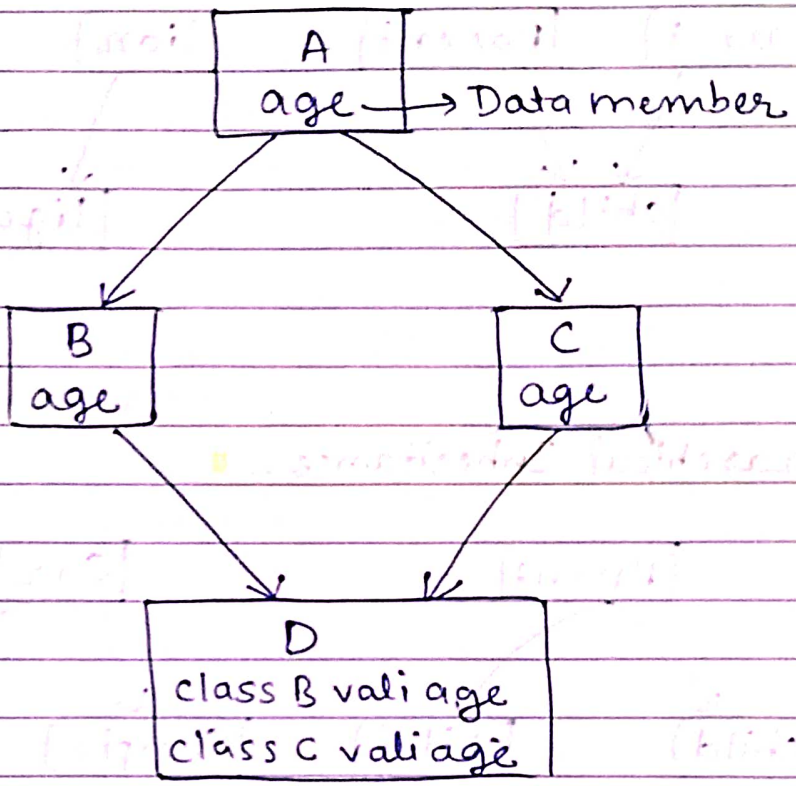


5. **Hybrid Inheritance –** Combination of two or more inheritances.



* Inheritance ambiguity or diamond problem

let,



That's why, compiler confuse ho jata hai kis class ke data member ko access krna hai or error throw krta hai.
To remove this ambiguity, use scope resolution (::) operator.

Code →

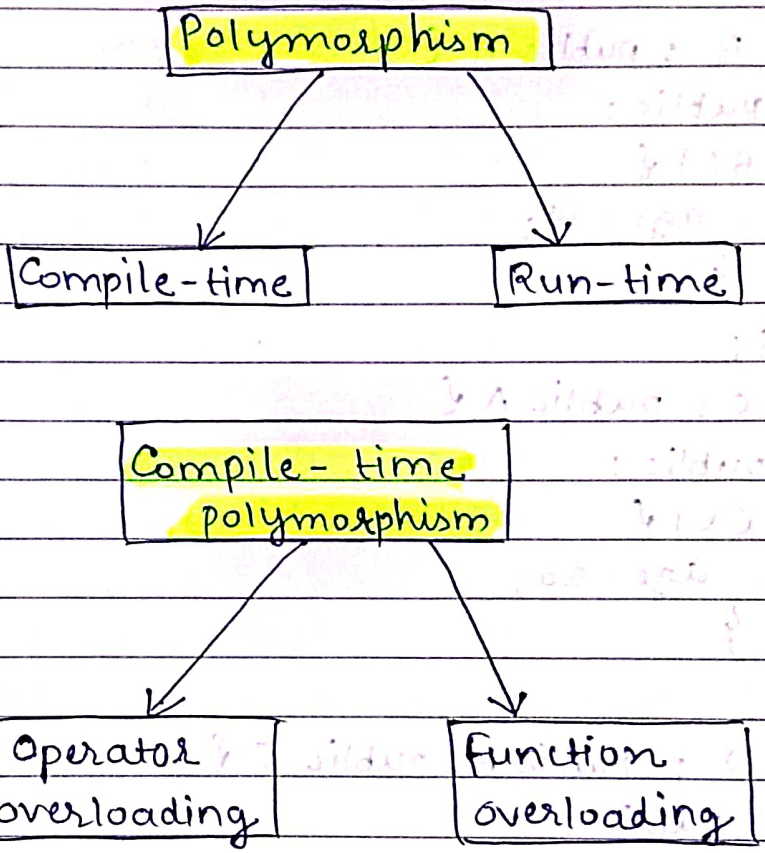
```
#include <bits/stdc++.h>
using namespace std;

Class A {
    public:
    int age;
};
class B : public A {
    public:
    B() {
      age = 19;
    }
};
class C : public A {
    public:
    C() {
      age = 20;
    }
};
class D : public B, public C {
    public:
    int weight;
    D() {
      weight = 50;
    }
};
int main() {
    D obj;
  cout << obj.B::age << " " << obj.C::age << " "
    obj.weight << endl;
}
```

O/P → 19 20 50

* **Polymorphism –**

Poly means many. Morph means forms.
So, polymorphism means existing in many forms.



Function overloading –

Functions having same name but different signature i.e. number of parameter or type of parameter. Return type of functions must be same. By doing this, we can achieve function overloading.

Code →

```
#include <bits/stdc++.h>
using namespace std;

class abc {
    public:
    int sum(int a, int b){
        return a+b;
    }
    int sum(int a, int b, int c){
        return a+b+c;
    }
    int sum(int a, double b){
        return a+b;
    }
    int sum(int a, float b){
        return a+b;
    }
};

int main(){
    abc a;
    cout << a.sum(2,3) << endl;
    cout << a.sum(2,3,5) << endl;
    cout << a.sum(2,3.12) << endl;
    cout << a.sum(2,3.2f) << endl;
}
```

O/P →

```
5
10
5
5
```

* **Operator overloading –**

Kisi operator se uske kaam ke alawa or dusra kaam karvana is operator overloading.

let, operator + can do sum and concatenation. If i want that using + operator, i can subtract two numbers, then this can be achieved by operator overloading.

Simply, it is assigning an extra task to an operator other than its own task.

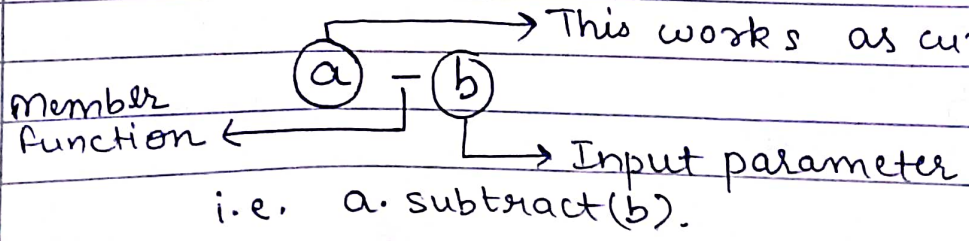
Syntax →

```
return type  operator  operator_symbol ( )
    {
        }
```

For instance,

```
int operator + ( )
    {
        }
```

= Let overload krna hai (-) operator ko so that subtract krne par divide ho. let the object a and b.



i.e. a.subtract(b).

During binary operator overloading →

- First object works as current object.
- operator is member function or function call.
- Second object works as input parameter.

Code →

```
#include <bits/stdc++.h>
using namespace std;

class abc {
    public:
    int val;

    void operator - (abc &obj2) {
        int obj1VALUE = this->val;
        int obj2VALUE = obj2.val;
      cout << (obj1VALUE / obj2VALUE) << endl;
    }
};

int main() {
      abc obj1, obj2;
      obj1.val = 10;
      obj2.val = 5;
// dono object ko (-) krne pr divide honge due to
// operator overloading of (-) operator.

      obj1 - obj2;
    }
```

o/p → 2

02/11/2023
Thursday

* Const keyword →

- The const keyword is used to declare that a variable, function or object is immutable i.e. its value cannot be changed after initialisation.

Eg- `const int x=5;`
`x=10;` → Any attempt to modify the value of x will give compilation error.

- Compiler stores const variables in read only memory which results in faster access time.

= Const with pointers

1. const data, non-const pointer

```
int main() {

 const int* a = new int(2);
// int const* a = new int(2); // same as line above
// *a = 5; can't change the content of pointer
 cout << *a << endl; // 2
  int b = 10;
  a = &b; // pointer itself can be re-assigned
 cout << *a << endl; // 10
}
```

O/P → 2
10

__/__/____

Note- If i write const before *(asterisk), then content of pointer remains constant.
If i write const after *(asterisk), then pointer remains constant.

2. Const pointer, non-const data

```
int main(){

    int* const a = new int(2);
    cout<<*a<<endl; // 2
    *a = 5; // can change content of pointer
    cout<<*a<<endl; // 5
     int b=10;
    // a= &b; // pointer can't be re-assigned.
}
```

3. Const data, const pointer

≤ If i write const before and after *(asterisk), then content of pointer and pointer, both remains constant.

```
int main(){

    const int* const a = new int(2);
    // *a = 5; can't change content of pointer
     int b=10;
    // a=&b; can't change or re-assign pointer
}
```

Note – If class mai member function ko constant bna rkha hai to us member function mai data member ko reassign nhi kr skte.

* **const with functions →**

```
#include <bits/stdc++.h>
using namespace std;

class abc {
    int x;
    int *y;
    int z;

    // constructor
    public:
    abc(int x, int _y, int z=10) {
        this->x = x;
        this->y = new int(_y);
        this->z = z;

    int getX() const
    {
        //x = 10; → this line will give error as function is constant
        return x;
    }

    void setX(int val1){
        this->x = val1;
    }

    int getY() const
    {
        return *y;
    }
```

→ default arguments Always write them at the end in argument list

→ we can't make setter constant as we assign values in setters

Note → Agr kisi function mai object constant le liya to us function mai koi bhi function call esi nhi honi caahiye ki vo non-const ho.

```
int getz() const{
    return z;
  }
};
```

```
void print (const abc &a){
  cout<<a.getX()<<" "<<a.getY()<<" "<<a.getZ<<endl;
}

int main(){
    abc a(2,5);
    print(a);
}
```

O/P → 2 5 10

Braking the constantness using mutable →

If koi member function const hai but fir bhi hum kisi data member ko us member function mai re-assign krna caahte hai to data member ko **mutable** bna do.

```
class abc{
    mutable int x;

    public:
    abc (int_x){
      this->x = _x;
    }
    int getx() const{
        x=10; // → This line will not give error now.
        return x;
    } };
```

Note → Mutable is used only for debugging purposes.

```
int main(){
    abc a(2);
    cout<< a.getX()<<endl;
}
```

* **Initialization list →**

New style of writing ctor i.e. constructor.

Syntax –

```
class_name (datatype val1, datatype val2 ....):

datamember(val1), datamember(val2)..... { }
```

Code →

```
class abc {
    const int x;
    int *y;

    //initialization list
    public:
abc( int _x, int _y): x(_x), y(new int(_y)) {
    cout<<"inside initialization list"<< endl;
}

int getX() const {
    return x;
}
int getY() const {
    return *y;
}
};
```

Note - If koi data member const hai to ctor mai use initialize nhi kar skte while initialization list mai ye possible hai.

```
void print (const abc &a) {
    cout<< a.getX() <<" "<< a.getY() << endl;
}

int main( ) {
    abc a(2, 3);
    print(a);
}
```

o/p → inside initialization list

* **Macros →**

Macros are preprocessor directives that allows us to define constants, functions or code snippets that can be used throughout code. They are defined using #define directive and are evaluated by the preprocessor before code is compiled.

It can be used for defining constants or creating shorthand for commonly used expressions.

Macros increases the readability of the code.

Code →

```
#include<bits/stdc++.h>
using namespace std;

#define MAX(x,y) (x>y?x:y)

int findMax(){
    int x=20;
    int y=10;
    return MAX(x,y);
}

int main(){
    cout<<findMax();
}
```

O/P → 20

* lvalues and rvalues →

lvalues → variable having memory location.

for eg - intx;
char ch;

rvalues → variable don't have memory location.

for eg → • int x = 5; ← This is rvalue.

• int b=10;
int &a = b;
→ Reference variable doesn't have memory location. They are rvalues.

* **Static keyword in class →**

• **Static data members →**

That variable is going to share memory with all of the class instances i.e. objects.

Code →

```
class abc{
    public:
    static int x,y;

    void print(){
      cout<<x<<" "<<y<<endl;
    }
};

int abc::x;
int abc::y;

int main(){
    abc obj1;
    obj1.x=1;
    obj2.y=2;
    obj1.print(); // 1 2
    abc obj2;
    obj2.x=3;
    obj2.y=4;
    obj1.print(); // 3 4
    obj2.print(); // 3 4
}
```

O/P →

```
1 2
3 4
3 4
```

There is nothing like obj1 ka x and y, obj2 ka x and y. Every instance of class abc have same x and y. Whatever change we made in x and y, it will be visible in every instance of class abc as we have made data members static.

* Static member function mai non-static data members ko access nahi kar skte.

Static member functions →

There is no instance of that class being passed into that function. Static member functions shares the single copy of member function to any number of class instances.

Code →

```
class abc {
    public:
    static int x, y;

    static void print(){
      cout << "inside static function" << endl;
      cout << " x << " " << y << endl;
    }
};

int abc::x;
int abc::y;

int main(){
    abc::x = 2;
    abc::y = 4;
    abc obj1;
    abc::print();
    abc obj2;
    abc::print();
}
```

Agr kisi member function ko static banate hai inside class to saare instances ke liye member function same rhega. Kisi bhi object ka this pointer ni banega.

O/P →

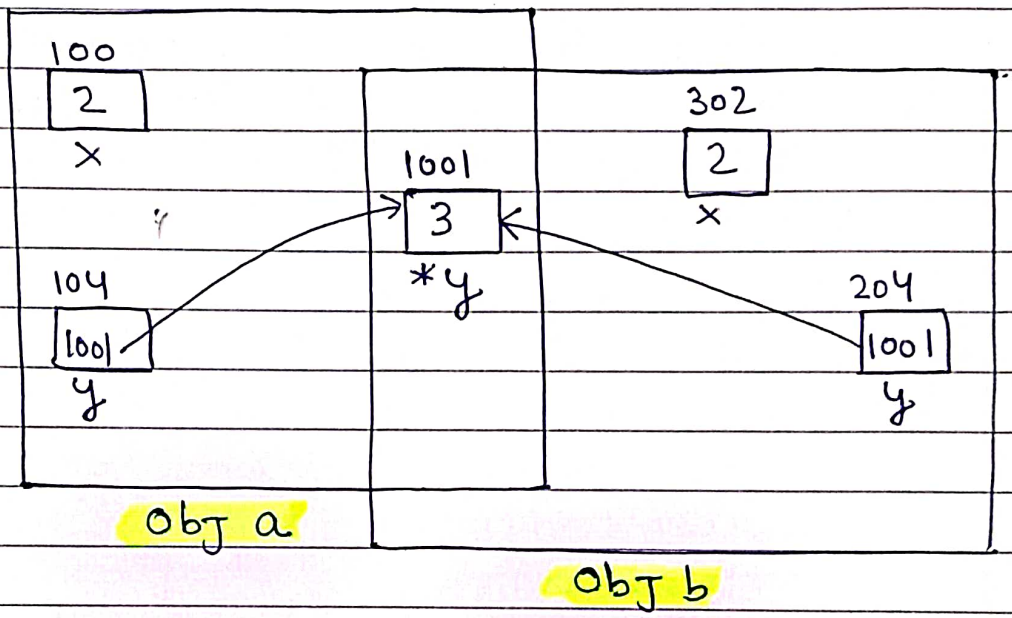
```
Inside static function
2 4
Inside static function
2 4
```

__/__/____

* **Shallow copy vs Deep Copy →**

Shallow copy →

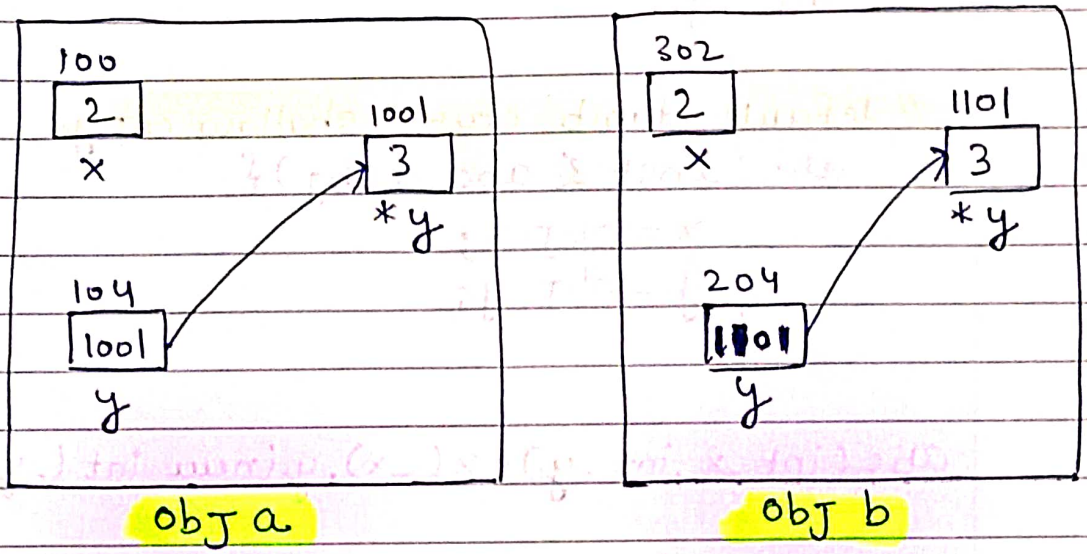
let two data members- int x;
int *y;



⇒ Object a create kiya hai and object b ko copy kiya hai object a se using copy constructor. Object a ka y & object b ka y same location pr point kr re hai. i.e. *y
If I delete content of pointer y^ in object a, as pointer y of object b is also pointing to the same location, it will also be deleted. This is a security flaw. To prevent this, we use deep copy.

Deep copy →

let two data members- int x;
int *y;



⇒ Deep copy mai shallow copy jesa nahi hoga. Jb obj a ka y copy hoga obj b mai to obj b ke y ke liye heap mai ek nyi location banegi so that any changes made in y of obj a does not reflect upon y of obj b. That's why we use deep copy.

Shallow copy →

Code →

```
class abc {
    public:
    int x;
    int *y;

    // default dumb ctor - shallow copy
    abc(const abc &obj){
        x = obj.x;
        y = obj.y;
    }

abc(int _x, int _y): x(_x), y(new int(_y)) { }

void print(){
    cout << "x:" << x << endl;
    cout << "y:" << y << endl;
    cout << "*y" << *y << endl;
}
};

int main(){
    abc a(1, 5);
    a.print();
    abc b(a);
    b.print();
    *b.y = 15;
    a.print();
    b.print();
}
```

O/P →

x : 1
y : 0x851350
*y : 5
} a

x : 1
y : 0x851350
*y : 5
} b

x : 1
y : 0x851350
*y : 15
} a

x : 1
y : 0x851350
*y : 15
} b

Here, we can observe in the output that address y is same. If we change *y of object b, then *y of objectab will also change.

Deep copy →

Code –

```
class abc {
    public:
    int x;
    int *y;

    abc (const abc &obj){
        x = obj.x;
        y = new int (*obj.y);
    }

abc(int _x, int _y): x(_x), y(new int(_y)){ }

 void print( ){
    cout<<"x:"<< x <<endl;
    cout<<"y:"<< y<<endl;
    cout<<"*y:"<< *y <<endl;
  }
};

int main( ) {
    abc a (1, 5);
    a.print( );
    abc b(a);
    b.print( );
    *b.y = 15;
    a.print( );
    b.print( );
}
```

o/p →

x : 1
y : 0x1e1350
*y : 5
} a

x : 1
y : 0x1e1360
*y : 5
} b

x : 1
y : 0x1e1350
*y : 5
} a

x : 1
y : 0x1e1360
*y : 15
} b

In deep copy, y of obj a and obj b are pointing to different locations. On changing *y of obj b, *y of obj a doesn't change.