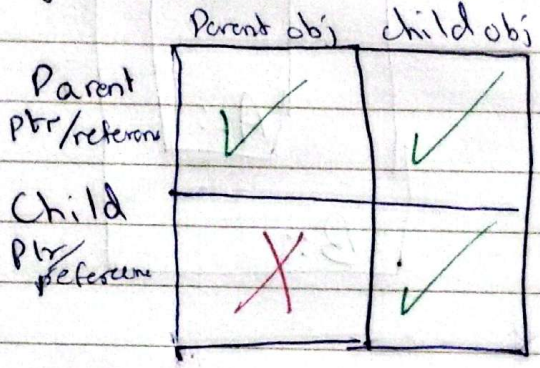
Day:

Date: / /

Polymorphism and virtual Function:

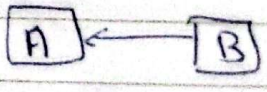
Polymorphism allows an object reference variable or an object pointer to reference objects of its derived classes.

This means that a parent pointer/reference can be used to point to a child object.



- a child pointer cannot point to a parent object.

Explain by example:




```
class A { };
class B: public A { };

int main() {
    A* a = new B;   // allowed.
    B* b = new A;   // Not allowed.
    B b;
    A& a_ref = (A&) b;   // allowed.
};
```

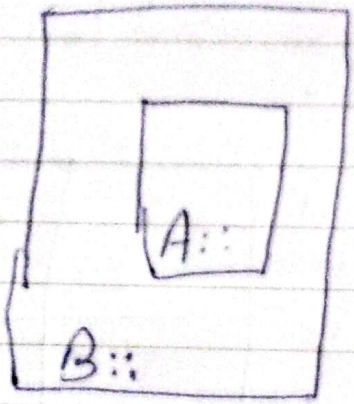
Yousaf

Day: Date: / /

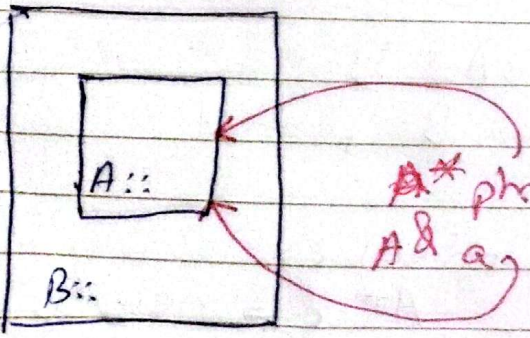
What and why is this Happening?

(How can a prent * or & point or make new child objects).

To understand this let's make a diagram.



A is a part of B. so when an A* ptr points to B it means that the ptr is pointing to the A inside B.



These both are pointing towards the inside of B, to A.

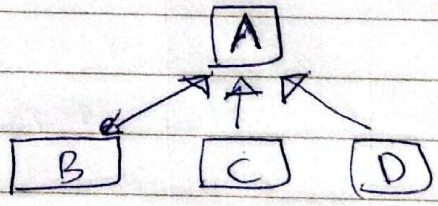
Yousaf

Day: Date: / /

- These pointers and references can only ACCESS the members of A.
- They CANNOT use any member of the derived class. (except if we use virtual, will learn that next).

Use of this:

e.g



now we can use A* to use B, C or D any thing we want with a single pointer.

consider these are made:

```
int main () {
    A *a[3];
    a[0] = new B;
    a[1] = new C;
    a[2] = new A;

// B *ptr_b = a[0];  ✗ Not allowed ↓ Fix.
   B* ptrB = (B*) a[0];  ✓ allowed.
}
```

This is allowed but Not good programing practice bez if we take delete a[0] b* will be unable to access it also.

Day: Date: / /

How to use functions or other members of derived classes using base pointer or referencee:

virtual functions:

We have virtual functions which can be used to access child members.

How to use them?

```
class A {

public :

        virtual void display ( ) {
                cout << "A" <<endl;
        }

};

class B : public A {

public:
  void virtual display( ) {
                cout << "B" <<endl;
    }
```

Yousaf

Day: Date: / /

```
class C: public B {

public:
        void display() {
            cout << "C" << endl;
        }

};
int main() {

    A* a1 = new C;
    A* a2 = new B;

    a1 -> display();
    a2 -> display();

}
```

Terminal

C

B

Explanation:

By using the virtual key word, when we call a function it always goes to the outer most class and checks if that function exists there. If it does IT is used.

Day: Date: / /

Important points:

1 - We only need to write virtual once in the most base class. And it will consider every function as virtual in derived classes.

2 - It doesn't matter if we write virtual before void or After it.

in the first point if in main we do:
B * b = new C;
now even if b doesn't have virtual written still display function is virtual as B still contains A as a parent.

3- we only need to write virtual in prototype or function definition. It should n't be written in outline function.

Yousaf

Date: / /

Overriding and Final keywords:

In the derive classes we don't need to but it is better to write override keyword for a virtual function. e.g

```
class B : public A {
Public:
    void display () override { ... }
};
```

Also if we want the chain to end e.g we only want display till B and not in C we can use final key word inside B.

```
class B: public A {
Public:
    void display () final { .... } };
```

Now even if we include a void display() fuction in C it will be an error.

Virtual destructors:

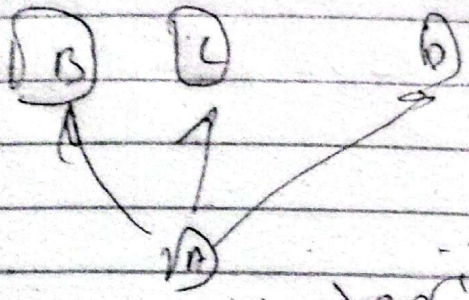
when we destroy a parent pointer pointing towards a child object only the parent inside the child is destroyed. So Destructors are made Virtuals.

Just write the virtual key word behind destructor and it will follow the same reverse pattern of destruction that destructors follow (deleting child first and then parent). (write virtual before ~).

Yousaf

OOP

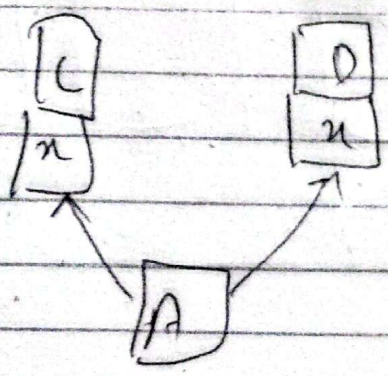
Multiple Inheritance



inheriting from more than one.

ambiguity error:

e.g.



so if we access x in A it gives ambiguity error but we can fix by using C::x and D::x.

- When we ~~read~~ inherit many classes they are created in that order:

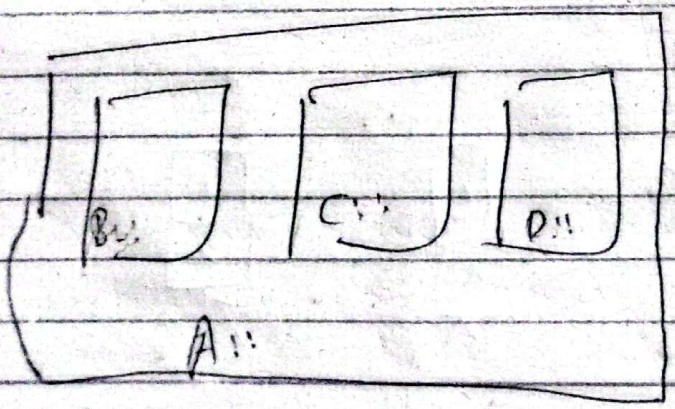
first B then C and lastly D created →

class A: public B, public C, private D {

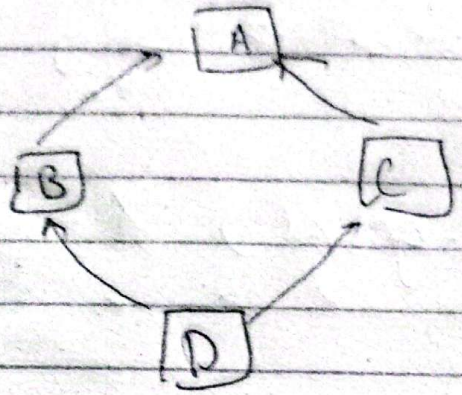
};

~~Base problem~~

On figure:



Diamond Problem:



no ambiguity in creation

but ambiguity comes when we access members

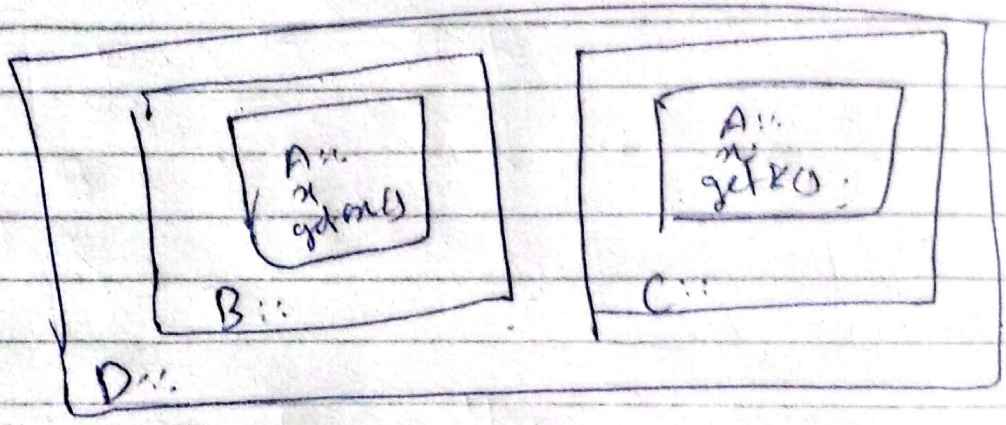
In creation simply 2 A's will be made but when we try to access then problem arises

code:

```
Class A{
    int x;

    +.
    };  int getX() { return x; }
Class B: public A{ },
Class C: public A{ };

Class D: public B, public C{ };

int main(){
    D dobj;
    dobj.getX(); ✗ error.
```

Now remember that D dobj ✓ is allowed as we are only creating not accessing.

But dobj.getX() X is error. why?,



In D now two ~~objects~~ ~~A~~ different A's have been made so .getX() is now 2 hence, which bring ambiguity.

Fixes:

dobj.B::getx(); ✓
dobj.C::getx(); ✓

Best fix because we only want one A:
1) to use virtual:

class A { — — };

class B: virtual public A { };
class C: public virtual B { };

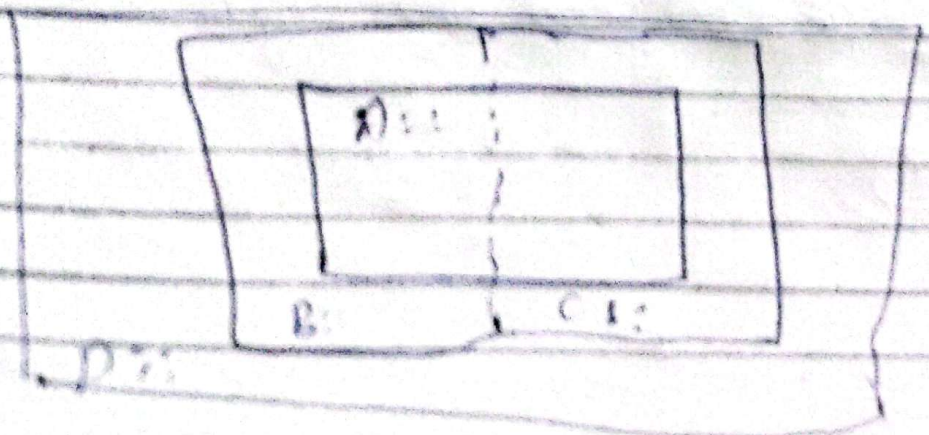
class D: public C, public B { ->};

Now

dobj.getX() // allowed.

How?

By using the virtual key word instead of B and making A, now d will make an A which will be share by both B and C.



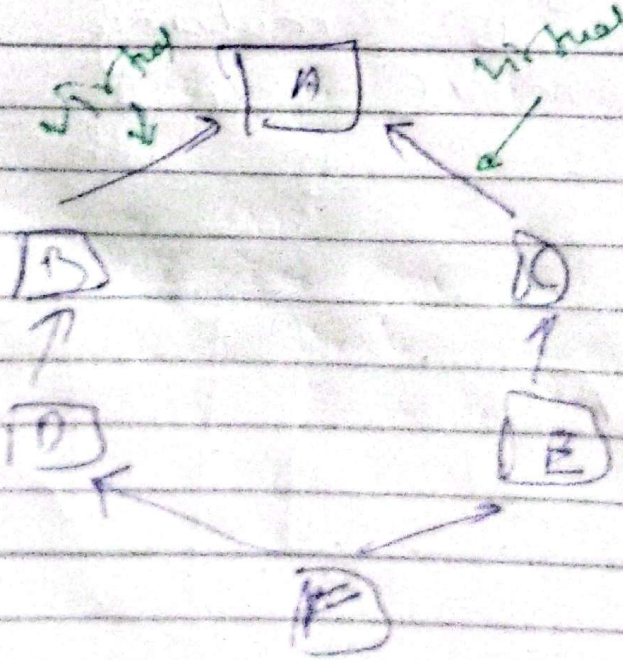
Now because D is making the A so in D's constructor we can also write

D() : A(...) { }; ✓ allowed.

as we are virtually going to A through D.

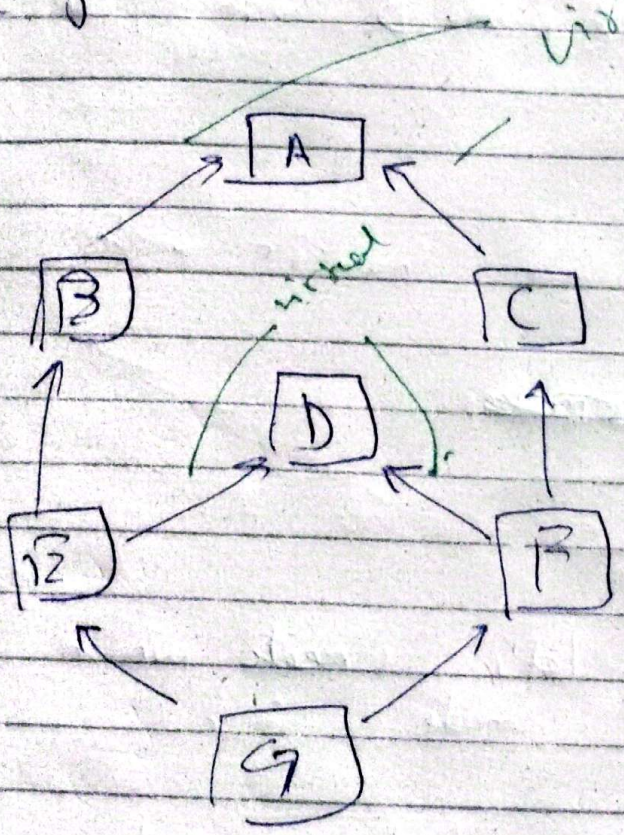
now B and C won't make A.

In these type of problem's, where should virtual really exist:



The virtual key word will be written with the class which will be made more than once

e.g.:



A should be modified as virtual as D should also be made virtual.

In

class B: virtual --- A { }.
class C: virtual --- A { }
class E: virtual --- D { }
class F: virtual --- D { }.

whichever class has chance of being made more than one is made virtual.

Considering the same diagram the way the program creates will be,

considering in class G : public B, public F {};

and in class E : public B, virtual public D {};

now let's start how it works:

~~G creates~~

Always first virtual classes are made by g.

so let's check which virtual made first.

we go follow path.

G goes E, E ~~goes~~ firstly goes to B which goes to A.

so A made first then come back and then D made.

if in G we had written.

class G: virtual public D, public B {};

Now

D would have been made first.

1— Highest priority → make all virtual classes first.

2) — by virtual by following normal order

3) — Then make other classes by following the again.