# A programozás alapjai 2.

Hivatalos laborsegédlet hallgatók számára.

## Alapprobléma

Tegyük fel, hogy olyan programot írunk, amiben különböző járműveket reprezentáló osztályokra van szükségünk. Rendelkezünk eddig *Car* és *Plane* osztályokkal (ne legyen közös ősosztályuk az egyszerűség kedvéért, ezzel a problémával később foglalkozunk).

A programunk tervezésekor tegyük fel, hogy már nagyon elterjedtek a repülő autók, így van igény arra, hogy ezt is lemodellezzük. Vegyük bele a repülőkre jellemző funkcionalitást a *Car* osztályba? Vagy fordítva, az autó funkcionalitását a *Plane* osztályba? Dehogy, hiszen nem minden autó tud repülni, és nem minden repülő használható autóként. Akkor hozzunk létre egy új osztályt *FlyingCar* néven. És ebbe hogy kerülnek bele a repülőkre és autókra jellemző tulajdonságok? Leimplementáljuk újból az egészet (esetleg copy-paste)?

Mielőtt erre válaszolnánk, egy kis emlékeztető: **nem duplikálunk** (copy-paste az ellenségünk)! Ha mégis így tennénk, és később kiderül, hogy valamit változtatni kell a *Car* vagy *Plane* funkcionalitásán, bővíteni kell vagy hibát találtunk, akkor ugyanezt a változtatást elvégezhetnénk a *FlyingCar* osztályban is. Ha pedig elfelejtjük, akkor rosszul fog működni a program, és sok **időbe** kerül kideríteni, hogy egy **elfelejtett** módosítás volt a baj.

DUPLIKÁCIÓT OLYAN MESSZIRŐL KERÜLJÜK, AMENNYIRE CSAK LEHET!

## Lehetséges megoldás: többszörös öröklés

Szóval nem, nem írjuk meg az összes funkcionalitást elölről egy új osztályban.

Ehelyett minden szükséges osztályból örököltetjük egyszerre a *FlyingCar* osztályunkat. Kibővíthetjük további tagváltozókkal, tagfüggvényekkel is (épp, mint az egyszerű öröklésnél), valamint rendelkezni fog minden ősosztály minden nem privát tulajdonságával. Minden ősosztályhoz külön megadható, hogy milyen láthatósággal örököltetünk belőle.

Pl. lehetne: `class FlyingCar: public Car, protected Plane {...}`

De most legyen mindkét öröklés public, csak megmutattuk, hogy akár ilyen is lehetne.

## Virtuális többszörös öröklés

### A közös ős

Írjuk tovább a programunkat, aztán feltűnik nekünk egy nagy hasonlóság az osztályaink között. Mindegyikkel lehet utazni, feltölteni valamilyen szükséges üzemanyaggal, javítani amikor elromlik stb., tehát van hasonló viselkedésük. Korábban tanultuk, hogy ha bizonyos osztályok **közös tulajdonságokkal** rendelkeznek, hozzunk létre nekik egy **közös ősosztályt**, és ebben implementáljuk a közös tulajdonságokat. (Az autónak, hajónak, repülő autónak stb. egyébként is van *közös összefoglaló neve*, ilyenkor gyanakodhatunk, hogy valószínűleg közös ősosztályuk is lesz).

Tehát hozzunk létre egy osztályt *Vechicle* néven, ami rendelkezik minden közös viselkedéssel. Ebből öröklődjön a *Car* és a *Plane* osztály. A *FlyingCar* osztálynak nem kell közvetlenül öröklődnie belőle, mivel már öröklődik pl. a *Car*-ból, így rajta keresztül a *Vechicle*-ből is közvetetten.

## 1 FlyingCar = 2 Vechicle? Nagy baj van!

Viszont itt belefutunk egy új problémába. A *FlyingCar* osztályunk jelenleg közvetlenül 2 osztályból öröklődik, ezek mindegyike öröklődik a *Vechicle*-ből. Ezzel az a nagyon nagy baj, hogy így a *FlyingCar* összesen kétszer öröklődik közvetetten a *Vechicle*-ből.

Nézzük meg, mit is jelent ez, és mi fog történni. Amikor létre szeretnénk hozni egy *FlyingCar* objektumot, először az ősosztályok konstruktorai futnak le. Legyen ez a sorrend most *Car*, majd *Plane.*

- Tehát először lefut a *Car* konstruktora. Ebben először a *Car* ősosztályának a konstruktora fut le (*Vechicle*). Így a memóriában lesz hely foglalva *Vechicle*-nek és *Car*-nak is.
- Aztán ugyanezt eljátsszuk a *Plane* ágon is: itt is foglalódik hely a *Plane* ősosztályának, a *Vechicle-nek*, és a *Plane*-nek is.
- Végül lesz hely foglalva a *Car* plusz adatainak, függvényeinek.

Így összesen kettő *Vechicle*-nek foglalódott hely a memóriában. Megkérdezhetjük, hogy na és ez miért baj?

- Memóriát nem pazarlunk.
- Semmi értelme, hogy egy repülő autó példány kettő jármű példány legyen egyszerre
- Amikor olyan függvényt szeretnénk hívni *FlyingCar* objektumon, ami a *Vechicle* osztályhoz tartozik, a fordító nem fogja tudni, hogy a kettő közül mi melyik *Vechicle* függvényére gondoltunk. Ez a felállás **fordítási hibát** okoz.

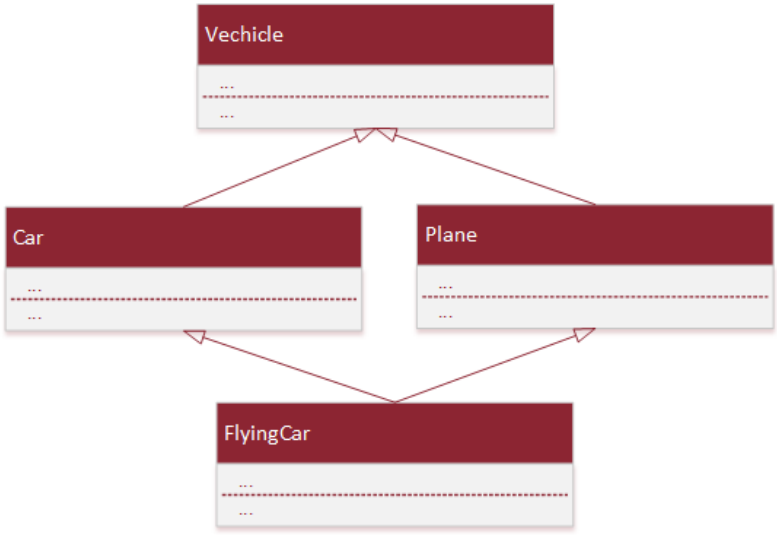
## Megoldás

Azt szeretnénk, hogy 1 *FlyingCar*-hoz 1 *Vechicle* tartozzon. Ebbe már a compiler sem köthetne bele, tudná melyik függvényt kell hívni, amikor *Vechicle* osztálybeli függvényt hívunk.

Pont ezt teszi lehetővé a **virtuális öröklés**. A *Car* és a *Plane* nem hoz létre külön-külön Vechicle memóriaterületeket, hanem egyen osztozkoznak.

Végső soron az öröklés így alakul:

```
class Vechicle {...};
class Car: public virtual Vechicle {...};
class Plane: public virtual Vechicle {...};
```

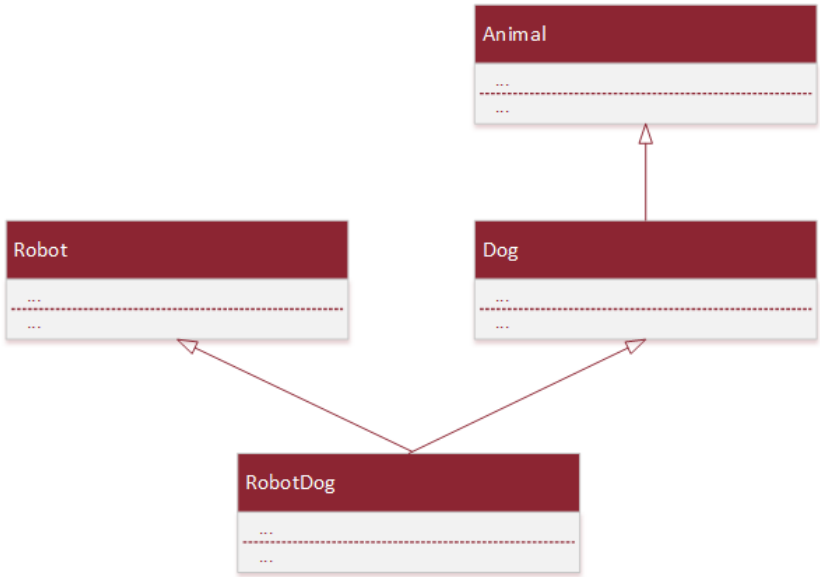



```
class FlyingCar: public Car, public Plane {...};
```

## Összefoglalás

Ha egy osztály olyan több, különböző osztályból öröklődik, melyeknek van közös ősosztálya, virtuális öröklésre van szükség. Könnyen felismerhető, mivel ha felrajzoljuk az öröklési hierarchiát, egy gyémánt-szerű alakot kapunk, ezért is hívják **“diamond problem”**-nek, vagy **“deadly diamond of death”**-nek.

Az viszont, ha két független, azaz különböző öröklési hierarchiából származó osztályból örököltetünk, nem jár ilyen gondokkal. Például ha van egy *Robot* osztályunk, és egy *Dog* osztályunk, melyeknek nincs közös őse, és ezekből örököltetünk egy *RobotDog* osztályt, nem szükséges virtuális öröklés, ez nem a *diamond problem* esete.



# Miért ne használjunk többszörös öröklést?

Többszörös öröklés használata előtt nagyon gondoljuk át, hogy tényleg ezt használva lehet-e a legjobban megoldani a problémát. Nem lehet hogy tartalmazás kellene helyette? Vagy valaminek egyáltalán nem kéne osztálynak lennie, hanem egy osztály egy tulajdonságának?

Egy olyan programot ami többszörös öröklést használ, nagyon nehéz fenntartani, bővíteni, debuggolni, vagy bármilyen minimális módon belenyúlni. Főként a *diamond problem* miatt keletkezhetnek olyan hibák, amik megkeseríthetik az életünket (azt pedig ugyebár nem szeretnénk). Természetesen előfordulhat, hogy muszáj használni, ekkor viszont nagyon nagyon figyeljünk oda a virtuális öröklésekre.

Ha nem is áll fenn a *diamond problem*, még lehet baj, amennyiben egy osztály több őse is definiál ugyanolyan nevű és paraméterezésű függvényeket. Ekkor nem egyértelmű, hogy ha ilyen függvényt hívunk a leszármazott osztály egy objektumán, melyik ősosztályban definiált függvényre gondoltunk. Szerencsére ezt explicit tudjuk egyértelműsíteni. Tegyük fel, hogy a fentebb említett *Robot-Dog-RobotDog* öröklési hierarchia esetén a *Robot* és *Dog* osztályban is definiáltunk *run()* függvényt. Ekkor, ha egy *RobotDog* példányon *run()* függvényt hívunk, a fordító nem fogja tudni, hogy a *Robot* vagy a *Dog* osztályban definiált függvényt kell-e itt meghívni. Így tudjuk egyértelműsíteni, hogy mit szeretnénk: robotDogObject.Robot::run() VAGY robotDogObject.Dog::run().

# De mikor érdemes mégis használni?

Ha már **előre adott osztályok** vannak egy (third party) keretrendszerben, akkor sokszor azt **írják elő**, hogy származtassunk le belőlük. Így implementálhatjuk azokat a műveleteket, amiket a keretrendszer szeretne meghívni. Vagyis úgy **illesztünk a keretrendszerhez**, hogy **több osztályból származtatunk**! Mivel így csak a művelet deklarációját adják meg, **a leszármazott egyfajta hívható felületet, interfészt implementál**.

**A gyakorlat az, hogy maximum egy osztályból öröklünk, viszont több interfészt implementálunk.** Magasabb szintű nyelvek, mint például a C# vagy a Java nem is engedik a többszörös öröklést, helyette egy külön nyelvi elemet vezettek be, az *interface*-t, ami analóg azon C++-os absztrakt osztályokkal, amelyek csak tisztán virtuális, publikus tagfüggvényeket tartalmaznak.

Ezzel részletesebben a laboron ismerkedhettek meg.