Kedves Hallgatók!

Az eddig feltett videók, kidolgozott feladatok, példatár mellé készítettem ezt a sort, mivel páran jeleztétek, hogy a félkész/megoldással is rendelkező feladatok kevésbé motiválnak az önálló gyakorlásra. Ebben a sorban olyan feladatok vannak, amik akár zárthelyin is előfordulhatnak. A tervezési feladatok a 10 pontosokhoz hasonlítanak, a többiek a tipikus 5 pontos feladatok jellegét mutatják be. Ezekhez a feladatokhoz tehát külön megoldókulcsot nem biztosítok, hiszen otthon, Visual Studioval, illetve az órai emléktetőkkel stb. tudjátok ellenőrizni az eredményeteket.

(Megjegyzés: ez több, mint 1 ZH sornyi anyag. Ha 130 perc alatt végzel vele, akkor a ZH időbe is jó eséllyel bele fogsz férni. Ajánlom, hogy kisebb csoportokban beszéljétek át a megoldásaitokat, és hangosan magyarázzátok el a döntéseket *(szerinted mire kérdez rá a feladat? Mi miatt lesz a megoldásodban valami konstans, statikus, miért írsz default konstruktort, stb.)*

Bertalan

1. Kompozíció vs. aggregáció

a) Egy objektum többféleképpen is tartalmazhat pointert egy másik objektumra. Mi az elvi különbség a kompozíció és az aggregáció között?

b) Az alábbi példában mely esetben beszélünk kompozícióról, illetve aggregációról a Room, illetve Owner típusú tagváltozók esetén?

```
class Room {
	//...adattagok, pl. méretek.
};

class Owner {
	//...adattagok, pl. név
};

class House {
private:
	Room* livingRoom; //(1) nappali
	Room* bedRoom; //(2) háló
	Owner* landlord; //(3) a főbérlő
public:

};
```

c) Írjon a House osztályhoz konstruktort, illetve destruktort, amely a b) pontban leírt értelmezésnek megfelelően kezeli a tagváltozókat (átvétel paraméterként és/vagy létrehozás, megszűntetés stb.). Feltételezze, hogy a Room és Owner osztályoknak van default konstruktora, amely jelen helyzetben megfelelő.

2. Függvénynév túlterhelés

a) Mit nevezünk függvénynév túlterhelésnek (overloading)?

b) Miért, mikor van rá szükség?

c) Mutasson egy egyszerű, életszerű példát túlterhelt függvényekre (nem kell tagfüggvénynek lennie) és a használatra.

d) Hogyan lehetne megoldani a leírt példát függvénynév túlterhelés nélkül?

3. Százlábú

Írjon egy példa osztály részletet *Centipede* (százlábú) néven, amelynek *legCount* nevű egész értékű tagváltozója minden példány esetén azonos. Az osztályt egy megfelelő setter függvénnyel kizárólag úgy lehessen módosítani, hogy a *legCount* csak olyan egész lehessen, amely osztható 2-tel. (A tagválozók, tagfüggvények közül elég csak

a fentiek megvalósításához szükséges tagfüggvények ismertetése, a többit „…”-tal jelezheti. Az inicializálással most ne törődjön.)

4. Tervezési feladat (10p)

Permetező folyadékok szétterítésére is alkalmas traktorok (*Tractor*) bevetését támogató alkalmazást készítünk. A traktorok korlátozás nélküli számú permetező tartályt (*Spray*) tudnak hordozni. Minden tartály a benne levő folyadéktípus azonosítására rendelkezik egy típussal (*type*), ezt egész számmal azonosítjuk. A tartályok tartalmát szét is lehet permetezni (*vaporize*()), ami kiír egy szöveget és a permet típusát (pl. “Permet: 1”). A traktorokhoz permetező tartályokat (*Spray*) tudunk hozzáadni (*addSpray*), a tartályok elvételével nem kell jelenleg tördődnünk. A permetezőszert szétteríteni (permetezni) a lenti kódban (*)-gal jelölt módon, a traktor osztályán keresztül szeretnénk megvalósítani. Ha a tartályban tárolt permetezőszert már felhasználták egyszer (*used*), onnantól ezt kell kiírnia (pl. “Mar felhasznaltak.”). A traktor maga nem tudja, hogy az adott permetezőszerét már használták-e.

```
int main() {
  Tractor zetor;
  Spray s1(1); //1-es tipus, pl. csigaolo permet
  Spray s2(2); //2-es tipus, pl. bekataszito permet
  zetor.addSpray(s1);
  zetor.addSpray(s2);
  zetor[0].vaporize(); //(*) “Permet: 1”
  //A túlcímzéssel nem kell foglalkoznunk.
  zetor[0].vaporize(); // “Mar felhasznaltak.”
  zetor[1].vaporize(); //”Permet: 2”
}
```

Tervezze meg és implementálja a szükséges osztályokat. Vázolja fel az osztályok hierarchiáját, tartalmazást stb. UML jelöléssel, az ábrán feltűntetve az esetleges tagfüggvények fejlécét és láthatóságát. Használja a dőlt betűs osztály- és tag neveket. Ügyeljen az elegáns OO megoldásokra!

5. Tervezési feladat (10p)

Egy edzésre szolgáló gokartpálya üzemeltetéséhez készítünk szoftvert. Az egyes versenyautóknak (*Car*) van egy egyedi, egész szám azonosítójuk (*number*), amit vásárláskor örökre hozzájuk rendelnek. Az edzések során az egyes versenyautók esetén fontos, hogy az autókat reprezentáló objektumok tudják, hogy hányadikként (*order)* hajtottak a pályára. A pályára hajtás esetén az egyes Car objektumoknak meghívjuk az *entersTrack* függvényét. (Nem kell törődnünk vele, hogy edzés végén hogyan állítjuk alaphelyzetbe ezt a számot, mint ahogy azt is adottnak vesszük, hogy egy autó egy edzés során egyszer kerül a pályára).
Célunk, hogy a (*) sorban jelölt módon ki tudjuk íratni az autó adatait (azonosítóval (*number*), pályára lépés sorszámával (*order*)). Amíg az autó nem lép pályára, az *order* értéke lehet -1. Emellett a (**) sorban jelölt módon össze tudjuk vetni, hogy melyik autó lépett korábban pályára.

```
Car c1(107);
Car c2(999);
Car c3(2000);
c1.entersTrack();
c3.entersTrack(); //a 2000-s azonosítójú autó másodikként lép pályára
c2.entersTrack();
cout << c3; //(*) Kiirjuk c3 adatait: "Car #2000 entered the track as the 2. car"
if (c3 < c2) //(**) Ha c3 előbb került pályára, mint c2
{
  cout << c3.getNumber() << " entered the track earlier.";
}
```

Készítse el a *Car* osztályt és a szükséges (tag)függvényeket, amelyekkel a leírt működés megvalósítható! Ügyeljen a helyes és elegáns objektumorientált tervezési elvekre!
Használja a dőlt betűs osztály- és tag neveket.

6. iMSC feladat

Az előző feladatban az autókat tömbben gyűjtjük a következőképpen:

```
Car* cars[3]; (***)
cars[0] = &c1;
cars[1] = &c2;
cars[2] = &c3;
```

Készítsen egy függvényt orderCars néven, amely a (***) sorban leírt módon létrehozott tömbökben az autókat a pályára kerülésük sorrendjébe rendezi át!