Objektumorientált Programozás C++

ADATSZERKEZETEK ÉS ALGORITMUSOK
2. GYAKORLAT

OOP vs. egyéb programozási módszerek

Az objektumorientált programozás abban különbözik leginkább más programozási módszerektől, hogy a programrészeket, **hasonló feladatokat**, de főleg hasonló feladatokat leíró **adatokat** és az azokat felhasználó **metódusokat csokorba foglaljuk** (saját típust hozunk létre).

Ezáltal az objektum-orientált programozás jobban igyekszik közelíteni a világban lejátszódó valódi folyamatokhoz.

Az **objektum-orientált** programozás tehát hatékony megoldást képes nyújtani a legtöbb problémára.

Az **osztályok újrafelhasználásával**, könnyű bővíthetőségével nagymértékben tudja csökkenteni a szoftverek fejlesztéséhez szükséges időt.

OOP vs. procedurális programozás

Procedurális programozás esetén a feladatokat **változókra**, **adatszerkezetekre** és **eljárásokra** tagoljuk.

Objektum-orientált programozás esetén a feladatok elvégzését **osztályokra**, és **objektumokra** tagoljuk, amelyek adatokat és eljárásokat **enkapszulálnak**.

Tehát a procedurális programozás eljárásokat használ, amelyek adatokon/adatszerkezeteken dolgoznak, az objektum-orientált programozás pedig objektumokat használ, amelyek a saját adataikon dolgoznak.

v.push(e) vs. push(v,e)

OOP alapfogalmai

Osztály

Objektum, példány, egyed

Osztálydefiníció

- Példányváltozók
- Osztályváltozók
- Metódusok (lehetnek osztály- és példányszintűek)
- Konstruktor
- Destruktor

Láthatósági szintek

Egységbezárás

Adatelrejtés

Osztály, objektum létrehozása

Egy osztálynak lehetnek

- **Példányváltozói, példánymetódusai**
- **Osztályváltozói, osztálymetódusai**

Példányváltozóból minden egyes példányosított egyedhez kerül **egy-egy**, míg a többiből az azonos osztályból példányosított összes egyed csak **ugyanazt az egyet**, az osztályhoz tartozót éri el.

Egy objektum (egyed) egy osztály megvalósítása, más szóval példányosítása.

```
{ …
  Apple a;
  Apple* a = new Apple();
}
```

Változók, metódusok láthatósága

public: Az objektumot használó bármely kód számára közvetlenül hozzáférhető.

protected: A definiáló osztály és leszármazottainak objektumai számára elérhető.

private: Azon osztály objektumai számára elérhető (meghívható), melyben deklarálták azt.

Példa – Autó osztály

Valósítsunk meg egy Car osztályt C++-ban:

```
class Car { //az osztály kezdete
   // az osztály megvalósítása
}; //az osztály vége
```

Struct esetén minden dolog láthatósága alapesetben public, class esetén private.

Példa – Car osztály változók

Az osztálydefiníción belül felsorolhatunk változókat, megadott láthatósági szint alatt:

```
class Car {
   protected:
     string type;
     double consumption = 0.5; // C++11
     int capacity;
     double kmh;
     double petrol;
   public:
     char plateNumber[6] = {'A', 'B', 'C', '1', '2', '3'}; //
     C++11
     static int petrolPrice; //statikus = osztályszintű változó

};
```

Konstruktor

Programkód, ami a **példányosításkor automatikusan** végrehajtódik.

- Minden osztálynak van konstruktora. Ha mi nem írtunk, a **fordító automatikusan létrehozza** és a példányosításkor meghívódik.
- **Ha már írtunk konstruktort, nem jön létre automatikusan az üres konstruktor.**
- Hasonlít a metódusokra, de nem pont ugyanolyan (nem tag, mert pl. nem öröklődik)
- Konstruktorból lehet több is, **túlterhelésükre** ugyanaz vonatkozik, mint a függvények túlterhelésére.
- Ennek **neve megegyezik az osztály nevével**, nincs visszatérési értéke.

Példa Autó – konstruktor I.

Írhatunk két konstruktort:

```
Car(string _type, double _consumption, int _capacity):
type(_type), consumption(_consumption), capacity(_capacity),
kmh(50)

{
   petrol = 354.9;

}


Car(string _type, double _consumption, int _capacity, double
_kmh): type(_type), consumption(_consumption),
capacity(_capacity), kmh(_kmh)

{
   petrol = 354.9;

}
```

A változók megadott értékekkel való inicializációjára kényelmes az **inicializáló listát** használni.

Példa Autó – konstruktor II.

Írhatunk egy konstruktort is a két konstruktor helyett:

```
Car(string _type, double _consumption, int _capacity,
double _kmh = 50.0): type(_type),
consumption(_consumption), capacity(_capacity),
kmh(_kmh)
{
    petrol = 354.9;
}
```

Megjegyzés: nem lehetséges c++-ban egyik konstruktort a másikból hívni (igen, az internet tele van kerülő utakkal, de nem csinálunk ilyeneket).

Megjegyzés: Konstruktorba default paraméterek mindig az utolsó paraméterek (_kmh = 50.0).

Metódusok

Alprogramok, melyek **objektumokhoz** vannak kapcsolva.

Az osztály metódusait, ahogy láttuk a konstruktornál, szintén **túl lehet terhelni**.

Túlterhelés: ugyanolyan azonosítójú (nevű), de különböző szignatúrájú függvények létrehozása.

- **Szignatúra**: azonosító, paraméterlista, const (C++ esetén)

Kiegészítés: léteznek **konstans tagfüggvény**ek is, ezek nem változtathatják meg az objektum állapotát.

- Jelölés: **const** a függvény szignatúra végén.

Példa Autó - Metódusok

```
class Car {
 public:
   void refuel (double);
   double cost (double) const;
 private:
   double petrolNeed (double) const;

};
```

Publikus metódusok:

```
void refuel(double _litre){

    petrol += _litre;

}

double cost(double _km) const{

    return petrolNeed(_km) *
petrolPrice;

}
```

Rejtett metódus:

```
double petrolNeed(double _km) const{

    return ((double)_km / 100) *
consumption;

}
```

static

A **static** kulcsszóval az osztályhoz rendelünk hozzá egy változót vagy metódust.

Mivel a static metódus az **osztályhoz** tartozik, így nem használhat fel (olvashat, írhat) változókat az osztály példányaiból.

Egy static metódusnak az osztály minden egyes példányán meghívva pontosan ugyanazt az eredményt kell biztosítania, függetlenül az egyes példányokban tárolt információktól.

Egy static metódus példányosítás nélkül is lefut.

Példa Autó - static

```
class Car{
 public:
    static int petrolPrice;

    static int petrolCost(int _litre) {
      return petrolPrice * _litre;
    }
};
```

Inicializálás: osztály definíción kívül!

```
double Car::petrolPrice = 354.9;
```

Getter és setter

Osztályok leírásánál legtöbbször **elrejtjük a belső változókat** a külvilág elől:

- nem akarjuk, hogy megváltoztassák (integritás)
- módosítottan akarjuk megjeleníteni vagy átadni
- a változtatásra reagálni akarunk (esemény)
- vagy egyszerűen nem akarjuk, hogy lássák

Ezért használjuk a **private/protected** láthatóságot a változóknál.

Ilyenkor az elérést a **getter** és **setter** függvények biztosítják.

A **getter** függvényünk legyen **const**, hisz nem akarjunk, hogy egy sima paraméter lekérése módosíthasson a belső állapoton.

Példa Autó – getter és setter

```
class Car{
  private:
    //…
    double petrol;
    //…
  public:
    doube get_petrol() const { return petrol; }
    void set_petrol(double _petrol) { petrol = _petrol;
    }
};
```

Destruktor - I.

MyClass::~MyClass()

A destruktor az a programrészlet, mely az osztály által lefoglalt dinamikus területeket/egyéb erőforrásokat felszabadítja. (prepare to die)

Egyszerűen: ha a **konstruktorban new** van, a **destruktorban delete**.

Nem szabad közvetlenül hívni!

Blokk lokális változóinál a blokk végrehajtása után, automatikusan hívódik:

```
{
    File f;
    // kód
} // f destruktora itt automatikusan hívódik
```

Destruktor - II.

Dinamikusan lefoglalt memória területeknél sem hívjuk közvetlenül a destruktort.

A destruktor a delete meghívásakor fog lefutni:

```
MyClass* c = new MyClass();
// kód …
delete c; // c->~MyClass() hívása
```

Egy pointereket tartalmazó tömb esetén minden tagra meg kell hívni a delete parancsot! Pl. „for” ciklussal.

Operátorok

Saját típusainknak is lehetnek operátorai ugyanúgy, mint a beépített típusoknak. Ha ezzel a lehetőséggel szeretnénk élni, akkor a már létező operátorokat kell túlterhelnünk. Erre a következőkben látunk példát, de előtte, definiáljuk a **this** pointert.

A **this** pointer egy implicit osztály paraméter. Ezzel tudunk magára az objektumra hivatkozni. Ez a következő példán keresztül lesz érthetőbb:

Ha azt szeretnénk, hogy össze lehessen adni az objektumainkat, túl kell terhelnünk az összeadás operátort:

Operátorok

```
class  Complex{
   double re = 0.0;
   double im = 0.0;
public:
   Complex operator+ (const Complex& _other) const;

 };

 Complex Complex::operator+ (const Complex& _other) const{
   Complex c;
   c.re = this->re + _other.re;
   c.im = this->im + _other.im;
   return c;  }
```

Mikor van egy Complex a, b és „meghívjuk” a + b túlterhelt operátort, akkor a+b, ahol _a_ lesz a **this** tag, és b lesz az _other tag. Tehát itt a this pointerrel a példányra, mint önmagára hivatkozhatunk.

További info az operátorokról:
http://en.wikibooks.org/wiki/C++_Programming/Operators/Operator_Overloading
http://hu.wikipedia.org/wiki/Oper%C3%A1torok_(C%2B%2B)

Friend

Operátorokat túlterhelhetünk úgy is, hogy az nem az objektum tagfüggvénye. Ekkor szembesülünk azzal a problémával, hogy nem tudjuk elérni az objektum privát tagjait.

Erre jelent megoldást a friend kulcsszó!

Egy osztálynak friend-je lehet egy függvény, egy osztály, vagy egy másik osztály tagfüggvénye, ami így eléri private vagy protected adattagjait.

Tipikus ilyen példa a (<<, >>) operátorok túlterhelése, amikkel streamekre/-ről írhatunk/olvashatunk:

```
friend std::ostream& operator<<(std::ostream& stream, const Car& z);
```

Ezzel túlterheltük, az << operátort, vagyis cout << c << endl értelmezhető, amennyiben c Car típusú változó.

Megjegyzés: a friend kulcsszót rengeteg dologra használhatjuk, de a tárgy keretein belül

Assignment operator, copy constructor

Mindegyikkel egy objektumot másolunk le.

Az assignment operátor automatikusan lefut értékadásnál, míg a copy constructor érték szerinti paraméter átadásnál (ahol a paraméter nem pointer), ill. függvényből való visszatérésnél (ahol a visszatérési érték nem referencia vagy pointer típusú).

Akkor kell őket megírni, ha az osztályban van dinamikusan lefoglalt memóriaterületre hivatkozó mutató.

Ha nem írjuk meg, akkor csak ún. shallow copy történik, vagyis csak a változók értéke másolódik. (Tehát pointerek esetén csak a tárolt cím másolódik át, a mutatott memóriaterület nem.)

Assignment operator, copy constructor

Formájuk a következő:

```
class A {
      A (const A& _other);
      A& operator= (const A& _other);
};
```

Öröklődés

Új osztályt szeretnék **egy meglévő alapján** megírni.

- Egy osztály **adattagjait, metódusait felhasználva** szeretnénk egy **új funkcionalitás**t létrehozni.
- Szeretnénk a **meglévő osztályunkat specializálni** új adattagok és metódusok felvételével.
- Szeretnénk már meglévő metódusokat, adattagokat **felüldefiniálni** az új funkciónak megfelelően.

Öröklődés – C++ megoldás

Public öröklődés: `class Child: public Parent`

Az ősosztály public és protected tagjai a leszármazott public, illetve protected tagjai lesznek (a private tagok közvetlenül nem használhatók a leszármazottban!)

IS-A reláció

- Jelentése pl: `class Apple : public Fruit` → az Alma egy Gyümölcs

altípus képzés (ld. előadás)

Öröklődés – C++ megoldás

Protected öröklődés: class Child: protected Parent

Az ős osztály public és protected tagjai a leszármazott protected tagjai lesznek (a private tagok nem használhatók a leszármazottban!)

- Ha használni szeretnénk deklarálnunk kell public tagokat a leszármazottban
- HAS-A reláció, melyet a leszármazottak is látnak

Private öröklődés: class Child: Parent

- Az ősosztály public és protected tagjai a leszármazott private tagjai lesznek (a private tagok nem használhatók a leszármazottban itt sem!)
- Ha használni szeretnénk deklarálnunk kell public tagokat a leszármazottban
- HAS-A reláció, melyet csak az adott osztály ill. annak objektumai látnak

- **Az alapértelmezett a private öröklődés**

Példa projekt a fenti öröklések tesztelésére: OOP1

Osztályhierarchia

Az **öröklődési relációt**, az osztályok között **gráfként** megadva, **osztályhierarchiának** is nevezik.

C++ban van többszörös öröklődés

Többszörös öröklődés: egy osztálynak több őse is lehet

Az osztályhierarchia C++ esetén egy **irányított gráf**

Körkörös öröklődés tilos

Lehetőségek - I.

A leszármazott osztályban **közvetlenül használhatjuk a szülőosztály public és protected metódusait, tagváltozóit** (egy leszármazott osztály örökli a szülő osztály tagjait, így a szülő osztály **private tagjaiból is van példánya**, de ezekhez nem férhet közvetlenül hozzá).

Elfedés: Deklarálhatunk olyan nevű adattagokat, amelyekkel a szülő osztály már rendelkezik, elfedve a szülőosztálybeli elemeket.

Felüldefiniálás: Írhatunk példánymetódust, mely szignatúrája megegyezik egy szülőosztálybeli példánymetódussal, felüldefiniálva azt.

Túlterhelés: Túlterhelhetünk egy szülőosztályban meglévő metódust.

Lehetőségek - II.

Írhatunk **osztálymetódust**, mely szignatúrája megegyezik egy szülőosztálybeli osztálymetódussal, **elfedve** azt.

Írhatunk teljesen **új metódusokat**.

Deklarálhatunk **teljesen új adattagokat**.

Hivatkozhatunk a szülőosztály elfedett, felüldefiniált elemeire.

Példa: Taxi osztály az Autó osztályból

Írjunk Taxi osztályt felhasználva a már megírt Car osztályt.

Vegyünk fel **újabb adattagokat** a Taxi osztályba:

- pénztárca: A sofőr pénzének regisztrálása
- kilométerdíj: a fuvarozás díja kilométerenként

```
class Taxi : public Car{
   protected:
     double wallet;
     double kmCost;
};
```

Példa: Taxi - konstruktor

A leszármazott **nem örökli a szülő konstruktorát**. De van lehetőségünk a leszármazott konstruktorában **meghívni a szülő** valamelyik **konstruktorát**. Ám ennek a hívásnak az **inicializáló lista első tagjaként** kell szerepelnie.

Ha ezt nem tesszük meg, vagy ha nem is definiálunk konstruktort, **akkor is végrehajtódik a szülő paraméter nélküli konstruktora** (ha van ilyen, **ellenkező esetben fordítási hiba**!).

```
Taxi(string _type, double _consumption, int _capacity,
double _kmCost): Car(_type, _consumption, _capacity),
kmCost(_kmCost) {

    wallet = 0;

}
```

Konstruktor - Fontos

Egy leszármazott osztály konstruktora mindig a szülő osztály konstruktora után fut le.

(Még akkor is, ha az nincs explicit módon meghívva!)

Példa Taxi - Felüldefiniálás

```
double cost(double _km) const {
//elérhetjük a felüldefiniált függvényt az ősosztályban
    return Car::cost(_km) + (kmCost * _km);
}
void refuel(int _litre){
    petrol += litre;
    wallet-= litre * petrolCost;
}
```

Ha azt akarjuk, hogy ezen függvények felüldefiniálják az „Autó beli” függvényeket, akkor **virtual** kulcsszót kell használni a Car osztályban (később).

Példa Taxi – Új metódusok

Írhatunk a leszármazott osztályban új metódusokat:

- **Fuvarozást** megvalósító függvény
- Olyan **költség függvény**, ami egy főre megmondja az utazás költségét, ha egyszerre több utast fuvaroz a taxi.

```
double transfer (double _km)
{
      if(moveOn(_km) == _km)
      {
            wallet += cost(_km);
            return _km;
      }
      return 0;
}
double costPerPassenger(double _km, int _passenger)
{
     return cost(_km) / _passenger;
}
```

Absztrakt osztályok - I.

A **tervezés eszközei (generalizálást, azaz általánosítást tesznek lehetővé).**

Az **altípus reláció megadása** tervezés szempontjából sokszor **megkívánja.**

Absztrakt osztály: **hiányosan definiált osztály**. → Vannak benne olyan funkciók, amelyekről még nem tudjuk, hogy hogyan fognak működni, amelyek megvalósítását a leszármazottakra bízzuk.

- **Absztrakt az az osztály amelyben van legalább egy pure virtual függvény:**

```
virtual [függvény szignatúra] = 0;
```

Nem lehet belőle példányosítani, de lehet belőle örökölni.

Absztrakt osztályok - II.

Megjegyzés: Absztrakt osztály leszármazottja lehet maga is absztrakt, ha nem implementálja az ősosztályban specifikált függvényeket, vagy ha önmaga is tartalmaz nem implementált függvényeket. **Absztrakt osztályt** egyébként **úgy is létrehozhatunk, hogy a szülő nem absztrakt**.

Tegyük fel, hogy szeretnénk a Car mellé egy újabb járművet definiálni: Bike.

Definiáljunk egy absztrakt Vehicle osztályt, majd ebből örököltessük le a Car és a Bike osztályainkat.

Most tehát a specializálással ellentétes folyamatról van szó, az általánosításról!

Példa – Absztrakt osztályok tervezése

Nézzük meg, mely **közös attribútumokkal tudjuk jellemezni** a két különböző járművet:

típus, kilométer óra, férőhely, és a haladást megvalósító függvény

```
class Vehicle{
  protected:
     string type;
     double kmh;
     int capacity;
  public:
     virtual double moveOn(double _km) = 0;
};
```

Bicikli osztály

```
class Bike : public Vehicle {
  public:
     Bike(string _type, double _kmh = 0, int _capacity = 1) :
     Vehicle(_type, _kmh, _capacity){ }
     double moveOn(double _km) {
        kmh += _km;
        std::cout << "Megtettünk " << _km << " km utat." <<
        std::endl;
        return _km;
     }
};
```

Nézzük meg az OOP2 csomagban a teljes hierarchiát.

Polimorfizmus

A **polimorfizmus**, vagy más néven **többalakúság** az a jelenség, hogy **egy rögzített típusú változó több típusú objektumra hivatkozhat.**

Ez az úgynevezett **altípusos polimorfizmus**.

Egy A típus altípusa egy B típusnak, ha minden olyan helyzetben, ahol B típusú objektum használható, A típusú objektum is használható.

```
Car bmw;

Taxi taxi("Audi", 11, 4, 230);

bmw = taxi;

Car& ta = taxi;

Car* a = new Taxi("Mercedes", 11, 2, 400);
```

Statikus, dinamikus típus

Statikus típus: a deklarációnál megadott típus.

Dinamikus típus: a változó által mutatott objektum tényleges típusa.

A dinamikus típus vagy maga a statikus típus, vagy egy leszármazottja (különben fordítási hibát kapunk).

A statikus típus állandó, a dinamikus típus változhat a program futása során.

Megjegyzések

Attól még, hogy az absztrakt osztály nem példányosítható, attól még, mint típust használhatjuk!

Helytelen:

- `Auto* a=new Bicikli("Magellan");`
- `Taxi* b=new Auto("Honda",5,5,300);`

A statikus típus dönti el azt, hogy milyen műveletek végezhetők a változón keresztül.

Helytelen:

- `Auto bmw;`
- `Taxi taxi("Audi", 11, 4, 230);`
- `bmw=taxi;`
- `bmw.fuvar(100);`

Dinamikus kötés

Dinamikus kötésnek nevezzük azt a jelenséget mely során egy objektumon a statikus típus által megengedett műveletet meghívva, a műveletnek mindig a dinamikus típusnak megfelelő implementációja fog lefutni.

A dinamikus kötés jelensége természetesen csak akkor jön létre, ha a leszármazott osztályban van **felüldefiniált** (és nem túlterhelt!) metódus.

Ez C++-ban csak akkor jön létre, ha használjuk a **virtual** kulcsszót azon függvényekre, melyeken keresztül dinamikus kötést szeretnénk létrehozni, továbbá **csak referenciákon és pointereken keresztül** valósul meg!

Nézzük meg az OOP2 projekt main.cpp-jét.

Virtual destructor

Ökölszabály: Amennyiben olyan osztályt írunk, amiből lehetséges, hogy később leszármazunk, mindenképpen írjuk meg a **virtuális destruktorát**!

Mi történik, ha nincs virtual destructor?

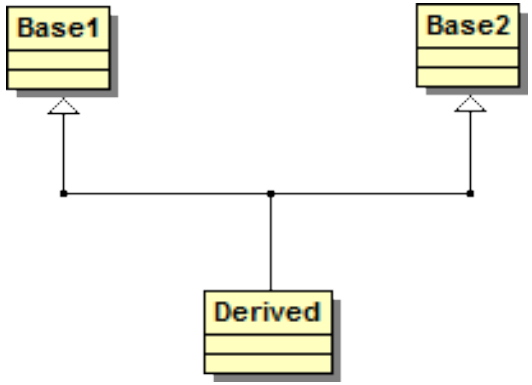
A statikus típus destruktora hívódik meg, így **csak** a statikus típus változói szabadulnak fel, és a dinamikus típus változóiból **memóriafolyás** lesz.

Nézzük meg a virtual_destructor projektet!

Többszörös öröklés

Lehetséges több osztályból is örököltetni!

```
class Derived: public Base1, public Base2 {
...
};
```



Mi történik ha mindkét ősben van azonos nevű függvény?

El **kell** döntenünk a függvény hívásánál, hogy melyiket szeretnénk használni… Pl: d.Base1::f(); d.Base2::f();

Nézzük meg a MultipleInheritance projektet.

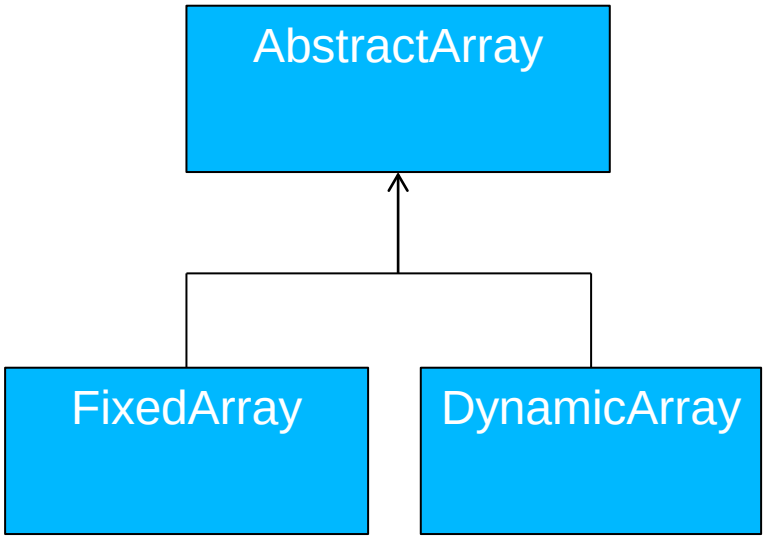
Feladat: Array osztályok G02F01

Az eddig tanultak alapján készítsük el az ábrán látható osztályokat a szokásos műveletekkel!

Az AbstractArray – mint ahogy neve is sugallja – legyen absztrakt.

FixedArray: ne legyen megváltoztatható a kezdeti mérete.

DynamicArray: ha a kezdeti lefoglalt memória nem elég, foglaljon le 2x annyit.



Gyakorló feladat G02F02

- Hozzatok létre egy absztrakt „Alakzat” osztályt, ennek legyen 2 metódusa, a kerulet_szamol() és a terulet_szamol()
- Hozzatok létre egy „Haromszog”, egy „Teglalap”, és egy „Kor” osztályt, amik az alakzat osztályból származnak.
- A „Teglalap” osztályból származzon le egy „Negyzet” osztály is.
- A kört leszámítva tartalmazzák az osztályok az oldalaik hosszát, a kör pedig egy sugarat.
- A main.cpp-ben hozzatok létre egy „Alakzat” pointer, vagy referencia tömböt, amit töltsetek fel tetszőleges alakzatokkal, és keressétek meg benne a legkisebb területűt, és a legnagyobb kerületűt!
- Írjatok olyan operátort minden alakzathoz, ami kiírja a az alakzat nevét, és paramétereit a „cout”-ra! Erre is csináljatok néhány próbát!

Gyakorló feladat G02F03

Írjunk egy iskolát modellező alkalmazást!

- Először is az iskolába járnak emberek, akiknek van nevük, születési évük és nemük.
- Az iskolában tanítanak tantárgyakat, amelyeknek van nevük, leírásuk és hozzá vannak rendelve évfolyamok, amikor ezeket oktatni kell.
- Az iskolába járó emberek között vannak tanulók, akik egy osztályba tartoznak, akiknek van egy tanulmányi átlaguk, és akiknél el kell tárolni a szülő/nevelő elérhetőségeit.
- Az iskolában tanítanak tanárok, akikhez bizonyos tárgyak vannak hozzárendelve. A tanárok között vannak osztályfőnökök, akiknek van osztályuk. A tanároknak van heti óraszámuk, órabérük, ami alapján a havi fizetésük kiszámolható.
- A tanárok között van egy igazgató, akinek a fizetése nem csak az órabér és a heti óraszám alapján képződik (van valamifajta plusz fizetés).
- Végül van az iskola, amiben vannak tanulók, tanárok, igazgató, aminek van egy címe, neve és ahol kiszámolható, hogy havonta összesen mennyi fizetést kell kiosztani.

Gyakorló feladat G02F04

Tervezzünk meg és írjunk meg egy alkalmazott osztályt!

Útmutatás:

- Gondold át, hogy egy alkalmazottat egy tetszőleges vállalatnál milyen adatokkal lehet reprezentálni.
- Ha ez megvan, gondold át azt, hogy mik lehetnek azok az eljárások/műveletek, amiket ehhez az alkalmazott osztályhoz hozzá lehetne kapcsolni. Gondold át, hogy a meglévő adattagokon milyen értelmes műveleteket/lekérdezéseket lehetne végrehajtani.
- Gondold át, hogy a fentiek összességéből melyek azok, amelyek minden alkalmazott esetén ugyanazok/ugyanazt az eredményt/hatást érik el, és melyek azok, amelyek minden alkalmazott esetén különbözőek.
- Majd gondold át, hogy mi az, ami hozzátartozna az osztály publikus interfészéhez, és mi az, amit érdemes elrejteni.
- Ha mindez megvan, a megfelelő nyelvi elemeket felhasználva írd meg az alkalmazott osztályt!

Gyakorló feladat G02F04 (folyt.)

Feladat: A megírt Alkalmazott osztályból leszármaztatva valósítsunk meg egy Főnök osztályt!

Útmutatás:

- Melyek azok a változók, amelyeket célszerű lenne elfedni?
- Milyen új változókat lehetne bevezetni?
- Melyek azok a metódusok, melyeket felül kéne definiálni?
- Milyen új metódusokat kellene megírni?

Majd ezután írj egy olyan main függvényt, melyben kipróbálod a polimorfizmus és a dinamikus kötés lehetőségeit!

Feladat: Mátrix osztály G02F05

Készítsünk egy mátrix osztályt:

- a konstruktor paramétereivel megadhatjuk méreteit, amiket aztán le is lehet kérdezni,
- legyen assignment operátora és copy constructora,
- legyen ==, + operátora,
- a () operátorral lehessen elérni az értékeit (amit aztán be is lehet állítani),
- legyen << operátora, amivel streamre írhatunk,
- végül legyen az osztálynak egy statikus függvénye, amivel egységmátrixokat lehet gyártani.

Házi feladat

Egy városban reggel mindenki megy valahova. Szimuláld a városban élők utazását és készíts róla statisztikát.

- A városban van nyugdíjas, felnőtt és diák. Generálj mindből 1 - 80 között valamennyit.
- Legyen koruk is, tanuló 0 - 25, felnőtt 26 - 61, nyugdíjas 62 - 80.
- Utazhatnak vonattal, autóval, taxival és biciklivel. Véletlenszerűen válaszd ki, hogy az egyes emberek melyik tömegközlekedési eszközt használják.
- Egy ember 15 - 45 km-et utazik, ez is véletlenszerű.
- A járműveknek van kapacitása: vonat 60, autó 5, taxi 4, bicikli 1. Ezeket töltsd fel emberekkel úgy, hogy a járművek mindig maximum távra (45 km) mennek. (ha 2 nyugdíjas 20 km-t megy vonattal, és 3 diák 40 km-t vonattal, ők utazhatnak egy járművön)
- A járműveknek van kilométerenkénti díja. Vonat 18 Ft/km, autó 36 Ft/km, taxi 50 Ft/km és biciklinél elégetett kalória van 33 kalória/km.

Házi feladat

- Továbbá vannak egyedi tulajdonságok.
- Vonat: típus („Stadler Flirt", „Ganz V43", „Siemens Desiro", „Mallard"), pálya szám (1000, 1001, 1002, 1003). Nem kell minden vonatot megalkotni, elég annyit amennyi éppen használatban van. Értsd, ha csak 100 ember utazik vonattal elég 2 vonatot példányosítani. (Az első vonat így „Stadler Flirt" és 1000 a pályaszáma, a második vonat „Ganz V43" és 1001 a pályaszáma stb.)
- Autó: Mikor gyártották (1990 - 2010).
- Taxi: Mikor gyártották (1990 - 2010), és egy száma (1- 100) véletlenszerűen.
- Bicikli: méret (24 - 32).
- Statisztikák, amiket el kell készíteni:
  - Konzolra: mennyit költöttek összesen az emberek autóra, taxira vonatra, és mind háromra összesen, illetve összesen mennyi kalóriát égettek el. Milyen nevű és pályaszámú vonatok mentek, átlag bicikli méret és autók kora, taxik kora, taxik sorszáma.
  - Fájlba: minden emberről statisztikák így: 10. ember, aki 62 éves, 29 km-et utazik és vonattal.

Megjegyzés: a maximum ponthoz mindenképpen használni kell: absztrakt osztály, konstansok, polimorfizmus, dinamikus kötés, referencia, pointer.

A delete és a default tagfügvények

A fordító egy sor tagfüggvényt hoz létre, ha nem definiáljuk azokat. C++11-ben dönthetünk az automatikusan létrejövő tagfüggvények létrehozásáról (default) vagy elutasításáról (delete).

```
class Car {
 public:
   Car() = default; // Car c; OK
   Car(const Car&) = default; // Car b(c); OK
   Car& operator=(const Car&) = delete; // b = a; NEM OK
   virtual ~Car() = default;

};
```

A C++ fordító osztályszintű operátor-függvényeket (&, *, ->, new, delete) definiál az osztályokhoz, melyek használatát meg is tilthatjuk.

```
class C {
public:
   void *operator new(size_t) = delete;
   void *operator new[](size_t) = delete;
};
```

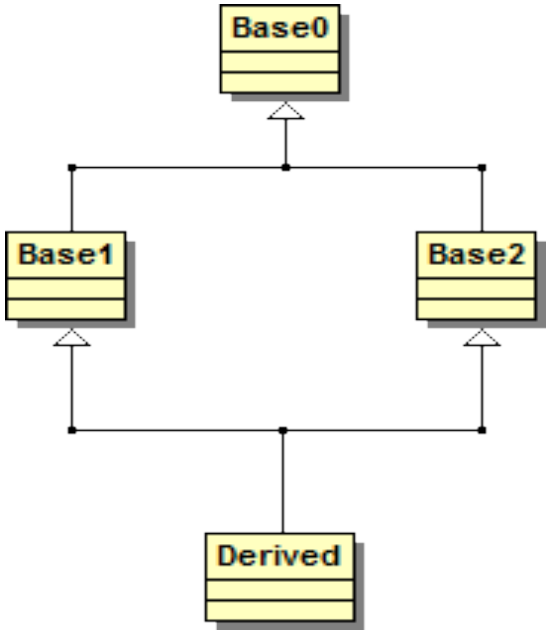
```
int main() {
   C *c = new C; // HIBA
   C *t = new C[3]; // HIBA
   C _c;
   C _t[10];
}
```

Felhasználó által definiált literálok

C++11 –ben lehetőség van saját literálok készítésére:

```
inline double operator"" _deg (long double degree) {
   return degree * 3.14159265 / 180.0;

}

double rad = 90.0_deg; // szög = 1.570796325


unsigned operator"" _Magic(const char* _magic) {
   unsigned b = 0;
   for(unsigned int i = 0; _magic[i]; ++i)
      b = b*2 + (_magic[i] == '1');
   return b;

}

int mask = 110011_Magic; // maszk = 51
```

Diamond Problem



Ilyenkor mi a helyzet???

Base1-ben és Base2-ben is megtalálhatók a Base0-ból örökölt tagok.

Megoldás: virtuális öröklődés

```
class Base0 {};
class Base1: public virtual Base0 {};
class Base2: public virtual Base0 {};
class Derived: public Base1, public Base2 {};
```