



Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

11. előadás

Objektumorientált programozás: öröklődés

Giachetta Roberto

`groberto@inf.elte.hu`

<http://people.inf.elte.hu/groberto>

Öröklődés

Példa

Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket.

- a négy alakzatot négy osztály segítségével ábrázoljuk (**Triangle**, **Square**, **Rectangle**, **Hexagon**)
- a háromszöget, négyzetet, hatszöget egy egész számmal reprezentáljuk (**_a**), a téglalapot két számmal (**_a**, **_b**)
- mindegyik osztálynak biztosítunk lekérdező műveleteket a területre (**area**) és a kerületre (**perimeter**)

Öröklődés

Példa

Tervezés:

Hexagon
- <code>_a :int</code>
+ <code>Hexagon(int)</code>
+ <code>area() :double {query}</code>
+ <code>perimeter() :int {query}</code>

Rectangle
- <code>_a :int</code>
- <code>_b :int</code>
+ <code>Rectangle(int, int)</code>
+ <code>area() :int {query}</code>
+ <code>perimeter() :int {query}</code>

Triangle
- <code>_a :int</code>
+ <code>Triangle(int)</code>
+ <code>area() :double {query}</code>
+ <code>perimeter() :int {query}</code>

Square
- <code>_a :int</code>
+ <code>Square(int)</code>
+ <code>area() :int {query}</code>
+ <code>perimeter() :int {query}</code>

Öröklődés

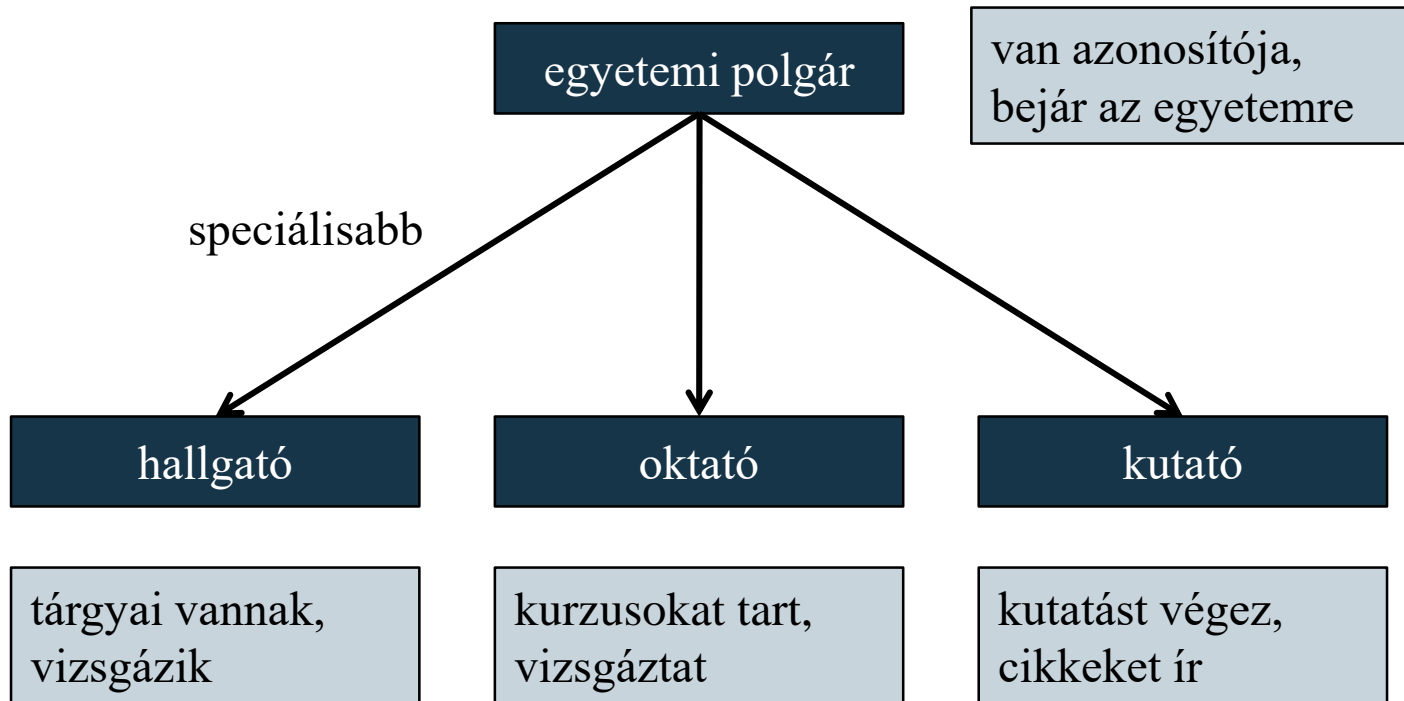
Kódismétlődés objektum-orientált szerkezetben

- Az objektum-orientált programokban a különböző osztályok felépítése, viselkedése megegyezhet
 - ez *kódismétlődés*hez vezet, és rontja a kódminőséget
- Hasonlóan procedurális programozás esetén is előfordulhat kódismétlődés, amely alprogramok bevezetésével kiküszöbölhető
 - objektumorientált programok esetén a működés szorosan összekötött az adatokkal
 - így csak együttesen emelhetőek ki, létrehozva ezzel egy új, *általánosabb* osztályt, amelyet össze kell kapcsolnunk a jelenlegi, *speciálisabb* osztállyal

Öröklődés

Általánosítás és specializáció

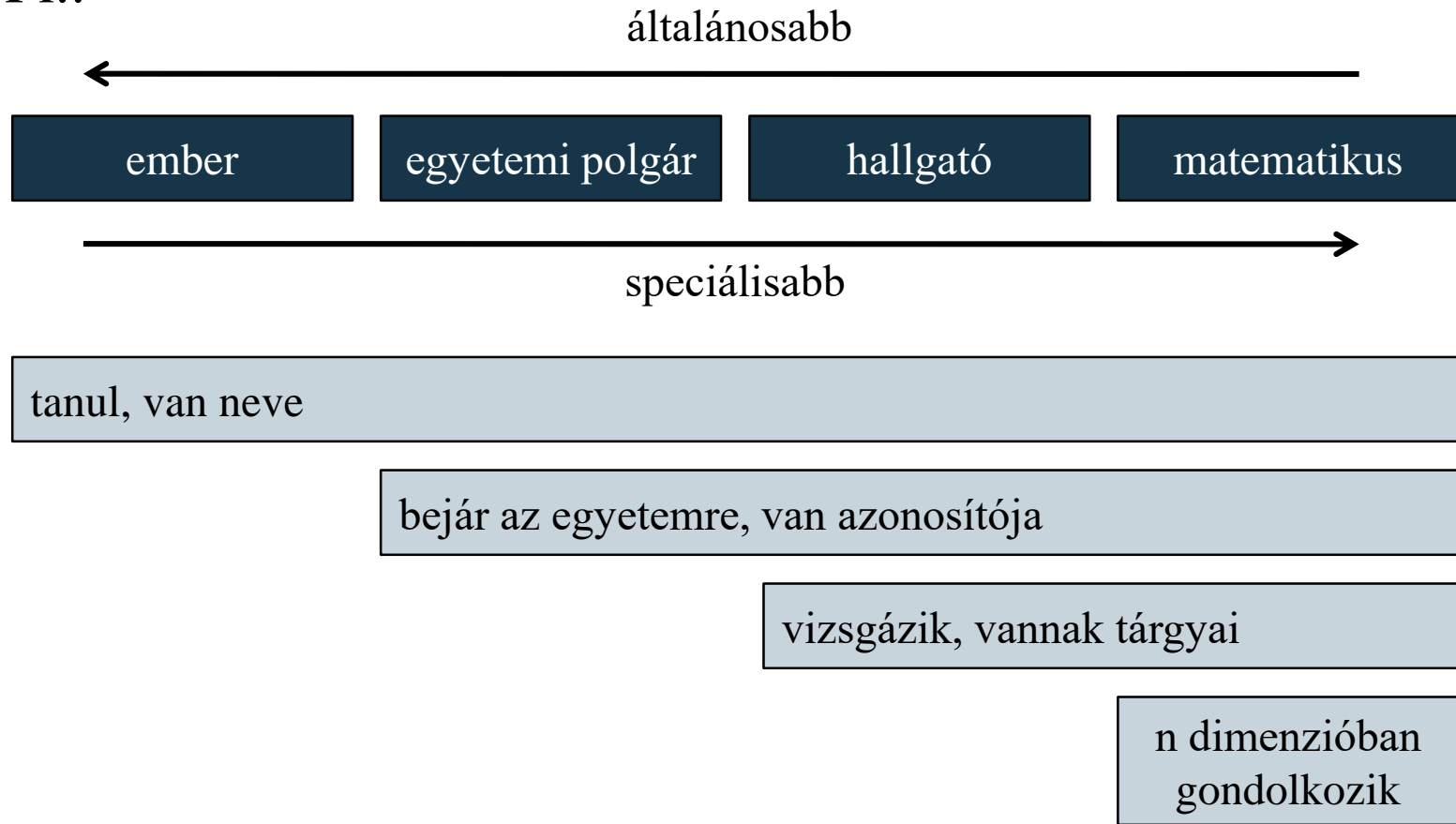
- Pl.:



Öröklődés

Általánosítás és specializáció

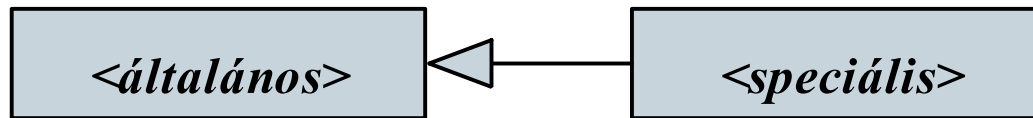
- Pl.:



Öröklődés

Általánosítás és specializáció

- Az általánosabb, illetve speciálisabb osztályok között fennálló kapcsolatot nevezzük *általánosításnak* (*generalization*)



- a speciális átveszi az általános összes jellemzőjét (tagok, kapcsolatok), amelyeket tetszőlegesen kibővíthet, vagy újrafogalmazhat
- az ellentétes irányú relációt nevezzük *specializációnak* (*specialization*)
- az általános osztályt *ősnek* (*base, superclass*), a speciális osztályt *leszármazottnak* (*descendant, subclass*) nevezzük

Öröklődés

Megvalósítása

- Az általánosítást a programozási nyelvekben az *öröklődés* (*inheritance*) technikájával valósítjuk meg, amely lehet
 - *specifikációs*: csak az általános absztrakt jellemezőit (interfészét) veszi át a speciális
 - *implementációs*: az osztály absztrakt és konkrét jellemzőit (interfészét és implementációját) veszi át a speciális

- Öröklődés C++ nyelven:

```
class <osztálynév> : <láthatóság> <ősosztály>
{
    <kiegészítések>
};
```


Öröklődés

Megvalósítása

- Pl.:

```
class Superclass // általános osztály
{
public:
    int value; // mező
    Superclass() { value = 1; } // konstruktor
    void setValue(int v) { value = v; }
    int getValue() { return value; } // metódusok
};
```

```
Superclass super; // osztály példányosítása
cout << super.value; // 1
super.setValue(5);
cout << super.value; // 5
```

Öröklődés

Megvalósítása

- Pl.:

```
class Subclass : public Superclass
    // speciális osztály, amely megkapja a value,
    // setValue(int), getValue() tagokat
{
public:
    int otherValue;
    Subclass() {
        value = 2; // használhatjuk az örökölt mezőt
        otherValue = 3;
    }
    void setOtherValue(int v) { otherValue = v; }
    int getOtherValue() { return otherValue; }
};
```

Öröklődés

Megvalósítása

- Pl.:

```
Subclass sub; // leszármazott példányosítása
```

```
// elérhetjük az örökölt tagokat:
```

```
cout << sub.value; // 2
```

```
sub.setValue(5);
```

```
cout << sub.value; // 5
```

```
// elérhetjük az új tagokat:
```

```
cout << sub.otherValue; // 3
```

```
sub.setOtherValue(4);
```

```
cout << sub.getOtherValue(); // 4
```

Öröklődés

Láthatóság

- A tagok láthatósága az öröklődés során is szerepet játszik
 - a látható (*public*) tagok elérhetőek lesznek a leszármazottban, a rejtett (*private*) tagok azonban közvetlenül nem
 - ugyanakkor a rejtett tagok is öröklődnek, és közvetetten (örökölt látható műveleteken keresztül) elérhetőek
 - sokszor hasznos, ha a leszármazott osztály közvetlenül elérheti a rejtett tartalmat, ezért bevezetünk egy harmadik, védett (*protected*) láthatóságot
 - az osztályban és leszármazottaiban látható, kívül nem
 - az osztálydiagramban # jelöli

Öröklődés

Láthatóság

- Pl.:

```
class Superclass {  
private:  
    int value; // rejtett mező  
public:  
    Superclass() { value = 1; }  
    void setValue(int v) { value = v; }  
    int getValue() { return value; }  
};
```

```
Superclass super;  
cout << super.getValue(); // 1  
super.setValue(5);  
cout << super.getValue(); // 5
```

Öröklődés

Láthatóság

- Pl.:

```
class Subclass : public Superclass {  
    // a value már nem látszódik, de öröklődik  
private:  
    int otherValue;  
public:  
    Subclass() {  
        setValue(2);  
        // nem látja a value mezőt, de  
        // közvetetten használhatja  
        otherValue = 3;  
    }  
    ...  
};
```

Öröklődés

Láthatóság

- Pl.:

```
Subclass sub; // leszármazott példányosítása
```

```
// elérhetjük az örökölt tagokat:
```

```
cout << sub.getValue(); // 2
```

```
sub.setValue(5);
```

```
cout << sub.getValue(); // 5
```

```
// elérhetjük az új tagokat:
```

```
cout << sub.getOtherValue(); // 3
```

```
sub.setOtherValue(4);
```

```
cout << sub.getOtherValue(); // 4
```

Öröklődés

Láthatóság

- Pl.:

```
class Superclass {  
protected:  
    int value; // védett mező  
public:  
    Superclass() { value = 1; }  
    void setValue(int v) { value = v; }  
    int getValue() { return value; }  
};
```

```
Superclass sup;  
cout << sup.getValue(); // 1  
sup.setValue(5);  
cout << sup.getValue(); // 5
```


Öröklődés

Láthatóság

- Pl.:

```
class Subclass : public Superclass {
    // minden elérhető lesz az ősből
private:
    int otherValue;
public:
    Subclass() {
        value = 2; // használhatjuk az örökölt mezőt
        otherValue = 3;
    }
    ...
};
```

Öröklődés

Példa

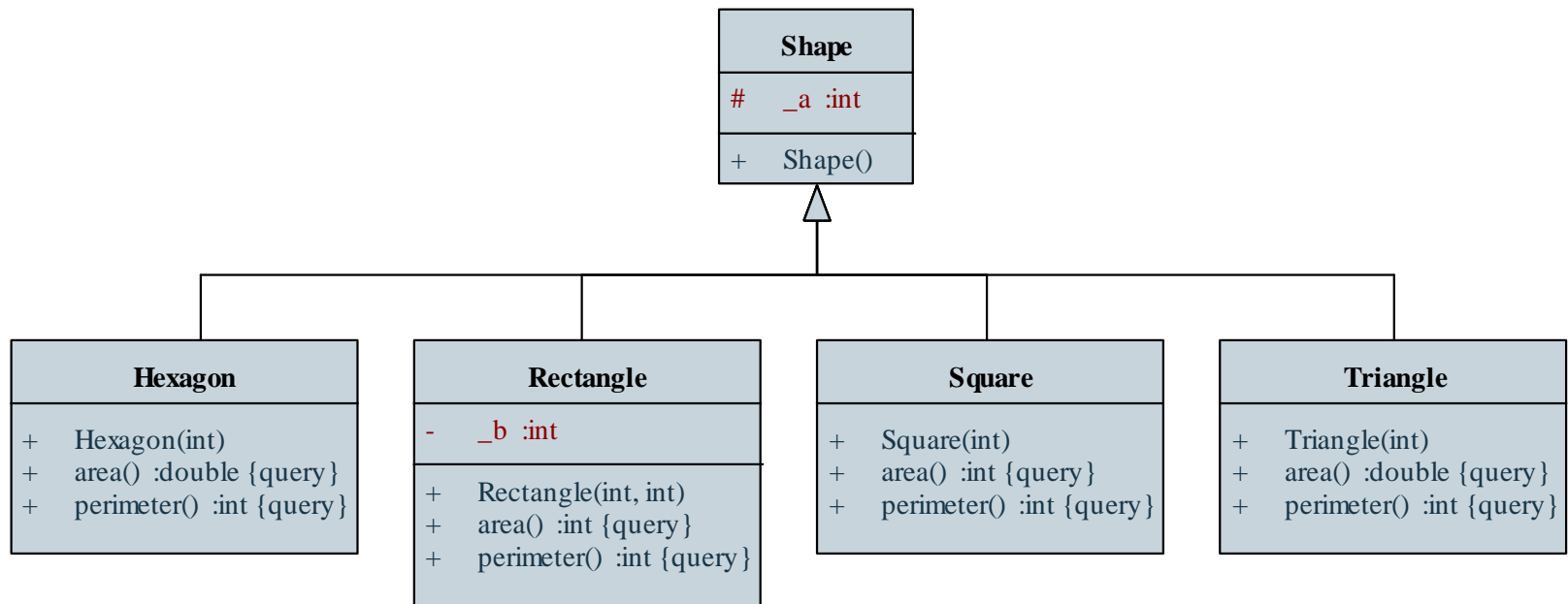
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket.

- javítsunk a korábbi megoldáson öröklődés segítségével
- kiemelünk egy általános alakzat osztályt (**Shape**), amelybe helyezzük a közös adatot (**_a**), ennek védett (**protected**) láthatóságot adunk
- a többi osztályban csak az öröklődést jelezzük, a működés nem változik (továbbra is a konstruktorok állítják be **_a** értékét)

Öröklődés

Példa

Tervezés:



Öröklődés

Tagok elrejtése és elérése

- Öröklődés során lehetőségünk van a viselkedés újrafogalmazására
 - a leszármazottban az ősével megegyező szintaktikájú metódusok *elrejtik*, vagy *felüldefiniálják* az örökölt metódusokat
 - lehetőségünk van explicit hivatkozni az ős bármely látható tulajdonságára az **<ős osztály>::** előtaggal
- Pl.:

```
class SuperClass {  
    ...  
    void getDefaultValue() { return 1; }  
};
```

Öröklődés

Tulajdonságok elrejtése és elérése

```
class SubClass : public SuperClass {
    ...
    void getDefaultValue() { return 2; }
        // elrejtő metódus
    void getDefaultSuperValue() {
        return SuperClass::getDefaultValue();
    } // ős metódusának meghívása
};

...
SuperClass sup;
cout << sup.getDefaultValue(); // kiírja: 1
SubClass sub;
cout << sub.getDefaultValue(); // kiírja: 2
cout << sub.getDefaultSuperValue(); // kiírja: 1
```

Öröklődés

A konstruktor és destruktor öröklődése

- A konstruktor automatikusan öröklődik
 - a paraméter nélküli konstruktor automatikusan (implicit módon) meghívódik amikor a leszármazottból létrehozunk egy példányt
 - elsőként az ős konstruktora hajtódik végre, azután a leszármazott konstruktora
 - lehetőségünk van az ős konstruktorának explicit meghívására is `<osztálynév> <konstruktor> : <ős konstruktornév>(<átadott paraméterek>)` formában
 - paraméteres konstruktorokra csak az explicit hívás használható

Öröklődés

Konstruktor és destruktork

- A destruktork automatikusan öröklődik és minden ős destruktork meghívódik a leszármazott destruktork meghívásakor
 - elsőként a leszármazott, majd az ős destruktork

- Pl.:

```
class FirstClass {  
public:  
    FirstClass() { cout << "1 start" << endl; }  
    ~FirstClass() { cout << "1 stop" << endl; }  
};
```

```
class SecondClass : public FirstClass {  
public:
```

Öröklődés

Konstruktor és destruktork

```
    SecondClass() { cout << "2 start" << endl; }  
    ~SecondClass() { cout << "2 stop" << endl; }  
};
```

```
int main() {  
    SecondClass second; // konstruktor hívás  
    return 0;  
} // destruktork hívás
```

```
/* eredmény:  
1 start  
2 start  
2 stop  
1 stop */
```


Öröklődés

Példa

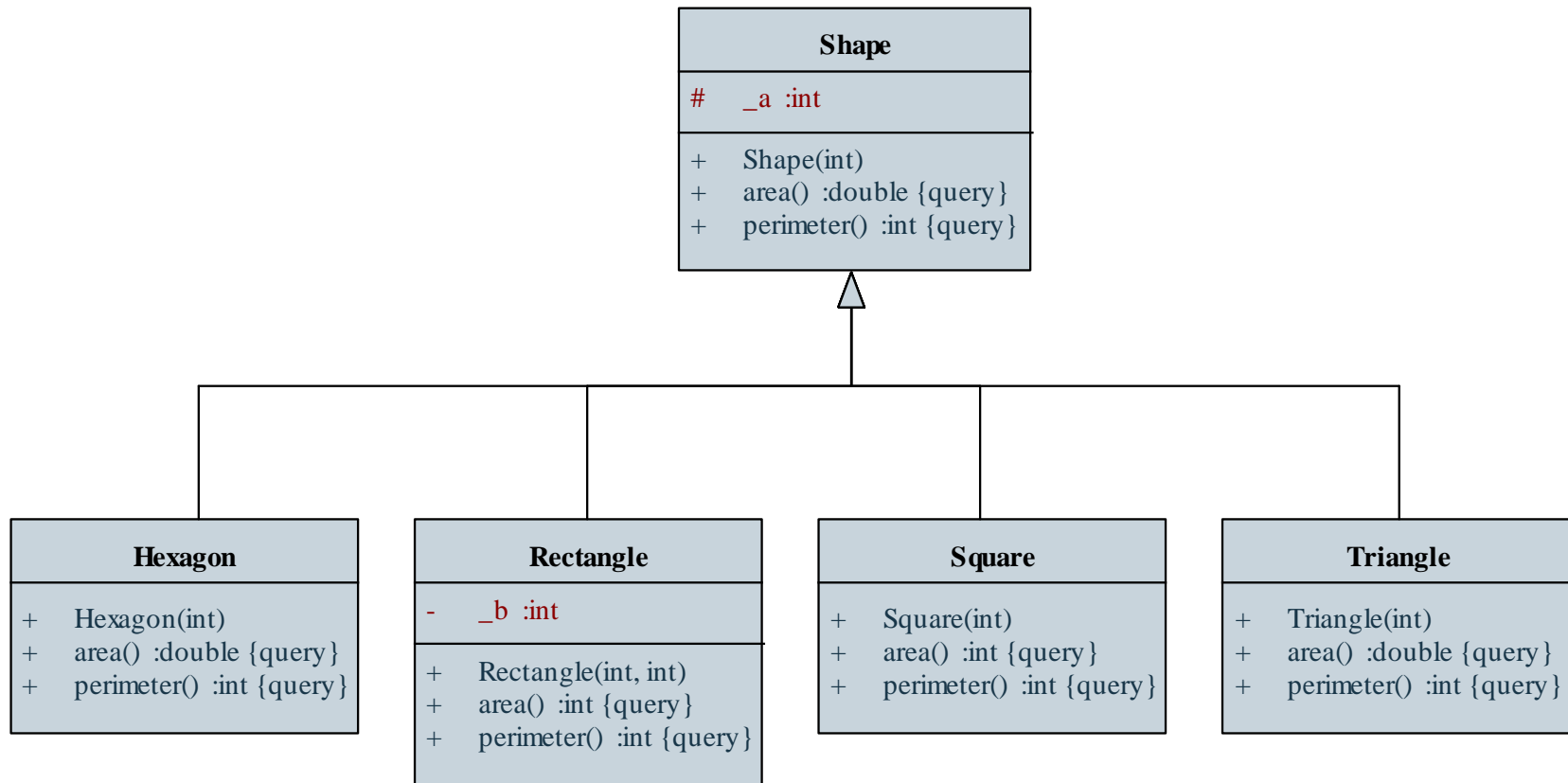
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket.

- javítsunk a korábbi megoldáson azzal, hogy az ős (**Shape**) konstruktorára bizzuk a mező (**_a**) inicializálását
- adhatunk az ősben műveleteket a terület és a kerület lekérdezésére, de ezeket a leszármazottban elrejtjük
- a leszármazott osztályok konstruktorai csak meghívják az ős konstruktorát, és tovább adják a kapott paramétert

Öröklődés

Példa

Tervezés:



Öröklődés

Polimorfizmus

- Mivel a leszármazott példányosításakor egyúttal az ősből is létrehozunk egy példányt, a keletkezett objektum példánya lesz mindkét típusnak
 - a leszármazott objektum bárhova behelyettesíthető lesz, ahol az ős egy példányát használjuk
- Ezt a jelenséget (*altípusos*) *polimorfizmusnak* (*polymorphism*, *subtyping*), vagy *többalakúságnak* nevezzük
 - pl. a **Subclass** **sub**; utasítással egyúttal a **Superclass** példányát is elkészítjük
 - öröklődés nélkül is fennállhat dinamikus típusrendszerű programozási nyelvekben (ez a *strukturális polimorfizmus*)

Öröklődés

Polimorfizmus

- A programozási nyelvek az objektumokat általában dinamikusán kezelik (referencián, vagy mutatón keresztül)
 - pl.:

```
Subclass* sub = new Subclass();  
cout << sub->getValue(); // 2
```
- A polimorfizmus lehetővé teszi, hogy a dinamikusán létrehozott objektumra hivatkozzunk az őosztály segítségével
 - pl.:

```
Superclass* sub = new Subclass();  
// a mutató az őosztályé, de ténylegesen  
// a leszármazott objektummal dolgozunk  
cout << sub->getValue(); // 2
```

Öröklődés

Polimorfizmus

- A dinamikusan létrehozott objektumot így két típussal rendelkeznek:
 - a hivatkozás osztálya az objektum *statikus típusa*, ezt értelmezi a fordítóprogram, ennek megfelelő tagokat hívhatunk meg
 - a példányosított osztály a változó *dinamikus típusa*, futás közben az annak megfelelő viselkedést végzi
- Pl.:

```
Superclass* s1 = new Subclass();  
    // s1 statikus típusa Superclass,  
    // dinamikus típusa Subclass  
Superclass* s2 = new Superclass(); // itt egyeznek
```

Öröklődés

Polimorfizmus

- A dinamikus típus futás közben változtatható, mivel az ősz típusú hivatkozásra tetszőleges leszármazott példányosítható
 - pl.:

```
Superclass* sup = new Superclass();  
cout << sup->getValue(); // 1  
delete sup;  
sup = new Subclass();  
cout << sup->getValue(); // 2
```
 - ugyanakkor a statikus típusra korlátozott az elérhető tagok köre, pl.:

```
cout << sup->getOtherValue();  
    // fordítási hiba, a statikus típusnak nincs  
    // getOtherValue() művelete
```

Öröklődés

Polimorfizmus

- A dinamikus típus műveleteinek elérésére típuskonverziót használhatunk a `dynamic_cast<típus>(mutató)` utasítás segítségével
 - helytelen konverzió esetén `NULL` mutatót ad vissza
 - pl.:

```
Superclass* sup = new Subclass();  
if (dynamic_cast<Subclass*>(sup)) {  
    // ha konvertálható az adott típusra  
    Subclass *sub = dynamic_cast<Subclass*>(sup);  
    // elvégezzük a konverziót  
    sub->getOtherValue();  
    // így már futtatható a művelet  
}
```

Öröklődés

Polimorfizmus

- A polimorfizmus azt is lehetővé teszi, hogy egy gyűjteményben különböző típusú elemeket tároljunk
 - az gyűjtemény elemtípusa az ős hivatkozása lesz, és az elemek dinamikus típusát tetszőlegesen váltogathatjuk

• Pl.:

```
Superclass* array[3];  
array[0] = new Superclass();  
array[1] = new Subclass();  
array[2] = new Subclass();  
  
for (int i = 0; i < 3; i++)  
    cout << array[i]->getValue(); // 1 2 2
```


Öröklődés

Dinamikus kötés

- Dinamikus példányosítást használva a program a dinamikus típusnak megfelelő viselkedést rendel hozzá az objektumhoz futási idő alatt, ezt *dinamikus kötésnek* (*dynamic binding*) nevezzük
 - tehát amennyiben *felüldefiniálunk* (*override*) egy műveletet, a dinamikus típusnak megfelelő végrehajtás fog lefutni
 - ehhez azonban a műveletnek engedélyeznie kell a felüldefiniálást, ekkor beszélünk *virtuális* (*virtual*) műveletről
 - virtuális műveletet a **virtual** kulcsszóval hozható létre
 - a konstruktor sohasem lehet virtuális

Öröklődés

Dinamikus kötés

- a nem virtuális műveletek a *lezárt*, vagy *véglegesített* (*sealed*) műveletek
 - lezárt műveletet csak elrejtteni lehet, amely esetben a statikus típus szerint hajtódik végre a művelet
 - alapesetben a műveletek lezártak

- Pl.:

```
class Superclass {  
    ...  
    virtual void getValue() { return value; }  
    // virtuális metódus  
    void getDefaultValue() { return 1; }  
    // véglegesített metódus  
};
```

Öröklődés

Dinamikus kötés

```
class Subclass : public Superclass {
    ...
    void getValue() { return otherValue; }
        // felüldefináló metódus
    void getDefaultValue() { return 2; }
        // elrejtő metódus
};

...
Superclass *sup = new Superclass();
cout << sup->getValue(); // 1
cout << sup->getDefaultValue(); // 1
sup = new Subclass();
cout << sup->getValue(); // 3
cout << sup->getDefaultValue(); // 1
```

Öröklődés

Dinamikus kötés

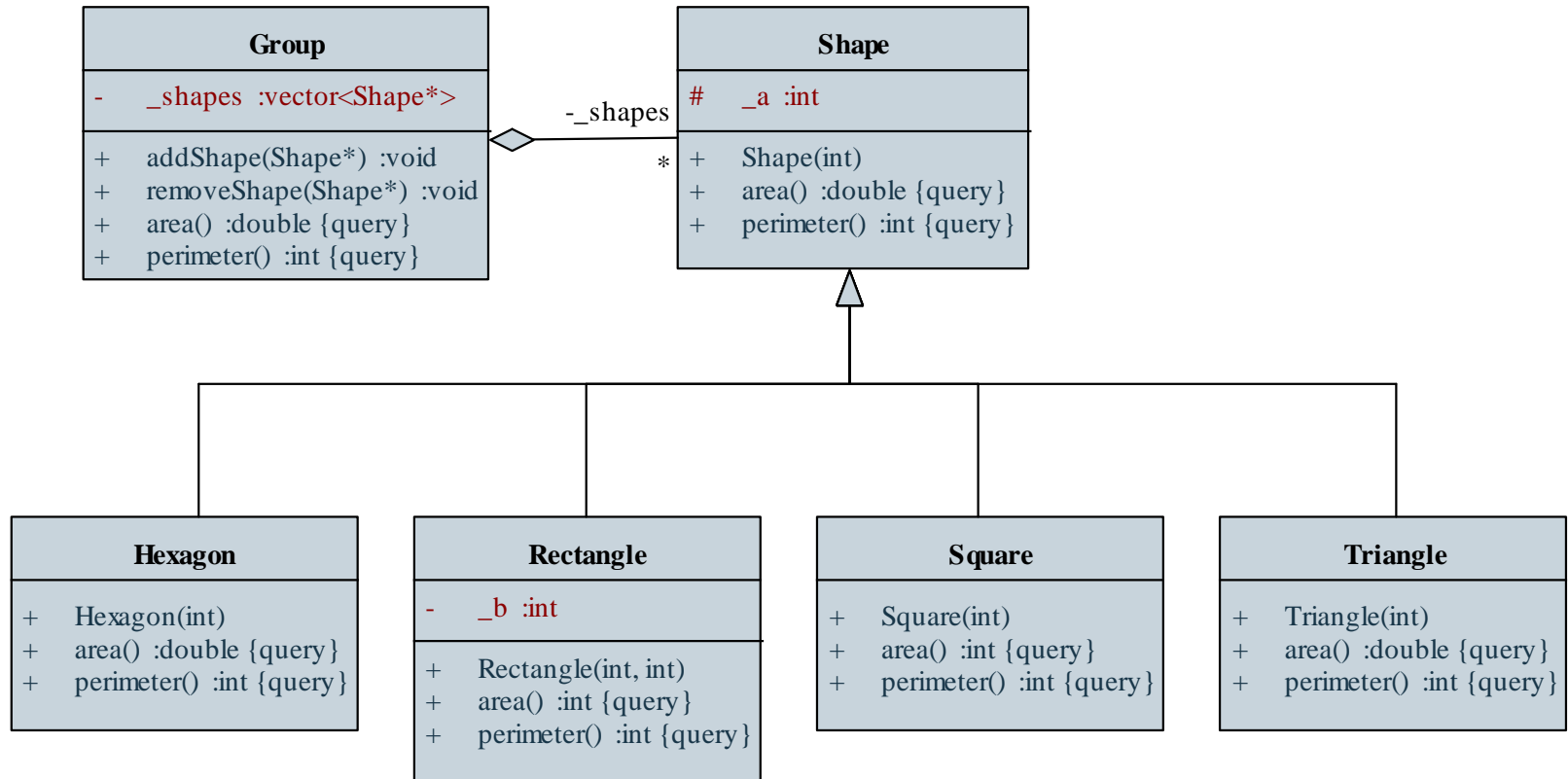
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket. Az alakzatokat csoportosíthatjuk is.

- az ősből a terület (**area**), illetve kerület (**perimeter**) lekérdezés metódusait virtuálissá változtatjuk, így már felüldefiniáljuk őket a leszármazottban
- létrehozuk az alakzatok csoportját (**Group**), amelybe behelyezzük az alakzatok gyűjteményét, polimorfizmus segítségével
- lekérdezhetjük a csoportba lévő elemek összterületét és összkörületét

Öröklődés

Dinamikus kötés

Tervezés:



Öröklődés

Virtuális destruktork

- A destruktork meghívása a többi metódushoz hasonlóan történik
 - amennyiben véglegesített, akkor a statikus típus szerinti destruktork hívódik meg
 - vagyis előfordulhat, hogy a leszármazott dinamikus típusban létrehozott dinamikus elemek nem törlődnek a memóriából
 - amennyiben virtuális, akkor a dinamikus típus szerinti destruktork hívódik meg
 - célszerű a destruktorkt minden osztályban virtuálisnak megadni, így az objektumot megsemmisítésével soha nem adódik probléma

Öröklődés

Virtuális destruktork

- Pl.:

```
class FirstClass {
public:
    FirstClass() { cout << "1 start" << endl; }
    ~FirstClass() { // véglegesített destruktork
        cout << "1 stop" << endl;
    }
};

class SecondClass : public FirstClass {
public:
    SecondClass() { cout << "2 start" << endl; }
    ~SecondClass() { cout << "2 stop" << endl; }
};
```

Öröklődés

Virtuális destruktork

```
int main() {  
    FirstClass* first = new SecondClass();  
    // konstruktor hívás  
    delete first;  
    // destruktork hívás  
    // (a FirstClass destruktorkára)  
    return 0;  
}
```

```
/* eredmény:  
1 start  
2 start  
1 stop */
```


Öröklődés

Virtuális destruktork

- Pl.:

```
class FirstClass {  
public:  
    FirstClass() { cout << "1 start" << endl; }  
    virtual ~FirstClass() { // virtuális destruktork  
        cout << "1 stop" << endl;  
    }  
};
```

```
class SecondClass : public FirstClass {  
public:  
    SecondClass() { cout << "2 start" << endl; }  
    ~SecondClass() { cout << "2 stop" << endl; }  
};
```

Öröklődés

Virtuális destruktork

```
int main() {  
    FirstClass* first = new SecondClass();  
    // konstruktor hívás  
    delete first;  
    // destruktork hívás  
    // (a SecondClass destruktorkára)  
    return 0;  
}
```

```
/* eredmény:  
1 start  
2 start  
2 stop  
1 stop */
```

Öröklődés

Absztrakt osztályok

- Amennyiben egy őosztály olyan általános viselkedéssel rendelkezik, amelyet konkrétan nem tudunk alkalmazni, vagy általánosságban nem tudunk jól definiálni, akkor megtilthatjuk az osztály példányosítását
- A nem példányosítható osztályt *absztrakt osztálynak* (*abstract class*) nevezzük
 - a diagramban dőlt betűvel jelöljük
 - csak statikus típusként szerepelhetnek
 - absztrakt osztályban létrehozható olyan művelet, amelynek nincs megvalósítása, csak szintaxisa (ezt =0 jelöli), ezek az *absztrakt*, vagy *tisztán virtuális* műveletek

Öröklődés

Absztrakt osztályok

- a leszármazottak *megvalósítják* (*realize*) az absztrakt műveletet (vagy szintén absztrakt osztályok lesznek)
- absztrakt osztály létrehozható a konstruktor elrejtésével, vagy absztrakt művelet definiálásával
- Pl.:

```
class Superclass { // absztrakt osztály
    ...
    virtual void getValue() =0; // absztrakt metódus
};
Superclass *sup = new Subclass();
cout << sup.getValue(); // 3
sup = new Superclass(); // fordítási hiba
```

Öröklődés

Példa

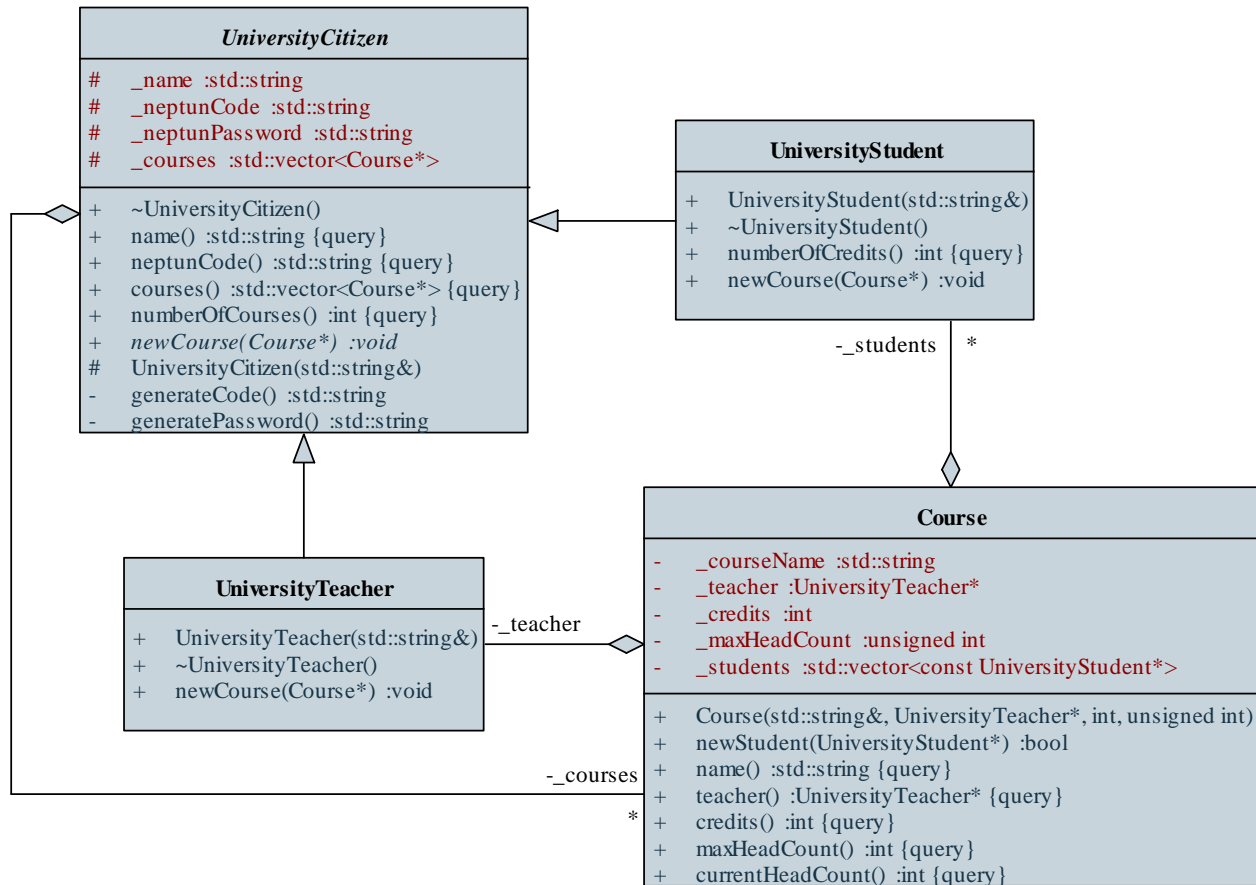
Feladat: Készítsünk egy programot, amelyben egyetemi oktatók, hallgatók és kurzusok adatait tudjuk tárolni.

- a hallgató (**UniversityStudent**) és az oktató (**UniversityTeacher**) közös tagjait kiemeljük az egyetemi polgár (**UniversityCitizen**) absztrakt őssosztályba
- a leszármazottak definiálják a **newCourse (...)** műveletet külön-külön, ezért az ősbőben tisztán absztrakt lesz, továbbá a hallgatónál megjelenik a kreditek lekérdezése (**numberOfCredits ()**)
- a változtatás a kurzus (**Course**) osztályra nincs hatással

Öröklődés

Példa

Tervezés:



Öröklődés

Példa

Megoldás (universitystudent.hpp):

...

```
class UniversityStudent : public UniversityCitizen  
{
```

```
    // hallgató osztálya, speciális egyetemi polgár  
public:
```

```
    UniversityStudent(const std::string& n);
```

```
    ~UniversityStudent() {}
```

```
    int numberOfCredits() const;
```

```
    void newCourse(Course* c);
```

```
};
```

...

Öröklődés

Példa

Megoldás (main.cpp):

```
...
UniversityTeacher groberto("Giachetta Roberto");
...
vector<UniversityCitizen*> citizens;
citizens.push_back(&groberto);
    // polimorfizmust használunk
...
cout << "Az egyetem polgárai: " << endl;
for (int i = 0; i < citizens.size(); i++){
    cout << citizens[i]->name() << ", NEPTUN: "
        << citizens[i]->neptunCode() << endl;
}
...
```