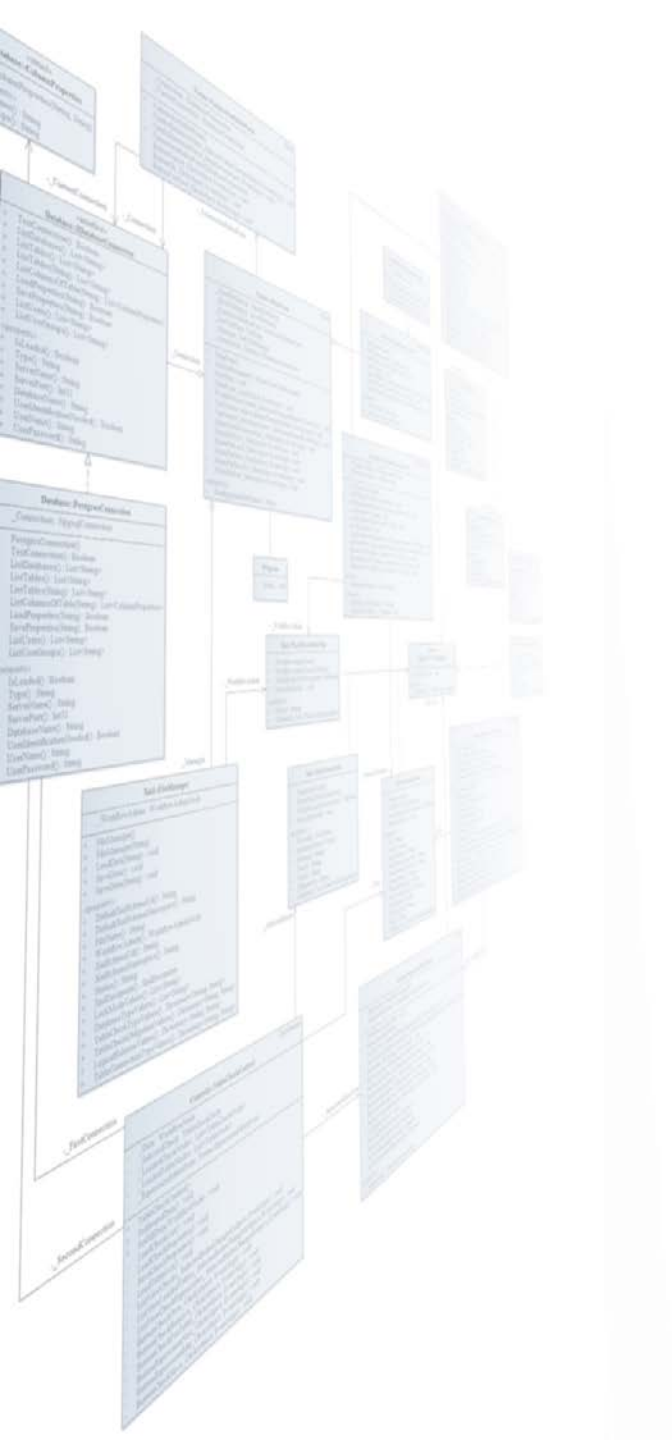


Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

# Bevezetés a Programozásba II

## 8. előadás

## Polimorfizmus

© 2014.03.31. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

# Polimorfizmus

## Öröklődés

- Objektumorientált programozásban kapcsolatot építhetünk ki osztályok között általánosabb-speciálisabb viszonylatban
  - a speciálisabb osztály átveszi az általás működését, adatait, ezt *öröklődés*nek nevezzük
  - pl.:

```
class MyBaseClass { … }; // ősosztály
class MyChildClass : public MyBaseClass { … }
   // leszármazott osztály
```

  - az öröklődés feloldja a *kódismétlődés*t, hiszen az ősosztályban definiálhatóak a közös részek, ugyanakkor a használatuk során az objektumok más típussal fognak rendelkezni (így másként kell kezelnünk őket)

# Polimorfizmus

## Az öröklődésben

- Pl.:

```
class MyBaseClass {
protected: // leszármazottban látható mezők
   int _val;
public:
   MyBaseClass() { _val = 1; }
   void PrintValue() { cout << _val; << endl; }
};
class MyChildClass : public MyBaseClass {
public:
   MyChildClass() { _val = 2; }
   // a konstruktor felülírja az értéket
   void DoSomething() { … }
}
```

# Polimorfizmus

## Az öröklődésben

- Ha példányosítunk egy osztályt, akkor az objektum típusa az osztály lesz, pl.: **`MyChildClass mcc;`**
  - ugyanakkor, ha az osztály leszármazott a példányban magában foglalja az ős minden tagját, így tekinthető az ős példányaként is
  - vagyis öröklődés esetén a példányoknak egyszerre több típusa is adott (a konkrét osztály, és annak valamennyi őse), pl. **`mcc`** típusa **`MyChildClass`**, de egyben **`MyBaseClass`** is
  - adott környezetben az objektum viselkedhet bármely típusának függvényében, és ez a környezetben szabályozható, ezt nevezzük *polimorfizmus*nak (*többalakúság*nak)

# Polimorfizmus

## Dinamikus kötés

- Az objektumok létrehozhatóak dinamikusan is
  - pl.:
    ```
    MyChildClass* mcp = new MyChildClass();
    mcp->PrintValue(); // kiírja: 2
    ```
  - az objektumra a típusának megfelelő mutató állítható rá, azonban a polimorfizmus miatt több típusa is lehet az objektumnak, így többféle mutató is alkalmazható, pl.:
    ```
    MyBaseClass* mbp = mcp;
       // az ős mutatóját állítjuk rá
    mbp->PrintValue(); // kiírja: 2
    ```
  - tehát egy mutatót ráállíthatunk bármely leszármazott típusának egy példányára, ezt *dinamikus kötés*nek nevezzük

# Polimorfizmus

## Statikus és dinamikus típus

- Dinamikus kötés esetén
  - az ősosztály a változó *statikus típusa*, ezt értelmezi a fordítóprogram, ennek megfelelő tagokat hívhatunk meg
    - statikus típusa több is lehet egy objektumnak
  - a leszármazott a változó *dinamikus típusa*, futás közben az annak megfelelő viselkedést produkálja
- Pl.:

```
MyBaseClass* mbp = new MyChildClass();
   // mbp statikus típusa MyBaseClass,
   // dinamikus típusa MyChildClass
MyChildClass* mcp = new MyChildClass();
   // itt a kettő megegyezik
```

# Polimorfizmus

## Statikus és dinamikus típus

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

```
MyBaseClass* mbp = new MyBaseClass();
mbp->PrintValue(); // kiírja: 1
delete mbcp;
mbp = new MyChildClass();
mbp->PrintValue(); // kiírja: 2
```

  - ügyelni kell arra, hogy az ős mutatóján keresztül a leszármazott műveletei nem érhetőek el, pl.:

```
mbp->DoSomething();
    // fordítási hiba, a statikus típusnak nincs
    // DoSomething művelete
```

# Polimorfizmus

## Típuskonverzió

- A dinamikus típus műveleteinek elérésére típuskonverziót használhatunk (megfelelő ősosztállyal)
  - típuskonverziót mutatókra biztonságos módon **`dynamic_cast<`*`típus`*`>(<`*`mutató`*`>)`** utasítással végezhetünk
  - helytelen konverzió esetén **`NULL`** mutatót ad vissza
- Pl.:

```
MyBaseClass* mbp = new MyChildClass();
if (dynamic_cast<MyChildClass*>(mbp))
   // ha konvertálható az adott típusra
   dynamic_cast<MyChildClass*>(mbp)
      ->DoSomething(); // így már futtatható
```

# Polimorfizmus

## Adatszerkezetbe szervezés

- A polimorfizmus azt is lehetővé teszi, hogy egy adatszerkezetben különböző típusú elemeket tároljunk
  - az adatszerkezet elemtípusa az ős mutatója lesz, és az elemek dinamikus típusát tetszőlegesen váltogathatjuk
- Pl.:

```
MyBaseClass* mbps[3];
mbps[0] = new MyBaseClass();
mbps[1] = new MyChildClass();
mbps[2] = new MyChildClass();

for (int i = 0; i < 3; i++)
   mbps[i]->PrintValue(); // kiírja: 1 2 2
```

# Polimorfizmus

## Példa

*Feladat*: Készítsünk egy programot, amelyben téglalapokat és köröket tudunk a képernyőre rajzolni, amelyek három színben (piros, fehér, zöld) váltakoznak. A színeket utólag le tudjuk cserélni (új véletlenszerű színre), és a teljes rajzot el tudjuk tolni egy vektorral valamilyen irányba.

- javítsunk a korábbi megoldáson azzal, hogy polimorfizmus segítségével egy közös vektorban tároljuk az összes adatot
- a vektor elemtípusa alakzat mutató (**`Shape*`**) lesz, az elemeket dinamikusan fogjuk létrehozni, majd a program végén törölni
- a speciális osztályokban lévő művelet eléréséhez típuskonverziót kell alkalmaznunk

# Polimorfizmus

## Példa

*Megoldás (`main.cpp`):*

```
…
int main() {
   …
   vector<Shape*> shapes;
      // alakzatok vektora, amely mutatókat tárol
   …
      if (isCircle)
         shapes.push_back(new Circle(…));
      else
         shapes.push_back(new Rectangle(…));
      // az alakzatokat dinamikusan hozzuk létre
   …
```

# Polimorfizmus

## Példa

*Megoldás (*`main.cpp`*):*

```
// a típust át kell alakítanunk a megfelelő
// leszármazottra, de előre nem tudjuk melyikre
// ezért mindkettőt kipróbáljuk
Rectangle* rec =
   dynamic_cast<Rectangle*>(shapes[i]);
if (rec != NULL && rec->contains(…)) {
   // a leszármazott mutatón keresztül már
   // elérhető az összes művelet
   shapes[i]->setColor(…);
   break;
}
Circle* cir = dynamic_cast<Circle*>(shapes[i]);
…
```

# Polimorfizmus

## Viselkedés felüldefiniálás

- A leszármazott amellett, hogy örököl minden tulajdonságot az őstől, lehetősége van *felüldefiniálni* (*override*) az örökölt viselkedést
  - az örökölt metódusok törzsét átírhatja, de a szintaxisát nem (más szintaxis esetén túlterhelésnek minősül)
  - a felüldefiniálható metódusokat nevezzük *virtuális metódusok*nak
    - a konstruktor sohasem lehet virtuális
  - az ősben előre kell jelezni, ha megengedjük a működés felüldefiniálhatóságát a **`virtual`** kulcsszóval
    - a kulcsszó csak egy felüldefiniálást tesz lehetővé

# Polimorfizmus

## Viselkedés felüldefiniálás

- Pl.:

```
class MyBaseClass {
public:
   virtual void PrintValue(){ cout << 1 << endl; }
   // felüldefiniálható (virtuális) metódus
};
class MyChildClass : public MyBaseClass {
public:
   void PrintValue(){ cout << 2 << endl; }
   // felüldefiniáló metódus
   void PrintBaseValue() {
      MyBaseClass::PrintValue();
   } // ős metódusának meghívása
};
```

# Polimorfizmus

## Viselkedés felüldefiniálás

```
…
MyBaseClass mbc; mbc.PrintValue(); // kiírja: 1
MyChildClass mcc;
mcc.PrintValue(); // kiírja: 2
mcc.PrintBaseValue(); // kiírja: 1
```

- Azon metódusokat, amelyek nem definiálhatóak felül nevezzük *véglegesített*nek
  - tehát alapértelmezetten minden metódus véglegesített
  - a konstruktor csak véglegesített lehet, viszont automatikusan meghívódik
- A virtuális műveleteket az osztálydiagramban *dőlt betű*vel jelöljük

# Polimorfizmus

## Viselkedés felüldefiniálás

- Polimorfizmus esetén, amennyiben a metódus
  - *virtuális*, életbe lép a viselkedés felüldefiniálás, és a dinamikus típus szerint kerül lefutásra, pl.:

```
… virtual void PrintValue() …
MyBaseClass* mbc = new MyChildClass();
mbc->PrintValue(); // kiírja: 2
```

  - *véglegesített*, akkor csak elrejtés történik, amit a program nem vesz figyelembe, így a statikus típus szerint kerül lefutásra, pl.:

```
… void PrintValue() …
MyBaseClass* mbc = new MyChildClass();
mbc->PrintValue(); // kiírja : 1
```

# Polimorfizmus

## Példa

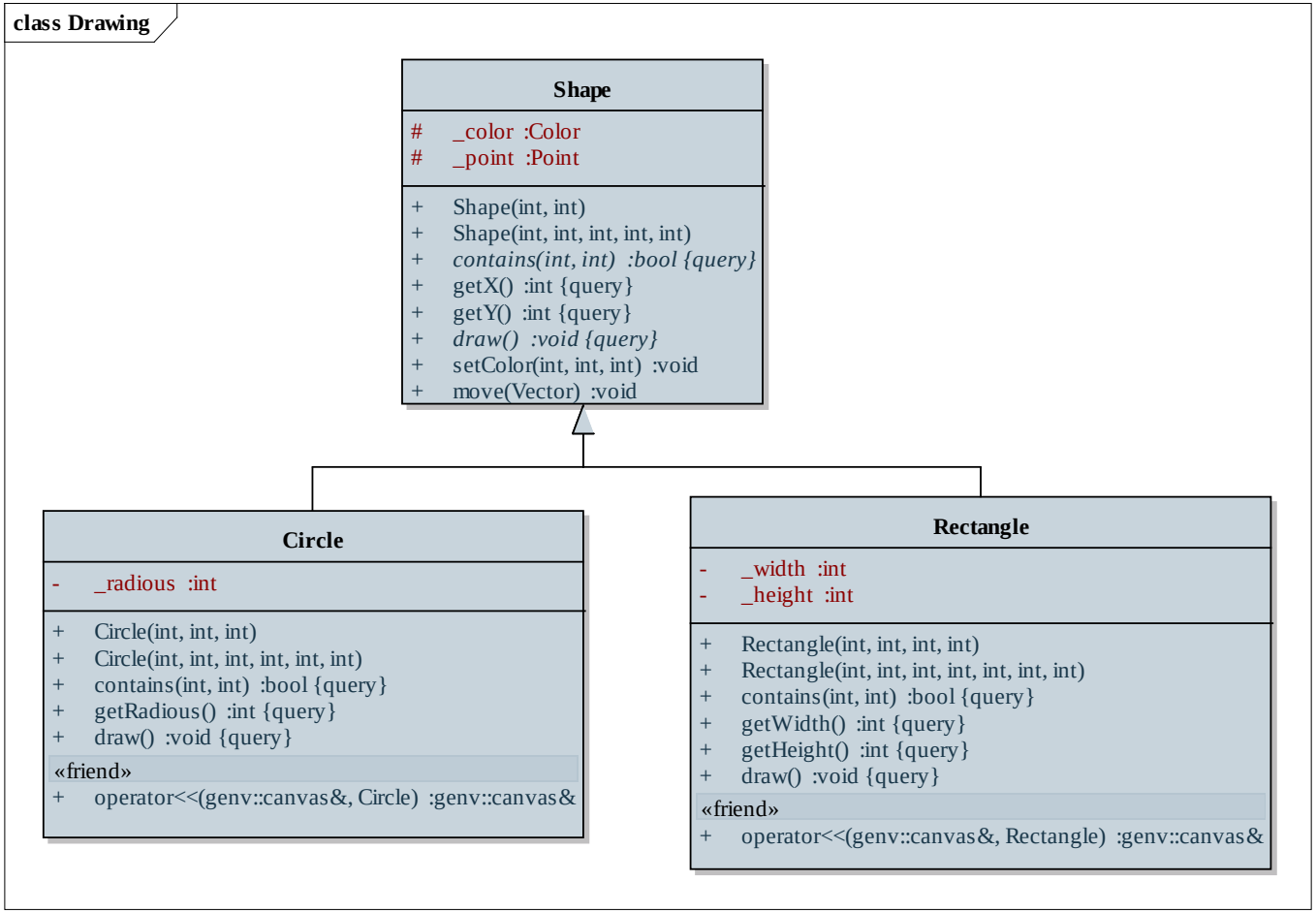
*Feladat*: Készítsünk egy programot, amelyben téglalapokat és köröket tudunk a képernyőre rajzolni, amelyek három színben (piros, fehér, zöld) váltakoznak. A színeket utólag le tudjuk cserélni (új véletlenszerű színre), és a teljes rajzot el tudjuk tolni egy vektorral valamilyen irányba.

- javítsunk a korábbi megoldáson azzal, hogy megadjuk az ősben a különböző tevékenységet végrehajtó műveleteket (**`contains`**, **`draw`**) virtuális műveletként, így nem kell típuskonverziót használnunk
- továbbra is a két speciális alakzat példányait helyezzük a vektorba, és azok típusa szerint fog lefutni a művelet

# Polimorfizmus

## Példa

*Tervezés:*



# Polimorfizmus

## Példa

*Megoldás (**shape.hpp**):*

```
class Shape { // alakzat osztály
   …
   virtual bool contains(int x, int y) const {
      return false;
   }  // virtuális (felüldefiniálható),
      // alapértelmezetten hamisat ad vissza
   …
   virtual void draw() const { }
      // virtuális (felüldefiniálható),
      // alapértelmezetten nem csinál semmit
   …
};
```

# Polimorfizmus

## Példa

*Megoldás (`main.cpp`):*

```
…
for (int i = shapes.size() - 1; i >= 0; i--) {
    // a műveletek meg vannak adva az ősben, így
    // nyugodtan meghívhatjuk őket
    if (shapes[i]->contains(ev.pos_x, ev.pos_y)) {
        // a dinamikus típus szerint fog
        // végrehajtódni a művelet
        shapes[i]->setColor(…);
        break;
    }
}
…
```

# Polimorfizmus

## Absztrakt metódusok

- Virtuális metódusok esetén a felüldefiniálás opcionális, ugyanakkor kötelezővé is tehető azáltal, hogy nem adjuk meg a metódus törzsét az ősben
  - ekkor egy *absztrakt*, vagy *tisztán virtuális* metódust hozunk létre, amelyet a leszármazottnak kötelező felüldefiniálnia
  - ekkor a törzs helyébe **`=0;`**-t kell írnunk
- Pl.:

```
class MyBaseClass {
public:
   virtual void PrintValue() = 0;
   // felüldefiniálandó (absztrakt) metódus
};
```

# Polimorfizmus

## Absztrakt osztályok

- Amennyiben egy osztályban van absztrakt metódus, akkor az osztály nem példányosítható (mivel a metódusa nem végrehajtható), ekkor az osztályt *absztrakt osztály*nak nevezzük
  - általában akkor hasznos, ha öröklődéssel fogjuk össze az osztályokat egy közös ősbe, de az őst nem akarjuk példányosítani, mert olyan általánosság mellett nem adható meg a működése
  - egy osztály úgy is lehet absztrakt, hogy a konstruktorát elrejtjük a külvilág elől (ekkor szintén nem példányosítható)
- Absztrakt osztályokat, illetve metódusokat a osztálydiagramban *dőlt betűvel* jelöljük

# Polimorfizmus

## Interfészek

- Pl.:

```
MyBaseClass* mbc = new MyBaseClass();
// hiba, absztrakt osztály nem példányosítható
MyBaseClass* mbc = new MyChildClass();
// a MyChildClass már példányosítható, mert ott
// meg van adva a SomeMethod törzse
```

- Definiálhatunk speciális absztrakt osztályokat, amelyek nem tartalmaznak megvalósítást, csak felületet, ezek az *interfész*ek
  - interfészben csak publikus, absztrakt metódusok lehetnek
  - leginkább többszörös öröklődés kapcsán használatosak
  - az interfészeket *megvalósítjuk* (nem örököljük)

# Polimorfizmus

## Interfészek

- Pl.:

```
class MyInterface { // interfész
public:
   virtual void PrintValue() = 0;
}; // csak absztrakt függvényeket tartalmazhat

class MyBaseClass : public MyInterface {
public:
   virtual void PrintValue() {
      cout << 1 << endl;
   }
   // megvalósítjuk az interfészt, de a művelet
   // felüldefiniálható marad
};
```

# Polimorfizmus

## Interfészek

```
class MyChildClass : public MyBaseClass {
public:
   void PrintValue() { cout << 2 << endl; }
   void PrintBaseValue() {
      MyBaseClass:: PrintValue();
      // meghívjuk az ős metódusát
   }
};

…
MyInterface* mi = new MyBaseClass();
mi->PrintValue(); // kiírja: 1
mi = new MyChildClass();
mi->PrintValue(); // kiírja: 2
```

# Polimorfizmus

## Példa

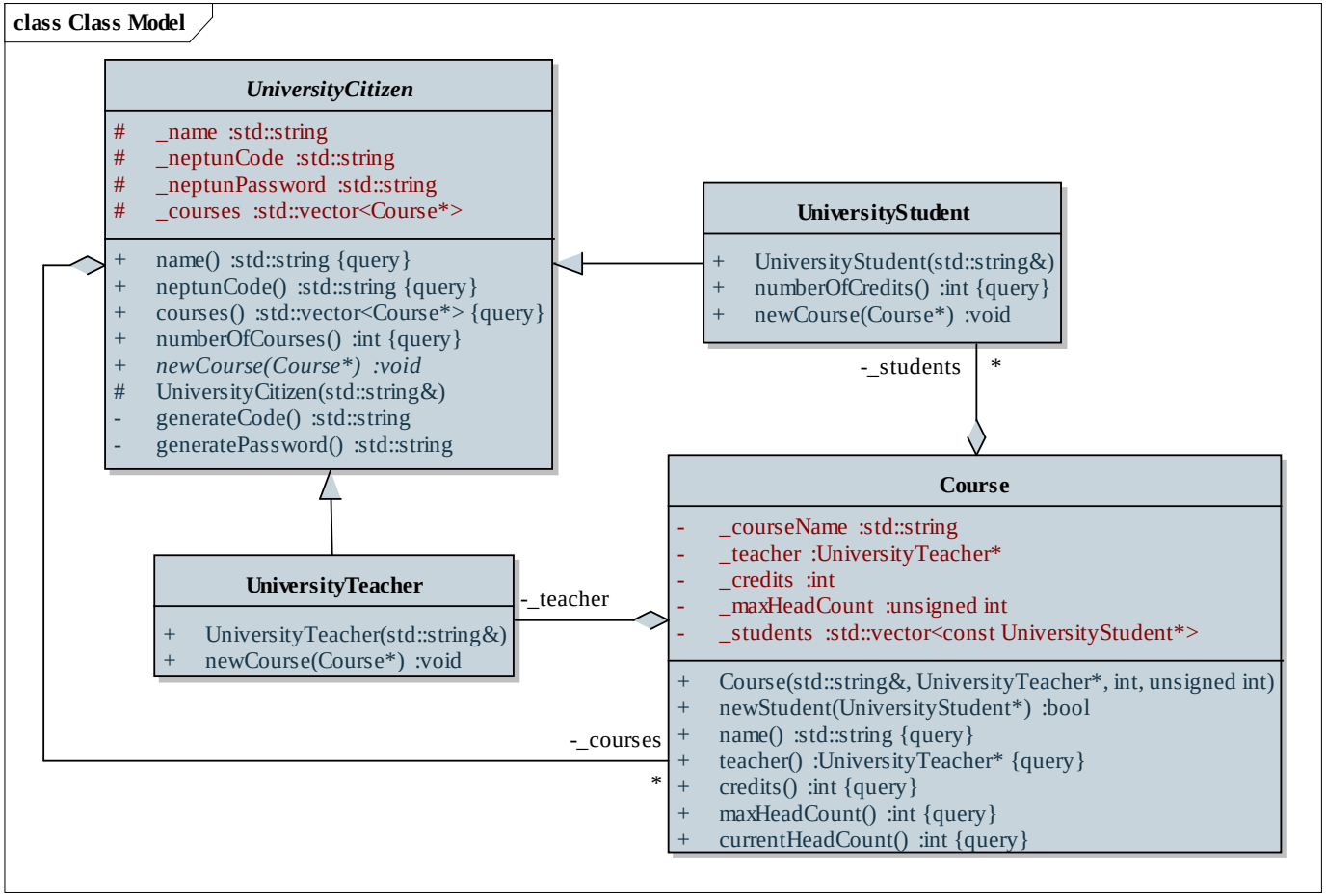
*Feladat*: Javítsuk az egyetemi modellünket öröklődés és polimorfizmus segítségével. A program írja ki az összes egyetemi polgár nevét és Neptun kódját.

- létrehozzuk az egyetemi polgár (**`UniversityCitizen`**) absztrakt ősosztályt, amely tartalmazza a közös részeket a hallgató és oktató osztályokból, amelyek ennek leszármazottai
- a leszármazottak definiálják a **`newCourse(…)`** műveletet külön-külön, ezért az ősben tisztán virtuális lesz
- a főprogramban egy közös tömbben tároljuk a polgárokat, polimorfizmust használva

# Polimorfizmus

## Példa

*Tervezés:*



# Polimorfizmus

## Példa

*Megoldás (*`universitystudent.hpp`*):*

```
…
class UniversityStudent : public UniversityCitizen
{
   // hallgató osztálya, speciális egyetemi polgár
public:
    UniversityStudent(const std::string& n);

    int numberOfCredits() const;

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

# Polimorfizmus

## 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;
}
…
```