

Bevezetés a Programozásba II

6. előadás

Objektumok kezelése és kapcsolatai

© 2014.03.17. Giachetta Roberto
groberto@inf.elte.hu
<http://people.inf.elte.hu/groberto>

Objektumok kezelése és kapcsolatai

Objektumok közötti kapcsolatok

- *Objektum-orientáltak* nevezzük azt a programot, amely egymással kommunikáló objektumok összessége alkot
 - az objektumoknak típusa van, amely meghatározza viselkedési mintáját, ez az osztály
 - a programban számos objektum kaphat helyet, amelyek kapcsolatait relációkon keresztül adjuk meg
 - a legegyszerűbb kapcsolat az *egyszerű kommunikáció (asszociáció)*



Objektumok kezelése és kapcsolatai

Objektumok közötti kapcsolatok

- Az asszociáció speciális esetei:
 - *tartalmazás (kompozíció)*: az osztály egy példányát a másik osztály tartalmazza egy, vagy több mezőjében
 - az objektum élettartama megegyezik a tartalmazó objektum élettartamával, ezért mindig elérhetővé válik
 - Pl. egy racionális szám tartalmazza számlálóját, nevezőjét, egy téglalap tartalmazza a méreteit, vagy tartalmazhatja koordinátáit



Objektumok kezelése és kapcsolatai

Objektumok közötti kapcsolatok

- *hivatkozás (aggregáció)*: az osztály egy, vagy több példánya meg van hivatkozva a másik osztály egy, vagy több mezőjében
 - pl. egy egyetemi hallgatóhoz tartozik egy kurzus, a kurzushoz pedig egy oktató, ám ezek egymástól függetlenül is kezelhetők, változtathatók
 - a tényleges objektum nincs benne, ezért az élettartama nem függ tőle, így nem garantált, hogy elérhető



Objektumok kezelése és kapcsolatai

Példa

Feladat: A téglalap rajzoló alkalmazásban elég könnyen be tudjuk tölteni a képernyőt téglalapokkal, ezért legyen lehetőség eltolni a téglalapokat egy megadott vektor segítségével.

- a vektort középső egérgomb kattintással rögzítjük a képernyő közepéhez képest történő elmozdulás függvényében
- a vektort új osztályként (**Vector**) felvesszük az alkalmazásba, a vektor a két irányú eltáolást (**_deltaX**, **_deltaY**) tárolja, lehetőséget adunk vektor kirajzolására (<<), valamint vektorok összeadására (+)
- a vektorral való eltolás igazából a téglalap bal felső sarkának eltolása, így célszerűen ezt is kiemeljük, és külön osztályba (**Point**) helyezzük

Objektumok kezelése és kapcsolatai

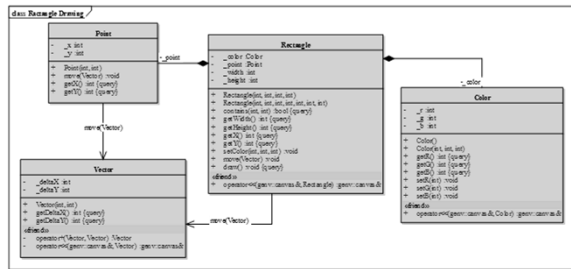
Példa

- így a pont és a téglalap (**Rectangle**) megkapja az eltolás műveletét (**move**)
- a főprogramban egy harmadik ágban kezeljük az eltolást, amely egy új, megfelelő méretű vektort vesz fel, és azzal eltolja az összes téglalapot
- mivel az eltolások így összegződnek, felvesszük egy összeg vektort is (**sumVector**), hogy ki tudjuk rajzolni az összesített eltolást
- az eltolás miatt egy fekete hátteret is kell rajzolnunk
- a pont, és a téglalap színe (**Color**) a téglalap mezői lesznek, azaz kompozíciós kapcsolat köti őket össze, míg a vektorral egyszerű asszociáció

Objektumok kezelése és kapcsolatai

Példa

Tervezés:



Objektumok kezelése és kapcsolatai

Példa

Megoldás (vector.hpp):

```

class Vector { // vektor osztály
private:
    int _deltaX;
    int _deltaY;
public:
    Vector(int dx = 0, int dy = 0) { ... }
    ...
    friend Vector operator+(Vector v, Vector w);
    // két vektor összege
    friend genv::canvas& operator<<(genv::canvas&
        c, Vector v); // vektor kirajzolása
};
    
```

Objektumok kezelése és kapcsolatai

Példa

Megoldás (main.cpp):

```

...
case btn_middle: // középső kattintásra
    Vector v(ev.pos_x - 250, ev.pos_y - 250);
    // létrehozuk az elmozdulás vektorát

    sumVector = sumVector + v;
    // hozzáadjuk az eddigi eltoláshoz

    for (int i = 0; i < recs.size(); i++)
        recs[i].move(v);
    // elmozgatjuk az összes téglalapot
    break;
...
    
```

Objektumok kezelése és kapcsolatai

Objektumok élettartama

- Az objektumaink állapottal és *élettartammal* rendelkeznek
 - az élettartam adja meg, mikor vehetjük igénybe az objektumot, és annak műveleteit
 - az objektum élettartama alatt a memóriában helyezkedik el
 - az élettartamot szeretnénk minél jobban testre szabni, hatékonysági (memóriatakarékos) és kezelési (elérhetőség) szempontok miatt
- A procedurális programozásban a változóink kötött élettartammal rendelkeztek
 - a *globális változók* a program teljes futása alatt léteztek
 - a *lokális változók* csak egy adott programblokkon belül

Objektumok kezelése és kapcsolatai

Memóriahely foglalás

- Az objektumok élettartamát alapvetően befolyásolja, milyen módon foglaljuk le számukra a memóriaterületet
- Memóriahelyeket kétféleképpen foglalhatunk le:
 - automatikusan*: változó létrehozásakor lefoglalódik hozzá egy memóriahely is, létrehozását és törlését nem befolyásolhatjuk
 - ez történik lokális és globális változók létrehozásakor
 - pl.:


```

{
    double d; // itt létrejön a változó
    ...
} // itt megsemmisül
                    
```

Objektumok kezelése és kapcsolatai

Memóriahely foglalás

- manuálisan (dinamikusan)*: lehetőségünk van explicit megadni a kódban, hogy lefoglalunk egy a memóriahelyet
 - ehhez a **new** operátort használjuk, és meg kell adnunk a típusát is, pl. **new double**;
 - a létrehozás visszaad egy memóriacímét, amelyet a helyfoglalás megkapott (illetve annak az első bájttját)
 - az így létrehozott változót nem közvetlenül használjuk, hanem a memóriacímén keresztül, amelyet külön változóba helyezünk, ez a *mutató (pointer)*, pl.:



```
double* d = new double;
```
 - a lefoglalt memóriaterületet bármikor törölhetjük a **delete** operátorral, függetlenül a programblokk végétől, pl.: **delete d**;

Objektumok kezelése és kapcsolatai	
A mutatók	
<ul style="list-style-type: none"> A <i>mutató</i> tehát egy speciális változó típus, amely memóriacímét tárol értékként létrehozásával egy új adatot viszünk a memóriába, amely másik adat memóriacímét tartalmazza lényegében a <i>referencia</i> (♣) általánosítása a mutatókat * jelöli a létrehozáskor, egy már létező változóra, vagy egy újonnan, manuálisan lefoglalt memóriaterületre állíthatjuk rá később ismét a * segítségével érhetjük a mutató mögötti adatot, pl.: <code>*d = 1.0;</code> önmagában kevés területet foglal (32, vagy 64 bit) 	
PPKE ITK, Bevezetés a programozásba II	6:13

Objektumok kezelése és kapcsolatai	
Dinamikus memóriafoglalás	
<ul style="list-style-type: none"> Pl.: <pre>double *d; // létrejön a mutató, a tényleges valós érték // még nem { d = new double; // létrejön az érték is *d = 1.0; // értéket adunk a változónak } // a változó itt nem semmisül meg cout << *d << endl; *d = 2.0; // ugyanúgy elérhetjük és felhasználhatjuk az // értéket delete d; // manuálisan megsemmisítjük az értéket</pre> 	
PPKE ITK, Bevezetés a programozásba II	6:14

Objektumok kezelése és kapcsolatai	
Dinamikus memóriafoglalás	
<ul style="list-style-type: none"> Minden <code>new</code> operátornak kell rendelkeznie egy <code>delete</code> párral, azaz a dinamikusan létrehozott változókat törölni is kell a nem törölt változók használatuk után is foglalják a memóriát, bár már nincs mutató rájuk állítva, az ilyen területeket nevezzük <i>memóriaszemétnek</i>, pl.: <pre>double *d = new double; // dinamikusan lefoglaltuk a memóriaterületet d = new double; // ekkor az előző terület memóriaszemét lesz</pre> sosem a mutatót, csak a dinamikusan lefoglalt területet töröljük, így ha több mutató hivatkozik ugyanarra a területre, elég egyszer elvégeznünk a törlést 	
PPKE ITK, Bevezetés a programozásba II	6:15

Objektumok kezelése és kapcsolatai	
Dinamikus memóriafoglalás	
<ul style="list-style-type: none"> Bármilyen osztályt kezelhetünk dinamikusan: <pre><típusnév> *<mutatónév> = new <típusnév>; delete <mutatónév>;</pre> <ul style="list-style-type: none"> amennyiben a saját típusunk konstruktorparaméterekkel rendelkezik, azokat meg kell adnunk a létrehozáskor: <pre><mutatónév> = new <típusnév>(<paraméterek>);</pre> Továbbra is lehetőségünk van hivatkozni a típusunk adataira (<code>*<mutatónév>.<tagnév></code> formában) <ul style="list-style-type: none"> a zárójel az operátor precedencia miatt kell mivel ez elég összetett jelölés, lehet egyszerűsíteni a <code>-></code> operátorral: <code><mutatónév>-><tagnév></code> 	
PPKE ITK, Bevezetés a programozásba II	6:16

Objektumok kezelése és kapcsolatai	
Aggregációk megvalósítása	
<ul style="list-style-type: none"> Amennyiben aggregációs kapcsolatot valósítunk meg a modellben (és nem kompozíciót), az adatokat csak hivatkozáson (mutatón, vagy referencián) keresztül kezelhetjük, így <ul style="list-style-type: none"> nem készül belőle másolat nem semmisül meg a tartalmazó objektum megsemmisülésével Pl.:  <pre>classDiagram class MyType class MyOtherType { + MyType myType } MyType o-- MyOtherType : myType</pre> 	
PPKE ITK, Bevezetés a programozásba II	6:17

Objektumok kezelése és kapcsolatai	
Aggregációk megvalósítása	
<ul style="list-style-type: none"> Pl.: <pre>class MyType { // egyik osztály ... }; ... class MyOtherType{ // másik osztály MyType* myType; // csak egy mutató tárolunk el }; ... MyType* t = new MyType; MyOtherType o; o.myType = t; // nem másoljuk le az objektumot, csak // hivatkozást állítunk rá</pre> 	
PPKE ITK, Bevezetés a programozásba II	6:18

Objektumok kezelése és kapcsolatai

Példa

Feladat: Módosítsuk a téglalap rajzoló alkalmazást úgy, hogy minden színt egyszerre cseréljük, ne csak egy adott téglalapot.

- egy ciklussal végigmehetünk az összes téglalapon, és egyenként kicserélhetjük, vagy
- elég egy színt változtatnunk, amelynek csak a hivatkozását (mutatóját) adjuk át a téglalapoknak, így ha egy értéket megváltoztatunk, az összes téglalap ezt látja
- ehhez dinamikusan létre kell hoznunk egy színt a főprogramban (`color`), amelyet a bezárás előtt törölünk is
- nem csak a téglalapoknak, de a háttérnek is van színe, így azt is létre kell hoznunk (`black`), és fel kell használnunk

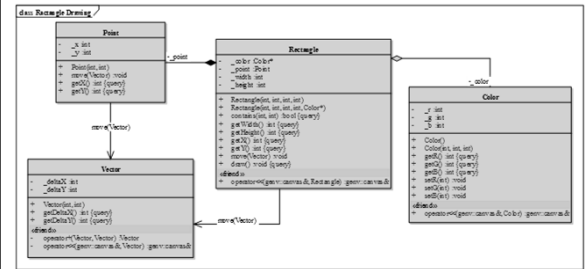
PPKE ITK, Bevezetés a programozásba II

6:19

Objektumok kezelése és kapcsolatai

Példa

Tervezés:



PPKE ITK, Bevezetés a programozásba II

6:20

Objektumok kezelése és kapcsolatai

Példa

Megoldás (main.cpp):

```
Color* color = new Color(255, 0, 0);
...
case btn_left:
    recs.push_back(Rectangle(ev.pos_x - 50,
        ev.pos_y - 50, 100, 100, color));
    // a színnek csak a hivatkozását adjuk át
    break;
case btn_right:
    color->setR(rand() % 256);
    color->setG(rand() % 256);
    color->setB(rand() % 256);
    // csak egy színt kell beállítanunk
...

```

PPKE ITK, Bevezetés a programozásba II

6:21

Objektumok kezelése és kapcsolatai

Példa

Feladat: Módosítsuk a téglalap rajzoló alkalmazást úgy, hogy alaphoz három különböző színben (piros, fehér, zöld) jelenjenek meg a téglalapok, és egy adott téglalapra történő kattintás az összes ugyanolyan színűt cserélje le.

- egy vektorban (`colors`) eltároljuk a három lehetséges színt (mutatók segítségével), amelyek közül egy újonnan létrehozott téglalapnak mindig a megfelelőt adjuk át
- jobb kattintásra ismét megkeressük a megfelelő téglalapot, majd a hozzá tartozó színt módosítjuk, így az összes téglalap át fog színeződni
- a főprogram végén a vektor valamennyi elemét töröljük (a háttérszín mellett)

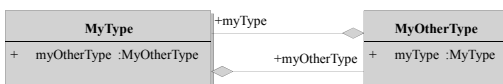
PPKE ITK, Bevezetés a programozásba II

6:22

Objektumok kezelése és kapcsolatai

Körbehivatkozások

- Az osztályok közötti kapcsolatok könnyen azt eredményezhetik, hogy két osztály kölcsönösen hivatkozik egymásra, pl.:



- Ez a megvalósításban azt jelentené, hogy mindeket osztálynak ismernie kell a másikat, az `#include` direktíva segítségével, ez azonban *körbehivatkozást* eredményezne, ami nem engedélyezett

PPKE ITK, Bevezetés a programozásba II

6:23

Objektumok kezelése és kapcsolatai

Osztálydeklarációk

- A körbehivatkozások feloldásáért osztálydeklarációkat kell alkalmaznunk, ekkor csak az osztály nevét adjuk meg előre a fejlécfájlban

```
pl.:
// mytype.hpp:
class MyOtherType; // osztály deklarációja
class MyType { // egyik osztály
...
    MyOtherType* otherType;
    // már felhasználható az osztály
};

```

- ekkor nem használhatóak a tagjai, csak a típus meglétét biztosítja a deklaráció

PPKE ITK, Bevezetés a programozásba II

6:24

Objektumok kezelése és kapcsolatai

Osztálydeklarációk

- ha szükséges, a forrásfájlban megadható a teljes osztály az `#include` direktívával, így kikerülve a körbehivatkozást
- pl.:

```
// mytype.cpp:  
#include "myothertype.hpp"  
// itt már definiálhatjuk az osztályt  
...  
  
// myothertype.hpp:  
#include "myothertype.hpp"  
// itt meghagyhatjuk a direktívát  
  
class MyOtherType { ... };
```

PPKE ITK, Bevezetés a programozásba II

6:25

Objektumok kezelése és kapcsolatai

Példa

Feladat: Valósítsunk meg egyetemi hallgatókat, oktatókat, valamint kurzusokat modellező osztályszerkezetet.

- a hallgató (`UniversityStudent`) és az oktató (`UniversityTeacher`) rendelkezik névvel, Neptun kóddal és jelszóval, valamint kurzusokkal, továbbá lekérdezhető a kurzusszáma
- a kurzusok (`Course`) rendelkezik névvel, oktatóval, hallgatókkal, kreditszámmal és maximális létszámmal
- a kurzus létrehozásakor megkapja az oktatót, amely szintén felveszi azt saját kurzusai közé, míg a hallgató felveheti a kurzust, amennyiben van szabad hely (ekkor a kurzus megjelenik a hallgatónál, és a hallgató is a kurzusnál)

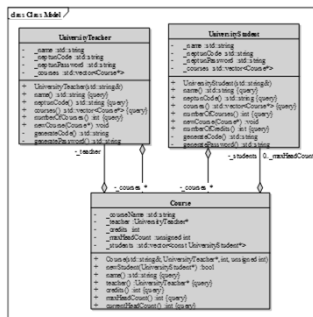
PPKE ITK, Bevezetés a programozásba II

6:26

Objektumok kezelése és kapcsolatai

Példa

Tervezés:



PPKE ITK, Bevezetés a programozásba II

6:27

Objektumok kezelése és kapcsolatai

Példa

Megoldás (universityteacher.hpp):

```
#include <vector>
#include <string>

class Course;

// osztályok deklarálása a körbehivatkozás
// elkerülése végett

class UniversityTeacher // oktató osztály
{
    ...
    std::vector<Course*> _courses;
    // mutatókat tárolunk
    ...
};
```

PPKE ITK, Bevezetés a programozásba II

6:28

Objektumok kezelése és kapcsolatai

Példa

Megoldás (universitystudent.cpp):

```
...
void UniversityStudent::newCourse(Course* c) {
    if (c->newStudent(this))
        // ha a kurzus felveszi a hallgatót
        _courses.push_back(c);
        // csak akkor veheti fel a hallgató a
        // kurzust
}
...
```

PPKE ITK, Bevezetés a programozásba II

6:29

Objektumok kezelése és kapcsolatai

Memóriaterületek

- A programok a használat szempontjából három területet különböztetnek meg:
 - *globális terület (global):* konstansok és globális változók, amelyek a program futása során mindig jelen vannak
 - *verem (stack):* a lokális változók, amelyeket automatikusan hoztunk létre
 - működésében olyan, mint egy verem, mert mindig az utolsó blokkban létrehozott változó törlődik elsőként a blokk végétével
 - *kupac (heap):* a manuálisan lefoglalható memóriaterület, általában a legnagyobb részét képezi a szegmensnek
 - a tömbök és szövegek is ide kerülnek

PPKE ITK, Bevezetés a programozásba II

6:30

Objektumok kezelése és kapcsolatai

Memóriaterületek

- Az objektumorientált programok adataikat jórészt a *heap* memóriaterületen tárolják
 - így jobban személyre szabható az objektumok élettartama
 - C++-ban erre a mutatók és a dinamikus memóriaterület foglалás ad lehetőséget, más nyelvekben is hasonló megoldást alkalmaznak (a háttér minden esetben ugyanaz)
 - egyes programozási nyelvek (pl. Java) eleve csak a heapen tárolt objektumok használatát engedélyezik
 - amely objektumra elfogytak a hivatkozások (mutatók), törölhető, hiszen a későbbiekben úgyse használható, ennek automatikus megvalósítása a *szemétyűjtés* (pl. Java)