A Programozás Alapjai 2
Objektumorientált szoftverfejlesztés

Dr. Forstner Bertalan
forstner.bertalan@aut.bme.hu

BME AUT
Department of Automation and Applied Informatics

C++11 hozta: nullptr

- A NULL makró volt és nem típus. A típusa *int* volt.

```
void func(int n);
void func(char *s);
//*****************************************//
func( NULL ); // Hoppá, valójában az 1. függvényt hívja!
```

- *nullptr*: egy pointer literál, aminek a típusa nullptr_t
  - Ahogy pl. a true és a false is litrálok, aminek a típusa bool.
  - Ezek egyben foglalt kulcsszavak is.

A C++ mint OOP nyelv

Mi az objektum-orientáltság?

- Szemléletmód, paradigma
- Nem csak a programozás (implementáció) során jelenik meg
  - Analízis, tervezés
- Hogyan ragadjuk meg a valóság, a probléma lényegét

3 fontos kritérium

- Egységbezárás (encapsulation)
  - Ami logikailag egy helyre tartozik, legyen is egy helyen: attribútumok és a rajtuk dolgozó műveletek

3 fontos kritérium

- Egységbezárás (encapsulation)
  - Ami logikailag egy helyre tartozik, lebyen is egy helyen: attribútumok és a rajtuk dolgozó műveletek
- Adatelrejtés (data hiding)
  - Csak az interfészen keresztül lehet kommunikálni az objektummal

3 fontos kritérium

- Egységbezárás (encapsulation)
  - Ami logikailag egy helyre tartozik, lebyen is egy helyen: attribútumok és a rajtuk dolgozó műveletek
- Adatelrejtés (data hiding)
  - Csak az interfészen keresztül lehet kommunikálni az objektummal
- Általánosítás/specializáció (specialization)
  - Megragadása az öröklés segítségével

A programozás fejlődése

- gépi kód
- assembly
- C, Pascal
- C++
  - kevert: OO és procedurális programozás eszközei is (tudunk globális függvényt)
- Java, C#, …
  - ezek nem csak nyelvek, hanem teljes platformok saját futtató környezettel, tisztán OO
- Szoftver komponensek
  - Egy komponens binárisan újrafelhasználható (akár több programozási nyelven is), szigorúan csak az interfészen keresztül érhetők el a szolgáltatásai, több interfésze is lehet, önállóan telepíthető.
- SOA: szolgáltatás orientált szemléletmód

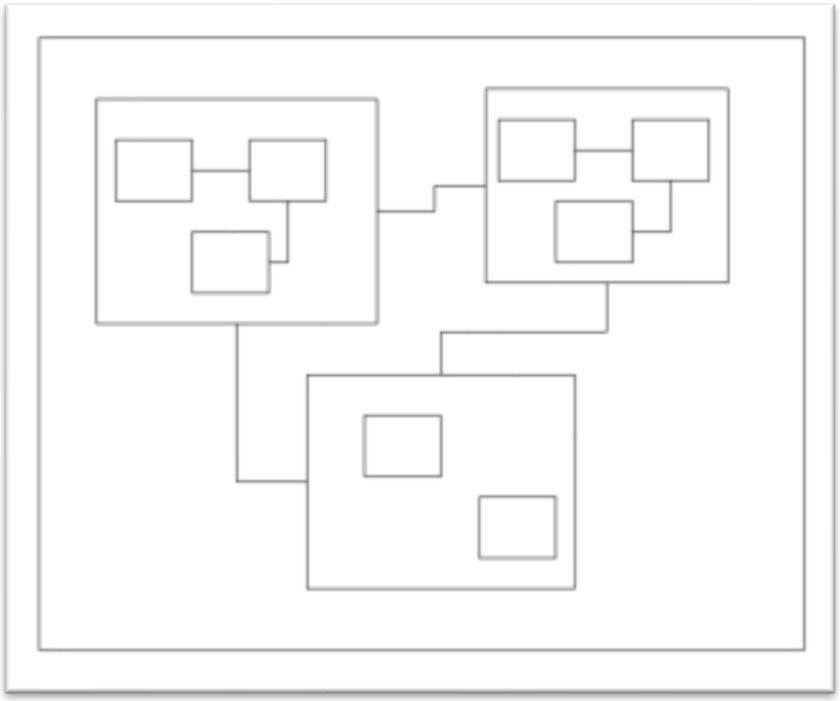
A szoftverfejlesztés ma…

- Nagy méret, komplex feladatok
- Több embernek kell együtt dolgozni
  - > hogyan tudjuk felosztani a feladatot, hogy többen tudjanak dolgozni rajta, a részek összeálljanak, valamint mindenki csak a saját részéhez férjen hozzá (a többihez csak jól definiált módon)
- meglevő függvény/osztálykönyvtárakhoz, API-khoz kell illeszkedni
- továbbfejleszthetőség: ne kelljen 20 helyen belenyúlni
- Újrafelhasználhatóság
  - > költségcsökkentés (más kódját is, pl. leszedem a Web-ről, GitHubról, ne kelljen megérteni, hogyan is működik)
- A valós életben nehéz kitalálni, mit is akar a felhasználó

Komplex rendszerek tulajdonságai

- Hierarchikus felépítésűek. A komplex rendszert modellezni kell:
  - > Dekompozícióval egymással együttműködő részekre bontjuk.
  - > A részeken belül szorosabb a kapcsolat, mint a részek között.
  - > Az egyes részek tovább bonthatók...
  - > A részek függenek a feladattól, a tervező tapasztalatától, stb.
- Alkalmazzuk az absztrakció eszközét is
  - > a számomra fontos részeket kiemelem, a többit elhanyagolom, nem foglalkozok vele, nem veszem be a modellbe.
- Csak a szemlélőtől függ, hogy milyen absztrakciós szinten vizsgálja a rendszert
  - > Nem tudjuk egyben átlátni az egészet, nem is kell. Egy adott absztrakciós szinten lássuk át!

Komplex rendszerek tulajdonságai



- Csak a szemlélőtől függ, hogy milyen absztrakciós szinten vizsgálja a rendszert
  - > Nem tudjuk egyben átlátni az egészet, nem is kell. Egy adott absztrakciós szinten lássuk át!

Kétfajta hierarchia:

- Része, „part of”

Kétfajta hierarchia:

- Része, „part of”

Kétfajta hierarchia:

- Része, „part of”

Kétfajta hierarchia:

- Része, „part of”



- „kind of”, „-féle” hierarchia

Kétfajta hierarchia:

- Része, „pa

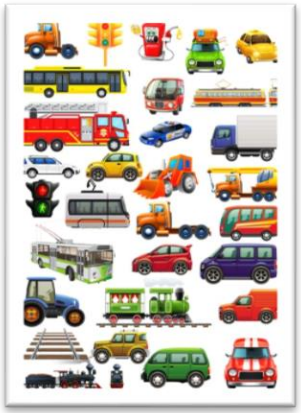
- „kind of”,

Kétfajta hier

- Része, „part of“



- „kind of”, „-féle

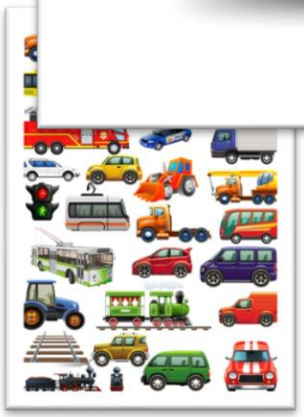



Kétfajta hierarchia:

- Része, „part of” hierarchia



- „ki

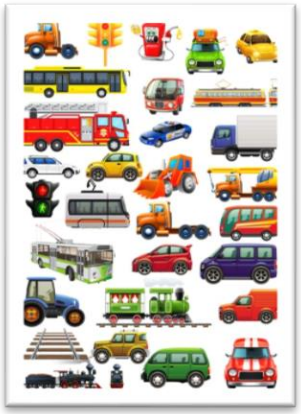



Kétfajta hierarchia:

- Része, „part of” hierarchia



- „kind of”, „-féle” hierarchia



A kettőt ne keverjük…

Az objektum

- Alap építőkő
  - Egy adott személy, könyv, adott tömb az elemeivel
  - Vagy akár fogalom, pl. piros szín
- Egy konkrét entitás
  - Különbözik a többitől
  - Pl. az én autómentő autóm, egyedi rendszámmal, stb.

Az objektum

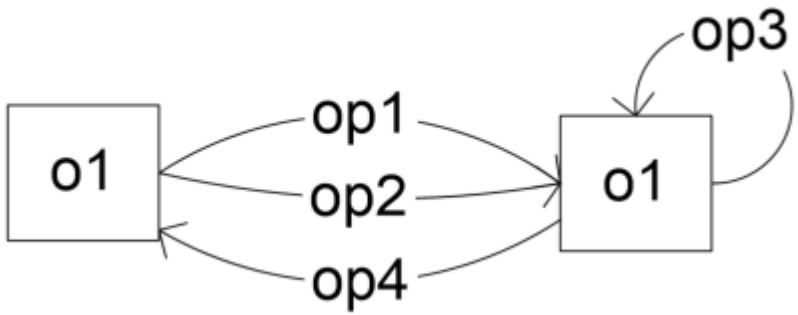
- Vannak tulajdonságai (attribútumai)
  - Pl. bicikli attribútumai?
  - Az attribútumok értéke egy adott pillanatban: az objektum *állapota*

Az objektum

- Vannak tulajdonságai (attribútumai)…
- …és rajta értelmezett műveletek.
  - Pl. bicikli műveletei?

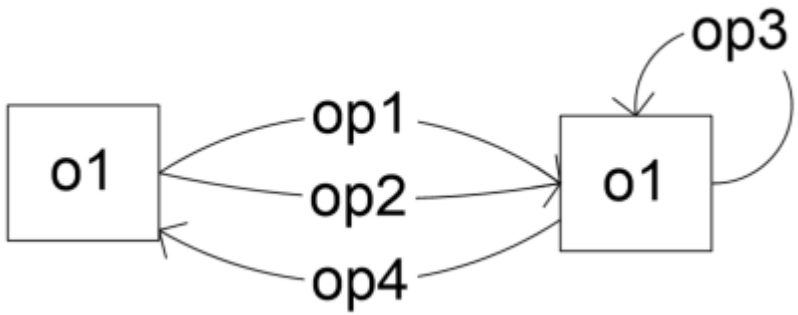
Az objektum

- Vannak tulajdonságai (attribútumai)…
- …és rajta értelmezett műveletek.
- Az objektum fekete doboz: a rajta értelmezett műveletekkel kommunikál a külvilággal



Az objektum

- Vannak tulajdonságai (attribútumai)…
- …és rajta értelmezett műveletek.
- Az objektum fekete doboz: a rajta értelmezett műveletekkel kommunikál a külvilággal
  - Pl.: Stack esetén az interfész?



Előnyök

- Nem kell ismerni a belső szerkezetet a használathoz
- Az objektum konzisztens állapotban marad
- Összeköti az adatot a rajta végzendő művelettel
- Közelíti az emberi gondolkodásmódot

Hogyan fog működni?

- A rendszert objektumokra bontjuk
- Ezek együttműködnek
  - Felhasználják egymás szolgáltatásait
- Az objektum
  - Tartalmazhat másik objektumot
  - Vagy pointert (referenciát) rá
  - → így tud üzenetet küldeni neki

Az osztály

Az osztály: „nem beépített típus”

- Meghatározza:
  - Az adatszerkezetet (mem-ben hogyan)
  - Értelmezett műveleteket (nem kell minden objektumra!)
  - Hogy kell a műveleteket végrehajtani
  - Hogyan kell létrehozni, megszűntetni…
- Futáskor az objektumnak hely foglalódik a memóriában (mint egy struktúrának),
- az egyedisége pedig az ő egyedi címéből adódik.

Az osztály: „nem beépített típus”

- Meghatározza:
  - Az adatszerkezetet (mem-ben hogyan)
  - Értelmezett műveleteket (nem kell minden objektumra!)
  - Hogy kell a műveleteket végrehajtani
  - Hogyan kell létrehozni, megszűntetni…
- Futáskor az objektumnak hely foglalódik a memóriában (mint egy struktúrának)
- az egyedisége pedig az adódik.

```
int a, b;
Szemely szalacsiSandor;
Szemely karacsonTamas
Stack stack1;
Stack stack2;
```

1. Egységbezárás

- Példa: pontok és körök

1. Egységbezárás példa

1. kísérlet. Újdonságok: class, függvény, illetve tagváltozók

```
class Point {
public:
        int x;
        int y;
        void draw() {
                printf("Point here: %d, %d\n", x, y);
        }
        int getX() { return x; }
        void setX(int ax) { x = ax; }
};
```

Tagváltozó

- Mint a struktúránál
- Szinonima: attribútum.
  - A Point osztály attribútumai: x és y
- Minden *objektumnak* (példánynak) külön hely foglalódik a memóriában.
  - Az osztály meghatározza, hogy az objektumainak milyen az adatszerkezete a memóriában.
  - Vigyázat, ne bitvadászkodjunk, extra dolgok is vannak az attribútumokon kívül

Tagfüggvény

- Szinonima: metódus, művelet.
  - A Point osztályon, illetve annak objektumain a draw, a getX és a setX műveletek értelmezettek.
  - A p1.draw() a p1 objektumra meghívja a draw() műveletet, amire az kirajzolja magát.
- A tagfüggvények:
  - az adott osztály objektumainak állapotát állítják be (setX)
  - ... kérdezik le (getX)
  - egyéb műveletet végeznek (draw)
- A tagfüggvény kódja az egész osztályra egyszer tárolódik, nem objektumonként.

Tagfüggvény szintaktika

- Kétféle lehet:

1. ***inline***, mint a példában (tényleg inline!)
2. Külön definiálva
   - .h-ba az osztály definíció (és tagfüggvény deklaráció)
   - .cpp fájlba a tagfüggvények definíciója
     - **A scope operátor ::**

- Példa átalakítás a másodikra

A this pointer

- A tagfüggvényen belül elérhető az aktuális objektumra mutató pointer
  - A this kulcsszóval érhető el
- A *this a metóduson belül magát az objektumot jelenti, amire az adott metódust meghívták
  - Rajta keresztül saját tagváltozót, tagfüggvényt érünk el

A this pointer

- Mint egy közönséges globális függvény, aminek van egy rejtett 0. paramétere, a this
  - > valójában ez történik a színfalak mögött

```
void  draw(Circle* const this) {
        printf("circle here: x %d, y %d\n", this->x, this->y);
}
```

- Praktikus, ha pl. ütközés van nevekben (azonos nevű függvényparaméter és tagváltozó)

2. Adatrejtés

- Probléma:

```
Point p1;
p1.x = -50;
p1.y = 22340;
p1.draw();
```

- Inkonzisztensé tettük.
- Oka: közvetlenül hozzáférünk az objektum belső állapotához
- Megoldás: *definiáljuk a műveletek egy adott csoportját és csak azokon keresztül lehessen hozzáférni az állapotához* (megváltoztatás): **interfész**!

Láthatóság szabályozása nyelvi szinten

- 3 kulcsszó:
- *public*:
  - elérhető kívülről is
  - adott osztály tagfüggvényei,
  - más osztályok tagfüggvényei
  - globális függvények

Láthatóság szabályozása nyelvi szinten

- 3 kulcsszó:
- *public*:
  - elérhető kívülről is
  - adott osztály tagfüggvényei,
  - más osztályok tagfüggvényei
  - globális függvények
- *private*:
  - csak az adott osztály tagfüggvényein belül érhető el
  - (más osztályok tagfüggvényeiből és globális függvényekből nem)

Láthatóság szabályozása nyelvi szinten

- 3 kulcsszó:
- *public*:
  - elérhető kívülről is
  - adott osztály tagfüggvényei,
  - más osztályok tagfüggvényei
  - globális függvények
- *private*:
  - csak az adott osztály tagfüggvényein belül érhető el
  - (más osztályok tagfüggvényeiből és globális függvényekből nem)
- *protected*:
  - csak az adott és a közvetlen leszármazott osztály tagfüggvényein belül érhető el
  - (más osztályok tagfüggvényeiből és globális függvényekből nem)

A point osztály átdolgozása

- Példa

A point osztály átdolgozása

- Példa

```
const int MAXCOORD = 1000;

class Point {
private:
        int x;
        int y;
public:
        void draw();
        int getX();
        void setX(int ax)
        {
            if(ax>=0 && ax<=MAXCOORD)
                x = ax;
        }
};
```

A point osztály átdolgozása: init

- Példa

```
void init()
{
        x = 0;
        y = 0;
}
```

```
Point p1;
p1.init();
p1.setX(50);
```

Adatrejtés best practice

- Első lépésben tervezzük meg az interfészt. Csak ezek legyenek láthatók kívülről: public.
  - init(), setX(), draw(), …
- A többi, ami a belső működéshez kell: private v. protected.

- Előny #1: konzisztens állapot megtartása
- Előny #2: bonyolult osztály felhasználójának csak az interfészt kell ismernie.
  - Pl. Stacknél: push és pop. Egyszerű ahhoz képest, mintha látnánk az egész implementációt.
- Előny #3: az interfész mögött megváltoztathatjuk az implementációt (hatékonyabbra, stb.).
  - Az adott osztály felhasználója ebből semmit nem érez, nem kell a kódjához hozzányúlni.

- A lényeg: az osztály, objektum felhasználását a rendszer megvalósításában egyszerűbbé teszi).
- A class és a struct közötti egyetlen különbség az alapértelmezett láthatóságban van (ha nem írok semmit): struct-nál public, class-nál private.

A konstruktor

- Probléma: init-et elfelejtjük hívni
  - Mégis memóriaszemét lesz az objektumban? ☹

A konstruktor

- Probléma: init-et elfelejtjük hívni
  - Mégis memóriaszemét lesz az objektumban? ☹
- Megoldás: konstruktor
  - Az objektum létrehozása után automatikusan meghívódik, feladata az új objektum inicializálása (tagváltozók beállítása).

A konstruktor

- A konstruktor egy függvény, ami az objektum létrehozása (adatterület lefoglalása) után hívódik meg.
- Neve az osztály neve.
- Nem lehet visszatérési értéke (void sem!), az maga az új objektum lesz.
- Több is lehet, túlterhelhető.

A Példa átalakítása

Konstruktor típusok

- Default
  - Ha nem írok egyet sem, létrejön egy default, ami nem csinál semmit.
  - Ha írok legalább egyet, akkor nem jön létre a default.
  - Használata
    - Point p1;

Konstruktor típusok

- Egyéb, többargumentumú
- Egyargumentumú vagy konverziós
  - Használata
    - Point p1(50);
    - Point p2=60;

Konstruktor típusok

- Másoló konstruktor
  - Ez egy speciális egyargumentumú.
  - Feladata az objektum inicializálása egy másik, **ugyanolyan osztálybeli** (pl. másik Point) objektum alapján
  - Ha nem írok, létre jön egy, ami bitről-bitre másol
  - (Fontos, ld. jövő órán)

```
Point(const Point& other) {
        x=other.x; //látom a privátot,
        y=other.y; //hisz ez is Point
    }
```

Destruktor

- Olyan függvény, ami az objektum megszűnése előtt hívódik meg.
- Takarításra szoktuk használni (pl. dinamikusan lefoglalt memória felszabadítása)
- Ha nem írunk, ebből is létrejön egy default, ami nem csinál semmit
- A neve: ~osztálynév (Példa)
- Nem lehet paramétere se visszatérési értéke (void sem!)
- (Ha leszármaztatunk az osztályból, legyen virtual.)

Értékadás és inicializálás

- A következő kettő nem ekvivalens:

```
int x;
x=2;
```

- Illetve

```
int x=2;
```

- Mikor történik a helyfoglalás, illetve inicializálás, és mivel? Hány lépés van?

Összefoglalás

- Az osztály (class) bevezetése
  - Egységbezárást, adatrejtést közelebbről megnéztük
- Tagváltozók, tagfüggvények
  - A láthatóság módosítása (public, private)
- Konstruktorok
- Destruktor