A Programozás Alapjai 2
Objektumorientált szoftverfejlesztés

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

BME AUT
Department of Automation and Applied Informatics

ZH

Mert az jó

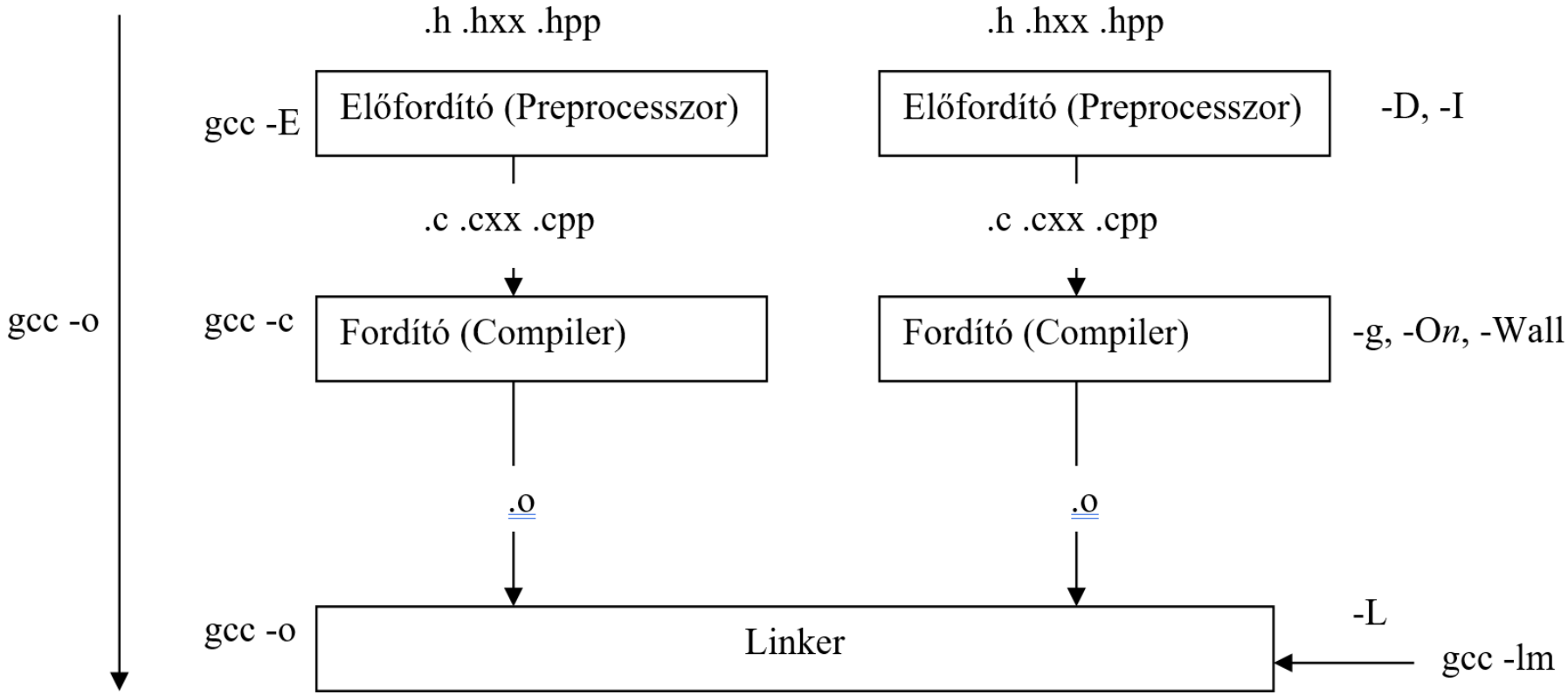
ZH felkészülés

Mert az még jobb

Gyors ismétlés: a fordítás folyamata

- A **preprocesszor** behelyettesítést végez
  - pl. #define, #include, stb.
  - DE: tokenizál
    - a „*#define a A*” nem fog minden *char*t kicserélni *chA*rra.
- a **fordító** (compiler) végzi az “oroszlánrészt”,
- a **linker** összefűzi a különböző fordítókimeneteket egy állományba.

Gyors ismétlés: a fordítás folyamata



Példa

- A függvényhívás folyamata és paraméter-átadás

A verem a függvényben:

Elmentett további regiszterek (Ide mutat a függvényen belüli ESP)
Lokális változók
Régi EBP (ide mutat a régi ESP)
Régi EIP
c Függvényparaméterek b a

Tanulságok

- Paraméterek jobbról balra a stack-be másolódnak. **Kivétel: tömb!**
- A lokális változók a vermen foglalódnak le, és a függvényből való kilépés után eltűnnek
- Érték szerint átadott paraméterek a függvényben megváltoztva sem változnak kívül
- A hívó takarít, mert lehetnek változó paraméterek is
- A verem elfogyhat: pl.: rekurzió

A linker feladata: címfeloldás

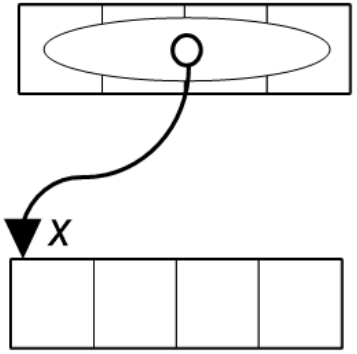
- Az object fájl még tartalmazhat ismeretlen hivatkozást
  - Ld. pl. prototípus + megvalósítás C függvényeknél
- Extern: változó vagy függvény előtt
- Példa

Referencia

```
int x=10;
int* p=&x;
int& r=x;
```

- r egy int referencia
- Alias rá, ugyanazt jelenti, mint x
- **Nem keverendő** a veszem-a-címét operátorral!

r jelenti annak a rekesznek a tartalmát, ahova ez a pointer mutat



- **Memóriakép** szempontjából *r* egy pointer lesz az *x* változóra

Visszatérés referenciával

- Példa
- Ökölszabály:

Sose szolgáltassuk ki lokális változó címét a függvényen kívülre!

(Sem pointer, sem referencia által).

Nézd át a C++ nem objektumorientált újdonságait! Mire jók a referenciák? Mikor használjuk? Hogy kapnak kezdőértéket?
Mi van, ha tagváltozók? (Inic. Lista)

A C++ mint egy jobb C nyelv

Függvénynév túlterhelése

- Mi azonosít egy függvényt C++ -ban?

  A neve és az argumentumlistája!

  (Visszatérési érték nem!)

Mi az a függvénynév túlterhelés? Mire jó?
Tudj példát mondani
Hogy lehetne kikerülni?

Makrók és inline függvények

- Gyakran nagyon rövid kódrészeket is külön függvénybe teszünk (pl. max):
  - olvashatóság, átláthatóság
  - Módosíthatóság
- A függvényhívásnak megvan a maga költsége, lassítja a kódot. Pl.:

```
int max(int a, int b)
{
    return a>b ? a:b;
}
…
max(x, y);
…
```

Mi történik híváskor?

1. visszatérési cím a stack-re
2. paramétereknek hely a stack-en
3. ugrás a címre
4. lokális változóknak hely a stack-en
5. ***törzs végrehajtás***
6. visszatérés érték a stack-re
7. lokális változók „felszabadítása”
8. ugrás vissza
9. paraméterek és visszatérési érték „felszabadítása”

Megj: a foglalás és felszabadítás: SP és BP növelés és csökkentés.

C-s makrók: problémák

- Szövegszerű behelyettesítés
- Nincs kontextusa, nem végez hibaellenőrzést

```
printf("%s\n", MAX("GYULA", "BELA"));
```

- Ha hiba van a makróban, annyiszor jelez a compiler hibát, ahány helyen használtuk

Inline függvények

```
inline int max(int a, int b) { … }
```

- Írjuk a definíciónál a függvény neve elé az inline kulcsszót.
  - > (amikor deklarálom, nem kell az inline, de azzal is lefordul)
- Bemásolódik a függvény törzse a hívás helyére, emiatt gyorsabb
- A makrókkal szemben lokális környezete van a hívásnak és szintaktikai ellenőrzés is. Teljesen biztonságos.
- Ahányszor használom, annyiszor másolódik be a függvény törzse: nő a kód mérete.

Inline függvények

- Akkor van értelme használni, ha:
  - a függvénytörzs végrehajtási ideje összemérhető a függvényhívás karbantartási műveletek idejével. t(1..4, 6..9)~t(5)
  - egy-két soros függvények esetén

Inline függvények

- Az inline csak egy javaslat a fordítónak, ő felül tudja bírálni. Kizáró okok is vannak:
  - rekurzió: önmagát hívja vagy két függvény hívja kölcsönösen egymást
  - használom a címét a függvénynek (függvény pointer)
  - címkét használok benne (goto)
  - Egyebek
- Tegyük a definíciót (törzset) is a header-be
  - Linker: unresolved external symbol

Alapértelmezett argumentumok

- Hátulról előrefelé haladva alapértelmezett értéket adhatunk meg
- Híváskor hátulról sorban elhagyhatjuk
  - Fordító automatikusan lenyomja helyettünk a stacken

```
void showWindow(int id, int x=320, int y=480, char* title="Hiba")
{
        printf("Uj ablak (%d) a %d,%d koordinatan:
                                '%s'", id,x,y,title);
}
```

Konstansok

- C++-ban:

const double BASE_YEAR_SALARY_MILLION = 6.0;

(C++11: *constexpr* ha fordítási időben rendelkezésre áll az érték)

- Típusos.
- Inicializálni kell
- Mi az értelme? Minél inkább megkötjük a programozó kezét, annál kevésbé fog (vagy fogunk mi) hibázni.

Konstans pointerek

- Külön törődést és gondolkodást igényel
- Példa:

```
char szo[] = { 'L', 'a', 'p', 'o', 's', '\0' };

const char* p1 = szo;
//*p1 = 'W'; //hiba!
p1++; //OK, 'a'-ra mutat

char* const p2 = szo;
*p2 = 'W'; // OK
//p2++; //Hiba!

const char* const p3 = szo;
//*p3 = 'W'; // Hiba
//p3++; //Hiba!
```

Konstans paraméterek

- Volt: nagyobb méretű változót referenciaként adjunk át függvénynek mert gyorsabb.
- Milyen problémákat vet ez fel?

- Figyelj rá: egy függvénynek konstanst adunk át (nincs aut. Konverzió nem-konstansra)

Automatikus konverzió

- Automatikus konverzió const-ról nem const-ra nincs
  - > ekkor nem lenne értelme a const-nak

- fordítva van konverzió

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

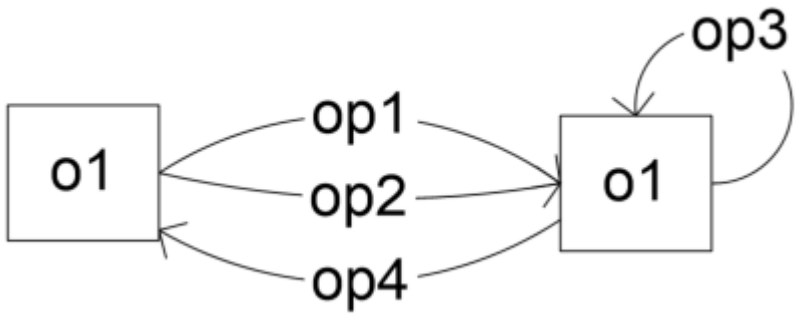
- Egységbezárás (encapsulation)
  - Ami logikailag egy helyre tartozik, lebyen is egy helyen: attribútumok és a rajtuk dolgozó műveletek
- Adatelrejtés (data hiding)
  - Csak az interfészen keresztül lehet kommunikálni az objektummal
- Általánosítás/specializáció (specialization)
  - Megragadása az öröklés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

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



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.

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 ::**

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, amre 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
- *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

```
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;
        }
};
```

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.

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: konziszte
- Előny #2: bonyolult
  - > Pl. Stacknél: push
- Előny #3: az interfé
  (hatékonyabbra, st
  - > Az adott osztály
- A lényeg: az osztály
  egyszerűbbé teszi).
- A class és a struct
  (ha nem írok semm

Mik az objektumorientált alapelvek? Hogyan vigyáz magára az objektum? Hogyan intézi az egységbe zárást? Mik a hozzá tartozó kulcsszavak?

A getter/setter függvények mit oldanek meg? Validálás?

A fenti alapelvekre ügyelj a tervezési feladatnál!
(Láthatóságok, statikus/konstans tagváltozók stb)

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.

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.
    - Gondold végig: nincs-e rá szükség, pl. tömböt hozunk létre
  - Használata
    - Point p1; //nem kell kiírni a zárójeleket

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
    }
```

Konstruktor típusok

- Másoló konstruktor
  - Ez egy speciális egyargumentumú.
  - Feladata az objektum inicializálása egy másik
    ugyan
    alapj
  - Ha ne
  - (Font

```
Point(co
```

```
    }
```

Milyen konstruktorokat ismersz? Mire jók? Hogyan állítják be a tagváltozókat? Mikor milyen konstruktor hívódik meg? Pl. Complex c = 54.3; Complex c2 = c; stb.

Mikor történhet másoló konstruktor hívás? Mi történhet, ha nincs vagy rosszul van megírva?

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
- Nem lehet paramétere se visszatérési értéke (void sem!)

Destruktor

- Olyan függvény, ami az objektum megszűnése előtt hívódik meg.
- Takarításra szoktuk használni (pl. dinamikusan lefoglalt memória felszabadítása)
- Ha nem
  ami ner
- A neve:
- Nem lehet paramétere se visszatérési értéke (void sem!)

- Gondold végig: hol történik konstruktor hívás (pl. érték szerint átvett obj.)
- Dinamikus adattagok esetén másoló konstruktor, op=, destruktor feladatok

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?

Dinamikus tagváltozók

A malloc hátránya

- A *malloc* nem hívja meg a konstruktorokat, nem tudunk a szintaxis miatt paramétereket átadni
- Új nyelvi elem: *new*, *delete*

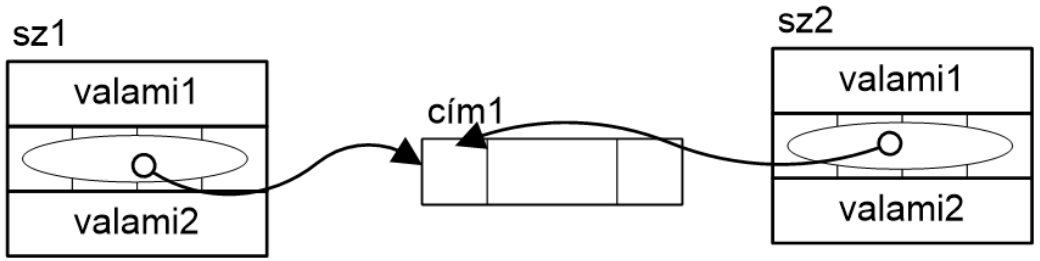
Hogyan kezelünk a heapen adatszerkezeteket? És ha tömböket hozunk létre? Hogy szabadítjuk fel? Milyen konstruktorok hívódnak meg? Referencia paraméterek és visszatérési értékek használata.

Másoló konstruktor

- Feladata az objektum inicializálása egy másik, ugyanolyan osztálybeli objektum alapján.
  - Ha nem írok, **létrejön egy alapértelmezett**, ami bitről bitre másol.
- Milyen esetekben kell megírni?
  - Ha ez a viselkedés nem felel meg nekünk. Például a legtipikusabb eset:
  - Amikor az adott **objektum** maga **felelős** a valamilyen hozzá tartozó dinamikusan allokált (malloc/free, new/delete) **memória lefoglalásáért** és **felszabadításáért**.

A Person szemléltetése memóriaképpel

- Mit kapunk helyette:



- Mostantól ketten felelősek ugyanazon memória terület menedzseléséért. Probléma pl. a következő:
  - > Amikor az sz1 (pisti) felszabadul (main-ből kilépéskor) meghívódik a destruktora, ami felszabadítja az objektumhoz tartozó, általa dinamikusan lefoglalt memória területet (név).
  - > Amikor az sz2 (iker) megszűnik, neki is meghívódik a destruktora, ami felszabadítaná a már felszabadított területet: durva futás idejű hiba!
  - > Ugyanígy hiba lenne: ha módosítjuk az egyik stringet, a másik is módosul.

Másoló konstruktor írás

- A **megoldás**: mivel a **default másoló nem megfelelő**, felül kell írni és a megfelelő viselkedést le kell programozni.

- Példa

```
Person(const Person& other)
{
  name = new char[strlen(other.name) + 1];
  strcpy_s(name, strlen(other.name) + 1, other.name);
  printf("Copy %s!\n", name);
}
```

Másoló konstruktor írás

- **Mikor hívódik meg** a másoló konstruktor? Amikor csak másolat készül.
  - **explicit**, mint a példában
  - függvénynek **paraméterként** átadva (nem referencia vagy pointer)
    - Példa: kick függvény
  - **visszatérési érték** függvényben
  - **bonyolult** kifejezésekben temporális változóként

Másoló konstruktor írás

- **Mikor hívódik meg** a másoló konstruktor?
  Amikor csak másolat készül.
  - **explicit**, mint a példában
  - függvénynek **paraméterként** átadva (nem referencia vagy pointer)
  - **v**
  - **b** nt

- Többfajta konstruktor vs. Alapértelmezett argumentumok
- Kétértelmű függvényhívás kérdése

Mivel sok esetben szükséges, mindig írjunk másoló konstruktort, ha az osztály objektumaihoz dinamikusan lefoglalt terület tartozik, aminek kezeléséért maga az osztály felelős.

Kompozíció vs. Aggregáció

- A birtokolja B-t: kompozíció. B-nek semmi értelme, létcélja nincs a rendszerben A nélkül.
  - Például: **Személynek** van **neve**.

- A “használja” B-t: aggregáció. B koncepcionálisan teljesen függetlenül létezik A-tól.
  - Például: **Személynek** van **apja**, aki egy másik Személy, de független.

Kompozíció vs. Aggregáció

- A birtokolja B-t: kompozíció. B-nek semmi értelme, létcélja nincs a rendszerben A nélkül.
  - Például: **Személynek** van **neve**.

- A "használja" B-t: aggregáció. B
  kon k A-
  tól.
  - P
    S

- Objektum kétféle megközelítésben is tartalmazhat pointert egy másik objektumra
- Mi a kettő közt a külöbség?
- Ki felel a megsemmísítésért? Destruktor…

Statikus és konstans tagok

Tagváltozók inicializálása

- Inicializációs lista
- Példa

- Az inicializálási lista **hamarabb lefut**, mint a **konstruktor törzse**.

Statikus tagváltozók :

- **Egy darab** van belőle az egész osztályra vonatkozóan.
  - Közös az osztály minden objektuma számára, (ugyanaz az értéke).
- Már **azelőtt is létezik**, hogy objektumot hoznánk létre az osztályból.
- Mikor szoktuk használni: amikor minden objektum számára közös változót szeretnénk.

Statikus tagfüggvény:

- Tipikusan statikus tagváltozókon dolgoznak.
- Olyan, mint egy globális függvény (nem kapja meg a this-t), csak éppen az osztályhoz tartozik.
- Statikus tagfüggvényen belül nincs is this pointer, ebből következik:
  - > Statikus tagfüggvényből nem statikus tagváltozó nem érhető el. Melyik objektumét is változtatná? **Pl. írja ki az EUR balance-ot.**
  - > Ugyanígy nem statikus tagfüggvény sem hívható (melyik objektumra hívná!). **Pl. hívja meg a balance kiíró függvényt.**
  - > Nem statikus tagfüggvényből statikus tagváltozó elérhető: a közös értéket jelenti.
  - > Ugyanígy statikus tagfüggvény is hívható.

Statikus változó inicializálása

- Mindig kell, az előző példa is csak így teljes
- Itt történik meg a helyfoglalás a változó számára
  - > Ne a headerbe tegyük…

Mik azok a statikus tagváltozók? Mire jók? Mikor lehet azokat használni? Mondj 4-5 példát.
Ha van stat. Tagváltozó, van-e szükség stat. tagfüggvényre?
Ismerd a szabályokat (honnan hívható stb.): Józan paraszti ész! (van-e this pointer?)

Konstansok

- Volt: konstans paraméterek
- Konstans tagváltozó
  - Az osztályomnak van egy tagváltozója, amit nem szeretnék megváltoztatni
  - Valamikor kezdőértéket kell kapnia! Az objektum létrehozásakor
- Példa: accountId
- Statikus ÉS konstans tagváltozó

Konstansok

- Mit jelent, ha egy objektum konstans?
  - Hogy nem változhat meg, vagyis az állapotát nem változtathatjuk meg.
  - Vagyis a tagváltozóit nem írjuk át, még akkor sem, ha public.
- De ez nem elég: hívhatok rajta tagfüggvényt, ami ezt kijátszhatja.

Konstans tagfüggvény

- Jelezni kell, hogy ez a függvény nem fogja megváltoztatni az állapotot.

- Ez a **konstans tagfüggvény**. Olyan, mintha a 0. paraméter, a **this, konstans** lenne.

```
int getBalanceHUF() const {
    return balanceEUR * rate;
}
```

Konstans tagfüggvény

- Jelezni kell, hogy ez a függvény nem fogja megváltoztatni az állapotot.

- Ez a **konstans tagfüggvény**. Olyan, mintha a 0. paraméter, a

```
int getBalanc
    return bal
}
```

Mik a konstans változók? Miért szeretnénk őket függvény paraméternek használni? Mik a konstans tagváltozók, hogy kapnak értéket? Mire jók a konstans tagfüggvények? Mikor muszáj használni őket?

A láthatóság enyhítése

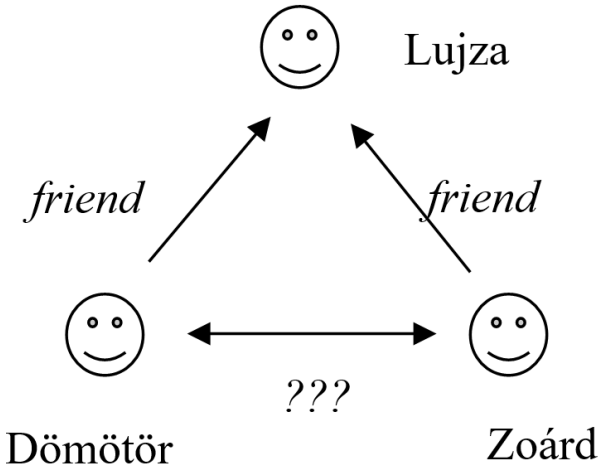
- Írhatunk olyan globális függvényt, amit egy adott osztály felhatalmaz arra, hogy a védett (private, protected) tagjait is elérje.
  - Így mindazokkal a lehetőségekkel bír, mint a tagfüggvény, de mégsem az.
- *Friend* kulcsszó
- Csak akkor használd, ha elkerülhetetlen!
  - A legtöbbször getter, setter függvények a jó megoldás
- Példa

Friend osztályok

- Egész osztályt hatalmazunk fel a hozzáférésre
- Példa

Friend osztályok

- Egész osztályt hatalmazunk fel a hozzáférésre
- Példa
- Vajon tranzitív?



Mi az a friend mechanizmus? Mikor lehet csak friend függvényekkel megoldani egy feladatot?

Névterek

- Miért állományszintű a hozzáférés szabályozása?
- Hogyan lehet több osztály ugyanolyan láthatóságú?
- Pl.: sort függvény létezik a standard kódkönyvtárban. A string osztály is létezik. Attól még én is írhatok. Melyiket használjuk?

Névterek

- Megoldás: Névterek.
- Függvények, osztályok, típusok (typedef), konstansok, globális változók definíciójának hierarchiába szervezését teszi lehetővé.

C++ IO, operátorok túlterhelése

Streamek

- C: stdin, stdout, stderr
- C++: cin, cout, cerr

- Lehetnek input és output streamek
- istream csak olvasható, ostream csak írható.
- << és >> operátorokkal lehet rájuk írni, illetve egymás után fűzhetőek ezek a műveletek.

4 bit:

- good – nincs hiba
- eof – file vége
- bad – adatvesztés történt
- fail – formátumhiba

4 bit:

- Beállítása (pl. cout streamen):
  - cout::clear(ios::failbit);
- Lekérdezése (pl. cin streamen):
  - cin.good()
  - cin.eof()
  - cin.bad()
  - cin.fail()
    - ez akkor is, ha a bad be van állítva.
    - Ilyenkor további írás-olvasás nem történik a clear-ig.

Formázás

```
printf("(%8.2f,%8.2f)\n", x, y);
```

- Itt a mezőszélesség 8, és két tizedesjegyet írat ki. Mi a megfelelője ennek C++-ban?
- Hogy néz ki C++-ban? Példa

```
cout << '(' << setprecision(2) << setiosflags(ios::fixed) << setw(8)
   << x << ',' << setw(8) << y << ')' << endl;
```

Manipulátorok

- A példában *setprecision*, *setiosflag*, *setw* és *endl* úgynevezett IO manipulátorok
- Van, amelyiknek paramétere van (*setprecision*), van, amelyiknek nincs (*endl*)
- Ezek az *iomanip* állományban találhatók.

Operátorok túlterhelése

Operátorok

- a+b, a=b, a==b, new, delete
  - > +, =, ==, new, delete, … mind operátorok

Operátorok

- a+b, a=b, a==b, new, delete
  - +, =, ==, new, delete, … mind operátorok
  - Úgy is nézhetjük, mint egy függvényhívás, csak más a szintaktika
- c=a+b;

Operátorok

- a+b, a=b, a==b, new, delete
  - +, =, ==, new, delete, … mind operátorok
  - Úgy is nézhetjük, mint egy függvényhívás, csak más a szintaktika
- c=a+b;
  - c=operator+(a,b);

Operátorok

- a+b, a=b, a==b, new, delete
  - +, =, ==, new, delete, … mind operátorok
  - Úgy is nézhetjük, mint egy függvényhívás, csak más a szintaktika
- c=a+b;
  - c=operator+(a,b);
  - operator=(c, operator+(a,b))

  //Persze ezek pl. int-re nem működnek így!

Operátorok túlterhelése

- A függvények túlterhelhetők: többet is írhatunk ugyanazzal a névvel, csak legyen más az argumentum lista (a visszatérési érték nem megkülönböztető).

Operátorok túlterhelése

- A függvények túlterhelhetők: többet is írhatunk ugyanazzal a névvel, csak legyen más az argumentum lista (a visszatérési érték nem megkülönböztető).
- Az operátorokat is túl tudom terhelni a következő feltételekkel:
  - jelölés (név) és argumentum lista alapján legyen egyértelmű
  - legalább az egyik argumentum nem beépített típus kell legyen (pl. két int összeadását nem tudom megváltoztatni)
  - Új operatátort nem lehet bevezetni: pl. **-ot.
  - Ha felül is definiálom: marad a precedencia szint és az asszociativitás.
  - Nem felüldefiniálható a . struktúramező elérés, a :: scope, a ?:

Szintaktika

- Hasonló a függvényekhez, csak a neve speciális.
- visszateresi_típus operator<opjel>(arg lista) {…}

Operátor tagfüggvények

- Az objektum tud magára vigyázni, és az adatot és a rajta végezhető műveletet egységbe szeretnénk zárni

==> Jó lenne, ha az operátor tagfüggvény lenne.

- Lehet is, ha csak lehet, így írjuk meg.
- Az ilyen tagfüggvények esetén az operátor 1. operandusa mindig az az objektum, amire meghívtuk.

Egyargumentumú operátorok

Kifejezés	Operátor (@)	Taggfügvény	Globális függvény
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A, int)

Kétargumentumú operátorok globális függvénnyel

Kifejezés	Operátor (@)	Taggfügvény	Globális függvény
a@b	+ - * / % ^ & | < > == != <= >= <<	A::operator@(B)	operator@(A, B)

További kétargumentumú operátorok

Kifejezés	Operátor (@)	Taggfügvény	Globális függvény
a@b	= += -= *= /= %= ^= &= |= <<= >>= [ ]	A::operator@(B)	-
a(b, c...)	()	A::operator()(B, C...)	-
a->b	->	A::operator->()	-

További kétargumentumú operátorok

Kifejezés	Operátor (@)	Taggfügvény	Globális függvény
a@b	= += -= *=	A::operator@(B)	-

Mire jó az operátorok túlterhelése? Jó ötlet lehet egy példát végigvinni. Például, bankszámlákat nyilvántartó osztály, ahol a pénzegyenleg dinamikus tagváltozó (int*). Hogy hozzuk létre (konstruktorok, destruktor), melyik konstruktor, operátort muszáj felüldefiniálnunk, hogyan? Szeretnénk összeadni a példányokat, definiáld felül a megfelelő operátorokat: pl. Account acc1(5399); Account acc2(3322); acc2 -= acc1; acc2 = acc1 + 4432; stb.

Hogyan érjük el, hogy streamre lehessen őket írni (pl. cout << acc1 //az eredmény: „3322 Ft”), illetve beolvasni ugyanilyen formátumban? Mi a 4 bit, amit ilyenkor be kell/lehet állítani?

Gondold végig, mikor KELL a globális változat. (Illetve a tagv. Változatnál kik az operandusok?)

Feladat

1. Adott a következő kódrészlet egy szerveralkalmazás szolgáltatásainak leírására:

```
class Service {
    char* serviceName;
  public:
    int port;
    Service(char* iServiceName, int iPort);
    … //további tagfüggvények stb.
};
```

A serviceName a kiszolgáló által nyújtott szolgáltatások nevét írja le, míg a port annak a portnak a számát, ahol a szerveren a szolgáltatás elérhető.

a) Hogyan kell megváltoztatni a fenti osztálydefiníciót, ha tudjuk, hogy minden szolgáltatás azonos porton keresztül lesz elérhető (amely néha változik)? Magyarázza el a felhasznált koncepciót: mire jó, mit módosít, hogyan.

b) Egészítse ki a fenti kódot az esetleg szükséges inicializálással együtt.

c) Megváltoztathatjuk-e a port tagváltozó értékét a konstruktorban a fenti kiegészítés után is (függetlenül attól, hogy lenne-e értelme)? Indokolja a válaszát.

Feladat2

- Készítsünk egy húsvéti nyuszi osztályt, ami húsvéti tojásokat tárol dinamikus adatszerkezetben. Minden csokinyusziba különböző súlyú tojásokat tesznek (nincs két egyforma súlyú tojás benne). A tojásoknak ne legyen getter függvénye.
- Terheljük túl a <<, =, + és == operátorokat a nyulakon!
- Két húsvéti nyuszi összeadása a listák összefűzését jelenti
- Két nyuszi akkor egyenlő, hogy ha ugyanazokat a súlyú tojásokat és ugyanabban a sorrendben tartalmazza
- Mutassunk rá, hogy a nyuszik egyenlőségének előfeltétele, hogy pl. a tojások egyenlőségét is vizsgálni tudjuk.

A teszt main:

```
Egg redEgg(9);
Egg glitterEgg(27);
Egg greenEgg(42);

EasterBunny bunny1;
EasterBunny bunny2;
bunny1.insert(redEgg);
bunny1.insert(greenEgg);
bunny2.insert(glitterEgg);
bunny2.insert(greenEgg);

cout << bunny1 << endl;
cout << bunny2 << endl;

EasterBunny bunny;
bunny = bunny1 + bunny2;

cout << bunny << endl;          // 9 42 27

if (bunny1 == bunny2)
        cout << "Twins." << endl;
```