A Programozás Alapjai 2
Objektumorientált szoftverfejlesztés

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

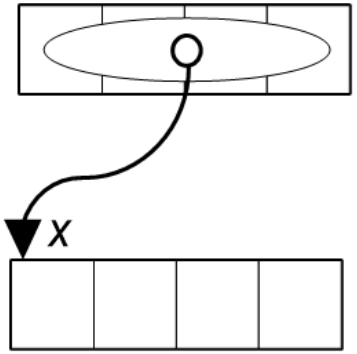
BME AUT
Department of Automation and Applied Informatics

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?

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

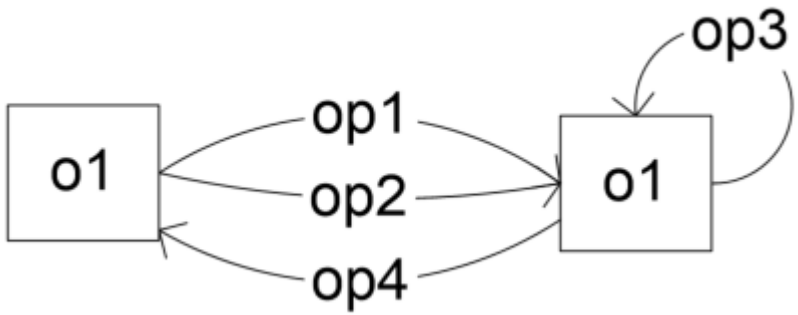
- Egységbezárás (encapsulation)
  - Ami logikailag egy helyre tartozik, lebyen is egy helyen: attribútumok és a rajtuk dolgozó műveletek
- Adatelrejtés (data hiding)
  - Csak az interfészen keresztül lehet kommunikálni az objektummal
- Általánosítás/specializáció (specialization)
  - Megragadása az öröklés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

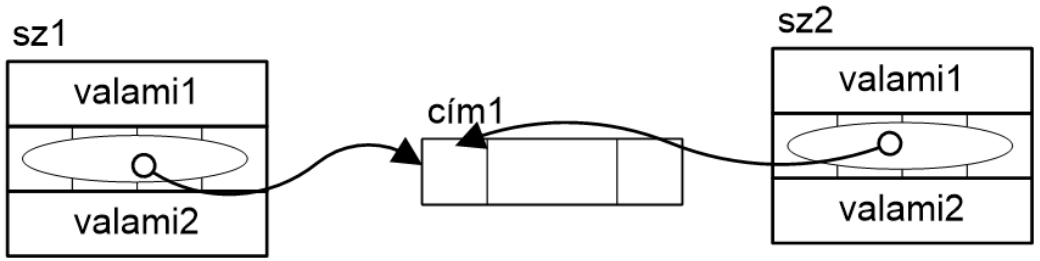
Hogyan kezelünk a heapen adatszerkezeteket? És ha tömböket hozunk létre? Hogy szabadítjuk fel? Milyen konstruktorok hívódnak meg? Referencia paraméterek és visszatérési értékek használata.

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

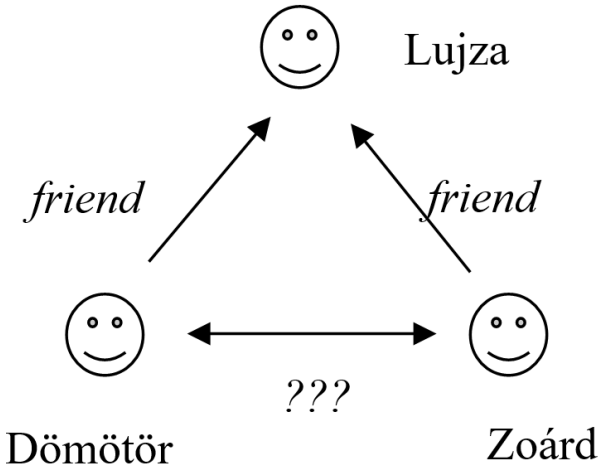
- Írhatunk olyan globális függvényt, amit egy adott osztály felhatalmaz arra, hogy a védett (private, protected) tagjait is elérje.
  - Így mindazokkal a lehetőségekkel bír, mint a tagfüggvény, de mégsem az.
- *Friend* kulcsszó
- Csak akkor használd, ha elkerülhetetlen!
  - A legtöbbször getter, setter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

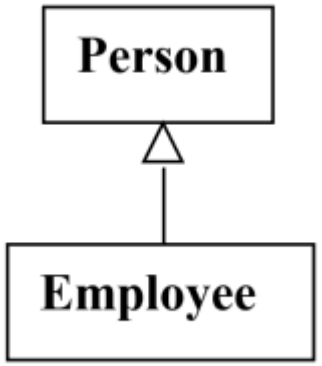
Mire jó az operátorok túlterhelése? Jó ötlet lehet egy példát végigvinni. Például, bankszámlákat nyilvántartó osztály, ahol a pénzegyenleg dinamikus tagváltozó (int*). Hogy hozzuk létre (konstruktorok, destruktor), melyik konstruktor, operátort muszáj felüldefiniálnunk, hogyan? Szeretnénk összeadni a 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?)

Öröklés

- Az OOP nyelvek harmadik fontos eleme
- Kétféle megközelítés:

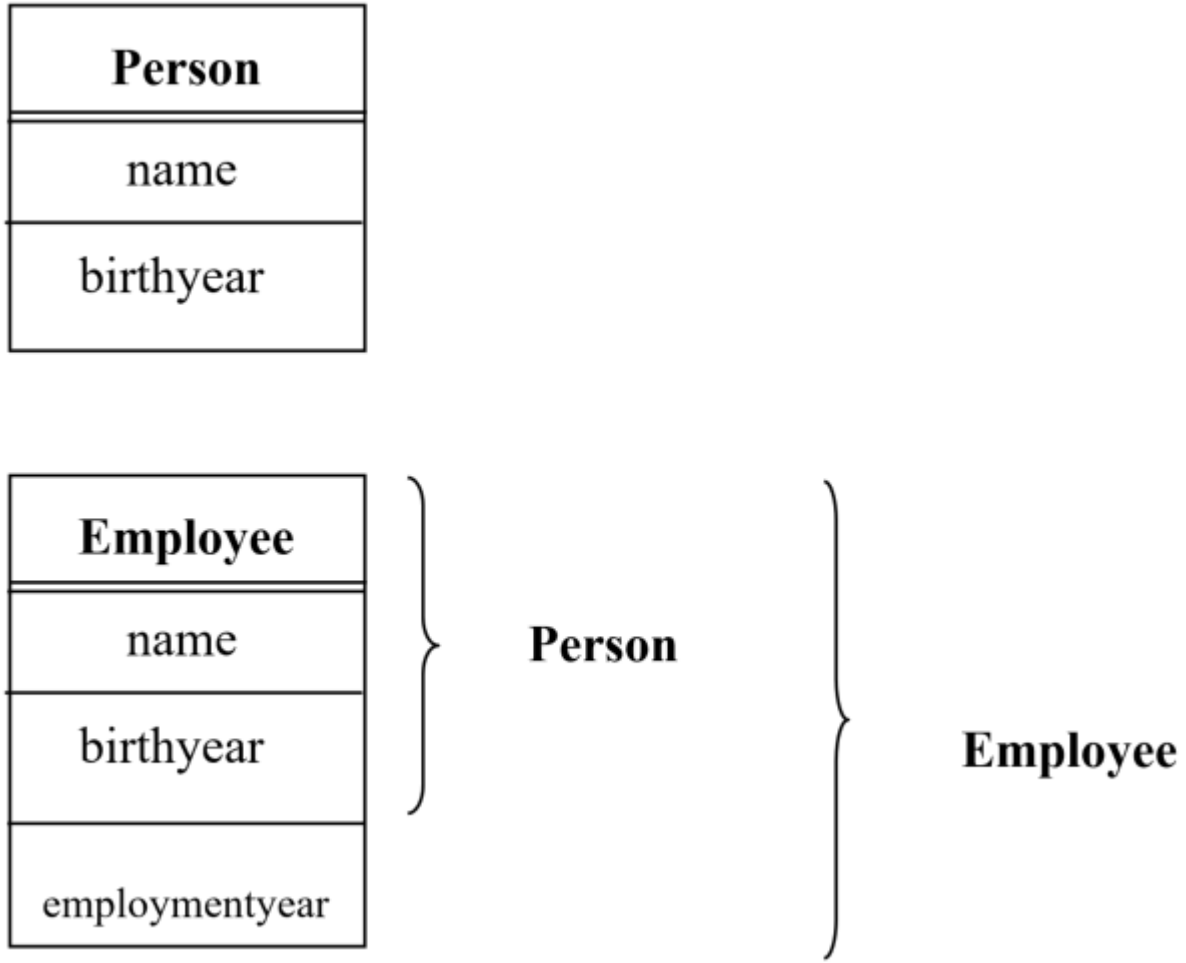


  - **Specializáció**: ha az Alkalmazott egyfajta személy, akkor „örökölnie” kell a Személy tulajdonságait.
  - **Általánosítás**: ha az alkalmazott egyfajta személy, akkor az Alkalmazott bárhol használható, ahol a Személy.
    - A nyíl az ábrán a behelyettesíthetőséget jelenti.

Tulajdonságok

- A leszármazott örökli a szülő **összes attribútumát** (tulajdonságait) és újakat adhat hozzá.
  - **Elvenni nem lehet**: felborulna az a fontos tulajdonság, hogy a dolgozó valójában egy személy.
- A leszármazott örökli a szülő **összes műveletét** (viselkedését)
  - újakat adhat hozzá, valamint a meglevőket felüldefiniálhatja (más implementációt adhat hozzá).
  - *A fejléc ugyanaz, csak a törzse más!*

Memória rajz



Behelyettesíthetőség

- Egy dolgozóra tekinthetünk úgy, mint egy személyre, ha nem vesszük figyelembe a csak a leszármazottra érvényes tulajdonságokat (ez teljesen logikus).

Előnyök

- kevesebbet kell írni
- normalizált megoldás, csak egy helyen kell módosítani a közös részeket
- ha felveszünk egy új dolgozót (pl. beszerző), akkor nem kell a meglevő osztályokat módosítani.
- az automatikus konverzió miatt: egységes kezelése különböző objektumoknak (majd később…)

Összefoglalva

- A leszármazott örökli a szülő:
  - attribútumait, tulajdonságait
  - interfészét: ugyanazokat az üzeneteket elfogadja, ugyanazok a műveletek végrehajthatók rajta.
    - Ezek a publikus műveletek.
    - Ebből adódik a behelyettesíthetőség: egy dolgozó egy személy is, így is tekinthetünk rá, és **minden**, ami fennáll a személyre, fennáll a dolgozóra is
- Implementációját
  - Az öröklés emiatt a kód-újrafelhasználás eszköze.

Láthatóság

- Ami **public**: nem védett, mindenki eléri
- Ami **private**: csak az adott osztály metódusaiból érhető el
  - A többi osztály és globálisak nem érik el.
- Ami **protected**: az adott osztályban és a közvetlen leszármazottban elérhető.
  - Ő is védett.

Az öröklött tagok láthatósága

- Tudjuk: az ősosztályban levő tagokat (attr. és metódus) a leszármazottak öröklik.
  - Mintha a leszármazottban is definiálnánk implicit módon.
- Mi lesz ezek láthatósága a leszármazottban?
- Háromféleképpen lehet leszármaztatni. Szintaktika:
  - class B: _____ A {…};
    - Mi kerül a vonalra?

Az öröklött tagok láthatósága

	Öröklés típusa		
Ősosztályban lévő láthatóság	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	nem látható	nem látható	nem látható

- Ha nincs kiírva, akkor private az öröklés.

A konstrukciós sorrend egy Leszarmazott létrehozásánál:

1. Területfoglalás az ősosztály, *Szulo* részeinek. Ha *Szulo* tartalmaz objektumokat, azok konstruktorainak hívása (*Seged* konstruktora)
2. *Szulo* konstruktorának hívása
3. Területfoglalás *Leszarmazott* részeinek. Ha *Leszarmazott* tartalmaz objektumokat, azok konstruktorainak hívása (*Tartalmazott* konstruktora)
4. *Leszarmazott* konstruktorának hívása

Problémák:

- Ha nem hívunk semmit a leszármazottban, az ősosztály defaultja hívódik meg
  - Hogyan hívhatunk egy leszármazottban az ősnek egy nem default konstruktorát?

Megoldás: inicializációs lista

```
class Leszarmazott : public Szulo {
   Tartalmazott adat;
 public:
   Leszarmazott(int szuloAdatParam, int tartalmazottAdatParam)
       : Szulo(szuloAdatParam), adat(tartalmazottAdatParam)
   {}
};
```

Szabályok az inic. listához

- Csak a közvetlen szülő konstruktorát lehet hívni
- Ha a szülőnek nincs def. konstruktora, akkor muszáj itt meghívni egy megfelelő konstruktort
- Az inic. Listában a sorrend tetszőleges
  - A tagváltozók a felsorolásuknak a sorrendjében jönnek létre

Behelyettesíthetőség, virtuális függvények

- **A C++-ban a változó típusában csak egy megfelelő típusú változó lehet.**

```
int a;
double b=3.14;
a=b;
```

- Ettől *a* nem lett *double* típusú, sem *b int* típusú.
- Csak meghívódott egy konverziós függvény a kettő között.

- Ez így van osztályok között is:

```
class A{…};
class B: public A{…};

B b;
A a=b;
```

- Ezután *b* marad *B* típusú, és *a* marad *A* típusú.
- A konstruktor működésétől függ az eredmény.

- A pointer esetén már két paramétertől függ

- Példa

```
double a = 0;
char*p = (char*)&a;
(*p)++;
cout << a; //4.94066e-324
```

TV és nem hozzá való távirányító…

TV és nem hozzá való távirányító…

- A fogantyú határozza meg a műveleteket
- A benne tárolt érték és a fogantyú teljesen különálló
- **polimorfizmus, polimorf mutató**

TV és nem hozzá való távirányító…

- A fogantyú határozza meg a műveleteket
- A benne tárolt érték és a fogantyú teljesen különálló
- **polimorfizmus, polimorf mutató**
- Ez így OK: **Az ősosztály típusú fogantyú mögött baj nélkül lehet leszármazott típusú objektum!**
  - > Persze ezen keresztül csak a leszármazottra érvényes részt érjük el.
- Ez a referenciára is igaz.

Virtuális függvények

- A virtuális azt jelenti, ha
  - > ősosztályban virtuálisnak definiálok egy függvényt,
  - > és a leszármazottban létezik ugyanolyan névvel, paraméterekkel,
  - > és a leszármazotton ezt a függvényt egy ősosztály típusú pointeren keresztül hívjuk meg,
  - > akkor a leszármazottbeli tag fog meghívódni.

Heterogén kollekció

- A példában a shapes tömb
- Ősosztály típusú pointereket tárol
- De ezek leszármazottakra mutatnak

Absztrakt osztály

- Tisztán virtuális függvény (pure virtual)

- **virtual double Area() = 0;**

Destruktor kérdése

- Melyik destruktor hívódik meg?
  - Miért, virtuális?
- Ha egy osztályból **várhatóan leszármazunk**, vagy **van virtuális függvénye**, akkor legyen a **destruktora is virtuális**.
  - (Hisz **ősosztály pointerén** keresztül hívjuk meg a **delete**-t.)
  - Ha nem ezt tesszük, akár le is állhat az alkalmazás, de mindenképpen számítani kell memóriaszivárgásra!

Virtuális függvénytábla

- A virtuális függvények megvalósításának alapja az indirekció.
- Az a függvény cím, amire meghíváskor ugrani kell, nem fordításkor dől el, hanem futás közben

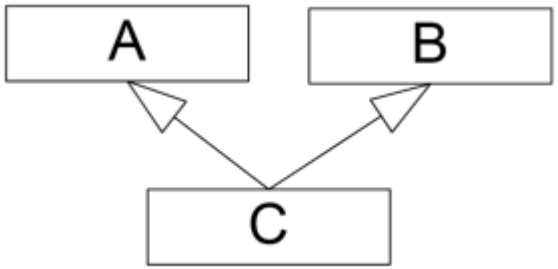
Virtuális függvénytábla

- Minden **osztály** rendelkezik a memóriában egy ugrótáblával az összes virtuális függvényére
  - akár benne definiált, akár örökölt függvényről van szó
  - Annyi címet tartalmaz ez a tábla, ahány virtuális függvénye van az adott osztálynak
- Minden **objektum** rendelkezik egy **vfptr** (vagy hasonló) nevű pointerrel az osztályának virtuális ugrótáblájára.
- (megvalósítás-függő)

Többszörös öröklés

Többszörös öröklés

- Egy adott osztálynak több ősosztálya van.
  - Több, mint kettő is lehet.
- Mindegyik tulajdonságait (attribútumok) és viselkedését (műveletek) örökli.



Két célja lehet

- Implementáció öröklése: Ritkán használják többszörös öröklésnél, ugyanis problémákat okozhat.
  - > Automatikusan rendelkezésre áll egyes függvények implementációja az őszosztályban
  - > Egy helyen kell csak módosítani változtatás esetén

Két célja lehet

- Implementáció öröklése: Ritkán használják többszörös öröklésnél, ugyanis problémákat okozhat.
  - > Automatikusan rendelkezésre áll egyes függvények implementációja az őszosztályban
  - > Egy helyen kell csak módosítani változtatás esetén
- Interfész öröklése: Az osztály műveletei az interfésze.
  - > Volt: Interfészt egy absztrakt osztállyal lehet készíteni.
  - > Ha több interfészt implementál egy osztály, mindből le kell származtatni.
  - > A leszármaztatás miatt behelyettesíthető mindnek a helyére.
  - > A különböző típusú objektumok egységesen kezelhetőek lesznek

Interfészörökés példa

```
class IWare {
public:
    virtual int getPrice() const = 0;
};

class IAccountable {
public:
    virtual int getVAT() const = 0;
};



void PrintPrice(const IWare& param)
{
    cout << "Price: " << param.getPrice();
}

void AccountingDoSomethingWithVAT(const IAccountable& param) {
    cout << "VAT = " << param.getVAT();
}
```

Elérés pointeren keresztül

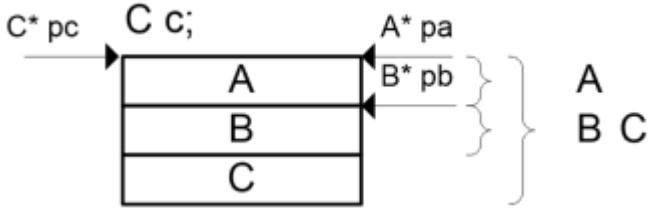
- Gond lehet az elérés az ősosztály pointerén keresztül
  - hiszen a pointer típusa határozza meg az elvégezhető műveletet

Elérés pointeren keresztül

- Szabály1: a pointeren keresztül ***csak az adott osztálybeli rész*** érhető el
  - ez a műveletekre is igaz

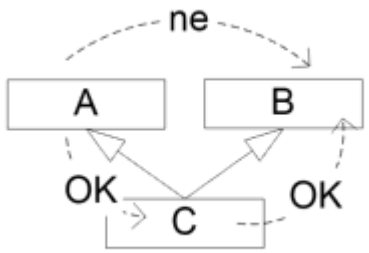
Elérés pointeren keresztül

- Szabály1: a pointeren keresztül ***csak az adott osztálybeli rész*** érhető el
  - ez a műveletekre is igaz
- Szabály2: a pointer mindig ***a pointer típusának megfelelő rész elejére*** mutat.
  - Emiatt a példában a B*-ra castolás megváltoztatja a pointer értékét, el is tolja.
  - Ezt a fordító megoldja, ilyen intelligens a cast.

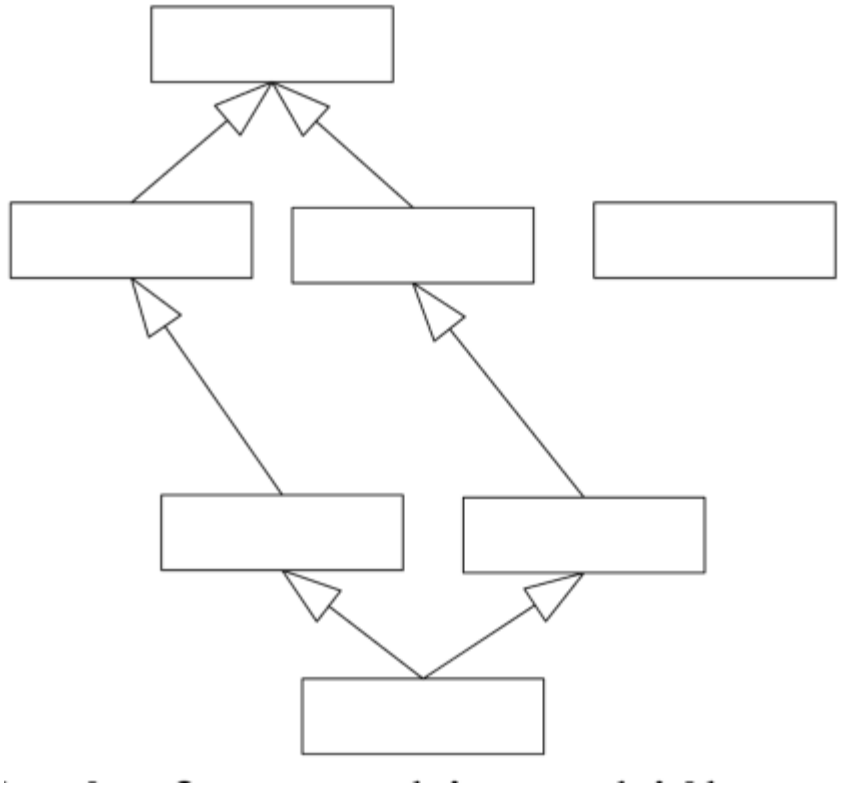


Keresztbe castolás

- Keresztbe ne castoljunk, mert akkor az eltologatás nem tud működni.
  - > A kereszbe cast előtt castoljunk vissza fel a szülőre.
  - > Keresztbe cast (pl. A*→B*) automatikusan nincs is (fordítói hiba), de explicit kiírva lefordul és hibás lesz.
- Példa

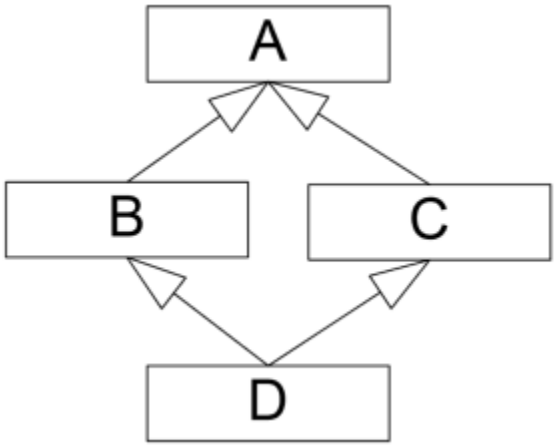


Újabb probléma

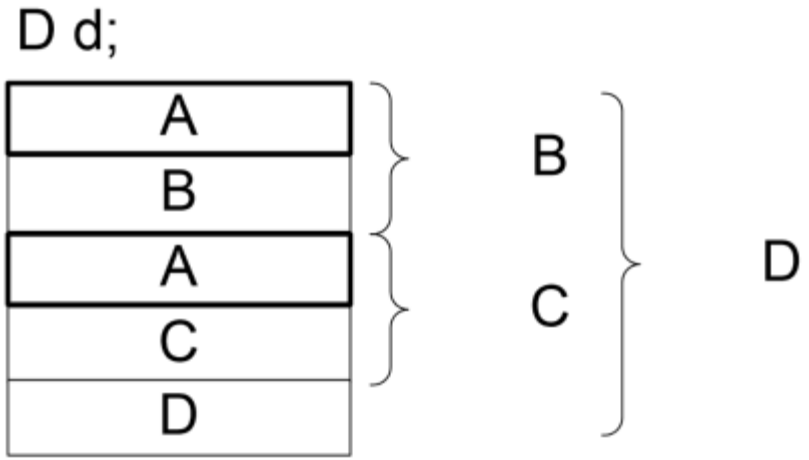


Gyémánt forma

- Példa



Memóriakép

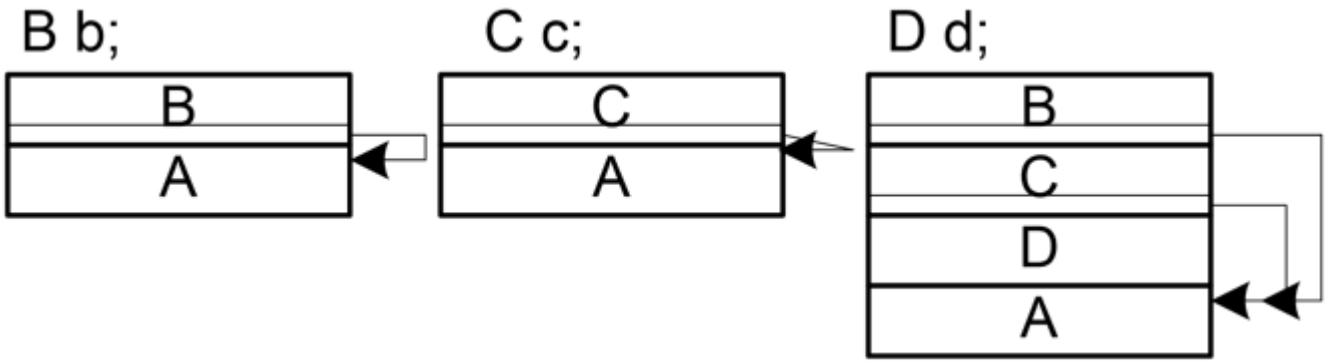


Virtuális öröklés

- A két úton is érkező ős csak egyszer szerepeljen
- Semmi köze a **virtuális függvényekhez**
  - Azok sokkal fontosabbak.
- Szintaktika: a az **öröklés leírása** helyére be kell írni a *virtual* kulcsszót.
- class B :public virtual A { …

Virtuális öröklés

- Egy lehetséges megvalósítás



Konstruktor hívási sorrend többszörös öröklésnél

- Virtuális ősosztályok konstruálása
- Nem-virtuális közvetlen ősosztályok konstruálása
- Saját részek konstruálása
  - Pointerek: virtuális ősosztály
  - Pointerek: Virtuális függvények táblája
  - Tagváltozók inicializálása
  - Konstruktortörzs lefutása

Destruktorok hívási sorrendje

- Ugyanez, csak vissza:

- Saját destruktortörzs meghívása
- Tagváltozók destruktora
- Közvetlen, nem-virtuális ősosztályok destruktora
- Virtuális ősosztályok destruktora

Példa (tipikus 4. feladat)

- Egy erdei kisvasutat üzemeltető cég különböző árkategóriájú jegyeket (*Ticket*) ad el.
- Jelenleg **három** típust értékesítenek: normál (*StandardTicket*), diák (*StudentTicket*) és nyugdíjas jegy (*SeniorTicket*).
- A jegyek ára a konkrét távolságtól függ, amelyet kilométerben megadva állítanak be egy-egy jegyhez.
- Egy kiadott jegyen utólag az érvényes távolságot nem lehet módosítani.
- A távolságdíj normál alapértéke (*basePrice*) kilométerenként 30 Ft., amely a programfutás során nem változik. A különböző kategóriájú jegyek esetén az ár az alapérték konstansszorosa, értéke rendre 1.0 (*normalFactor*), 0.5 (*studentFactor*) és 0.3 (*seniorFactor*).
- A jegyeket reprezentáló osztályoktól a konkrét ár lekérdezhető (*getPrice() fv.* adja vissza az árat).
- Szoftverünk az adott napon kiadott jegyeket egy kategóriától független közös tömbben tárolja. A célunk ez alapján a tömb alapján összegezni és kiírni a napi bevételt.

Példa (folyt)

- **Tervezze** meg és vázolja fel az osztályok öröklési hierarchiáját! Használja fel a fenti dőlt betűs osztály-, függvény- és változóneveket! Az osztályok téglalapjaiban tüntesse fel a kategóriával és árral kapcsolatos tagváltozók és/vagy tagfüggvények deklarációját és láthatóságát! Ügyeljen az elegáns OO megoldásokra!
- **Implementálja** az osztályokat és konstansokat, figyelve arra, hogy esetlegesen egyes konstansokat is tagként vagy statikus tagként érdemes implementálni. Ne legyen egy függvénytörzsben sem felesleges, nem használt kód! Egy új jegytípus esetleges felvételéhez ne kelljen a már meglévő osztályokat módosítani!
- **Írjon** egy egyszerű programrészletet (nem dinamikus) tömbbel, ami megmutatja a három különböző típusú jegy felvételét, valamint egy ciklusban kiszámolja, majd kiírja a tömbben tárolt jegyek árának összegét.

Generikus típusok

Generikus típus

- Generikus vagy paraméterezett típus
  - C++-ban template-nek hívják
- Típus, ami nem teljes, csak ha bizonyos paraméterek meg vannak adva használatkor
- Fogalmi szinten megérteni nehéz, ezért nézünk példákat bőven.

Függvény template-ek

- A függvény törzse ugyanígy néz ki minden típusra.
- Jó lenne, ha tudnánk írni egy olyan függvény öntőformát, amiben a típus, amin dolgozik egy paraméter, nincs rögzítve, hanem amikor használjuk, akkor lehetne megadni.
- Erre szolgál a (3) függvény template.
  - Példa

A példa

```
template <class T>
inline T max(T lhs, T rhs)
{
   return lhs > rhs ? lhs : rhs;
}

int main(int argc, char* argv[]) {
   int a = 5;
   int b = 6;
   cout << "The winner is " << max<int>(a, b) << endl;
   double c = 3.14159265359;
   double d = 4.28318530618;
   cout << "The winner is " << max<double>(c, d) << endl;

   //A fordito kitalalja a tipust:
   cout << "The winner is " << max(c, d) << endl;
}
```

Függvény template

- A ***class*** kulcsszó, lehet **typename**-et is, ~ekvivalens

  template <typename T>

- A T paraméter **nevet mi adtuk neki**
- Amikor használjuk, természetesen a template paraméterek egy adott típussal értéket kapnak
- **Példa** a használatra

Feltételek támasztása

- Feltételeket támasztunk a T paraméterre, amit a forráskód nem kényszerít ki!

Konverzió és a template-ek

- Template argumentumok közt nincs minimális konverzió sem
- A nézett példában a

```
cout << "The winner is " << max(a,d) << endl;
```

nem fordul le, mert konverzióra lenne szükség. (a: int, d: double)

- Ha nem változók állnának, akkor sem, ez egy szigorú szabály a template függvényekre.
- A hibaüzenet elég pontosan megmondja a baját (vagy int, vagy double legyen a T).

Konverzió és a template-ek

- Két megoldás:

```
cout << "The winner is " << max((double)a,d) << endl;
cout << "The winner is " << max<double>(a,d) << endl;
```

- Utóbbinál **explicit példányosítjuk** a template függvényt.
  - A max<double> már nem template, hanem egy teljesen közönséges, normál függvény
  - Rá már azok a szabályok érvényesek, melyek a közönséges függvényekre

Code bloat - kódburjánzás

- Bármilyen típussal használjuk, példányosodni fog, nagyon megnőhet a kód mérete.
  - Pl. int és a long is különböző, legenerálódik mindkettőre, hiába van automatikus konverzió.
- Megoldás lehet, ha kiírjuk, hogy melyiket használjuk:

```
max<int>(a, b);
```

  - Itt ha ***a*** és ***b*** long vagy char, akkor sem fog külön generálni neki függvényt, hanem az int-eset használja konverzióval.

Explicit specializáció

- A template függvényeket explicit specializálni (felül lehet definiálni) lehet adott típusokra.
- Akkor kell megírni, ha egy adott típusra az általános verzió nem működne.

- Példa

```
template<>
inline const char* max(const char* lhs, const char* rhs)
{
   return strcmp(lhs, rhs) > 0 ? lhs : rhs;
}
```

```
cout << "Max " << max("Gyula", "Payne") << endl;
```

Argumentum egyeztetés

- Milyen prioritással veszi figyelembe a függvény kiválasztásnál a lehetőségeket?
  1. Ha van olyan függvény, aminek az argumentumai pontosan egyeznek típusban, akkor azt választja ki.
  2. Template-et keres, van-e, ami pontosan egyezik.
  3. Automatikus konverzióval egyezik, de nem lehet template a konvertálandó paraméter.

Több template paraméter

- Lehet több template paramétere is a függvénynek
  - vesszővel elválasztva,
  - olyan, mint a függvények argumentum listája, csak itt típus is lehet paraméter.
  - Függvényen belül fel tudjuk használni ezeket.
- **Példa** árfolyamváltó függvény. Nem tudom, mi lesz a konverziós ráta, illetve az összeg típusa

```
template<typename RATE_TYPE, typename AMOUNT_TYPE>
AMOUNT_TYPE convert(AMOUNT_TYPE sum, RATE_TYPE exchange_rate)
{
        return sum*exchange_rate;
}
```

Parametrizált vagy generikus osztályok

Mi lehet template paraméter?

1. Típus
2. Típusos konstans
3. Template

- Példák

```
template <int DIM>
class Hypercube {
   const int dimensions;
   in origo[DIM];
public:
   Hypercube() : dimensions(DIM) {}
};
```

```
Hypercube<2> square();
const int haromde = 3;
Hypercube<haromde> cube();
```

Default template argumentumok

- Ugyanazok a szabályok vonatkoznak rá, mint a függvények default paramétereinél
- Példa

```
template <int DIM=3>
class Hypercube {
   const int dimensions;
   in origo[DIM];
 public:
   Hypercube() : dimensions(DIM) {}
};
```

Tagfüggvény template-ek (member templates)

- Olyan függvény template, ami egy tagfüggvény.
  - Annyi verzió generálódik automatikusan, ahány különböző paraméterrel használom.
  - Az osztálynak az adott típus nem paramétere
  - Ezt is lehet specializálni, stb.
  - Természetesen közönséges osztályra is működik.
  - Lehet több ilyen tagfüggvény is, mindben lehet használni ugyanazokat a paraméter neveket
- Példa

```
template <int DIM>
template<class T>
T Hypercube<DIM>::getVerticesCount()
{
    T result = pow(2, dimensions);
    return result;
}
```

Template-ek és öröklés

- Egy szabály: csak osztályból lehet leszármaztatni, template-ből nem.
- Template valami, amíg van legalább egy nem rögzített paramétere. Ha nincs, akkor már osztály.
- **Példák** a leszármazási típusokra

Template-ek és öröklés összefoglalás

1. Az ős template-ből példányosított osztály (tehát nem template!), leszármazott közönséges osztály
2. Egy osztály alapján leszármazott template-et hozunk létre
3. Az ős template-ből példányosított osztály (tehát nem template!), a leszármazott template osztály
4. Template paraméterben megadott ősosztály

Template specializáció

- Lehet osztálynál is, a függvényekhez hasonló módon

Öröklés vagy template?

- **Mikor használunk öröklést és mikor template-eket?**

- A template-nél: ugyanaz a viselkedés több típusra.
- Öröklés: a leszármazottakban felüldefiniálható a viselkedés (az egyes függvényekre).

Példa (tipikus 1. feladat)

- **Készítsen** egy sablon osztályt, amely szállításra való konténerek (*Container*) reprezentálására képes. Minden konténernek van egy azonosítója (*id*). Az osztályt úgy szeretnénk megírni, hogy bármilyen azonosító típussal működhessen (egész szám, sztring, tetszőleges saját osztály stb.), és az azonosítója lekérdezhető legyen (*getId()*).
- **Készítsen** egy (több konténerből álló) rakományokat reprezentáló osztálysablont (*Cargo),* amely bármilyen, azonos azonosító-típusú konténereket tud nyilvántartani (például csupa egész számmal azonosított *Container*). Nem tudjuk előre, mi lesz a tárolt konténerek azonosító típusa. A szállítmányhoz tetszőleges számú konténert adhatunk egyesével (*az addContainer* tagfüggvénnyel), kivenni viszont nem kell tudni belőle.

Példa (tipikus 1. feladat)

- Célunk, hogy a Cargo osztályból lekérdezzük egy adott azonosítójú konténert (visszakapjunk a megfelelő, eltárolt objektumra pointert) a *getContainerWithId* tagfüggvénnyel. Ha nincs megfelelő azonosítójú konténer, NULL-t adjunk vissza. (Példa: myCargo.getContainerWithId("ABCD123")).
- Milyen követelményeket támaszt az azonosító típusával szemben? Mutassa meg a megfelelő kódrészletet.
- Egy teljes példán mutassa be konténerek létrehozását egy szabadon választott azonosító típussal. Hozzon létre hozzá egy szállítmányt, helyezze el benne az elemeket, majd kérjen vissza a szállítmánytól egy adott azonosítójú konténer elemet. A feladat során ne használjon STL-t.

Konverziók

Típuskonverzió

- Ha a függvény/operátor paraméter típusa eltérő,
- Ha a függvény/operátor visszatérési érték típusa eltérő
- Ha az inicializálandó és az inicializáló objektum típusa eltérő
- Egyéb, összetett kifejezésekben

- típuskonverzióra lehet szükség

Beépített típusokra

- Általában kisebb méretűből nagyobbra van automatikus konverzió, vagyis ahol nem veszíthetünk adatot.
- Fordítva:

```
double d4 = 3.13;
int y1 = d4; // warning: possible loss of data
int y2 = (int)d4;
int y3;
y3 = d4; // warning: possible loss of data
```

Ha nem szeretnénk warningot látni, castoljunk explicit.

Referenciára

- Referenciára nincs automatikus cast, még akkor sem, ha bővebb típusra is castolunk.

```
void f(double& d) {d=1;}

void g() {
        int m=2;
        f(m);           // hiba
        f((double)m);   // hiba
        f((double&)m);  // lefordul, de …
}
```

Referenciára

```
f((double&)m);    // lefordul, de …
```

- Öngyilkosság: **nem születik** új változó (bár ha születne egy temporális, arra referenciát atadni is őrültség lenne),
- Az **eredeti** intre lesz egy double referencia: **túlcímzi** a területet, amikor állítja a double változót.
- Olyan, **mintha int*-ot** adnánk át és double*-ként értelmezve használnánk.

Referenciára

- Ellenben ha konstans referenciát veszünk át:

```
void f(const double& d) {d=1;}
```

- Úgy OK:
  - létrehoz egy temporális változót az új típusból, és arra referenciát ad át.

Szülőpointerrel/referenciával leszármazottra

- Van automatikus konverzió
- Láttuk a virtuális tagfüggvényeknél
- Ha A szülője B-nek:

```
B b;
A* pa = &b;
```

Leszármazottból szülőre, de nem pointerrel

```
A a; //ős

B b; //leszármazott

a = b; // OK,

a = (A)b; // ki is írható,
          //ezt jelenti az előző
```

Szülőről a gyerekre nem működik: mivel is egészítené ki?

Két nem kapcsolódó típusra

```
Stack s;

Person p;

s = (Stack)p;
```

- Nem fordul le
  - Mit is csinálna a fordító (Személy osztály objektumából Stack-et)

Két nem kapcsolódó típusra pointerrel

- Pointerrel lehet, de veszélyes, és nem biztos, hogy van értelme:

```
Person jozsi;
Person* pjozsi = &jozsi;
Stack* ps = (Stack*) pjozsi;
// OK, meggyőztük
ps->push();
```

- Egy személy objektumnak mondjuk, hogy push
- => elszáll

Két nem kapcsolódó típusra

- Van, amikor van értelme automatikus konverziónak nem a hierarchiában is
  - pl. char* és egy String osztály között
  - Vagy egy Complex szám és egy double között

```
void f1(string s) {…}
char* pc = "aaaa";
f1( pc );

void f2(char* s) {…}
String s1("abc");
f2( s1 );

void f3(Complex c) {…}
double d = 87;
f3( d );
```

Két nem kapcsolódó típusra

- Hogy lehet ezt elérni?
- 2 módszer létezik:
  - konverziós konstruktor
  - konverziós operátor

1. Konverziós konstruktor

- Minden egyargumentumú konstruktor.
  - vagy lehet többargumentumú is, csak a többinek legyen default értéke

```
class Complex {
      double re, im;
public:
      Complex(double d) {re = d; im = 0;}
};
…
Complex c = 5.0;
```

Konverziós konstruktor szabályok

- Csak egy lépés automatikus. Pl. létezik konverzió ilyen módon A-ból B-re, B-ről C-re. Az A→C konverzió nem működik (explicit sem). N-1 lépést írjunk ki explicit.
- Az egy lépésbe a const nem számít bele: ha a konstruktor const-ot vár nem const-tal is lehet hívni.
- Lehet mégis több, mint egy, de csak egy lépés legyen nem beépített típusra.

Példa

```
class Complex {
      double re, im;
public:
      Complex(double d) {re = d; im = 0;}
};
…
void f(Complex c) {…}
f(10); //működik, ugyanaz, mint:
f((Complex)(double)10);
```

Explicit

- Ha egy konstruktort valóban csak objektum konstruálására szeretnénk alkalmazni, és nem szeretnénk, hogy automatikus konverzióra legyen használva
  - pl. rejtett hibák miatt

2. Konverziós operátor

- Amit a konverziós konstruktor megoldott: egy másik típusból sajátot készített
  - Pl. *char** → *string*,
  - így ha valami *string-et* vár, *char**-al is tudtam hívni.

- Amit a konverziós operátor megold: fordítottja, vagyis egy saját típusból egy más típust készít
  - Pl. *string* → *char**,
  - vagyis ami *char**-ot vár *string-et* is használhasson.
- A konverziós operátor hasonlóan készíthető, mint egyéb operátorok.

Konverziós operátor példa

- A szintaktika: nincs visszatérés és paraméter sem (void sem lehet):

```
class String {
        char* data;
public:
        String( const char* c ) {//…itt a tobbi kihagyva}
        operator char*() const { return data; }
        operator const char*() const { return data; }
};
```

- Használat:

```
String s1("abc");
char* ps1 = s1; // operator char*()-ot hív
const char* ps2 = s1; // operator const char*()
```

Konverziós operátor szabályok

- Csak egy lépés automatikus

```
class A { //Egy osztály, ami int-té tud válni
   public:
       operator int() { …}
};

class B { //Egy osztály, ami A-vá tud válni
   public:
       operator A() { … }
};

void f() {
       B b;
       int x = b; // nem jó, nem tud B intté válni
       int x = (A)b; // jó, mivel előbb A lett
```

Konverziós operátor szabályok

- **Öröklődik**
  - Írunk az ősben egyet, ami int-re konvertál. A leszármazottban nem írunk.
  - Ekkor működik a leszármazottra is.
- **Lehet virtuális**
  - Pl. char *-gá konverziót megírunk az ősben virtuális tagfüggvényuként, és a leszármazottban felüldefiniáljuk.
  - Ekkor a leszármazottbeli konverizós operátor hívódik meg ősosztály pointerén keresztül elérve

Konverzió – többértelműség

- Ne írjuk meg mindkét konverziót, csak ha mindenképp szükséges
- Preferáljuk a konverziós konstruktort (más típusból a mi típusunkat).
  - A konverziós operátort pedig csak akkor, ha nem férünk hozzá a típushoz. Pl.:
    - Beépített típusra, pl. Complex-ből double-t
    - Ősből konverzió a leszármazottra és nem férünk hozzá a leszármazotthoz (a slicing ellentettje)
    - Egyéb

C++ stílusú típuskonverziók

- A nagyobb biztonságra való tekintettel a C++ saját konverziós szintaxist definiál
  - A C ugyanis jobban bízik a programozóban, a C++ viszont szigorúan típusos nyelv.
- A C megoldás egy kalap alá vesz bizonyos konverziós szándékokat
- Jobb lenne, ha pontosabban tudnánk megadni a konverzió célját. Ezeket segítik a következő operátorok

Statikus típuskonverzió

- static_cast < type-id > (kifejezés)
- Fordítás időben történik
- „biztonságos”, mivel tipikusan létezik konverziós út
- a konstansságot nem távolíthatja el
- Nem alkalmaz futási idejű típusellenőrzést
  - nem garantálja, hogy a konverzió eredményeképp a célobjektum teljes lesz

Újraértelmező típuskonverzió

- reinterpret_cast < type-id > (kifejezés)
- Mint a C-s elődje: a pointer típusát változtatja
  - vagyis azt, hogy milyen műveletek értelmezhetők egy memóriaterületen
- megengedi az egész típusok és a pointerek közötti konverziókat is

Dinamikus típuskonverzió

- dynamic_cast < type-id > (kifejezés)
- a hierarchián felfele, illetve lefele történő konverziókhoz szükséges
- Futásidőben történik
- Az osztályoknak polimorfaknak kell lenniük (legyen virtuális függvénye), különben ősről-leszármazottra konvertálás nem működik
- Használatához a futásidejű **típusinformációk kezelését be kell kapcsolnunk** a fordításkor
- A keresztbe konverziót is megoldja többszörös öröklés esetén, vagyis nagyon biztonságos.

Konstans típuskonverzió

- const_cast < type-id > (kifejezés)
- Képes konstans típust nem konstanssá tenni.
  - Ugyanis ez egy olyan veszélyes művelet, amelyet külön át kell gondolni

Kivételkezelés

Kivételek

- Exception
- Egy olyan mechanizmus, ami biztosítja, hogy ha hiba keletkezett valahol, akkor a futás (azonnal) a hibakezelőre ugorjon

A mechanizmus

- A **try** egy **védett blokkot** jelent, amihez hibakezelő catch blokkok tartoznak.
- Ha valahol a hívási fában hibafeltételt találunk: **dobunk** egy kivételt a **throw** kulcsszóval.
- A throw-nak **paramétert** kell adni.
  - A throw paramétere lehet beépített típusú változó, de tetszőleges osztálybeli objektum is.

A mechanizmus

- A throw olyan, mint egy return,
  - de addig ugrik felfelé a fában, amíg egy megfelelő catch elkapja.
  - Vagyis addig, amíg egy olyan catch blokkot nem talál, aminek a paramétere kompatíbilis a dobott típussal
  - Ha van megfelelő catch blokk, annak paraméterébe beíródik a throw által dobott paraméter, ezt a catch blokkon belül fel tudjuk használni.
- A catch csak akkor fut le, ha hiba történt.
- A catch-ből kilépés után a try-catch blokk utántól folytatódik a végrehajtás, nem lép vissza a throw utáni utasításra.

Egymásba ágyazás

- A try-catch blokkok egymásba ágyazhatók
- Példa

Hiba újradobása

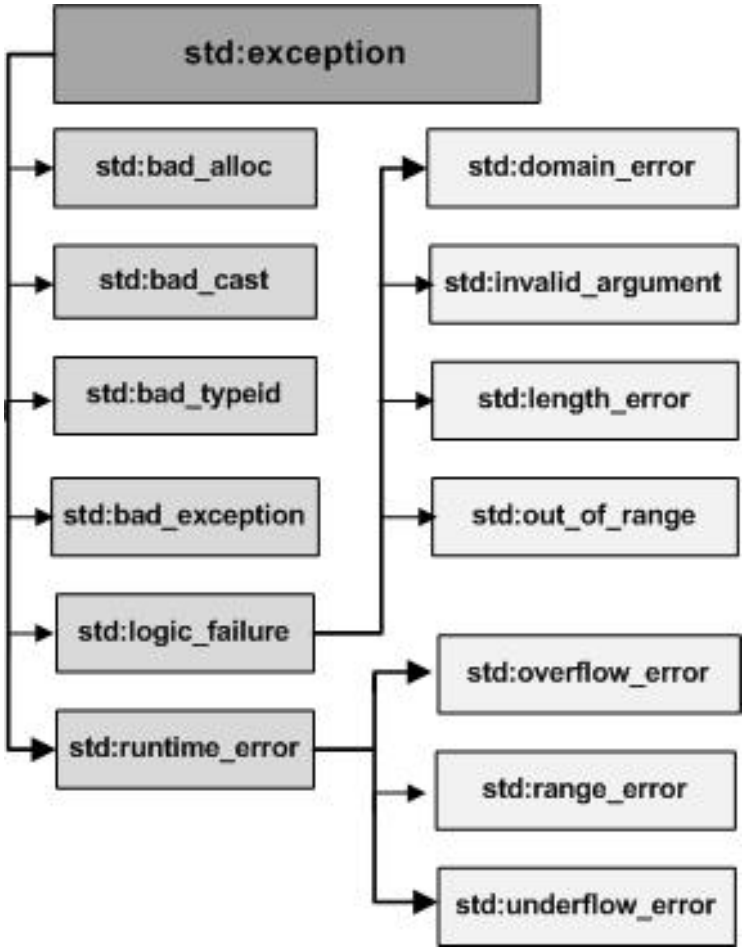
- Csak catch blokkon belül lehet
- Itt nincs paramétere a throwNak
- Ez az aktuális kivételt újradobja

- Példa

Típusosság

- Valójában nem szoktunk int-eket és char*-t dobni kivételként
  - hanem olyan objektumokat, amelyek valamilyen leírását tartalmazzák a hibának
- a Standard C++ library-ban vannak előre definiált szabványos exception típusok (hierarchiában)
- ezeket a kivételeket használjuk mi is
  - illetve ha saját kivétel típust szeretnénk létrehozni, akkor ezekből származtassunk le egy saját osztályt

Standard kódkönyvtár hierarchia



Kivétel elkapása

- Típus szerint lehet elkapni őket
  - ez szűrésre használható
- Több catch blokkot is meg lehet adni egymás alatt
  - ha a try blokkban kivételt dobott egy függvény a throw-val, akkor *sorrendben* végignézi a catch blokkokat és egy catch blokknál megáll, ha:

…megáll, ha:

1. catch(...)-ot talál
   - Ez mindent megfog, de nem kapjuk meg paraméterben az exception objektumot
2. típusa pontosan egyezik a kivétel objektum típusával
   - pl. int, string, exception
   - Ekkor másolat készül az eredeti exception objektumról, meghívódik a másoló konstruktora.
   - Olyan mintha függvény paraméter átadás lenne.

…megáll, ha:

3. Referencia vagy const referencia ugyanarra a típusra
   - catch(*exception*& e) vagy catch(const *exception*& e)
   - Mindig másolat készül a kivételről, nem a lokális objektumra kapnánk tehát referenciát
   - Ezt fogjuk szeretni.
4. A típus alaposztálya
   - catch(*exception* e), ahol a throw *spec_exception* volt, ahol a *spec_exception* az *exception* leszármazottja
   - Másolat készül.
5. Referencia vagy const referencia az osztály ősére
   - catch(*exception* & e) vagy catch(const *exception*& e).
   - Ezt is fogjuk szeretni.

…megáll, ha:

6. Pointerként, amire van konverzió a standard pointer konverziós szabályok szerint
   > Pl. catch(*exception** e) , akár ha leszármazottat dobtunk: throw new *spec_exception*;

- Az exception-kezelés egyik sarokköve, hogy ősosztálybeli típuson keresztül leszármazottat is meg tud fogni

Pointerdobással vigyázni!

```
void someFunction()
{
  exception ex;     //lokális objektum!
  throw &ex;
}
```

- Jobbnak tűnik:

```
void someFunction()
{
        throw new exception;
        // throw a pointer to a new heap-based object
}
```

- De:
  - Maga a new is dobhat kivételt
  - Amikor elkapjuk, esetleg nem tudjuk, hogy a pointer a new-val lett létrehozva: ekkor bajban vagyunk, mert nem tudjuk, hogy meg kell-e hívni a delete-et

A stack visszacsévélése

- amíg a catch el nem kapja, az összes (stack-en lefoglalt) lokális objektumot visszafelé haladva felszabadítja
  - a heap-en, new-val foglaltakat nem
  - természetesen meghívódnak az objektumok destruktorai.
- A destruktorban soha ne dobjunk kivételt,
  - mert ha aktív kivétel közben újabb kivételt dobunk, akkor azt nem tudjuk kezelni:
    - meghívódik a terminate, alapértelmezésben kilép az alkalmazás.
  - Ha mégis dobnánk, kapjuk is el a destruktorban és kezeljük le. Ekkor OK.

Köszönöm a féléves megtisztelő figyelmeteket!

Kérdés - válaszok