Programozás alapjai II.
(8. ea) **C++**

többszörös öröklés, cast, perzisztencia

Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

Öröklés ism.

- Egy osztályból olyan újabb osztályokat származtatunk, amelyek rendelkeznek az eredeti osztályban már definiált tulajdonságokkal, viselkedéssel.

- Analitikus - Korlátozó
- Egyszeres - Többszörös

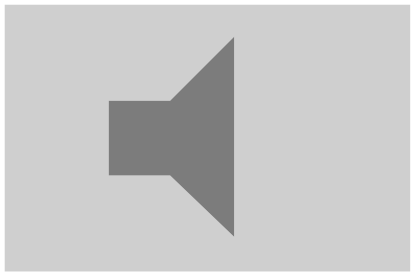
Többszörös öröklés

- Ha két osztály merőben különbözik, de mindkettőben valamit meg kell valósítani a másik számára.
- Az új osztálynak többféle arcot kell mutatnia (mutatókonverzió).
- Sokszor kiváltható barát függvényekkel, de nem a legjobb megoldás.
- Objektum és a környezet (pl. grafikus) kapcsolata.

Többszörös öröklés/2

- Két vagy több bázisosztályból származtatunk.
- Több OO nyelv nem támogatja, mert bonyolult implementálni.
- Ezekben a nyelvekben interfésszel váltják ki a többszörös öröklést.
- Leggyakrabban grafikus interfész és a modell kapcsolatánál használjuk.

Példa: Nyomógomb és callback fv.



A grafikus rendszer kezeli a felhasználói eseményeket.

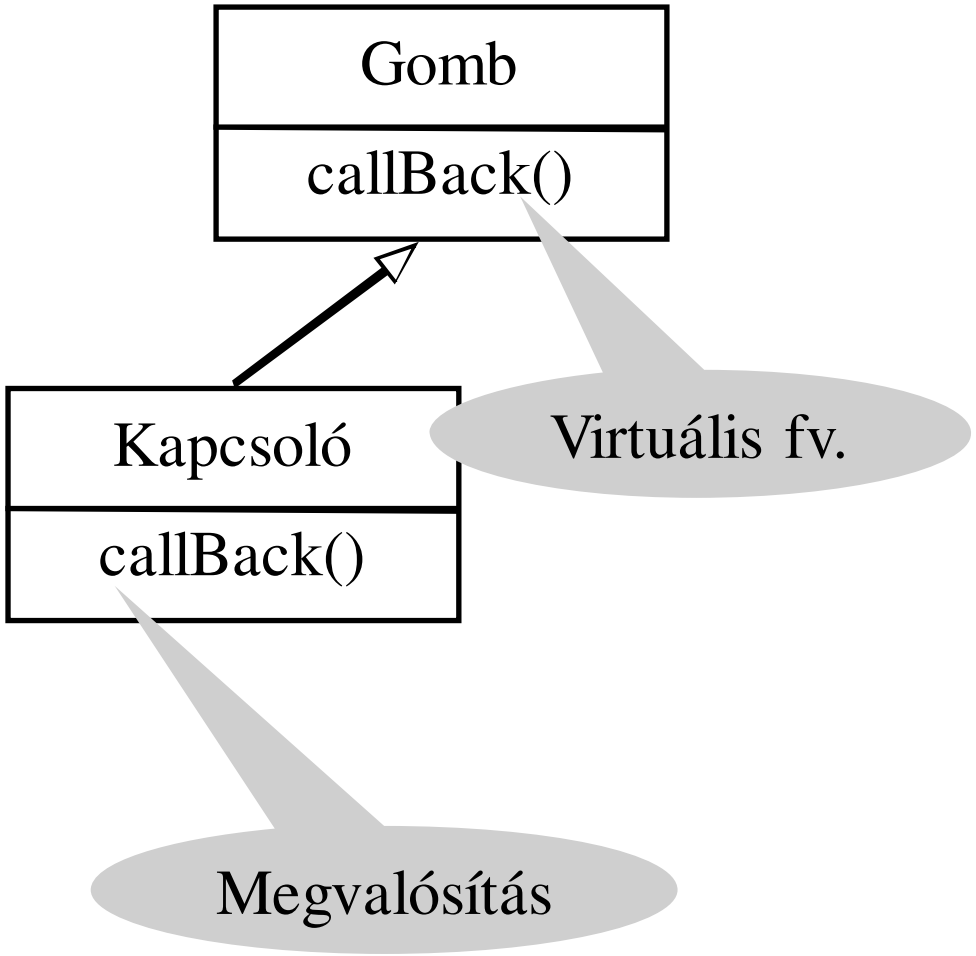
Ha megnyomják (áthaladnak fölötte, elengedik, stb.), meghívja az alkalmazás megfelelő függvényét (callback), amit korábban az alkalmazás közölt a gombbal.

Példa: Kapcsoló

Gombnyomásra ki-be lehessen kapcsolni

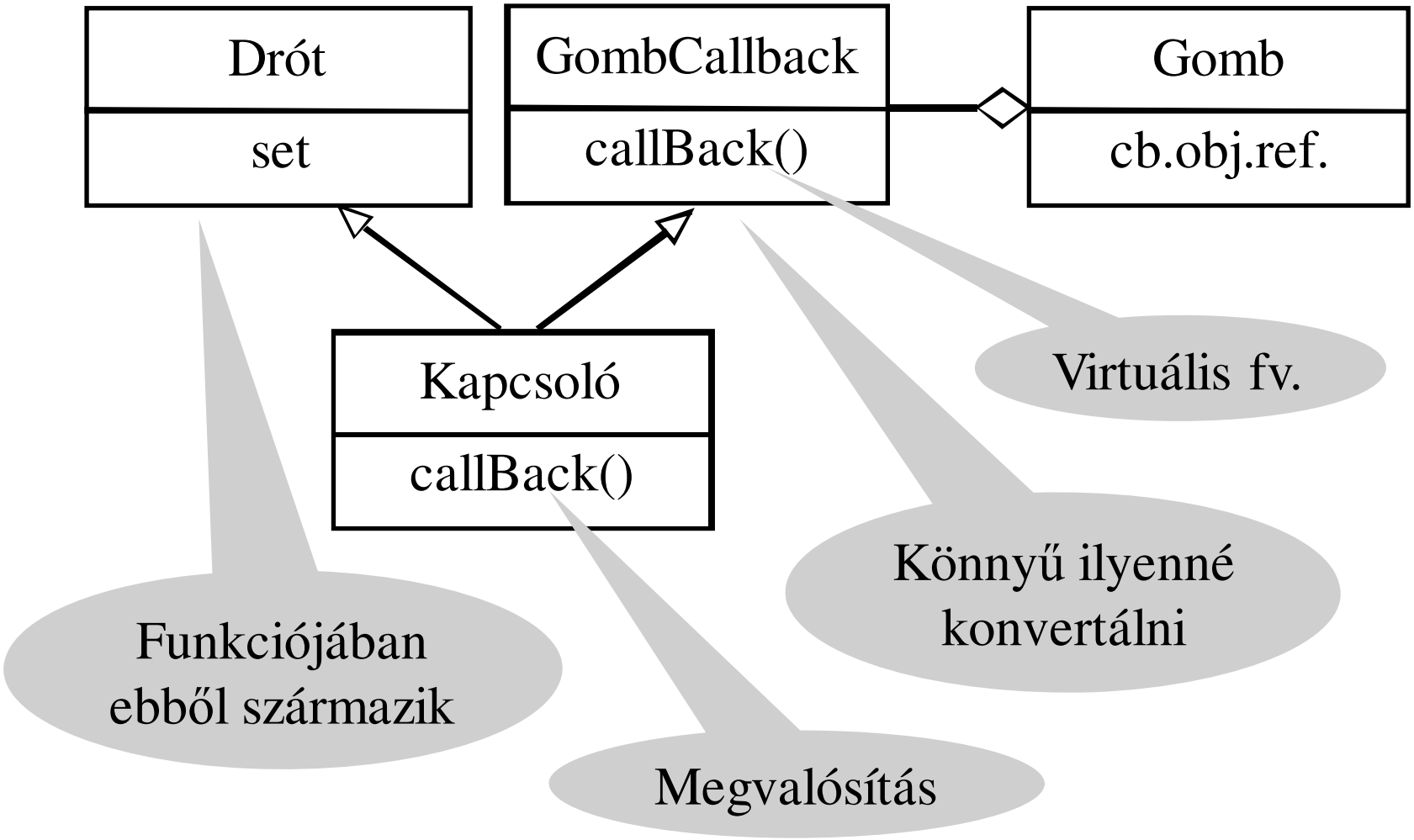
- A (grafikus) felhasználói felületen megvalósított gomb megkap minden felhasználói inputot. Amikor azt kapja, hogy "megnyomták", akkor szól a kapcsolónak.
  - Hogyan szól neki, ha nem ismeri?
  - Próbálkozzunk származtatással!

Kapcsoló a grafikus felületből?



- A grafikának semmi köze a funkcionalitáshoz!
- Eddig úgy tudtuk, hogy a Kapcsoló a Drót-ból származik!
- A grafikus felülettől függ a működés?

Kapcsoló többszörös öröklléssel



Kapcsoló megvalósítása

```
class GombCallback {            // callback funkcióhoz
public:
   virtual void callBack() = 0; // virtuális cb. függvény
};

class Gomb {     // felhasználói felület objektuma
   GombCallback &cb;          // objektum referencia
public:
   Gomb (GombCallback &t) :cb(t) {}// referencia inic.
   void Nyom() { cb.callBack(); }       // megnyomták
   ....
};
```

Kapcsoló megvalósítása/2

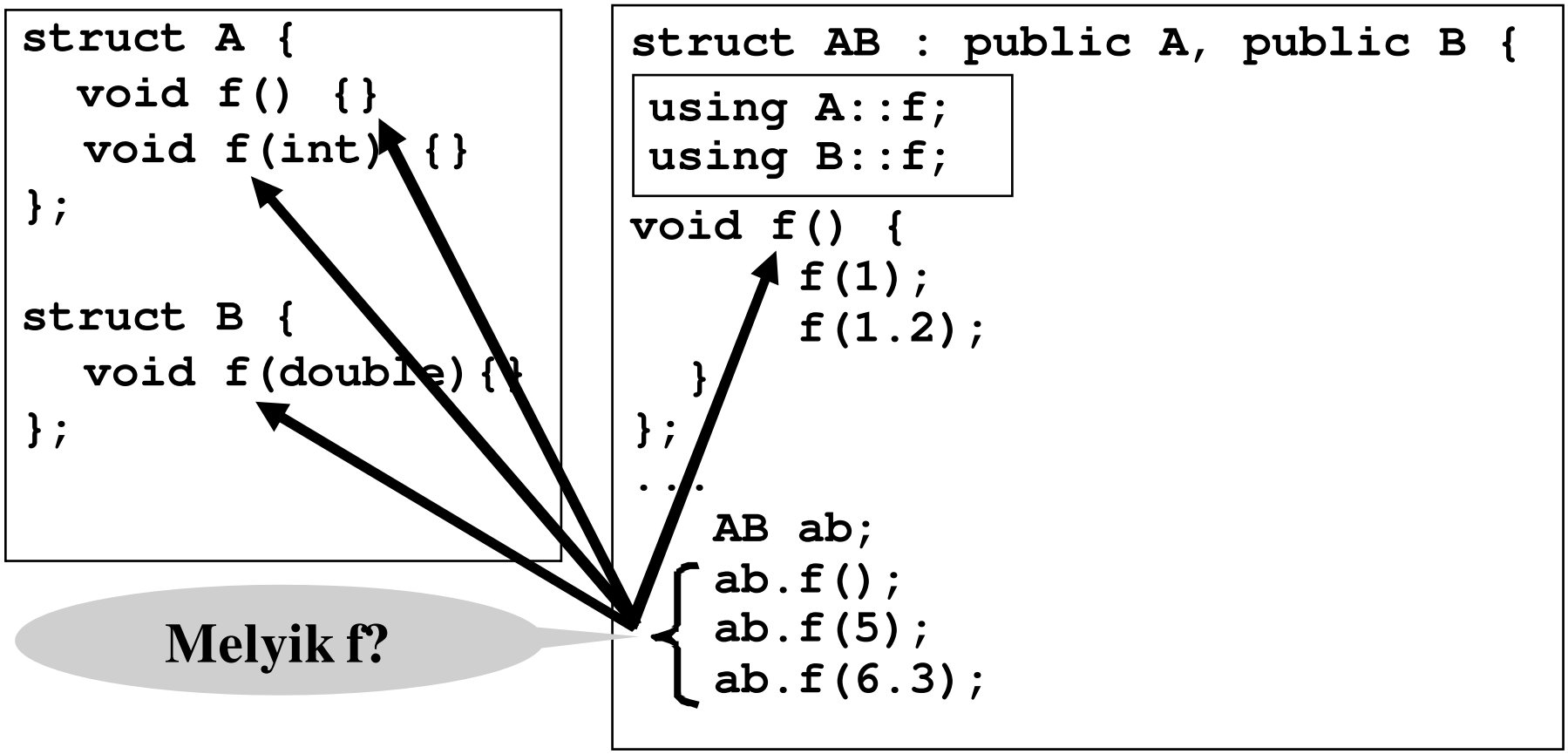
```
class Kapcsolo :public Drot, public GombCallback {
    int be;  // állapot
public:
    void ki();
    void be();
   ....
   void callBack() { if (be) ki(); else be(); } // callback
};
```

Az osztály kompatibilis a **GombCallback** osztállyal, amin keresztül a **Gomb** osztály eléri a **callBack** függvényt.

```
Kapcsolo k1;
Gomb g1(k1);
```

```
class Gomb {
   GombCallback &cb;
public:
   Gomb (GombCallback &t) :cb(t) {}
   void Nyom() { cb.callBack(); }
};
```

Többsz. öröklés + fv. overload

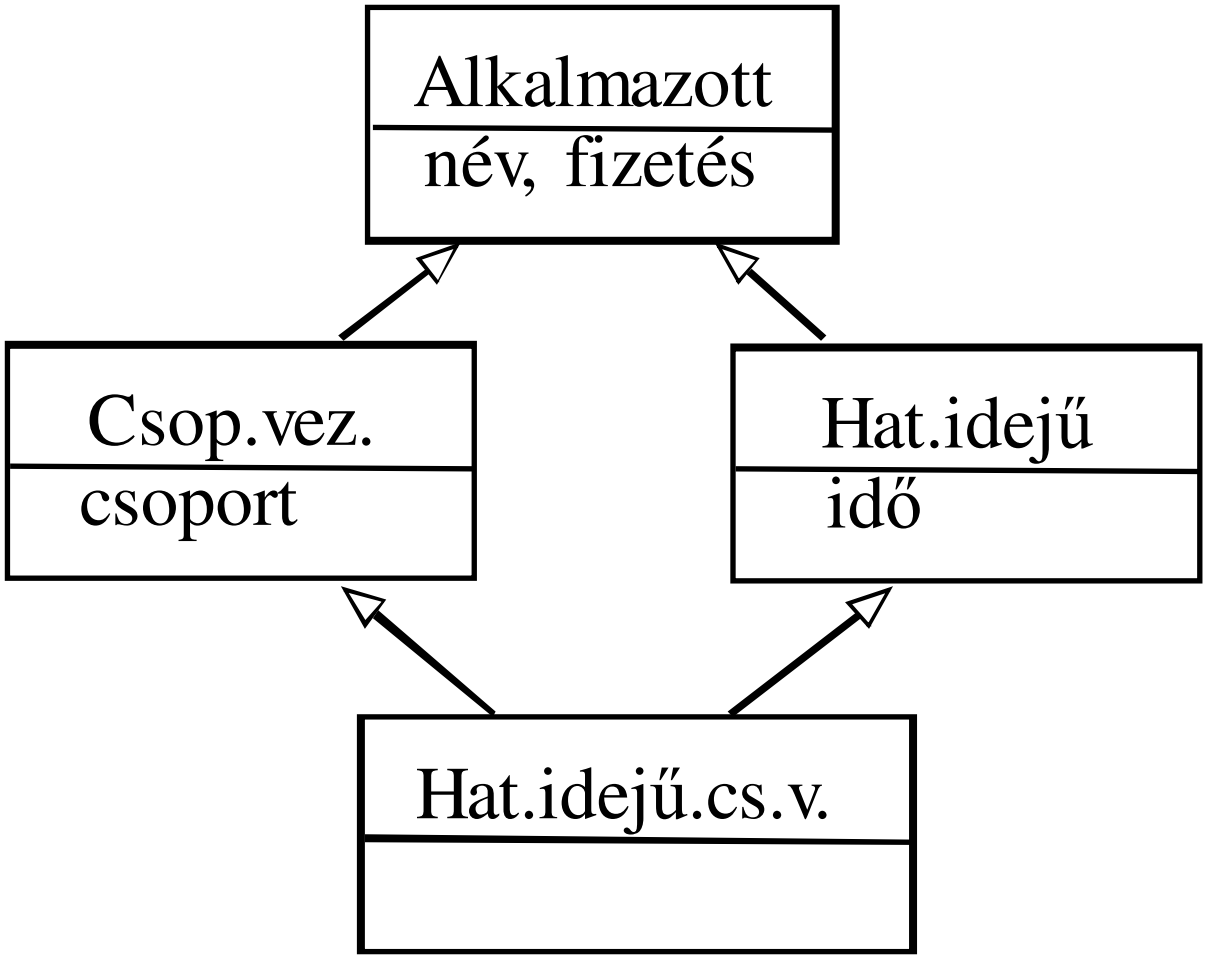


Feltételezés: A többszörös öröklésnél merőben eltérőek az alap-osztályok, az azonos nevű függvények más-más funkciót látnak el.

Többszörös öröklés problémái

- Többszörös öröklés különös figyelmet igényel, ha előfordulhat, hogy egy alaposztály különböző leszármazottjai "összetalálkoznak".
- Ekkor ún. rombusz v. "diamond" struktúra alakul ki.
- Példa: irodai hierarchia

Irodai hierarchia



Irodai hierarchia /2

```
class Alkalmazott {
protected:
   char      nev[20];              // név
   double   fizetes;               // fizetés
public:
   Alkalmazott(char  *n, double fiz);
};
class  CsopVez :public Alkalmazott {
   csop_t   csoport;               // csoport azon.
public:
   CsopVez(char *n, double f, csop_t cs)
                  :Alkalmazott(n, f), csoport(cs) { }
};
```

Irodai hierarchia /3

```
class HatIdeju :public Alkalmazott {
   time_t  ido;                  // szerződése lejár ekkor
public:
   HatIdeju(char *n, double f, time_t t)
                        :Alkalmazott(n, f), ido(t) { }
};
class HatIdCsV :public CsopVez,
                        public HatIdeju {
public:
   HatIdCsV(char* n, double f, csop_t cs, time_t t)
                :CsopVez(n, f, cs), HatIdeju(n, f, t) { }
};
```

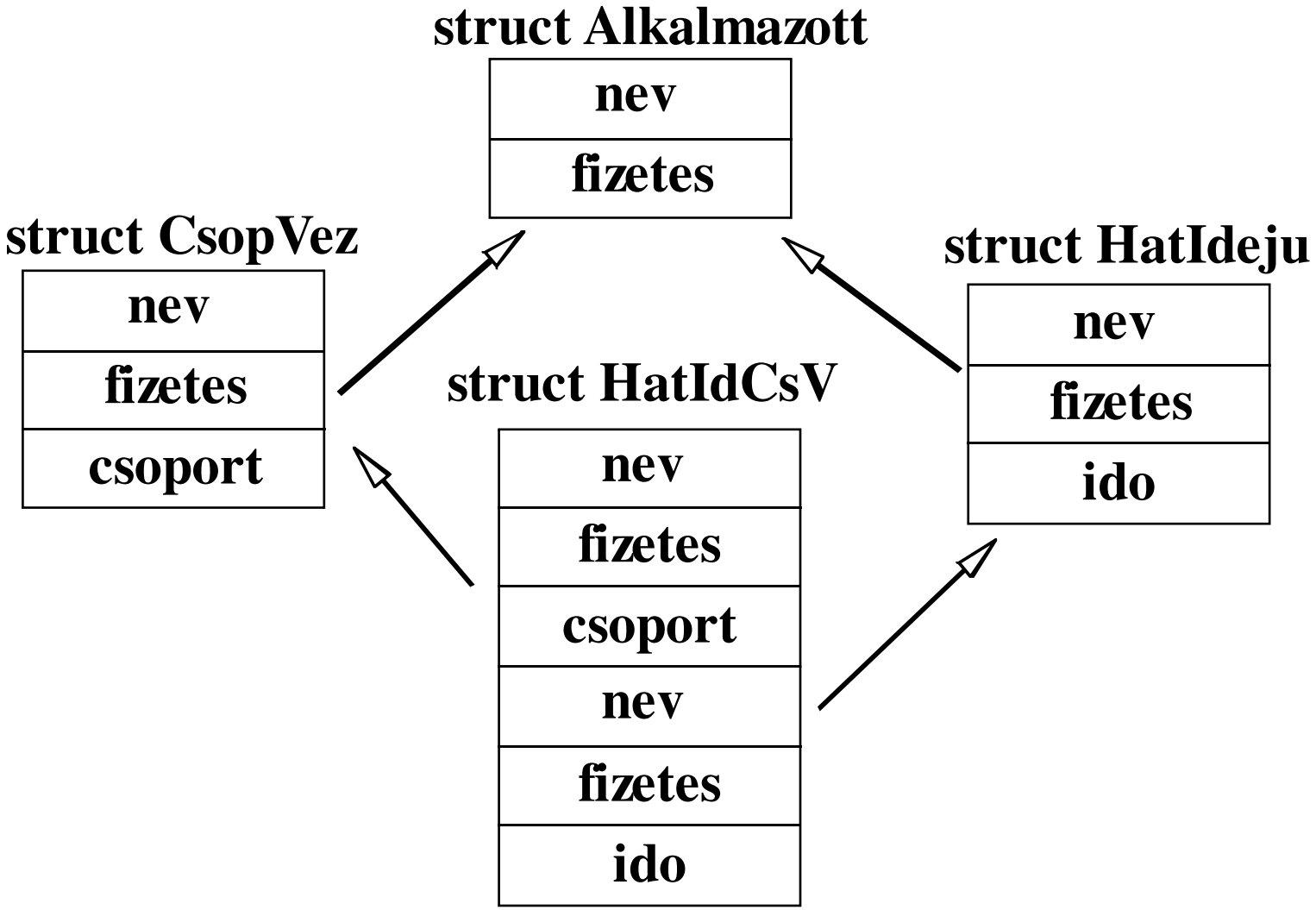
Két neve és két fizetése van ?

Elérhetők ezek a mezők?

```
class HatIdCsV :public CsopVez,
                public HatIdeju {
public:
   HatIdCsV(char* n, double f, csop_t cs, time_t)
            :CsopVez(n, f, cs), HatIdeju(n, f, t) { }
   void Kiir() {
        cout << CsopVez :: nev << endl;
        cout << HatIdeju :: nev << endl;
    }
};
```

A scope operátorral kiválasztható, tehát önmagában ez még nem lenne baj, de baj lehet belőle.

Memóriakép


struct Alkalmazott
nev
fizetes
struct CsopVez
nev
fizetes
csoport
struct HatIdeju
nev
fizetes
ido
struct HatIdCsV
nev
fizetes
csoport
nev
fizetes
ido

Miből fakad a probéma ?

- Többszörös elérés az öröklési gráfon.

Miért nem vonja össze a fordító ?

- A nevek ütközése az öröklés megismert szabályai alapján még nem jelent bajt.
- Lehet hogy szándékos az ütközés.
- Automatikus összevonás esetén a kompatibilitás veszélybe kerülhet.

Megoldás: Virtuális alaposztály

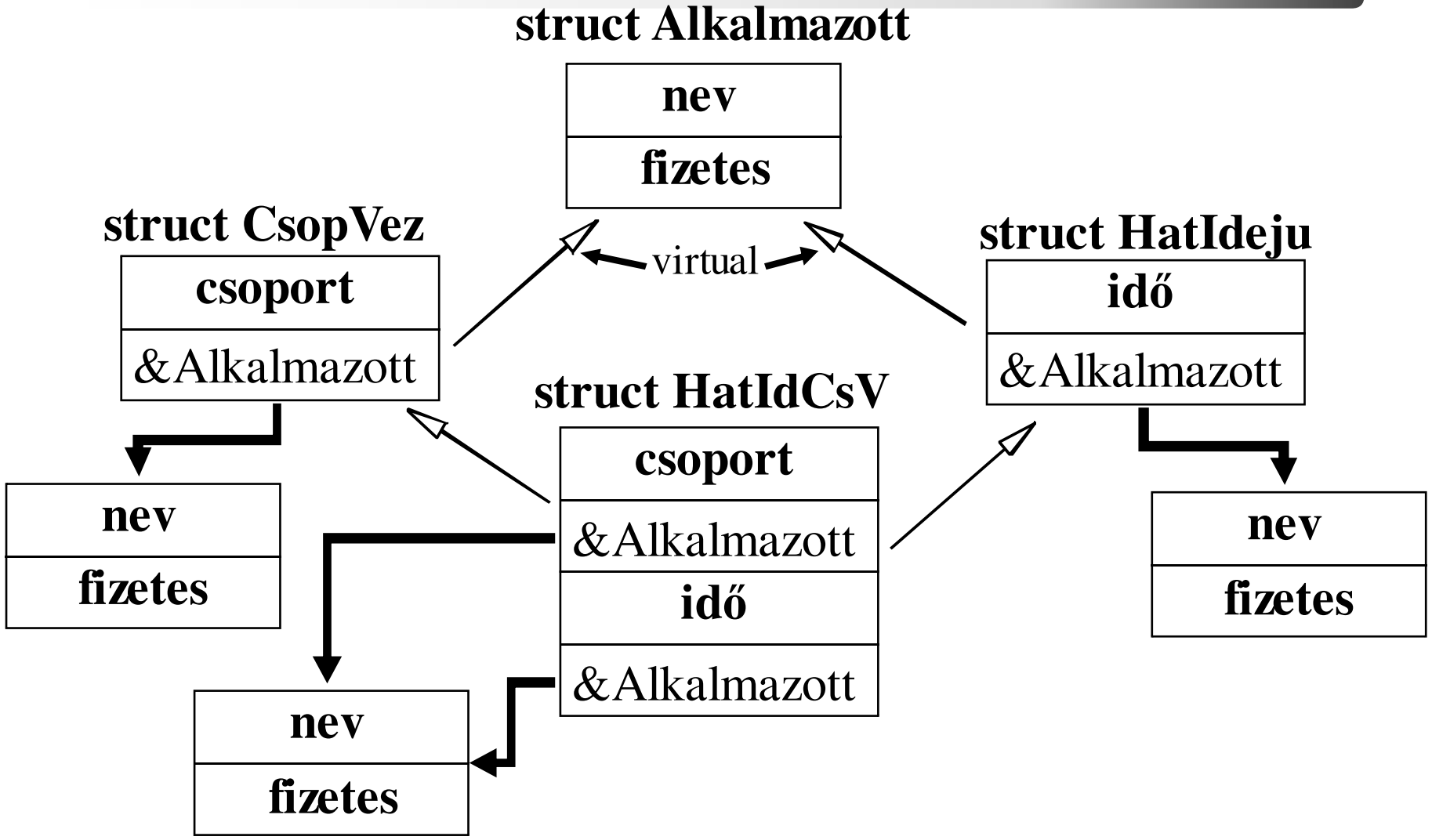
```
class CsopVez :virtual public Alkalmazott {
....
public:
   CsopVez(char *n, double f, csop_t cs)
                              :Alkalmazott(n, f),// csak lánc végén
                                csoport(cs) { }
};
class HatIdeju :virtual public Alkalmazott{ .... };
class HatIdCsV :public CsopVez, public HatIdeju {
public:
   HatIdCsV(char* n, double f, csop_t cs, time_t t)
         :Alkalmazott (n, f),   // csak ha a lánc vége
          CsopVez(NULL, 0, cs),   // tudjuk, hogy nem
          HatIdeju(NULL, 0, t) { } // hívódik, ezért lehet NULL
```

Csak az **öröklési lánc legvégén** hívódik meg az alaposztály konstruktora.

Virtuális alaposztály

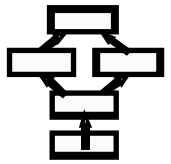
1. Alaposztály adattagjai nem épülnek be a származtatott osztály adattagjaiba. A virtuális függvényekhez hasonlóan indirekt elérésűek lesznek.
2. Az alaposztály konstruktorát nem az első származtatott osztály konstruktora fogja hívni, hanem az öröklési lánc legvégén szereplő osztály konstruktora.

Memóriakép virt. alaposztállyal



Irodai példa virt. alaposztállyal

```
class Alkalmazott { ... };
class HatIdeju :virtual public Alkalmazott{  ... };
class CsopVez :virtual public Alkalmazott { ... };
class HatIdCsV :public CsopVez, public HatIdeju { ... };
class HatIdCsVH :public HatIdCsV { ... };
```



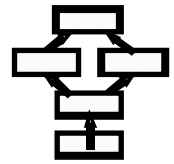
Melyik konstruktor hívja az Alkalmazott konstruktorát ?

```
Alkalmazott melos("Lusta Dick", 100); // Alkalmazott
HatIdeju grabo("Grabovszki", 300); // HatIdeju
CsopVez fonok("Mr. Gatto ", 5000); // CsopVez
HatIdCsV gore("Mr. Tejfel", 3000); // HatIdCsV
HatIdCsVH ("Safranek", 500); // HatIdCsVH
```

Aki a lánc végén van.

És, ha nem lenne virtuális ?

```
class Alkalmazott { ... };
class HatIdeju :public Alkalmazott{ ... };
class CsopVez :public Alkalmazott { ... };
class HatIdCsV :public CsopVez, public HatIdeju { ... };
class HatIdCsVH :public HatIdCsV { ... };
```



Melyik konstruktor hívja az Alkalmazott konstruktorát ?

```
Alkalmazott melos("Lusta Dick", 100); // Alkalmazott
HatIdeju grabo("Grabovszki", 300); // HatIdeju
CsopVez fonok("Mr. Gatto ", 5000); // CsopVez
HatIdCsV gore("Mr. Tejfel", 3000); // CsopVez, HatIdeju
HatIdCsVH ("Safranek", 500); // CsopVez, HatIdeju
```

Aki az első a láncban.

Konstruktor feladatai

- Öröklési lánc végén hívja a virtuális alaposztályok konstruktorait.
- Hívja a közvetlen, nem virtuális alaposztályok konstruktorait.
- Létrehozza a saját részt:
  - beállítja a virtuális alaposztály mutatóit
  - beállítja a virtuális függvények mutatóit
  - hívja a tartalmazott objektumok konstruktorait
  - végrehajtja a programozott törzset

Lehet-e konstruktorban virt. fv?

- Lehet, de nem az történik, amit várunk. Az alaposztály konstruktora a származtatatott o. konstruktora előtt fut le, így a virtuális függvénypointerek beállítása nem történik meg.

B konstr. **még nem futott.**

```
struct A {
   A() { f(); }
   virtual void f() { cout << "A::f";}
};
```

```
struct B :public A {
   B() { }
   void f() {  cout << "B::f"; }
};
```

Kiírás: A::f

Destruktor feladatai

- Megszünteti a saját részt:
  - végrehajtja a programozott törzset
  - tartalmazott objektumok destruktorainak hívása
  - virtuális függvénymutatók visszaállítása
  - virtuális alaposztály mutatóinak visszaállítása
- Hívja a közvetlen, nem virtuális alaposztályok destruktorait.
- Öröklési lánc végén hívja a virtuális alaposztályok destruktorait.

Lehet-e destruktorban virt. fv?

- Lehet, de nem az történik, amit várunk. Az alaposztály destruktora a származtatatott o. destruktora után fut le, így a virtuális függvénypointerek már visszaálltak.

B destr. **már lefutott.**

```
struct A {
   ~A() { f(); }
   virtual void f() { cout << "A::f";}
};
```

```
struct B :public A {
   ~B() { }
   void f() {  cout << "B::f"; }
};
```

Kiírás: A::f

Lehet-e konst/destr virtuális?

- Konstruktor **NEM lehet** virtuális!
- Virtuális **destruktor** használata viszont nagyon fontos, ha fentről szüntetünk meg.

```
struct  A {
    virtual ~A() { };
};
```

```
A *pa = new B;
delete pa;
```

```
class B :public A {
    char *p;
    C c;
public:
    B() { p=new char[10]; }
    ~B() { delete[] p; }
};
```

Ha nem virtualis, akkor nem szabadul fel a c adattag, és a dinamikus terület sem!

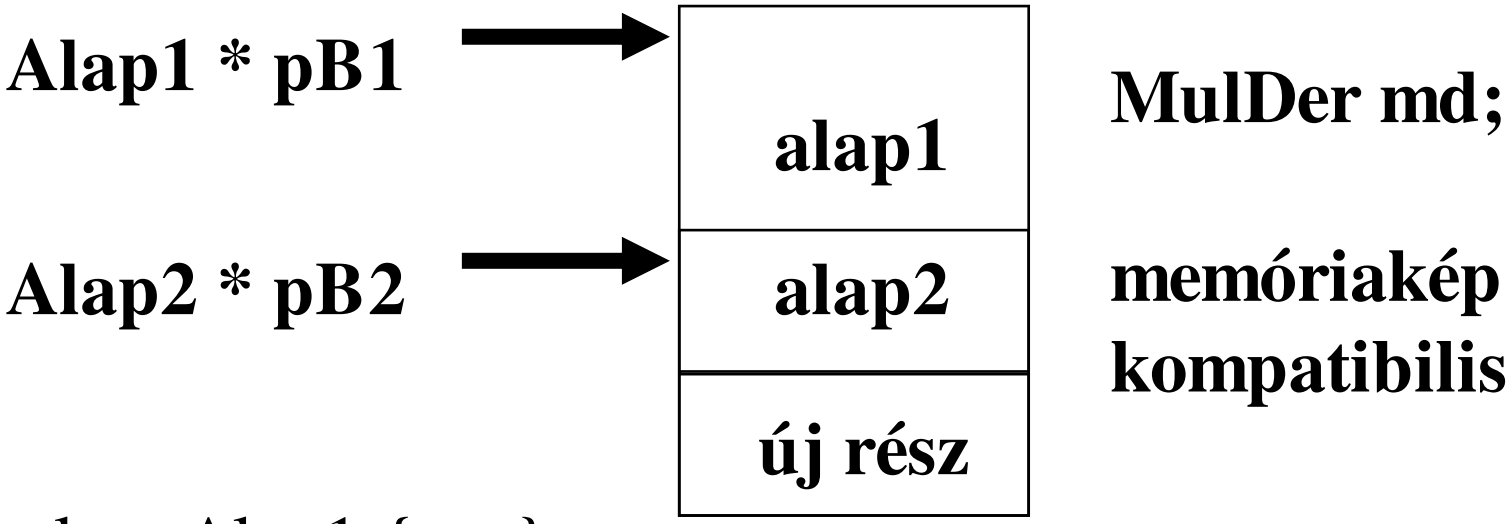
Elkerülhető a többsz. öröklés?

- Egyes OO nyelvekben nincs többszörös öröklés, de van helyette interfész, amivel pótolható a hiánya. C++ -ban ilyen nincs, ezért teljesen nem kerülhető el.
- Biztosan nem kerülhető el, ha
  - mindkét osztály "arcát" mutatni kell (pl. ny.gomb)
  - mindkét osztályt valamiért alaposztállyá kell konvertálni (upcast)

Mutatókonverzió újra

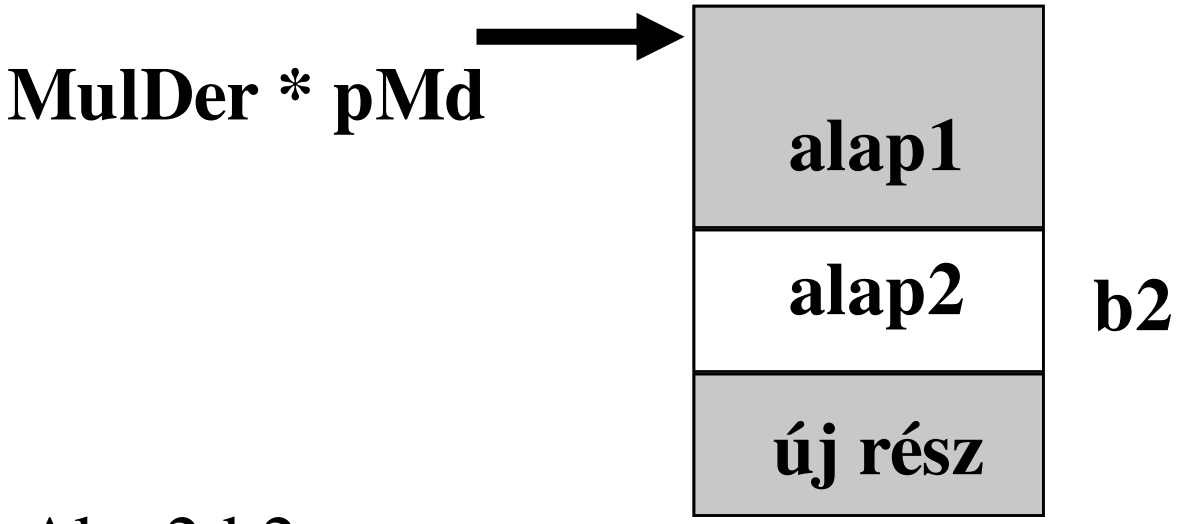
- Származtatott osztály a kompatibilitás révén könnyen konvertálható alaposztályra.
- Ez hívjuk "upcast"-nak.
- Leggyakrabban pointereket konvertálunk (heterogén gyűjtemény).
- Néha referenciát. (copy konstruktor hívása)
- Többszörös öröklésnél a konverzió nagy figyelmet igényel.

Mut. konv. többszörös öröklésnél




```
class Alap1 { ... };
class Alap2 { ... };
class MulDer: public Alap1, public Alap2 { ... };
MulDer md;
Alap1 * pB1 = &md;
Alap2 * pB2 = &md;      // mutató értékmódosítás!
```

Konv. többsz. öröklésnél szárm.-ra




```
Alap2 b2;
MulDer  *pmd = (MulDer *) &b2;
        // mutató értékmódosítás!
        // nem létező adatmezőket és üzeneteket el lehet érni
        // veszély: explicit konverzió
```

Megbocsátható down cast

- Néha előfordul, hogy tudjuk, hogy egy adott mutató egy származtatott osztályból származik, de valamiért alappá konvertáltuk (pl. heterogén kollekció)
- Ekkor biztonságos a dynamic_cast használata, ami futási időben értékelődik ki.

Ha MulDer* tipusú, akkor OK, egyébként NULL

```
MulDer md;
Alap *p = &md;
MulDer *pa = dynamic_cast<MulDer*>(p);
```

cast átértékelése

C: (típus)

Explicit típuskonverziót végez. Használata körültekintést igényel. C programokban leginkább a malloc környékén fordul elő joggal. Eredménye nem lvalue.

C++:

Sokkal többször és sokkal több okból fordulhat elő joggal. A veszélyek csökkentésére több változata létezik: (dynamic_cast, static_cast, const_cast, reinterpret_cast, (tipus), tipus())

dynamic_cast

szintaxis: ***dynamic_cast<T>(v)***

v – alaposztályra hivatkozik, vagy pointer

T – származtatott osztályra hivatkozik, vagy void pointer

```
struct A {
  virtual ~A(){}
};
struct B :A {};
struct C :A {};
```

polimorf

```
A* apB = new B;
A* apC = new C;
```

NULL

```
B* bp1 = dynamic_cast<B*>(apB);
B* bp2 = dynamic_cast<B*>(apC);
```

static_cast

szintaxis: ***static_cast<T>(v)***

v – alaposztályra hivatkozik, vagy pointer

T – származtatott osztályra hivatkozik, vagy void pointer

```
struct A {
};
struct B :A {};
struct C :A {};
```

```
A* apB = new B;
A* apC = new C;
```

```
B* bp1 = static_cast<B*>(apA);
B* bp2 = static_cast<B*>(apC);
```

Fordítási időben történik, nincs futás idejű ellenőrzés!

Jelzi, ami fordítási időben tilos. (pl. int* -> B*)

const_cast

szintaxis: ***const_cast<T>(v)***

Csak a *const* vagy a *volatile* minősítő eltávolítására vagy előírására használható.

```
const int a = 10;
const int* b = &a;
int* c = const_cast<int*>(b);
*c = 30; //nincs ford.hiba. !!Nem definit!!

int a1 = 40;
const int* b1 = &a1;
int* c1 = const_cast<int*>(b1);
*c1 = 50; // minden OK, mert a1 nem konst.
```

reinterpret_cast

szintaxis: ***reinterpret_cast<T>(v)***

Ellenőrzés és változtatás nélkül átalakítja v-t a megadott típusúvá. Igen veszélyes, de legalább könnyen felismerhető a forrásban.

A *const* ill. *volatile* minősítés nem módosítható vele.

```
struct B {};
B* p = new B;
long l = reinterpret_cast<long>(p);
```

(típus), típus( )

A C stílusú cast mellett használható még az úgynevezett funkció stílusú cast is.

Lehetőleg az úgynevezett nevesített (dynamic_cast, static_cast, const_cast, reinterpret_cast) változatokat kell használni. Használatukkal a konverziókból adódó problémák sokkar könnyebben felfedezhetők.

```
int i, j; double d;
d = (double)i/j;
d = double(i)/j;  // funkció stílusú cast
```

Explicit konstruktor (ism.)

- Az egyparaméterű konstruktorok egyben automatikus konverziót is jelentenek:

  pl: String a = "hello"; → String a = String("hello");

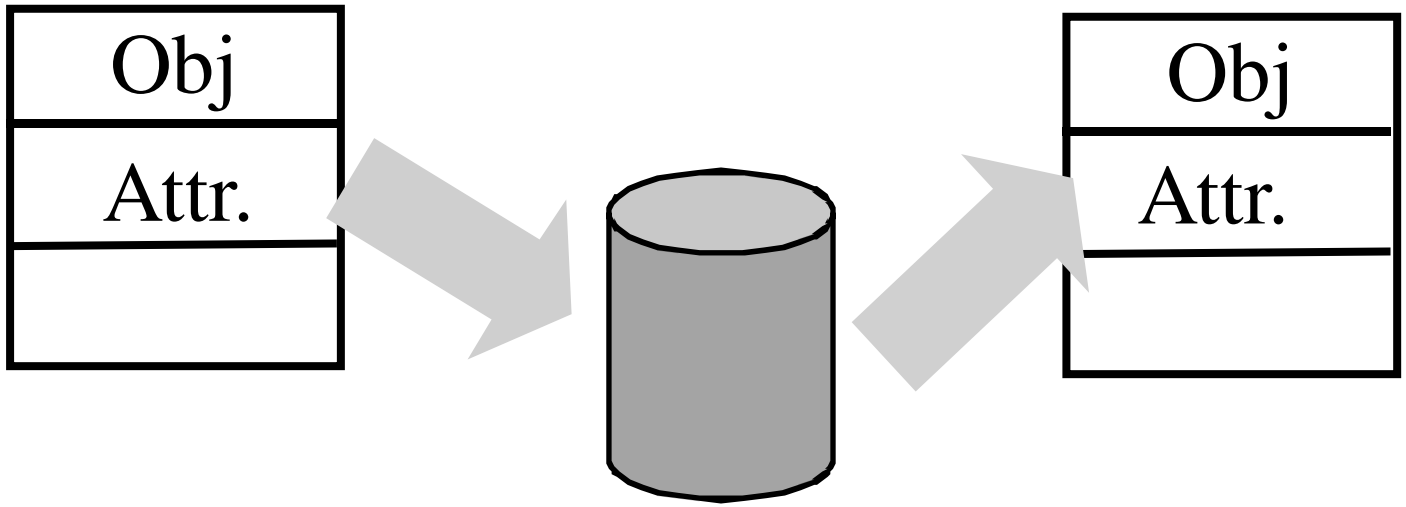
- Ez kényelmes, de zavaró is lehet:
  - tfh: van String(int) – konstruktor, ami megadja a string hosszát, de nincs String(char) konstruktor;
  - ekkor: String b = 'x'; → String b =String(int('x')); nem biztos, hogy kívánatos.
- Az aut. konverzió az explicit kulcsszóval kapcsolható ki. (pl: explicit String(int i);)

Explicit konstruktor példa:

```
class Komplex {
      double re, im;
public:
      Komplex(double re = 0, double im  = 0)
                        :re(re), im(im){ }
};
void valamifv(class Komplex);

int main() {
    Komplex k1 = 12;  // OK.
...
    valamifv(12); // Lehet, h. nem ezt akarta
```

Perzisztencia



- A perzisztens objektumok állapota elmenthető és visszatölthető egy későbbi időben, esetleg másik gépen létrehozott objektumba.
- A visszatöltött objektum "folytatja" a működést.

Perzisztencia/2

- Az objektum a saját állapotát képes kiírni egy adatfolyamba → serializáció
- Az objektum a saját állapotát képes beolvasni egy adatfolyamból → deserializáció
- Szempont lehet a hordozható külső formátum is. Ez különösen fontos elosztott rendszereknél.
- A perzisztenciát gyakran többszörös öröklléssel vagy interfésszel oldják meg.

Perzisztencia öröklléssel I./1

```
class Serialize {
   int size;                    // kírandó adat mérete
public:
   Serialize(int s) :size(s) {}     // méret beállítása
   void write(ostream&  os) const {
        os.write((char *)this, size);      // kiírás
   }
  void read(istream&  is) const {
        is.read((char *)this, size);       // beolvasás
   }
};
```

Perzisztencia öröklléssel I./2

```
class Complex {
   double re, im; ....
}
class PComplex : public Serialize, public Complex {
public:
   PComplex(double re, double im) : Complex(re, im),
   Serialize(sizeof(PComplex)) {}
};

int main() {
  ofstream f1("f1.dat");
  PComplex k1(1, 8);
  k1.write(f1);
  return 0;
}
```

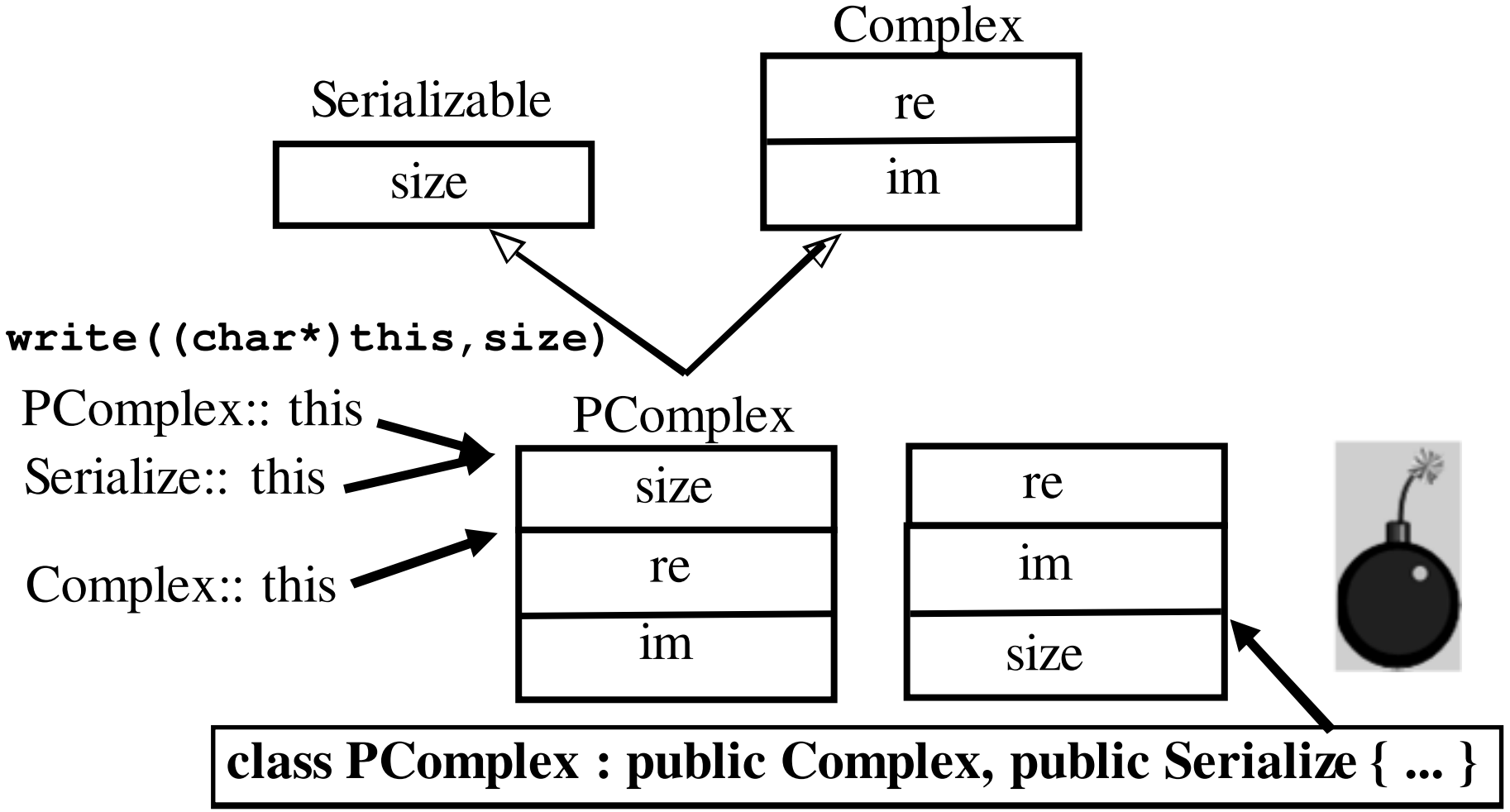
Képes kiírni ill. visszatölteni

Származtatott mérete

```
int main() {
  ifstream f1("f1.dat");
  PComplex k1;
  k1.read(f1);
  return 0;
}
```

A sorrendre érzékeny!

class PComplex : public Serialize, public Complex { ... }



Problémák

- Pointerek visszatöltésének nincs értelme.
- Veszélyes a kód: A öröklés sorrendjére érzékeny. A Serialize osztálynak kell elsőnek szerepelni.
- Mi van a virtuális függvények mutatóival?
- Külső reprezentáció nem hordozható.
- Ötletnek nem rossz, de a gyakorlatban használhatatlan!

Van megoldás?

- C++-ban nehéz automatizálni a serializációt. Néhány könyvtár ad támogatást (pl. boost, MFC, s11n)
- Fapados, de működő megoldás:
  - magára az objektumra kell bízni az adatfolyammá történő alakítást ill. visszaalakítást.
- Kellő körültekintéssel el kell készíteni a megfelelő virtuális függvényeket.

Használható megoldás

```
class Serializable {
public:
   virtual void write(ostream& os) const = 0; // kiíró
   virtual void read(istream& is) = 0;          // beolvasó
   virtual ~Serializable() {} // ne legyen probléma az upcast
};
class String {
protected:                  // hogy hozzáférjen a származtatott
   char *p;                 // van dinamikus adattag
   int len;
public:
   String(char *s = "" ) {
      len = strlen(s); p = new char[len+1]; strcpy(p, s);
   }
   .....
};
```

Használható megoldás

```
class PString : public Serializable, public String {
public:
   PString(char *p = "") : String(p) {}
   void write(ostream& os) const {
         os << len << ' ';          // separator
         os.write(p, len);          // a string kiírása
   }
   void read(istream& is) {
         delete[] p;
         (is >> len).ignore(1); // len beolv., eldobjuk a szepart.
         p = new char [len+1];// új terület kell
         is.read(p, len);           // len db karaktert olvasunk
         p[len] = 0;                // lezáró nulla
   }
};
```

Kérdések, megjegyzések

- Fontos, hogy a read() metódus pontos tükörképe legyen a write()-nak.
- Fontos a megfelelő reprezentáció kiválasztása.
- Kézenfekvő mindent szöveggé alakítani. Ekkor numerikus kiírások után kell szeparátor, amit a beolvasáskor el kell dobni.
- Pointerek és referenciák kezelése külön figyelmet igényel.
- Miért kell többszörös öröklés?
  - ha módosítható az osztály – megoldható többsz. nélkül
  - ha nincs kezünkben az osztály – csak ez a lehetőség