

## Programozás alapjai II. (9. ea) C++

többszörös öröklés, cast, perzisztencia, kivételek

Szeberényi Imre, Somogyi Péter  
BME IIT

<szebi@iit.bme.hu>



MŰEGYETEM 1782

## 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.

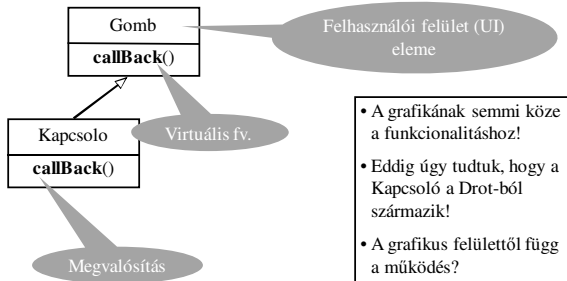
## Példa: Kapcsoló

Gombnyomásra ki-be lehessen kapcsolni

– A (grafikus) felhasználói felületen megvalósított **gomb** megkap minden felhasználói inputot. Amikor azt kapja, hogy "megnyomták", akkor szól a **kapcsolónak**.

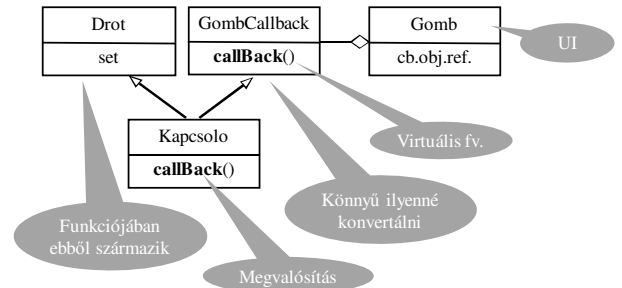
- Hogyan szól neki, ha nem ismeri?
- Próbálkozzunk származtatással!

## Kapcsoló a grafikus felületből?



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

## Kapcsoló többszörös örökléssel



## 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
    ....
};
  
```

GombCallback lehetne funktor mintájú, amitől talán elegánsabb a kód, de könnyen összeütközhet a modell függvényhívás operátorával.

## Kapcsoló megvalósítása/2

```

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

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

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

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_09](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_09) → nyomogomb

## Többsz. öröklés + fv. overload

```

struct A {
    void f() {}
    void f(int) {}
};

struct B {
    void f(double) {}
};

struct AB : public A, public B {
    using A::f;
    using B::f;
    void f() {
        f(1);
        f(1.2);
    }
};
  
```

Melyik f?

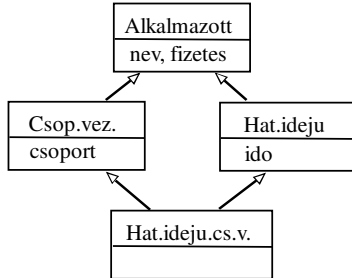
Feltételezés: A többszörös öröklésnél merőben eltérőek az alap-osztályok, az azonos nevű függvények más-más funkciót látnak el.

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_09](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_09) → polymorf

## Többszörös öröklés problémái

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

## Irodai hierarchia



## Irodai hierarchia /2

```

class Alkalmazott {
protected:
    String nev;           // név
    double fizetes;      // fizetés
public:
    Alkalmazott(String n, double fiz);
};
class CsopVez :public Alkalmazott {
    csop_t csoport;      // csoport azon.
public:
    CsopVez(String 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(String n, double f, time_t t)
        :Alkalmazott(n, f, ido(t) {}
};
class HatIdCsV :public CsopVez,
                public HatIdeju {
public:
    HatIdCsV(String n, double f, csop_t cs, time_t t)
        :CsopVez(n, f, cs), HatIdeju(n, f, t) {}
};
    
```



Két neve és két fizetése van ?

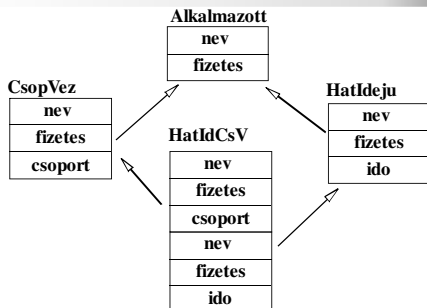
## Elérhetők ezek a mezők?

```

class HatIdCsV :public CsopVez,
                public HatIdeju {
public:
    HatIdCsV(String n, double f, csop_t cs, time_t 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



## Miből fakad a probléma ?

- 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 alapsztály

```
class CsopVez :virtual public Alkalmazott {
....
public:
    CsopVez(String n, double f, csop_t cs)
        :Alkalmazott(n, f), // csak lán végén
          csoport(cs) {}
};
class HatIdeju :virtual public Alkalmazott{ .... };
class HatIdCsV :public CsopVez, public HatIdeju {
public:
    HatIdCsV(String n, double f, csop_t cs, time_t t)
        :Alkalmazott (n, f), // csak ha a lán vége
          CsopVez(NULL, 0, cs), // tudjuk, hogy az ő konstr.
          HatIdeju(NULL, 0, t) {} // nem itt hívódik, ezért null
```

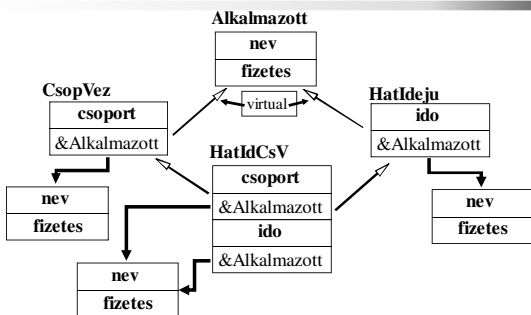
Csak az öröklési lán végén hívódik meg az alapsztály konstruktora.

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_09](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_09) → iroda

## Virtuális alapsztály

1. Alapsztály adatai nem épülnek be a származtatott osztály adataiba. A virtuális függvényekhez hasonlóan indirekt elérésük lesznek.
2. Az alapsztály konstruktora nem az első származtatott osztály konstruktora fogja hívni, hanem az öröklési lán végén szereplő osztály konstruktora.

## Memóriakép virt. alapsztállyal



## Irodai példa virt. alapsztállyal

```
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 lán 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 lánban.

## 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 alapsztállyá kell konvertálni (upcast)

## Konstruktor feladatai

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

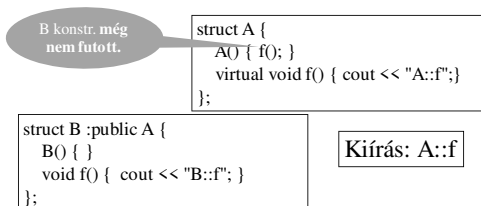
## Destruktor feladatai

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

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_09](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_09) → ctor\_dtor

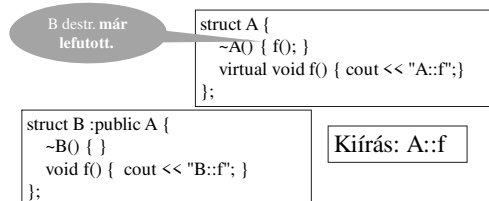
## Hívható-e konstruktorból saját virt. fv?

- Hívható, de nem az történik, amit várunk. Az alapsztály konstruktora a származtatott obj. konstruktora előtt fut le, így a virtuális függvénypointerek beállítása nem történik meg.



## Hívható-e destruktorból saját virt. fv?

- Hívható, de nem az történik, amit várunk. Az alapsztály destruktor a származtatott obj. destruktor után fut le, így a virtuális függvénypointerek már visszaálltak.



## Konstruktor/destruktor, mint őrsem

- Gyakori, hogy erőforrásként kell kezelni valamit (memória, fájl, eszköz, stb.):
  - lefoglalás
  - feldolgozás
  - felszabadítás
- Az ilyen esetekben külön figyelmet kell fordítani arra, hogy a feldolgozás közben észlelt hiba esetén is gondoskodjunk a felszabadításról.

## Konstruktor/destruktor, mint őrsem/2

```
{ // Kódrészlet
...
std::ifstream inp(f); // megnyitás
... // file feldolgozása
if (inp.fail())
    throw "baj_van"; // !!! nincs lezárva a fájl !!!
inp.close(); // lezárás normál esetben
}
```

A példa sántít, mert az ifstream a példány megszűnésekor bezárja a fájlt. Köszönöm Anonymus előadáson tett megjegyzését!  
Így azzal a hamis feltételezéssel nézzék a példát, mint a nem zárna be, vagy ifstream helyett valami eszközt (pl. hálózati) használunk, amit külön be kell zárni.  
**A lényeg a konstruktor/destruktor őrsem jellegű felhasználásában van.**

## Konstruktor/destruktor, mint őrsem/3

```
// Adapter, ami mindent tud, amit az ifstream + megszűnéskor bezár
struct Inputstream : std::ifstream {
    Inputstream(const char *f) :std::ifstream(f) {}
    ~Inputstream() { close(); }
};

{ // Kódrészlet
    Inputstream inp(f); // megnyitás
    ...                // file feldolgozása
}
```

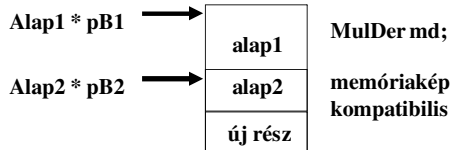
automatikusan  
bezár

destruktor megszüntet (bezár)

## 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 érték módosítás!
```

## Konv. többsz. öröklésnél szárm.-ra



```
Alap2 b2;
MulDer *pmd = (MulDer *) &b2;
// mutató érték módosítás!
// nem létező adatmezőket és üzeneteket el lehet érni
// explicit konverzió mindig veszélyes!
```

## Megbocsátható down cast

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

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

Ha MulDer\* típusú, akkor  
OK, egyébként NULL

## 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`, (típus), `tipus()`)

## dynamic\_cast

szintaxis: `dynamic_cast<T>(v)`

v – alapsztályra 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 – alapsztályra 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*>(apB);
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 sokkal könnyebben felfedezhetők.

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

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_09](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_09) → polimorf

## Explicit konstruktor (ism.)

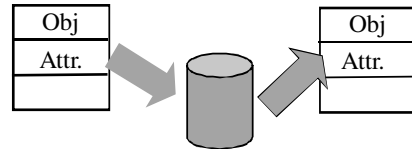
- Az egyparaméterű konstruktorok egyben automatikus konverziót is jelentenek:  
pl: `String a = "hello";` → `String a = String("hello");`
- Ez kényelmes, de zavaró is lehet:
  - tfh: van `String(int)` – konstruktor, ami megadja a string hosszát, de nincs `String(char)` konstruktor;
  - ekkor: `String 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, hogy nem ezt akarta, esetleg
                // elfelejtette megvalósítani az int-es változatot.
```

## Perzisztencia



- A perzisztens objektumok állapota **ementhető** 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 → szerializáció
- Az objektum a saját állapotát képes beolvasni egy adatfolyamból → deszerializáció
- Szempont lehet a hordozható külső formátum is. Ez különösen fontos elosztott rendszereknél.
- A perzisztenciát gyakran többszörös örökléssel vagy interfésszel oldják meg.

## Perzisztencia örökléssel 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 örökléssel 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("fl.dat");
    PComplex k1(1, 8);
    k1.write(f1);
    return 0;
}

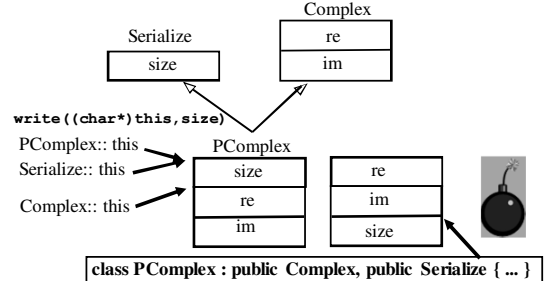
int main() {
    ifstream f1("fl.dat");
    PComplex k1;
    k1.read(f1);
    return 0;
}
```

Képes kiírni ill. visszatölteni

Származtatott mérete

## 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 szerializációt. Néhány könyvtár ad támogatást (pl. boost, MFC, s11n)
- Fapados, de működő megoldás:
  - Magára az objektumra kell bízni az adatfolyammá történő alakítást ill. visszaalakítást.
  - Ehhez célszerű egy olyan absztrakt osztályból származtatni, ami megfelel az elvárt interfésznek („arcnak”). Id: **Serializable**
- Kellő körültekintéssel el kell készíteni a megfelelő virtuális függvényeket.

## Használható megoldás

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

Szebb lenne protected nélkül. Hogyan? Pl. getter/setter, vagy inserter/extractor (kl. laboron)

## Használható megoldás /2

```
class PString : public Serializable, public String { Így két „arca” lesz
public:
    void write(ostream& os) const;
    void read(istream& is);
    ... // Milyen tagfüggvény kell még, ha azt akarjuk, hogy
        // PString String helyett használható legyen ?
};
```

- String összes létrehozási lehetősége → **Kell minden konstruktor**
- Kell másoló? Kell op=?
  - Van dinamikus adattagja a PStringnek? Nincs. → **Így nem kell.**
- Mi van, ha valamilyen String művelet eredményét PString-re kellene konvertálni?
  - Kell String → PString konverzió → **Konstruktorok megoldják**
- Mi van, ha valamilyen PString művelet eredményét String-re kellene konvertálni? → **Kompatibilitás megoldja**
- Destruktor? → **Jó az örökölt**

## Használható megoldás /3

```
class PString : public Serializable, public String {
public:
    PString() : String() {};
    PString(char ch) :String(ch) {}
    PString(const char *p) :String(p) {}
    PString(const String& s) :String(s) {}
    void write(ostream& os) const;
    void read(istream& is);
};

// Már csak a write és a read megvalósítása hiányzik!
```

## write és read megvalósítása

```
// Feltételezzük, hogy String::p és String::len protected tag,
// de később belátjuk, hogy nem is kell!

void PString::write(ostream& os) const {
    os << len << ' '; // szeparátor
    os.write(p, len); // a szöveg kiírása
}

void PString::read(istream& is) {
    delete[] p;
    (is >> len).ignore(1); // len és szeparátor beolvasás
    p = new char [len+1]; // új terület kell
    is.read(p, len); // len db karaktert olvasunk
    p[len] = 0; // lezáró nulla
}
```

## write és read megvalósítása /2

```
void PString::write(ostream& os) const {
    os << size() << '!'; // méret + szeparátor
    os.write(c_str(), size()); // a szöveg kiírása
    os << '\n'; // csak a könnyebb debug miatt
}

void PString::read(istream& is) {
    size_t len;
    (is >> len).ignore(1); // len és szeparátor beolvasása
    char *p = new char [len+1]; // új terület
    is.read(p, len).ignore(1); // len db karaktert olvasunk + a \n
    p[len] = 0; // lezáró nulla
    *this = String(p); // kihasználjuk, hogy az op= működik
    delete [] p; // p már nem kell
}

```

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_09](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_09) → pkomplex (a pstring házi feladat)

## Kérdések, megjegyzések

- Fontos, hogy a read() 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.
- Célszerű binárisan megnyitott adatfolyamot (stream) használni, így elkerülhetők a \n → \r\n konverzióból adódó problémák a különböző rendszerek között.
- 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

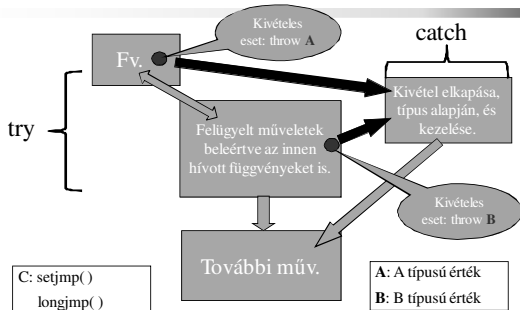
## Kivételes esetek kezelése (ism.)

- Kinek kell jelezni?
  - felhasználó, másik programozó, másik program
  - saját magunknak
- A kivételes eset kezelése gyakran nem annak keletkezési helyén történik. (Legtöbbször nem tudjuk, hogy mit kell tenni. Megállni, kiírni valami csúnyát, stb.)

## Kivétel kezelés (ism.)

- C++ típus orientált kivételkezelést támogat, amivel a kivételes esetek kezelésének szinte minden formája megvalósítható.
- A kivételkezeléshez tartozó tevékenységek:
  - figyelendő kódrészlet kijelölése (try)
  - kivétel továbbítása (throw)
  - esemény lekezelése (catch)

## Kivételkezelés = globális goto (ism)



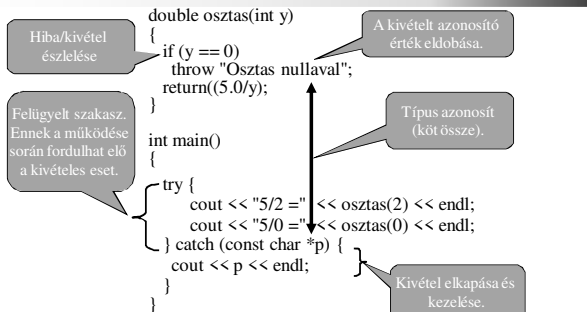
## Kivételkezelés/2 (ism)

```
try {
    .... Kritikus művelet1 pl. egy függvény hívása, aminek a
    belsejében: {if (hiba) throw kifejezés_típus1;}
    .... Kritikus művelet2 pl. egy másik függvény hívása,
    aminek a belsejében: {if (hiba) throw kifejezés_típus2;}
} catch (típus1 param) {
    .... Kivételkezelés1
} catch (típus2 param) {
    .... Kivételkezelés2
}

```

A hiba tetszőleges mélységben keletkezhet. Közvetlenül a try-catch blokkban a throw-nak nincs sok értelme, hiszen akkor már kezelni is tudnánk a hibát.

## Kivételkezelés példa (ism)



## Újdonságok a korábbiakhoz

- A dobott kivétel alap ill. származtatott objektum is lehet.

```

try {
    throw E();
} catch(H) {
    // mikor jut ide ?
}
    
```

- H és E azonos típusú,
- H bázisosztálya E-nek,
- H és E mutató és teljesül rájuk 1. vagy 2.,
- H és E referencia és teljesül rájuk 1. vagy 2.

## Következtetések

- Célszerű kivétel osztályokat alkalmazni, (pl. std::exceptions) amiből származtatással újabb kivételeket lehet létrehozni.
- A dobás értékparamétert dob, ezért az elkapáskor számolni kell az alaposztályra történő konverzióval (adatvesztés).
  - Célszerű pointert, vagy referenciát alkalmazni.
  - Kell másoló konstruktor.

## Továbbdobás

```

try {
    throw E();
} catch(H) {
    if (le_tudjuk kezelni) {
        ....
    } else {
        throw;
    }
}
    
```

Az eredeti dobódik tovább, nem csak az elkapott változat.

Paraméter nélkül

## Minden elkapása

```

try {
    throw E();
} catch(...) {
    // szükséges feladatok
    throw;
}
    
```

Minden kivétel

A kezelők sorrendje fontos!

## Rollback (stack unwinding)

```

try {
    A a;
    B b;
    C *cp = new C;
    if (hiba) throw "Baj van";
    delete cp;
} catch(const char *p) {
    // A létezik ?
    // B létezik ?
    // *cp által mutatott obj. létezik ?
}
    
```

Minden a blokkban deklarált, "létező" objektum destruktora meghívódik.

Hiba esetén C példánya nem szabadul fel, de cp megszűnik.

## Melyik obj. létezik ?

- Csak az az objektum számít létezőnek, amelynek a konstruktora lefutott.
- Ha a konstruktor nem fut le, akkor a rollback során a destruktorkor sem fog végrehajtódni.
- Előző példában C konstruktora lefutott ugyan, de nem deklarációval hoztuk létre, hanem dinamikusan.

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_9](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_9) → kodreszletek

## Kivétel a konstruktorban

- Lényegében a kivételkezelés az egyetlen mód arra, hogy a konstruktor hibát jelezzon.
- Hiba esetén gondoskodni kell a megfelelő obj. állapot előállításáról.

Inicializáló listán keletkező kivétel elfogása:

```
struct A {
    B b;
    A() try
        :b() { // konstruktor programozott része
    } catch (...) {
        ... // kivételkezelés
        // ha eljut idáig itt továbbdob
    }
};
```

## Kivétel a destruktorkorban

Destruktor hívás oka:

1. Normál meghívás
2. Kivételkezelés (rollback) miatti meghívás.  
Ekkor a kivétel nem léphet ki a destruktorkorból.

Destruktorban keletkező kivétel elfogása:

```
A::~~A() try {
    // destruktor programozott törzse
} catch (...) {
    ... // kivételkezelés
    // ha eljut idáig, továbbdob
}
```

## Kivételek specifikálása

- Függvény deklarációjakor/definíciójakor megadható, hogy milyen kivételeket generál az adott függvény.
  - Ha mást is generálna, akkor az automatikusan meghívja az `unexpected()` handler-t.
- ```
void f1() throw (E1, E2); // csak E1, E2
void f2() throw(); // semmi
void f3(); // bármi
void f3() noexcept(true) // semmi (C++11-től)
```
- ☞ C++17-től megszűnik a függvények din. kivétel specifikációja. Csak dob/nem dob.

## Kapcsolódó függvények

- `std::terminate_handler`  
`set_terminate(std::terminate_handler) throw();`
  - beállítja a terminate handler-t
- `void terminate();`
  - meghívja a terminate handler-t
- `bool uncaught_exception() throw();`
  - kivételkezelés folyamatban van, még nem talált rá a megfelelő handlerre (nem kapták el). Destruktorban lenne szerepe, de ...

☞ C++11-től a szignatúra változik

## Összefoglalás

- Többszörös öröklés: Ha egy osztálynak több „arcot” kell mutatnia.  
pl.
  - UI és modell kapcsolata
  - perzisztencia
- Sorosítás fontos eszköz a kommunikációban. C++-ban nehéz automatizálni, de egyedi megoldásokra van működő recept.
- Explicit cast veszélyes. Könnyebb felismerhetőséghez több eszközünk van (pl. `dynamic_cast`, `const_cast`, ...)
- Kivételes esetek kezelését átértékeljük az öröklés ismeretében