

Programozás alapjai II. (6. ea) C++

Mutatókonverziók, heterogén kollekció

Szeberényi Imre, Somogyi Péter
BME IIT

<szebi@it.bme.hu>

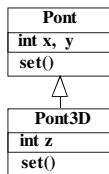


Öröklés (ismétlés)

- Egy osztályból olyan újabb osztályokat származtatunk, amelyek rendelkeznek az eredeti osztályban már definiált tulajdonságokkal és viselkedéssel.
- Analitikus – Korlátozó
- A tagfüggvények felüldefiniálhatók (override)
- virtuális függvény: hogy a tagfüggvény alaposztály felől (pointerén, referenciáján keresztül) is elérhető legyen.

Analitikus öröklés példa (ism.)

```
class Pont {
    int x, y;
public:
    Pont(int x1, int y1) :x(x1), y(y1) {}
    void set(int x1, int y1) {x = x1; y = y1;}
};
class Pont3D :public Pont {
    int z;
public:
    Pont3D(int x1, int y1, int z1)
        :Pont(x1, y1), z(z1) {}
    void set(int x1, int y1, int z1) {
        Pont::set(x1, y1); z = z1; }
};
```



Bővült

Korlátozó öröklés példa/1 (ism.)

Szeretnénk egy stack és egy queue osztályt:

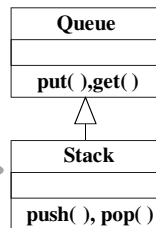
- mindkettő tároló
- nagyon hasonlítanak, de
- eltér az interfészüik:
 - put, get
 - push, pop
- önállóan vagy örökléssel ?

```
class Queue { ....
public:
    void put( int e );
    int get( );
};
```

```
class Stack { ....
public:
    void push( int e );
    int pop( );
};
```

Korlátozó öröklés példa/2 (ism.)

```
class Stack : private Queue { // privát: eltakar a külvilág felé
    size_t nelem;
public:
    Stack() : nelem(0) {}
    int pop() { nelem--; return(get()); }
    void push(int e) {
        put(e); // betesz
        for( size_t i = 0; i < nelem; i++ )
            put(get()); // átfogat
        nelem++;
    }
};
```

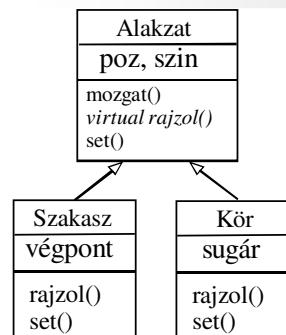


Továbbhívjuk a get()-et

Nem hatékony, csak példa!

Stack s1; s1.pop(); ~~s1.get()~~

Virtuális tagfüggvény (ism.)



rajzol() "átdefiniálása"
virtuális függvénynel

↑ késői
↓ kötés

set() átdefiniálása:
fv. override

**Alakzat::rajzol();
Mít hív ?**

Fontos C++ sajátosságok

- Konstruktorkor nem lehet virtuális
- Destruktor igen, és érdemes odafigyelni:
 - Ha alapsztályból dinamikus adattagot tartalmazó osztályt hozunk létre, majd ezt az alapsztály „felől” használjuk (töröljük).
- A konstruktorból hívott (saját) virtuális függvény **még nincs** átdefiniálva! A virt. táblát maga konstruktor tölti ki! (kötés)
 - absztrakt osztály esetén NULL pointer!

Inicializálás /1 (ism.)

```
class Pont {
protected:
  int x, y;
public:
  Pont(int x1 = 0, int y1 = 0) { x = x1; y = y1; }
};
class Pont3D :public Pont {
  int z;
public:
  Pont3D(int x1, int y1, int z1)
    { x = x1; y = y1; z = z1; }
};
```

Kell a default, mert

Mindig lehet így?

Alapsztály konstruktora mikor hívódik?

Inicializálás /2 (ism.)

```
class Pont {
protected:
  int x, y;
public:
  Pont(int x1, int y1) : x(x1), y(y1) { }
};
class Pont3D :public Pont {
  int z;
public:
  Pont3D(int x1, int y1, int z1) : Pont(x1, y1), z(z1)
    { }
};
```

Most nem kell paraméter nélküli, mert paraméterest hívunk.

Alapsztály konstruktora

Inicializálás /3 (ism.)

```
class FixKor :public Pont {
  double& r;
  const double PI;
  int x, y;
public:
  Kor(int x, int y, double& r) :x(x), y(y), r(r), PI(3.14) { }
};
```

Kötelező inicializálni!

Melyik y? Van már this?

```
class FixKor :public Pont {
  double& r;
  static const double PI;
  ...
};
const double Kor::PI = 3.14; // statikus tag, létre kell hozni
```

Explicit konstruktor

- Az egyparaméteres 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 String {
  char *p;
  size_t len;
public:
  String(const char *s = "");
  explicit String(int);
  virtual ~String( );
  ...
};
```

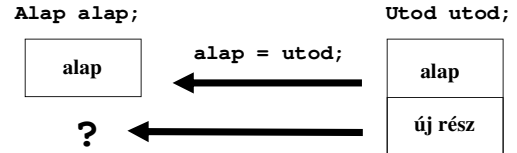
Nincs automatikus konverzió

Öröklés és polimorfizmus

```
struct A {
    void valami() { cout << "A valami" << endl; }
    void semmi() { cout << "A semmi" << endl; }
};
struct B: public A{
    void valami() { cout << "B valami" << endl; }
    void valami(int) { cout << "B valami int" << endl; }
};
...
B b;
b.valami();           // B valami
b.valami(1);         // B valami(int)
b.semmi();           // A semmi
b.A::valami();       // A valami
b.A::valami(int)    // HIBA
```

Értékadás és kompatibilitás

```
struct Alap { int a; void f(); };
struct Utod : Alap { double d; int f1(); };
```



A kompatibilitás miatt az értékadás formálisan rendben, de az új résznek nincs helye a memóriában. **Elveszik. Szeletelő** (slicing) történik.

Mutatókonverzió és kompatibilitás

```
struct Alap { int a; void f(); };
struct Utod : Alap { double d; int f1(); };
```

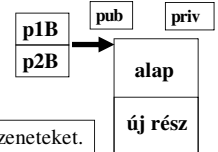


`Alap* p = &utod;` ✓

Memóriakép rendben van, de mi a helyzet a viselkedéssel?

Konverzió alapsztály mutatóra

```
struct Alap { void f(); };
struct Pub : public Alap { void f(); };
struct Priv : private Alap { void f(); };
Alap *p1B, *p2B;
```



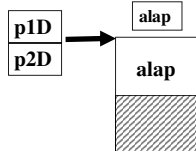
Pub pub; // pub kaphat Alap-nak szóló üzeneteket.
p1B = &pub; // nem kell explicit típuskonverzió
p1B->f() // alap o. f() elérhető

Priv priv; // priv nem érti Alap üzeneteit pl: priv.Alap::f()
p2B = (Alap*)&priv; // explicit konverzió kell
p2B->f() // így már érti

Viselkedés és a memóriakép is kompatibilis.

Konverzió származtatott o. mutatóra

```
struct Alap { void f(); };
struct Pub : public Alap { void f(); };
struct Priv : private Alap { int a; };
Alap alap;
Pub* p1D; Priv* p2D;
```



```
p1D = (Pub*)&alap;
p1D->f(); // ?????
p2D = (Priv*)&alap;
p2D->a = 0 // ?????
```

Explicit konverzióval **nem** létező adatmezőket és függvényeket is el lehet érni! Ne használjuk! Veszélyes!

Viselkedés és a memóriakép SEM kompatibilis.

Típuskonverzió és az öröklés

- Kompatibilitás esetén a konverzió automatikus.
- A másik irányba (alaptól származtatottra) explicit módon ki lehet kényszeríteni, de a legtöbb esetben értelmetlen és veszélyes!
- Típuskonverzió = objektumkonverzió
- Mutatókonverzió = rejtett objektumkonverzió
- Kompatibilitás:
 - kompatibilis memóriakép
 - kompatibilis viselkedés (tagfüggvények)

Függv. elérése alap. o. mutatóval

```
class Alakzat { ... virtual void rajz() = 0; void k(); };
class Szakasz : public Alakzat { void rajz(); void k(); };
class Kor : public Alakzat { void rajz(); void k();... };
Alakzat* tar[100];
tar[0] = new Szakasz(...); // konverzió, (kompatibilis)
tar[1] = new Kor(...);     // konverzió, (kompatibilis)
....
```

```
for (int i = 0; i < 100; i++) {
    tar[i] ->rajz(); tar[i]->k();
}
```

Származtatott o. fv.

Alap oszt. függvénye

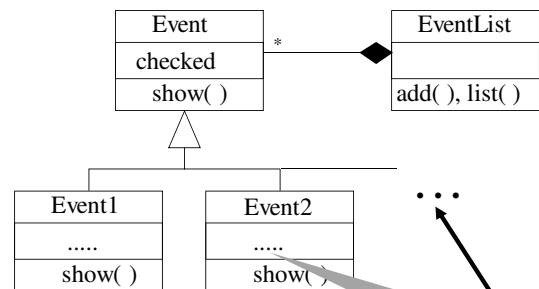
Heterogén gyűjtemények

- Különböző típusú objektumokat egy **közös** gyűjteménybe tesszük
- Egységes kezelés: valamilyen viselkedési kompatibilitásra építve
 - egy öröklési hierarchiából származó objektumokat tehetünk heterogén szerkezetbe
 - kompatibilitásból származó előnyök kihasználása

Heterogén kollekciónak példa

- Egy rendszer eseményeit kell naplózni.
- Az események egymástól eltérő adattartamúak, és esetleg új események is lesznek, amit még nem ismerünk.
- Események sorrendje fontos, ezért célszerűen egy tárolóban kell lenniük.
- Az eseménynapló megnézésekor meg kell mutatni azt is, hogy mely eseményeket néztük meg már korábban.

Eseménynapló osztályai



Még nem ismerjük, de nem is kell!

Esemény és leszármazottai

```
class Event {
    bool checked;
public:
    Event() :checked(false) {}
    virtual void show() { cout<<" Checked: ";
        cout<<checked<<endl; checked = true; }
    virtual ~Event() {}
};

class Event1 :public Event {
    ...
public:
    Event1();
    void show () { cout << ..... ;
        Event::show();
    }
};
```

Hurok ?

Eseménylista: pointerok tárolója

```
class EventList {
    size_t nevent;
    Event* events[100];
public:
    EventList() : nevent(0) {}
    void add(Event* e) {
        if (nevent < 100) events[nevent++] = e;
    }
    void list() {
        for (size_t i = 0; i < nevent; i++) events[i]->show();
    }
    ~EventList() {
        for (size_t i = 0; i < nevent; i++) delete events[i];
    }
};
```

Alaposztályi pointerokat tárolunk

Származtatott osztály függvénye

Megszűnik az esemény is (komponens reláció)

Eseménynapló használata

```
class Event { ... };
class Event1 :public Event { ... };
class Event2 :public Event { ... };
class EventList { ... };
...
EventList list;
list.add(new Event1(...));
...
list.add(new Event2(...));
...
list.add(new Event9(...));

list.list();
```

Új esemény:
csupán definiálni
kell az új osztályt

Ezzel a list-re bízunk az objektumot, hiszen a pointerét nem jegyezzük meg.

Ki szabadít fel?

```
class EventList {
    size_t nevent;
    Event* events[100];
public:
    EventList() : nevent(0) {}
    void add(Event* e) {
        if (nevent < 100) events[nevent++] = e;
        else { delete e; throw „nem fért be”; } }
    void list() {
        for (size_t i = 0; i < nevent; i++) events[i]->show(); }
    ~EventList() {
        for (size_t i = 0; i < nevent; i++)
            delete events[i]; // -> ~Event();
    }
}; ...
list.add(new Event1(...));
```

Virtuális kell!

Virtuális destruktor újból

```
class Event {
public:
    virtual void show() {}
    virtual ~Event() {} 2
};
```

```
class Event1 :public Event {
    int *p;
public:
    Event1(int s) { p = new int[s]; }
    void show() {}
    ~Event1() { delete[] p; } 1
};
```

```
Event *ep = new Event1(120);
ep->show();
delete ep;
```

Virt. destr. más, mint a többi virt. fv., mert az ősz destruktorra mindig meghívódik!

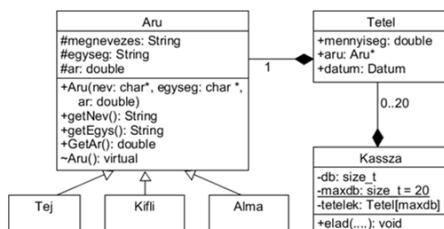
https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_06 → virt_destruktor2

Összetettebb példa: CppBolt

- Pénztárgépet modellezünk: A pénztáros megadja az eladott mennyiséget és az árut.
- A gép nyilvántartást vezet. Lekérdezhető a napi/összes eladás, napi bevétel, stb.
- Ötlet: heterogén kollekción:
 - Közös attr.: eladott mennyiség, dátum, (összeg)
- Probléma: minden áru ebből származzon?
 - Nem célszerű
- Megoldás (1): közbenső osztály az áruk valós ösze (pointerre) fölé.

CppBolt

Csomagoló osztály: Tetel, ami Aru pointereket tárol.
Aru az őse a „valós” áruknak. A Kassza pedig a tároló



http://svn.iit.bme.hu/proga2/cporta_peldak/CppBolt/

Heterogén kollekción összefoglalás

- Különböző típusú objektumokat egy közös gyűjteménybe tesszük.
- Kihasználjuk az öröklésből adódó kompatibilitást.
- Nagyon gyakran alkalmazzuk
 - könnyen bővíthető, módosítható, karbantartható
- Rossz alkalmazásánál: slicing!!!

Tipikus halálfejes hiba

```
class EventList {
public:
    int nevent;
    Event events[100];
    EventList() { nevent = 0; }
    void add(Event& e) { if (nevent < 100)
        events[nevent++] = e; }
    void list() {
        for (int i = 0; i < nevent; i++)
            events[i].show();
    }
};
```

Nem pointert tárol!

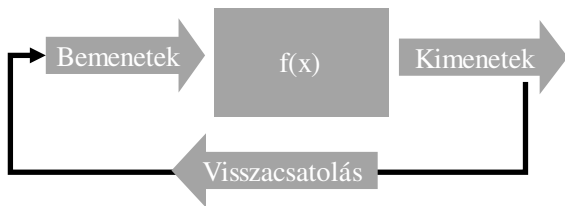
Event::show()

Adatvesztés!
Szeletelés (slicing)
A származtatott rész elveszik!

Digitális áramkör modellezése

- Digitális jel: üzenet (objektum)
- Áramköri elemek: objektumok
 - bemenet, kimenet, viselkedés (f(x))
 - kapcsoló, kapu, drót, forrás, csomópont
- Objektumok a valós jelterjedésnek megfelelően egymáshoz kapcsolódnak. (üzennek egymásnak)
- Visszacsatolás megengedett.

Modell

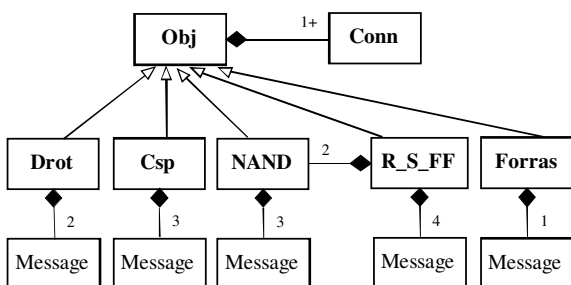


A változásokat üzenetek továbbítják. Ha nincs változás, nem küldünk újabb üzenetet.
Csak véges számú iterációt engedünk meg.

Áramköri elemek felelőssége

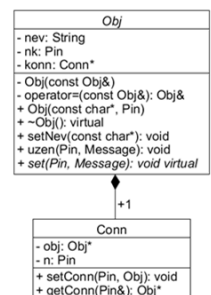
- Kapcsolatok (bemenet/kimenet) kialakítása, nyilvántartása.
- Bejövő üzenetek tárolása összehasonlítás céljából.
- Válaszüzenetek előállítás és továbbítása a bejövő üzeneteknek és a működésnek megfelelően.

Osztályhierarchia



Obj: alapsztály

- Minden áramköri elem ebből származik
- Felelőssége:
 - az objektumok közötti kapcsolatok leírása (a *Conn* osztály dinamikus tömbje)
 - kapcsolatokon keresztül az üzenetek (*Message* objektum) továbbítása,
 - a működést (viselkedést) megvalósító függvény elérése (a *set* virtuális függvényen keresztül).



Obj: absztrakt alapsztály

```
class Obj {
    String nev;           // objektum neve
    Pin nk;              // kapcsolódási pontok száma
    Conn *konn;          // kapcsolatok leírása
    Obj(const Obj&);     // hogy ne lehessen használni
    Obj& operator=(const Obj&); // hogy ne lehessen haszn.
public:
    Obj(const char *n, Pin k) : nev(n) {
        konn = new Conn[nk = k]; }
    virtual ~Obj() { delete[] konn; } // tömb felszab.
    void setNev(const char *n) { nev =String(n); } // név beáll.
    void setConn(Pin k, Obj& o, Pin on); // összekapcs.
    void uzen(Pin k, Message msg); // üzen
    virtual void set(Pin n, Message msg) = 0; //működött
};
```

Conn: kapcsolatok tárolása

- Egy objektumkapcsolatot leíró osztály
- Példányaiból felépített dinamikus tömb (*Obj::konn*) írja le egy objektum összes kapcsolatát

Miért nem referencia ?

```
class Conn {
    Obj *obj; // ezen objektumhoz kapcsolódik
    Pin n; // erre a pontra
public:
    Conn() :obj(NULL) {}
    void setConn(Pin k, Obj& o) { n = k; obj = &o; } // beállít
    Obj *getConn(Pin& k) { k = n; return(obj); } // lekérdez
};
```

Message: jel mint üzenet

- Digitális jelet reprezentáló osztály
 - undef, jel 0 és jel 1 értéke van.
- A végtelen iteráció elkerülése végett a jelszint mellett egy iterációs számláló is van.
- Megvalósítása struktúrával, mivel az adattakarás csak nehezítene.
- Műveletei:

```
msg1 == msg2      msg1 != msg2
msg1 + msg2      --msg
```

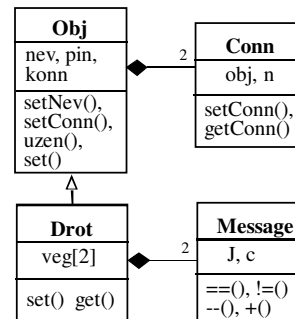
Message: jel mint üzenet /2

```
struct Message {
    enum msgt { undef, jel } typ; // típus
    bool J; // jelszint 0 v. 1
    int c; // iterációs számláló
    Message(msgt t = undef, bool j = false, int n = 20)
        :typ(t), J(j), c(n) {}
    // két üzenet egyenlő, ha az típusuk és jelszintjük is azonos
    bool operator==(const Message& m) const {
        return(typ == m.typ && J == m.J); }
    bool operator!=(const Message& m) const {
        return(!operator==(m)); }
    Message operator+(const Message &m) const {
        return Message(std::max(typ, m.typ, J+m.J, std::max(c,m.c)); }
    Message& operator--() { // pre-dekremens op.
        if (--c <= 0) throw "Sok Iteracio!";
        return(*this); }
};
```

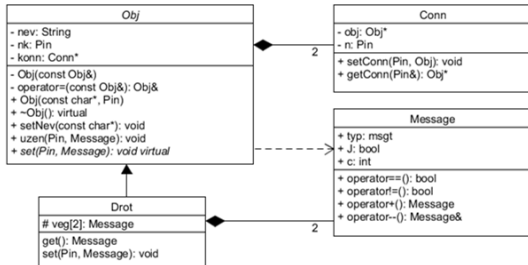
Üzenet továbbítása

```
/**
 * Üzenet (msg) küldése a k. pontra kapcsolódó obj. felé
 */
void Obj::uzen(Pin k, Message msg) {
    Pin n; // kapcsolódó objektum kapcs. pontja
    if (k >= nk)
        throw "Uzenet hiba"; // hiba, nincs ilyen végpont
    if (Obj *o = konn[k].GetConn(n)) {
        o->set(n, --msg); // szomszéd működtető függvénye
    }
}
```

Drót obj. modellje



Drót kicsit precízebben



A diagram szerkesztéséhez az UMLet (<http://www.umlet.com/>) programot használtam.

Drót

```

class Drot :public Obj {
protected:
    // megengedjük a származtatottnak
    Message veg[2]; // két vége van, itt tároljuk az üzeneteket
public:
    Drot(const char *n = "") : Obj(n, 2) {} // 2 végű obj. Létrehozása
    Message get() const { return veg[0] + veg[1]; } //bármelyik vég
    void set(Pin n, Message msg); // működtet
};
void Drot::set(Pin n, Message msg) {
    if (veg[n] != msg) { // ha változott
        veg[n] = msg; // megjegyezzük és
        uzen(n^1, msg); // elküldjük a másik végére (vezet)
    }
}

```

Gonosz trükk!

Csomópont

```

class Csp :public Obj {
protected:
    // megengedjük a származtatottnak
    Message veg[3]; // három vége van, itt tároljuk az üzeneteket
public:
    Csp(const char *n = "") : Obj(n, 3) {} // 3 végű objektum
    void set(Pin n, Message msg); // működtet
};
void Csp::set(Pin n, Message msg) {
    if (veg[n] != msg) { // ha változott
        veg[n] = msg; // megjegyezzük és
        uzen((n+1)%3, msg); // elküldjük a másik 2 végére
        uzen((n+2)%3, msg);
    }
}

```

Kapcsoló

```

class Kapcsolo :public Drot { // Drótból
    bool be; // állapot
public:
    Kapcsolo(const char *n = "") : Drot(n, be(false) {}
    void set(Pin n, Message msg); // jel, false, lehetne undef
    void kikap() { be = false; uzen(0, Message(Message::jel));
        uzen(1, Message(Message::jel)); }
    void bekap() { be = true; uzen(0, veg[1]); uzen(1, veg[0]); }
};
void Kapcsolo::set(Pin n, Message msg) {
    if (be) Drot::set(n, msg); // be van kapcsolva, drótként viselk.
    else veg[n] = msg; // ki van kapcsolva, csak megjegyezzük
}

```



NAND kapu

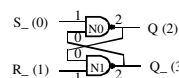
```

class NAND :public Obj {
    Message veg[3]; // három "vége" van
public:
    NAND(const char *n = "") : Obj(n, 3) {} // 3 végű obj. létreh.
    void set(Pin n, Message msg); // működtet
    Message get() { return(veg[2]); } // kim. lekérdezése
};
void NAND::set(Pin n, Message msg) {
    if (n != 2 && veg[n] != msg) { // ha változott bemenet
        veg[n] = msg; // megjegyezzük
        uzen(2, veg[2] = Message(Message::jel,
            !(veg[0].J * veg[1].J), msg.c)); // üzenünk a kimeneten
    }
}

```

kimenet előállítás

ciklusszám marad



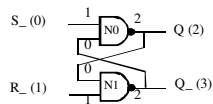
R_S tároló

```

Class R_S_FF :public Obj {
protected:
    Message veg[4]; // négy "vége" van
    NAND N[2]; // két db NAND kapu, komponens
public:
    R_S_FF(const char *n) : Obj(n, 4) {
        N[0].setConn(2, N[1], 0); // összekötések létrehozása
        N[1].setConn(2, N[0], 0);
    }
    void set(Pin n, Message msg); // működtet
    Message get(int i) { // kimenet lekérdezése
        if (i >= 2) i = 0; return(veg[i+2]);
    }
};

```


R_S tároló /2



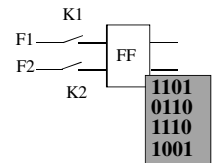
```
Void R_S_FF::set(Pin n, Message msg) {  
    if (n < 2 && veg[n] != msg) { // ha input és változott,  
        veg[n] = msg; // letárolja  
        N[n].set(1, msg); // megfelelő bemenetre küldi  
        uzen(2, veg[2] = N[0].get()); // üzen a kimeneten  
        uzen(3, veg[3] = N[1].get()); // üzen a kimeneten  
    }  
}
```

Kimenetek előállítás, mert belül nincs csomópont.

Szimulátorunk próbája

```
Kapcsoló K1("K1"), K2("K2");  
Forras F1("F1"), F2("F2"); R_S_FF FF("FF");
```

```
try {  
    F1.setConn(0, K1, 0); FF.setConn(0, K1, 1);  
    F2.setConn(0, K2, 0); FF.setConn(1, K2, 1);  
  
    F1.init(); F2.init();  
    K1.bekap(); K2.bekap();  
    cerr << FF.get(0).J << FF.get(1).J << FF.get(2).J << FF.get(3).J << endl;  
    K1.kikap();  
    cerr << FF.get(0).J << FF.get(1).J << FF.get(2).J << FF.get(3).J << endl;  
    K1.bekap();  
    cerr << FF.get(0).J << FF.get(1).J << FF.get(2).J << FF.get(3).J << endl;  
    K2.kikap();  
    cerr << FF.get(0).J << FF.get(1).J << FF.get(2).J << FF.get(3).J << endl;  
} catch (const char *s) { cerr << s << endl; }
```



https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_05 → digit

Összefoglalás

Öröklés

- újrafelhasználhatóság
- kompatibilitás
- heterogén kollekció
- pointer konverzió
- adatvesztés
- virtuális tagfüggvények
- absztrakt alapsztály