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

OO paradigmák, osztály, operátorok átdefiniálása

Szeberényi Imre, Somogyi Péter
BME IIT

<szebi@iit.bme.hu>

MŰEGYETEM 1782

Programfejlesztés

- Feladatanalízis
  - világ = dolgok + tevékenységek
- Modellezés
- Tervezés
  - absztrakció (elvonatkoztatás a részletektől)
  - dekompozíció (részfeladatra bontás)
- Implementáció (programozás)
  - program = adatstruktúrák + algoritmusok

Néhány programozási módszer

- Korai szoftverkészítés
- Strukturált
- Moduláris
- Objektum-orientált
- Funkcionális
- Deklaratív
- Adatfolyam-orientált
- Aspektus-orientált
- ...

Korai szoftverkészítés jellemzői

- többnyire gépi nyelvek
- nehezen követhető
- nehezen módosítható
- nincsenek letisztult vezérlési szerkezetek
  - ciklusba nem illik beugrani
- zsenigyanús programozók
- pótolhatatlan emberek, nem dokumentált
- szoftverkrízis kezdete (1968)

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports

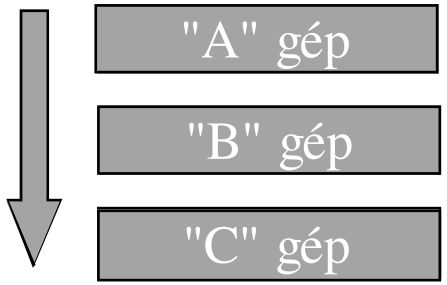
Gépi nyelv ?

```
// Kiírunk egy stringet void print(String str)
00401350   push        ebp
00401351   mov         ebp,esp
00401353   sub         esp,40h
00401356   push        ebx
00401357   push        esi
00401358   push        edi
00401359   lea         edi,[ebp-40h]
0040135C   mov         ecx,10h
00401361   mov         eax,0CCCCCCCCh
00401366   rep stos    dword ptr [edi]
00401368   mov         eax,dword ptr [ebp+8]
0040136B   push        eax
0040136C   push        offset string "%s" (0042201c)
00401371   call        printf (004037d0)
00401376   add         esp,8
00401379   pop         edi
0040137A   pop         esi
0040137B   pop         ebx
0040137C   add         esp,40h
0040137F   cmp         ebp,esp
00401381   call        __chkesp (00403620)
00401386   mov         esp,ebp
00401388   pop         ebp
00401389   ret
```

```
00401350  55 8B EC 83 EC 40 53 56  U‹ě.ě@SV
00401358  57 8D 7D C0 B9 10 00 00  WŤ}ŔĄ...
00401360  00 B8 CC CC CC CC F3 AB  .¸ĚĚĚĚó«
00401368  8B 45 08 50 68 1C 20 42  ‹E.Ph. B
00401370  00 E8 5A 24 00 00 83 C4  .čZ$...Ä
00401378  08 5F 5E 5B 83 C4 40 3B  ._^[.Ä@;
00401380  EC E8 9A 22 00 00 8B E5  ěčš"..‹Í
00401388  5D C3 CC CC CC CC CC CC  ]ĂĚĚĚĚĚĚ
```

Strukturált tervezés

- "oldd meg a feladatot" -> "gépen futó pr."
  (E.W.Dijkstra, C.A.Hoare)
- fokozatos finomítás
- absztrakt gépek rétegei



  - absztrakció:
    - részletektől való elvonatkoztatás, hasonlóságok felismerése, ábrázolás, műveletvégzés, axiómák felállítása
  - dekompozíció:
    - részekre bontás, egymástól függetlenül kezelhető kisebb feladatok elhatárolása, határfelületen "látható" viselkedések meghatározása

Strukturált tervezés /2

- strukturált adatok, tipizálás
- strukturált műveletek, tipizálás
- előnyök:
  - áttekinthetőbb, minden réteghez önálló döntések,
  - hordozhatóság
- hátrányok:
  - adatstruktúrákat nagyon pontosan kell definiálni a magasabb absztrakciós szinteken is,
  - hatékonysági problémák
- PASCAL nyelv (blokkok fa struktúrája)

Moduláris tervezés

- modul: önálló egység meghatározott kapcsolódási felülettel (interface)
- cserélhető
- önállóan fordítható
- önállóan tesztelhető
- információ elrejtése
- funkcionális megközelítés
- modulban a belső kötés erős
- modulok között a kötés gyenge

Moduláris tervezés /2

- egy adatszerkezeten egy funkció
- előnyök:
  - funkcionális bontás magától értetődő
  - interfészek jól kézben tarthatók
- hátrányok:
  - esetenként több példány az elrejtés miatt
  - az adatok megjelennek az interfészeken, így azok "kőbe" lettek vésve
- FORTRAN, MODULA-2, ADA

Dekompozíció

- Felbontás egyszerűbb részfeladatokra
- A felbontás absztrakt, ha
  - a felbontás anélkül történik, hogy a részeket pontosan meg kellene oldani, vagy meg kellene érteni;
  - csak a felület megadására szorítkozik (a kapcsolódáshoz);
  - a részletek megadását elodázza

Funkcionális dekompozíció

- Mit csinál a rendszer?
  - Strukturáló szempont: tevékenység
- Tevékenység: résztevékenységekre bontunk
  - absztrakt: mit csinál a résztevékenység anélkül, hogy kellene tudni, hogy hogyan csinálja
- Adatok: résztevékenységek ki-bemenete
  - nem absztrakt, mert tudnunk kell a pontos adatszerkezetet

Feladat: komplex számok

- Olvassunk be 10 komplex számot és írjuk ki a számokat és abszolút értéküket fordított sorrendben!
- Funkcionális dekompozíciónál az adatokon végzett tevékenységekre koncentrálunk:

Tevékenység	Adat
beolvasás() és tárolás	Komplex, KomplexTömb
kiírás()	Komplex, KomplexTömb
abs()	Komplex

Funkcionális dekompozícióval

```
struct Komplex {
  double re, im;
};

int main() {
  Komplex t[10];  // adatok
  beolvasas(t);   // művelet
  kiiras(t);      // művelet
  return 0;
}
```

Funkcionális dekompozícióval/2

```
double abs(Komplex k){//adatot ismerni kell
  return sqrt(k.re*k.re + k.im*k.im);
}
void beolvasas(Komplex t[]){//ismerni kell
for (int i=0; i<10; i++)
  cin >> t[i].re >> t[i].im;
}
void kiiras(Komplex t[]) {//ismerni kell
for (int i=9; i>=0; i--)
  cout << t[i].re << '+' << t[i].im << 'j'
       << abs(t[i]) << endl;
}
```

Kőbe vésett adatszerkezet

- Ahhoz, hogy dekompozíció során nyert funkciók megvalósíthatók legyenek, rögzíteni kell a funkciók által kezelt adatok formátumát, struktúráját.
  - pl. el kell dönteni, hogy tömböt használunk, melynek a szerkezetét pontosan meg kell adni.
- Nehezen módosítható (pl. átállás polár koordinátákra)
- Nehezen használható fel újra.
- Az adat nem absztrakt

Absztrakt adattípus

Az adat matematikai modellje

- viselkedésre koncentrálunk (viselkedési osztály)
- értékkészlet és az azon értelmezett
- a művelet halmaz a lényeges
- művelet: leképezés az értelmezési tartomány és az értékkészlet között
- a művelek algebrai leírással megadhatók
- nem kell ismerni a megvalósítást, azt sem, hogy mi a konkrét adat, csak a műveleteket
- egy adaton több funkció
- pl: komplex, verem, sor, tömb, lista, fa, stb.

Objektum

- Az OBJEKTUM testesíti meg a konkrét adatot és a rajta végezhető műveleteket
- egyedileg azonosítható
- viselkedéssel és állapottal jellemezhető
- felelőssége és jogköre van
- képes kommunikálni más objektumokkal
- a belső adatszerkezet, és a műveleteket megvalósító algoritmus rejtve marad
- könnyen módosítható
- újrafelhasználható
- általánosítható

Objektum orientált dekompozíció

- Kik a probléma szereplői?
  - Strukturáló szempont: dolgok (alany, adatok)
- Dekompozíció: szereplőkre (objektumokra) bontunk
- Adat:
  - absztrakt: a belső szerkezetet eltakarjuk
- Tevékenységek: műveletek a szereplőkön (ige)
  - absztrakt: nem kell tudni, hogy hogyan működik.

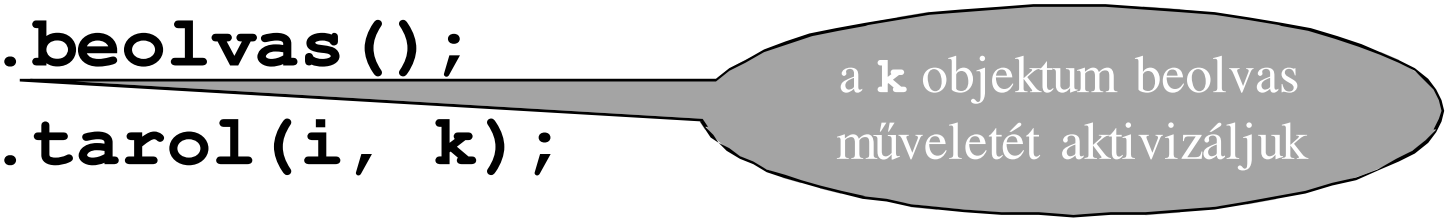
A feladat OO dekompozícióval

- Olvassunk be 10 komplex számot és írjuk ki a számokat és abszolút értéküket fordított sorrendben!
- Objektum orientált dekompozíció használatakor az absztrakt adatra koncentrálunk:

Szereplő (objektum)	Művelet (üzenet)
Komplex	beolvas(), kiir() abs()
KomplexTar	tarol() elovesz()

A feladat OO dekompozícióval/2

```
Komplex k;    // beolvas, kiir, abs
KomplexTar t;// tarol, elovesz
for (int i = 0; i < 10; i++) {
   k.beolvas();
   t.tarol(i, k);
}
for (int i = 9; i >= 0; i--) {
   k = t.elovesz(i); k.kiir();
   cout << ' ' << k.abs() << endl;
}
```

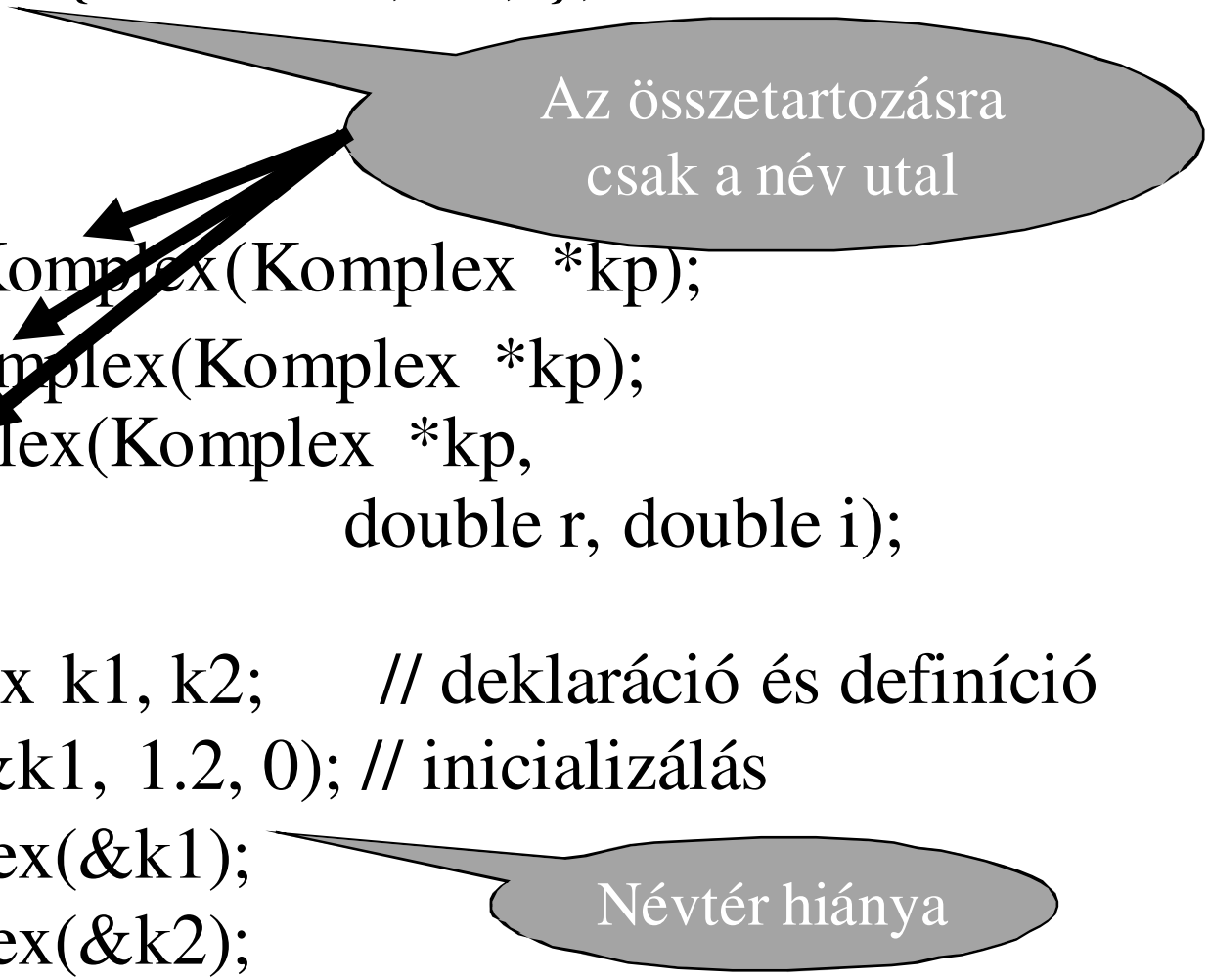


Objektum orientált modell

- az objektumok jelentik a valóság és a modell kapcsolatát
- együttműködő objektumok
- megvalósítás: objektumokat „szimuláló” programegységekkel

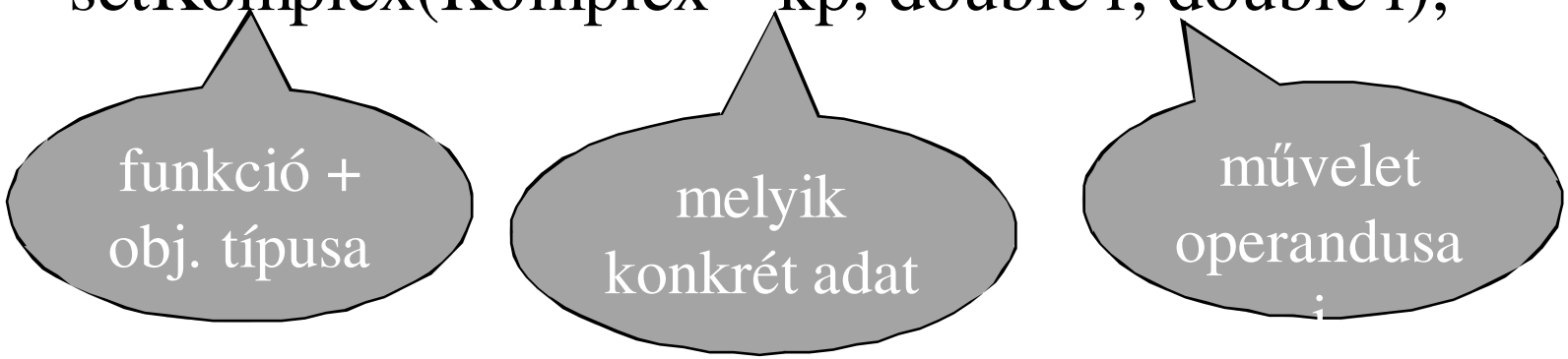
Komplex obj. megvalósítása C-ben

```
struct Komplex { double re, im; };

void beolvasKomplex(Komplex *kp);
double absKomplex(Komplex *kp);
void setKomplex(Komplex *kp,
                double r, double i);

struct Komplex k1, k2;      // deklaráció és definíció
setKomplex(&k1, 1.2, 0); // inicializálás
f = absKomplex(&k1);
f = absKomplex(&k2);
```



Interfész függvények paraméterei

setKomplex(Komplex *kp, double r, double i);



void beolvasKomplex(Komplex *kp);

double absKomplex(Komplex *kp);

Ilyen paraméterezést használtunk a laborban a String esetében is.

OO paradigmák

- egységbezárás (encapsulation)
  - osztályok (adatszerkezet, műveletek egységbezárása)
- többarcúság (polymorphism)
  - műveletek paraméter függőek, tárgy függőek (kötés)
- példányosítás (instantiation)
- öröklés (inheritance)
- generikus adatszerkezetek és algoritmusok

Egységbezárás C++-ban

```
struct Komplex {
    double re, im;
    void set(double r, double i);
    double abs();
};
Komplex k1, k2;
k1.re = 1.2; k1.im = 0;
k1.set(1.2, 0);
f = k1.abs();
```

adatok

tagfüggvények

A fv. névben elég a funkciót jelölni.
A saját adatot sem kell átadni.

```
setKomplex(&k1, 1.2, 0);
```

k1, k2 objektum: adatok és a rajta végezhető műveletek

Adattakarás C++-ban

```
struct Komplex {
 private:
   double re, im;
 public:
   void set(double r, double i);
   double abs();
};
Komplex k1;
k1.re = 1.2; k1.im = 0;
```

privát adatok

nyilvános tagfüggvények

Közvetlen hozzáférés
a priváthoz TILOS

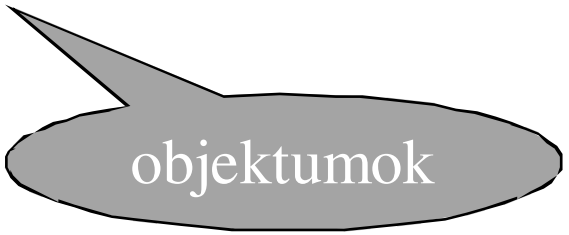
CSAK ÍGY

```
k1.set(1.2, 0);
f = k1.abs();
```

Osztály

- Objektum osztály ≡ objektum fajta, típus (viselkedési osztály)
- Osztály ≠ Objektum
- Objektum ≡ Egy viselkedési osztály egy konkrét példánya.

Komplex k1, k2, k3;



C++-ban a struct egy osztály !

Adatelérés megvalósítása

C++

```
class Komplex {
    double re, im;
  public:
    void set(double r, double i) { re = r; im = i; }
};
Komplex k1;        k1.set(1.2, 3.4);
```

C

```
struct Komplex { double re, im; };
void setKomplex(struct Komplex *this, double r, double i) {
        this -> re = r;
        this -> im = i;
}
struct Komplex k1;        set(&k1, 1.2, 3.4);
```

a konkrét objektumra mutat

this pointer ≡ példányra mutató ptr.

```
class Komplex {
    double re, im;
  public:
    void set(double re, double im)    {
          this->re = re; this->im = im;
    }
    ......
};
double Komplex::abs() {
    return sqrt(this->re * this->re + this->im * this->im);
}
Komplex k1, k2; double f = k1.abs();
```

k2.set(2,1, -4);

*this azt az objektumot jelenti,
amelyre a tagfüggvényt meghívták.

Kívül és belül definiált tagfvüggvény

```
class Komplex {
   double re, im;
 public:
   void set(double r, double i)  { re = r; im = i; }
   double abs();
};
double Komplex::abs()  { return sqrt(re*re+im*im);  }
int main() {
   Komplex k1; k1.set(1.2, 3.4);
   cout << k1.abs();
}
```

adatok privátak

inline-nak megfelelő

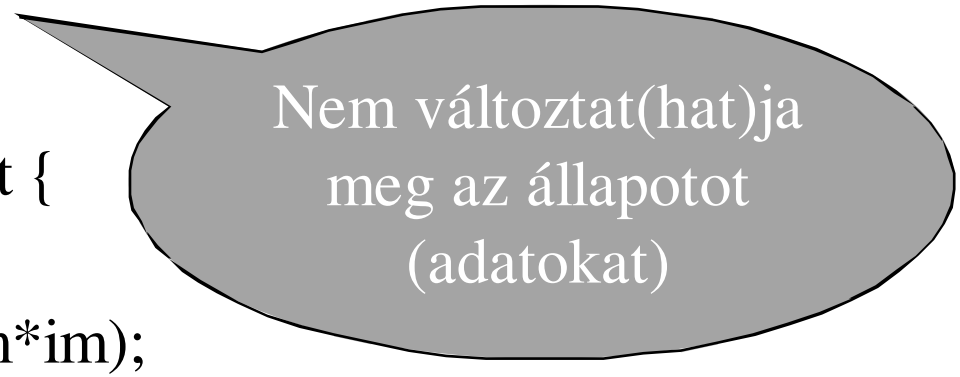
scope operátor

Tagfüggvények szerepe

- Privát adatok lekérdezése (getter fv.)
- Privát adatok beállítása (setter fv.)
- Objektum állapotának (adatainak) változtatása
- Műveletek az adatokkal
- Adatok létrehozása
- Adatok megszüntetése

Konstans tagfüggvények

```
class Komplex {
    double re, im;
  public:
    void set(double r, double i)  { re = r; im = i; }
     double getRe() const { return re; }
     double getIm() const { return im; }
    double abs() const;
};
double Komplex::abs() const {

        return sqrt(re*re + im*im);
}
```



Alapértelmezett tagfüggvények

Automatikusan keletkező (implicit deklarált):

- Konstruktor
  - Létrehozza az objektumot
- Destruktor
  - Megszünteti az objektumot
- Másoló konstruktor
  - Másolás útján hoz létre új objektumot
- Értékadás (értékadó operátor)
  - Új érteket ad egy létező objektumnak
- Címképző és dereferáló operátorok

Konstruktor

KONSTRUKTOR: Az objektum létrejöttekor hívódik.

Feladata, hogy alapállapotba hozza az objektumot.

Ha nem deklarálunk egyet sem, akkor implicit jön létre.

```
class Komplex {
    double re, im;
  public:
    Komplex() { }        // konstruktornak nincs típusa
    Komplex(double r, double i)      { re = r; im = i; }
};
Komplex k1;              // paraméter nélkül hívható (default)
Komplex k2 = k1;         // másoló (copy) ctr. (ez most implicit)
Komplex k3 = Komplex(1.2, 3.4);
```

Ilyen üres programozott törzs keletkezik implicit módon

ideiglenes objektum

Destruktor

DESTRUKTOR: Az objektum megszüntetésekor hívódik.
Alapvető feladata, hogy megszüntesse az obj. által din. mem. területen létrehozott objektumokat/adatokat.

```
class Komplex {
    double re, im;
  public:
    ~Komplex()    {}   // destruktornak paramétere sincs
};
{
  Komplex k1;           // paraméter nélkül hívható (default)
   Komplex k2 = k1;     // másoló (copy) konstruktor
}
```

Ez keletkezne implicit módon

destruktorok hívódnak

Komplex példa újból

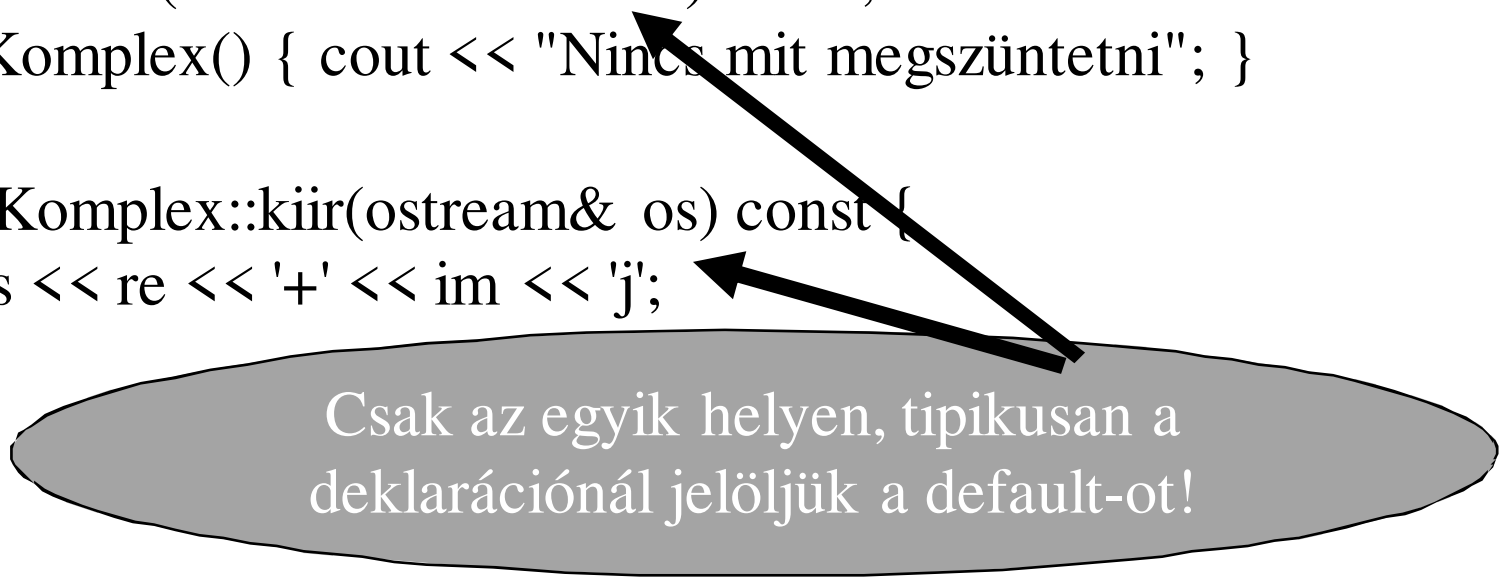
```
class Komplex {
    double re, im;
  public:
    Komplex(double r) { re = r; }
    Komplex(double r, double i) { re = r; im = i; }
    double getRe() const { return re; }
    double getIm() const { return im; }
    ~Komplex() { cout << "Nincs mit megszüntetni"; }
};
{
   Komplex k1(1.3, 0);    // definíció és inic.
   Komplex k2(3), k3;
}
```

Nincs ilyen konstr.

destruktorok meghívódnak

Default argumentummal

```
class Komplex {
    double re, im;
  public:
    Komplex(double r = 0, double i = 0) { re = r; im = i; }
    double getRe() const { return re; }
    double getIm() const { return im; }
    void kiir(ostream& os = cout) const;
    ~Komplex() { cout << "Nincs mit megszüntetni"; }
};
void Komplex::kiir(ostream& os) const {
     os << re << '+' << im << 'j';
}
```



Paraméter nélküli (default) konstr

Objektum létrehozása alapállapottal.
Automatikusan hívódik minden olyan esetben, amikor az objektumnak alapállapotban kell létrejönnie. Pl:

- nem paraméteres ctor-t hívtunk (Komplex k1;)
- tömbelemek létrehozásánál (Komplex kt[10];)
- tartalmazott objektumoknál
  struct Fraktal { Komplex c; int i; .... };
- származtatásnál (ld. később)

- Nem keletkezik implicit, ha van legalább 1 explicit

Most itt tartunk

```
class Komplex {
    double re, im;
  public:
    Komplex(double r = 0, double i = 0) { re = r; im = i; }
    double getRe() const { return re; } // hasonlóan getIm() is
    void kiir(ostream& os = cout) const;
    void setRe(double r) { re = r; } // hasonlóan setIm(double) is
    void beolvas(isteram& is = cin);
};
int main() {
   Komplex k1,  k2(1, 1),  kt[10], k3;
   kt[2].kiir();    // itt mit ír ki?
   Komplex *kp = new Komplex[100];  // mi történik itt?
   delete[] kp; }
```

Mit tud az osztályunk?

- Azokat a műveleteket (metódusokat), amit implementáltunk (set/get, kiír, beolvas, ...)
- + néhány alapértelmezett dolgot, amit az ajándékba kapott implicit deklarált tagfüggvények valósítanak meg pl:

```
Komplex k4 = Komplex(1,8) // inicializálás
k1 = k2;                  // értékadás
Komplex *p = &k1;         // címképzés
k2 = *p;                  // dereferálás
```

- Összeadni nem tud?

Tud összeadni, ha megtanítjuk

- Az operátorokat függvények valósítják meg.
- A függvények túlterhelhetők.
- Majdnem minden operátor túlterhelhető, ha legalább az egyik operandus objektum.

Komplex összeadás:

- Globális operátorral (eddig ilyenek voltak)
- Tagfüggvénnyel (miért ne lehetne op. tagf.)

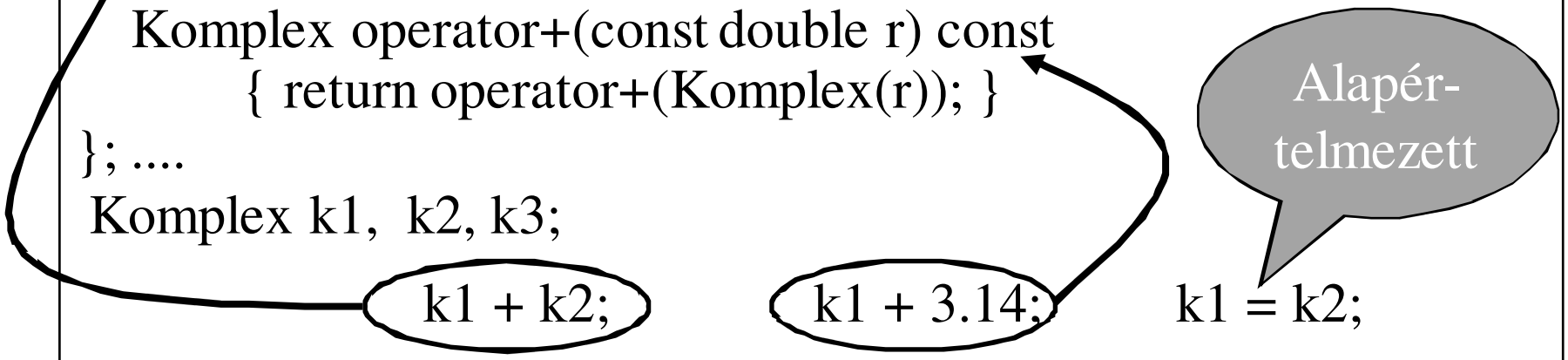
k1 = k2 + k3

- először a + -t kell kiértékelni:
  - ha a bal oldal objektum, akkor van-e megfelelő, azaz k2.operator+(k3) formára illeszkedő tagfüggvénye

  ha nincs, vagy beépített típus és a jobb old. obj., akkor
  - van-e megfelelő globális függvény, azaz operator+(k2, k3) formára illeszkedő függvény.
- Ugyanez történik az = -vel is, de ehhez van implicit deklarált függvény abban az esetben, ha mindkét oldal azonos típusú, aminek a hatása az, amit várunk: értékadás.

Műveletekkel bővített Komplex

```
class Komplex {
  double re, im;
public:    ....
   Komplex operator+(const Komplex& k) const
         { Komplex sum(k.re + re, k.im + im); return sum; }
   Komplex operator+(const double r) const
         { return operator+(Komplex(r)); }
}; ....
 Komplex k1,  k2, k3;
```




```
3.14 + k1;    // bal oldal nem objektum !
              // Ezért globális függvény  kell !
```

double + Komplex

```
class Komplex { ..... };
Globális fv., nem tagfüggvény:
Komplex operator+(const double r, const Komplex& k) {
      return Komplex(k.re + r,    k.im);
}
```

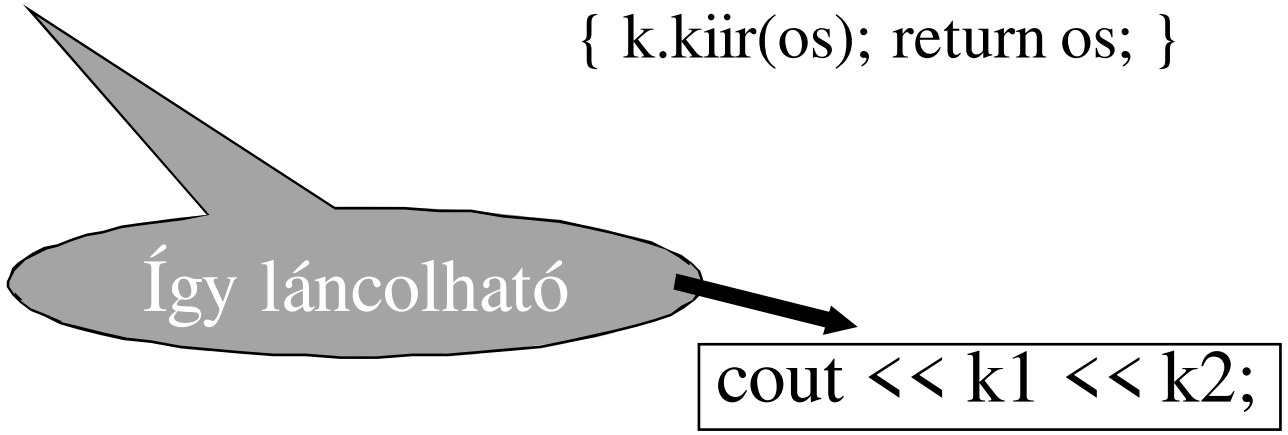
1. megoldás: privát adat elérése pub. fv. használatával:

```
Komplex operator+(const double r, const Komplex& k) {
       return Komplex(k.getRe() + r,   k.getIm());
}
```

Kiírás: cout << k1

A bal oldal objektum ugyan, de nincs a kezünkben. Ezért csak egy operator<<(cout, k1) hívásra illeszthető globális függvénnyel lehet megoldani:

```
ostream& operator<<(ostream& os, const Komplex& k)
                        { k.kiir(os); return os; }
```



Beolvasás: cin >> k1

A bal oldal objektum ugyan, de nincs a kezünkben. Ezért csak egy operator>>(cin, k1) hívásra illeszthető globális függvénnyel lehet megoldani:

```
istream& operator>>(istream& is, Komplex& k)
                              { k.beolvas(is); return is; }
```

A kiir() és a beolvas() tagfüggvény akár el is hagyható:

```
ostream& operator<<(ostream& os, const Komplex& k) {
        return os << k.getRe() << '+' << k.getIm() << 'j';
}
```

Op. túlterhelés szabályai

- Minden túlterhelhető kivéve:
  `.  :: ?: sizeof`
- A szintaxis nem változtatható meg
- Az egyop./kétop. tulajdonság nem változtatható meg
- Precedecia nem változtatható meg
- operator++() -- pre (++i)
- operator++(int) -- post (i++)
- operator double() -- cast (double)
- operator[ ](typ i) -- index (typ tetszőleges)
- operator()() -- függvényhívás

Op. túlterhelés előnye/hátránya

Előnyök

- Szokásos aritmetikai, logikai funkciók
  - Teljes aritmetika (pl: komplex)
  - Összegzés növelés (pl. dátum)
  - Összehasonlítás

Hátrányok

- Szokásostól eltérő funkciók esetén zavaró lehet
  - (double)Komplex(3, 5) – mit jelent?
  - „almás” + „rétes” =?= „rétes” + „almás”
    - A kommutativitás sérül. Lehet, hogy zavaró.
  - cout << 1;

Egy furcsa példa

```
Komplex k1, k2;
double d = (double)k1;       // mit jelent?  valós rész? abs?
```

Jelentse a valós részt:

```
Komplex {
    …
    operator double() { return re; }  // formálisan nincs típusa !!!
};
```

Veszély! A típuskonverzió automatikus is lehet!

Pl: k1 + 3.14 ⟹ (double)k1 + 3.14 lesz, ha nincs operator+(Komplex, double)

Demo (ural2: ~szebi/proga2)

```
#include <iostream>
using std::cout;
using std::endl;
struct Valami {
  Valami()  { cout << "HAHO!" << endl; }
  ~Valami() { cout << "Jaj!" << endl; }
};
int main() {
  cout << "1." << endl; Valami o1;
  cout << "2." << endl; Valami o2;
  Valami *o3 = new Valami;
  return 0;
}
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_03