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

OO paradigmák, osztály, operátorok átdefiniálása

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

Programfejlesztés

- Feladatanalízis
  - világ = dolgok + tevékenységek
- Modellezés
- Tervezés
  - absztrakció
  - dekompozíció
- 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
- követhetetlen
- módosíthatatlan
- nincsenek letisztult vezérlési szerkezetek
  - ciklusba nem illik beugrani
- zsenigyanús programozók
- pótolhatatlan emberek, nem dokumentált
- szoftverkrízis kezdete

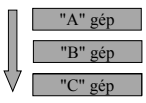
Gépi nyelv ?

```
// Kiirunk 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‹ì.ì@SV
00401358  57 8D 7D C0 B9 10 00 00  WÐ}ÀÌ...
00401360  00 B8 CC CC CC CC F3 AB  .¸ÌÌÌÌó«
00401368  8B 45 08 50 68 1C 20 42  ‹E.Ph. B
00401370  00 E8 5A 24 00 00 83 C4  .èZ$...Ä
00401378  08 5F 5E 5B 83 C4 40 3B  ._^[.Ä@;
00401380  EC E8 9A 22 00 00 8B E5  ìèš"..‹å
00401388  5D C3 CC CC CC CC CC CC  ]ÃÌÌÌÌÌÌ
```


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, axiomák 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, C, MODULA-2

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

Absztrakt adat

- Absztrakt adatszerkezetek
  - a működésre koncentrálunk és nem az adatra
  - működés: leképezés az értelmezési tartomány és az értékkészlet között
  - a művelek algebrai leírással megadhatók
  - nem kell ismerni a megvalósítást, azt sem, hogy mi az adat, csak a műveleteket
  - egy adaton több funkció
  - pl: verem, sor, tömb, lista, fa, stb.
- OBJEKTUM: a konkrét adat és a rajta végezhető műveletek megtestesítője

Feladat: komplex számok

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

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

Feladat: komplex számok/2

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

void main()
{
  Komplex t[10];  // adat
  beolvasas(t);   // funkciók
  kiiras(t);
}
```


Feladat: komplex számok/3

```
double abs(Komplex k) {
{
  return sqrt(k.re*k.re + k.im*k.im);
}
void beolvasas(Komplex t[])
{
for (int i=0; i<10; i++)
  cin >> t[i].re >> t[i].im;
}
void kiiras(Komplex t[])
{
for (int i=9; i>=0; i--)
  cout << t[i].re << ' ' << t[i].im << ' '
       << 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)
- Az eredmény nehezen használható fel újra

Adatorientált dekompozíció

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

Feladat: komplex számok újra

- 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()

Feladat: komplex számok újra /2

```
Komplex k;
KomplexTar t;
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

- OBJEKTUM: a konkrét adat és a rajta végezhető műveletek megtestesítője
- 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 modell

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

Leképezés a C++ eszközkészletére

```
...
for (int i = 0; i < 10; i++){
   cin >> k;
   t[i] = k;
}
for (int i = 9; i >= 0; i--) {
   k = t[i];
   cout << k << ' ' << (double)k
        << endl;
}
```

Nem biztos, hogy tömb, csupán jelölés!!

Absz. érték jelölése lehetne ez is.

Ez egy lehetséges jelölés a műveletekre. Nem biztos, hogy javítja az olvashatóságot! A példa itt csak a lehetőséget demonstrálja.

OO paradigmák

- egységbezárás (encapsulation)
  - osztályok (adatszerkezet, műveletek összekapcsolása)
- többarcúság (polymorphism)
  - műveletek paraméter függőek, tárgy függőek (kötés)
- példányosítás (instantiation)
- öröklés (inheritance)
- generikus adatszerkezet alapú megoldások

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;          // deklaráció és definíció
setKomplex(&k1, 1.2, 0); // inicializálás
f = abs(&k1);
```

Az összetartozásra csak a név utal

Interfész függvények

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

funkció + obj. típusa | melyik adat | művelet operandusai

void beolvasKomplex(Komplex *kp);
double absKomplex(Komplex *kp);

Adattakarás

Ha egy db objektum van, akkor modullal:

komplex.c: static struct Komplex {....}
void setKomplex(double r, double i) {....}
double absKomplex() {...}

komplex.h: extern void setKomplex(double r, double i);
double absKomplex();

program.c: #include "komplex.h"
setKomplex(1.2, 3.1);

OO paradigmák csak önfegyelemmel tarthatók be!

Összetartozás C++-ban

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

egységbezárás

nem tilos

OO paradigmák csak önfegyelemmel tarthatók be!

Adattakarás C++-ban

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

adatok privátak

tagfüggvények nyilvánosak

TILOS, mert privát

CSAK ÍGY

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


Class

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

```
Komplex k1, k2, k3;
```

osztály

objektumok

Egy osztály: Komplex

```
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)); }
void main() {
   Komplex k1; k1.set(1.2, 3.4);
   cout << k1.abs();
}
```

adatok privátak

inline-nak megfelelő

scope operátor

A class megvalósítása

C++

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

C

```
struct Komplex { double re, im; };
void setKomplex(struct Komplex *this, double r, double i)
{
        this -> re = r;
        this -> im = i;
}
double absKomplex(Komplex *this) {...}
```


This pointer ≡ példányra mutató poi.

```
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*im));
}
Komplex k1; double f = k1.abs();
```

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

Műveletek objektumokkal

```
class Komplex {
        ......
};
Komplex k1;          // definició és inicializálás
k1.set(1.2, 3.4);    // két lépésben
Komplex k2 = k1;     // definició másolással *
k2.set(1.3, 8.1);    // állapotváltás
double f =k2.abs();  // állapot lekérdezése
k2 = k1;             // értékadás *
Komplex kt[10];      // 10 elemű tömb (inicializálatlan)
```

*Nem is adtunk meg hozzá metódust (tagfüggvényt)!!! ?
A C++-ban vannak alapértelmezett metódusok!

Konstruktor és destruktor

KONSTRUKTOR: Definíció és inicializálás összevonása.
DESTRUKTOR: Az objektum megszüntetése.

```
class Komplex {
   double re, im;
  public:
   Komplex() { }        // konstruktornak nincs típusa
   Komplex(double r, double i)      { re = r; im = i; }
   ~Komplex()   {}   // destruktornak nincs paramétere
   ...
};
Komplex k1;
Komplex k2 = k1;     // másoló (copy) konstruktorral
Komplex k3 = Komplex(1.2, 3.4);
```

ez az alapértelmezés

ideiglenes objektum

Komplex példa újból

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

Nincs ilyen!

destruktorok meghívódnak

Mi a helyzet a műveletekkel?

```
class Komplex {
   double re, im;
  public:
   Komplex(double r = 0, double i = 0) { re = r; im = i; }
...
};
int main()
{
   Komplex k1,  k2(1, 1),  kt[10];
   Komplex k3 = Komplex(2, 0);

   k1 = k2 + k3;
}
```

default arg. előnyei

Tömb, ha van paraméter nélküli konstruktor

Művelet, hogyan ?

k1 = k2 + k3

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

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

Műveletekkel bővített Komplex

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

Alapér-telmezett

k1 + k2; k1 + 3.14; k1 = k2;

```
3.14 + k1;    // bal oldal nem osztály !
              // 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));
}
```

Baj van! Nem férünk hozzá, mivel privát adat!

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

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

Publikus lekérdező fv. (l. 35. dián)

2. megoldás: védelem enyhítése

- Szükséges lehet a privát adatok elérése egy globális, függvényből, vagy egy másik osztály tagfüggvényéből.
- Az ún. barát függvények hozzáférhetnek az osztály privát adataihoz.

```
class Komplex { ..... public:
// FONTOS! Ez nem tagfüggvény, csak így jelöli, hogy barát
friend Komplex operator+(const double r, const Komplex& k);
};
```

```
Komplex operator+(const double r, const Komplex& k) {
        k.re ..... k.im.... // hozzáfér a privát adatokhoz
}
```


Majdnem kész a Komplex

```
class Komplex {
  double re, im;
public:  .....
  Komplex operator+(const Komplex& k)
        { Komplex sum(k.re + re, k.im + im); return(sum); }
  Komplex operator+(const double r)
        { return(operator+(Komplex(r))); }
  friend Komplex operator+(const double r, const Komplex& k);
  friend ostream& operator<<(ostream& s, const Komplex& k);
};
 ostream& operator<<(ostream& s, const Komplex& k)
        { s << k.re << ',' << k.im << 'j'; return(s); }
```

Így láncolható

```
cout << k1 << k2;
```


Op. átdefiniálás szabályai

- Minden átdefiniálható 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