Osztályok

Elemi Alkalmazások Fejlesztése II.

1. Feladat

Készítsünk egy olyan programot, amelyben sokszögekkel dolgozunk. Az egyszerűség érdekében csak két műveletet szeretnénk használni: sokszög eltolását és súlypontjának kiszámítását. A pontok koordinátái egészek legyenek! (A súlypont meghatározásakor is ennek megfelelően kell eljárnunk, azaz annak koordinátái is egészek lesznek.)

2. A megoldás menete

Első lépésben elkészítjük az egész koordinátájú síkbeli pontoknak megfelelő típust, azaz a `Pont` osztályt. Ezt felhasználva megoldjuk a feladatot, amelynek során létrehozzuk a `Sokszog` osztályt. Eközben megvizsgáljuk az osztályok alapvető tulajdonságait:

- az osztály, mint a típusmegvalósítás eszköze,
- elérhetőség,
- objektumok létrehozása, megszüntetése – konstruktor, destruktor,
- műveletek,
- műveletek túlterhelése,
- operátorok,
- objektumok másolása, értékadás.

3. Az osztályokról röviden

Az előző félévben láthattuk, hogy típusok megvalósításának eszköze C++-ban az osztály. Egyszerű osztályokat is készítettünk a megoldott feladatokban használt típusoknak megfelelően (halmaz, illetve absztrakt fájlok). Az osztályok segítségével lehetőségünk van annak szabályozására is, hogy a külvilág az osztály példányainak, *objektumainak* milyen szolgáltatásait veheti igénybe.

A típusspecifikáció és típus egy-egy rendezett hármas:

$$\begin{aligned} \mathcal{T}_{\mathcal{S}} &= (T, I_S, \mathbb{F}), \\ \mathcal{T} &= (\rho, I, \mathbb{S}). \end{aligned}$$

A típusspecifikációban T a típusértékek halmaza, amelyből I_S invariáns határozza meg a számunkra ténylegesen érdekes halmazt, és $\mathbb{F}$ az elvégezhető feladatok, másként műveletek halmaza, amelynek segítségével a típusértékeket használhatjuk, kezelhetjük. A típusban ρ a reprezentációs függvény, amely megadja, hogy a típusértékeket miként ábrázolhatjuk, I a reprezentációra tett megszorítás, $\mathbb{S}$ a műveleteket a reprezentáción keresztül megvalósító műveletek halmaza.

C++-ban az osztály lehetséges objektumai alkotják a típusértékhalmazt. Az osztály külvilág számára elérhető függvényei alkotják a feladatok halmazát. Minden feladatnak egy program felel meg. A reprezentációt az objektumok ábrázolásához felhasznált adattagok biztosítják. Az invariánst explicit nem tüntethetjük fel, azt a műveleteknek kell fenntartaniuk. Az osztály megadásakor a felületet a `.h` kiterjesztésű fájlban adjuk meg. Ez tartalmazza T, $\mathbb{F}$ és ρ elemeknek megfelelő részeket. A megvalósítás a `.cpp` kiterjesztésű fájlba kerül, ami esetünkben (első megközelítésben) a műveleteket megvalósító programokat jelenti.

```
////////////////////////////////////
// O.h: Az O osztály felülete
////////////////////////////////////
#ifndef __O_H
#define __O_H

class O
{
// műveletek deklarációja
// reprezentáció
};
#endif

////////////////////////////////////
// O.cpp: Az O osztály megvalósítása
////////////////////////////////////
#include "O.h"

// műveletek O:: formában
```

Az osztály felületében meghatározhatjuk az egyes összetevők elérhetőségét. Az összetevő lehet művelet, vagy a reprezentációhoz felhasznált adattag. Három módot adhatunk meg:

public bármely más objektum elérheti ezt az összetevőt;

private csak az adott osztály objektumai használhatják ezt az összetevőt;

protected az adott osztály, és a belőle származtatott osztályok objektumai használhatják ezt az összetevőt.

A megadott elérhetőség érvényes egészen addig, amíg a felület megadásakor mást nem írunk elő.

```
class O
{
public:
    // mindenki által elérhető elemek
    ...
private:
    // csak az osztály objektumai által elérhető elemek
    ...
public:
    // mindenki által elérhető elemek
    ...
protected:
    // csak az osztály és a származtatott osztályok
    // objektumai által elérthető elemek
    ...
};
```

Van két kitüntetett művelet: a konstruktor és a destruktor, amelyek segítségével biztosíthatjuk például az invariáns fennállását. A konstruktor művelet végrehajtásra kerül az objektum létrejöttekor, mielőtt még bármit is tehetnénk az objektummal. Ennek segítségével érhető el, hogy az adattagok megfelelő kezdőértékkel rendelkezzenek. A destruktor művelet az objektum megszűnésekor hajtódik végre. Ez lehetőséget ad arra, hogy az objektum „rendet tegyen maga után”, azaz például felszabadítsa a dinamikusan lefoglalt tárterületet. (Másképp: az invariáns csak ezután szűnik meg.) A jellegükből adódóan ezen műveleteknek nem lehet visszatérési értékük, hiszen a konstruktor egy objektumot hoz létre, a destruktor pedig azt szünteti meg.

```
class O
{
public:
     O(...);     // konstruktor
     ~O();       // destruktor
};

O::O(...)    // a konstruktort megvalósító program
{   ...      }
O::~O()      // a destruktor implementációja
{   ...      }
```

4. A `Pont` osztály

A feladat megoldása során kezelnünk kell a síkbeli pontokat. Ezek alkotják a sokszögek csúcsait, illetve határozzák meg az eltolás vektorát, és egy alakzat súlypontja is pont típusú lesz. Célszerűnek látszik ezért bevezetni ezt az osztályt, illetve C++-ban az ennek megfelelő osztályt.

Egy pontot a két egész koordinátájával jellemezhetünk, azaz a reprezentáció két egész számmal adható meg. A vízszintes koordináta legyen `_x`, a függőleges pedig `_y`, mindkettő típusa `int`. Az adattagok legyenek rejtettek, azaz `private` elérhetőségűek.

A konstruktornak és a destruktornak ebben az esetben nincs szerepe (az invariáns azonosan igaz), így az üres program megfelel. A konstruktort módosítsuk úgy, hogy az origót adja meg. (Ez lesz az alapértelmezett pont objektum.)

Szükség van még mindenképpen az adattagokat beállító, illetve értékeiket lekérdező műveletekre. Legyenek ezek `UjX`, `UjY`, illetve `X` és `Y`. A lekérdező függvények egy pont objektum értékét nem változtatják meg, amit célszerű jelölni. Ezt a függvény neve után írt `const` attribútummal tehetünk meg. Ezt a megkülönböztetést azért kell jelölni, mert ha valahol csak konstans pont fordulhat elő (például paraméter átadásakor), akkor arra csak az ilyen műveletek alkalmazhatóak.

Az eddigiek alapján a következőkhöz jutottunk:

```
////////////////////////////////////////////////////////////
// Pont.h: A Pont osztály felülete
////////////////////////////////////////////////////////////
#ifndef __PONT_H
#define __PONT_H

class Pont
{
public:
    Pont();
    virtual ~Pont();
    void    UjX(int x);
    void    UjY(int y);
    int     X() const;
    int     Y() const;
private:
    int     _x;
    int     _y;
};
#endif
```

```cpp
//////////////////////////////////////////////////////////////
// Pont.cpp: A Pont osztály megvalósítása
//////////////////////////////////////////////////////////////

#include "Pont.h"

Pont::Pont()
{
    _x = _y = 0;
}

Pont::~Pont()
{
}

void Pont::UjX(int x)
{
    _x = x;
}
```

```
void Pont::UjY(int y)
{
    _y = y;
}

int Pont::X() const
{
    return _x;
}

int Pont::Y() const
{
    return _y;
}
```

Ez az osztály az adattagok „elrejtésén” kívül nem jelent többletet egy hagyományos rekordhoz (pl. `struct`) képest. Akkor válik kényelmesebbé, ha megfelelő műveletekkel egészítjük ki.

Egy sokszöget úgy tudunk eltolni, ha az összes pontját eltoljuk, ezért célszerű bevezetni a `Pont` osztály esetén egy `Eltol` műveletet. Ennek paramétere az eltolás vektorát meghatározó pont, visszaadott értéke pedig az eltolt pont, maga a pont nem változik. A művelet tehát a következőképpen nézne ki.

```
class Pont
{
    ...
    Pont Eltol(const Pont &p) const;
    ...
};
Pont Pont::Eltol(const Pont &p) const
{
    Pont    r;
    r.UjX(_x + p._x); r.UjY(_y + p._y);
    return r;
}
```

Látható, hogy az eltolás során létrehozunk egy pontot, majd beállítjuk a koordinátáit. A koordinátákat azonban már a létrehozás pillanatában ismerjük, ezért jobb lenne egyből ezekkel az értékekkel létrehozni a pontot. Ezt megtehetjük, ha kiegészítjük az osztályt egy új konstruktorral, amelynek paraméterei a koordináták.

Ezt megtehetjük, mert a két konstruktor különbözni fog egymástól, ugyanis C++-ban egy függvényt a név, a visszaadott érték típusa, valamint a paraméterek száma és típusa együttesen azonosít. Esetünkben tehát két különböző konstruktor szerepel majd a `Pont` osztályban.

```
class Pont
{
public:
    Pont();
    Pont(int x, int y);     // új konstruktor
    ...
};
Pont::Pont(int x, int y)
{
    _x = x; _y = y;
};
```

Ezt felhasználva az eltolás megvalósítása lényegesen egyszerűbb.

```
Pont Pont::Eltol(const Pont &p) const;
{
    return Pont(_x + p._x, _y + p._y);
}
```

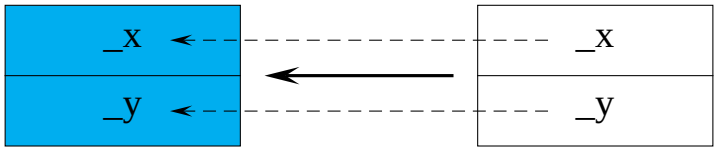
Vizsgáljuk meg mi történik a művelet végrehajtásakor a következő kód esetén!

```
    Pont    a(3, 4);
    Pont    b;
    b = a.Eltol(Pont(1, 1));
```

Az eredmény a `b = (4, 5)` pont lesz. Az első két sorban létrehozunk két pont objektumot, az `a`-t a megadott értékekkel, a `b`-t pedig az eredeti konstruktor segítségével (origó). Az `a` pont `Eltol` műveletét használjuk, az eltolás mértéke `(1, 1)`. Ennek megadásához létrehozunk egy konstans pontot, és ez a művelet paramétere. Ezt megtehetjük, mert az eljárás deklarálásakor a paraméter előtt szerepelt a `const` alapszó. Ellenkező esetben csak változó állhatna itt.

A műveletben létrehozunk egy pontot. A visszatéréskor erről készül egy másolat, és ezt a másolatot adjuk értékül `b`-nek. Vegyük észre, hogy ez két újabb művelet végrehajtását jelenti! Hol vannak ezek a műveletek?

Minden osztályhoz tartozik két alapértelmezett művelet: a másolat létrehozás (copy konstruktor) és az értékadás. Az alapértelmezett viselkedése ezen műveleteknek az, hogy az adattagok értékeit átmásolják a megfelelő objektumba. (Másolat esetén létre is jön egy objektum.)



A pontok esetén ez megfelelő, ezért ezzel most többet nem foglalkozunk.

Sokkal kifejezőbb lenne a programkód, ha a be lehetne vezetni a pontok esetén egy + műveletet, ami két pontot adna össze, azaz létrehozna egy pontot, amelynek koordinátái a pontok koordinátáinak összege lenne. Ezt megtehetjük egy új, a pontokon értelmezett + operátor bevezetésével. A + művelet is egy függvény, amelynek paraméterei az argumentumai. Így az új + minden eddigitől különbözni fog.

```
class Pont
{
public:
    ...
    Pont    operator+(const Pont &p) const;
    ...
};

Pont Pont::operator+(const Pont &p) const
{
    return Pont(_x + p._x, _y + p._y);
}
```

Így az eltolás:

```
    b = a + Pont(1, 1);     // b = a.+(Pont(1, 1)
```

Nézzük meg miként számíthatnánk ki egy sokszög súlypontját! Tegyük fel, hogy a sokszög csúcspontjai az `s` tömbben találhatóak, amelynek mérete `n`.

```
Pont    sp;
for ( int i = 0; i < n; i++ )   sp = sp + s[i];
sp.UjX(sp.X() / n); sp.UjY(sp.Y() / n);
```

Az utolsó sor sokkal szemléletesebb lenne a következő formában.

```
sp = sp / n;
```

Ezt megtehetjük, ha bevezetünk egy új `/` operátort a pontokra. Az operátor paramétere egy egész szám lesz, az első paraméter a pont.

```
class Pont
{
public:
    ...
    Pont    operator/(int f) const;
    ...
};

Pont Pont::operator/(int f) const
{
    return Pont(_x / f, _y / f);
}
```

A kitűzött feladat az eddig bevezetett műveletekkel már megoldható, de tegyük fel, hogy szükség lenne egy olyan műveletre is, amellyel egy pont koordinátáit egy egész számmal szorozzuk meg (nyújtás). Ez megvalósítható a `*` operátor segítségével.

```
class Pont
{
public:
    ...
    Pont    operator*(int f) const;
    ...
};

Pont Pont::operator*(int f) const
{
    return Pont(_x * f, _y * f);
}
```

Ekkor a következő kódrészlet első sora helyes, a második helytelen.

```
    b = a * 5;      // legális
    b = 5 * a;      // illegális
```

Ennek oka, hogy a `*` operátor paramétereinek sorrendje rögzített, az első csak pont, a második csak egész szám lehet. Ugyanakkor elvárnánk, hogy a második sor is helyes legyen, ha az első sor az. (A `/` nem szimmetrikus, így ott ez nem okozott gondot.)

Ezt úgy tehetnénk meg, ha az egészek osztályát bővítenénk egy új szorzás művelettel, aminek paramétere (második argumentuma) egy pont lenne. Az egészek osztályát azonban nem változtathatjuk meg, az rögzített. A megoldás egy `friend` művelet bevezetése a pont osztályban, ami az egészeket használja.

```
class Pont
{
public:
    ...
    Pont    operator*(int f) const;
    friend Pont operator*(const int f, const Pont &p);
    ...
};

Pont Pont::operator*(int f) const
{
    return Pont(_x * f, _y * f);
}

Pont operator*(const int f, const Pont &p)
{
    return Pont(f * p._x, f * p._y);
}
```

5. A `Sokszog` osztály

A `Pont` osztály felhasználásával elkészítjük a sokszögeknek megfelelő `Sokszog` osztályt. Egy sokszög ábrázolható, ha megadjuk az oldalak számát, és a csúcsainak koordinátáit. Azaz szükségünk van egy egész számra, és pontok dinamikus tömbjére, miután az oldalak száma előre nem ismert.

Egy sokszög létrehozásakor meg kell adnunk az oldalak számát és a csúcsokat, ezért a konstruktornak két paramétere lesz: egy egész szám és egy ponttokból álló tömb. Ennek megfelelően dinamikusan allokálni kell egy tömböt, és abba átmásolni az értékeket. A sokszög megszűnésekor a dinamikus tömböt fel kell szabadítani a destruktorban. Nyilvánvalóan szükségünk lesz még az oldalszám, illetve egy adott csúcs lekérdezésére. A feladat megoldásához célszerű még két műveletet bevezetni. Az egyik eltolja a sokszöget (nem hoz létre újat), a másik meghatározza a súlypontját.

```
//////////////////////////////////////////////////////
// Sokszog.h: A Sokszög osztály felülete
//////////////////////////////////////////////////////
#ifndef __SOKSZOG_H
#define __SOKSZOG_H
#include "Pont.h"
class Sokszog
{
public:
    Sokszog(int n, Pont *csucsok);
    virtual ~Sokszog();
    int     Oldalszam() const;
    Pont    Csucs(int i) const;
    void    Eltol(const Pont &p);
    Pont    Sulypont() const;
private:
    int     oldalszam;
    Pont    *pontok;
};
#endif
```

```
////////////////////////////////////////////////////////////////////
// Sokszog.cpp: A Sokszög osztály megvalósítása
////////////////////////////////////////////////////////////////////

#include "Sokszog.h"

Sokszog::Sokszog(int n, Pont *csucsok)
{
    oldalszam = n;
    pontok = new Pont[oldalszam];
    for ( int i = 0; i < oldalszam; i++ )
         pontok[i] = csucsok[i];
}

Sokszog::~Sokszog()
{
    delete [] pontok;
}
```

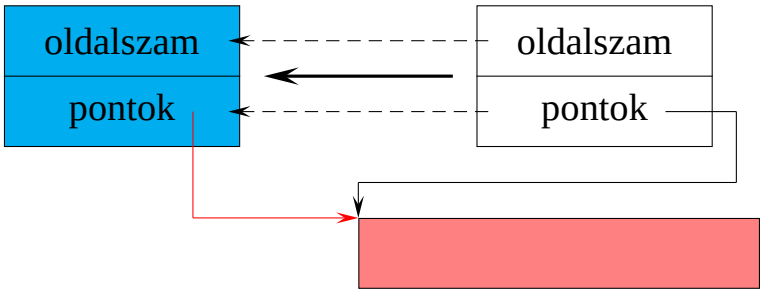
```
int Sokszog::Oldalszam() const
{
    return oldalszam;
}

Pont Sokszog::Csucs(int i) const
{
    if ( i < 0 || i >= oldalszam )  return Pont(0, 0); // kivétel
    return pontok[i];
}

void Sokszog::Eltol(const Pont &p)
{
    for ( int i = 0; i < oldalszam; i++ )
         pontok[i] = pontok[i] + p;
}
```

```
Pont Sokszog::Sulypont() const
{
    Pont    sp(0,0);
    for ( int i = 0; i < oldalszam; i++ ) sp = sp + pontok[i];
    sp = sp / oldalszam;
    return sp;
}
```

Ez a megoldás látszólag helyes, de jobban megvizsgálva van benne egy súlyos hiba. A hiba oka az alapértelmezett másolat létrehozása, illetve értékadás operátor. Ezek ugyanis az adattagokat másolják le. Ez esetünkben egy egész számot és egy mutatót jelent. Ez helytelen, mert a művelet végrehajtása után több objektum is ugyanazt a tömböt használja. Ez legkésőbb a megszűnéskor biztosan gondot okoz, ugyanis az egyik előbb szűnik meg, és felszabadítja a tárterületet, amelyet a másik még használna.



Ezért esetünkben ezt a két műveletet meg kell valósítani.

Másolás esetén az oldalszámot át kell írni, létre kell hozni a dinamikus tömböt, és átmásolni a pontokat.

Az értékadás kicsit bonyolultabb:

- `s1 = s2 = s3;`.
- Egy meglevőt írunk át → a dinamikus tömböt fel kell szabadítani az átírás előtt, de `s = s;`.

```
class Sokszog
{
public:
    ...
    Sokszog(const Sokszog &s);
    Sokszog &operator=(const Sokszog &s);
    ...
};

Sokszog::Sokszog(const Sokszog &s)
{
    oldalszam = s.oldalszam;
    pontok = new Pont[oldalszam];
    for ( int i = 0; i < oldalszam; i++ )
        pontok[i] = s.pontok[i];
}
```

```
Sokszog &Sokszog::operator=(const Sokszog &s)
{
    if ( this == &s )   return *this;
    delete [] pontok;
    oldalszam = s.oldalszam;
    pontok = new Pont[oldalszam];
    for ( int i = 0; i < oldalszam; i++ )
        pontok[i] = s.pontok[i];
    return *this;
}
```

Az eddigiek felhasználásával a feladat első változatának egy lehetséges megoldása a következő.

```
#include <iostream>
#include <fstream>
#include <string>
#include "Pont.h"
#include "Sokszog.h"
using namespace std;

void kiir(Sokszog *s)                    // sokszög kiírása
{
    cout << "<";
    for ( int i = 0; i < s->Oldalszam(); i++ )
        cout << " (" << s->Csucs(i).X() << "," <<
             s->Csucs(i).Y() << ")";
    cout << " >" << endl;
}
```

```
void olvas(ifstream &inp, Sokszog *&s)  // sokszög beolvasása
{
    Pont    tomb[100];  // max. 100 csúcs lehet
    int     aktoldal;
    int     kord;

    inp >> aktoldal;
    for ( int i = 0; i < aktoldal; i++ )
    {
        inp >> kord;
        tomb[i].UjX(kord);
        inp >> kord;
        tomb[i].UjY(kord);
    }
    s = new Sokszog(aktoldal, tomb);
}
```

```
int main()
{
    Sokszog     **s;        // sokszög-mutatók dinamikus tömbje
    int         sokszogszam;
    ifstream    inp;
    string      fn;
    int         i;

    cout << "A sokszogeket tartalmazo fajl: ";
    cin >> fn;
    inp.open(fn.c_str());
    inp >> sokszogszam;
    s = new Sokszog *[sokszogszam];
    for ( i = 0; i < sokszogszam; i++ ) olvas(inp, s[i]);
    inp.close();

    Pont    p(20,20);
    Pont    sp;
```

```
    for ( i = 0; i < sokszogszam; i++ )
    {
        s[i]->Eltol(p);
        kiir(s[i]);
        sp = s[i]->Sulypont();
        cout << "(" << sp.X() << "," << sp.Y() << ")" << endl;
    }
    return 0;
}
```

Egy másik megoldást mutatunk a problémára. Ebben bevezetünk egy új konstruktort, amelyben csak a sokszög oldalszámát kell megadni, valamint egy új indexelés operátort, amelynek segítségével a sokszög csúcsait elérhetjük, és az eddigi `Csucs` művelettel szemben meg is változtathatjuk az értéket. Az új konstruktor csak a csúcsok tömbjét foglalja le dinamikusan, a csúcsok beállítása ezután egyenként lehetséges az indexelés felhasználásával.

```
class Sokszog
{
public:
    Sokszog(int n);
    ...
    Pont    &operator[](int i);
    ...
};

Sokszog::Sokszog(int n)
{
    oldalszam = n;
    pontok = new Pont[oldalszam];
}
```

```
Pont &Sokszog::operator[](int i)
{
    return pontok[i];
}
```

Ezt felhasználva elkészíthetünk egy új olvasó műveletet, legyen ez `tolt`.

```
void tolt(ifstream &inp, Sokszog *&s)
{
    Pont    p;
    int     aktoldal;
    int     kord;

    inp >> aktoldal;
    s = new Sokszog(aktoldal);
    for ( int i = 0; i < aktoldal; i++ )
    {
        inp >> kord;
        p.UjX(kord);
        inp >> kord;
        p.UjY(kord);
        (*s)[i] = p;
    }
}
```

Az eddigieket összefoglalva megkapjuk az első megoldás teljes programját.

6. Osztályszintű elemek

Az eddigiekben felhasznált adattagokat és műveleteket egy osztály objektumain értelmeztük. Ez azt jelenti, hogy az adatokat minden objektum külön-külön tárolja és kezeli. Lehetőség van olyan adatok bevezetésére, amelyek közösek, azaz az osztály összes objektuma ugyanazt az adatot használja. Ezeket nevezzük osztályszintű adatoknak.

Az osztályszintű adatok nem tartoznak egyetlen objektumhoz sem, ezért műveleteket lehet végrehajtani ezekkel akkor is, ha nem hoztunk létre egyetlen objektumot sem az osztályból. Erre szolgálnak az osztályszintű műveletek. Ezeket használhatjuk objektumtól függetlenül `Osztály::Művelet()` formában, illetve egy objektum segítségével is a szokott formában.

Osztályszintű elemek megadására szolgál a `static` minősítő. Az ilyen adattagok és műveletek osztályszintűek. Az adatokat külön definiálni kell a megfelelő `.cpp` fájlban.

Példaként tekintsük a sokszögeket, és tegyük fel, hogy szükségünk van a létrehozott sokszögek számára. Ez objektum független érték, ezért osztályszintű adattag lesz, akárcsak az értéket lekérdező művelet.

```
//////////////////////////////////////////////////////////
// Sokszog.h: A Sokszög osztály felülete
//////////////////////////////////////////////////////////
#ifndef __SOKSZOG_H
#define __SOKSZOG_H

#include "Pont.h"

class Sokszog
{
public:
    ...
    Pont        Sulypont() const;
    static int  Darab()   { return darab; }
private:
    int         oldalszam;
    Pont        *pontok;
    static int  darab;
};
#endif
```

A megvalósítás eleje:

```
////////////////////////////////////////////////////////////////
// Sokszog.cpp: A Sokszög osztály megvalósítása
////////////////////////////////////////////////////////////////

#include "Sokszog.h"

int Sokszog::darab = 0;

Sokszog::Sokszog(int n, Pont *csucsok)
{
    oldalszam = n;
    pontok = new Pont[oldalszam];
    for ( int i = 0; i < oldalszam; i++ )
        pontok[i] = csucsok[i];
    darab++;
}
```

```
Sokszog::Sokszog(int n)
{
    oldalszam = n;
    pontok = new Pont[oldalszam];
    darab++;
}

Sokszog::~Sokszog()
{
    delete [] pontok;
    darab--;
}
    ...
```

Lehetséges használatot szemléltet a következő kódrészlet.

```
cout << Sokszog::Darab() << endl;
Sokszog  s(10);
cout << s.Darab() << endl;
```