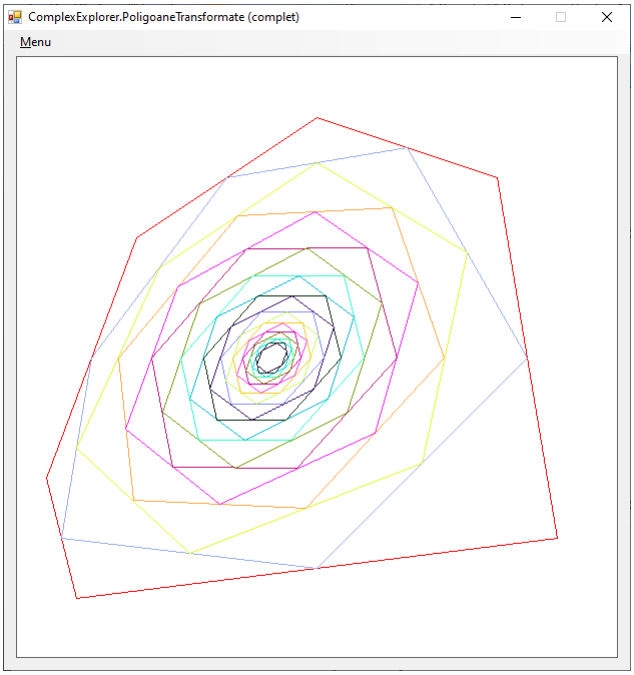
Tablouri în C#

Vom prezenta în continuare câteva observaţii privind utilizarea tablourilor în C#, deoarece aici apar diferenţe semnificative faţă de C/C++. Începem prin a rezolva şi comenta următorul exerciţiu de programare:

Exemplul 1. *Fiind dat un poligon oarecare* (*cel din exterior din în figura următoare*) *să se traseze poligonul cu vârfurile în mijloacele laturilor poligonului dat şi apoi să se repete de mai multe ori această transformare pentru fiecare poligon nou obţinut.*



Iată programul care a trasat figura de mai sus:

```
public class PoligoaneTransformate : ComplexForm
    {
        static Complex i = new Complex(0, 1);

        void transformaPeLoc(Complex[] t)
        {
            int n = t.Length;
            Complex t0 = t[0];
            for (int k = 0; k < n - 1; k++)
            {
                t[k] = (t[k] + t[k + 1]) / 2;
            }
            t[n - 1] = (t[n - 1] + t0) / 2;
        }
```

```
        void traseazaNgon(Complex[] t, Color col)
        {
            int n = t.Length;
            for (int k = 0; k < n - 1; k++)
            {
                setLine(t[k], t[k + 1], col);
            }
            setLine(t[n - 1], t[0], col);
        }

        public override void makeImage()
        {
            setXminXmaxYminYmax(0, 10, 0, 10);
            ScreenColor = Color.White;
            initScreen();
            Complex[] baza = new Complex[6];
            baza[0] = 1 + i;
            baza[1] = 9 + 2 * i;
            baza[2] = 8 + 8 * i;
            baza[3] = 5 + 9 * i;
            baza[4] = 2 + 7 * i;
            baza[5] = 0.5 + 3 * i;
            traseazaNgon(baza, Color.Red);

            for (int k = 0; k < 20; k++)
            {
                transformaPeLoc(baza);
                traseazaNgon(baza, getColor(100 + 100 * k));
            }
            resetScreen();
        }
    }
```

Observăm că vârfurile poligonului iniţial, aici un hexagon, au fost memorate în tabloul de numere complexe numit **baza**. Simpla declaraţie

```
Complex[] baza;
```

este permisă, ea declară variabila **baza** ca fiind o referinţă (un pointer pe 32 de biţi) capabilă să păstreze adresa corpului obiectului referit, corp care apoi poate fi obţinut direct cu operatorul **new**

```
baza= new Complex[6];
```

sau printr-o atribuire de forma

```
baza = poligonulInitial();
```

dacă metoda apelată returnează un tablou de numere complexe. De exemplu,

```
Complex[] poligonulInitial()
 {
     return new Complex[] { 1 + i, 9 + 2 * i, 8 + 8 * i, 5 + 9 * i,
                            2 + 7 * i, 0.5 + 3 * i };
 }
```

returnează exact acelaşi hexagon ca cel dat în exemplu. Verificaţi:

```
public override void makeImage()
 {
     setXminXmaxYminYmax(0, 10, 0, 10);
     ScreenColor = Color.White;
     initScreen();
     //Complex[] baza=new Complex[6];
     //baza[0] = 1 + i;
     //baza[1] = 9 + 2 * i;
     //baza[2] = 8 + 8 * i;
     //baza[3] = 5 + 9 * i;
     //baza[4] = 2 + 7 * i;
     //baza[5] = 0.5 + 3 * i;
     Complex[] baza = poligonulInitial();
     traseazaNgon(baza, Color.Red);
     for (int k = 0; k < 20; k++)
     {
         transformaPeLoc(baza);
         traseazaNgon(baza, getColor(100 + 100 * k));
     }
     resetScreen();
 }
```

Reţinem: în C# tablourile sunt obiecte de tip clasă (instanţe ale unei clase), mai precis ale clasei **Arrays**, (vezi https://docs.microsoft.com/en-us/dotnet/api/system.array?view=netframework-4.8), în consecinţă ele sunt *obiecte de tip referinţă*, numele tabloului este o variabilă alocată pe stivă care trebuie să conţină, la momentul utilizării, adresa din heap a corpului unde sunt memorate elementele tabloului, corpul se obţine, direct sau indirect (prin atribuiri) numai cu operatorul **new**, cel care invocă constructorul clasei.

Să observăm că în sintaxa declaraţiei

```
Complex[] baza;
```

nu apare dimensiunea tabloului, referinţa **baza** poate, de exemplu, să ţintească pe rând tablouri cu dimensiuni diferite. În rest tablourile în C# au aproape aceeaşi sintaxă ca tablourile statice din C/C++, dar sunt foarte diferite de acestea, fiind de fapt echivalente cu tablourile alocate dinamic cu operatorul **new** în C++.

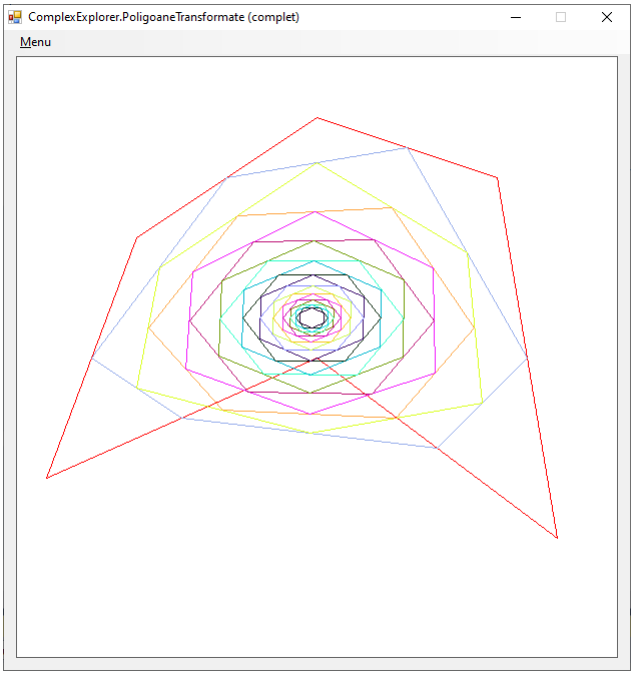
Menţionăm că datorită mecanismului de dealocare automată, tablourile din C# sunt mult mai uşor de folosit decât tablourile dinamice din C++, dacă sunt manipulate corect, desigur.

Să exersăm atribuirile de tablouri: funcţia **makeImage()** următoare

```
public override void makeImage()
 {
     setXminXmaxYminYmax(0, 10, 0, 10);
     ScreenColor = Color.White;
     initScreen();
     Complex[] baza = new Complex[6];
     baza[0] = 1 + i;
     baza[1] = 9 + 2 * i;
     baza[2] = 8 + 8 * i;
     baza[3] = 5 + 9 * i;
     baza[4] = 2 + 7 * i;
```

```
        baza[5] = 0.5 + 3 * i;


        Complex[] hexa = baza;
        hexa[0] = 5 + 5 * i;
        traseazaNgon(baza, Color.Red);

        for (int k = 0; k < 20; k++)
        {
            transformaPeLoc(baza);
            traseazaNgon(baza, getColor(100 + 100 * k));
        }
        resetScreen();
    }
```

are ca rezultat imginea modificată



deşi, aparent, noi nu am modificat cu nimic poligonul din tabloul **baza**. Totuşi, atribuirea

```
hexa = baza;
```

acceptată de compilator deoarece aici avem o atribuire între două instanţe ale aceleiaşi clase, a făcut ca referinţele **baza** şi **hexa** să fie sinonime: ele ţintesc acelaşi corp din heap[1], deci

```
hexa[0] = 5 + 5 * i;
```

are acum acelaşi efect ca

```
baza[0] = 5 + 5 * i;
```

[1] în C/C++ **baza** şi **hexa** ar fi doi pointeri care reţin adresele returnate de apelul operatorului **new** la alocarea unor tablouri dinamice

Spre deosebire de cazul obiectelor de tip valoare, cum sunt în C# structurile, o atribuire între obiecte de tip referinţă nu copie corpul obiectului atribuit ci numai adresa sa, nu crează o copie ci numai o nouă denumire a obiectului. Dacă dorim să realizăm o copie, limbajul ne pune la dispoziţie, printre altele, metoda **Clone(),**

```
public object Clone();
```

care este accesibilă în acest caz, deoarece clasa **Arrays** implementează şi interfaţa **ICloneable**, pe lângă alte interfeţe:

```
public abstract class Array : ICloneable, System.Collections.IList,
System.Collections.IStructuralComparable, System.Collections.IStructuralEquatable
```

Observăm că metoda **Clone()** returnează o referinţă de tip **object**, prin urmare[2] pentru ca aceasta să poată fi atribuită mai departe este nevoie de o conversie explicită[3], aşa

```
hexa = (Complex[]) baza.Clone();
```

sau aşa

```
hexa = baza.Clone() as Complex[];
```

Acum **hexa** a primit adresa unui corp nou, în care au fost depuse copii ale elementelor tabloului **baza**, astfel că secvenţa cod

```
Complex[] hexa = (Complex[]) baza.Clone();
hexa[0] = 5 + 5 * i;
traseazaNgon(baza, Color.Red);
for (int k = 0; k < 20; k++)
{
    transformaPeLoc(baza);
    traseazaNgon(baza, getColor(100 + 100 * k));
}
```

nu mai modifică imaginea iniţială.

Continuăm acum cu metoda care trasează un poligon

```
void traseazaNgon(Complex[] t, Color col)
 {
     int n = t.Length;
     for (int k = 0; k < n - 1; k++)
     {
         setLine(t[k], t[k + 1], col);
     }
     setLine(t[n - 1], t[0], col);
```

[2] Dacă **a** şi **b** sunt variabile de tipuri diferite, **clasaA** şi **clasaB**, atribuirea **a** = **b** este permisă numai dacă cele două clase sunt pe aceeaşi linie ierarhică şi **clasaB** este descendentă, direct sau indirect, din **clasaA** , astfel încât obiectul referit de **b** să posede orice câmp sau metodă care poate fi accesată prin variabila **a**, conform tipului ei declarat, **clasaA**. Cum clasa **object** stă în capul oricărei linii ierarhice, o variabilă de tip **object** poate sta în stânga oricărei atribuiri de clase diferite, dar niciodată în dreapta (fără conversie, desigur).

[3] Operatorul de conversie explicită (**Complex**[]) aplicat unei variabile **c** de tip **object** verifică la momentul execuţiei, folosind operatorul **is**, dacă tipul obiectului referit de **c** este chiar **Complex**[] sau dacă există o conversie definită de la acest tip la **Complex**[], caz în care o apelează, altfel aruncă o excepţie de tip **System.InvalidCastException**.

```
}
```

unde observăm un avantaj considerabil faţă de C++ în utilizarea tablourilor: când trimitem un tablou într-o funcţie nu mai este necesar să trimitem şi dimensiunea acestuia. Proprietatea

```
public int Length { get; }
```

(vezi https://docs.microsoft.com/en-us/dotnet/api/system.array.length?view=netframework-4.8#System_Array_Length) ne returnează dimensiunea tabloului, adică numărul de elemente al acestuia.

Reţinem: în C# tablourile ştiu singure câte elemente au, mai precis, tabloul **tab** are **tab.Length** elemente.

Metoda **traseazaNgon()** citeşte tabloul **t** fără să-l modifice, aşa că nu mai e nimic de spus despre ea[4], analizăm acum metoda care îl modifică:

```
void transformaPeLoc(Complex[] t)
 {
     int n = t.Length;
     Complex t0 = t[0];
     for (int k = 0; k < n - 1; k++)
     {
         t[k] = (t[k] + t[k + 1]) / 2;
     }
     t[n - 1] = (t[n - 1] + t0) / 2;
 }
```

Observăm că, exact ca în C/C++, modificările elementelor unui tablou transmis prin valoare sunt persistente la încetarea apelului, iar explicaţia este aceeaşi: funcţia apelată primeşte o copie a referinţei către corpul tabloului, prin urmare ea modifică indirect corpul ţintit de referinţa din funcţia apelantă.

Metoda funcţionează corect chiar dacă noi modificăm în final, experimental, referinţa primită:

```
void transformaPeLoc(Complex[] t)
 {
     int n = t.Length;
     Complex t0 = t[0];
     for (int k = 0; k < n - 1; k++)
     {
         t[k] = (t[k] + t[k + 1]) / 2;
     }
     t[n - 1] = (t[n - 1] + t0) / 2;

     t = new Complex[2] { 1, 1 + i };
 }
```

Încercaţi. La revenirea în **makeImage()** noua valoare a referinţei **t** se pierde, iar noul corp referit de ea este dealocat automat de către **Garbage Collector**.

[4] Metoda ar fi avut o formă mai elegantă dacă *n*-gonul ar fi fost memorat ca o *linie poligonală închisă*, adică într-un vector **t** de dimensiune *n*+1, cu **t[*n*]=t[**0**].** Am ales varianta de aici din motive didactice.

Deoarece transformarea dată ca exerciţiu este foarte simplă, am reuşit uşor să o implementăm transformând pe loc tabloul **t**, fără un tablou auxiliar de sprijin. Am avut nevoie să punem "la păstrare" doar valoarea iniţială lui **t**[0]. O astfel de rezolvare nu este totdeauna posibilă, de exemplu dacă exerciţiul ar fi cerut să înlocuim fiecare vârf cu mijlocul segmentului care îl leagă de cel mai îndepărtat alt vârf al poligonului dat.

Să analizăm aşadar şi posibilele soluţii cu un tablou de sprijin, mai ales acum când se pot aloca foarte uşor tablouri de dimensiune variabilă.

Prima soluţie este clasică

```
void transformaCuAux(Complex[] t)
 {
     int n = t.Length;
     Complex[] aux = new Complex[n];
     for (int k = 0; k < n - 1; k++)
     {
         aux[k] = (t[k] + t[k + 1]) / 2;
     }
     aux[n - 1] = (t[n - 1] + t[0]) / 2;

     //transferam elementele lui aux in elementele lui t
     for (int k = 0; k < n; k++)
     {
         t[k] = aux[k];
     }

 }
```

şi funcţionează foarte bine, dacă actualizăm **metoda makeImage()**, bineînţeles:

```
for (int k = 0; k < 20; k++)
     {
         //transformaPeLoc(baza);
         transformaCuAux(baza);

         traseazaNgon(baza, getColor(100 + 100 * k));
     }
```

Acum vrem să scăpăm de **for**-ul din finalul metodei **transformaCuAux(),** cel în care are loc transferul element cu element al elementelor vectorului auxiliar. Următoarea încercare

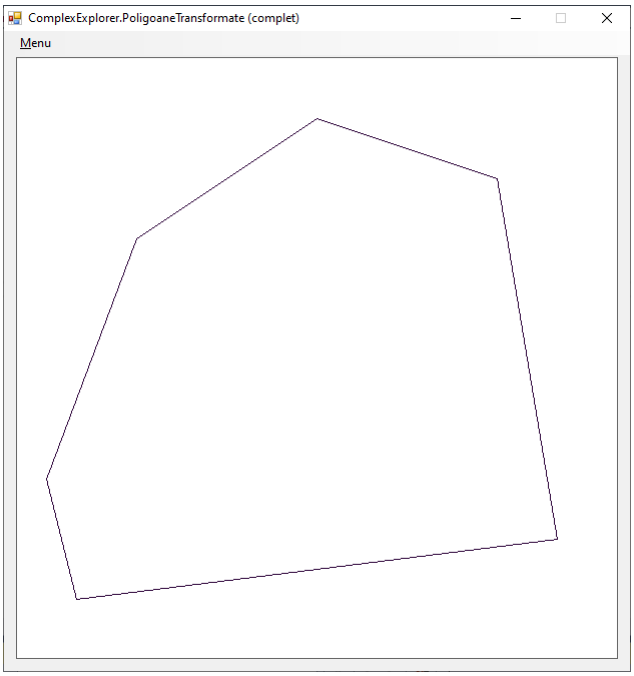
```
void transformaCuAux(Complex[] t)
 {
     int n = t.Length;
     Complex[] aux = new Complex[n];
     for (int k = 0; k < n - 1; k++)
     {
         aux[k] = (t[k] + t[k + 1]) / 2;
     }
     aux[n - 1] = (t[n - 1] + t[0]) / 2;
     ////transferam elementele lui aux in elementele lui t
     //for (int k = 0; k < n; k++)
     //{
     //    t[k] = aux[k];
```

```
        //}
        t = aux;
    }
```

este sortită eşecului deoarece acum apelul

```
transformaCuAux(baza);
```

nu modifică nici elementele tabloului **baza**, nici referinţa **baza**. Poligonul iniţial rămâne neschimbat, după cum se vede în următorul rezultat:



O rezolvare poate fi dată cu apelarea prin referinţă. Metoda

```
void transformaPrinRef(ref Complex[] t)
 {
     int n = t.Length;
     Complex[] rez = new Complex[n];
     for (int k = 0; k < n - 1; k++)
     {
         rez[k] = (t[k] + t[k + 1]) / 2;
     }
     rez[n - 1] = (t[n - 1] + t[0]) / 2;
     t = rez;
 }
```

apelată astfel

```
    for (int k = 0; k < 20; k++)

    {
        //transformaPeLoc(baza);
        //transformaCuAux(baza);
        transformaPrinRef(ref baza);
        traseazaNgon(baza, getColor(100 + 100 * k));
```

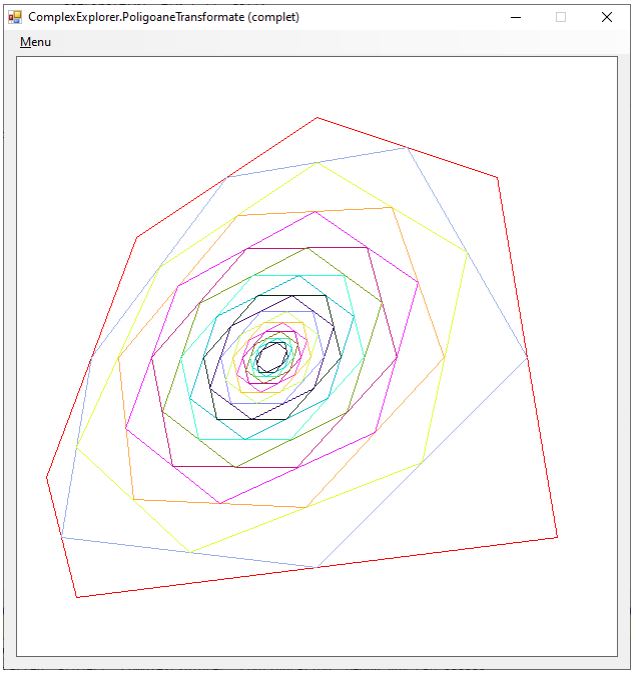
```
}
```

trimite înapoi în variabila **baza** din **makeImage()** valoarea referinţei **rez**[5] mutată în **t** prin atribuirea finală; prin urmare în **baza** avem acum o referinţă nouă către un corp nou, mai precis către corpul tabloului **rez**, tabloul care conţine mijloacele laturilor poligonului trimis.

Observaţi obligativitatea folosirii cuvântului cheie **ref** atât în declaraţia metodei cât şi la apelarea sa.

Şi în cazul apelului prin valoare se poate da o rezolvare pentru evitarea trasferului element cu element al tabloului auxiliar, folosind clonarea, de exemplu, dar aceasta trebuie făcută de la bun început, pentru a obţine mai întâi o copie martor cu ajutorul căreia modificăm apoi elementele tabloului primit.

Următoarea imagine corectă



a fost obţinută cu metoda

```
void transformaPrinClonare(Complex[] t)
 {
     Complex[] clona = (Complex[])t.Clone();

     int n = t.Length;
     for (int k = 0; k < n - 1; k++)
     {
         t[k] = (clona[k] + clona[k + 1]) / 2;
     }
```

[5] Observaţi că la trecerea la apelul prin referinţă am schimbat identificatorul **aux**, de la *auxiliar*, în **rez,** prescurtare a cuvântului *rezultat*, deoarece acum tabloul nou alocat nu mai are un rol auxiliar, el este chiar rezultatul returnat. Această schemă de numire este o obişnuinţă personală formată în limbajul Delphi, unde în orice funcţie este creată în mod automat o variabilă cu numele **Result** care conţine rezultatul returnat de funcţie.

```
        t[n - 1] = (clona[n - 1] + clona[0]) / 2;
    }
```

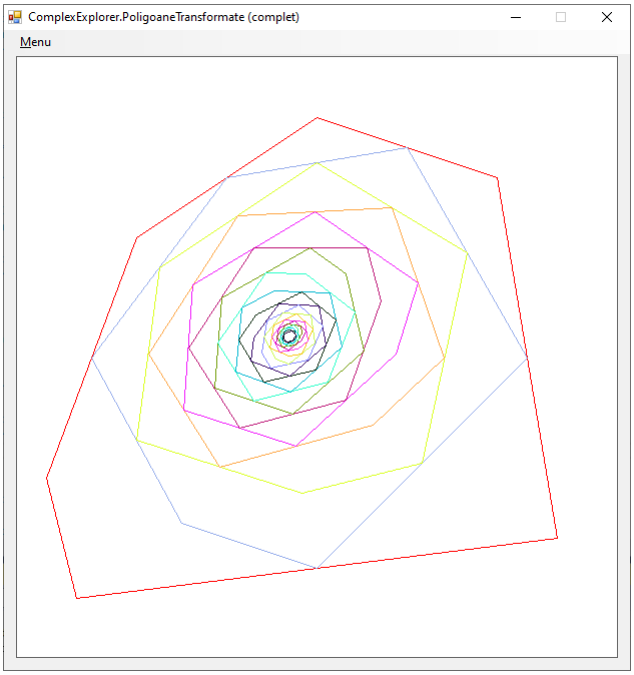
apelată în for-ul

```
    for (int k = 0; k < 20; k++)
        {
            //transformaPeLoc(baza);
            //transformaCuAux(baza);
            //transformaPrinRef(ref baza);
            transformaPrinClonare(baza);
            traseazaNgon(baza, getColor(100 + 100 * k));
        }
```

În final, o întrebare: de ce varianta cu atribuire directă

```
void transformaPrinClonare(Complex[] t)
 {
     //Complex[] clona = (Complex[])t.Clone();
     Complex[] clona = t;
     int n = t.Length;
     for (int k = 0; k < n - 1; k++)
     {
         t[k] = (clona[k] + clona[k + 1]) / 2;
     }
     t[n - 1] = (clona[n - 1] + clona[0]) / 2;
 }
```

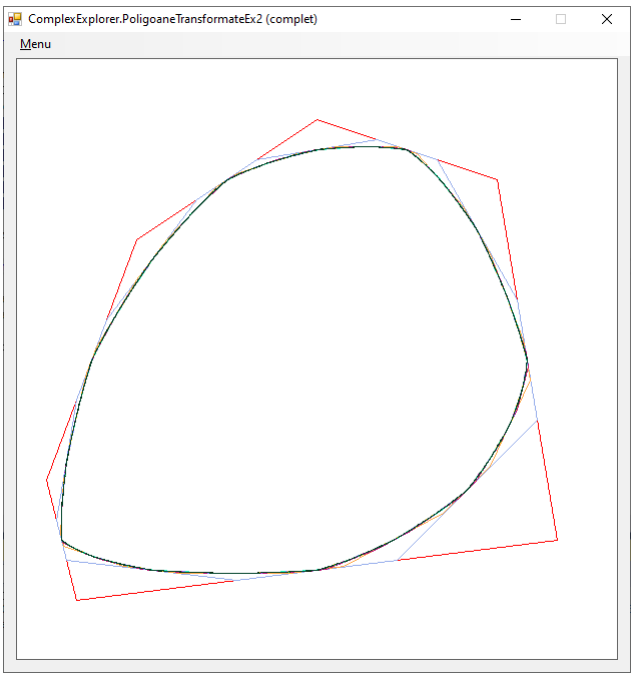
produce următoarea imagine greşită:



Priviţi cu atenţie colţul din stânga-jos: toate poligoanele interioare au câte un colţ în aer. De ce?

Dintre cele patru metodele descrise mai sus, cea mai simplă şi mai sigură, şi care poate fi folosită în orice situaţie, este **transformaPrinRef()**, pe care o recomand, chiar dacă uneori nu este şi cea mai eficientă. Rezolvaţi acest exerciţiu:

Exemplul 2. *Pe fiecare latură a unui un poligon dat se consideră cele două puncte intermediare care împart latura în trei părţi egale. Să se traseze poligonul cu vârfurile în aceste puncte şi apoi să se repete de mai multe ori această operaţie pentru fiecare poligon nou obţinut.*



Altfel spus: tăiem colţurile n-gonului iniţial, obţinem un $2n$-gon. Tăiem şi colţurile acestuia, obţinem un $4n$-gon, şi aşa mai departe. Acum nici să vrem nu mai putem trimite la prelucrat tabloul **baza** prin valoare, deoarece tabloul rezultat are altă dimensiune faţă de cel iniţial.

Urmăriţi rezolvarea:

```
public class PoligoaneTransformateEx2 : ComplexForm
    {
        void transformaPrinRef(ref Complex[] t)
        {
            int n = t.Length;
            Complex[] rez = new Complex[2 * n];
            Complex delta;
            for (int k = 0; k < n - 1; k++)
            {
                delta = t[k + 1] - t[k];
                rez[2 * k] = t[k] + delta / 3;
                rez[2 * k + 1] = t[k] + 2 * delta / 3;
            }
            delta = t[0] - t[n - 1];
            rez[2 * n - 2] = t[n - 1] + delta / 3;
            rez[2 * n - 1] = t[n - 1] + 2 * delta / 3;
            t = rez;
        }
```

```
        void traseazaNgon(Complex[] t, Color col)
        {
            int n = t.Length;
            for (int k = 0; k < n - 1; k++)
            {
                setLine(t[k], t[k + 1], col);
            }
            setLine(t[n - 1], t[0], col);
        }

        public override void makeImage()
        {
            Complex i = new Complex(0, 1);

            setXminXmaxYminYmax(0, 10, 0, 10);
            ScreenColor = Color.White;
            initScreen();
            Complex[] baza = new Complex[6];
            baza[0] = 1 + i;
            baza[1] = 9 + 2 * i;
            baza[2] = 8 + 8 * i;
            baza[3] = 5 + 9 * i;
            baza[4] = 2 + 7 * i;
            baza[5] = 0.5 + 3 * i;

            traseazaNgon(baza, Color.Red);

            for (int k = 0; k < 10; k++)
            {
                transformaPrinRef(ref baza);
                traseazaNgon(baza, getColor(100 + 200 * k));
            }
            resetScreen();
        }
    }
```

În concluzie: când aveţi de prelucrat în mod consistent un tablou printr-o metodă de tip **void**, îl trimiteţi prin referinţă şi nu prin valoare. În prima sau a doua linie a metodei declaraţi şi iniţializaţi din start tabloul rezultat, **rez**, în care depuneţi elementele nou calculate şi, în ultima linie de cod, îl atribuiţi variabilei primite prin referinţă[6].

Stilul de muncă de până acum, prelucrarea tablourilor cu metode de tip **void**, este o reminescenţă din C/C++, unde funcţiile nu pot returna tablouri. În C# această restricţie nu mai apare, tablourile fiind obiecte de tip clasă. Am văzut deja o astfel de metodă:

```
        Complex[] poligonulInitial()
        {
            return new Complex[] { 1 + i, 9 + 2 * i };
        }
```

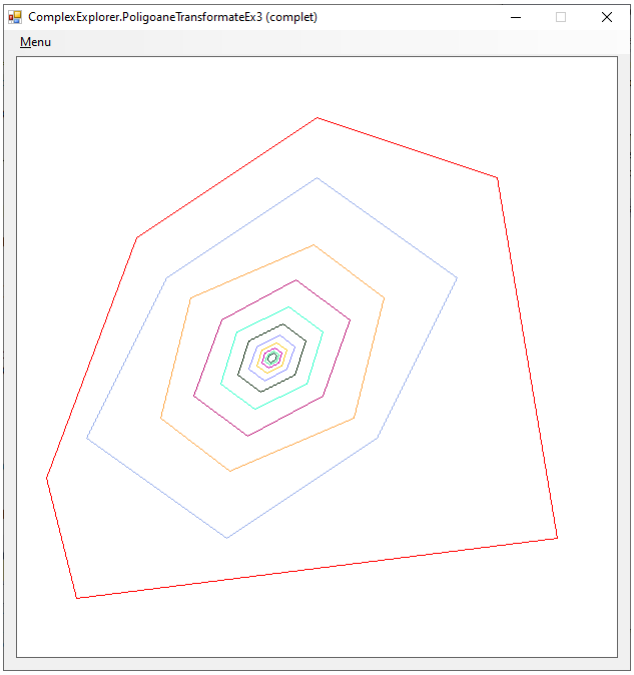
[6] Acest mod de acţiune este valabil pentru orice tip de obiecte, nu numai pentru tablouri.

Rezolvarea precedentă poate fi schimbată imediat în una care foloseşte pentru prelucrare o metodă care returnează tablouri; pur şi simplu trebuie mutată doar o singură atribuire:

```
public class PoligoaneTransformateExx2 : ComplexForm
 {
     Complex[] poligonulTransformat(Complex[] t)
     {
         int n = t.Length;
         Complex[] rez = new Complex[2 * n];
         Complex delta;

         for (int k = 0; k < n - 1; k++)
         {
             delta = t[k + 1] - t[k];
             rez[2 * k] = t[k] + delta / 3;
             rez[2 * k + 1] = t[k] + 2 * delta / 3;
         }
         delta = t[0] - t[n - 1];
         rez[2 * n - 2] = t[n - 1] + delta / 3;
         rez[2 * n - 1] = t[n - 1] + 2 * delta / 3;

         return rez; // in loc de  t = rez; o atribuire in minus
     }
     void traseazaNgon(Complex[] t, Color col)
     {
         int n = t.Length;
         for (int k = 0; k < n - 1; k++)
         {
             setLine(t[k], t[k + 1], col);
         }
         setLine(t[n - 1], t[0], col);
     }

     public override void makeImage()
     {
         Complex i = new Complex(0, 1);
         setXminXmaxYminYmax(0, 10, 0, 10);
         ScreenColor = Color.White;
         initScreen();
         Complex[] baza = new Complex[6];
         baza[0] = 1 + i;
         baza[1] = 9 + 2 * i;
         baza[2] = 8 + 8 * i;
         baza[3] = 5 + 9 * i;
         baza[4] = 2 + 7 * i;
         baza[5] = 0.5 + 3 * i;
         traseazaNgon(baza, Color.Red);
         for (int k = 0; k < 10; k++)
         {
             baza = poligonulTransformat(baza); //o atribuire in plus
             traseazaNgon(baza, getColor(100 + 200 * k));
         }
         resetScreen();
     }
 }
```

În final, o temă pentru acasă:

Exemplul 3. *Să se traseze poligonul obţinut înlocuind fiecare vârf al unui poligon dat cu centrul de greutate al triunghiului format de vârful curent şi cele două vârfuri vecine lui în poligonul iniţial. Să se repete de mai multe ori această transformare pentru fiecare poligon nou obţinut.*



Observaţie: de la un loc încolo hexagoanele interioare par a avea laturile opuse paralele două câte două. E un fapt[7] sau numai o aparenţă? Justificaţi răspunsul.

[7] vezi https://ro.wikipedia.org/wiki/Fapt