Structuri versus clase în C#

În limbajul C++, după cum se ştie, diferenţele dintre clase şi structuri sunt nesemnificative (constau numai în modurile implicite de acces public/privat), dar această afirmaţie nu mai este valabilă în C#, după cum vom exemplifica aici.

Dacă rulăm următorul program C# în mod consolă

```
using System;
namespace Exemplu
{
    struct Punct
    {
        public int x, y;
        public Punct(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            int dim = 10;
            Punct[] tab = new Punct[dim];
            Punct p = new Punct(0, 0);
            for (int i = 0; i < dim; i++)
            {
                p.x = i;
                tab[i] = p;

            }
            for (int i = 0; i < 10; i++)
            {
                Console.Write(tab[i].x + " ");
            }
            //0 1 2 3 4 5 6 7 8 9 Press any key to continue . . .
        }
    }
}
```

obţinem rezultatul trecut în comentariu, nimic surprinzător, dar dacă schimbăm numai cuvântul cheie `struct` cu `class` obţinem un cu totul alt rezultat:

```
using System;
namespace Exemplu
{
    class Punct
    {
        public int x, y;
        public Punct(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int dim = 10;
            Punct[] tab = new Punct[dim];
            Punct p = new Punct(0, 0);
            for (int i = 0; i < dim; i++)
            {
                p.x = i;
                tab[i] = p;

            }
            for (int i = 0; i < 10; i++)
            {
                Console.Write(tab[i].x + " ");
            }
            //9 9 9 9 9 9 9 9 9 9 Press any key to continue . . .
        }
    }
}
```

Pentru a înţelege de ce în C# clasele au alt comportament faţă de structuri, trebuie să revedem implementarea lor în limbajul în care au apărut, C++.

1. Clase C++.

Limbajul C++ a fost proiectat astfel încât tipurile definite de utilizator (**enum**, **struct, union** şi **class**) să se comporte exact ca tipurile fundamentale, cum ar fi **char** sau **int**. Mai precis, simpla declarare a unui obiect dintr-o anumită clasă (structură) provoacă alocarea spaţiului necesar pentru toate variabilele membru (şi a pointerilor interni de acces la funcţiile membru) iar iniţializarea şi atribuirea obiectelor se execută prin copiere membru cu membru:

```
#include<iostream>
using namespace std;
class Complex{
public:
      double x;
      double y;
};
int main(void){
      Complex z1,z2;          //    alocare statica, pe stiva
      z1.x=1;
      z1.y=2;
      z2=z1;
      cout<<z2.x<<endl;       //1
      z1.x=11;
      cout<<z2.x<<endl;       //1   copiere membru cu membru

      Complex *p1,*p2;        //    alocare dinamica stack & heap
      p1=new Complex;
      p1->x=100;
      p1->y=200;
      p2=p1;
      cout<<p2->x<<endl;      //100
      p1->x=111;
      cout<<p2->x<<endl;      //111 copiere de suprafata (shallow copy)
```

```
}
```

Verificăm că în C++ clasele şi structurile se comportă la fel:

```
#include<iostream>
using namespace std;

struct Complex{
public:
	double x;
	double y;
};
int main(void){
	Complex z1,z2;
	z1.x=1;
	z1.y=2;
	z2=z1;
	cout<<z2.x<<endl; //1
	z1.x=11;
	cout<<z2.x<<endl; //1
	Complex *p1,*p2;
	p1=new Complex;
	p1->x=100;
	p1->y=200;
	p2=p1;
	cout<<p2->x<<endl;	//100
	p1->x=111;
	cout<<p2->x<<endl;	//111

}
```

Acest mod de atribuire între obiecte - prin copiere membru cu membru (*memberwise copy*) - nu mai este mulţumitor în cazul în care printre variabilele membru avem şi pointeri care ţintesc către zone alocate dinamic. Simpla copiere a valorilor acestor pointeri creează "balauri cu două capete", cei doi pointeri ţintind către acelaşi corp din heap. Spunem că avem o copiere de suprafaţă (*shalow copy*). Pentru creearea de clone distincte (*deep copy*) trebuie să definim explicit un *constructor de copiere* şi să *supraîncărcăm operatorul de atribuire*. Constructorul de copiere va fi apelat automat la iniţializarea obiectelor şi la transmiterea prin valoare a lor între funcţii, dar nu şi la atribuire:

```
#include<iostream>
using namespace std;

class Vector{                          //<=> struct Vector{
public:
	int dim;
	int* tab;

	Vector(const int dim=1){
		if(0<dim && dim<1000001)
			this->dim=dim;
		else
			this->dim=1;
		this->tab=new int[dim];
		cout<<"CONSTRUCTORUL PRINCIPAL!"<<endl;
	}
```

```
	Vector(const Vector& old){	//constructorul de copiere
		dim=old.dim;
		tab=new int[dim];
		for(int i=0;i<dim;i++) tab[i]=old.tab[i];
		cout<<"CONSTRUCTORUL DE COPIERE!"<<endl;
	}

	Vector& operator=(const Vector& old){ //redefinirea atribuirii
		if(this!=&old) {
			dim=old.dim;
			delete[] tab;		//eliberam vechiul corp
			tab=new int[dim]; //si ocupam unul cu noul dim
			for(int i=0;i<dim;i++) tab[i]=old.tab[i];
		}
		cout<<"OPERATORUL \"=\" SUPRAINCARCAT"<<endl;
		return *this;
	}

	~Vector(void){			//destructor obligatoriu, trebuie sa
		delete[] tab;		//eliberam zona alocata cu new in heap
		cout<<"DESTRUCTORUL!"<<endl;
	}
};

int sumaVal(Vector v){
	int s=0;
	for(int i=0;i<v.dim;i++) s+=v.tab[i];
	v.tab[0]=0; //pentru testare
	return s;
}

int sumaRef(Vector& v){
	int s=0;
	for(int i=0;i<v.dim;i++) s+=v.tab[i];
	v.tab[0]=0; //pentru testare
	return s;
}

int main(void){
	cout<<"\n>ZERO initializarea lui v1"<<endl;
	Vector v1(3);
	v1.tab[0]=1; v1.tab[1]=10;	v1.tab[2]=100;

	cout<<"\n>UNU initializarea lui v2"<<endl;
	Vector v2=v1; //initializarea apeleaza constructorul de copiere

	cout<<"\n>DOI v2 este o clona a lui v1"<<endl;
	cout<<v1.tab[2]<<"=="<<v2.tab[2]<<endl;
	v1.tab[2]=222;
	cout<<v1.tab[2]<<"!="<<v2.tab[2]<<endl;

	cout<<"\n>TREI transmiterea prin valoare a obiectelor"<<endl;
	cout<<"suma="<<sumaVal(v1)<<endl;
		//pentru depunerea pe stiva a valorii lui v1
		//este apelat automat constructorul de copiere!
		//la iesirea din functie clona este distrusa
	cout<<v1.tab[0]<<endl;
		//v1 este neschimbat pentru ca a fost transmis prin valoare

	cout<<"\n>PATRU transmiterea prin referinta a obiectelor"<<endl;
	cout<<"suma="<<sumaRef(v1)<<endl;
```

```
            //nu mai este apelat constructorul de copiere
            //si nici destructorul
      cout<<v1.tab[0]<<endl;
            //v1 este schimbat pentru ca a fost transmis prin referinta

      cout<<"\n>CINCI clonare prin redefinirea atribuirii"<<endl;
      Vector v3;  //apeleaza constructorul fara parametri
      v3=v1;      //Atentie: atribuirea nu invoca constructorul de copiere!
                  //pentru clonare atribuirea trebuie redefinita
      cout<<"\n>SASE v3 este o clona a lui v1"<<endl;
      cout<<v1.tab[2]<<"=="<<v3.tab[2]<<endl;
      v1.tab[2]=666;
      cout<<v1.tab[2]<<"!="<<v3.tab[2]<<endl;

      cout<<"\nSAPTE: iesirea din main, sunt distruse v1, v2 si v3"<<endl;
      return 0;
}
/*
>ZERO initializarea lui v1
CONSTRUCTORUL PRINCIPAL!

>UNU initializarea lui v2
CONSTRUCTORUL DE COPIERE!

>DOI v2 este o clona a lui v1
100==100
222!=100

>TREI transmiterea prin valoare a obiectelor
CONSTRUCTORUL DE COPIERE!
DESTRUCTORUL!
suma=233
1

>PATRU transmiterea prin referinta a obiectelor
suma=233
0

>CINCI clonare prin redefinirea atribuirii
CONSTRUCTORUL PRINCIPAL!
OPERATORUL "=" SUPRAINCARCAT

>SASE v3 este o clona a lui v1
222==222
666!=222

SAPTE: iesirea din main, sunt distruse v1, v2 si v3
DESTRUCTORUL!
DESTRUCTORUL!
DESTRUCTORUL!
Press any key to continue . . .
*/
```

In concluzie, în C++, dacă nu dotăm în mod explicit o clasă (sau structură) cu un constructor de copiere, compilatorul generează un constructor de copiere implicit care copie obiectele membru cu membru (utilizând modul de copiere adecvat tipului fiecarei variabie membru), iar dacă nu supraîncărcăm operatorul de atribuire, şi atribuirea se execută membru cu membru.

2. Clase C#

Comportamentul *de tip valoare* descris mai sus al claselor şi structurilor din C++ duce la încărcarea rapidă a stivei de execuţie, care este mult mai restrânsă decât zona de memorie liberă, *heap*-ul. Din acest motiv modul uzual de lucru în C++ cu obiecte (instanţe ale claselor sau ale structurilor) constă în alocarea lor dinamică cu operatorul **new** şi accesarea lor indirectă prin pointeri. Alocarea dinamică a memoriei este foarte eficientă dar presupune atenţie mărită din partea programatorului la alocarea şi mai ales la dealocarea acesteia în timpul rulării.

In C#, după modelul limbajului Java, a fost generalizată alocarea dinamică a claselor prin introducerea comportamentului *de tip referinţă* al acestora: simpla declarare a unui obiect (instanţă a unei clase) provoacă doar alocarea statică (pe stivă, la compilare) a unei "variabile referinţă" (un pointer de fapt) care are atribuit numai spaţiul strict necesar (32 de biţi) pentru păstrarea unei referinţe (o adresă) care să permită accesarea zonei de memorie alocate ulterior obiectului cu operatorul **new**. Eliberarea memoriei a fost simplificată la maximum (tot după modelul limbajului Java): nu mai există operatorul **delete**, dealocarea este executată automat prin numărarea referinţelor către obiecte (tehnica *reference counting*) şi apelarea din când în când a programului specializat de curăţare a memoriei, *garbage collector*-ul.

Spre deosebire de Java unde s-a renunţat complet la pointeri şi la structuri, în C# pointerii au fost reţinuţi. Ei mai pot fi folosiţi dar cu restricţii, pentru a nu încurca delocarea automată a heap-ului; în plus, porţiunile de cod unde apar trebuie declarate "nesigure" cu cuvântul cheie **unsafe**. Au fost reţinute şi structurile din C++ cu tot cu comportamentul lor de tip valoare, având astfel acum un comportament complet diferit de cel al claselor. In C# instanţele de tip structură sunt *valori* iar cele de tip clasă sunt *referinţe.* In plus, structurile în C#, spre deosebire de cele din C++, nu suportă moştenirea.

In cazul în care dorim să copiem în întregime obiectele dintr-o clasă, putem defini un constructor de copiere de tipul celui din C++, dar care acum trebuie apelat manual:

```
using System;
namespace ConsoleApplication1 {
    class Punct
    {
        public double x, y, z;
        //constructorul obisnuit
        public Punct(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
        //constructorul de copiere
        public Punct(Punct old)
        {
            this.x = old.x;
            this.y = old.y;
            this.z = old.z;
        }
        //public Punct(Punct old)
        //{
        //    this = old;     //Error     Cannot assign to '<this>'
                              //because it is read-only
        //}
    }
```

```
struct Complex
{
    public double x, y;
    public Complex(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    //Constructor de copiere: (inutil)
    //public Complex(Complex z)
    //{
    //    this.x = z.x;
    //    this.y = z.y;
    //}
    //Constructor de copiere: (inutil)
    public Complex(Complex z)
    {
        this = z;
    }

}
class Program
{
    static void test0(Punct pp, Complex zz)
    {
        pp.x = 0;
        zz.x = 0;
    }
    static void test555(Punct pp, Complex zz)
    {
        pp = new Punct(555, 555, 555);
        zz = new Complex(555, 555);
    }
    static void test0ref(ref Punct pp, ref Complex zz)
    {
        pp.x = 0;
        zz.x = 0;
    }
    static void test555ref(ref Punct pp, ref Complex zz)
    {
        pp = new Punct(555, 555, 555);
        zz = new Complex(555, 555);
    }
    static void Main(string[] args)
    {
        //clase:

        Punct p1, p2, p3;
        //p1.x = 100;   //      Use of unassigned local variable 'p1'
        p1 = new Punct(100, 200, 300);  //alocam corpul in heap
        p2 = p1;
        Console.WriteLine(p2.x);        //100
        p1.x = 111;
        Console.WriteLine(p2.x);        //111, shallow copy

        p3 = new Punct(p1);             //apelarea manuala a
                                        //constructorului de copiere
        Console.WriteLine(p3.x);        //111
        p1.x = 300;
        Console.WriteLine(p3.x);        //111, deep copy.
```

```
//structuri:

Complex z1, z2, z3;     //alocare statica
z1.x = 100;
//z2 = z1;              //Use of unassigned local variable 'z1'
z1.y = 200;
z2 = z1;                //initializare fara new
Console.WriteLine(z2.x);        //100
z1.x = 111;
Console.WriteLine(z2.x);        //100, deep copy

z1 = new Complex(100, 200);
z2 = new Complex(z1);
z1.x = 111;
Console.WriteLine(z2.x);        //100, deep copy

z3 = new Complex();
Console.WriteLine(z3.x);        //0
                        //initializare implicita cu zero!


Punct p = new Punct(1, 1, 1);
Complex z = new Complex(1, 1);
test0(p, z);
Console.WriteLine("test0");
Console.WriteLine(p.x);         //0
//p si pp tintesc acelasi corp
//modificarea din heap facuta prin pp
//este persistenta si este accesibila prin p
Console.WriteLine(z.x);         //1
//corpul lui z este copiat pe stiva
//z ramane nemodificat

p.x = 1;
z.x = 1;
test555(p, z);
Console.WriteLine("test555");
Console.WriteLine(p.x);         //1,
//capul lui p este copiat pe stiva in pp
//noua valoare a lui pp (pe stiva)
//nu se pastreaza la revenirea din functie,
//noul corp tintit de pp este  aruncat la gunoi.
Console.WriteLine(z.x);         //1
//corpul lui z este copiat pe stiva
//z ramane nemodificat

p.x = 1;
z.x = 1;
test0ref(ref p, ref z);
Console.WriteLine("test0ref");
Console.WriteLine(p.x);         //0
Console.WriteLine(z.x);         //0
//test0ref utilizeaza chiar pe p si z

p.x = 1;
z.x = 1;
test555ref(ref p, ref  z);
Console.WriteLine("test555ref");
Console.WriteLine(p.x);     //555
Console.WriteLine(z.x);     //555
//test555ref utilizeaza chiar pe p si z
```

```
            //vechiul corp al lui p este aruncat la gunoi
        }
    }
}
```

Să observăm că, spre deosebire de clase, structurile pot fi instanţiate şi fără utilizarea operatorului **new**, dar în acest caz structura nu poate fi folosită până când nu sunt iniţializate manual toate câmpurile componente (vezi prima încercare de atribuire `z2 = z1` de mai sus).

O altă modalitate de copiere constă în moştenirea interfeţei `ICloneable` prin implementarea corespunzătoare a metodei `Clone()`, în conformitate cu cerinţele clasei proiectate.

In concluzie, deoarece în C# obiectele de tip structură sunt valori alocate static (pe stivă sau *in line*) iar obiectele de tip clasă sunt referinţe alocate dinamic (cu **new**) în heap, pentru optimizarea raportului viteză de execuţie/ocuparea stivei vom adopta următoarea strategie: obiectele "uşoare", cu un număr redus de câmpuri, pentru care nu avem în vedere derivarea lor în clase descendente ci folosirea lor intensivă într-un număr mare de operaţii, vor fi proiectate ca structuri, urmând ca obiectele "grele" să fie de tip clasă.

Pentru a înţelege mai bine comportamentul *obiectelor de tip referinţă* – clasele din C# şi Java – vom prezenta în continuare o încercare de a construi o versiune C++ a clasei Punct de mai sus. Vom defini o structură Corp formată numai din membrii x, y şi z, şi o clasă Punct cu un singur câmp, un pointer către un Corp. Vom dota clasa cu un constructor adecvat, care să aloce dinamic structura corp:

```
#include<iostream>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>

using namespace std;
struct Corp{ public: int x,y,z; };
class Punct{
public:
	Corp *cap;
	Punct(int x=0, int y=0, int z=0){
		cap=new Corp;
		cap->x=x;
		cap->y=y;
		cap->z=z;
	}
};
void test(){
	Punct p(15,15,15);
	cout<<p.cap->x<<endl;	//15
	Punct q=p;  //initializare
	Punct qq;
	qq=p;		//atribuire
	p.cap->x=666;
	cout<<p.cap->x<<endl;	//666
	cout<<q.cap->x<<endl;	//666
	cout<<qq.cap->x<<endl;	//666
}
int main(void){
	test();
	_CrtDumpMemoryLeaks();
	return 0;
}
/*
Detected memory leaks!
Dumping objects ->
```

```
{148} normal block at 0x002E5128, 12 bytes long.
 Data: <            > 00 00 00 00 00 00 00 00 00 00 00 00
{127} normal block at 0x002E5048, 12 bytes long.
 Data: <            > 9A 02 00 00 0F 00 00 00 0F 00 00 00
Object dump complete.*/
```

Observăm că am obţinut exact comportamentul de tip referinţă al claselor C#, totuşi în C++ avem o problemă: au apărut scurgeri de memorie deorece nu am dotat clasa Punct cu destructorul necesar dealocării efectuate cu **new** în constructor.

Mai precis, la terminarea apelului funcţiei **test()** au dispărut de pe stivă pointerii p.cap, q.cap şi qq.cap dar au rămas ocupate în heap cele două corpuri obţinute prin declaraţiile lui p şi qq. Corpul lui p are adresa 0x002E5048 iar al lui qq are adresa 0x002E5128. Declaraţia cu iniţializare `Punct q=p;` a apelat constructorul implicit de copiere, care a alocat pe stivă loc pentru q.cap şi i-a atribuit valoarea lui p.cap, fără nici o altă alocare de memorie în heap.

Dacă dotăm clasa Punct cu destructorul adecvat

```
~Punct(){
	delete cap;
}
```

vom avea alte necazuri: la sfârşitul apelului funcţiei **test()** toţi cei trei pointeri ţintesc corpul lui p, care va fi dealocat de trei ori, eroare. In plus corpul iniţial (cel cu x=y=z=0) al lui qq a rămas orfan după atribuirea qq=p şi nu mai poate fi dealocat.

In C# managementul memoriei rezolvă automat toate problemele de mai sus, dar în C++ trebuie să schimbăm comportamentul clasei Punct la iniţializare şi atribuire, prin dotarea ei cu un constructor explicit de copiere şi prin supraîncărcarea corespunzătoare a operatorului de atribuire:

```
#include<iostream>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
using namespace std;
struct Corp{ public: int x,y,z;};
class Punct{
public:
	Corp *cap;
	Punct(int x=0, int y=0, int z=0){
		cap=new Corp;
		cap->x=x;
		cap->y=y;
		cap->z=z;
	}
	Punct(const Punct& old){            //constructorul  de copiere
		cap=new Corp;
		cap->x=old.cap->x;
		cap->y=old.cap->x;
		cap->z=old.cap->x;
	}
	Punct& operator=(const Punct& old){ //supraincarcarea atribuirii
		if(this!=&old) {
			cap->x=old.cap->x;
			cap->y=old.cap->x;
			cap->z=old.cap->x;
		}
		return *this;
	}
	~Punct(void){
```

```
            delete cap;
        }
};


void test(){
        Punct p(15,15,15);
        cout<<p.cap->x<<endl;    //15
        Punct q=p;  //initializare
        Punct qq;
        qq=p;       //atribuire
        p.cap->x=666;
        cout<<p.cap->x<<endl;    //666
        cout<<q.cap->x<<endl;    //15
        cout<<qq.cap->x<<endl;   //15
}
int main(void){
        test();
        _CrtDumpMemoryLeaks();
        return 0;
}
```

Nu mai apar scurgeri de memorie şi nici nu mai avem dealocare multiplă. Problemele au dispărut deorece atât iniţializarea cât şi atribuirea se efectuează prin clone distincte.

Observăm că versiunea C++ cu alocare dinamică a clasei Punct este mult mai eficientă decât versiunea C#: amândouă alocă static numai un pointer pe stivă şi restul câmpurilor în heap, dar în C++, prin utilizarea facilităţilor oferite de apelarea automată a constructorului de copiere şi de supraîncărcarea atribuirii, s-a reuşit şi obţinerea comportamentului de tip valoare (copiere în profunzime, realizarea de clone distincte, etc). Preţul plătit: multă migală în proiectarea clasei, orice eroare de alocare/dealocare fiind fatală.

In final, reamintim că în Java nici un operator nu poate fi supraîncărcat, în C++ aproape toţi pot fi supraîncărcaţi iar în C# numai câţiva, printre care operatorii aritmetici şi relaţionali, dar nu şi operatorul de atribuire.