Universitatea Politehnică Bucureşti

Facultatea de Electronică, Telecomunicații şi Tehnologia Informației

Tema :

Programare orientata pe obiect din perspectiva ingineriei software :

Initializarea si distrugerea obiectelor

Zanfir Catalin
441A

Cuprins:

Initializarea si distrugerea obiectelor in C++

Constructorul.

Constructorul in limbajul C++ reprezinta o metoda speciala, este membru al unei clasei si trebuie sa aiba acelasi nume ca si clasa respectiva. Ei sunt apelati atunci cand se instantiaza obiecte din clasa respectiva, ei sunt cei responsabili de initializarea corecta a tuturor variabilelor membru ale unui obiect si garantand ca initializarea unui obiect se efectueaza o singura data.

Constructorul este, deci, o functie membra speciala a unei clase ce se apeleaza in mod automat la crearea unui obiect.

O clasa poate avea un numar infinit de constructori, insa trebuie sa fie diferiti prin tipul si numarul de parametri. Compilatorul foloseste constructorul potrivit in functie de parametrii pe care ii contine instantierea obiectului, adica tipul si numarul de parametri .

Tipuri de constructori

Exista doua tipuri de constructori :

- constructor implicit (“default constructor”);
- constructor de copiere (“copy constructor”);

Constructorii impliciti pot fi :

- constructori definiti de programator;
- generarati de catre compilator. Daca programatorul nu defineste pentru o clasa un constructor, clasa e trata ca o clasa fara constructor, iar compilatorul genereaza automat un constructor ce nu are parametrii, si nici corp de instructiuni.

exemplu constructor implicit :

```
class Complex {
        …
        public:
                Complex(){
                        re = 0;
                        im = 0;
                        printf (“Apel constructor\n”);
                            }
                };
…
        Complex z;
        z.afisare();
```

Iesire este : 0 + 0i

Tipuri de constructori

- Constructori cu parametri
  - Cu parametrii ce nu iau valori implicite
  - Cu parametrii ce nu iau valori implicite

- Functii cu parametrii impliciti : tip _r nume_functie (tip_1 p_1, … , tip_n p_n = vi_{n+1} ,… , tip_m p_m = vi_m); unde p_{n+1} , …, p_m = parametrii impiciti; la apelul functiei acestia pot sa lipseasca, caz in care ei au valori implicite specificate de declarare

Exemplu : constructor cu parametri impliciti :

```
class Complex {
        …
        public:
                Complex(float r = 0, float i = 0)
                {       re = r;
                        im = i;

                }

…
               };
…
Complex z1(2,3), z2(4), z3=5, z4;
z.afisare();
…
```

Iesirea este : 2+3i , 4 + 0i, 5+ 0i , 0 + 0i

Exemplu constructor cu parametri:

```
class Persoana
{       private:
                char *nume;
                int varsta;
        public:
                Persoana(char *n, int v){
                        nume = new char[strlen(n)+1];
                        strcpy(nume, n);
                        varsta = v;    }
```

```
void afisare()
{       cout<<"Nume:"<<nume<<endl;
        cout<<"Varsta:"<<varsta<<endl;
}
void setNume(char *n) {
        if(strlen(nume)<strlen(n)){
                        delete nume;
                         = new char[strlen(n)+1];  }
        strcpy(nume, n);  }
void setVarsta(int v){  varsta = v;  }  };
int main(){
        Persoana p1("Mihai",21);
        Persoana p2=p1;
        p1.afisare();
        p2.afisare();
        p2.setNume("Misu");
        p2.setVarsta(24);
        p1.afisare();
        p2.afisare();
        getch();
}
```

Acest exemplu are ca iesire :

P1.afisare : Nume : Mihai , Varsta= 21
P2.afisare: Nume: Mihai , Varsta =21
P1: afisare : Nume: Misu, Varsta 21
P2.afisare: nume : Misu , Varsta 24.

Constructori de copiere:

Un alt tip de constructor este constructorul de copiere. Ei pot fi definiti de programator sau poti fi generati automat de catre compilator.

Constructorul de copiere in modul sau obisnuit, default, copiaza membru cu membru toate variabilele din argumentul obiectului care apelaza metoda respectiva. Compilatorul va genera, in mod automat, un constructor de copiere in fiecare clasa in care programatorul nu a declarat unul in mod explicit.

Sintaxa constructor de copiere:

```
class IdNumeClasa {
        . . .
        IdNumeClasa (IdNumeClasa &ob);
        sau
        IdNumeClasa (const IdNumeClasa &ob);
        . . .
};
```

Exemplu constructor de copiere:

```
class Complex {
        …
        public:
                Complex(const Complex &z)

                {       re = z.re;
                        im = z.im;
                        printf("Apel constructor de copiere\n") ;    }
…
                };
void f(Complex z){
        ...
                }
        …
Complex z1(2,3);
        z1.afisare();
        Complex z2 = z1;
        z2.afisare();
        f(z1);
```

produce iesirea:

```
2+3*i
Apel constructor de copiere
2+3*i
Apel constructor de copiere
```

Destructorul

Destructorul este opusul constructorului. Are acelasi nume ca si clasa careia ii apartine, dar este precedat de "~". Constructorii sunt utilizati in mod special pentru a aloca memorie si pentru a efectua anumite operatii, iar destructorii se utilizeaza pentru eliberarea memoriei alocate de constructori si pentru efectuarea unor operatii inverse.

Destructorul, deci, este o functie membră specială a unei clase ce apelează în mod automat distrugerea unui obiect ce are rolul de a elibera zonele alocate dinamic, a resurselor.

Este de 2 tipuri : definit de utilizator si generat de compilator.

Destructori - Caracteristici

- Trebuie sa aiba acelasi numele cu numele clasei si este precedat de ~
- Nu necesita parametri
- Nu va returna nimic (nici macar void)
- Spre deosebire de construtori, o clasa poate avea un singur destructor
- Pot fi functii virtuale

Sintaxa destructor:

```
class IdNumeClasa {
        . . .
~IdNumeClasa ();
        . . .
                };
IdNumeClasa::~IdNumeClasa (){
        //instructiuni  }
```

Exemplu:Destructor (I)

```
class Complex {
        …
        public:
                void ~Complex()
                {
                printf("Distrugere obiect:");
                afisare();
                }
…
                };
void main()
{       Complex z1(2,3), z2(4,7);
        z1.afisare();
        z2.afisare();
         }
```

Iesirea este : 2+3*i
4+7*i
Distrugere obiect: 4+7*i
Distrugere obiect: 2+3*i

Destuctorii sunt apelati implicit in doua situatii:

1. cand se elibereaza memorie alocata dinamic, folosind operatorul “delete” (a se vedea linia 10 din programul de mai jos);

2. la parasirea domeniului de existenta al unei variabile (vezi linia 17, variabila pb). Daca in al doilea caz este vorba de variabile globale sau definite in “main”, distrugerea lor se face dupa ultima instructiune din “main”, dar inainte de incheierea executiei programului.

Utilizatorul dispune de doua moduri pentru a apela un destructor:

1. prin specificarea explicita a numelui sau – metoda directa;

Exemplu:

```
class B
{          public:    ~B();
};



void  main (void)
{     B b;
       b.B::~B();   // apel direct : e obligatoriu prefixul "B::"
}
```

2. folosind operatorul “delete” (metoda indirecta – a se vedea linia 10 din programul urmator).

```
#define NULL 0
struct s
{   int nr;
    struct s *next;
};
class B
{    int i;
            struct s *ps;
           public:    B (int);
                      ~B (void);
};
```

```
            B :: B (int ii = 0)                                  // 3 si 7
{     ps = new s; ps->next = NULL; i = ps->nr = ii;              // 4 si 8
}                                                                // 5 si 9
            B :: ~B (void)                                       // 11 si 14
{      delete ps;                                                // 12 si 15
}                                                                // 13 si 16
           void main (void)                                      // 1
{            B *pb;
             B b = 9;                                            // 2
             pb = new B(3);                                      // 6
             delete pb;                                          // 10
}                                                                 // 17
```

Se mentioneaza ordinea efectuarii operatiilor in comentarii.

Initializarea si distrugerea obiectelor in JAVA

Crearea obiectelor in java, ca in orice limbaj de programare orientat –obiect , crearea obiectelor se realizeaza prin instatierea unei clase si implica urmatoarele lucruri

- Declararea :

Se specifica tipul obiectului,adica se specifica clasa acestuia: NumeClasa numeObiect;

- Instantierea

Se efectueaza prin intermediul operatorului new. Are ca efect crearea efectiva a obiectului cu alocarea spatiului de memorie corespunzator : numeObiect = new NumeClasa();

- Initializarea:

Se efectueaza prin constructorii din clasei respective. Imediat dupa alocarea memoriei prin operatorul new , este apelat constructorul specificat. Parantezele rotunde de dupa numele clasei indica faptul ca acolo este de fapt un apel la unul din constructorii clasei si nu simpla specificare a numelui clasei.

Mai general, instantierea si initializarea apar sub forma:

numeObiect = new NumeClasa([argumente constructor]);

Urmatorul exemplu ,declaram si instantiem doua obiecte din clasa Rectangle , clasa ce descrie suprafete grafice rectangulare, definite de coordonatele coltului stanga sus si latimea, respectiv inaltimea.

```
Rectangle r1, r2;
r1 = new Rectangle();
r2 = new Rectangle(0, 0, 100, 200);
```

In primul caz Rectangle() este un apel al constructorului implicit, nu are parametrii, nu are instuctiuni. r2 este un apel al unui constructor cu 4 parametrii, cu conditia sa existe un astfel de constructor al clasei respective care sa accepte parametrii respectivi.

De exemplu, clasa Rectangle are urmatorii constructori :

```
public Rectangle()
public Rectangle(int latime, int inaltime)
public Rectangle(int x, int y, int latime, int inaltime)
public Rectangle(Point origine)
public Rectangle(Point origine, int latime, int inaltime)
public Rectangle(Point origine, Dimension dimensiune)
```

Declararea unui obiect nu implica sub nici o forma alocarea de spatiu de memorie pentru acel obiect.. Numai prin operatorul new se aloca memorie pentru acel obiect.

Obiectul,dupa ce a fost creat, poate fi folosit pentru : aflarea unor informatii despre obiect, schimbarea starii sale sau executarea unor actiuni. Aceste lucruri se realizeaza prin aflarea sau schimbarea valorilor variabilelor sale, respectiv prin apelarea metodelor sale.

Referirea valorii unei variabile se face prin obiect.variabila. De exemplu, clasa mai sus mentionata Rectangle are variabilele publice x, y, width, height, origin. Aflarea valorilor acestor variabile sau schimbarea lor se face prin constructii :

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
System.out.println(patrat.width);
patrat.x = 10;
patrat.y = 20; //schimba originea
patrat.origin = new Point(10, 20);
```

Accesul la variabilele unui obiect se face in functie de drepturile de acces pe care le ofera variabilele respective celorlalte clase. Apelul unei metode: .metoda([parametri]).

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
patrat.setLocation(10, 20); //schimba originea
patrat.setSize(200, 300); //schimba dimensiunea
```

Valorile variabilelelor nu pot fi modificate direct, sau este descurajata aceasta tehnica, in schimb, ele pot fi modificate indirect prin intermediul metodelor sale. Programarea orientata obiect descurajeaza folosirea directa a variabilelor unui obiect deoarece acesta poate fi adus in stari inconsistente. In schimb, pentru fiecare variabila care descrie starea obiectului trebuie sa existe metode care sa permita modificarea/verificarea valorilor variabilelor sale. Acestea se numesc metode setter-getter si au numele de forma setVariabila , respectiv getVariabila.

```
patrat.width = -100; //stare inconsistenta
patrat.setSize(-100, -200); //metoda setter
```

Deci, constructorii sunt metode speciale ale unei clase care au acelasi nume cu clasa, nu returneaza nimic si sunt folositi pentru initializarea obiectelor acelei clase ın momentul instantierii lor.

```
class NumeClasa {
    [modificatori] NumeClasa([argumente]) {
        // Constructor
        }
    }
```

O clasa poate cel putin 1 constructor, insa poate avea si mai multi, dar ei trebuie sa difere prin lista de argumente/parametrii primiti. In felul acesta sunt permise diverse tipuri de initializari ale obiectelor la crearea lor, in functie de numarul parametrilor cu care este apelat constructorul.

Sa consideram ca exemplu declararea unui clase care descrie notiunea de dreptunghi si 3 posibili constructori pentru aceasta clasa :

```
class Dreptunghi {
    double x, y, w, h;
    Dreptunghi(double x1, double y1, double w1, double h1) {
        // Cel mai general constructor
        x=x1; y=y1; w=w1; h=h1;
        System.out.println("Instantiere dreptunghi");
            }
```

```
Dreptunghi(double w1, double h1)
        {  // Constructor cu doua argumente
           x=0; y=0; w=w1; h=h1;
           System.out.println("Instantiere dreptunghi");
              }
Dreptunghi() {
           // Constructor fara argumente
           x=0; y=0; w=0; h=0;
System.out.println("Instantiere dreptunghi");
                 }
}
```

Constructorii sunt apelati automat la instantierea unui obiect. In cazul in care dorim sa apelam explicit constructorul unei clase folosim expresia : this(argumente) , care apeleaza constructorul corespunzator (ca argumente) al clasei respective. Aceasta metoda este folosita atunci cand sunt implementati mai multi constructori pentru o clasa, pentru a nu repeta secventele de cod scrise deja la constructorii cu mai multe argumente(mai generali). Mai eficient, fara a repeta secvente de cod in toti constructorii , clasa de mai sus poate fi rescrisa astfel:

```
class Dreptunghi
        {  double x, y, w, h;
        Dreptunghi(double x1, double y1, double w1, double h1) {
                // Implementam doar constructorul cel mai general
                x=x1; y=y1; w=w1; h=h1;
                System.out.println("Instantiere dreptunghi");
        }
Dreptunghi(double w1, double h1)
        {       this(0, 0, w1, h1);
                // Apelam constructorul cu 4 argumente
        }
Dreptunghi()
        {       this(0, 0);
                // Apelam constructorul cu 2 argumente
        }   }
```

Dintr-o subclasa putem apela explicit constructorii superclasei cu expresia : super(argumente). Daca dorim sa cream clasa Patrat, derivata din clasa dreptunghi :

```
class Patrat extends Dreptunghi
        {       Patrat(double x, double y, double d)
                {       super(x, y, d, d);
                        // Apelam constructorul superclasei } }
```

Distrugerea obiectelor :

Multe limbaje de programare impun ca programatorul sa tina evidenta obiectelor create si de memoria alocata si sa le distruga obiectele in mod explicit, atunci cand nu mai este nevoie de ele sau are nevoie de memorie, cu alte cuvinte sa gestioneze singur memoria ocupata de obiectele sale.Practica a demonstrat ca aceasta tehnica este una din principalele furnizoare de erori ce duc la functionarea defectuoasa a programelor.

In Java programatorul, nu mai are aceasta sarcina, a distrugerii obiectelor sale, intrucat, in momentul rularii unui program, simultan cu interpretorul Java, ruleaza automat si un proces care are functia de a distrugere obiecte care nu mai sunt necesare, obiecte nefolosite. Acest proces pus la dispozitie de platforma Java de lucru se numeste garbage collector (colector de gunoi), prescurtat gc.

Un obiect este eliminat din memorie de procesul de colectare atunci cand nu mai exista nici o referinta la acesta. Referintele (care sunt de fapt variabile) sunt distruse doua moduri:

- natural, atunci cand variabila respectiva iese din domeniul sau de vizibilitate, de exemplu la terminarea metodei ın care ea a fost declarata;
- explicit, daca atribuim variabilei respective valoare null.

Colectorul de gunoaie este un proces de prioritate scazuta care se executa periodic si marcheaza acele obiecte care au referinte directe sau indirecte. Dupa ce toate obiectele au fost parcurse, cele care au ramas nemarcate sunt eliminate automat din memorie.

Apelul metodei gc din clasa System sugereaza masinii virtuale Java sa ‘depuna eforturi’ in recuperarea memoriei ocupate de obiecte care nu mai sunt folosite , fara a forta, insa, pornirea procesului.

Inainte ca un obiect sa fie eliminat din memorie, procesul gc da acelui obiect posibilitatea sa ‘curete dupa el’, apeland metoda de finalizare a obiectului respectiv. Uzual , in timpul finalizarii un obiect isi inchide fisierele si socket-urile folosite, distruge referintele catre alte obiecte (pentru a usura sarcina colectorului de gunoaie), etc.

Codul pentru finalizarea unui obiect trebuie scris intr-o metoda speciala numita finalize a clasei ce descrie obiectul respectiv.

Spre deosebire de destructorii din C++ , metoda finalize nu are rolul de a distruge obiectul, ci este apelata automat inainte de eliminarea obiectului respectiv din memorie.

Initializarea si distrugerea obiectelor intr-o aplicatie .net C#

Constructor

Un constructor initializeaza un obiect atunci cand este creat . Au aceleasi nume cu clasa din care fac parte. Nu au tip explicit. Au forma generala:

```
acces nume-clasa (parametri)
        {
                // codul constructorului
        }
```

Se utilizeaza pentru atribuirea valorilor initiale pentru variabilele instanta definite in cadrul clasei sau pentru efectuarea altor operatii initiale necesare la crearea unui obiect complet initializat.

C# pune la dispozitie un constructor implicit care initializeaza toate variabilele membru cu zero (pt. tipuri valorice) respectiv cu null (pt. tipuri referinta). (Astfel, in Exemplul 1, imediat dupa crearea obiectelor punct1 si punct2, variabilele x si y ale ambelor obiecte au valorile 0. Apoi acestea sunt reinitializate manual). Daca este definit propriul constructor, cel implicit nu va fi utilizat.

Exemplu : La crearea obiectelor punct1 si punct2 se utilizeaza acest constructor care initializeaza variabilele x si y.

```
using System;
class Point
{
        public double x;
        public double y;
        public Point(double a, double b)
          {
                 x = a;
                y = b;
          }
}
class Segmdr
        {   public static void Main()
                {  Point punct1 = new Point(3,4);
                   Point punct2 = new Point(5,3);
                   double dist;
                  dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y -
                punct2.y) * (punct1.y - punct2.y));
                Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}",
punct1.x, punct1.y,  punct2.x, punct2.y, dist);
                  }
        }
```

Rezultat:

Distanta dintre punctele (3,4) si (5,3) este: 2,24

Utilizand operatorul new se aloca dinamic memorie libera pentru memorarea obiectelor create. Intrucat memoria nu este infinita, operatorul new va esua daca nu mai are memorie disponibila. Se pune astfel problema recuperarii spatiului de memorie din obiectele care nu mai sunt folosite.

Limbajul C# pune la dispozitie o metoda diferita si anume colectarea automata a spatiului neutilizat.

Sistemul de colectare automata recupereaza spatiul ocupat de obiectele care nu mai sunt necesare programului. Daca nu mai exista nici o referinta la un obiect atunci se presupune ca acel obiect nu mai este necesar, iar memoria ocupata de el poate fi eliberata.

Colectarea automata se declanseaza destul de rar pe parcursul executiei programului. Nu se va declansa doar pentru ca exista obiecte care nu mai sunt folosite. Colectarea automata se declanseaza atunci cand se indeplinesc urmatoarele doua conditii: exista o mare necesitate de memorie si exista obiecte care pot fi reciclate.

Colectarea necesita un timp mare. Astfel aceasta se va declansa doar daca este imperios necesar. Nu se poate determina cu exactitate cand are loc colectarea spatiului neutilizat

Destructor

Destructorii au forma generala:

```
~nume-clasa()
        {
                // codul destructorului
        }
```

Destructorul este deci declarat similar cu un constructor, fiind insa precedat de caracterul ~(tilda).

Pentru adaugarea unui destructor la o clasa, il includeti ca si pe orice alt membru al clasei. Acesta va fi apelat inainte ca un obiect din acea clasa sa fie reciclat. In interiorul destructorului, se vor specifica actiunile care trebuie executate inainte ca obiectul sa fie distrus.

Destructorul se apeleaza imediat inaintea colectarii automate. Acesta nu va fiapelat cand se iese din domeniul de valabilitate al obiectului. Lucrul acesta difera de C++, unde destructorii sunt apelati cand domeniul de valabilitate se incheie.Nu se poate determina cu exactitate cand se va executa un destructor.

Mai mult, este posibil ca programul dumneavoastra sa se termine inaintea inceperii operatiei de colectare automata, caz in care destructorul nu va fi apelat deloc.

Managementul memoriei este facut sub platforma .NET in mod automat de catre garbage collector , parte componenta a CLR-ului.

Acest mecanism de garbage collection scuteste programatorul de grija dealocarii memoriei. Dar exista situatii in care se doreste sa se faca management manual al dealocarii resurselor. In C# exista posibilitatea de a lucra cu destructori sau cu metode de tipul Dispose(), Close().

Un destructor nu are modificator de acces, nu poate fi apelat manual, nu poate fi supraincarcat, nu este mostenit.

Un destructor este o scurtatura sintactica pentru metoda Finalize(), care este definita in clasa System.Object. Programatorul nu poate sa suprascrie sau sa apeleze aceasta metoda.

Problema cu destructorul este ca el e chemat doar de catre garbage collector , dar acest lucru se face nedeterminist. Exista cazuri in care programatorul doreste sa faca dealocarea manual astfel incat sa nu astepte ca garbage collectorul sa apeleze destructorul.Programatorul poate sa scrie o metoda care sa faca acest lucru. Se sugereaza definirea unei metode Dispose() care ar trebui sa fie explicit apelata atunci cand resurse de sistem de operare trebuie sa fie eliberate. In plus, clasa respectiva ar trebui sa implementeze interfata System.Idisposable, care contine aceasta metoda.

In acest caz, Dispose() ar trebui sa inhibe executarea ulterioara a destructorului pentru instanta curenta . aceasta manevra permite evitarea eliberarii unei resurse de doua ori. Daca clientul nu apeleaza explicit Dispose() , atunci garbage collector ul va apela el destructorul la un moment dat. Intrucat utilizatorul poate sa nu apeleze Dispose() , este necesar ca tipurile care implementeaza aceasta metoda sa defineasca deasemenea destructor:

```
public class ResourceUser: IDisposable
	{
		public void Dispose()
			{
			hwnd.Release();			//elibereaza o fereastra in Win32
			GC.SuppressFinalization(this);	//elimina apel de Finalize()
			}
	~ResourceUser()
		{
			hwnd.Release();
		}
	}
```

Pentru anumite clase C# se pune la dispozitie o metoda numite Close() in locul uneia Dispose() : fisiere, socket-uri, ferestre de dialog, etc. Este indicat ca sa se adauge o metoda Close() care sa faca doar apel de Dispose():

```
	//in interiorul unei clase
		public void Close()
			{
				Dispose();
			}
```

Modalitatea cea mai indicata este folosirea unui bloc using, caz in care se va elibera obiectul alocat (via metoda Dispose()) la sfarsitul blocului:

```
	using( obiect )
		{
			//cod
		}	//aici se va apela automat metoda Dispose()
```

Bibliografie:

1. http://docs.oracle.com/javase/tutorial/
2. http://en.wikipedia.org/wiki/.NET_Framework
3. Sommerville – Software engineering , 8th edition
4. Introducere in programarea orientata-obiect - Mircea Cezar Preda
5. https://developers.google.com/edu/c++/

*exemplele de cod sunt preluate din cartea ‘ Introducere in programare orientata obiect’ – Mircea Cezar Preda.