Capitolul 5

Constructori şi destructori

Crearea şi distrugerea obiectelor instanță ale claselor reprezintă o operație foarte importantă pentru asigurarea unor programe sigure şi stabile. Programatorul care proiectează o ierarhie de clase trebuie să ia în considerare toate situațiile posibile în care se pot crea şi distruge obiectele pentru a putea pune la dispozitia programatorilor ce folosesc clasele respective, un cod simplu şi sigur de utilizat.

Constructorii şi ***destructorii*** sunt funcții membru speciale ale claselor care sunt responsabile de executarea unor acțiuni cum sunt crearea obiectelor unei anumite clase, inițializarea datelor membre, copierea obiectelor şi distrugerea lor. Datorită particularității lor, ei prezintă atât caracteristici proprii funcțiilor membre obişnuite ale claselor, dar şi proprietăți specifice.

O primă caracteristică specifică acestor funcții o constituie numele lor. În toate variantele limbajului C++ s-a păstrat soluția aleasă de `Stroustrup` de a numi constructorii prin chiar numele clasei din care fac parte, iar destructorii prin numele clasei precedat de caracterul '~'. Această soluție este naturală, deoarece constructorii şi destructorii se apelează în mod automat de către compilator şi numele lor trebuie să fie predeterminat.

O altă particularitate o constituie faptul că dacă o anumită clasă nu conține în declarația ei constructori şi/sau destructori, aceştia sunt generați în mod automat de către compilator.

Însă ceea ce îi face deosebiți de celelalte funcții membru şi în general de toate celelalte funcții este faptul că nu returnează valori, nici măcar de tipul `void`. Acestea au un efect special, deoarece în caz contrar compilatorul ar trebui să ştie ce se va face cu valoarea returnată.

5.1 Constructori

Există două situații în care se apelează constructorii unei clase: când se creează un obiect al clasei respective şi eventual unele dintre datele membru ale obiectului creat sunt inițializate cu anumite valori, sau când se crează un obiect care se inițializează cu valorile datelor membru ale unui alt obiect existent din aceeaşi clasă.

Observație. Ultima situație nu trebuie confundată cu o instrucțiune de atribuire, deoarece ca şi în limbajul C, inițializarea unui obiect se face la definirea acestuia:

```
// definirea variabilelor i,j si initializarea lui j
int i, j = 3;
// operator de atribuire
i = 4;
```

De fapt noțiunile de definire şi inițializare a unui obiect sunt legate una de cealaltă şi ele nu pot exista separat.

Exemplul 5.1. Clasa *timp* permite determinarea unui interval de timp scurs de la o dată inițială (de forma: an, luna, zi, ora, minut, secundă) la data curentă, considrând timpul măsurat în secunde.

```
class timp {
  int ora, minut, secunda;
  double t;
  static int ora_0, minut_0, secunda_0;
  void SetTimp() {
    t = 3600*(ora - ora_0) +
        60*(minut - minut_0) +
        secunda - secunda_0;
  }
public:
  timp (int Ora = 0, int Minut = 0,
              int Secunda = 0)
  {
    ora = Ora;
    minut = Minut;
    secunda = Secunda;
  }
  double GetTimp() {
    SetTimp();
    return t;
  }
};

int timp::ora_0 = 0;
int timp::minut_0 = 0;
int timp::secunda_0 = 0;

void Problema() {
  timp m1(7, 3, 24);
  timp m2(20, 4, 12);
  cout << ”t1= “ << m1.GetTimp() << endl;
  cout << ”t2= “ << m2.GetTimp() << endl;
}
// ...
```

Operația de creare a unui obiect presupune două etape distincte: alocarea de către compilator a unui bloc de memorie neinițializat de o mărime bine definită şi apelarea apoi a unui constructor corespunzător al clasei obiectului respectiv. Operația de alocare este transparentă programatorului. Zona de memorie unde se alocă memorie depinde de modul de creare al obiectului (în zona de date a programului, în zona stivă, sau în zona `heap`).

Rolul apelului constructorului este acela de a iniţializa anumite date membru ale obiectului. Pentru aceasta, adresa de memorie a blocului alocat obiectului este transmis constructorului prin intermediul parametrului ascuns `this`.

În exemplul precedent există două apeluri implicite la constructorul timp al clasei *timp*, ceea ce înseamnă că de fapt compilatorul a inserat în locul celor două definiţii urmatoarele apeluri:

```
m1.timp::timp(7, 3, 24);
m2.timp::timp(20, 4, 12);
```

În cazul în care un obiect este definit ca o variabilă globală în afara oricărei funcţii, constructorul unui asemenea obiect este apelat înainte de execuţia funcţiei `main`.

Revenind la prima etapă a procesului de creare a unui obiect (alocarea memoriei pentru acesta), rezultă trei tipuri de alocare:

a) în zona de alocare statică, pentru obiectele externe definite în afara oricărei funcţii a unui program; în acest caz constructorul se apelează inaintea funcţiei `main`, iar durata de viaţă a obiectului corespunde timpului de execuţie al programului;
b) în zona stivă a programului, în cazul obiectelor locale definite în cadrul unor blocuri; alocarea şi apelul constructorului se efectuează când execuţia programului ajunge în locul definiţiei obiectului respectiv; durata de viaţă a unui astfel de obiect se reduce la timpul în care blocul a fost definit obiectul este activ;
c) în zona `heap` a programului, în cazul obiectelor create dinamic cu ajutorul operatorului `new`; durata de viaţă a unui astfel de obiect corespunde timpului între apelul consecutiv al operatorilor pereche `new` şi `delete`, aferenţi aceluiaşi pointer.

În primele două cazuri destructorii unui obiect sunt apelaţi în mod automat de către compilator, pe când în ultimul caz destructorul este apelat implicit prin intermediul operatorului `delete`.

Tot în cazul al treilea, apelul constructorului unui obiect creat dinamic este implicit prin intermediul operatorului `new`. De exemplu, crearea unui obiect cu alocare dinamică din clasa *timp* se poate realiza astfel:

```
// declararea unui pointer; nu se creeaza nici un obiect
timp* pt;
// creare obiect si apelul implicit al constructorului
timp
pt = new timp (7, 3, 1);
```

sau:

```
// declarare pointer si creare obiect
timp* pt = new timp (7, 3, 1);
```

Toate avantajele operatorului `new` faţă de funcţiile clasice de alocare a memoriei din limbajul C specificate în capitolul 2, sunt valabile şi în cazul obiectelor cu alocare dinamică.

5.2 Tipuri de constructori

În afară de cele prezentate anterior, mai există o situație în care se crează obiecte în mod implicit de către compilator: în cazul apelului unei funcții, când se efectuează transferul parametrilor prin valoare, precum şi atunci când se revine în funcția apelantă cu ajutorul instrucțiunii `return`.

Asa cum s-a precizat anterior, transferul parametrilor se relizează în limbajul C++ în două moduri distincte: prin valoare şi prin referință. Transmiterea prin referință nu crează obiecte suplimentare, însă transmiterea unui parametru prin valoare înseamnă de fapt transmiterea unei copii a parametrului actual spre parametrul formal. Cu alte cuvinte, se creează un nou obiect (temporar) care este o copie a obiectului ce reprezintă parametrul actual, iar acest obiect reprezintă valoarea de inițializare a parametrului formal corespunzător.

Operația inversă, cea de revenire într-o funcție apelantă, impune de asemenea crearea unui obiect ce reprezintă valoarea returnată de funcție.

În cazurile descrise mai sus (dar nu numai în acestea), se utilizează un alt tip de constructor numit ***constructor de copiere***. Acesta se apelează ori de câte ori crearea unui obiect impune inițializarea lui cu un alt obiect din aceeaşi clasă.

În general, constructorii se pot impărți în urmatoarele categorii:

- constructori generali,
- constructori impliciți,
- constructori de copiere,
- constructori de conversie de tip.

În mod uzual o clasă este prevazută cu mai mulți constructori diferiți, ce permit crearea obiectelor în diferite cazuri tipice.

Deoarece constructorii şi descructorii sunt funcții membre ale unei clase, sintaxa limbajului C++ permite ca ei să poată fi declarați atât membri publici, cât şi privați. În mod uzual, aceste funcții speciale trebuie să fie declarate publice, pentru că în caz contrar dacă o clasă are constructori şi/sau descructori privați, accesul la aceste funcții este interzis şi deci nu se pot crea/distruge obiectele instanță ale clasei.

Există totuşi anumite cazuri în care existența unor constructori privați este justificată: când se doreşte ca anumiți constructori ce pot fi generați automat de către compilator să nu se genereze.

A. Constructori generali

Constructorii generali sunt constructori care au cel puțin un argument care nu este o referință la tipul clasei respective, valorile argumentelor fiind utilizate pentru inițializarea datelor de tip membru ale obiectului creat. Marea majoritate a constructorilor din exemplele precedente au fost constructori generali.

Notând cu *X* numele clasei curente şi cu *T1, T2,…,* tipurile de date ale argumentelor, un constructor general are declarația de forma:

```
X(T1, T2, ...);
```

Deoarece constructorii sunt funcţii ale limbajului C++, toate elementele specifice funcţiilor sunt valabile şi în cazul constructorilor. De exemplu, este permis ca aceştia să aibă parametri cu valori implicite (lucru destul de des folosit la constructori, care măreşte eficienţa proiectării şi implementării claselor).

Constructorul clasei *timp* este un exemplu de constructor general cu parametrii impliciți. Urmatoarea definiție creează trei obiecte de tipul *timp*:

```
timp o1 (7, 3, 2);  // (7, 3, 2)
timp o2 (7, 3);     // (7, 3, 0)
timp o3 ();         // (0, 0, 0)
```

Observație. În cazul în care un constructor general are valori implicite pentru argumente, acestea trebuie specificate în declarația clasei şi nu în implementare.

B. Constructori impliciți

Constructorii impliciți nu au argumente, fiind, în cazul unei clase cu numele *X*, de forma următoare:

```
X (void);
```

Aceştia, ca şi constructorii de copiere, sunt *singurii constructori* ce pot fi generați automat de compilator în cazul în care o clasă nu are constructori.

Exemplul 5.2.

```
class timp {
  // ...
public:
  timp();
  // ...
};

timp::timp() {
  cout<<"Introduceti valori pentru ora, minut,\
     secunda: ";
  cin >> ora >> min >> sec;
}

void prelucrare() {
  // ...
  timp t;
  // ...
}
```

Observații:
1. Constructorii unei clase pot fi supraîncărcați.
2. Un constructor general cu toate argumentele implicite nu este un constructor implicit.

Datorită faptului că un constructor implicit poate fi generat de către compilator pentru o clasă care nu are constructori, poate apare o situație bizară: în cazul în care o clasă nu are definit decât un singur constructor şi acela este privat, obiectele instanță ale acelei clase nu mai pot fi create, deoarece constructorul privat nu poate fi accesat, iar un constructor implicit nu mai poate fi generat de către compilator.

Deoarece constructorii impliciți şi cei cu parametri impliciți se apelează folosind aceeaşi sintaxă, nu trebuie definiți în cadrul aceleiaşi clase împreună.

De exemplu, secvența următoare constituie o eroare de definire:

```
class timp {
   // ...
public:
  timp (int h = 0, int m = 0, int s = 0);
  timp();
  // ...
};
```

deoarece definiția urmatoare este ambiguă:

```
timp t;
```

Se impune o atenție sporită în cazul claselor care nu posedă nici un constructor, deoarece constructorul generat implicit de compilator nu efectuează nici o inițilizare a membrilor.

De exemplu, secvența următoare reprezintă o eroare, deoarece data membru *s* nu este inițializată la crearea obiectelor clasei *String*.

```
#define MaxString 100
class String {
  char s[MaxString + 1];
public:
  void set(const char str[]);
  const char* get() { return s; }
};
// ...
void main() {
  string s1;   //s nu este initializat
  // eroare de acces a memoriei !!
  cout << s1.get() << endl;
  // ...
}
```

O variantă de corectare a erorii precedente constă în scrierea unui constructor implicit ce creează un şir vid:

```
#define MaxString 100
class String {
  char s[MaxString + 1];
public:
  String() { s[0] = '\0'; }
```

```
  void set(const char str[]);
  const char* get();
};
```

O altă utilizare frecventă a constructorilor impliciți se referă la *inițializarea tablourilor de obiecte*. În cazul în care un program conține o definiție a unui tablou de obiecte instațe ale unei anumite clase, dacă tabloul nu este inițializat în mod explicit, pentru fiecare componentă a tabloului, compilatorul apelează automat constructorul implicit al clasei respective.

Exemplul 5.3. Programul următor:

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

unsigned int n = 0;

class A {
public:
  A() {
     cout << ”Constructor pentru obiectul A” << \
     ++n << endl;
  }
};

A v[7];

void main() {}
```

generează următoarea ieşire:

```
Constructor pentru obiectul A1
Constructor pentru obiectul A2
Constructor pentru obiectul A3
Constructor pentru obiectul A4
Constructor pentru obiectul A5
Constructor pentru obiectul A6
Constructor pentru obiectul A7
```

C. Constructori de copiere

Constructorii de copiere reprezintă o clasă importantă de constructori. În cazul în care în declarația unei clase nu este specificat nici un constructor de copiere, compilatorul va genera automat un asemenea constructor.

Rolul unui asemenea constructor poate fi judecat prin analogie cu inițializarea unei variabile de definire. De exemplu, definiția:

```
int k = 3;
```

presupune efectuarea a două acțiuni distincte: alocarea unei zone de memorie pentru variabila *k*, precum şi copierea constantei 3 în zona de memorie respectivă.

În mod asemănător, în cazul definirii unui obiect al unei clase se pot iniţializa datele membru ale obiectului cu valorile datelor membru ale unui alt obiect de acelaşi tip. De exemplu, sunt permise urmatoarele definiţii:

```
timp t(1, 0, 0);    // constructor general
timp t1 = t;        // constructor de copiere
timp t2(t1);        // constructor de copiere
```

Observaţie. Ultimele două definiţii reprezintă ambele forme de apel ale constructorului de copiere.

În acest caz iniţializarea obiectului *t1* se face prin copierea valorilor obiectului *t* prin intermediul unui constructor de copiere:

```
class timp {
  // ...
public:
  timp(const timp& t) {
    ora = t.ora;
    min = t.min;
    sec = t.sec;
  }
  // ...
};
```

Observaţii:

1. Întotdeauna primul argument al unui constructor de copiere trebuie să fie o *referinţă* spre un obiect al clase curente, sau o *referinţă* spre un obiect constant al clasei curente.
2. În cazul în care un constructor de copiere posedă suplimentar şi alţi parametri, aceştia trebuie să aibe toţi valori implicite; în caz contrar este vorba despre un constructor general. Aceasta restricţie se datorează modului de apel al unui constructor de copiere într-o definiţie de forma:
   ⟨clasa⟩ ⟨obiect1⟩ = ⟨obiect2⟩ ;

Exemplul 5.4. În secvenţa următoare, la iniţializarea obiectului *x2* cu valorile obiectului *x1* nu se pot specifica şi alti parametrii de iniţializare.

```
class X {
  // ...
  int a;
public:
  X(){ a = 0; }
  X(X& x, int k = 0) {
    a = x.a;
    // ...
  }
  // ...
};

// ...
```

```
X x1;
X x2 = x1;
X x3(x2, 5);
// ...
```

În mod uzual, un constructor de copiere generat de un compilator va efectua o copie membru cu membru a obiectului parametru. În cazul clasei *timp* anterioare, constructorul de copiere generat implicit este identic cu cel definit explicit.

Există cazuri însă în care un constructor de copiere implicit generat de compilator nu este suficient pentru o corectă inițializare a obiectului curent, în special în cazurile în care există date membru de tip pointer, sau anumite date membre sunt obiecte ale altor clase.

Exemplul 5.5. Clasa *list* implementează o listă liniară simplu legată, iar clasa *node* structura valorilor unei asemenea liste.

```
struct node {
  int val;
  node* next;
  node() {val = 0; next = 0;}
  node(int v, node* n = 0) {
    val = v;
    next = n;
  }
  // constructor de copiere generat implicit
  ~node(){ next = 0; }
  void Add (int);   // adauga un nod dupa nodul curent
  void Print() const { cout << val << endl; }
  // ...
};

struct list {
  node* first;
  void Copy(list& l);
  void Delete();
  list() { first = 0; }
  list(list&);
  ~list();
  list& operator=(list&);
  node* Last() const;
  // adauga un element la inceputul listei
  void AddFirst(int);
  // adauga un element la sfarsitul listei
  void AddLast(int);
  // adauga un element dupa un element specificat
  void Add(node*, int);
  void Print() const;
  // ...
};

void node::Add(int k) {
```

```
  node* p = new node(k);
  next = p;
};

void list::Copy (list& l) {
  node* p = new node(l.first->val);
  first = p;
  for (node*q=p->next; p; p=p->next)
    Last()->Add(q->val);
}

void list::Add(node* n, int k) {
  if (n) {
    node *p = new node(k);
    p->next = n->next;
    n->next = p;
  }
}

void list::AddFirst(int k) {
  node *p = new node(k);
  p->next = first;
  first = p;
}

node* list::Last() const {
  node* p;
  for(p=first; p->next; p=p->next);
    return p;
}

void list::AddLast(int k) {
  if (first)
    Last()->Add(k);
  else {
    node *p = new node(k);
    first = p;
  }
}

void list::Print() const {
  for (node* p=first; p; p=p->next)
    p->Print();
}

void list::Delete() {
  // implementare ulterioara (la destructori)
}

list::list(list& l) {
  Copy (l);
```

```
}

list::~list() {
  Delete();
  first = 0;
}

list& list::operator=(list& l) {
  if(&l != this) {
    Delete();
    Copy(l);
  }
  return *this;
}

// ...
```

Observație. Un constructor de copiere al unei clase *X* trebuie să aibă drept parametru o referință la clasa *X* (de tip `X&`) sau una la clasa însăşi (un parametru de tip `X`).

Un constructor de copiere nu se apelează doar la inițializarea obiectelor cu valorile altor obiecte, ci şi în cazul mecanismului de transfer al parametrilor la apelul funcțiilor.

În cazul transmiterii prin valoare, se creează o copie temporară a obiectului ce este parametru actual care se transmite prin copiere obiectului parametru formal corespunzător. În momentul revenirii în funcția apelată cu ajutorul unei instrucțiuni `return`, valoarea ce reprezintă obiectul rezultat se transmite tot prin copiere funcției apelante, deci se creează încă un obiect prin copiere.

Exemplul 5.6. O funcție ce crează o listă formată din primul şi ultimul element al unei alte liste.

```
list FirstLast (list l) {
  list l1;
  l1.Add(l.first->val);
  l1.Add(l.Last()->val);
  return l1;
}
void Prelucrare() {
  list l1, l2;
  l1.Add(3);
  l1.Add(7);
  l2 = FirstLast(l2);
  // ...
}
```

În momentul apelului funcției *FirstLast*, se apelează constructorul de copiere pentru parametrul *l*, care are ca parametru o referință la obiectul *l2*. Acest obiect temporar se va distruge după ieşirea din funcția *FirstLast*.

Instrucţiunea `return` are următorul efect: crearea în mod automat a unui obiect suplimentar de tip *list* prin copierea obiectului *l1*. Acest obiect nou creat reprezintă obiectul care se returnează spre funcția *Prelucrare* şi care este preluat de operatorul de atribuire.

Există o situație care face justificată utilizarea unui constructor privat: în cazul în care se doreşte ca să nu se permită la apelul funcțiilor transferul prin valoare a obiectelor instanță ale unei clase, se poate defini un constructor de copiere privat. În acest caz compilatorul nu mai generează un constructor de copiere, iar orice tentativă de transfer prin valoare a unui obiect instanță dintr-o asemenea clsă va genera un mesaj de eroare.

Exemplul 5.7. Nu se permite apelul unui constructor de copiere în clasa *C*. Din acest motiv, acest constructor a fost doar declarat (nu mai este nevoie de definiția lui efectivă, deoarece nu se ajunge aici).

```
class C {
     int n;
     C(C& c);
public:
     C(int a=0):n(a){}
     void Print() {cout << n << endl;}
};

void f(C c) { }

void main() {
     C o;      // O.K. Constructor de conversie
     C o1 = o; // Eroare! Constructor de copiere
     f(o);     // Eroare! Constructor de copiere
}
```

D. Constructori de conversie de tip

Un ***constructor de conversie de tip*** este în mod uzual un constructor cu un singur argument (ca şi constructorii de copiere), dar tipul acestuia este diferit de tipul clasei curente. În cazul în care există mai mulți parametri, aceştia trebuie să fie toți cu valori implicite pentru a putea fi vorba de un constructor de conversie de tip.

De exemplu, al doilea constructor al clasei *node* din exemplul precedent este un constructor de conversie de tip. Rezultă astfel că un constructor general este considerat fie un constructor cu toți parametrii impliciți, fie un constructor care are cel puțin doi parametri ce nu sunt impliciți.

Constructorii de conversie sunt des utilizați de către compilator pentru realizarea ***conversiei implicite a tipurilor***. În mod uzual, ori de câte ori un operand dintr-o expresie nu respectă tipul de date al expresiei respective, se încearcă o conversie automată a acestuia la tipul expresiei. De exemplu, pentru tipurile predefinite, în expresia următoare se realizează o conversie de la `int` la `double`:

```
int n = 3;
double x, y = 2.5;
x = y + n;
```

O conversie asemănătoare se realizează şi în cazul obiectelor, compilatorul încercând să găsească un constructor de conversie corespunzător.

Exemplul 5.8. Pentru clasa *String* anterioară se defineşte un constructor de conversie:

```
#include <string>
#include <iostream>
using namespace std;
#define MaxString 100

class String {
  char s[MaxString + 1];
public:
  String() { s[0] = '\0'; }
  String(const char str[]) { strcpy(s, str); }
  void set(const char str[]);
  const char* get() { return s; }
};
// ...

void f(String s) { cout << s.get() << endl; }

void main() {
  String s1;
  f(s1);
  f("abc");
  // ...
}
```

La al doilea apel al funcției *f*, se utilizează constructorul de conversie pentru a converti şirul de caractere `"abc"` într-un obiect de tip *String* ce va fi transmis funcției ca parametru. La primul apel se apelează constructorul de copiere.

5.4 Destructori

Destructorii se utilizează pentru a elibera zonele de memorie ocupate de membrii unui anumit obiect, înainte de dealocarea memoriei pentru obiectul respectiv. Ca şi în cazul constructorilor, dealocarea memoriei pentru un obiect nu reprezintă o acțiune proprie destructorului.

De aici rezultă că un destructor se utilizează uzual în cazul în care obiectele unei clase folosesc alocarea dinamică pentru anumite date membre ale lor.

În cazul în care o clasă nu conține o definiție explicită a unui destructor, compilatorul va genera implicit un destructor pentru aceasta.

Spre deosebire de constructori, destructorii nu pot avea argumente. De asemenea, destructorii nu pot fi supraîncărcați, fiecare clasă trebuind să conțină un singur destructor.

În cazul în care un obiect nu are alocată memorie în zona `heap` (nu s-a apelat operatorul `new`), apelul constructorului pentru respectivul obiect se face în mod automat de către compilator: pentru obiectele locale definite în cadul blocurilor apelul se face la ieşirea din blocul curent în care au fost definite, iar pentru obiectele globale definite în afara oricărei funcţii, destructorii sunt apelaţi după ieşirea din funcţia `main`, sau când se apelează explicit funcţia `exit`.

Exemplul 5.9.

```
#include <iostream>
using namespace std;
class X {
  int k;
public:
  X(int i) {
    k = i;
    cout << ”x() pentru ” << k << endl;
  }
  ~X() { cout << ”~x() pentru “ << k << endl; }
};

X ob1(5);

void f() {
  cout << ”incepe funcţia f” << endl;
  static X ob2(7);
  X ob3(9);
  cout << ”se termina functia f” << endl;
}

void main() {
  cout << ”incepe functia main” << endl;
  X ob4(11);
  f();
  cout << ”se termina functia main” << endl;
}
```

Execuţia programului generează următoarea ieşire:

```
x() pentru 5
incepe functia main
x() pentru 11
incepe functia f
x() pentru 7
x() pentru 9
se termina functia f
~x() pentru 9
se termina functia main
~x() pentru 11
```

```
~x() pentru 7
~x() pentru 5
```

Din exemplul anterior se observă faptul că, în cazul în care sunt mai multe obiecte care trebuie distruse, destructorii sunt apelați *în ordinea inversă* a apelurilor pentru constructori.

În cazul constructorilor pentru un obiect static local se apelează la prima întâlnire a definiției obiectului, dar destructorul se apelează după ieşirea din funcția `main` (ceea ce corespunde regulii duratei de viață pentru obiectele statice).

În exemplul următor se observă apelul constructorilor şi destructorilor în cazul transmiterii obiectelor prin valoare ca argumente în apelul funcțiilor.

Exemplul 5.10. O clasă care-şi numără obiectele instanță.

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

class Contor {
  char c;
  static int contor;
public:
  void Print() {
     cout << "obiect " << c << " contor " \
     << contor << endl;
  }
  Contor(const char& ch) {
    c = ch;
    ++contor;
    cout << "Constructor conversie: ";
    Print();
  }
  Contor(const Contor& h) {
    c = h.c;
    ++contor;
    cout << "Constructor copiere: ";
    Print();
  }
  ~Contor() {
    --contor;
    cout << "Destructor: ";
    Print();
  }
};

int Contor::contor = 0;

Contor f(Contor x) {
  cout << "Incepe functia f" << endl;
  cout << "Se termina functia f" << endl;
  return x;
```

```
 }

void main() {
  Contor o1('a');
  cout << "Inainte de apelul lui f cu valoare \
     de intoarcere" << endl;
  Contor o2 = f(o1);
  cout << "Dupa apelul lui f"<<endl;
  cout <<"Inainte de apelul lui f fara valoare \
          de intoarcere" << endl;
  f(o1);
  cout << "Dupa apelul lui f fara valoare \
          de intoarcere" << endl;
}
```

Ieşirea programului:

```
Constructor conversie : obiect a contor 1
Inainte de apelul lui f cu valoare de intoarcere
Constructor copiere: obiect a contor 2
Incepe functia f
Se termina functia f
Constructor copiere: obiect a contor 3
Destructor: obiect a contor 2
Dupa apelul lui f cu valoare de intoarcere
Inainte de apelul lui f  fara valoare de intoarcere
Constructor copiere : obiect a contor 3
Incepe functia f
Se termina functia f
Constructor copiere : obiect a contor 4
Destructor : obiect a contor 3
Destructor : obiect a contor 2
Dupa apelul lui f fara valoare de intoarcere
Destructor : obiect a contor 1
Destructor : obiect a contor 0
```

Se observă faptul că iniţializarea argumentului funcţiei se realizează prin intermediul constructorului de copiere. Parametrul *x* devine astfel un obiect temporar local funcţiei şi va fi distrus imediat după terminarea execuţiei funcţiei şi revenirea în funcţia `main`.

Evaluarea expresiei din instrucţiunea `return` generează un al doilea obiect temporar (valoarea ce trebuie returnată) care este creat tot cu ajutorul constructorului de copiere. În cazul în care funcţia apelantă utilizează valoarea returnată, acest obiect nu este distrus, el reprezentând de fapt obiectul *o2* din funcţia main. În cazul în care funcţia apelantă nu utilizează valoarea returnată, acest obiect este distrus după apelul funcţiei şi înainte de revenire în funcţia `main` (la al doilea apel există doi destructori apelaţi succesiv, unul pentru obiectul temporar şi altul pentru valoarea returnată).

În cazul folosirii pointerilor, constructorii şi destructorii trebuie apelaţi explicit prin intermediul operatorilor `new` şi `delete`.

Observații.

1. Chiar dacă un pointer a ieşit din domeniul său de definiție, dacă nu se apelează operatorul `delete`, obiectul asociat pointerului nu se distruge (destructorul nu este apelat implicit).
2. În cazul în care la sfârşitul execuției programului au ramas obiecte alocate în zona `heap`, compilatorul forțează apelul destructorilor pentru acestea după ieşirea din funcția `main`.

Exemplul 5.11: Destructorul pentru clasa *list* din exemplul 5.5 trebuie să distrugă toate obiectele de tip *node* conținute în lista curentă:

```
struct list {
  node* first;
  void Copy (list& l);
  void Delete();
  list() { first = 0; }
  list(list&);
  ~list();
  // ...
};

void list::Delete() {
  for(node* p=first; p ;) {
    node*q = p->next;
    delete p;
    p = q;
  }
}

list::~list() {
  Delete();
  first = 0;
}

void Prelucrare() {
  list* l1 = new list;
  l1->Add(3);
  l1->Add(7);
  l1->Print() ;
  // ...
  delete l1;
  // ...
}
```