7. STUDIUL SISTEMATIC AL FUNCȚIILOR DE TIPURILE CONSTRUCTOR ȘI DESTRUCTOR

În acest capitol se vor aborda următoarele topici: restricții ale funcțiilor de tipurile constructor și destructor, inițializarea obiectelor, durata de viață a obiectelor, tipuri (categorii) de obiecte, tipuri de constructori, aplicații ale constructorilor de inițializare de forma C(C &).

7.1. Restricții ale funcțiilor de tip constructor

- Funcțiile de tip constructor (constructorii) nu pot întoarce nici o valoare. De aceea în corpul funcției nu trebuie să apară instrucțiunea **return**;
- Constructorii nu pot fi de tipurile **friend** sau **virtual**;
- Lista de argumente de intrare este opțională; în lipsa ei se folosește (**void**);
- Obiectele create sub forma unor masive de obiecte trebuie să aibă un constructor fără listă de argumente. Un astfel de caz este prezentat în listingul programului P7_1.CPP de mai jos, în care clasa MASIV conține un constructor cu listă vidă. Definirea masivului de obiecte se face în funcția main().

```
// Program P7_1.CPP
// Exemplu de inițializare a unui masiv pentru un compilator
// care nu permite inițializarea masivelor de tipul AUTO
# include <iostream.h>
struct S {int i, j, k; };

// Nu se poate realiza inițializarea în bloc a unei clase ce conține membri de tip private.
// Se poate însă declara un masiv de obiecte
class MOBIECTE {
                int i;
                char *nume;
                double d;
public:
        MOBIECTE() { i = 0; nume = " "; d = 3.14;}          // Constructorul cu lista vidă
        MOBIECTE (int ii, char *denum, double dd)           // Constructorul cu lista nevidă
        { i = ii; nume = denum; d = dd; }
};
# define nr_obiecte    6                  // Se stabilește numărul de obiecte din masiv

// Programul apelant
void  main (void)
{
 S  a = {100, 200, 300};                  // Se creează obiectul "a" de tipul S
 struct {                                 // Se definește o structură fără tag, utilizată la
        int i;                            // crearea unui masiv de structuri
        char *nume;
        double d;
 } masiv_de_structuri [] = {
        1, "alfa", 1.00,
        2, "beta", 1.11,
        3, "gama", 1.22,
        4, "delta", 1.33,
```

```
        5, "epsilon", 1.44,
        6, "miu", 1.55
};
MOBIECTE masiv[nr_obiecte];      // Observație: Putem declara un masiv de obiecte
                                 // numai când el posedă un constructor cu lista vidă
for (int k = 0; k < nr_obiecte; k++) {
        cout << "MOBIECTE[" << k << "].i = " << masiv_de_structuri[k].i << endl;
        // Introducerea unei linii de genul următor
        // cout << "MOBIECTE[" << k << "].i = " << masiv[k].i << endl;
        // conduce la o eroare de tipul MOBIECTE :: i is not accessible
        cout << "MOBIECTE[" << k << "].nume = " << masiv_de_structuri[k].nume << endl;
        cout << "MOBIECTE[" << k << "].d = " << masiv_de_structuri[k].d << endl;
}
cout << "Elementele obiectului \"a\" de tip S sunt: " << a.i << "," << a.j << "," << a.k << endl;
}
```

- Într-o structură de tipurile **class** și **union** pot exista mai mulți constructori;
- Nici când se lucrează cu clase derivate, funcțiile constructor nu pot fi declarate ca funcții virtuale;
- Când se creează obiecte de tipurile **automatic**, **static** sau **dinamic**, constructorii au anumite proprietăți ce vor fi prezentate în acest capitol.

7.2. Restricții ale funcțiilor de tip destructor

- Ca și constructorii, destructorii nu trebuie să întoarcă nici o valoare;
- Destructorii nu trebuie să posede listă de argumente;
- Clasele și uniunile pot avea destructori, nu și structurile de date de tip **struct**; membrii unei clase pot avea destructori, nu și membrii unei uniuni;
- Destructorii pot fi virtuali;
- Destructorii nu pot fi apelați în mod explicit. Ei sunt automat apelați la terminarea acelui bloc (fișier, program) în care au fost create obiectele care acum trebuie să "dispară" (în sensul eliberării spațiului de memorie);
- Destructorii nu pot fi redefiniți.

7.3. Durata de viața a obiectelor

Se știe că în limbajul C există patru specificatori ai claselor de memorare: **auto**, **extern**, **static** și **register**. Similar, în C++, există:

- *Obiecte automate*. Acestea sunt locale pentru funcția respectivă. Rezultă că, durata de viață a acestor obiecte este durata de viață a funcției. Dacă o funcție conține blocuri, obiectele create în interiorul acestor blocuri, vor "viețui" numai în interiorul acestor blocuri. Locul de păstrare a obiectelor automate este stiva.
- *Obiecte statice*. Acestea sunt create în afara funcției, la momentul transferului controlului către program.
- *Obiecte dinamice*. Diferă de cele automate, nu ca durată de viață, ci după locul unde sunt memorate. Memoria folosită drept gazdă a acestor obiecte este cea de tipul "*heap*". Operatorii **new** și **delete** rezervă și respectiv eliberează spațiul de memorie corespunzător acestor obiecte.
- *Obiecte de tip membru ale unei clase*. Sunt obiecte create și distruse când este creată, respectiv distrusă o instanțiere a clasei căreia îi aparțin.
- *Obiecte ce aparțin clasei derivate*. Pentru crearea acestor obiecte se va activa în prealabil constructorul clasei de bază, apoi constructorul clasei derivate va crea obiectele respective.

- *Obiecte fără nume*. Au o existență efemeră, fiind necesare pe durata calculelor unor expresii (vezi programul P6_4.CPP ce utilizează clasa PUNCT din Cap. 6)

În următorul exemplu sunt create și distruse trei obiecte: unul **dinamic**, unul **automatic** și unul **static**.

```
// Program P7_2.CPP   Obiecte cu durata de viață limitată
# include <iostream.h>
# include <string.h>
class PREOPINENT {
        char *cine;
        double d;
public:
        PREOPINENT (char *s){                          // Constructorul clasei
                cine = strdup (s);
                cout << "A fost creat " << cine << endl;
        }
        ~PREOPINENT (void) {                           // Destructorul clasei
                cout << "Ce e val ca valul trece ! Deci si " << cine << endl;
                delete cine;
        }
};
PREOPINENT *Efemer (void)              // Crearea dinamică a obiectului "Sistemul solar"
{
        PREOPINENT *ss = new PREOPINENT ("Sistemul solar");
        return ss;
}
void Geneza (char *cine)
{
        PREOPINENT ss (cine);
        {
                cine = strdup ("Omul");
                PREOPINENT om (cine);     // "Omul" este un obiect de tip automatic
        }
}
PREOPINENT Creatorul ("Universul");       // Obiectul "Universul" este de tip static
// Programul apelant
void  main (void)
{
        PREOPINENT *ra;
        ra = Efemer();
        Geneza("Pamantul");               // "Pământul" este un obiect de tip automatic
        delete ra;
}
```

Observație. Funcția strdup() din <string.h> copiază un șir într-o zonă nou creată. Prototipul funcției este

```
        char *strdup (const char *s);
```

Deci, strdup() realizează o copie a șirului s, obținând spațiul de memorie printr-un apel al funcției malloc(). Spațiul alocat este de (strlen(s) + 1) octeți. Dacă funcția lucrează corect, aceasta va întoarce un pointer la prima locație unde începe copia șirului, iar în caz de eroare, întoarce NULL.

Rezultate obținute prin rularea acestui program sunt:

```
A fost creat Universul
A fost creat Sistemul solar
A fost creat Pamantul
A fost creat Omul
Ce e val ca valul trece ! Deci si Omul
Ce e val ca valul trece ! Deci si Pamantul
Ce e val ca valul trece ! Deci si Sistemul solar
Ce e val ca valul trece ! Deci si Universul
```

Obiectele s-au format în ordinea Universul -> Sistemul solar -> Pământul -> Omul, iar dispariția lor s-a produs în ordine inversă. Obiectul "Universul" fiind **static**, a avut durata cea mai lungă (egală cu a programului). Obiectul ss (Sistemul solar) a făcut parte din categoria obiectelor de tip **dinamic**, celelalte obiecte fiind de tip **automatic**.

7.4. Reguli privind ordinea creării obiectelor ce aparțin aceleiași categorii

R1. Obiectele din categoria **automatic** sunt create în ordinea în care apar în funcția respectivă. Distrugerea lor se face, evident, în ordine inversă, deoarece ele sunt alocate pe stivă.
R2. Obiectele dintr-o funcție declarate explicit drept **statice** nu vor fi create dacă această funcție nu este apelată. Constructorul poate să le creeze, după ce s-au creat în prealabil obiectele statice exterioare funcției.
R3. Obiectele **membre** ale unei clase sunt create înainte de a fi apelat constructorul clasei în care se află aceaste obiecte.

7.5. Tipuri de constructori

În acest paragraf se sistematizează câteva probleme despre constructori, probleme întâlnite în exemplele de până aici. Notăm cu C și K două clase de obiecte, cu K diferit de C. În exemplele de până acum am întâlnit constructori de forma:

```
C :: C ();                          // cu lista vidă
C :: C (lista_de_parametri);    // cu lista nevidă
C :: C (C &);   // atribuirea unui obiect de un anumit tip (C) unui obiect de același tip
C :: C (K &);   // convertirea și atribuirea unui obiect de tip (K) unuia de tip C
```

De obicei, construcția C :: C (C &) se numește *constructor inițializator*, deoarece acest constructor este chemat la momentul inițializării (vezi programul P6_5.CPP din Cap. 6).
Pentru a vedea mai clar modul de apel al constructorilor în cazul creării unor obiecte, prezentăm în continuare un exemplu simplu și edificator.

```
// Program P7_3a.CPP   Verificarea ordinii de apelare a constructorilor și destructorului
# include <iostream.h>
# include <string.h>
class FORTA {
        double masa;
        double acceleratia;
public:
        FORTA (double, double);                 // Primul constructor
        FORTA (FORTA &);                        // Al doilea constructor
        ~FORTA() { cout << "Trec prin destructor" << endl; }
};
```

```
FORTA :: FORTA (double mm, double aa)
{       masa = mm;
        acceleratia = aa;
        cout << "Trec prin constructorul FORTA (double, double)" << endl;
}
FORTA :: FORTA (FORTA &g)                       // Constructor inițializator de forma C(C &)
{       masa = g.masa;
        acceleratia = g.acceleratia;
        cout << "Trec prin constructorul FORTA (FORTA &)" << endl;
}
void idle (FORTA f)
{
        cout << "Sunt in functia idle()" << endl;
}
// Programul apelant
void  main (void)
{       FORTA f1(10., 9.8);
        FORTA f2 = f1;
        idle(f2);
}
```

În urma rulării, programul va arăta următoarea ordine de apel a constructorilor:

```
        Trec prin constructorul FORTA (double, double)
        Trec prin constructorul FORTA (FORTA &)
        Trec prin constructorul FORTA (FORTA &)
        Sunt in functia idle()
        Trec prin destructor
        Trec prin destructor
        Trec prin destructor
```

Vom modifica acum funcția idle(), în sensul că apelul obiectului "f" se va face prin referință și nu prin valoare, iar funcția va returna o copie a obiectului. Listingul anterior devine:

```
// Program P7_3b.CPP   Verificarea ordinii de apelare a constructorilor și destructorului
// Funcția idle() a fost modificată pentru a returna copia obiectului
# include <iostream.h>
# include <string.h>
class FORTA {
        double masa;
        double acceleratia;
public:
        FORTA (double, double);
        FORTA (FORTA &);
        ~FORTA() { cout << "Trec prin destructor" << endl; }
};
FORTA :: FORTA (double mm, double aa)
{       masa = mm;
        acceleratia = aa;
        cout << "Trec prin constructorul FORTA (double, double)" << endl;
}
```

```
FORTA :: FORTA(FORTA &g)                    // Constructor inițializator de forma C(C &)
{       masa = g.masa;
        acceleratia = g.acceleratia;
        cout << "Trec prin constructorul FORTA (FORTA &)" << endl;
}
FORTA idle (FORTA & f)                      // Definiția funcției idle() modificată
{
        cout << "Sunt in functia idle()" << endl;
        return f;
}
// Programul apelant
void  main (void)
{
        FORTA f1(10., 9.8);
        FORTA f2 = f1;
        idle (f2);
}
```

Ordinea mesajelor va fi acum:

```
        Trec prin constructorul FORTA (double, double)
        Trec prin constructorul FORTA (FORTA &)
        Sunt in functia idle()
        Trec prin constructorul FORTA (FORTA &)
        Trec prin destructor
        Trec prin destructor
        Trec prin destructor
```

Se observă inversarea rândurilor 3 cu 4 din ordinea afișată anterior. În primul caz, în funcția idle(), argumentul a fost pasat prin valoarea sa (FORTA f), ceea ce a echivalat cu activarea constructorului respectiv. În al doilea caz, argumentul funcției idle() a fost pasat prin referința acestuia (&f). Atunci, în funcția idle(), constructorul a fost chemat numai la momentul execuției instrucțiunii **return** f (întoarcerea valorii obiectului f), deci după execuția instrucțiunii cout << "Sunt in functia idle()" << endl;

7.6. O altă aplicație a constructorului inițializator de forma C (C &)

O utilizare mai complicată a constructorului inițializator de forma C (C &), așa numitul *copy intializer*, constă în pasarea unui obiect drept parametru de intrare al unei funcții și returnarea în exteriorul unei funcții a unui obiect rezultat. În ambele cazuri se folosește mecanismul apelului prin valoare. Compilatorul va apela la serviciile constructorului inițializator ori de câte ori se pasează un mesaj sub forma unui obiect. În acest moment, obiectul din corpul acelei funcții, considerat obiect local (automatic), va fi inițializat cu obiectul "importat" prin această pasare. La întoarcerea rezultatului unei funcții, procedura va fi exact inversă, și anume obiectul local (interior) va inițializa obiectul exterior. Deci, valoarea primului va fi "exportată" în cel de al doilea obiect. Programul din exemplul următor evidențiază modul de utilizare a unui constructor pe funcție de *copy initializer*. Mesajul, de fapt un obiect, este pasat între funcțiile main() și fvect().

```
// Program P7_4.CPP  Folosirea constructorului de tip copy initializer
// pentru transmiterea valorii într-o funcție și pentru returnarea valorii dintr-o funcție
# include <stdio.h>
# include <process.h>
```

```
class MASIV {                           // Clasa MASIV definește un vector de numere reale
        int n_elem;                     // n_elem = numărul de elemente ale vectorului
        double *p_masiv;                // p_masiv = pointer la elemente de tip double
public:
        MASIV (int marime = 0, double val = 0.);   // Primul constructor
        MASIV (MASIV & sursa);                     // Al doilea constructor de tip copy initializer
        ~MASIV() { delete p_masiv; }               // Destructorul
        MASIV operator = (MASIV & sursa);          // Atribuire prin calea operator =
        void Imp (char *mesaj = " ");
        double & operator [] (int m);
        int dimens_masiv() {return n_elem; }
};

MASIV :: MASIV (int ii, double dd)              // Definiția primului constructor
{       p_masiv = new double[n_elem = ii];      // Se ocupă o zonă de memorie
                                                // de n_elem*sizeof(double) octeți
        for ( int k = 0; k < n_elem; k++ )
                p_masiv[k] = dd;                // Valoarea dd este copiată în fiecare element din
}                                               // vectorul p_masiv

MASIV :: MASIV (MASIV & sursa)          // Forma de tip copy initializer a constructorului
{       p_masiv = new double[n_elem = sursa.n_elem];    // Se ocupă o zonă de memorie
                                                        // de n_elem*sizeof(double) octeți
        for ( int k = 0; k < n_elem; k++ )
                p_masiv[k] = sursa.p_masiv[k];  // Valoarea fiecărui element al masivului "sursa"
                                                // este copiată în fiecare element al vectorului p_masiv
}

MASIV MASIV :: operator = (MASIV & sursa)   // Atribuire prin calea operator =
{       delete p_masiv;                 // Se eliberează vechea resursă de memorie,
                                        // apoi se realocă o zonă tot de n_elem elemente în care
                                        // urmează să se facă atribuirea cu funcția de tip operator =
        p_masiv = new double[n_elem = sursa.n_elem];    // Se ocupă o zonă de memorie
                                                        // de n_elem*sizeof(double) octeți
        for ( int k = 0; k < n_elem; k++ )
                p_masiv[k] = sursa.p_masiv[k];          // Dedublare valori în fiecare element
        return *this;
}

void MASIV :: Imp (char *mesaj) {               // Definiția funcției care asigură listarea
        printf ("%s", mesaj);                   // elementelor vectorului p_masiv
        for ( int k = 0; k < n_elem; k++ )
                printf ("%lf ", p_masiv[k]);            // Se listează elementele
        printf("\n");
}

double &MASIV :: operator [] (int m)
{       if (m < n_elem )
```

```
                return p_masiv[m];                    // Nu se depășește dimensiunea masivului
        else {
                puts ("Depasire de dimensiune !");
                exit (1);
        }
}
MASIV fvect (MASIV & valoare)  {          // Transmitere și întoarcere de parametri prin valoare
        if (valoare.dimens_masiv() == 1)
                valoare[0] = 0.;
        return valoare;
}
// Programul apelant
void  main (void)
{       MASIV m1(10, 0.5);
        MASIV m2;
        m1.Imp ("Primul masiv este = ");
        m2.Imp ("Al doilea masiv este = ");  // Nu trebuie să scrie nimic !
        printf ("\n");
        m2 = fvect (m1);                      // Se apelează constructorul de tip copy initializer
        m1.Imp ("Primul masiv, dupa folosirea lui fvect,  este = ");
        m2.Imp ("Al doilea masiv, dupa folosirea lui fvect,  este = ");
}
```

Clasa MASIV conține doi constructori. Al doilea constructor este de tip *copy initializer*. În funcția main() se definesc două obiecte notate cu m1 și m2. Primul obiect m1 are 10 elemente, fiecare având valoarea 0.5; al doilea obiect m2 are dimensiunea inițiala nulă. Dovada este și faptul că prin rularea acestui program, la momentul execuției liniei sursă

```
        m1.Imp ("Primul masiv este = ");
```

se obțin 10 valori egale cu 0.5, în timp ce linia sursă

```
        m2.Imp ("Al doilea masiv este = ");
```

nu va lista nimic.

În linia sursă m2 = fvect (m1); obiectul m1 va fi pasat prin valoare funcției fvect(), care la rându-i îl va returna funcției main(). Comilatorul va folosi acum cel de-al doilea constructor. Prin executarea liniei

```
        p_masiv = new double[n_elem = sursa.n_elem];
```

acesta va aloca o zonă de memorie (un obiect local de tip masiv) având aceeași dimensiune cu obiectul specificat "sursa ". Deci variabila sursa.n_elem conține valoarea 10 și aparține obiectului importat m1. În continuare bucla **for** va copia de fapt cele 10 valori egale cu 0.5 din elementele sursa.p_masiv[k] ale obiectului m1, în zona rezervată celor n_elem elemente ale obiectului m2. Rezultatele obținute prin rularea ultimelor două linii sursă, prin care se apelează funcția membră Imp(), sunt edificatoare. Atât obiectul m1 cât și obiectul m2 au acum câre 10 elemente cu valoarea 0.5.

7.7. Clase cu obiecte încuibate

În exemplul care urmează se consideră trei clase: o clasă de bază denumită PARTID, din care vom "deriva" a clasă denumită MEMBRU. În corpul clasei PARTID se va defini un obiect ce aparține unei a treia clase, "nederivate", denumită FACTIUNE. Acest exemplu evidențiază ordinea de apel a constructorilor și destructorilor pentru cazul când obiectele conțin, la rândul lor, alte obiecte (de exemplu: partidul conține facțiuni și, în același timp, membri).

```
// Program P7_5.CPP   Clasă cu obiecte încuibate de tipul clasa
// Ordine de apel a constructorilor în situația în care există
// o clasă de bază PARTID, una de tip FACTIUNE încuibată în cadrul clasei PARTID
// și o a treia clasă numită MEMBRU, derivată din clasa de bază PARTID
# include <iostream.h>
class FACTIUNE {                        // Va fi utilizată pentru crearea de obiecte încuibate
                                        // în clasele definite mai jos
        int  f;                         // Variabilă de tip private
public:
   FACTIUNE (int i) {                   // Constructorul clasei FACTIUNE
        cout << "Apelare constructor clasa FACTIUNE, cu i = " << i << endl;
        f = i;
   }
   ~FACTIUNE() {                        // Destructorul clasei FACTIUNE
        cout << "Apelare destructor clasa FACTIUNE, cu f = " << f << endl;
   }
};
class PARTID {                          // Clasa de bază, PARTID
        int  p;                         // Variabilă de tip private
        FACTIUNE  F;                    // Obiectul F este încuibat în clasa PARTID
public:
   PARTID (int a, int b) : F(b) {       // Constructorul clasei PARTID
        // Se observă modul de apelare al membrului încuibat
        cout << "Apelare constructor clasa PARTID, cu a si b = "
        << a << " respectiv " << b << endl;
      p = a;
   }
   ~PARTID() {                          // Destructorul clasei PARTID
        cout << "Apelare destructor clasa PARTID, cu p = " << p << endl;
   }
};
// O clasa derivată
class MEMBRU : public PARTID {          // Definiția clasei derivate MEMBRU
        int  m;                         // Variabilă de tip private
        FACTIUNE FF;                    // Obiectul FF este încuibat în clasa MEMBRU
public:
   MEMBRU (int a, int b, int c) : (a, b), FF(c) {      // Constructorul clasei PARTID
        cout << "Apelare constructor clasa MEMBRU, cu a, b si c = "
        << a << " , " << b << " respectiv " << c << endl;
      m = a;
   }
   ~MEMBRU() {                                  // Destructorul clasei MEMBRU
        cout << "Apelare destructor clasa MEMBRU, cu m = " << m << endl;
   }
};
void  main (void)
{
 cout << endl << "Creare obiect derivat MM (1, 2, 3)" << endl;
```

```
 MEMBRU MM (1, 2, 3);
 cout << endl << "Partidul se dizolva ! Nu s-au inteles factiunile !" << endl;
 cout << "Sunt chemati toti destructorii in ordinea inversa:" << endl;
 cout << "MEMBRU, FACTIUNE si PARTID care cheama FACTIUNE" << endl;
}
```

Prin rularea acestui program se obțin următoarele rezultate:

```
Creare obiect derivat MM (1, 2, 3)
Apelare constructor clasa FACTIUNE, cu i = 2
Apelare constructor clasa PARTID, cu a și b = 1 respectiv 2
Apelare constructor clasa FACTIUNE, cu i = 3
Apelare constructor clasa MEMBRU, cu a, b și c = 1, 2 respectiv 3

Partidul se dizolva ! Nu s-au inteles factiunile !
Sunt chemati toti destructorii in ordinea inversa:
MEMBRU, FACTIUNE si PARTID care cheama FACTIUNE
Apelare destructor clasa MEMBRU, cu m = 1
Apelare destructor clasa FACTIUNE, cu f = 3
Apelare destructor clasa PARTID, cu p = 1
Apelare destructor clasa FACTIUNE, cu f = 2
```

Fiecare funcție constructor și destructor conține o instrucțiune de scriere pentru a anunța că ea este cea executată în acel moment. Totodată, se inițializează și unicele variabile de tip **private** ale claselor. Trebuie remarcată sintaxa puțin mai deosebită a constructorilor claselor PARTID și MEMBRU. Astfel, constructorul clasei PARTID, PARTID (**int** a, **int** b) va apela constructorul clasei FACTIUNE sub forma : F(b), apelul complet fiind de forma:

```
PARTID (int a, int b) : F(b) {... }
```

Constructorul clasei derivate MEMBRU, MEMBRU (**int** a, **int** b, **int** c) va apela constructorul clasei din care derivă (dovadă, prezența unui șablon format din două argumente a și b) **: (a, b)** și după aceasta va apela constructorul clasei FACTIUNE, FF (c). Apelul complet arată astfel:

```
MEMBRU (int a, int b, int c) : (a, b), FF(c) {... }
```

Se pune următoarea întrebare: de ce în apelul constructorului unei clase a fost nevoie și de apelul constructorilor unor alte clase ? Răspunsul este următorul: întrucât în clasa de bază PARTID, am dorit să fie un obiect FACTIUNE, el trebuie inițializat. De aceea s-a folosit forma:

```
PARTID (lista_partid) : F (lista_factiune)
```

În cazul clasei MEMBRU, se apelează constructorul clasei de bază (din care derivă MEMBRU), fără însă a mai specifica numele obiectului, ca în cazul de mai sus, adică direct **: (a, b)** și deoarece clasa MEMBRU conține la rândul său un obiect încuibat (în cazul de față FF - FACTIUNE), va trebui inițializat și acest obiect.

Generalizând, se poate formula întrebarea: Care este ordinea de apel a constructorilor când se creează obiecte "conglomerate" (adică obiecte care conțin la rândul lor alte obiecte, fie din cauză că "moștenesc" proprietăți ale clasei de bază, fie pentru că se dorește ca ele să conțină aceste obiecte încuibate) ? Regula este următoarea: Primul este apelat constructorul clasei de bază, apoi constructorii pentru obiectele membre conținute în clasa de bază și în final, constructorul clasei derivate. În contextul acelorași obiecte conglomerate, regula care acționează la momentul distrugerii obiectului conglomerat (eliberării resurselor de memorie ocupate de către obiectul conglomerat) constă în apelul destructorilor în ordine exact inversă celei în care au fost apelați constructorii: primul va fi apelat destructorul clasei derivate, apoi destructorii obiectelor încuibate în acel obiect conglomerat și în final destructorul clasei de bază.

Revenind la programul de mai sus, se constată că după anunțarea începerii execuției programului prin mesajul: Creare obiect derivat MM (1, 2, 3), urmează crearea obiectului MM în sine. Acesta este însă un obiect conglomerat. El aparține de clasa derivată MEMBRU; aceasta conține la rândul său un obiect încuibat al clasei FACTIUNE, denumit FF; totodată, clasa MEMBRU este derivată din clasa de bază PARTID care, la rândul său, conține un obiect încuibat de tip FACTIUNE, denumit F. Începe crearea obiectului MM de tip MEMBRU. Conform regulii de mai sus, se apelează constructorul clasei de bază, PARTID. Cum clasa PARTID conține obiectul F de tip FACTIUNE, se dă controlul constructorului obiectului F, care anunță:

Apelare constructor clasa FACTIUNE, cu i = 2

Acest constructor conține două instrucțiuni:

```
cout << "Apelare constructor clasa FACTIUNE, cu i = " << i << endl;
f = i;
```

De ce variabila i a avut valoarea 2 ? Aceasta rezultă din repartizarea valorilor din lista obiectului MM (1, 2, 3), valori care la momentul intrării în constructorul clasei MEMBRU se repartizează astfel: a = 1, b = 2 și c = 3. Când se apelează constructorul clasei de bază, a și b au aceleași valori. Se vede că în instrucțiunea PARTID (**int** a, **int** b) : F(b) {... } se apelează F(b). Dar b = 2, deci i va fi egal cu 2.

În continuare, după părăsirea constructorului clasei FACTIUNE, se revine în contextul clasei de bază. Cum aici nu mai există și alte obiecte încuibate, se execută instrucțiunile constructorului acestei clase:

```
cout << "Apelare constructor clasa PARTID, cu a si b = " << a << " respectiv " << b << endl;
p = a;
```

Prima instrucțiune anunță: Apelare constructor clasa PARTID, cu a si b = 1 respectiv 2, după care în a doua instrucțiune, p va lua valoarea variabilei a, deci 1.

În sfârșit vine rândul constructorului clasei derivate. Se dă controlul fragmentului FF(c) din secvența de apel. Cum însă variabila c transmisă în acest context are valoarea 3, la momentul execuției instrucțiunilor acestui constructor, se vor anunța pe rând rezultatele următoare:

Apelare constructor clasa FACTIUNE, cu i = 3

Apelare constructor clasa MEMBRU, cu a, b și c = 1, 2 respectiv 3

Precizăm că ordinea apelului destructorilor este exact inversă ordinii apelului constructorilor.