CLASE ŞI OBIECTE

10

10.1. DEFINIȚIA CLASELOR ŞI ACCESUL LA MEMBRII

10.1.1. LEGĂTURA CLASĂ-STRUCTURĂ-UNIUNE

Aşa cum s-a subliniat în capitolul 9, o clasă reprezintă un ***tip abstract de date***, care încapsulează atât elementele de date (datele membre) pentru care s-a adoptat un anumit mod de reprezentare, cât şi operațiile asupra datelor (funcțiile membre, metode). În limbajul C++, structurile şi uniunile reprezintă cazuri particulare ale claselor, putând avea nu numai date membre, câmpuri de date (vezi capitolul 8), dar şi funcții membre. Singura diferență între structuri şi uniuni constă în faptul că la uniuni, pentru memorarea valorilor datelor membre se foloseşte aceeaşi zonă de memorie. Deosebirea esențială între structuri şi uniuni - pe de o parte - şi clase - pe cealată parte - constă în ***modul de acces*** la membrii: la ***structuri şi uniuni*** membrii (datele şi metodele) sunt ***implicit publici***, iar la ***clase*** - ***implicit privați*** (membrii sunt încapsulați). Lipsa unor modalități de protecție a datelor, face ca tipurile de date introduse prin structuri sau uniuni să nu poată fi strict controlate în ceea ce priveşte operațiile executate asupra lor. În cazul claselor, modul de acces la membrii tipului de date (în scopul protejării acestora) poate fi schimbat prin utilizarea modificatorilor de control ai accesului: `public`, `private`, `protected`.

10.1.2. DECLARAREA CLASELOR

Modul de declarare a unei clase este similar celui de declarare a structurilor şi a uniunilor:

```
class nume_tip{
modificator_control_acces:
                lista_membrilor;
} lista_variabile;
```

Clasa reprezintă un tip de date (definit de utilizator).
Membrii unei clase sunt:

- Datele membre - datele declarate în cadrul clasei;
- Metodele - funcțiile membre, funcțiile declarate sau definite în cadrul clasei. Se admite că în cadrul declarației de clasă să se specifice doar prototipurile funcțiilor membre, definițiile putând fi făcute oriunde în fişier, sau în alt fişier.

Pentru membrii care apar în `lista_membrilor` se poate preciza un anumit mod de acces.
`Modificator_control_acces` poate fi `public`, `private` sau `protected` (eventual `friend`, vezi paragraful 10.4.). Dacă nu se specifică, este considerat cel implicit (`private`). Modificatorul de acces `public` se utilizează pentru membrii care dorim să fie neprotejați, ultimii doi modificatori asigurând protecția membrilor din domeniul de acțiune a lor. Membrii cu acces `private` pot fi accesați numai prin

metodele clasei (sau prin funcțiile prietene, capitolul 10.4.). Cei cu acces `protected` posedă caracteristicile celor privați, în plus, putând fi accesați şi din clasele derivate. Specificatorii modului de acces pot apare în declararea clasei de mai multe ori, în orice ordine.
Domeniul unui modificator de acces ține din punctul în care apare modificatorul respectiv, până la sfârşitul declarației clasei sau al întâlnirii altui modificator de acces (exemplele 1,2).
Observațiile legate de prezența `nume_tip` sau `lista_variabile` (din capitolul 8) sunt valabile şi în cazul claselor.
Variabilele din `lista_variabile` sunt de tipul `nume_tip` şi se numesc ***instanțe*** (***obiecte***) ale clasei.
Observație: În cazul tipurilor de date definite cu ajutorul *structurilor*, se pot aplica modificatorii de acces. În cazul tipurilor definite cu ajutorul *uniunilor*, accesul implicit (`public`) nu poate fi modificat.

Exemplu:
```
class masina{
      char *culoare;     // dată membru la care accesul este, implicit, private
      int capacit_cil;   // dată membru la care accesul este, implicit, private
public:
      void citire_date();
      //metodă cu acces public, care permite introducerea datelor despre o instanță a clasei
      int ret_capacit();        //metodă cu acces public
};
```
Membrii `culoare` şi `capacitate_cil` (accesul `private`) pot fi accesați doar prin intermediul metodelor clasei.
Exemplu:
```
class persoana{
      char *nume; //dată membru privată
public:
      void citire_inf_pers(); //metodă publică
private:
      int varsta; //dată membru privată
};
```

Exercițiu: Să se definească tipul de date `dreptunghi`, cu ajutorul unei structuri, a unei uniuni şi a unei clase.
Datele membre sunt lungimea şi lățimea (variabilele `Lung`, `lat`).
Funcțiile membre sunt:

`void seteaza_dimen(double, double)` - primeşte ca argumente două valori reale şi inițializează datele membre cu valorile argumentelor.
`double arata_Lung( )` - returnează valoarea lungimii (a datei membre `Lung`).
`double arata_Lat( )` - returnează valoarea lățimii (a datei membre `lat`).
`double calcul_arie( )` - returnează valoarea ariei dreptunghiului.

//a) Implementarea tipului dreptunghi cu ajutorul unei structuri.
```
#include <iostream.h>
struct dreptunghi{
      double Lung, lat;
      void seteaza_dimen(double, double );//prototipul funcției seteaza_dimen
      double arata_Lung()
            {return Lung;}
      double arata_Lat()
            {return lat;}
      double calcul_arie()
            {return Lung*lat;}
};

void dreptunghi::seteaza_dimen(double L, double l)
      {Lung=L; lat=l;}
```

```
      void main()
      { dreptunghi a;
            double l1, l2; cout<<"Lungime="; cin>>l1;
            cout<<"Latime="; cin>>l2; a.seteaza_dimen(l1, l2);
            // sau: Lung=l1; lat=l2;
            cout<<"Dimensiunile dreptunghiului sunt:"<<a.arata_Lung();
            cout<<" si"<<a.arata_Lat()<<'\n';
            cout<<"Aria dreptunghiului:"<<a.calcul_arie()<<'\n';
            cout<<"Dimens structurii:"<<sizeof(a)<<'\n';
      }
```

//b) Implementarea tipului dreptunghi cu ajutorul unei uniuni

```
      #include <iostream.h>
      union dreptunghi{
            double Lung, lat;
            void seteaza_dimen(double, double );
            double arata_Lung()
                  {return Lung;}
            double arata_Lat()
                  {return lat;}
            double calcul_arie(double s)
                  {return s*lat;}
      };

      void dreptunghi::seteaza_dimen(double L, double l)
            {Lung=L; lat=l;}

      void main()
      { dreptunghi a; double l1, l2;
            cout<<"Lungime="; cin>>l1; cout<<"Latime="; cin>>l2;
            a.seteaza_dimen(l1, l1); cout<<"Lung. drept:"<<a.arata_Lung()<<'\n';
            double s1=a.arata_Lung(); a.seteaza_dimen(l2, l2);
            cout<<"Latimea dreptunghiului este:"<<a.arata_Lat()<<'\n';
            cout<<"Aria dreptunghiului:"<<a.calcul_arie(s1)<<'\n';
            cout<<"Dimens. uniunii:"<<sizeof(dreptunghi)<<'\n';
      }
```

În exercițiul 1 a, b se defineşte tipul `dreptunghi` printr-o structură, respectiv o uniune. Tipul conține atât datele membre, cât şi metodele care implementează operațiile care pot fi realizate asupra variabilelor de tipul dreptunghi. Metodele `arata_Lung`, `arata_Lat`, `calcul_arie` sunt *definite* în structură (uniune). Metoda `seteaza_dimen` este doar *declarată* în interiorul structurii (uniunii), fiind abia apoi definită. În varianta b (implementarea cu uniune, unde pentru memorarea valorilor datelor membre se utilizează aceeaşi zonă de memorie), pentru a păstra valorile atât pentru lungimea dreptunghiului, cât şi pentru lățime, metodele au fost modificate.

Nespecificând nici un modificator de control al accesului, toți membrii (date şi metode) sunt implicit **publici**. De aceea, de exemplu, atribuirea unei valori pentru data membră `Lung` se putea realiza, la fel de bine, în corpul funcției `main`, astfel: `Lung=l1;` (în exercițiul 1a, atribuirea se realizează cu ajutorul metodei `seteaza_dimen`).

//c) Implementarea tipului dreptunghi cu ajutorul unei clase

```
      #include <iostream.h>
      class dreptunghi{
            double Lung, lat;
      public:
            void seteaza_dimen(double, double );
            double arata_Lung()
                  {return Lung;}
            double arata_Lat()
                  {return lat;}
            double calcul_arie()
```

```
            {return Lung*lat;}
};
void dreptunghi::seteaza_dimen(double L, double l)
      {Lung=L; lat=l;}
void main()
{ dreptunghi a;double l1, l2;
      cout<<"Lungime="; cin>>l1;cout<<"Latime="; cin>>l2;
      a.seteaza_dimen(l1, l2);cout<<"Dimensiunile dreptunghiului sunt:";
      cout<<a.arata_Lung()<<" si"<<a.arata_Lat()<<'\n';
      cout<<"Aria dreptunghiului:"<<a.calcul_arie()<<'\n';
      cout<<"Dimens :"<<sizeof(a)<<'\n';
}
```

În exercițiul 1c se defineşte tipul de date `dreptunghi` cu ajutorul unei clase. Nivelul de acees implicit la membrii clasei este **`private`**. Dacă pentru metode nu s-ar fi folosit modificatorul de acces **public**, metodele ***nu*** ar fi putut fi folosite în funcția `main`.

10.1.3. OBIECTE

Un obiect este o dată de tip definit printr-o clasă. În exercițiul anterior, punctul `c`, în funcția `main`, se declară obiectul (variabila) **`a`** de tip `dreptunghi`. Spunem că obiectul **`a`** este o instanță a clasei `dreptunghi`. Se pot declara oricâte obiecte (instanțe) ale clasei. Aşa cum se observă din exemplu, declararea obiectelor de un anumit tip are o formă asemănătoare celei pentru datele de tip predefinit:

`nume_clasa lista_obiecte;`

Exemple:

```
dreptunghi a;
dreptunghi b, c, d;
```

10.1.4. MEMBRII UNEI CLASE

Datele membru se alocă distinct pentru fiecare instanță (atribute ale instanței) a clasei (pentru declararea obiectelor a, b, c, d de tip dreprunghi, vezi figura 10.1.). **Excepția** de la această regulă o constituie ***datele membru statice***, care există într-un singur exemplar, comun, pentru toate instanțele clasei.

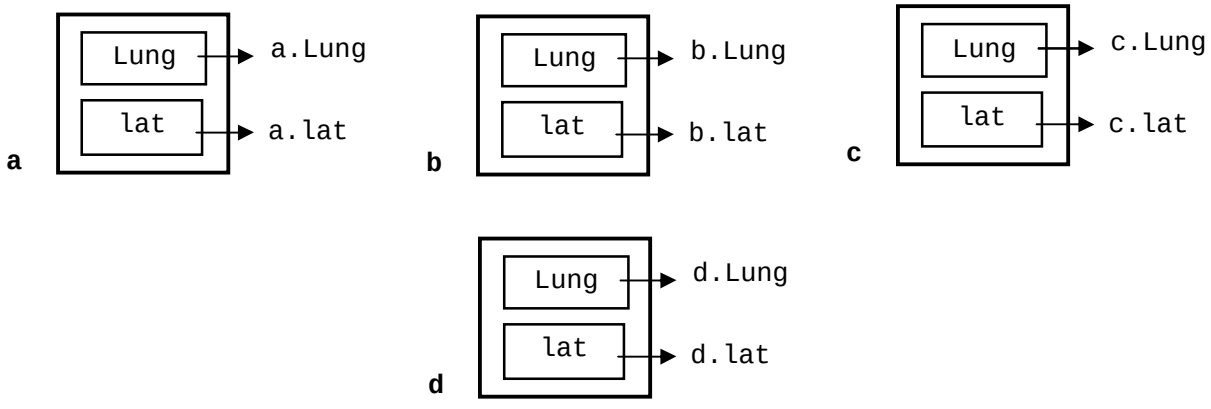

Figura 10.1. Alocarea memoriei pentru datele membre nestatice

Metodele figurează într-un singur exemplar, oricâte instanțe ale clasei ar exista.

În exemplul anterior, metoda `seteaza_dimen` este doar declarată în interiorul clasei, fiind abia apoi definită. La definirea funcției (`void dreptunghi::seteaza_dimen(double L, double l)`) s-a folosit operatorul **`::`** (***scope resolution operator***) care specifică relația de apartenență a metodei la tipul

dreptunghi. Operatorul cuplează **nume_clasa::nume_functie_membru** şi defineşte domeniul (scopul) în care acea funcţie va fi recunoscută. Prezenţa numelui clasei în faţa funcţiei membru este obligatorie, deoarece, altfel nu s-ar putea face distincţia între metode cu nume identice, care aparţin unor clase diferite.

O funcţie membru se apelează totdeauna în strânsă dependenţă cu un obiect din clasa respectivă. Legătura dintre obiect şi funcţia membră se face prin operatorul **.** sau operatorul **->**, după cum obiectul este desemnat prin nume sau prin pointer (vezi exemplu). În plus, **metodele statice** *pot fi apelate* ***independent*** *de un obiect al clasei*, folosind operatorul de rezoluţie (**::**).

Accesul la o metodă presupune o activitate de adresare, transparentă utilizatorului. De fapt, în interiorul obiectului creat se află doar punctatori la clasa din care provin. În interiorul definiţiei clasei se alocă o singură copie a fiecărei funcţie membră şi punctatorii respectiv, prin care obiectul va avea acces la metoda respectivă. Excepţia de la această regulă o constituie ***metodele virtuale*** (capitolul 12).

Exemplu: Fie clasa dreptunghi din exerciţiul anterior.

```
class dreptunghi{
      double Lung, lat;
public:
      void seteaza_dimen(double, double );
      double arata_Lung();
      double arata_Lat();
      double calcul_arie();
};
//.....................................................................
void main()
      { dreptunghi a;
      //.....................................................
      cout<<"Aria dreptunghiului:"<<a.calcul_arie()<<'\n';
      dreptunghi *pa;
      pa=&a;
      double arie=pa->calcul_arie();
}
```

Exerciţiu: Să urmărim exerciţiul următor, care ilustrează problemele legate de **membrii statici** ai unei clase (figura 10.2.).

```
#include <iostream.h>
#include <conio.h>
class exemplu
{     int i;       // dată membră privată, acces la ea doar prin metode
public:
      static int contor; // dată membră publica, neprotejată (scop didactic)
      void inc(void)
            {i++;}
      void arata_i()
            {cout<<"i="<<i<<'\n';}
      void inc_contor(void)
            {contor++;}
      void init(void)
            {i=0;}
      static void arata_contor()
            {cout<<"Contor="<<contor<<'\n';}
      static void functie(exemplu*);
} a1, a2, a3;

int exemplu::contor=0;   //iniţializarea datei membru statice

void exemplu::functie (exemplu *pe)
{//i+=3;     //eroare, nu se cunoaste obiectul care-l poseda pe i
pe->i++;     //corect, este specificat proprietarul lui i
contor++;    //variabilă statică, comună tuturor obiectelor
}
```

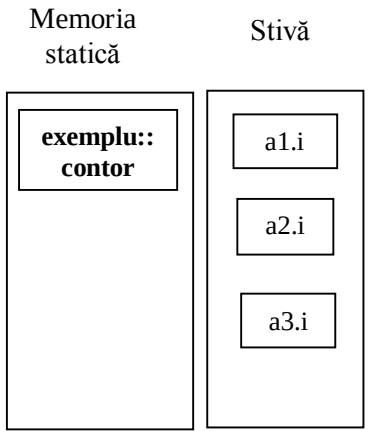

Figura 10.2. Alocarea memoriei pentru datele membru statice şi nestatice

```
void main()
{clrscr();
a1.init(); a2.init(); a3.init();          //a1.i=0, a2.i=0, a3.i=0
a1.arata_i();a2.arata_i();a3.arata_i();   //i=0, i=0, i=0
a1.inc(); a2.inc(); a3.inc();             //a1.i=1, a2.i=1, a3.i=1
a1.arata_i();a2.arata_i();a3.arata_i();   //i=1, i=1, i=1
a1.functie(&a1);                          //contor=1, i=2
exemplu::functie(&a2);                    //contor=2, i=2
//functie();                      //incorect
a1.inc_contor();                  //contor=3
exemplu::arata_contor();
a2.inc_contor();                  //contor=4
exemplu::arata_contor();
a3.inc_contor();                  //contor=5
exemplu::arata_contor();
exemplu::arata_contor();
exemplu::contor+=100;                     //membru public; contor=105
cout<<"Contor="<<exemplu::contor<<'\n';   //Contor=105
}
```

Din exemplul anterior, se poate observa că ***metoda statică*** `funcţie` poate fi apelată ca o metodă obişnuită, sau folosind operatorul **`::`**. Pentru ***data membră statică*** `contor` se rezervă o zonă de memorie comună obiectelor a1, a2, a3. Pentru *data membră* `i` se realizează o copie pentru fiecare instanță a clasei. Deasemenea, deoarece data membră contor este statică, nu aparține unui anume obiect, ea apare prefixată de numele clasei şi operatorul de apartenență.

Exerciţiu: Să se urmărească următorul exercițiu, care defineşte tipul `ex_mstat`, cu o dată membru statică (`s`) şi metodele statice (`set_s, ret_s`).

```
#include <iostream.h>
class ex_mstat{
      int a; static double s;
public:
      int ret_a(){return a;}
      void set_a(int x){a=x;}
      static double ret_s(){return s;}
      static void set_s(double x){s=x;}
      void set1_s(double x){s=x;}
      double ret1_s(){return s;}
};
```

`double ex_mstat::s;`
`/*`se rezervă spațiu în memoria statică pentru data membră statică `s`, care figurează într-un singur exemplar pentru toate instațele clasei `ex_mstat` (figura 10.3.)*/

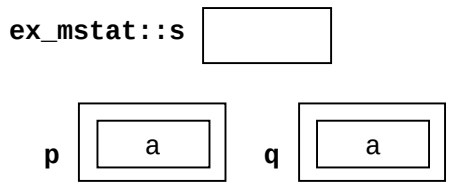

Figura 10.3. Alocarea memoriei pentru membrii clasei `ex_mstat`

```
void main()
{ex_mstat p,q;
p.set_a(100);
p.set_s(200); q.set_a(300);
cout<<"p.a="<<p.ret_a()<<" p.s="<<p.ret_s();
cout<<" ex_mstat::ret_s="<<ex_mstat::ret_s()<<'\n';//p.a=100 p.s=200 ex_mstat::ret_s=200
cout<<"q.a="<<q.ret_a()<<" q.s="<<q.ret1_s();
cout<<" ex_mstat::ret_s="<<ex_mstat::ret_s()<<'\n';//q.a=300 q.s=200 ex_mstat::ret_s=200
ex_mstat::set_s(500.20);
cout<<"p.a="<<p.ret_a()<<" p.s="<<p.ret_s();
cout<<" ex_mstat::ret_s()="<<ex_mstat::ret_s()<<'\n';
      //p.a=100 p.s=500.20 ex_mstat::ret_s=500.20
```

```
cout<<"q.a="<<q.ret_a()<<" q.s="<<q.ret1_s();
cout<<" ex_mstat::ret_s()="<<ex_mstat::ret_s()<<'\n';
      //q.a=300 q.s=500.20 ex_mstat::ret_s=500.20
q.set1_s(800.80);
cout<<"p.a="<<p.ret_a()<<" p.s="<<p.ret_s();
cout<<" ex_mstat::ret_s()="<<ex_mstat::ret_s()<<'\n';
      //p.a=100 p.s=800.20 ex_mstat::ret_s=800.20
cout<<"q.a="<<q.ret_a()<<" q.s="<<q.ret1_s();
cout<<" ex_mstat::ret_s()="<<ex_mstat::ret_s()<<'\n';
      //q.a=300 q.s=800.20 ex_mstat::ret_s=800.20
p.set1_s(999);
cout<<"p.a="<<p.ret_a()<<" p.s="<<p.ret_s();
cout<<" ex_mstat::ret_s()="<<ex_mstat::ret_s()<<'\n';
      //p.a=100 p.s=999 ex_mstat::ret_s=999
cout<<"q.a="<<q.ret_a()<<" q.s="<<q.ret1_s();
cout<<" ex_mstat::ret_s()="<<ex_mstat::ret_s()<<'\n';
      //q.a=300 q.s=999 ex_mstat::ret_s=999
}
```

Aşa cum se observă din exemplul anterior, **data membru statică** `s` figurează *într-un singur exemplar pentru instanţele* `p` şi `q`. Ea **poate fi modificată** prin **metoda statică** `set_s` sau prin **metoda** `set1_s`.
Apelul unei metode statice poate fi realizat *ca un apel al unei metode obişnuite*: `p.ret_s()`, sau folosind *operatorul de rezoluţie*: `ex_mstat::ret_s()`. Datorită *ultimului mod de apel, în care metoda statică* ***nu*** *este asociată unui obiect anume*, **în corpul funcţiilor statice, nu pot fi accesate decât datele membre statice**.

Observaţie: ***Nu*** trebuie confundaţi ***membrii statici ai unei clase*** cu datele care au ***clasa de memorare static***.

10.1.5. POINTERUL *THIS*

Fiecare ***funcţie membră*** posedă un argument ascuns, numit ***this***, argument transmis în mod automat de către compilator. Această variabilă (locală funcţiilor membru) reprezintă ***pointerul către obiectul curent*** (cel care apelează metoda). Să reimplementăm metoda `calcul_arie`, folosind acest pointer (deşi în această situaţie utilizarea pointerului `this` este redundantă).

Exemplu:

```
class dreptunghi{
      double Lung, lat;
public:
      //.....................................................
      void seteaza_dimen(double, double);
};
void dreptunghi::seteaza_dimen(double L, double l)
      {//Lung=L; lat=l;
      this->Lung=L; this->lat=l;}
```

Deoarece o **metodă statică** se apelează independent de un obiect al clasei, pointerul `this` **nu** *mai poate fi utilizat*.

10.1.6. DOMENIUL UNUI NUME, VIZIBILITATE ŞI TIMP DE VIAŢĂ

Înainte de a citi acest paragraf, trebuie revăzut capitolul 6.8.

10.1.6.1. Domeniul unui nume

Unui nume îi corespunde un domeniu, specificat prin declaraţia variabilei. În funcţie de poziţia declaraţiei (definirii) unui nume, domeniul poate fi:

- local (dacă numele este declarat într-un bloc);
- fişier (dacă numele este declarat în afara oricărui bloc sau declarație (definiție) de clasă);
- clasă.

Dacă un nume care are ca domeniu un fişier este redefinit într-un bloc inclus în domeniul său, el poate fi folosit în acest bloc dacă este **precedat de operatorul rezoluție**.

Exemplu:

```
#include <iostream.h>
int i=80;                       // i declarat în afara oricărei funcții, domeniul numelui este fişierul
void main()
{double i=99.9;                 // redeclararea variabilei i , începe domeniul local
cout<<"Valoarea lui i="<<i<<'\n';                   //Valoarea lui i=80
cout<<"Valoarea lui i="<<::i<<'\n';                 // Valoarea lui i=99.9
}
```

Domeniul numelui unui tip de date definit printr-o clasă (`struct` sau `union`, deoarece structurile şi uniunile sunt cazuri particulare de clase cu membrii publici) se stabileşte în mod similar domeniului oricărei variabile. ***Numele unui membru*** al unei clase are un domeniu de tip clasă. Ca orice nume, un nume de clasă poate fi redeclarat.

Exemplu:

```
class a{
//……..
};
//……..
{       //instrucțiune bloc aflată în domeniul numelui a
double a=99;          //redeclararea lui a
class::a x;  //x este un obiect de tipul a (definit printr-o clasă)
}
```

10.1.6.2. Vizibilitate

Domeniul de vizibilitate a unei variabile (obiect) este determinat de clasa de memorare a variabilei (obiectului), aşa cum prezintă tabelul 6.1. (capitolul 6). De obicei, domeniul de vizibilitate al unui nume coincide cu domeniul numelui.

10.1.6.3. Timp de viață

Timpul de viață a unei variabile (obiect) este determinat de clasa de memorare a variabilei (obiectului), aşa cum prezintă tabelul 6.1. (paragrafele 6.8., 6.9.). În limbajul C++, alocarea dinamică a memoriei se realizează cu operatorii `new` şi `delete` (capitolul 11).

10.2. FUNCȚII INLINE

La apelul unei funcții obişnuite se întrerupe execuția funcției apelante şi se execută un salt la adresa de memorie la care se găseşte corpul funcției apelate. La terminarea execuției funcției apelate se revine în funcția apelantă, reluându-se execuția cu instrucțiunea imediat următoare apelului de funcție. În situațiile în care corpul funcției apelate este format din câteva instrucțiuni, operațiile descrise anterior (implicate în apel şi revenire) pot fi mai complicate decât un apel prin expandare (în care apelul funcției este înlocuit cu însuşi corpul funcției apelate). Pentru eliminarea acestor dezavantaje, se folosesc funcțiile **`inline`**.

Prezența funcțiilor inline anunță compilatorul să nu mai genereze instrucțiunile în cod maşină necesare apelului şi revenirii, ceea ce conduce la mărirea timpului de compilare în favoarea micşorării timpului de execuție. Utilizarea funcțiilor inline se justifică doar în situațiile în care codul generat de compilator pentru execuția corpului funcției este mai mic decât codul generat pentru apel şi revenire.

Practic, funcțiile care au corpul format din maximum trei instrucțiuni şi nu conțin instrucțiuni repetitive (`for`, `while`, `do-while`), pot fi *declarate* ***`inline`***.

Declararea unei funcţii inline se realizează ***explicit***, specificând în antetul funcţiei respective cuvântul cheie **`inline`**.

```
inline tip_val_ret nume_fct (lista_declar_par_formali);
```

În cazul ***metodelor unei clase***, dacă acestea sunt definite în interiorul clasei, ele sunt considerate, ***implicit***, funcţii inline (în exerciţiul anterior, funcţiile `arată_Lung`, `arată_Lat` şi `calcul_arie` sunt, implicit, funcţii inline). Există şi posibilitatea de a declara metoda la declararea clasei şi de a specifica, explicit, că este funcţie inline la definirea funcţiei.

Exemplu: Dacă se doreşte ca metoda `seteaza_dim` din exerciţiul anterior să fie funcţie inline, fără a modifica declaraţia tipului `dreptunghi`, se poate proceda astfel:

```
class dreptunghi{
      // . . .
public:
      // . . .
      void seteaza_dimen(double, double );// declararea metodei
      // . . .
};
inline void dreptunghi::seteaza_dimen(double L, double l)//funcţie inline, explicit
      {Lung=L; lat=l;}
```

Prefixarea definiţiei funcţiei `seteaza_dimen` cu cuvântul cheie `inline` este echivalentă cu ***definirea metodei*** în cadrul declaraţiei clasei `dreptunghi`.

Exerciţiu: Să se definească tipul de date `complex`, cu *datele membru* parte reală şi parte imaginară. *Operaţiile* care pot fi realizate asupra datelor de acest tip, vor fi:
Citirea unei date de tip complex (citirea valorilor pentru partea reală şi cea imaginară); afişarea unei date de tip complex; calculul modulului unui complex; calculul argumentului unui complex; incrementarea părţii imaginare; decrementarea părţii imaginare; funcţii care returnează valoarea părţii reale şi a părţii imaginare a unei date de tip complex; adunarea a două date de tip complex; înmulţirea a două date de tip complex.

```
#include <iostream.h>
#include <math.h>
#define PI 3.14159
class complex{
      double real, imag;
public:
      int citire();
      void afisare();
      double modul();
      double arg();
      void incrpi()
      //incrementeaza partea imaginara; FUNCŢIE INLINE, implicit, fiind definită în interiorul clasei
            { imag++;}
      inline void decrpi();//decrementarea partii imaginare
      double retreal(); //returneaza partea reala
      double retimag(); //returneaza partea imaginara
      void adun_c(complex, complex);//aduna 2 numere complexe
      void inm_c(complex*, complex*);//produsul a 2 numere complexe
};

inline double complex::modul()
      { return sqrt(real*real+imag*imag);}
int complex::citire()
      { cout<<"P. reala:"; if (!(cin>>real)) return 0;
      cout<<"P. imag:";if (!(cin>>imag)) return 0 ;
      return 1; }
void complex::afisare()
      { if (imag>=0)
            cout<<real<<"+"<<imag<<"*i"<<"\n";
        else cout<<real<<imag<<"*i\n";}
double complex::arg()
```

```
      {if (real==0 && imag==0)      return 0.0;
      if (imag==0)      //z=p. reala
            if (real>0) return 0.0;
      else return PI;
      if (real==0)
            if (imag>0) return PI/2;
            else  return (3*PI)/2;
      double x=atan(imag/real);
      if (real<0) return PI+x;
      if (imag<0) return 2*PI+x;
      return x;}

inline void complex::decrpi()
      { imag--;}
double complex::retreal()
      { return real;}
double complex::retimag()
      { return imag; }
void complex::adun_c (complex x1, complex x2)
      {real=x1.real+x2.real;
      imag=x1.imag+x2.imag;}
void complex::inm_c(complex *x1, complex *x2)
      {real=x1->real*x2->real-x1->imag*x2->imag;
      imag=x1->real*x2->imag+x1->imag*x2->real;}

void main()
{complex z1;z1.citire();
cout<<"z1=";z1.afisare();
complex z2;z2.citire();cout<<"z2=";z2.afisare();
cout<<"Modulul z2="<<z2.modul()<<'\n';
cout<<"Agument z2="<<z2.arg()<<'\n';
cout<<"P. reala z2="<<z2.retreal()<<"P imag z2="<<z2.retimag()<<'\n';
z2.incrpi();cout<<"Dupa increm p imag="<<z2.retimag()<<'\n';z2.afisare();
complex z3;z3.adun_c(z1,z2);cout<<"Adunare z1+z2=";z3.afisare();
complex*pz1=&z1,*pz2=&z2;z3.inm_c(pz1,pz2);
cout<<"Inmultire z1+z2=";z3.afisare();
}
```

10.3. CONSTRUCTORI ŞI DESTRUCTORI

10.3.1. INIŢIALIZAREA DATELOR

La declararea datelor de tip predefinit sau definit de utilizator prin structuri, uniuni sau clase, compilatorul alocă o zonă de memorie corespunzătoare tipului respectiv. Este indicat ca în cazul în care datele structurate au ca membrii pointeri, să se aloce memorie în mod dinamic.

În general, datele statice sunt iniţializate automat cu valoarea 0. Celelalte categorii de date, nu sunt iniţializate. Iniţializarea datelor simple de tip predefinit se poate realiza după declararea acestora, sau în momentul declarării.

Exemple:

`int i;  i=30;`//declararea variabilei `i`, apoi iniţializarea ei prin atribuire

`char c='A';` //declararea şi iniţializarea variabilei `c`

Iniţializarea datelor structurate se poate realiza în momentul declarării acestora, prin listele de iniţializare.

Exemple:

//1

`int a[]={20, 30, 40, 50};` //declararea şi iniţializarea vectorului `a`

//2

`double m[2][3]={{1.1,2.2,3.3}, {11.11,22.22,33.33}};`//declararea şi iniţializarea matricii `m`

```
//3
struct pers{
      char nume[20]; int varsta; double salariu;
}p={"Popescu", 20, 3000000};
```

În cazul tipurilor de date definite cu ajutorul claselor, care au date membru private, listele de iniţializare nu pot fi utilizate. Pentru a elimina aceste neajunsuri, limbajul C++ oferă mecanismul ***constructorilor*** şi al ***desctructorilor***.

10.3.1. CONSTRUCTORI

Constructorii sunt **metode** speciale care folosesc la crearea şi iniţializarea instanţelor unei clase. Constructorii au acelaşi nume ca şi clasa căreia îi aparţin şi sunt apelaţi de fiecare dată când se crează noi instanţe ale clasei. Constructorii asigură iniţializarea corectă a tuturor variabilelor membre ale obiectelor unei clase şi constituie o garanţie a faptului că iniţializarea unui obiect se realizează o singură dată. O clasă poate avea mai mulţi constructori (exemplul 3), care diferă între ei prin numărul şi tipul parametrilor acestora. Acest lucru este posibil deoarece limbajul C++ permite ***supradefinirea*** (***overloading***) funcţiilor.

Supraîncarcarea (supradefinirea) reprezintă posibilitatea de a atribui unui nume mai multe semnificaţii, care sunt selectate în funcţie de context. Practic, se pot defini funcţii cu acelaşi nume, dar cu liste de parametri diferite, ca număr şi/sau ca tipuri de parametri. În momentul apelului funcţiei, selectarea funcţiei adecvate se face în urma comparării tipurilor parametrilor efectivi cu tipurile parametrilor formali. De aceea, declararea unor funcţii cu acelaşi nume şi acelaşi set de parametri este ilegală şi este semnalată ca eroare la compilare.

La întâlnirea declaraţiei unui obiect, se apelează automat un constructor al clasei respective. La fiecare instanţiere a clasei se alocă memorie pentru datele membre. Deci pentru fiecare obiect declarat se alocă memorie pentru datele membre ale clasei. Excepţie de la această regulă o constituie ***datele membru statice***. Acestea figurează într-un singur exemplar pentru toate instanţele clasei respective. Funcţiile membru există într-un singur exemplar pentru toate instanţele clasei. Ordinea în care sunt apelaţi constructorii corespunde ordinii declarării obiectelor.

Proprietăţile constructorilor:

- Constructorii au acelaţi nume ca şi numele clasei căreia îi aparţin;
- Nu întorc nici o valoare (din corpul lor lipseşte intrucţiunea `return`; în antetul constructorilor nu se specifică niciodată - la tipul valorii returnate - cuvântul cheie `void`);
- Constructorii unei clase nu pot primi ca parametri instanţe ale clasei respective, ci doar pointeri sau referinţe la instanţele clasei respective;
- Apelul constructorului se realizează la declararea unui obiect;
- Adresa constructorilor nu este accesibilă utilizatorului;
- Constructorii nu pot fi metode virtuale (capitolul 12);
- În cazul în care o clasă **nu** *are nici constructor declarat de către programator*, compilatorul generează un ***constructor implicit***, fără nici un parametru, cu lista instrucţiunilor vidă. Dacă există un constructor al programatorului, compilatorul nu mai generează constructorul implicit (exemplul 2);
- Parametrii unui constructor nu pot fi de tipul definit de clasa al cărei membru este constructorul.

Ca orice altă funcţie în limbajul C++, constructorii pot avea parametri impliciţi (vezi capitolul 6.5.), fiind numiţi *constructori impliciţi.* Varianta constructorului cu parametri impliciţi poate fi adoptată în toate cazurile în care constructorul nu necesită argumente. Dacă toţi parametrii unui constructor sunt impliciţi, apelul constructorului are forma unei simple declaraţii (exemplul 1). Constructorii pot fi apelaţi şi în mod explicit (exemplul 1). În cazul în care dorim să instanţiem obiecte atât iniţializate, cât şi neiniţializate se poate folosi un constructor implicit vid, care se va apela la instanţierea obiectelor neiniţializate (exemplul 3).

Exemplul1: Pentru clasa complex s-a definit un constructor cu parametri impliciţi; din acest motiv s-a putut face declaraţia `"complex z1;"` . În ultima linie a programului, pentru obiectul z4, constructorul este apelat în mod explicit.

```
class complex
{     double real,imag;
 public:
            complex(double x=0, double y=0);     // Constructor implicit
};
complex::complex(double x, double y)
      {real=x;  imag=y; }
void main()
{complex z1;              //z1.real=0, z1.imag=0
complex z2(1);            //z2.real=1, z2.imag=0
complex z3(2,4);          //z3.real=2, z3.imag=4
complex z4=complex();     //apel explicit al constructorului
}
```

La apelul explicit al constructorului: **`complex z4=complex();`**
Evaluarea expresiei **`complex()`** conduce la:

- ✓ Crearea unui ***obiect temporar*** de tip punct (obiect cu o adresă precisă, dar inaccesibil);
- ✓ Apelul constructorului pentru acest obiect temporar;
- ✓ Copierea acestui obiect temporar în `z4`.

Exemplul2: Pentru clasa complex există un constructor explicit, compilatorul nu mai creează unul implicit.

```
class complex
{     double real,imag;
public:
      complex(double x,double y)     // funcție constructor inline
            { real=x; imag=y;}
};
void main()
{complex z1(2,3);
complex z; // Eroare : nu există constructor implicit
}
```

Exemplul3: Definirea unui constructor implicit vid, care se va apela la instanțierea obiectelor neinițializate.

```
#include<iostream.h>
class data{
      int zi,luna,an;
public:
      data()   { }                  // constructor implicit vid
      data(int z,int l,int a)       // constructor cu parametri
            { zi=z;luna=l;an=a; }
};
void main()
{data d;                            // apelul constructorului vid
data d1(12,11,1998);                // apelul constructorului cu parametri
}
```

10.3.1.1. Constructori cu liste de inițializare

În exemplele anterioare, constructorii inițializau membrii unui obiect prin atribuiri. Există şi modalitatea de a iniția membrii printr-o ***listă de instanțiere*** (***inițializare***), care apare în implementarea constructorului, între antetul şi corpul acestuia. Lista conține operatorul **:**, urmat de ***numele*** fiecărui membru şi ***valoarea de inițializare,*** în **ordinea** în care membrii apar în definiția clasei.

Exemplu: Pentru clasa complex s-a implementat un constructor de inițializare cu listă de instanțiere

```
class complex {
      double real,imag;
 public:
```

```
      complex(double x, double y);  //constructor
};
complex::complex(double x, double y)
  :real(x),imag(y)      // Listă de inițializare a membrilor
      { return; }
void main()
      { complex z1(1,3),z2(2,3); }
// Sau:
class complex {
      double real,imag;
 public:
      complex(double x, double y) :real(x),imag(y) { }
      //constructor cu listă de inițializare
};
```

10.3.1.2. Constructori de copiere

Pentru o clasă, se poate defini un contructor de copiere, care să permită copierea obiectelor. Deoarece parametrii unui constructor nu pot fi de tipul definit de clasa al cărei membru este, constructorul de copiere pentru clasa `cls`, are, de obicei, prototipul:

```
cls (const cls &);
```

Parametrul transmis prin referință este obiectul a cărui copiere se realizează, modificatorul de acces `const` interzicând modificarea acestuia. Constructorul de copiere poate avea şi alți parametri, care trebuie să fie impliciți.
Dacă programatorul nu defineşte un constructor de copiere, compilatorul generează un asemenea constructor, implicit.

În situațiile în care un tip de date are ca membrii pointeri, este necesară implementarea unui constructor pentru inițializare (este de dorit să se aloce dinamic memorie) şi a unui constructor de copiere.

10.3.2. DESTRUCTORI

Destructorii sunt metode ale claselor care acționează în sens invers, complementar, față de constructori. Constructorii sunt folosiți pentru alocarea memoriei, inițializarea datelor membru sau alte operații (cum ar fi, incrementarea unui contor pentru instanțele clasei). Constructorul este apelat în momentul declarării obiectelor.
Destructorul eliberează memoria alocată de constructori. Destructorul este apelat automat, la ieşirea din blocul în care este recunoscut acel obiect.

Proprietățile destructorilor

- Destructorul are acelaşi nume ca şi clasa a căror metodă este;
- Numele destructorului este precedat de semnul **~**;
- O clasă are un singur destructor;
- Destructorul nu are parametri şi nu returnează nici o valoare (antetul nu conține cuvântul cheie `void`, iar în corpul destructorului nu apare instrucțiunea `return;`);
- Dacă programatorul nu a definit un destructor, compilatorul generează automat un destructor pentru clasa respectivă;
- Destructorii se apelează la încheierea timpului de viață a obiectelor, în ordine inversă apelurilor constructorilor;
- Obiectele dinamice nu se distrug automat, deoarece doar programatorul ştie când nu mai este necesar un astfel de obiect.

Exerciţiu:

Să se definească tipul **punct**, cu datele membre `x` şi `y`, reprezentând abscisa şi ordonata unui punct. Operaţiile care pot fi realizate asupra obiectelor de tip punct, sunt: `afişare` (afişează coordonatele unui punct), `deplasare` (deplasează un punct, noile coordonate ale punctului fiind obţinute prin adunarea unor valori transmise ca parametri, la valorile anterioare ale coordonatelor), `abscisa` (returnează valoarea abscisei), `ordonata` (returnează valoarea ordonatei). Se vor implementa, deasemenea, *constructor cu parametri impliciţi*, *constructor având ca parametri* valorile abscisei şi a ordonatei, *constructor de copiere* şi *destructor*.

Să se definească tipul **segment**, cu datele membre `A` şi `B`, de tip `punct`, reprezentând capetele unui segment (originea şi vârful). Operaţiile care pot fi realizate asupra obiectelor de tip segment, sunt: `afişare` (afişează coordonatele capetellor segmentului), `deplasare` (translatează un segment, deplasând capetele acestuia cu valorile transmise ca parametri), `origine` (returnează originea segmentului), `vârf` (returnează vârful segmentului). Se vor implementa, deasemenea, *constructor*, *constructor de copiere* şi *destructor*.

Să se testeze tipurile de date punct şi segment.

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
//CLASA PUNCT
class punct
{ double x,y;
public:
      punct()
      {x=0;y=0;cout<<"Constr. implicit pentru punct("<<x<<","<<y<<")\n";}
      //constructor initializare
      punct(double,double);
      punct(punct&);                  //constructor copiere
      ~punct();                       //destructor
      double abscisa(){return x;}
      double ordonata(){return y;}
      void afisare();
      void deplasare(double,double);
};

//CLASA SEGMENT
class segment
{private:
      punct A,B;
 public:
      segment(punct&,punct&);         //constructor
      segment(segment&);              //constructor copiere
      ~segment();                     //destructor
      punct origine();
      punct varf();
      void afisare();
      void translatie(double,double);
};

//METODELE CLASEI PUNCT
punct::punct(double valx,double valy)
      { x=valx;y=valy; cout<<"Constructor punct ("<<x<<","<<y<<")\n"; }
punct::~punct()
      {cout<<"Destructor punct ("<<x<<","<<y<<")\n";}
punct::punct( punct &P)
      { x=P.x;y=P.y; cout<<"Constructor copiere pct ("<<x<<","<<y<<")\n";}
void punct::deplasare(double dx,double dy)
      {x+=dx; y+=dy;}
```

```
void punct::afisare()
      { cout<<"Punct ("<<x<<','<<y<<')';}

//METODELE CLASEI SEGMENT
segment::segment(punct &A1,punct &B1)
      { A=A1;B=B1;cout<<"Constructor segment[";A.afisare();B.afisare();
      cout<<"]\n";}
segment::segment(segment &AB)
      { A=AB.A; B=AB.B; cout<<"Constructor copiere segment [";
      A.afisare(); B.afisare();cout<<"]\n";}
punct segment::origine()
      { return A;}
punct segment::varf()
      { return B;}
void segment::afisare()
      { cout<<"[";A.afisare();cout<<','; B.afisare();cout<<"]"<<'\n'; }
segment::~segment()
      { cout<<"Destructor segment [";A.afisare(); cout<<",";B.afisare();
      cout<<"]\n";}
void segment::translatie(double dx,double dy)
      { A.deplasare(dx,dy); B.deplasare(dx,dy);}

void main()
{clrscr();
punct P(7.8,-20.4),Q(-4.82,8.897),A,B;
      /*Constructor punct (7.8,-20.4) (Pentru punctul P)
      Constructor punct (-4.82,8.897) (Pentru punctul Q)
      Constr. implicit pentru punct(0,0)
      Constr. implicit pentru punct(0,0) (pentru punctele A, B)*/
punct P3, Q3;
      // Constr. implicit pentru punct(0,0)Constr. implicit pentru punct(0,0) (pentru punctele P3, Q3)
segment S(P,Q);
      /* Constr. implicit pentru punct(0,0)Constr. implicit pentru punct(0,0)
      (pentru membrii A, B ai obiectului S, deci pentru S.A şi S.B)
      Constructor segment[Punct (7.8,-20.4)Punct (-4.82,8.897)] (pentru obiectul S, de tip segment) */
segment S1(P3,Q3);
      /* Constr. implicit pentru punct(0,0)
      Constr. implicit pentru punct(0,0) (pentru membrii A, B ai obiectului S1, deci pentru S1.A şi S1.B)
      Constructor segment[Punct (0,0)Punct (0,0)] (pentru obiectul S1, de tip segment) */
printf("Apasa un car. ptr. continuare!\n");  getch();
cout<<"Punctele:\n";
P.afisare();cout<<'\n';       Q.afisare();cout<<'\n';
P3.afisare();cout<<'\n';      Q3.afisare();cout<<'\n';
A.afisare(); cout<<'\n';      B.afisare();cout<<'\n';
cout<<"\nSegment:";     S.afisare();      cout<<'\n';
      /* Punctele:
      Punct (7.8,-20.4) (pentru obiectul P)
      Punct (-4.82,8.897) (pentru obiectul Q)
      Punct (0,0) (pentru obiectul A)
      Punct (0,0) (pentru obiectul B)
      Punct (0,0) (pentru obiectul P3)
      Punct (0,0) (pentru obiectul Q3)
      Segment:[Punct (7.8,-20.4),Punct (-4.82,8.897)] */
punct D(1,2); punct C;
      // Constructor punct (1,2)Constr. implicit pentru punct(0,0) (pentru punctele D, C)
C=D;                  //operație de atribuire
C.afisare();          // Punct (1,2) (pentru punctul C)
getch();
punct CC=C;           // Constructor copiere pct (1,2)
cout<<"In urma copierii:"; CC.afisare();
```

```
        // În urma copierii:Punct (1,2) (pentru punctul CC)
cout<<"Se deplaseaza punctul CC cu valorile 10, 20. Noile coord.=";
CC.deplasare(10, 20); CC.afisare();
        // Se deplaseaza punctul CC cu valorile 10, 20. Noile coord.=Punct (11,22)
cout<<"Abscisa CC="<<CC.abscisa()<<" Ordonata CC="<<CC.ordonata()<<'\n';
        //Abscisa CC=11 Ordonata CC=22
cout<<"Varf segment S="; (S.varf()).afisare();
        // Varf segment S=Constructor copiere pct (-4.82,8.897) (metoda varf returneaza un punct, copiere)
        // Punct (-4.82, 8.897)
        //Destructor punct (-4.82,8.897)
cout<<"Origine segment S="; CC=S.origine();
        /* Origine segment S=Constructor copiere pct (7.8,-20.4) (metoda origine returneaza un punct,
        copiere) Destructor punct (7.8,-20.4) */
CC.afisare();       // Punct (-4.82, 8.897)
S1=S;               //operatie de atribuire
S1.afisare();
        // Punct (7.8,-20.4)[Punct (7.8,-20.4),Punct (-4.82,8.897)]
cout<<"Translatie S1 cu 100,1000. S1 translatat este:";
S1.translatie(100, 1000); S1.afisare();
        // Translatie S1 cu 100,1000. S1 translatat este:[Punct (107.8,979.6),Punct (95.18,1008.897)]
segment S2=S1;      /* Constr. implicit pentru punct(0,0) (pentru S2.A)
Constr. implicit pentru punct(0,0) (pentru S2.B)
Constructor copiere segment [Punct (107.8,979.6)Punct (95.18,1008.897)]
Destructor segment [Punct (107.8,979.6),Punct (95.18,1008.897)]*/
cout<<"Segment S2 obtinut prin copiere:";
S2.afisare();       // Segment S2 obtinut prin copiere:[Punct (107.8,979.6),Punct (95.18,1008.897)]
cout<<"Iesire din main\n";      // Iesire din main
}
```

/* La ieşirea din funcţia main, deci la terminarea duratei de viaţă a obiectelor, se apelează automat destructorii, în ordinea inversă în care au fost apelaţi constructorii, astfel:

Destructor segment [Punct (107.8,979.6),Punct (95.18,1008.897)] *(pentru segmentul S2)*
Destructor punct (95.18,1008.897) *(pentru membrii B, respectiv A, ai segmentului S2: S2.B, apoi S2.A)*
Destructor punct (107.8,979.6)
Destructor punct (7.8,-20.4) (pentru CC)
Destructor punct (1,2) (pentru C)
Destructor punct (1,2) (pentru D)
Destructor segment [Punct (107.8,979.6),Punct (95.18,1008.897)] *(pentru segmentul S1)*
Destructor punct (95.18,1008.897) *(pentru membrii B, respectiv A, ai segmentului S1: S1.B, apoi S1.A)*
Destructor punct (107.8,979.6)
Destructor segment [Punct (7.8,-20.4),Punct (-4.82,8.897)] *(pentru segmentul S)*
Destructor punct (-4.82,8.897) *(pentru membrii B, respectiv A, ai segmentului S: S.B, apoi S.A)*
Destructor punct (7.8,-20.4)
Destructor punct (0,0) *(pentru punctul Q3)*
Destructor punct (0,0) *(pentru punctul P3)*
Destructor punct (0,0) *(pentru punctul B)*
Destructor punct (0,0) *(pentru punctul A)*
Destructor punct (-4.82,8.897) *(pentru punctul Q)*
Destructor punct (7.8,-20.4) *(pentru punctul P)* */

Exerciţiul evidenţiază următoarele probleme:

1. În situaţia în care o clasă `C1` are ca *date membre obiecte* ale altei clase `C2` (clasa `segment` are ca date membre obiecte de tipul `punct`), la *construirea unui obiect* din `C1`, se apelează întâi constructorul `C2` pentru membrii (de tip C2), apoi constructorul `C1`.
   Un astfel de exemplu îl constituie declararea segmentului S: **`segment S(P,Q);`**

Se apelează întâi constructorul implicit al clasei punct pentru membrii A şi B ai segmentului S (deci pentru S.A şi S.B), apoi constructorul clasei segment (figura 10.4.). La distrugerea obiectului din clasa C1, destructorii sunt apelaţi în ordinea inversă constructorilor (întâi se apelează destructorul clasei segment - învelişul exterior, apoi destructorul pentru membrii de tip punct).

2. Să revedem secvenţa:
   ```
   punct D(1,2); punct C; C=D;
   ```
   În acest caz, se realizează o atribuire, membru cu membru, echivalentă cu `C.x=D.x` şi `C.y=D.y`.

3. Să revedem secvenţa:
   ```
   punct CC=C;
   ```
   În acest caz, se apelează constructorul de copiere, care crează punctul CC prin copierea punctului C. Apelul constructorului de copiere se poate realiza şi explicit:
   ```
   punct CC=punct(C);
   ```
4. Parametrii transmişi unei funcţii pot fi obiecte, pointeri către obiecte sau referinţe către obiecte. Valoarea returnată de o funcţie poate fi un obiect, pointer către obiect sau referinţă către obiect.

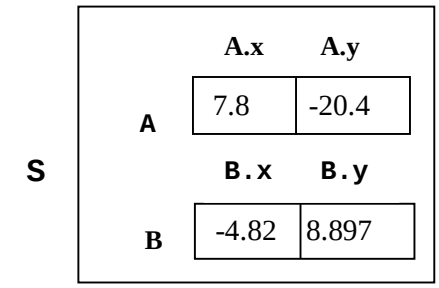

Figura 10.4. Apelul constructorilor

```
segment::segment(punct &A1,punct &B1)
      { A=A1;B=B1;cout<<"Constructor segment\n";}
```

Constructorul primeşte ca parametri ***referinţe*** către obiecte de tipul punct. Apelul constructorului:

```
segment S(P, Q);
```

Parametrii efectivi P şi Q sunt referinţe pentru A1 şi B1 (aceleaşi obiecte). Ca urmare, se apelează cei doi constructori impliciţi pentru membrii A şi B ai segmentului S. În urma operaţiei de atribuire din corpul constructorului segmentului, ei sunt iniţializaţi. Mesajele:

"Constructor pct implicit!!" *(pentru membrul A al segmentului S)*
"Constructor pct implicit!!" *(pentru membrul B al segmentului S)*
"Constructor segment" *(pentru segmentului S)*

Constructorului puteau să i se transmită parametri prin **pointeri**:

```
segment::segment(punct *A1,punct *B1)
      { A=*A1;B=*B1;cout<<"Constructor segment\n";}
```

Apelul: `segment S(&P, &Q);`

Parametrii formali `A1` şi `B1` sunt iniţializaţi în momentul apelului constructorului cu adresele punctelor P, respectiv Q. Situaţia este similară celei anterioare, mesajele obţinute sunt identice celor obţinute în cazul transmiterii parametrilor prin referinţă.

Constructorului puteau să i se transmită parametri prin **valoare**:

```
segment::segment(punct A1,punct B1)
      { A=A1;B=B1;cout<<"Constructor segment\n";}
```

Apelul: `segment S(P, Q);`

În această situaţie, la apel, pentru parametrii formali A1 şi B1 se rezervă memorie pe stivă: obiectele locale constructorului, A1 şi B1, sunt iniţializate prin copiere (la transmiterea parametrilor prin valoare, se realizează o copiere a parametrilor efectivi în parametrii formali, vezi capitolul 6.3.1.). La terminarea execuţiei corpului funcţiei, punctele A1 şi B1 sunt distruse. De aceea, mesajele din această situaţie, sunt:

"Constructor copiere punct!!" *(pentru A1, local constructorului)*
"Constructor copiere punct!!" *(pentru B1, local constructorului)*
"Constructor pct. implicit!!" *(pentru membrul A al segmentului)*
"Constructor pct. implicit!!" *(pentru membrul B al segmentului)*
"Constructor segment!" *(pentru segmentul S)*
"Destructor punct!" *(pentru B1, la ieşirea din constructor)*
"Destructor punct!" *(pentru A1, la ieşirea din constructor)*

Exerciţiu: Pentru tipurile punct şi segment implementate anterior, să se scrie şi să se testeze următorul program, în care obiectele A, B (de tip punct) şi AB (de tip segment) sunt globale (declarate în afara oricărei funcţii). Se folosesc, deasemenea, variabile locale statice. Pentru variabilele globale (`A`, `B`, `AB`) şi cele locale declarate explicit statice (`P1` din `test1`, `U` şi `V` din blocul interior funcţiei `main`), se alocă memorie statică.

Pentru variabilele locale se alocă memorie automatic, pe stivă. Să se urmărească evidenţieze crearea şi distrugerea obiectelor statice şi automatici, domeniul de vizibilitate şi timpul de viaţă.

```
class punct{ //.....
};

class segment
{           //.....
};
//Implementarea metodelor clasei punct
//Implementarea metodelor clasei segment

punct test1()
      { cout<<"Intrare in test1\n";
      static punct P1(20,10);
      P1.deplasare(1,1); cout<<"Iesire din test1\n";return P1;}
punct test2()
      { punct P1(100,100);P1.deplasare(10,20);return P1; }

punct A(1,2), B;
segment AB(A, B);
void main()
{cout<<"S-a intrat in main!\n";     punct E(1, 1), F, G;
F=test1();  cout<<"F="; F.afisare();
getch();G=test2();cout<<"Intrare in test1\n";cout<<"G="; G.afisare();
      {
      cout<<"Intrare in blocul interior\n";
      static punct U(5,2);punct C(9,9), D(5.5,6.6);static punct V(8,3);getch();
      F=test1();  cout<<"Intrare in test1\n";F.afisare();G=test2();
      cout<<"Intrare in test2\n";G.afisare();
      cout<<"Iesire din blocul interior\n";
      }
getch();A.afisare();F.afisare();AB.afisare();AB.translatie(10, 10);
cout<<"Segment translatat:";  AB.afisare();
cout<<"Segmentul AB are originea:"; (AB.origine()).afisare();
cout<<"Segmentul AB are varful:";   (AB.varf()).afisare();
cout<<"Iesire din main()\n";
}
```

10.3.4. TABLOURI DE OBIECTE

Obiectele de acelaşi tip pot fi grupate în tablouri. Dacă un tablou este declarat fără a fi iniţializat, pentru construirea elementelor sale se apela constructorul implicit. Elementele unui tablou pot fi iniţializate şi cu ajutorul constructorilor cu parametri.

Exemplu: Fie clasele punct şi segment din exerciţiul anterior.

```
class punct{       //.........
};
class segment{     //.........
};
//implementarea metodelor claselor punct si segment
void main()
{punct P(7, -7), Q(-8, 8);
punct PC=P, QC=Q;
punct V[3]; //apelul constructorului implicit pentru fiecare din cele 3 elemente ale vectorului V
punct V1[2]={P, Q};      //apelul constructorului de copiere pentru elementele V1[0] si V1[1]
punct V2[2]={punct(10,10), punct(100,100)};
//apelul constructorului cu parametri pentru fiecare din cele 2 elemente, V2[0] si V2[1]
segment SV[2];
//EROARE: deoarece exista un constructor cu parametri, nu se genereaza automat constructorul implicit
```

```
segment SV[2]={segment(PC, P), segment(QC, Q)};
segment SV[2]={segment(punct(5,5), punct(55,55)), segment (punct(10,10),
punct(100,100))};
}
```

10.4. FUNCȚII PRIETENE

Funcțiile prietene (`friend`) sunt funcții ne-membre ale unei clase, care ***au acces la datele membre private*** ale unei clase. Funcțiile prietene ale unei clase trebuie precizate în definiția clasei. În acest sens, prototipurile unor astfel de funcții sunt precedate de cuvântul cheie **`friend`**. Spre deosebire de funcțiile membre, funcțiile prietene ale unei clase ***nu posedă pointerul implicit*** **`this`**. De aceea, deosebirea esențială două funcții care realizează aceleaşi prelucrări, o funcție membră şi o funcție prietenă, constă în faptul că funcția prietenă are un parametru în plus față de funcția membru.
O funcție poate fi în acelaşi timp funcție membră a unei clase şi funcție prietenă a altei clase. În exemplul următor, f1 este funcție membră a clasei *cls1* şi funcție prietenă a clasei *cls2*.

Exemplu:

```
class cls1{       //...
      int f1(int, char);                  // f1 - metodă a clasei cls1
      //...
};
class cls2{       //...
      friend int cls1::f1(int, char);     // f1 - prietenă a clasei cls2
      //...
};
```

În cazul în care se doreşte ca toate funcțiile membre ale unei clase să aibă acces la membrii privați ai altei clase (să fie funcții prietene), prima clasă poate fi declarată ***clasa prietenă*** pentru cea de-a doua clasă. În exemplul următor, clasa `cls1` este *clasa prietenă* a clasei `cls2`.

Exemplu:

```
class cls1;
class cls2{       //...
      friend cls1;                        // clasa cls1 este clasă prietenă a clasei cls2
            //...
};
```

Relația de clasa prietenă nu este tranzitivă. Astfel, dacă clasa `cls1` este clasa prietenă a clasei `cls2`, iar clasa `cls2` este clasă prietenă a clasei `cls3`, aceasta ***nu*** implică faptul că `cls1` *este clasă prietenă pentru* `cls3`.

ÎNTREBĂRI ŞI EXERCIȚII

Chestiuni teoretice

1. Când acționează constructorul unei clase?
2. Când acționează destructorul unei clase?
3. Când este absolut necesară definirea unui constructor de copiere?
4. Când se justifică utilizarea funcțiilor inline?
5. Caracteristicile destructorului unei clase.
6. Care este utilitatea moştenirii?
7. Care sunt deosebirile între o funcție membră a unei clase şi o functie prietenă a unei clase?
8. Ce fel de metode pot acționa asupra datelor membre statice ale unei clase?
9. Ce funcții au acces la membrii privați ai unei clase?
10. Ce observație aveți în legătură cu metodele definite în interiorul clasei şi funcțiile inline?
11. Ce operator permite referirea unui membru al structurii ?
12. Ce sunt clasele?
13. Ce sunt constructorii impliciți?
14. Ce sunt destructorii ?
15. Ce sunt funcțiile inline?
16. Ce este o metodă?

17. Constructorii unei clase pot primi ca parametri instanțe ale clasei respective? Dacă da, în ce condiții?
18. Ce sunt funcțiile prietene?
19. Cine impune comportamentul unui obiect?
20. Cum se alocă memoria pentru datele membre nestatice în momentul declarării mai multor obiecte din aceeasi clasă?
21. Cum se declară funcțiile prietene?
22. Deosebiri între stucturi şi clase.
23. Enumerati facilitatile oferite de programarea orientata obiect.
24. Explicați conceptul de încapsulare a datelor.
25. Explicați în câteva cuvinte ce este mostenirea multiplă.
26. Explicați în câteva cuvinte ce este moştenirea.
27. Niveluri de acces la membrii şi metodele unei clase.
28. O clasă poate avea mai mulți desctructori? Dacă da, în ce condiții?
29. O clasă poate fi prietenă a altei clase ? Dacă da, ce înseamnă acest lucru?
30. Operatorul **::** şi rolul său.
31. Prin ce se caracterizează datele membre statice?
32. Prin ce se realizează comunicarea între obiectele unei clase?
33. Prototipul constructorului de copiere.
34. Nici funcțiile prietene, nici metodele statice ale unei clase nu primesc ca argument implicit pointerul this. Explicați care sunt, totuşi, diferențele dintre ele.

Chestiuni practice

1. Completați tipul de date `complex`, cu funcțiile (membre sau prietene) care vor realiza următoarele operații: Adunarea unei date de tip complex cu o dată de tip real; scăderea a două date de tip complex; scăderea unui real dintr-un complex; împărțirea a două date de tip complex; înmulțirea unui real cu un complex; ridicarea la o putere întreagă a unei date de tip complex; compararea a două date de tip complex.
2. Să se scrie un program care citeşte câte două date de tip complex, până la întâlnirea perechii de date (z1=0+i*0, z2=0+i*0). Pentru fiecare pereche de date, să se afişeze suma, diferența, produsul şi câtul.
3. Să se scrie un program care citeşte datele a, b, c de tip complex, rezolvă şi afişează rădăcinile ecuației de gradul doi: $ax^2+bx+c=0$.
4. Care sunt greşelile din următoarele secvențe?

```
a) class ex1{
       char *nume;  int lungime;
       void init (char *s, int l)
            {strcpy(nume, s); lungime=l;}
   };
   ex1 A; A.init("teava", 20);
b) union numar{
       private:
            char exponent, mantisa[3];
       public:
            char exp();
       };
c) class complex{
       int real, imag;
       complex (int x, int y)
            {real=x; imag=y;}
};
void main()
{     complex Z(20, 30);      }
```