De la o PROBLEMĂ de CODIFICARE la elemente POO cu C++

Doina Hrinciuc-Logofătu

Ne propunem rezolvarea unei probleme de codificare folosind limbajul C++. Mai întâi vom realiza analiza problemei propuse si vom proiecta un algoritm de rezolvare. Vom continua cu implementarea algoritmului construit utilizând facilitățile oferite de limbajul C++ dintre care: abstractizarea datelor, supraîncărcarea operatorilor, polimorfism, funcții inline, biblioteci standard STL, modificatori de acces, utilizarea namespace-urilor, funcții const. Mai departe se va face analiza programului, în care se vor evidenția aspecte ale programării orientate obiect, dar și alte detalii de implementare.

Problema codificării complexe

Definiție

Dacă X și B sunt numere complexe cu părțile reală și imaginară întregi, iar secvența a_n, a_{n-1}, ..., a_1, a_0 conține numere naturale mai mici decât $|B|$ cu proprietatea că:

$X = a_0 + a_1 \cdot B + a_2 \cdot B^2 + ...+ a_n \cdot B^n$

atunci spunem că secvența a_n, a_{n-1}, ..., a_1, a_0 este o codificare pentru perechea (X, B).

Se poate spune că a_n, a_{n-1}, ..., a_1, a_0 sunt cifrele reprezentării lui X într-un sistem numeric cu baza B.

Scopul problemei este decodificarea pe baza perechii de numere complexe (X, B), adică determinarea "cifrelor" a_0, a_1, ..., a_{n-1}, a_n în sistemul de numerație cu baza B a reprezentării numărului complex X.

Date de intrare

Setul de date de intrare se află în fișierul **cod.in** și este compus din mai multe instanțe. Fiecare caz este descris pe o singură linie care conține patru numere întregi X_r, X_i, B_r, B_i ($|X_r|, |X_i| \le 1000000$, $|B_r|, |B_i| \le 16$). Aceste numere indică părțile reală, respectiv imaginară ale numerelor complexe X și B.

Cu alte cuvinte, avem $X = X_r + i \cdot X_i$, $B = B_r + i \cdot B_i$. B este baza sistemului ($|B| > 1$), iar X este numărul complex în care are loc reprezentarea codului.

Date de ieșire

Programul trebuie să scrie în fișierul de ieșire **cod.out** câte o singură linie pentru fiecare caz. Aceasta trebuie să conțină cifrele a_n, a_{n-1}, ..., a_1, a_0, separate prin câte o virgulă. În plus, trebuie să fie satisfăcute următoarele condiții:

- pentru $i \in \{0, 1, 2, ..., n\}$: $0 \le a_i < |B|$;
- $X = a_0 + a_1 \cdot B + a_2 \cdot B^2 + ...+ a_n \cdot B^n$;
- dacă $n > 0$, atunci $a_n \neq 0$;
- $n \le 100$.

Listing CODIFICARE.CPP

```
#include <cmath>
#include <fstream>
#include <vector>
#define MSG_ERR "Nu exista cod."
using namespace std;

// declararea clasei Complex
class Complex
{
  private:
    long _re,_im;
  public:
  Complex(){};
  Complex(long re,long im)
  {
    _re = re;
    _im = im;
  }
  ~Complex(){};
  double getModul() const;
  Complex operator-(Complex&);
  Complex operator-(long);
  Complex operator*(Complex&);
  Complex operator/(Complex&);
  friend int operator==
                  (Complex&,long);
  friend istream& operator>>
              (istream&,Complex&);
};
```

Dacă nu există numere care să satisfacă acest criteriu, atunci se va scrie în linia corespunzătoare din fișierul de ieșire mesajul `Nu exista cod.`

Exemplu

cod.in

```
0 1 1 1
191 -192 11 -12
0 0 1 1
46185 -80102 15 5
-262144 0 0 2
6222 -16809 -2 13
7973 -1108 2 2
3560 -1908 -16 4
28443 44463 -6 14
-11425 5080 4 -10
-8 -210 5 -6
```

cod.out

```
Nu exista cod.
16,15
0
Nu exista cod.
1,0,0,0,0,0,0,0,0,0,0,0,0,0,
                        0,0,0,0,0
8,11,7,3
2,0,2,1,1,2,1,2,1
15,3,8
Nu exista cod.
10,1,4,3
3,5,0
```

Analiza problemei și proiectarea soluției

Notăm cu C_Z mulțimea numerelor complexe cu părțile reală și imaginară întregi:

$$C_Z = \{re + i \cdot im / re, im \in Z\}$$

Ecuația #1

$$X = a_0 + a_1 \cdot B + a_2 \cdot B^2 + ... + a_n \cdot B^n$$
$$X = a_0 + B \cdot (a_1 + a_2 \cdot B + ... + a_n \cdot B^{n-1})$$
$$X - a_0 = B \cdot (a_1 + a_2 \cdot B + ... + a_n \cdot B^{n-1})$$

Deoarece numerele a_n, a_{n-1}, ..., a_1, a_0 sunt numere naturale și $B \in C_Z$, rezultă că există numerele întregi P și Q care satisfac condiția:

$$a_1 + a_2 \cdot B + ... + a_n \cdot B^{n-1} = P + i \cdot Q$$

Propoziție.

Dacă numărul a0 care satisface ecuația #1 și proprietatea $0 \le a_0 \le |B|$ există, atunci el este unic cu aceste proprietăți.

Demonstrație

Vom folosi metoda reducerii la absurd. Presupunem că $\exists\, a_0 \neq a_0'$ care satisfac cele două proprietăți.

În aceste condiții $\exists\, Q_0 \neq Q_0' \in C_Z$ astfel încât $X = a_0 + B \cdot Q_0$ și $X = a_0' + B \cdot Q_0'$. Efectuând diferența avem:

$$a_0 - a_0' = B \cdot (Q_0' - Q_0),$$

de unde rezultă:

$$|a_0 - a_0'| = |B| \cdot |(Q_0' - Q_0)|.$$

Datorită faptului că $a_0, a_0' \in \{0, 1, ..., |B| - 1\}$ rezultă $|a_0 - a_0'| < |B|$.

Deoarece $Q_0, Q_0' \in C_Z$ și sunt diferite rezultă $|(Q_0' - Q_0)| \ge 1$, deci avem:

$$|B| \cdot |(Q_0' - Q_0)| \ge |B|.$$

Obținem astfel:

$$|a_0 - a_0'| < |B| \cdot |(Q_0' - Q_0)|,$$

ceea ce contravine relației obținute anterior:

$$|a_0 - a_0'| = |B| \cdot |(Q_0' - Q_0)|.$$

În concluzie, presupunerea este falsă, deci există o unică valoare a_0 care satisface cele două proprietăți.

Prezentarea algoritmului

Pe baza propoziției anterioare, deducem că, dacă un cod pentru perechea (X, B) dată există, atunci acesta este unic. Algoritmul de rezolvare pe care îl propunem pentru această problemă este unul iterativ, în care se determină, pe rând, "cifrele" a_0, a_1, ..., a_{n-1}, a_n.

Pentru perechea (X, B) dată, determinarea valorii a_0 se realizează prin parcurgerea valorilor posibile 0, 1, 2, ..., $|B|$ - 1 și verificarea faptului dacă $X - a_0$ este ***multiplu*** al lui B, adică se poate scrie sub forma $B \cdot Q$, unde $Q \in C_Z$.

În caz afirmativ, această valoarea este unică și se adaugă în vectorul cu soluția parțială posibilă. Q devine noul X și se continuă în aceeași manieră până la o condiție de oprire, care poate fi:

- X devine zero (am determinat o codificare);
- în încercarea de a determina valoarea a_i nu am identificat nici o valoare posibilă dintre *0*, 1, 2, ..., $|B|$ - 1

```
// implementarea clasei Complex
26: inline double
          Complex::getModul() const
27: {
28:   double aux = _re*_re +
                            _im*_im;
29:   return sqrt(aux);
30: }

31: inline Complex
      Complex::operator-(Complex &z)
32: {
33:   return Complex(_re - z._re,
                      _im - z._im);
34: }

35: inline Complex
         Complex::operator-(long l)
36: {
37:   return Complex(_re - l, _im);
38: }

39: inline Complex
      Complex::operator*(Complex &z)
40: {
41:   return
42:     Complex(_re*z._re -
                         _im*z._im,
          _re*z._im + _im*z._re );
43: }

44: inline Complex
      Complex::operator/(Complex &z)
45: {
46:   long r = z._re;
47:   long i = z._im;
48:   long m2 = r*r + i*i;
49:   return
50:     Complex((_re*r + _im*i) /
        m2, (_im*r - _re*i) / m2);
51: }

// implementare functii prietene
52: inline int operator==
               (Complex &z, long l)
53: {
54:   return (z._re == l
                   && z._im == 0);
55: }

56: istream& operator>>
          (istream &is, Complex &u)
57: {
58:   char *s1, *s2;
59:   s1 = new char[100];
60:   s2 = new char[100];
61:   is >> s1 >> s2;
62:   u._re =  atol(s1);
63:   u._im =  atol(s2);
64:   return is;
65: }

// metode locale
66: inline Complex operator%
         (Complex &z1, Complex &z2)
67: {
68:   Complex z;
69:   z = z1 - (z2 * (z1 / z2));
```

cu proprietatea că $X - a_i$ este *multiplu* al valorii B (nu există codificare);
- am construit $a_0, a_1, ..., a_{100}$, și X nu este 0 (nu există codificare deoarece problema impune condiția ca n să fie cel mult egal cu 100).

Verificarea multiplului

În continuare vom analiza cum se poate decide dacă un număr complex Z este multiplu al numărului complex nenul B în mulțimea C_Z.

Presupunem că $Z, B \in C_Z, B \neq 0$. Z este multiplu al lui B în C_Z dacă și numai dacă $\exists\, Q \in C_Z$ astfel încât

$$Z = B \cdot Q \Leftrightarrow \frac{Z}{B} = Q \Leftrightarrow$$

$$\Leftrightarrow \frac{Z_r + iZ_i}{B_r + iB_i} = Q \Leftrightarrow$$

$$\Leftrightarrow \frac{(Z_r + iZ_i)\cdot(B_r - iB_i)}{B_r^2 + B_i^2} = Q \Leftrightarrow$$

$$\Leftrightarrow \frac{Z_r \cdot B_r + Z_i \cdot B_i}{B_r^2 + B_i^2} + i\frac{Z_i \cdot B_r - Z_r \cdot B_i}{B_r^2 + B_i^2} = Q.$$

Această ultimă egalitate este echivalentă cu faptul că numerele întregi $Z_r \cdot B_r + Z_i \cdot B_i$ și $Z_i \cdot B_r - Z_r \cdot B_i$ sunt divizibile cu $B_r^2 + B_i^2$.

Algoritmul în pseudocod

```
citește X, B
V ← {}
dacă X = 0 atunci
  V ← {0}
sfârșit dacă
pentru j ← 0, 100 execută
  gasit ← fals
  pentru j ← 0, |B| execută
  ZAux ← X - i
  dacă ZAux este multiplu al
                        lui B atunci
    adaugă i la vectorul V
    X ← ZAux / B
    gasit ← adevărat
    întrerupe ciclul pentru
  sfârșit dacă
  sfârșit pentru
  dacă nu gasit atunci
    scrie "Nu exista cod."
    întrerupe ciclul pentru
  sfârșit dacă
  dacă X = 0 atunci
    întrerupe ciclul pentru
  sfârșit dacă
sfârșit pentru
dacă X = 0 atunci
  scrie V
altfel
  dacă j = 100 atunci
    scrie "Nu exista cod."
  sfârșit dacă
sfârșit dacă
```

Programul care implementează acest algoritm în C++ este prezentat în casetele laterale din paginile acestui articol.

Implementarea ține cont de modelul matematic descris anterior și programul este implementat cât mai aproape de acesta, utilizând facilitățile puse la dispoziție de limbajul orientat pe obiecte C++. Liniile programului au fost numerotate pentru a putea analiza ulterior detaliile de implementare.

Analiza programului

În această secțiune vom analiza implementarea aleasă pentru algoritmul prezentat.

Utilizarea STL

O mare varietate de funcții din biblioteca standard ***STL*** (***S**tandard **T**emplate **L**ibrary*) pot fi apelate de programele C++. Aceste funcții oferă servicii de bază cum ar fi *input* și *output* și furnizează implementări eficiente ale operațiilor frecvent utilizate (de exemplu operațiile uzuale cu vectori, liste, cozi, stive, sortări, operații cu permutări etc.).

Liniile 1 - 3 includ librăriile *STL* folosite în program: `cmath` (definește macrourile definite în antetul *Standard C* `math.h`, necesare pentru apelul metodei `sqrt()`), `fstream` (definește câteva clase pentru operații care suportă fluxuri de intrare/ieșire pentru secvențe stocate în fișiere exterioare, în cazul nostru folosim `ifstream`, `ofstream`) și `vector` (definește șablonul *container* pentru clasa `vector` și câteva șabloane pe care acesta le suportă).

Toate metodele *STL* sunt definite în spațiul de nume `std`, motiv pentru care atunci când sunt utlilizate ar trebui să se scrie: `std::vector`, `std::ifstream`, `std::sqrt` etc.

```
 70:   return z;
 71: }

 72: inline int operator!=
                (Complex &z, long l)
 73: {
 74:   return (!(z == l));
 75: }

 76: void WriteSolution
      (vector<int> V, ofstream &out)
 77: {
 78:   vector<int>::reverse_iterator
                  itEnd = V.rend();
 79:   vector<int>::reverse_iterator
                                 it;
 80:   for (it = V.rbegin();
            it != (itEnd-1); it++ )
 81:     out << *it << ",";
 82:   out << *it;
 83:   out << endl;
 84: }

 85: void Process(Complex &X,
        Complex &B, ofstream& out)
 86: {
 87:   vector<int> V;
 88:   Complex ZAux;
 89:   double temp = B.getModul();
 90:   int aMax = (int) temp;
 91:   if (temp - aMax == 0) aMax--;
 92:   if (X == 0) {V.push_back(0);}
 93:   for(short j=0; (j<100) &&
                    (X != 0); j++)
 94:   {
 95:     bool gasit = false;
 96:     for (int i=0; i<=aMax; i++)
 97:     {
 98:       ZAux = X - i;
 99:       if (ZAux % B == 0)
100:       {
101:         V.push_back(i);
102:         X = ZAux / B;
103:         gasit = true;
104:         break;
105:       }
106:     }
107:     if (!gasit){
             out << MSG_ERR << endl;
                              break;}
108:   }
109:   if (X == 0){
            WriteSolution(V, out);}
110:   else if (j == 100){
          out << MSG_ERR << endl;}
111: }

// Program principal
112: void main(){
113:   Complex X ,B;
114:   ifstream in("cod.in");
115:   ofstream out("cod.out");
116:   while (in && !in.eof()){
117:     in >> X >> B;
118:     Process( X, B, out );
119:   }
120: }
```

O altă alternativă, atunci când nu există posibilitatea de ambiguități, este indicarea utilizării unui anumit spațiu de nume în cadrul unui bloc prin directiva `using namespace`, (linia 5 - prin eliminarea acestei linii va trebui să adăugăm tuturor elementelor *STL* prefixul `std::`).

Abstractizarea datelor - concept de bază al Programării Orientate Obiect.

Pentru a rămâne la nivelul modelului matematic descris în analiza problemei, definim clasa `Complex`, care permite manipularea într-un mod natural a numerelor complexe cu părțile reală și imaginară întregi (C_Z).

Definirea unui tip abstract de date implică specificarea reprezentării interne a datelor din acest tip, precum și a funcțiilor pe care alte module de program le vor utiliza pentru manipularea elementelor constituente.

Se realizează un proces foarte important de ascundere a datelor: limitarea posibilităților de acces și modificare a acestora dacă nu este necesar. Nu putem modifica sau accesa numai partea reală sau numai partea imaginară (s-ar putea scrie metode pentru acest scop, în caz că ar fi necesar, de exemplu *get*-eri și *set*-eri `getRe()`, `getIm()`, `setRe()`, `setIm()`).

Liniile 6-25 conțin declararea clasei `Complex`, în cadrul căreia atributele `_re` și `_im` sunt private (nu pot fi accesate pentru citire sau modificare ca atare). Clasa furnizează o serie de metode publice și înregistrează două funcții externe ca fiind funcții prietene, una din metodele oferite de *C++* pentru supraîncărcarea operatorilor.

Constructori și destructori

Liniile 11-16 conțin declararea și definirea a doi constructori pentru clasa `Complex`. Un costructor este o funcție care are numele clasei căreia îi aparține.

Constructorii se apelează automat atunci când se creează obiecte noi ale unei clase. Dacă o clasă nu are nici un constructor declarat în mod explicit, compilatorul va crea în mod automat unul. Acesta nu va avea parametri și va avea o listă vidă de instrucțiuni (ca primul constructor din program).

Apelul primului constructor în cazul programului de mai sus se realizează, de exemplu, în liniile 88 și 113, precum și în implementări ale funcțiilor de supraîncărcare ale operatorilor. Utilizarea celui de al doilea constructor este întâlnită în liniile 33, 37, 42, 50.

Destructorul realizează operația inversă creării. De regulă aceasta înseamnă eliberarea memoriei ocupate de obiect. Numele destructorului se fomează din numele clasei, prefixat de caracterul tilda (~), ca în linia 17. Destructorii se apelează automat atunci când urmează ca obiectele să dispară; de exemplu când se iese dintr-un bloc în care obiectele au fost create, dar pot apărea și situații în care destructorii să fie apelați explicit.

Supraîncărcarea operatorilor, funcții prietene

Limbajul *C++* nu furnizează operațiile standard uzuale de manipulare a numerelor complexe, cum ar fi adunarea, scăderea, înmulțirea sau citirea acestora, dar furnizează modalități care pot fi folosite pentru a stabili o anumită semnificație pentru operatorii standard atunci când aceștia sunt aplicați obiectelor unor clase specifice.

Regulile pentru supraîncărcarea operatorilor sunt următoarele:

- Pot fi supraîncărcați doar următorii operatori:

```
+ - * / % ^ ! = < > += -=
^= &= |= << >> <<= <= >=
&& || ++ -- ( ) [ ] new
delete & | ~ *= /= %= >>=
== != , -> ->*
```

- Dacă un operator poate fi folosit atât ca operator unar, cât și ca operator binar, atunci pot fi supraîncărcate ambele moduri de folosire.
- Un operator se poate supraîncărca utilizând atât o funcție membru nestatică, cât și folosind o funcție globală, care este sau nu funcție prietenă a clasei. O funcție globală trebuie să aibă cel puțin un parametru care este de tipul clasei sau o referință la tipul clasei.
- Dacă un operator unar este supraîncărcat folosind o funcție membru, aceasta nu ia nici un argument. Dacă este supraîncărcat folosind o funcție globală, aceasta ia un singur argument.
- Dacă este supraîncărcat un operator binar folosind o funcție membru, aceasta are un singur argument. Dacă este supraîncărcat folosind o funcție globală, aceasta are două argumente.

Liniile 19-22 conțin supraîncărcarea operatorilor -, *, / declarați ca funcții membru ale clasei.

Liniile 23-24 conțin supraîncărcarea operatorilor ==, >> ca funcții prietene ale clasei (acestea nu sunt funcții membru, ci funcții globale). Prin utilizarea cuvântului cheie **`friend`** se oferă acestor funcții accesul la membrii privați ai clasei (în cazul nostru ele vor accesa în mod excepțional membrii `_re` și `_im` ai obiectelor de tip `Complex`, așa cum se remarcă în implementările lor).

Liniile 66-71 și 72-75 conțin supraîncărcarea operatorilor %, și != ca metode locale obișnuite.

Implementarea operatorilor de mai sus ține cont de modelul matematic obișnuit al numerelor complexe, dar și de restricțiile și observațiile ce reies din partea de analiză a problemei, având în vedere faptul că părțile reală și imaginară sunt ambele numere întregi.

În liniile 92, 93, 99 și 109 remarcăm utlizarea operatorilor == și !=, prin care se compară un număr complex definit de noi și un număr întreg, lucru posibil prin implementarea ca atare a acestor operatori (liniile 23 și 52-55, respectiv 72-75).

În linia 98 se utilizează operatorul - (liniile 20 și 35-38) care execută operația de scădere dintre un număr complex din C_Z și un număr întreg, returnând un număr complex.

Linia 99 exemplifică supraîncărcarea operatorului % (ca funcție globală neprietenă, implementată în liniile 66-71).

Linia 102 execută operația de împărțire între două numere com-

plexe, returnând rezultatul ca număr complex (implementarea se bazează pe modelul matematic prezentat în demonstrația proprietății de unicitate din partea de analiză a problemei, liniile 44-51). De fapt instrucțiunea `X = XNou / B;` este echivalentă cu `X = XNou.operator/(B);` (prin înlocuire, programul va funcționa la fel) și acest fapt este valabil și pentru ceilalți operatori.

Linia 117 execută citirea într-un mod natural a numerelor complexe *X* și *B*, lucru imposibil fără implementarea operatorului >> (liniile 24 și 56-64).

Polimorfism

Termenul *polimorf* provine din limba greacă și înseamnă în traducere *a avea mai multe forme*. În cazul nostru supraîncărcarea operatorilor este o formă de polimorfism (parametric și de tip), pentru că aceleași operatori cunoscuți (deja implementați în C++ standard pentru tipurile de bază) li se dau noi înțelesuri: de exemplu operatorul minus va primi în plus înțelesul de diferență a două numere complexe sau dintre un număr complex și un număr întreg de tip **long**.

Funcții membru const

Limbajul C++ oferă posibilitatea de a stabili că anumite funcții ale unui obiect sunt constante, prin folosirea modificatorului **const**.

Aceasta înseamnă că respectivele funcții nu schimbă starea (datele) obiectului atunci când sunt apelate. În plus, ele pot fi apelate atât de obiecte constante, cât și de obiecte neconstante.

De obicei, dacă o funcție nu modifică starea obiectului, ea ar trebui declarată prin definiție **const**. Observăm utilizarea acestui cuvând cheie în liniile 18 și 26-30 pentru funcția `getModul()` care nu modifică atributele `_re` și `_im` ale obiectului.

Funcțiile inline

Ați observat probabil deja utilizarea cuvântului cheie **inline** în declararea unor metode pentru supraîncărcarea operatorilor.

O funcție **inline** este o combinație între macro-uri și funcții. La fel ca și în cazul macro-urilor, apelul unei funcții **inline** este expandat prin înlocuirea apelului său cu corpul corespunzător implementării funcției.

Spre deosebire de macro-uri, se execută verificarea tipurilor parametrilor, transmiși ca și în cazul unei funcții normale.

Un dezavantaj este necesarul de memorie, funcțiile **inline** fiind eficient folosite atunci când au un număr mic de instrucțiuni (nu mai mult de cinci). De asemenea, nu se pot utiliza pointeri la funcții **inline**.

inline este o cerere adresată compilatorului, care poate să fie îndeplinită sau nu; dacă cererea nu poate fi îndeplinită, compilatorul generează o funcție obișnuită.

Utilizarea std::vector

Tipul `std::vector` din *STL* este asemănător cu un tablou și permite stocarea elementelor de diferite tipuri. În cadrul programului folosim un astfel de vector pentru stocarea valorilor a_i. Observăm declararea vectorului `V` în linia 87, între parantezele unghiulare se specifică tipul elementelor sale (în cazul nostru **int**).

Pentru a adăuga un element la sfârșitul unui vector folosim metoda `push_back()`, așa cum se poate vedea în liniile 92 și 101.

Pentru a scrie un vector de numere întregi într-un flux (*stream*) de ieșire utilizăm metoda `WriteSolution(vector<int>, ofstream &)` de la liniile 76-84. În cadrul acestei metode folosim conceptul de iterator pentru accesarea iterativă a elementelor *container*-ului (vectorului). Iteratorii pot fi incrementați, decrementați, dereferențiați etc. STL ne pune la dispoziție, pe lângă interatorul obișnuit, și noțiunea de `reverse_iterator` utlilizată împreună cu metodele `rBegin()` (*reverse Begin*) și `rEnd()` (*reverse End*) pentru furnizarea primului și utlimului element din vector în ordine inversată. Folosind iteratorul obișnuit, funcția mai poate fi scrisă sub forma:

```
void WriteSolution(vector<int>
           V, ofstream &out){
  vector<int>::iterator
         itBegin = V.begin();
  vector<int>::iterator it;
  for (it = V.end() - 1;
       it != itBegin; it-- )
    out << *it << ",";
  out << *it << endl;
}
```

Elementele unui vector *STL* pot fi accesate folosind și operatorul obișnuit `[]`, funcția `WriteSolution()` se mai poate scrie astfel:

```
void WriteSolution(vector<int>
           V, ofstream &out){
  for (short i = (short)
      V.size() - 1; i>0; i--)
    out << V[i] << ",";
  if (V.size() >0)
    out << V[0];
  out << endl;
}
```

Exerciții propuse

- Scrieți un program *Pascal* sau *C* care implementează algoritmul, pentru a putea compara programarea procedurală și cea orientată obiect.
- Implementați în aceeași manieră ca și clasa `Complex` clasele `Fractie`, `Matrice`, `Polinom`, care să modeleze noțiunile matematice corespunzătoare. Scrieți programe demonstrative care să testeze metodele implementate.

Bibliografie

1. Hrinciuc-Logofătu, D., *C++. Probleme rezolvate și algoritmi*, Editura Polirom, Iași, 2001
2. Ivașc, C., Prună, M., Hrinciuc Logofătu, D., Condurache, L., *Informatică. Manual pentru clasa a XI-a*, Editura Petrion, București, 2001
3. *MSDN Library - Visual Studio .NET 2003*
4. *Standard Template Library Programmer's Guide*, http://www.sgi.com/tech/stl/
5. Stroustroup, B., *The C++ Programming Language, Special Edition*, Editura Addison-Wesley, Boston, 2000