Capitolul 2

Extensii ale limbajului C în limbajul C++

Limbajul C++ este un supraset al limbajului C şi a fost proiectat de Bjarne Stroustrup. El oferă suport atât pentru programarea procedurală şi pentru abstractizarea datelor, dar scopul principal este suportul oferit pentru programarea orientată pe obiecte.

Extensiile limbajului C se referă la două aspecte :

- adăugarea unor facilități care nu sunt legate direct de programarea orientată pe obiecte ;
- adăugarea elementelor de bază pentru a oferi suport programării orientate pe obiecte.

În primul caz este vorba despre elemente precum : tipul referință, substituția in-line a funcțiilor, etc., pe când în cazul al doilea este vorba despre elemente precum noțiunea de clasă, moştenire, polimorfism, etc.

În acest capitol se vor discuta doar aspectele legate de extensiile limbajului C ce nu privesc programarea orientată pe obiecte.

2.1 Noi tipuri de date

Limbajul C++ extinde tipurile de date fundamentale ale limbajului C cu încă două tipuri de date: `bool` şi `wchar_t`. În plus, el permite utilizarea încă a unui tip numit `string`, pentru care a fost creata o clasă `string`. Declarațiile acestei clase se află în fişierul `string.h`. Despre clasa `string` se va discuta într-un capitol ulterior.

A) Tipul `bool` reprezintă valorile logice (booleene), pentru care se utilizează constantele predefinite `true` şi `false`. Există o asemănare din acest punct de vedere cu limbajul Pascal, care are tipul predefinit `Boolean`, precum şi cu Java, care are tipul `boolean`.

Spre deosebire de Pascal însă, există o compatibilitate între tipul bool şi tipurile întregi de date. Astfel, variabilele de tipul `bool` pot primi valori de tip întreg, deoarece compilatorul efectuează în mod automat o conversie de tipul întreg respectiv la tipul `bool`. De exemplu, pentru secvența :

```
bool boolVar ;
int intVar ;
// ...
boolVar = intVar ;
```

se generează o instrucțiue echivalentă de forma :

```
boolVar = intVar ? true : false ;
```

În mod asemănător, valorile de tip bool pot fi utilizate în locul valorilor de tip întreg, compilatorul realizând o conversie automată de forma:

```
intVal = boolVal ? 1 : 0 ;
```

În acest mod există o compatibilitate cu regulile C de evaluare a expresiilor condiționale din instrucțiuni precum instrucțiunile repetitive şi instrucțiunea `if`.

Avantajul utilizării tipului `bool` constă în faptul că permite scrierea unui cod mai general şi mai intuitiv decât utilizarea tipului `int`. De exemplu, pentru funcția cu prototipul următor :

```
bool Apartine(double x, double a, double b) ;
```

se poate şti faptul că ea returnează o valoare logică, pe când un prototip de forma :

```
int Apartine(double x, double a, double b) ;
```

nu oferă o asemenea certitudine.

B) **Tipul `wchar_t`** (`wide character`) este o extensie a tipului `char`, care permite utilizarea mulțimilor de caractere reprezentate intern pe doi octeți (cum este `Unicode`). Pentru acest tip, `sizeof(wchar_t)` are valoarea 2, ceea ce permite utilizarea a peste 64000 de caractere diferite.

2.2 Declararea variabilelor şi spațiul numelor

Spre deosebire de limbajul C, în care declarațiile locale trebuie făcute doar la începutul unui bloc înainte de începutul instrucțiunilor, în limbajul C++ declarațiile locale pot fi făcute oriunde în cadrul unui bloc, acolo unde sunt permise instrucțiuni. Domeniul de referință al unor astfel de variabile declarate local reprezintă o parte din blocul în care au fost declarate, începând din linia declarației respective şi până la sfârşitul blocului curent.

Exemplu.

```
void Prelucrare() {
  int k = 5 ;       // incepe domeniul variabilei k
  // ...
  k = k + 3 ;
  // ...
  float x = 7 ;     // incepe domeniul variabilei x
  // ...
  // incepe domeniul variabilei i
  for (int i=0 ; i<k ; i++)
    printf("%d", i) ;
  // ...
  // se termina domeniul variabilelor k, x si i
}
```

O altă problemă a limbajului C se referă la ***spațiul numelor***, ceea ce face dificilă scrierea şi testarea programelor mari de către mai multe echipe de programatori.

Execuția unui program C se face în afara spațiului numelor, ceea ce înseamnă că toate variabilele utilizate în cadrul diferitelor module ale unui program se raportează la întregul program. Astfel, variabilele cu acelaşi nume declarate în module diferite ale unui program accesează aceeaşi zonă de memorie şi reprezintă aceleaşi variabile. O variantă de soluționare

a acestei probleme în C este de a declara variabilele `static` în fişierele respective şi a le ascunde astfel în exterior.

Limbajul C++ ataşează variabilele la un *spațiu al numelor*, ceea ce permite ca variabile cu acelaşi nume aflate în module diferite să reprezinte variabile distincte.

Toate variabilele declarate în cadrul bibliotecilor standard ale limbajului C++ au un spațiu al numelor predefinit, notat cu `std`. Pentru a utiliza un spațiu al numelor diferit de cel curent din cadrul unui program sau modul de program se utilizează directiva `using` :

```
using namespace std ;
```

De exemplu, pentru a utiliza funcțiile standard ce lucrează cu şirurile de caractere declarate în fişierul `string` se poate scrie astfel :

```
#include <string>
```

Observatie. Fişierele header nu mai au extensia `.h` ca în limbajul C, dar pentru a se păstra compatibilitatea cu programele scrise în C se poate utiliza în continuare aceeaşi sintaxă. De exemplu, directiva:

```
#include <string.h>
```

este echivalentă cu :

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

2.3 Referințe

Unul dintre marile dezavantaje ale limbajului C față de alte limbaje apropiate precum Pascal sau Ada îl constituie modul de transmitere al parametrilor la apelul funcțiilor. Limbajul C permite doar transferul prin valoare, ceea ce face necesară utilizarea pointerilor în cazul în care o funcție modifică valoarea unui anumit parametru.

Limbajul C++ adaugă în plus noțiunea de ***referință***. O referință este un nume alternativ (un alias) pentru o variabilă. Un asemenea tip de date este un tip derivat, care se obține dîntr-un tip de bază cu ajutorul operatorului &. dacă T este un tip de date, notația T& reprezintă tipul referință derivat din T, adica mulțimea tuturor elementelor referință la tipul T.

Valorilor elementelor de tip referință sunt asemănătoare pointerilor, în sensul ca o referință are ca valoare adresa de memorie a unei variabile ce aparține tipului de bază. Există însă câteva deosebiri importante între pointeri şi referințe :

a) O referință trebuie întotdeauna inițializată la declarare. De exemplu, declarația :

```
int k ;
int &r = k ;
```

declară variabila referință *r* ca având o valoare egală cu adresa de memorie a variabilei *k*. Aceasta inițializare se deosebeşte de atribuire, ea asigurând doar faptul că variabila de bază (k) are un nou nume (r). În cadrul blocului unde au fost declarate cele două variabile se poate folosi atat numele *k* cât şi *r* pentru a referi acelaşi obiect.

b) In momentul utilizării într-un program, referințele sunt în mod automat dereferite, nemaifiind nevoie de operatorul * pentru dereferire. Exemplu :

```
int k = 5 ;
int &r = k, *p ;
p = &k ;
r = r + 1 ;         // aceasta inseamna k = k + 1
*p = *p + 1 ;
```

Cele două observații anterioare impun câteva precizari :

- valoarea unei referințe nu poate fi modificată dupa inițializare, ea referind întotdeauna acelaşi obiect cu care a fost inițializată. De exemplu, instrucțiunea `r=r+1;` nu inseamnă că se adaugă 1 la valoarea lui *r*, ci la obiectul referit de *r*.
- operatorii nu acționează asupra referinței, ci asupra variabilei la care aceasta face referire

Principala utilizare însă a referințelor o constituie transferul parametrilor funcțiilor. În acest caz un parametru formal de tip referință reprezintă un alt nume pentru parametrul actual corespunzător în cazul unui apel. Orice modificare a valorii parametrului formal înseamnă de fapt modificarea valorii parametrului actual respectiv.

Exemplul 2.1. Interschimbarea a două valori. Funcția *Interschimbare1* utilizează pointeri, pe când *Interschimbare2* utilizează referințe.

```
void Interschimbare1(int *a, int *b) {
  int c = *a ;
  *a = *b ;
  *b = c ;
}
void Interschimbare2(int &a, int &b) {
  int c = a ;
  a = b ;
  b = c ;
}
void Prelucrare() {
  int x = 7, y = 5 ;
  Interschimbare1(&x, &y) ;
  printf(″%d%d″, x, y) ;
  x = 7 ; y = 5 ;
  Interschimbare2(x, y) ;
  printf(″%d%d″, x, y) ;
}
```

În momentul apelului, parametrii formali de tip referință se inițializează inițializează la adresa parametrilor actuali corespunzatori, ceea ce impune ca aceşti parametri actuali să fie nume de variabile cu un tip de date egal cu tipul de bază al referințeilor corespunzatoare şi nu adrese de memorie.

2.4 Funcții in-line

În cazul funcțiilor mici, cu un număr restrâns de instrucțiuni, mecanismul de apel (crearea cadrelor pe stivă, transferul parametrilor, etc.) poate deveni semnificativ în raport cu timpul

de execuţie al funcţiei, ceea ce duce la mărirea timpului de execuţie al programului şi scăderea eficienţei acestuia.

O altenativă o poate constitui folosirea macrodefiniţiilor, care sunt substituite de compilator în etapa de preprocesare.

Exemplulu 2.2. În secvenţa următoare se utilizează o macrodefiniţie pentru determinarea minimului a două valori.

```
#define minim(a, b) ((a < b) ? a : b)
void Prelucrare()
{
  int x = 7, y = 5, z ;
  z = minim(x, y) ;
  // ...
}
```

Limbajul C++ oferă în plus posibilitatea expandării ***inline*** a funcţiilor. Expandarea in-line înseamnă generarea de către compilator a codului corespunzător funcţiei, fără să se mai genereze o secvenţă de apel.

Declararea unei funcţii inline se face prin specificarea cuvantului cheie `inline` înaintea definiţiei acesteia, iar pentru funcţiile membre ale unei clase prin includerea corpului funcţiei în cadrul declaraţiei clasei .

Exemplul 2.3. Funcţia inline *minim* este echivalentă cu macrodefiniţia anterioară :

```
inline int minim(int a, int b)
{
  return ((a < b) ? a : b) ;
}
```

În cazul funcţiilor inline, compilatorul *încearcă* să plaseze o instanţă a funcţiei inline în acelaşi segment de cod ca şi cel al funcţiei apelate, dar nu se garantează însă acest lucru. Pentru funcţiile complicate (care conţin instrucţiuni repetitive sau care sunt recursive) nu se realizează mecanismul inline.

În general, utilizarea funcţiilor inline este mai eficientă decât a funcţiilor obişnuite, dar mai puţin eficientă decât utilizarea macrodefiniţiilor.

Exemplul 2.4. Pentru exemplificare, se va utiliza aceeaşi operaţie de ridicare la putere prin trei metode diferite : macrodefiniţie, funcţie inline şi funcţie uzuală. Se va prelua timpul sistem pentru fiecare din cele trei funcţii (f1, f2, f3), astfel încât să se determine timpul de execuţia al fiecărei funcţii.

```
#include <time.h>
#define SQR(x) (x)*(x)
inline float Sqr (float x)
  { return ( x*x ); }
float sqr (float x)
```

```
  { return ( x*x ); }
void f1() {
  float x = 2.5, y;
  for (long k=0; k<10000000; k++)
    k = SQR(x);
}
void f2() {
  float x = 2.5, y;
  for (long k=0; k<10000000; k++)
    k = Sqr(x);
}
void f3() {
  float x = 2.5, y;
  for (long k=0; k<10000000; k++)
    k = sqr(x);
}
void main() {
  time_t ltime;
  time(&ltime);
  printf("Timpul in secunde:\t%ld\n", ltime);
  f1();
  time(&ltime);
  printf("Timpul in secunde:\t%ld\n", ltime);
  f2();
  time(&ltime);
  printf("Timpul in secunde:\t%ld\n", ltime);
  f3();
  time(&ltime);
  printf("Timpul in secunde:\t%ld\n", ltime);
}
```

2.5 Argumente implicite ale funcțiilor

În mod uzual, o regulă de bază pentru multe dintre limbajele de programare impune ca la definiția şi la apelul unei funcții să fie utilizat acelaşi număr de parametri. Limbajul C permite definirea (destul de dificilă) a unor funcții cu număr variabil de parametri, cu ajutorul operatorului '**…' .** Sarcina de tratare a parametrilor revine total programatorilor, compilatorul neefectuând nici o verificare.

Limbajul C++ oferă în plus față de limbajul C o modalitate mai simplă şi mai eficientă de tratare a funcțiilor cu un număr variabil de parametri : este vorba despre ***funcții cu valori implicite pentru parametri***.

Un parametru cu valoare implicită este declarat în mod obişnuit în antetul unei funcții prin nume şi tip de date, dar în plus el este inițializat direct în antet. Dacă un apel al funcției respective conține un parametru actual cu o altă valoare decât cea specificată la inițializare, valoarea actuală este folosită ca valoare de inţializare ; dacă însă parametrul actual corepunzator lipseşte, se consideră ca valoare de inițializare valoarea specificată în antet.

Exemplul 2.5. Funcția *Distanta* poate determina atât distanța între două puncte în plan, cât şi între un punct şi origine.

```
double Distanta(doube x, double y,
          double x0 = 0, double y0 = 0)
{
  return sqrt((x-x0)*(x-x0)+(y-y0)*(y-y0)) ;
}

void Prelucrare() {
  double x1 = 3, y1 = 5, x2 = 4, y2 = 6, d1, d2 ;
  // distanta intre (x1,y1) si origine
  d1 = Distanta(x1, y1);
  // distanta intre (x1, y1) si (x2, y2)
  d2 = Distanta(x1, y1, x2, y2);
  // ...
}
```

Observații :

a) un parametru implicit poate fi inițializat doar cu o expresie constantă, care se poate evalua la compilare ;
b) se pot specifica mai mulți parametri impliciți în cadrul unei funcții, însă în acest caz ei trebuie să ocupe ultimele poziții, pentru că altfel nu se pot determina parametrii actuali corespunzători în cazul unui apel.

O problemă poate să apară în cazul funcțiilor care sunt atat declarate cât şi definite, pentru că antetul unei asemenea funcții poate să apară de mai multe ori în cadrul programului. În acest caz, limbajul C++ impune ca valorile de inițializare să fie specificate o singură dată, ori la definire, ori la declarare. O soluție care păstrează consecvența unui stil modular de programare este cea în care valorile implicite sunt specificate în prototipul funcțiilor din cadrul fişierului header corespunzator, dacă acesta există.

2.6 Funcții supraâncarcate

Supraâncarcarea numelui funcțiilor înseamnă de fapt posibilitatea existenței mai multor funcții cu acelaşi nume care efctuează operații diferite. Un exemplu de supraâncarcare există în limbajul Pascal, unde anumiți operatori se folosesc pentru operații diferite. De exemplu, operatorul + este utilizat atât pentru operația de adunare a numerelor, cât şi pentru operația de reuniune a mulțimilor, precum şi pentru cea de concatenare a şirurilor de caractere.

Limbajul C++ permite însă în plus definirea unor funcții utilizator supraâncarcate.

Exemplul 2.6. Se definesc mai multe funcții cu numele *add* :

```
int add(int a, int b) { return a + b ; }

double add(double a, double b) { return a + b ; }

char* add(char *a, char *b) { strcat(a, b) ; return a ; }
```

```
struct complex { double re ; double im ; } ;
complex add(complex a, complex b) {
  complex c ;
  c.re = a.re + b.re ;
  c.im = a.im + b.im ;
  return c ;
}

void Prelucrare() {
  int k = add(5, 1) ;
  double s = add(1.5, 8.4) ;
  char *s1 = ″abc″, *s2 = ″xyz″, *s3 = add(s1, s2) ;
  // ...
}
```

Compilatorul determină funcția efectivă ce va fi apelată în funcție de tipul parametrilor şi de numărul acestora.

Observații.

a) Pentru a defini două funcții supraîncarcate diferite, trebuie să difere numărul parametrilor sau cel putin tipul de date al unuia dintre parametri.
b) Deoarece nu se verifică şi tipul valorii returnate, două funcții supraîncarcate nu pot diferi doar prin tipul valoarii returnate.

2.7 Operatori de gestiune a memoriei

În cazul utilizării obiectelor dinamice în cadrul limbajului C, crearea şi distrugerea acestora înseamnă de fapt alocarea şi dealocarea memoriei pentru acestea. În mod uzual se folosesc funcții standard, precum `malloc` şi `free`.

Limbajul C++ posedă în plus doi operatori reprezentați prin cuvintele cheie `new` şi `delete`. Sintaxa uzuală de folosire este urmatoarea :

```
<nume pointer> = new [(] <tip> [)] ;
delete <nume pointer> ;
```

Exemple :

```
double *p = new double ;
double *p = new(double) ;
```

Se observă superioritatea operatorului `new` față de funcția `malloc`, prin faptul că el poate determina singur cantitatea de memorie ce trebuie alocată, precum şi tipul pointerului ce se returnează .

Opertatorul `new` se poate utiliza şi la alocarea memoriei pentru elemente compuse. În cazul tablourilor, trebuie specificat numărul de componente ce se doreste alocat.

Exemple:

```
struct punct { double x, y ; } ;
```

```
//se aloca o structura cu 2 componente
struct punct *p = new struct punct(2, 4) ;
//se aloca un tablou cu 10 componente
double *q = new double[10] ;
//se aloca un tablou de 10 pointeri la int
int **r = new int*[10] ;
```

Observații.

a) Ca şi funcţia `malloc`, operatorul `new` alocă memorie în zona heap a programului.
b) În cazul în care tipul de date este o clasă de obiecte, operatorul apelează în mod implicit constructorul clasei.

Complementarul lui `new` este `delete`. Acțiunea efectuată de acesta este asemănătoare cu a funcției `free`, eliberând zona de memorie spre care indică un anumit pointer. În plus față de `free`, el oferă protecție în cazul încercării eliberării memoriei pentru un pointer NULL.

Utilizarea lui `delete` are un dezavantaj în cazul în care se încearca dealocarea unui tablou. De exemplu :

```
int *p = new int[10];
delete p ;
```

În acest caz nu se eliberează efectiv decât zona ocupată de primul element al tabloului. Pentru a se elibera întreaga zona, trebuie specificat numărul de componente ale tabloului care trebuie eliberate. Acest număr se specifică la sfarsitul cuvantului `delete` între paranteze drepte, ca în exemplul urmator :

```
delete[10] p ;
```

Pentru a evita eventualele erori ce pot apare în cazul în care se dealocă un alt număr de componente decât cel alocat, se poate utiliza sintaxa simplificată :

```
delete[] p ;
```

caz în care numărul de elemente dealocate este determinat în mod automat de către compilator.

Operatorul `new` se poate utiliza şi pentru crearea tablourilor multidimensionale, caz în care trebuie specificate toate dimensiunile tabloului. De exemplu, expresia:

```
new int[2][3][4]
```

creează în memorie două tablouri consecutive de tipul:

```
int [3][4]
```

şi returnează un pointer la primul tablou, adică un pointer de tipul:

```
int (*)[3][4]
```

Exemple:

```
int a[2][4] = {1, 2, 3, 4}, (*p)[4];
p = new int[2][4];
for (int i=0; i<2; i++)
 for (int j=0; j<2; j++)
    p[i][j] = a[i][j];
// ...
delete[] p;
```

Observație. Indiferent de numărul de dimensiuni al unui tablou alocat cu operatorul `new`, sintaxa pentru dealocarea lui cu operatorul `delete` este aceeaşi (o singură pereche de paranteze drepte).

Ca şi `new`, operatorul `delete` apelează implicit descructorul unei clase, dacă pointerul asupra căruia se aplică indică spre o instanță a unei anumite clase.

2.8 Funcții template

Limbajul C++ oferă suport pentru abstractizarea şi parametrizarea datelor. Principalele noțiuni adăugate sunt cele de ***funcții template*** şi ***clase template***. Despre aceste noțiuni se va discuta într-un capitol separat, în acest paragraf se vor specifica doar câteva noțiuni privind funcțiile generice.

O ***funcție template*** conține cel puțin un tip de date generic (nespecificat), ceea ce măreşte gradul de generalitate al acesteia.

Sintaxa de definire a unei funcții template impune prezența construcției :

`template ‘<’ class ⟨nume⟩ ‘>’`

înainte de antetul funcției. În această contrucție, ⟨`nume`⟩ reprezintă numele tipului de date (sau clasei) ce este parametru pentru funcție (nu se confundă cu parametrii formali) şi poate fi utilizat în corpul funcției.

O funcție template descrie o clasă de funcții, iar fiecare apel al unei asemenea funcții reprezintă o ***instanță*** a funcției respective. Sintaxa pentru instanțierea funcției în mod uzual este aceeaşi ca şi pentru apelul unei funcții obişnuite. Anumite compilatoare impun specificarea explicită a tipului de date cu care se realizează instanțierea.

Exemplul 2.7. Funcția *Interschimb* interschimbă valorile a doi parametri la care tipul de date este generic. În funcția `main` se vor apela două instanțe ale acestei funcții, pentru care tipul generic *Tip* este instanțiat la `int` şi la `double`.

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

template <class Tip>
void Interschimb(Tip & a, Tip & b) {
  Tip temp;
  temp = b;
  b = a;
  a = temp;
}

void main() {
  int a=3, b=5;
  double x=33, y=55;
  Interschimb(a,b);
  //Este corect si: Interschimb<int>(a, b);
  cout<<a<<" "<<b<<endl;
```

```
  Interschimb(x,y);
  //Este corect si: Interschimb<double>(a, b);
  cout<<x<<" "<<y<<endl;
}
```

2.9 Operatori de intrare-ieşire

Ca şi limbajul C, limbajul C++ nu posedă instrucțiuni specifice pentru operațiile de intrare şi ieşire. În afara funcțiilor specifice limbajului C, limbajul C++ pune în plus la dispoziție două ierarhii de clase pentru realizarea acestor operații. Conceptul de bază în aceste ierarhii este cel de `stream`, care poate fi asimilat cu un flux de date între un program şi un dispozitiv periferic.

Operațiile de intrare-ieşire specifice limbajului C++ se vor descrie într-un capitol separat, în acest paragraf se vor prezenta doar câteva elemente de bază pentru a putea fi folosite pentru citirea şi scrierea datelor.

Există două clase importante, numite `istream` şi `ostream`, folosite pentru gestionarea operațiilor de intrare şi de ieşire ; de asemenea există clasa `iostream` (derivată din `istream` şi `ostream`) folosită pentru ambele tipuri de operații. În fişierul header `iostream` există declarațiile principalelor clase, constante şi obiecte folosite pentru aceste operații. Cele mai utilizate sunt urmatoarele obiecte :

a) `cin` – este folosit pentru citirea datelor de la dispozitivul standard de intrare ;
b) `cout` – este folosit pentru scrierea datelor la dispozitivul standard de ieşire ;
c) `cerr` _ folosit pentru afişarea mesajelor de eroare ;
d) `clog` – utilizat ca şi cerr, dar tamponul nu se goleşte pentru fiecare mesaj de eroare.

Pentru realizarea efectiva a operațiilor de intrar-ieşire s-au definit anumiți operatori, care reprezintă supraâncarcarea unor operatori standard ai limbajului C. De exemplu, pentru a citi valor de la tastatură a fost supraâncarcat operatorul de deplasare dreapta (>>), iar pentru operația de scriere a fost suparaincarcat operatorul <<.

De exemplu, pentru a citi o valoare de la tastatură şi a o atribui unei variabile de tip întreg *n*, se poate proceda astfel :

```
cin >> n ;
```

iar pentru a afisa pe display valoarea lui *n* se poate scrie astfel :

```
cout << n ;
```

Aceşti operatori au fost supraâncarcati pentru toate tipurile de date predefinite, inclusiv pentru şiruri de caractere. În acest fel operanzii celor doi operatori pot avea orice tip de date predefinit.

Deoarece operatorii returnează şiruri de date, ei pot fi concatenați, astfel încât se pot citi sau scrie mai multe date cu ajutorul unui singur operator. De exmplu :

```
int n = 7 ;
double x = 4.5 ;
cout << n << x ;
```

Exemplul 2.8. Un program simplu care determină suma elementelor unui tablou cu valori citite de la tastatură.

```
#include <iostream>
using namespace std ;
void main() {
  int n = 10 ;
  double s = 0, x[10] ;
  cout << ”Dati elem. sirului : ” ;
  for (int i = 0 ; i < n ; i++) {
    cout << ”x[” << i << ”]= ” ;
    cin >> x[i] ;
    s += x[i] ;
  }
  cout << endl << ”s=” << s << endl ;
}
```

Observatie. Cuvantul rezervat `endl` reprezintă caracterul ‘`\n`’.

Limbajul C++ posedă de asemenea funcţii membru ale claselor de intrare-ieşire ce pot fi utilizate pentru citirea şi scrierea datelor. De exemplu, funcţiile `get` (inserează un caracter în şirul de ieşire şi returnează şirul respectiv la ieşire) şi `put` (extrage următorul caracter din şirul de intrare şi returnează şirul de intrare) au urmatoarea declaraţie :

```
ostream& put(char c) ;
istream& get(signed char &c) ;
```

Exemplul 2.9. Program pentru copierea fişierului standard de intrare la fişierul standard de ieşire :

```
#include <iostream>
using namespace std ;
void main() {
  int c ;
  while ( (c = cin.get()) != EOF)
    cout.put(c) ;
}
```