

## Capitolul 2

### Extensii ale limbajului C în limbajul C++

Limbajul C++ este un supraset al limbajului C și a fost proiectat de Bjarne Stroustrup. El oferă suport atât pentru programarea procedurală și 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, moștenire, 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` și `wchar_t`. În plus, el permite utilizarea încă a unui tip numit `string`, pentru care a fost creată o clasă `string`. Declarațiile acestei clase se află în fișierul `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` și `false`. Există o asemănare din acest punct de vedere cu limbajul Pascal, care are tipul predefinit `Boolean`, precum și cu Java, care are tipul `boolean`.

Spre deosebire de Pascal însă, există o compatibilitate între tipul `bool` și 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 instrucțiune 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 și instrucțiunea `if`.

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

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

se poate ști 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 și 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 și până la sfârșitul 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 și 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 același nume declarate în module diferite ale unui program accesează aceeași zonă de memorie și reprezintă aceleași variabile. O variantă de soluționare

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

Limbajul C++ atașează variabilele la un *spațiu al numelor*, ceea ce permite ca variabile cu același 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 șirurile de caractere declarate în fișierul `string` se poate scrie astfel :

```
#include <string>
```

**Observatie.** Fișierele 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 aceeași 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 dintr-un tip de bază cu ajutorul operatorului `&`. Dacă T este un tip de date, notația `T&` reprezintă tipul referință derivat din T, adică mulțimea tuturor elementelor referință la tipul T.

Valorilor elementelor de tip referință sunt asemănătoare pointerilor, în sensul că o referință are ca valoare adresa de memorie a unei variabile ce aparține tipului de bază. Există însă câteva deosebiri importante între pointeri și 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 deosebește 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 atât numele `k` cât și `r` pentru a referi același obiect.

b) În 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 precizări :

- valoarea unei referințe nu poate fi modificată după inițializare, ea referind întotdeauna același obiect cu care a fost inițializată. De exemplu, instrucțiunea `r=r+1;` nu înseamnă 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 acești parametri actuali să fie nume de variabile cu un tip de date egal cu tipul de bază al referințelor corespunzătoare și 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 execuție al funcției, ceea ce duce la mărirea timpului de execuție al programului și scăderea eficienței acestuia.

O alternativă o poate constitui folosirea macrodefinițiilor, care sunt substituite de compilator în etapa de preprocesare.

**Exemplul 2.2.** În secvența următoare se utilizează o macrodefiniție pentru determinarea minimumului a două valori.

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

Limbaajul C++ oferă în plus posibilitatea expandării *inline* a funcțiilor. Expandarea in-line înseamnă generarea de către compilator a codului corespunzător funcției, fără să se mai genereze o secvență de apel.

Declararea unei funcții inline se face prin specificarea cuvântului cheie *inline* înaintea definiției acesteia, iar pentru funcțiile membre ale unei clase prin includerea corpului funcției în cadrul declarației clasei .

**Exemplul 2.3.** Funcția inline *minim* este echivalentă cu macrodefiniția anterioară :

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

În cazul funcțiilor inline, compilatorul *încearcă* să plaseze o instanță a funcției inline în același segment de cod ca și cel al funcției apelate, dar nu se garantează însă acest lucru. Pentru funcțiile complicate (care conțin instrucțiuni repetitive sau care sunt recursive) nu se realizează mecanismul inline.

În general, utilizarea funcțiilor inline este mai eficientă decât a funcțiilor obișnuite, dar mai puțin eficientă decât utilizarea macrodefinițiilor.

**Exemplul 2.4.** Pentru exemplificare, se va utiliza aceeași operație de ridicare la putere prin trei metode diferite : macrodefiniție, funcție inline și funcție uzuală. Se va prelua timpul sistem pentru fiecare din cele trei funcții (f1, f2, f3), astfel încât să se determine timpul de execuția al fiecărei funcții.

```
#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 și la apelul unei funcții să fie utilizat același 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 și 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 obișnuit în antetul unei funcții prin nume și 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 inițializare; dacă însă parametrul actual corepunzător lipsește, 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 și între un punct și origine.

```
double Distanta(double 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 implicați 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 atât declarate cât și 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 fișierului header corespunzător, dacă acesta există.

## 2.6 Funcții supraîncărcate

*Supraîncărcarea numelui funcțiilor* înseamnă de fapt posibilitatea existenței mai multor funcții cu același nume care efectuează operații diferite. Un exemplu de supraîncărcare 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 și pentru operația de reuniune a mulțimilor, precum și pentru cea de concatenare a șirurilor de caractere.

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

**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) ;
    // ...
}

```

Compilerul determină funcția efectivă ce va fi apelată în funcție de tipul parametrilor și de numărul acestora.

#### Observații.

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

## 2.7 Operatori de gestiune a memoriei

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

Limbajul C++ posedă în plus doi operatori reprezentați prin cuvintele cheie `new` și `delete`.

Sintaxa uzuală de folosire este următoarea :

```

<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 și tipul pointerului ce se returnează.

Operatorul `new` se poate utiliza și la alocarea memoriei pentru elemente compuse. În cazul tablourilor, trebuie specificat numărul de componente ce se dorește 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 și funcția `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 încearcă 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 sfârșitul cuvântului `delete` între paranteze drepte, ca în exemplul următor :

```

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 și pentru crearea tablourilor multidimensionale, caz în care trebuie specificate toate dimensiunile tabloului. De exemplu, expresia:

```

new int[2][3][4]

```

crează în memorie două tablouri consecutive de tipul:

```

int [3][4]

```

și 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<4; 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 aceeași (o singură pereche de paranteze drepte).

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

## 2.8 Funcții template

Limbaajul C++ oferă suport pentru abstractizarea și parametrizarea datelor. Principalele noțiuni adăugate sunt cele de *funcții template* și *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 mărește gradul de generalitate al acesteia.

Sintaxa de definire a unei funcții template impune prezența construcției :

```
template '<' class <nume> '>'
```

înainte de antetul funcției. În această construcție, <nume> reprezintă numele tipului de date (sau clasei) ce este parametru pentru funcție (nu se confundă cu parametrii formali) și 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 aceeași ca și pentru apelul unei funcții obișnuite. 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` și 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 și limbajul C, limbajul C++ nu posedă instrucțiuni specifice pentru operațiile de intrare și ieșire. 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 și un dispozitiv periferic.

Operațiile de intrare-ieșire 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 și scrierea datelor.

Există două clase importante, numite *istream* și *ostream*, folosite pentru gestionarea operațiilor de intrare și de ieșire; de asemenea există clasa *iostream* (derivată din *istream* și *ostream*) folosită pentru ambele tipuri de operații. În fișierul header *iostream* există declarațiile principalelor clase, constante și obiecte folosite pentru aceste operații. Cele mai utilizate sunt următoarele 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 ieșire ;
- c) `cerr` – folosit pentru afișarea mesajelor de eroare ;
- d) `clog` – utilizat ca și `cerr`, dar tamponul nu se golește pentru fiecare mesaj de eroare.

Pentru realizarea efectivă a operațiilor de intrare-ieșire s-au definit anumiți operatori, care reprezintă supraîncărcarea unor operatori standard ai limbajului C. De exemplu, pentru a citi valor de la tastatură a fost supraîncărcat operatorul de deplasare dreapta (>>), iar pentru operația de scriere a fost supraîncărcat operatorul <<.

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

```
cin >> n ;
```

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

```
cout << n ;
```

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

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

```
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 funcții membru ale claselor de intrare-ieșire ce pot fi utilizate pentru citirea și scrierea datelor. De exemplu, funcțiile get (inserează un caracter în șirul de ieșire și returnează șirul respectiv la ieșire) și put (extrage următorul caracter din șirul de intrare și returnează șirul de intrare) au următoarea declarație :

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

**Exemplul 2.9.** Program pentru copierea fișierului standard de intrare la fișierul standard de ieșire :

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