Cursul 13

Operaţii de intrare/ieşire în C++

In mod obişnuit, un program de calcul citeşte o serie de date de la tastatură sau de pe disc şi le transferă în memoria internă alocată programului, le prelucrează pe baza setului de instrucţiuni - prin transferuri repetate între regiştrii de memorie şi regiştrii micro-procesorului - şi în final le scrie sub forma prelucrată pe monitor sau într-un fişier pe disc. Cu alte cuvinte, un program de calcul execută *operaţii de intrare* (input, read operations) prin *citirea* datelor, *operaţii de prelucrare* a datelor şi *operaţii de ieşire* (output, write operations) prin *scrierea* datelor. Aceste operaţii nu se execută neapărat în ordinea dată aici, ele sunt intercalate şi executate în mod repetat conform cu necesităţile problemei de rezolvat.

Modul de derulare a operaţiilor de intrare/ieşire (*operaţii* I/O) este specific fiecărui dispozitiv hardware în parte, pentru a le executa sistemul de operare utilizează programe speciale (*drivere*) puse la dispoziţie de producătorii dispozitivelor. Existenţa acestor drivere asigură un anumit grad de independenţă a sistemului de operare faţa de maşină: fără ele ar trebui proiectat câte un sistem de operare pentru fiecare configuraţie hardware în parte. Acesta este şi motivul pentru care, atât în limbajul C iniţial cât şi în extensia sa C++, nu există instrucţiuni de intrare/ieşire: s-a evitat legarea limbajului de un anumit sistem de operare sau de o anumită configuraţie hardware. In limbajul C operaţiile I/O se execută utilizând funcţii de intrare/ieşire, iar în limbajul C++ utilizând clase de intrare/ieşire.

In C++ obiectele date de clasele specializate în operaţii I/O se numesc *fluxuri de date* sau *stream*-uri. Clasele de stream-uri nu aparţin limbajului, dar au un caracter standard şi o largă circulaţie printre programatori. Toate compilatoarele de C++ sunt dotate cu biblioteci I/O standardizate, care furnizează clase de stream-uri gata implementate. Programatorul poate să utilizeze stream-urile din aceste clase sau să-şi definească (prin derivare) propriile clase, după necesităţi. În MS Visual Studio, cele mai utilizate biblioteci I/O sunt **<iostream>**, în special pentru stream-urile predefinite **cin** şi **cout** care sunt instanţe ale claselor **ostream**, respectiv **istream**, şi **<fstream>**, pentru operaţii I/O cu fişiere stocate pe disc.

1. Generalităţi despre clase

Pentru înţelegerea utilizării claselor de intrare/ieşire trebuie să spunem câteva lucruri despre clase. Trecerea de la limbajul C la C++ s-a făcut prin introducerea acestui nou tip de dată, tipul **class**, care extinde tipul **struct,** permiţând programatorului să înglobeze la un loc, pe lângă un set de date şi o serie de funcţii specializate în prelucrarea acestor date.

Începem cu următorul exemplu în care definim clasa **Punct** care are trei *date membre*, **x**, **y** şi **mesaj**, un *constructor* şi o *funcţie membru,* funcţia **opus()**:

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

class Punct{
public:
    double x, y;
    static const char * mesaj;
    Punct(double xx=13, double yy=13); //constructor
    Punct opus();

};

const char* Punct::mesaj = "punctul";

Punct::Punct(double xx, double yy){
    x = xx;
    y = yy;
}

Punct Punct :: opus(){
    return Punct(-x, -y);

}

ostream& operator<< (ostream& xout, const Punct& p){
    xout << Punct::mesaj<<"("<< p.x << "," << p.y<<") ";
    return xout;
}


int main()
{
    Punct a;
    cout << a.x << endl; //13

    Punct b = Punct(1, 2);
    cout << b.opus().x << endl; //-1

    Punct c(11, 22);
    cout << c.mesaj << endl;  //punctul
    //.................................................
    return 0;
}
```

În funcţia **main()** de mai sus prezentăm trei modalităţi de instanţiere a clasei Punct. Declaraţia lui **a** cheamă constructorul clasei cu valorile implicite ale argumentelor, 13 şi 13, **b** este iniţializat prin apelarea "în clar" a constructorului, iar **c** este iniţializat "în stil C++", tot prin apelarea constructorului.

Spre deosebire de ceilalţi membrii, **mesaj** este declarat **static**, aceasta însemnă că toate instanţele clasei au aceeaşi valoare pentru membrul **mesaj**, din acest motiv el poate fi apelat la nivel de clasă, cu *operatorul de rezoluţie* **::** astfel:

```
cout << Punct::mesaj << endl; //punctul
```

Orice încercare de tipul

```
cout << Punct::x << endl;
//Error: illegal reference to non-static member 'Punct::x'
```

este sortită eşecului.

Să analizăm acum *supraîncărcarea* operatorului de deplasare << efectuată prin declaraţia

```
ostream& operator<< (ostream& xout, const Punct& p){
        xout << Punct::mesaj<<"("<< p.x << "," << p.y<<") ";
        return xout;
}
```

Dacă în **main()** scriem

```
cout << c << endl;  //punctul(11,22)
operator<<(cout, c) << endl;    //punctul(11,22)
```

obţinem rezultatele din comentarii deoarece cele două instrucţiuni sunt echivalente. Prin expresia `cout << c` compilatorul înţelege de fapt apelul de funcţie `operator<<(cout, c)`, în care trimitem prin referinţă stream-ul **cout**, care este o instanţă a clasei **ostream**, şi pe care îl primim prin referinţă ca rezultat după terminarea apelului, fapt care ne permite să concatenăm operatorul <<. Alt exemplu de instrucţiuni echivalente:

```
cout << b << c<< endl;
operator<<(operator<<(cout, b), c) << endl;
```

Facem observaţia că în clasele de stream-uri operatorii de deplasare pe biţi sunt supraîncărcaţi şi, din acest motiv, au denumiri specifice: *operatorul de inserţie* << şi *operatorul de extracţie* >>.

O clasă poate să ascundă sau nu anumiţi membri, după cum îi declară cu modificatorii de acces **public**, **protected** sau **private**. Datele private sunt acesibile numai din interiorul clasei (numai funcţiile memru ale clasei le pot modifica, de exemplu), cele protejate sunt accesibile şi de către clasele derivate, iar cele publice pot fi accesate din orice punct al programului din care se vede clasa respectivă. Acest mecanism permite *încapsularea informaţiei* în interiorul clasei astfel încât, pentru a utiliza o clasă dintr-o bibliotecă, de exemplu, programatorul trebuie să ştie numai membrii la care are acces şi ce anume fac ei, fără să cunoască cum sunt ei implementţi în corpul clasei.

Clasele pot fi definite prin *derivare*, declarând în momentul definirii că noua clasă extinde una mai veche, deja definită. Clasa derivată (*clasa copil*) moşteneşte membrii clasei iniţiale (numită *clasă mamă*, *clasă părinte* sau *clasă de bază*) şi poate adăuga noi membrii, sau îi poate redefini prin *suprascriere* pe cei moşteniţi.

De exemplu, unui obiect din clasa **Punct** îi putem ataşa un nume, format dintr-o singură literă:

```
class PunctNumit : public Punct{
public:
    char nume;
    PunctNumit(Punct a, char ch){
        x = a.x;
        y = a.y;
        nume = ch;
    }
    PunctNumit(double x, double y, char ch){
        this->x = x;
        this->y = y;
        this->nume = ch;
    }
```

```
};

ostream& operator<< (ostream& xout, const PunctNumit& p){
   xout << Punct::mesaj <<" "<<p.nume<< "(" << p.x << "," << p.y << ") ";
   return xout;
}
```

Un obiect din clasa derivată `PunctNumit` se declară în mod obişnut:

```
Punct q(111, 222);
PunctNumit qA(q, 'A');
cout << qA << endl; //punctul A(111,222)
```

O facilitate suplimentară pusă în ultimul timp la dispoziţia programatorilor în limbajul C++ este *programarea generică*, prin utilizarea *şabloanelor de clase*. În exemplul următor

```
template <class TIP> class PunctGeneric{
public:
       TIP x, y;
       static char * mesaj;
       PunctGeneric(TIP xx, TIP yy){
              x = xx;
              y = yy;
        }
       PunctGeneric opus(){
              return PunctGeneric(-x,-y);
       }
};
```

s-a definit şablonul `PunctGeneric<>` în care tipul membrilor x şi y nu este bine precizat, el va fi stabilit la instanţierea unui obiect din această clasă, printr-o declaraţie de forma:

```
PunctGeneric<int> aaa(1, 2);
cout << aaa.opus().x << endl;//-1
```

În practică se foloseşte foarte des redenumirea cu **typedef** a diverselor *specializări* ale unui şablon: în urma instrucţiunii

```
typedef PunctGeneric<double> Punctt;
```

identificatorul `Punctt` este sinonim cu tipul `PunctGeneric<double>` care este o specializare a şablonului `PunctGeneric<>` pentru tipul `double`.

```
Punctt z(1.1, 2.2);
cout << z.x << endl; //1.1
```

Această facilitate este intens folosită în construcţia bibliotecilor de stream-urilor, unde trebuie să avem atât fluxuri pentru caracterele clasice, pe un octet, adică de tip **char**, cât şi, de exemplu, fluxuri pentru caracterele de tip *wide*, pe doi octeţi. Este clar că funcţionalitatea acestor stream-uri este foarte asemănătoare şi utilizarea şabloanelor permite reutilizarea codului într-o proporţie considerabilă. De exemplu, pentru stream-uri de ieşire este implementată clasa şablon:

```
template <class CharT, class Traits = char_traits<class CharT>>
class basic_ostream : virtual public ios_basic<CharT, Traits>{...};
```

şi apoi sunt definite specializările

```
typedef basic_ostream<char, char_traits<char>> ostream;
typedef basic_ostream<wchar_t, char_traits<wchar_t>> wostream;
```

Prin acest mecanism este păstrată compatibilitatea cu versiunile mai vechi ale bibliotecilor I/O, unde **ostream**, de exemplu, era o clasă propriu-zisă şi nu doar o specializare a unui şablon.

2. Stream-urile **cin** şi **cout**

https://docs.microsoft.com/en-us/cpp/standard-library/iostream

Operaţiile I/O uzuale, cele care au loc pe ruta tastatură-program-monitor, sunt realizate cu stream-urile standard **cin**/**cout**, utilizând operatorii de inserţie/extracţie <</>>. Nu este singura posibilitate, dar este cea mai uşoară, deoarece **cin**/**cout** sunt stream-uri predefinite: sunt gata declarate şi iniţializate în **<iostream>** ca instanţe ale claselor **istream/ostream**, au gata ataşate fişierele logice **stdin/stdout** (consolele standard de intrare/ieşire, mai precis tastatura/monitorul), şi au operatorii <</>> supraîncărcaţi astfel încât să se descurce singuri cu datele din tipurile fundamentale (aritmetice sau pointer) şi chiar cu unele tipuri compuse (şiruri de caractere). La rândul lor clasele **istream** şi **ostream** sunt derivate din clasa **ios**, în consecinţă stream-urile **cin şi cout** moştenesc şi membrii aceastei clase.

Observaţie: In fişierul header **<iostream>** se găsesc declaraţiile

```
extern ostream cout;
extern istream cin;
```

care certifică afirmaţiile de mai sus.

Să exemplificăm utilizarea operatorul "." la selectarea unei metode a stream-ului **cout**. Mai precis, în următorul program vom apela metoda

```
int precision( int np );
```

definită în clasa **ios**

```
#include<iostream>
using namespace std;
int main(){
    double a = 1.0 / 7.0;
    int old_prec, new_prec = 16;
    cout << "a=" << a << endl;
    old_prec = cout.precision(new_prec);
    cout << "vechea afisare era cu " << old_prec << " cifre" << endl;
    cout << "acum afisam cu " << new_prec << " cifre:" << endl;
    cout << "a=" << a << endl;
    return 0;
}
```

Pe monitor va fi scris următorul fişier de ieşire:

```
a=0.142857
vechea afisare era cu 6 cifre
acum afisam cu 16 cifre:
a=0.1428571428571429
Press any key to continue . . .
```

Observăm că, în mod implicit, numărul de cifre cu care stream-ul **cout** scrie o valoare de tip double este 6, iar acest număr poate fi aflat şi modificat cu metoda `precision()`, care fiind o funcţie membru a clasei din care face parte stream-ul c**out** este indicată cu operatorul de selecţie ".". Reţinem: instrucţiunea

```
cout.precision(8);
```

setează noua precizie de scriere a stream-ului la 8 cifre, iar vechea precizie, returnată de funcţia `precision()`, se pierde deorece aici nu este atribuită nici unei variabile.

Menţionăm că aceasta nu este singura cale prin care se poate modifica precizia de scriere a valorilor în virgulă mobilă, mai precis vom vedea mai târziu că *manipulatorul de formatare* `setprecision` rezolvă mai comod aceeaşi problemă:

```
cout << "a=" << setprecision(16)<<a << endl;
```

Să punem în evidentă şi unele limitări ale operatorilor de inserţie/extracţie <</>>. De exemplu, dacă vrem să aflăm adresele unor variabile cu secvenţa:

```
char ch = 'A';
int i = 1;
cout << &i << endl;
cout << &ch << endl;
```

obţinem

```
0x0012FF78
A╠╠╠└ ↕
```

Explicaţie: `&ch` este o constantă de tip `char*` iar in cazul operandului de tip `char*` operatorul << înţelege că are de pus în flux un şir de caractere, aşa că în cazul nostru porneşte de la adresa lui `ch` şi scrie pe monitor toţi octeţii întâlniţi în memorie până dă peste unul nul. Dacă vrem să vedem totuşi adresa variabilei `ch`, trebuie să folosim o conversie explicită către tipul `void*`. Secvenţa

```
cout << &i << endl;
cout << (void*)&ch << endl;
```

are rezultatul scontat:

```
0x0012FF78
0x0012FF7C
Press any key to continue.
```

3. Citirea şi scrierea fişierelor pe disc. Prezentare rapidă

Clasele I/O standard sunt astfel concepute încât lucrul cu fişierele stocate pe disc să decurgă în mod similar cu operaţiunile I/O uzuale, deja cunoscute. Trebuie numai să ne alegem o clasă de stream-uri convenabilă, să instanţiem un obiect din acea clasă şi să-i ataşăm fişierul din care dorim să citim sau în care vrem să scriem. In rest, operaţiile I/O propriu-zise decurg, cel puţin în cazul fişierelor text, ca în cazul lucrului cu **cin/cout**.

Incepem cu următorul exemplu: scriem un text în fişierul “felicitare.txt” care va fi creat de către program în directorul curent:

```
#include<iostream>  //pentru cout
#include<fstream>   //pentru ofstream
using namespace std;

int main(){
    ofstream xout("felicitare.txt");
    xout << "Sa traiti bine!" << endl;
    cout << "Am scris." << endl;
    return 0;

}
```

Dacă totul decurge fară erori, pe monitor apare mesajul "`Am scris.`" iar în dosarul proiectului găsim fişierul `felicitare.txt` pe care îl putem citi cu orice editor de texte (cu notepad-ul , de exemplu). O singură precizare: în instrucţiunea

```
ofstream xout("felicitare.txt");
```

declarăm şi iniţializăm un stream din clasa **ofstream** (deci un output-file-stream) pe care l-am denumit **xout** şi căruia îi ataşăm fişierul `felicitare.txt` care va fi astfel fişierul de ieşire al stream-ului.

Acum dorim să citim printr-un program C++ fişierul creat şi să îl afişăm pe monitor:

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
	ifstream xin("felicitare.txt");
	char ch;
	do{
		xin >> ch;
		cout << ch;
	} while (ch != '!');//
	cout << "\nAm citit si am scris." << endl;
	return 0;
}

//Satraitibine!
//Am citit si am scris.
//Press any key to continue . . .
```

Am definit input-file-stream-ul **xin**, i-am ataşat fişierul `felicitare.txt` care acum are rolul de fişier de intrare, am citit fişierul caracter cu caracter şi l-am afişat pe monitor prin stream-ul **cout**. Rezultat: `Satraitibine!` Stim că textul a fost scris corect în `felicitare.txt`, am verificat cu notepad-ul, deci greşeala este la citire. In general citirea datelor este mai dificilă decât scrierea lor. Pentru operatorul de extracţie >>, *spaţiile albe* (blanc-ul, tab-urile, caracterul new-line, carriage return) joacă rolul de *delimitatori* de câmpuri de citire, şi sunt utile când citim o serie de string-uri sau de numere, la citirea "caracter cu caracter" operatorul ">>" extrage spaţiile albe din buffer-ul ataşat stream-ului dar nu le returnează programului apelant, sare pur şi simplu peste ele!

Corectăm ultimul exemplu: renunţăm la >> şi apelăm la funcţia membru **get()**. Stream-ul nostru, denumit de noi **xin**, fiind un obiect din clasa **ifstream,** moşteneşte de la clasa mamă **istream** funcţia **get()** sub două forme: **int get()**, care extrage orice caracter din buffer şi întoarce codul ASCII al caracterului citit, şi **istream& get(char&)**, care, fiind apelată prin referinţă, încarcă direct în parametrul actual (de tip char) caracterul citit şi întoarce o referinţă la stream-ul apelant (şi astfel poate fi apelată în mod succesiv, prin concatenare: xin.get(ch1).get(ch2).get(ch3);).

Prezentăm mai întâi varianta clasică, *stilul C*, în care folosim constanta predefinită EOF (end of file), care marchează sfârşitul oricărui fişier text:

```
#include<iostream>
#include<fstream>
```

```
using namespace std;
int main(){
    ifstream xin("felicitare.txt");
    int ch;
    while ((ch = xin.get()) != EOF)
        cout << (char)ch;
    cout << "\nAcum e OK!." << endl;
    return 0;
}
```

Urmează varianta în *stilul C++*, în care folosim metoda **eof()**, care precizează o *condiţie de stare* a stream-ului (dacă a ajuns sau nu la sfarşitul fişierului):

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

int main(){
    ifstream xin("felicitare.txt");
    char ch;
    while (!xin.get(ch).eof())
        cout << ch;
    cout << "\nGata." << endl;
    return 0;
}
```

Renunţarea la operatorul de extracţie >> nu s-a datorat lucrului cu fişiere pe disc ci limitărilor acestuia, limitări care apar şi în cazul stream-urilor **cin/cout**. De exemplu, în urmatorul program reuşim cu greu să încărcăm în tabloul de caractere sText, dintr-o singură citire, atât numele cât şi prenumele utilizatorului. La prima încercare, operatorul >> citeşte numai numele, prenumele nu ajunge în sText, mai mult, rămâne în buffer-ul tastaturii şi trebuie şters de programator ca să nu încurce noua încercare:

```
#include <iostream>
using namespace std;
int main(void)
{
    const int dimText = 100;
    char sText[dimText];

    cout << "nume prenume:" << endl;
    cin >> sText;
    cout << "deci te numesti " << sText << endl;
    cout << "scuze, din nou:" << endl;
    // pt operatorul '>>' spatiile albe ' ','\t','\n'
    // sunt delimitatori. Prenumele a ramas in cin.
    // acum trebuie sa curatam buffer-ul:
    //while(cin.get()!='\n')
    // ;
    // merge si asa, dar este mai elegant
    // sa folosim istream::ignore, astfel:
    cin.ignore(4096, '\n');
    cin.getline(sText, dimText);
    cout << "OK,te numesti " << sText << endl;
    cout << "OFF" << endl;
    return 0;
}
```

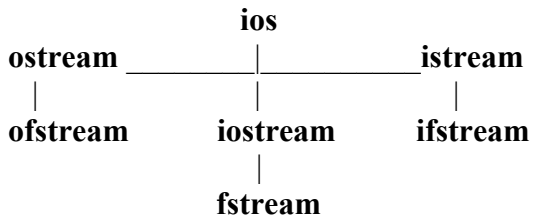
Stream-ul predefinit **cin** moşteneşte de la clasa **istream** două funcţii extrem de utile: **getline()** şi **ignore()**. Apelul `cin.getline(sText,dimText)` citeşte din fişierul ataşat stream-ului până la primul `'\n'` (delimitator implicit) sau până citeste `dimText-1` caractere şi apoi le depune în sText, adaugând, dacă mai este cazul, terminatorul de string `'\0'`. Apelul `cin.ignore(4096,'\n')` extrage şi aruncă la gunoi toate caracterele întâlnite până la primul `'\n'` (în acest caz) sau până la al 4096-lea caracter (în mod obişnuit buffer-ul ataşat unui stream are 4 Ko).

In concluzie, operaţiile de intrare/ieşire în C++ sunt simple şi eficiente, dar pentru utilizarea lor cu acurateţe este necesară o bună cunoaştere a ierarhiei claselor de stream-uri şi a metodelor publice ale acestora.

4. Citirea şi scrierea fişierelor pe disc. Prezentare în amănunt

Orice operaţie I/O se execută prin intermediul unui stream, fiecare stream avand ataşat câte un singur fişier. Aşa cum am mai spus, un stream poate fi înţeles ca fiind un canal de comunicaţie prin care trec datele într-un flux cu o direcţie bine precizată, având la un capăt programul nostru iar la celălat capăt un fişier pe post de *rezervor de date*. Datele circulă în pachete formate din octeţi (*bytes*). Pentru a nu deranja sistemul de operare pentru fiecare octet transferat, stream-urile uzuale sunt prevăzute cu câte un *buffer* (zona de memorie cu organizare tip "coadă de aşteptare", cu mărime standard de 4Ko), transferul memorie-dispozitiv de stocare având loc automat la umplerea buffer-ului sau la cererea expresă a programului (un apel **flush()** sau **close()**, de exemplu).

In funcţie de tipul operaţiilor pe care le pot efectua, avem *stream-uri de intrare* (clasele **istream**, **ifstream,** ş.a), *stream-uri de iesire* (clasele **ostream**, **ofstream**, ş.a.) şi *stream-uri de intrare-iesire* (clasele **iostream**, **fstream** ş.a.). Printr-un stream de intrare (*input stream*) se poate efectua numai citirea datelor din fişierul ataşat, printr-unul de ieşire (*output stream*) numai scrierea lor, iar printr-un stream de intrare-ieşire se pot executa ambele operaţii asupra aceluiaşi fişier. Clasele menţionate sunt derivate din clasa **ios** conform urmatoarei diagrame:



Intr-un program putem declara şi folosi oricâte stream-uri avem nevoie:

```
#include <fstream>
void lucrulPeDisc(){
    ofstream xout, yout, fluxDeIesire;    // stream-uri de iesire
    fstream  inOut;                       // stream de intrare-iesire
    ifstream fluxDeIntrare, xin;          // stream de intrare
    //.....................
 }
```

Orice lucrare asupra unui fişier se execută în trei paşi: *deschiderea* fişierului, *prelucrarea* sau *utilizarea* fişierului şi *închiderea* lui. Deschiderea fişierului se face odată cu asignarea (asocierea, atribuirea) lui unui stream. Un stream poate să aibă asignat un singur fişier la un moment dat.

Deschiderea şi asignarea unui fişier se pot efectua chiar în momentul definirii stream-ului (utilizând o variantă a constructorului clasei) sau se pot efectua ulterior, cu metoda **open()** a stream-ului. Exemplu: instrucţiunea

```
ofstream fluxout("C:\\docuri\\date.txt");
```

este echivalentă cu secvenţa

```
ofstream fluxout;
fluxout.open("C:\\docuri\\date.txt");
```

în care este declarat mai întâi stream-ul de ieşire `fluxout` şi apoi acestuia i se asociază `date.txt` din dosarul `C:\docuri`.

Inchiderea fişierului asociat stream-ului se execută cu metoda **close(),** astfel:

```
fluxout.close();
```

Inchiderea fişierului constă în completarea transmiterii tuturor datelor (prin golirea buffer-ului ataşat stream-ului) şi, eventual, din descărcarea fişierului din memorie şi rescrierea lui pe disc. Subliniem că operaţia de închidere are efect numai asupra fişierului ataşat, nu şterge şi stream-ul, acesta poate fi eventual refolosit pentru deschiderea altui fisier. Inchiderea trebuie cerută explicit de programator, altfel fişierul este închis de sistemul de calcul la terminarea execuţiei programului (dacă execuţia s-a terminat cu bine).

Un exemplu corect de lucru cu fişiere:

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    const int dim = 1024;
    char buf[dim];
    ofstream xout;
    ifstream xin;
    xout.open("date.txt");
    xout << "NU UITA SA INCHIZI FISIERUL!";
    xout.close();
    xin.open("date.txt");
    xin.getline(buf, dim);
    cout << buf << endl;
    xin.close();
    return 0;
}
```

Rezultat:

```
NU UITA SA INCHIZI FISIERUL!
Press any key to continue
```

Dacă nu folosim **close()**:

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

```
int main(){
    const int dim = 1024;
    char buf[dim];
    ofstream xout;
    ifstream xin;
    xout.open("date.txt");
    xout << "AI UITAT SA INCHIZI FISIERUL!";
    //xout.close();
    xin.open("date.txt");
    xin.getline(buf, dim);
    cout << buf << endl;
    //xin.close();
    return 0;
}
```

obţinem:

```
Press any key to continue
```

Citim cu notepad-ul, de exemplu, fişierul `date.txt` şi vedem ca el conţine textul pe care am vrut să-l scriem, `"AI UITAT SA INCHIZI FISIERUL!"`, şi atunci de ce **xin**-ul nu l-a citit? Răspunsul e simplu: în momentul când am vrut să-l citim, textul se găsea în bufferul lui **xout** - deorece nu am închis fişierul la timp - şi a ajuns în `date.txt` abia la terminarea execuţiei programului.

Asignarea aceluiaşi fişier la două stream-uri diferite nu este interzisă dar este un mod de operare impropriu, dacă într-un program dorim să scriem şi să citim din acelaşi fişier, ori îl deschidem cu un singur stream de intrare-ieşire (din clasa **fstream**), ori îl prelucrăm succesiv cu două streamuri uni-direcţionale (unul de intrare şi altul de ieşire) deschizând şi închizând fişierul ori de câte ori este nevoie.

Un fişier poate fi *creat*, *rescris*, *completat, modificat* sau *citit*, iar deschiderea fişierului este însoţită, explicit sau nu, de precizarea *modului de deschidere* a fişierului, mod care stabileşte operaţiile de prelucrare la care va fi supus fişierul precum şi modul de operare al stream-ului. Modurile de deschidere sunt indicate cu ajutorul unor constante numerice definite în clasa **ios**, conform tabelului următor:

open_mode	**Utilizare**
ios::app	Deschide un fişier de ieşire pentru completare (appending). Dacă fisierul nu există, este creat pe loc. Datele sunt scrise numai pe ultimul loc.
ios::ate	Deschide un fişier existent (pentru citire sau/şi scriere) şi poziţionează prompterul pe ultimul rând (at end). Primul octet este scris pe ultimul loc, următorii sunt scrisi acolo unde este pozitionat prompterul.
ios::in	Deschide un fişier pentru citire. Poate fi utilizat şi cu un **ofstream**, pentru a preveni trunchierea unui fişier deja existent.
ios::out	Deschide un fişier pentru scriere. Dacă fişierul nu există, este creat pe loc. Dacă fişierul există deja, iar **ios::out** este folosit fără modurile **ios::app**, **ios::ate** sau **ios::in**, atunci este implicat modul

	ios::trunc (fişierul este curăţat şi prompterul este poziţionat la start).
ios::trunc	Deschide un fişier nou şi îl şterge (curăţă) pe cel vechi (dacă acesta deja există).
ios::binary	Deschide un fişier în modul binar (modul text este implicit).

Modul **ios::in** este implicit pentru clasa **ifstream**, iar **ios::out** pentru **ofstream**. Astfel

```
ofstream fluxout("date.txt");
```

deschide fişierul în modul **ios::out**, iar

```
ifstream fluxin("date.txt");
```

îl deschide în modul **ios::in**. Pentru a deschide fişierul în mai multe moduri simultan, se utilizează operatorul *sau pe biţi* "|", conform următoarelor exemple:

```
ifstream xin("baza_de_date.txt", ios::in | ios::binary);
ofstream xout;
xout.open("date.txt", ios::out | ios::in);
```

Fişierul `date.txt` a fost deschis în modul indicat pentru ca, de exemplu, în cazul în care îl închidem fără să fi scris nimic în el vechiul conţinut să se păstreze (în modul implicit de deschidere are loc *rescrierea* fişierului: înainte de a scrie în fişier vechile date sunt şterse).

Toate fişierele sunt formate din biţi grupaţi câte opt în octeţi. Unitatea de măsură pentru lungimea unui fişier sau pentru poziţionarea prompterului I/O este octetul.

Fişierele sunt de două tipuri: *text* sau *binar*, după cum octeţii care îi compun au sau nu semnificaţia de coduri de caractere. Fluxurile de date lucrează implicit în modul text, comutarea în modul binar trebuie făcută explicit, cu constanta **ios::binary**. Modul binar presupune utilizarea funcţiilor **read** şi **write**, care, spre deosebire de operatorii de inserţie/extracţie, sunt indiferente la eventuala semnificaţie de caractere de control a octeţilor manipulaţi. Fişierele binare sunt mai compacte, controlul poziţionării prompterului de acces este exact, dar informaţia stocată în ele poate fi regasită numai dacă se cunoaşte riguros modul în care au fost organizate datele la crearea fişierului. Fişierele binare pot fi citite, în mod uzual, numai de programul care le-a creat.

Fişierele scrise în mod text sunt organizate pe linii de lungime variabilă, linii delimitate de anumite caractere de control (`'\n'` sau `'\r'`, de obicei), liniile se succed una după alta (nu se adauga "spaţii"), sfârşitul de fişier este marcat printr-un caracter special, caracterul "**eof**". In raport cu fişierele binare, fişierele text fac o oarecare risipă de spaţiu de stocare (nu prea mare), mai mult, controlul prompterului este dificil, deoarece poziţionarea sa depinde de diverse convenţii de scriere. Avantajul major al fişierelor text constă în faptul că informaţia conţinută în ele poate fi regasită cu orice program de editare de texte.

Consola de intrare (fişierul logic asociat tastaturii) şi consola de ieşire (monitorul) sunt fişiere text, deci **cin**/**cout** operează numai în modul text (şi numai secvenţial, vezi mai jos).

Accesarea datelor unui fişier poate fi facută în mod *secvenţial* sau în mod *aleator* (cu sensul: "pe sărite"). Accesarea secvenţială (citirea de la început până la sfârşit, scrierea numai pe "ultimul rând") este utilizată mai ales în cazul fişierelor text, şi are loc de la sine atât timp cât programatorul nu intervine în poziţionarea prompterului I/O. Accesul

aleator este utilizat mai ales în cazul fişierelor binare, deoarece în acest caz programatorul ştie exact în ce loc din fişier va ajunge fiecare octet trimis prin stream.

Fiecare stream are un *prompter* I/O, adică o variabilă de tip **int** care conţine poziţia de scriere/citire curentă din fişier. Prompterul I/O mai este numit şi *pointer* I/O, deoarece valoarea sa are semnificaţia unei adrese. In cazul stream-urilor de intrare prompterul este numit "get pointer", poziţia sa poate fi aflată cu funcţia membru **tellg()** şi poate fi poziţionat cu **seekg()**, în cazul stream-urilor de ieşire avem un "put pointer" şi funcţiile membru **tellp()** şi **seekp()**. Prompterul unui stream de intrare-ieşire este numit şi "get pointer" şi "put pointer" după cum îl utilizăm pentru citire sau pentru scriere, poziţionarea lui fiind făcută cu funcţiile corespunzătoare întrebuinţării. Trebuie reţinut că, în acest caz, orice poziţionare cu **seekg()**, de exemplu, modifică şi valoarea întoarsă de **tellp()**, deoarece stream-ul are numai un singur prompter!

Succesul unei operaţii de intrare/ieşire depinde de foarte mulţi factori care apar în cursul rulării şi asupra cărora programatorul nu poate interveni: dacă fişierul există sau nu acolo unde trebuie, dacă utilizatorul are sau nu dreptul să scrie pe disc, daca este suficient spaţiu de stocare, etc. Programatorul trebuie să testeze mereu, prin program, dacă operaţia I/O efectuată a eşuat sau nu.

Pentru semnalarea eventualelor erori fluxurile de date sunt dotate cu *indicatori de stare*, care pot fi consultaţi de următoarele funcţii membru:

Funcţia	**Valoare returnată: FALSE / TRUE (zero / diferit de zero)**
bad()	întoarce **TRUE** dacă a aparut o eroare I/O nerecuperabilă, de exemplu: o tentativă de scriere/citire dincolo de sfârşitul fişierului.
fail()	întoarce **TRUE** dacă a aparut o eroare I/O nerecuperabilă sau dacă o operaţie de deschidere nu a avut succes.
eof()	întoarce **TRUE** dacă a fost atins sfârşitul de fişier .
good()	întoarce **TRUE** dacă nici una din condiţiile de mai sus nu este adevarată.

După tratarea unor eventuale erori, indicatorii de stare pot fi resetaţi cu funcţia membru **clear()**. Pentru detalii, vezi https://docs.microsoft.com/en-us/cpp/standard-library/basic-ios-class

Imediat după deschiderea unui fişier, de exemplu, trebuie să testăm succesul operaţiei printr-o instrucţiune de forma:

```
if (xout.fail()) cout << "Au!" << endl;//eroare la deschidere.
```

Operatorul **!** a fost supraîncărcat pentru a da ca rezultat valoarea returnată de metoda **fail()** a stream-ului testat. Exemplul precedent poate fi rescris astfel:

```
if (!xout) cout << "Au!" << endl;//eroare la deschidere.
```

Operatorul **void*()** a fost supraîncărcat pentru a fi în opoziţie cu operatorul **!**, deci

```
if (xout)...
```

este echivalentă cu

```
if (!xout.fail())...
```

dar nu este echivalentă cu

```
if (xout.good())...
```

deoarece `xout` nu testează şi indicatorul **eof**. Recomandăm utilizarea acestei ultime forme pentru testarea "sănătăţii" stream-ului.

Pentru diagnosticarea erorilor de intrare/ieşire apărute şi întreruperea executării programului vom folosi, în exemplele care urmează, fişierul header **assert.h**, fişier în care este definit macroul **assert()**. Acest macrou, după expandare (adică după înlocuirea sa în faza de precompilare cu o secvenţă de cod corespunzătoare scopului urmărit) se comportă ca şi cum ar fi o funcţie declarată astfel:

```
void assert(int expresie);
```

La execuţie, **assert()** verifică dacă expresia este adevărată, iar dacă da nu se întâmplă nimic şi execuţia trece mai departe, iar dacă nu înseamnă că aserţiunea a picat şi atunci sistemul de operare opreşte programul şi afisează un mesaj de diagnostic al întreruperii.

5. Un exemplu de prelucrare de date

Prezentăm în continuare un exemplu de prelucrare a unor date numerice înregistrate într-un fişier pe disc. Am ales, pentru simplitate, lucrul în modul text cu acces secvential.

Este uşor de arătat că şirul (x_n), dat de recurenţa

$$x_{n+1}=4\,x_n\left(1-x_n\right),$$

rămâne în intervalul [0,1] dacă x_0 este din [0,1]. Comportarea şirului este foarte complexă şi se schimbă foarte mult de la o valoare la alta a datei iniţiale x_0, chiar pentru modificări mici ale acesteia. Dorim să ilustrăm prin rezultate numerice acest lucru: în programul următor calculăm, într-o primă etapă, primii `nrMax` termeni ai şirului, pornind de la un x_0 precizat în **main(),** şi îi scriem în fişierul `numere.txt` astfel:

```
x[0]=0.12345
x[1]=0.43284
x[2]=0.981958
.................
x[996]=0.410508
x[997]=0.967965
x[998]=0.124036
x[999]=0.434603
```

In etapa a doua afişăm pe monitor fişierul obţinut, pentru verificare. In final citim din nou fişierul `numere.txt` şi numărăm câte valori ale şirului sunt în fiecare dintre intervalele [0, 0.1), [0.1, 0.2), ... , [0.9, 1), şi afişăm această statistică.

```
#include<iostream>
#include<fstream>
#include<assert.h>          //pentru assert()
#include<stdlib.h>          //pentru atof()
using namespace std;

void pauza(void)
```

```
{
    char ch;
    cout << "\n\tPauza!\n\nPentru start tastati [ENTER]\n" << endl;
    cin.get(ch);                    //cin>>ch nu "simte" tasta ENTER
    cin.putback(ch);        //refacem buffer-ul tastaturii
    cin.ignore(4096, '\n'); //si apoi il golim
}

void creareFisier(char caleFis[], double x, int nrMax){
    ofstream xout(caleFis);
    assert(xout.good());
    cout << "Asteptati va rog!" << endl;
    for (int i = 0; i<nrMax; i++){
        xout << "x[" << i << "]=" << x << endl;
        x = 4.0*x*(1.0 - x);
    }
    cout << "Gata, am creat fisierul " << caleFis << endl;
    xout.close();
}


void afisareFisier(char caleFis[]){
    ifstream xin(caleFis);
    assert(xin.good());
    const int dim = 1024;
    char buffer[dim];
    cout << "Iata fisierul " << caleFis << ":" << endl;
    while (xin.good()){
        xin.getline(buffer, dim);
        cout << buffer << endl;
    };              //Atentie: la final avem un buffer gol!
    cout << "Gata, acesta a fost fisierul " << caleFis << endl;
    xin.close();
    return;
}
void consultareFisier(char caleFis[]){
    ifstream xin(caleFis);
    assert(xin.good());
    const int dim = 100;
    int i, j;
    double x;
    char buffer[dim], bufNum[dim];
    int cont[11] = {};
    while (xin.good()){
        xin.getline(buffer, dim);
        if (buffer[0] == '\0') break;     //nu analizam buffer-ul gol.
        for (i = 0; i<dim && buffer[i++] != '='; );
        for (j = 0; i + j<dim && (bufNum[j] = buffer[i + j]) != '\0'; j++);
        x = atof(bufNum);
        cont[(int)(10 * x)]++;
    }
    cout << "In fisier avem: " << endl;
    for (i = 0; i<10; i++) cout << cont[i] << " numere in [ " << i / 10.0
        << " , " << (i + 1) / 10.0 << " )" << endl;
    xin.close();
    return;
}
```

```
int main(){
    char nume[] = "numere.txt";
    int nrMax = 1000;
    double x0 = 0.12345;

    creareFisier(nume, x0, nrMax);
    pauza();

    afisareFisier(nume);
    pauza();

    consultareFisier(nume);
    return 0;

}
```

Rezultatul final:

```
In fisier avem:
217 numere in [ 0 , 0.1 )
81 numere in [ 0.1 , 0.2 )
81 numere in [ 0.2 , 0.3 )
60 numere in [ 0.3 , 0.4 )
60 numere in [ 0.4 , 0.5 )
62 numere in [ 0.5 , 0.6 )
74 numere in [ 0.6 , 0.7 )
76 numere in [ 0.7 , 0.8 )
92 numere in [ 0.8 , 0.9 )
197 numere in [ 0.9 , 1 )
Press any key to continue . . .
```

Pentru citirea datelor numerice dintr-un fişier text avem două strategii, prima: folosim operatorul de extracţie >> care ştie să interpreteze o secvenţă de caractere dintr-un stream de intrare ca o valoare numerică, şi a doua: folosim funcţiile de conversie **atof()** şi **atoi()** din **stdlib.h** care primesc ca date de intrare câte un **string**, îl citesc ca pe un număr şi returnează valoarea găsită, de tip **double** pentru **atof()** şi de tip **int** pentru **atoi()**. In program am ales a doua strategie, mai sigură: parcurgem fişierul linie cu linie utilizând **getline()** şi analizăm fiecare linie citită.

Prima metodă ar fi presupus să citim fişierul caracter cu caracter până la primul `'='`, apoi să citim cu >> valoarea numerică următoare, după care iar să citim caracter cu caracter până la primul `'='`, şi aşa mai departe. Lăsăm această cale ca exerciţiu.