

## Operații de intrare/ieșire în C++

În mod obișnuit, un program de calcul citește o serie de date de la tastatură sau de pe disc și le transferă în memoria internă alocată programului, le prelucrează pe baza setului de instrucțiuni - prin transferuri repetate între regiștrii de memorie și regiștrii micro-procesorului - și în final le scrie sub forma prelucrată pe monitor sau într-un fișier pe disc. Cu alte cuvinte, un program de calcul execută *operații de intrare* (input, read operations) prin *citirea* datelor, *operații de prelucrare* a datelor și *operații de ieșire* (output, write operations) prin *scrierea* datelor. Aceste operații nu se execută neapărat în ordinea dată aici, ele sunt intercalate și executate în mod repetat conform cu necesitățile problemei de rezolvat.

Modul de derulare a operațiilor de intrare/ieșire (*operații I/O*) este specific fiecărui dispozitiv hardware în parte, pentru a le executa sistemul de operare utilizează programe speciale (*drivere*) puse la dispoziție de producătorii dispozitivelor. Existența acestor drivere asigură un anumit grad de independență a sistemului de operare față de mașină: fără ele ar trebui proiectat câte un sistem de operare pentru fiecare configurație hardware în parte. Acesta este și motivul pentru care, atât în limbajul C inițial cât și în extensia sa C++, nu există instrucțiuni de intrare/ieșire: s-a evitat legarea limbajului de un anumit sistem de operare sau de o anumită configurație hardware. În limbajul C operațiile I/O se execută utilizând funcții de intrare/ieșire, iar în limbajul C++ utilizând clase de intrare/ieșire.

În C++ obiectele date de clasele specializate în operații I/O se numesc *fluxuri de date* sau *stream-uri*. Clasele de stream-uri nu aparțin limbajului, dar au un caracter standard și o largă circulație 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-și definească (prin derivare) propriile clase, după necesități. În MS Visual Studio, cele mai utilizate biblioteci I/O sunt `<iostream>`, în special pentru stream-urile predefinite `cin` și `cout` care sunt instanțe ale claselor `ostream`, respectiv `istream`, și `<fstream>`, pentru operații I/O cu fișiere stocate pe disc.

### 1. Generalități despre clase

Pentru înțelegerea utilizării claselor de intrare/ieșire 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`, permițând programatorului să înglobeze la un loc, pe lângă un set de date și o serie de funcții specializate în prelucrarea acestor date.

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

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

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

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 funcția **main()** de mai sus prezentăm trei modalități de instanțiere a clasei **Punct**. Declarația lui **a** cheamă constructorul clasei cu valorile implicite ale argumentelor, 13 și 13, **b** este inițializat prin apelarea “în clar” a constructorului, iar **c** este inițializat “în stil C++”, tot prin apelarea constructorului.

Spre deosebire de ceilalți membrii, **mesaj** este declarat **static**, aceasta înseamnă că toate instanțele clasei au aceeași valoare pentru membrul **mesaj**, din acest motiv el poate fi apelat la nivel de clasă, cu *operatorul de rezoluție ::* 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 eșecului.

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

```
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)
```

obținem rezultatele din comentarii deoarece cele două instrucțiuni sunt echivalente. Prin expresia `cout << c` compilatorul înțelege de fapt apelul de funcție `operator<<(cout, c)`, în care trimitem prin referință stream-ul **cout**, care este o instanță a clasei **ostream**, și pe care îl primim prin referință ca rezultat după terminarea apelului, fapt care ne permite să concatenăm operatorul <<. Alt exemplu de instrucțiuni echivalente:

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

Facem observația că în clasele de stream-uri operatorii de deplasare pe biți sunt supraîncărcați și, din acest motiv, au denumiri specifice: *operatorul de inserție* << și *operatorul de extracție* >>.

O clasă poate să ascundă sau nu anumiți membri, după cum îi declară cu modificatorii de acces **public**, **protected** sau **private**. Datele private sunt accesibile numai din interiorul clasei (numai funcțiile memru ale clasei le pot modifica, de exemplu), cele protejate sunt accesibile și 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 informației* în interiorul clasei astfel încât, pentru a utiliza o clasă dintr-o bibliotecă, de exemplu, programatorul trebuie să știe numai membrii la care are acces și ce anume fac ei, fără să cunoască cum sunt ei implementați 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*) moștenește membrii clasei inițiale (numită *clasă mamă*, *clasă părinte* sau *clasă de bază*) și poate adăuga noi membrii, sau îi poate redefini prin *suprascriere* pe cei moșteniți.

De exemplu, unui obiect din clasa **Punct** îi putem atașa 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 obișnuit:

```

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

```

O facilitate suplimentară pusă în ultimul timp la dispoziția programatorilor în limbajul C++ este *programarea generică*, prin utilizarea *șabloanelor 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 șablonul `PunctGeneric<>` în care tipul membrilor `x` și `y` nu este bine precizat, el va fi stabilit la instanțierea unui obiect din această clasă, printr-o declarație de formă:

```

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

```

În practică se folosește foarte des redenumirea cu **typedef** a diverselor *specializări* ale unui șablon: în urma instrucțiunii

```

typedef PunctGeneric<double> Puncttt;

```

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

```

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

```

Această facilitate este intens folosită în construcția bibliotecilor de stream-urilor, unde trebuie să avem atât fluxuri pentru caracterele clasice, pe un octet, adică de tip **char**, cât și, de exemplu, fluxuri pentru caracterele de tip *wide*, pe doi octeți. Este clar că funcționalitatea acestor stream-uri este foarte asemănătoare și utilizarea șabloanelor permite reutilizarea codului într-o proporție considerabilă. De exemplu, pentru stream-uri de ieșire este implementată clasa șablon:

```

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

```

și 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 și nu doar o specializare a unui șablon.

## 2. Stream-urile **cin** și **cout**

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

Operațiile I/O uzuale, cele care au loc pe ruta tastatură-program-monitor, sunt realizate cu stream-urile standard **cin/cout**, utilizând operatorii de inserție/extracție `<</>`. Nu este singura posibilitate, dar este cea mai ușoară, deoarece **cin/cout** sunt stream-uri predefinite: sunt gata declarate și inițializate în `<iostream>` ca instanțe ale claselor **istream/ostream**, au gata atașate fișierele logice **stdin/stdout** (consolele standard de intrare/ieșire, mai precis tastatura/monitorul), și au operatorii `<</>` supraîncărcați astfel încât să se descurce singuri cu datele din tipurile fundamentale (aritmetice sau pointer) și chiar cu unele tipuri compuse (șiruri de caractere). La rândul lor clasele **istream** și **ostream** sunt derivate din clasa **ios**, în consecință stream-urile **cin** și **cout** moștenesc și membrii acestei clase.

Observație: În fișierul header `<iostream>` se găsesc declarațiile

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

care certifică afirmațiile 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 moniotor va fi scris următorul fișier de ieșire:

```
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 și modificat cu metoda `precision()`, care fiind o funcție membru a clasei din care face parte stream-ul **cout** este indicată cu operatorul de selecție `."`. Reținem: instrucțiunea

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

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

Menționăm 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 aceeași problemă:

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

Să punem în evidență și unele limitări ale operatorilor de inserție/extracție `<</>>`. De exemplu, dacă vrem să aflăm adresele unor variabile cu secvența:

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

obținem

```
0x0012FF78
```

```
A|_|_|_|^L ↓
```

Explicație: `&ch` este o constantă de tip `char*` iar în cazul operandului de tip `char*` operatorul `<<` înțelege că are de pus în flux un șir de caractere, așa că în cazul nostru pornește de la adresa lui `ch` și scrie pe monitor toți octeții întâlniți în memorie până da peste unul nul. Dacă vrem să vedem totuși adresa variabilei `ch`, trebuie să folosim o conversie explicită către tipul `void*`. Secvența

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

are rezultatul scontat:

```
0x0012FF78
```

```
0x0012FF7C
```

```
Press any key to continue.
```

### 3. Citirea și scrierea fișierelor pe disc. Prezentare rapidă

Clasele I/O standard sunt astfel concepute încât lucrul cu fișierele stocate pe disc să decurgă în mod similar cu operațiunile I/O uzuale, deja cunoscute. Trebuie numai să ne alegem o clasă de stream-uri convenabilă, să instanțiem un obiect din acea clasă și să-i atașăm fișierul din care dorim să citim sau în care vrem să scriem. În rest, operațiile I/O propriu-zise decurg, cel puțin în cazul fișierelor text, ca în cazul lucrului cu `cin/cout`.

Începem cu următorul exemplu: scriem un text în fișierul "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 fără erori, pe monitor apare mesajul “Am scris.” iar în dosarul proiectului găsim fișierul `felicitare.txt` pe care îl putem citi cu orice editor de texte (cu notepad-ul, de exemplu). O singură precizare: în instrucțiunea

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

declarăm și inițializăm un stream din clasa **ofstream** (deci un output-file-stream) pe care l-am denumit **xout** și căruia îi atașăm fișierul `felicitare.txt` care va fi astfel fișierul de ieșire al stream-ului.

Acum dorim să citim printr-un program C++ fișierul creat și să îl afișăm 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 atașat fișierul `felicitare.txt` care acum are rolul de fișier de intrare, am citit fișierul caracter cu caracter și l-am afișat pe monitor prin stream-ul **cout**. Rezultat: `Satraitibine!` Stim că textul a fost scris corect în `felicitare.txt`, am verificat cu notepad-ul, deci greșeala este la citire. În general citirea datelor este mai dificilă decât scrierea lor. Pentru operatorul de extracție `>>`, *spațiile albe* (blanc-ul, tab-urile, caracterul new-line, carriage return) joacă rolul de *delimitatori* de câmpuri de citire, și sunt utile când citim o serie de string-uri sau de numere, la citirea “caracter cu caracter” operatorul “`>>`” extrage spațiile albe din buffer-ul atașat stream-ului dar nu le returnează programului apelant, sare pur și simplu peste ele!

Corectăm ultimul exemplu: renunțăm la `>>` și apelăm la funcția membru **get()**. Stream-ul nostru, denumit de noi **xin**, fiind un obiect din clasa **ifstream**, moștenește de la clasa mamă **istream** funcția **get()** sub două forme: **int get()**, care extrage orice caracter din buffer și întoarce codul ASCII al caracterului citit, și **istream& get(char&)**, care, fiind apelată prin referință, încarcă direct în parametrul actual (de tip `char`) caracterul citit și întoarce o referință la stream-ul apelant (și 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 sfârșitul oricărui fișier 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 *condiție de stare* a stream-ului (dacă a ajuns sau nu la sfârșitul fișierului):

```

#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;
}

```

Renunțarea la operatorul de extracție >> nu s-a datorat lucrului cu fișiere pe disc ci limitărilor acestuia, limitări care apar și în cazul stream-urilor **cin/cout**. De exemplu, în urmatorul program reușim cu greu să încercăm în tabloul de caractere sText, dintr-o singură citire, atât numele cât și prenumele utilizatorului. La prima încercare, operatorul >> citește numai numele, prenumele nu ajunge în sText, mai mult, rămâne în buffer-ul tastaturii și trebuie șters 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;
}

```



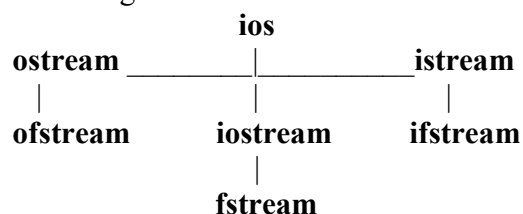
Stream-ul predefinit **cin** moștenește de la clasa **istream** două funcții extrem de utile: **getline()** și **ignore()**. Apelul `cin.getline(sText, dimText)` citește din fișierul atașat stream-ului până la primul '\n' (delimitator implicit ) sau până citește `dimText-1` caractere și apoi le depune în `sText`, adăugând, dacă mai este cazul, terminatorul de string '\0'. Apelul `cin.ignore(4096, '\n')` extrage și aruncă la gunoi toate caracterele întâlnite până la primul '\n' (în acest caz) sau până la al 4096-lea caracter (în mod obișnuit buffer-ul atașat unui stream are 4 Ko).

În concluzie, operațiile de intrare/ieșire în C++ sunt simple și eficiente, dar pentru utilizarea lor cu acuratețe este necesară o bună cunoaștere a ierarhiei claselor de stream-uri și a metodelor publice ale acestora.

#### 4. Citirea și scrierea fișierelor pe disc. Prezentare în amănunt

Orice operație I/O se execută prin intermediul unui stream, fiecare stream având atașat câte un singur fișier. Așa cum am mai spus, un stream poate fi înțeles ca fiind un canal de comunicație prin care trec datele într-un flux cu o direcție bine precizată, având la un capăt programul nostru iar la celălalt capăt un fișier pe post de *rezervor de date*. Datele circulă în pachete formate din octeți (*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 așteptare”, 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).

În funcție de tipul operațiilor pe care le pot efectua, avem *stream-uri de intrare* (clasele **istream**, **ifstream**, ș.a.), *stream-uri de ieșire* (clasele **ostream**, **ofstream**, ș.a.) și *stream-uri de intrare-ieșire* (clasele **iostream**, **fstream** ș.a.). Printr-un stream de intrare (*input stream*) se poate efectua numai citirea datelor din fișierul atașat, printr-unul de ieșire (*output stream*) numai scrierea lor, iar printr-un stream de intrare-ieșire se pot executa ambele operații asupra aceluiași fișier. Clasele menționate sunt derivate din clasa **ios** conform următoarei diagrame:



Într-un program putem declara și 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 fișier se execută în trei pași: *deschiderea* fișierului, *prelucrarea* sau *utilizarea* fișierului și *închiderea* lui. Deschiderea fișierului se face odată cu asignarea (asocierea, atribuirea) lui unui stream. Un stream poate să aibă asignat un singur fișier la un moment dat.

Deschiderea și asignarea unui fișier 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: instrucțiunea

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

este echivalentă cu secvența

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

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

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

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

Inchiderea fișierului constă în completarea transmiterii tuturor datelor (prin golirea buffer-ului atașat stream-ului) și, eventual, din descărcarea fișierului din memorie și rescrierea lui pe disc. Subliniem că operația de închidere are efect numai asupra fișierului atașat, nu șterge și stream-ul, acesta poate fi eventual refolosit pentru deschiderea altui fișier. Inchiderea trebuie cerută explicit de programator, altfel fișierul este închis de sistemul de calcul la terminarea execuției programului (dacă execuția s-a terminat cu bine).

Un exemplu corect de lucru cu fișiere:

```
#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;
}

```

obținem:

Press any key to continue

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

Asignarea aceluiași fișier la două stream-uri diferite nu este interzisă dar este un mod de operare impropriu, dacă într-un program dorim să scriem și să citim din același fișier, ori îl deschidem cu un singur stream de intrare-ieșire (din clasa **fstream**), ori îl prelucrăm succesiv cu două streamuri uni-direcționale (unul de intrare și altul de ieșire) deschizând și închizând fișierul ori de câte ori este nevoie.

Un fișier poate fi *creat*, *rescris*, *completat*, *modificat* sau *citit*, iar deschiderea fișierului este însoțită, explicit sau nu, de precizarea *modului de deschidere* a fișierului, mod care stabilește operațiile de prelucrare la care va fi supus fișierul precum și 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
<b>ios::app</b>	Deschide un fișier de ieșire pentru completare (appending). Dacă fișierul nu există, este creat pe loc. Datele sunt scrise numai pe ultimul loc.
<b>ios::ate</b>	Deschide un fișier existent (pentru citire sau/și scriere) și poziționează prompterul pe ultimul rând (at end). Primul octet este scris pe ultimul loc, următorii sunt scrisi acolo unde este poziționat prompterul.
<b>ios::in</b>	Deschide un fișier pentru citire. Poate fi utilizat și cu un <b>ofstream</b> , pentru a preveni trunchierea unui fișier deja existent.
<b>ios::out</b>	Deschide un fișier pentru scriere. Dacă fișierul nu există, este creat pe loc. Dacă fișierul există deja, iar <b>ios::out</b> este folosit fără modulele <b>ios::app</b> , <b>ios::ate</b> sau <b>ios::in</b> , atunci este implicat modul

	<b>ios::trunc</b> (fișierul este curățat și prompterul este poziționat la start).
<b>ios::trunc</b>	Deschide un fișier nou și îl șterge (curăță) pe cel vechi (dacă acesta deja există).
<b>ios::binary</b>	Deschide un fișier 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 fișierul în modul ios::out, iar
ifstream fluxin("date.txt");
```

îl deschide în modul **ios::in**. Pentru a deschide fișierul în mai multe moduri simultan, se utilizează operatorul *sau pe biți* “|”, conform următoarelor exemple:

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

Fișierul `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 conținut să se păstreze (în modul implicit de deschidere are loc *rescrierea* fișierului: înainte de a scrie în fișier vechile date sunt șterse).

Toate fișierele sunt formate din biți grupați câte opt în octeți. Unitatea de măsură pentru lungimea unui fișier sau pentru poziționarea prompterului I/O este octetul.

Fișierele sunt de două tipuri: *text* sau *binar*, după cum octeții care îi compun au sau nu semnificația 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 funcțiilor **read** și **write**, care, spre deosebire de operatorii de inserție/extracție, sunt indiferente la eventuala semnificație de caractere de control a octeților manipulați. Fișierele binare sunt mai compacte, controlul poziționării prompterului de acces este exact, dar informația stocată în ele poate fi regăsită numai dacă se cunoaște riguros modul în care au fost organizate datele la crearea fișierului. Fișierele binare pot fi citite, în mod uzual, numai de programul care le-a creat.

Fișierele 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 adaugă “spații”), sfârșitul de fișier este marcat printr-un caracter special, caracterul “**eof**”. În raport cu fișierele binare, fișierele text fac o oarecare risipă de spațiu de stocare (nu prea mare), mai mult, controlul prompterului este dificil, deoarece poziționarea sa depinde de diverse convenții de scriere. Avantajul major al fișierelor text constă în faptul că informația conținută în ele poate fi regăsită cu orice program de editare de texte.

Consola de intrare (fișierul logic asociat tastaturii) și consola de ieșire (monitorul) sunt fișiere text, deci **cin/cout** operează numai în modul text (și numai secvențial, vezi mai jos).

Accesarea datelor unui fișier poate fi făcută în mod *secvențial* sau în mod *aleator* (cu sensul: “pe sărite”). Accesarea secvențială (citirea de la început până la sfârșit, scrierea numai pe “ultimul rând”) este utilizată mai ales în cazul fișierelor text, și are loc de la sine atât timp cât programatorul nu intervine în poziționarea prompterului I/O. Accesul

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

Fiecare stream are un *prompter* I/O, adică o variabilă de tip **int** care conține poziția de scriere/citire curentă din fișier. Prompterul I/O mai este numit și *pointer* I/O, deoarece valoarea sa are semnificația unei adrese. În cazul stream-urilor de intrare prompterul este numit “get pointer”, poziția sa poate fi aflată cu funcția membru **tellg()** și poate fi poziționat cu **seekg()**, în cazul stream-urilor de ieșire avem un “put pointer” și funcțiile membru **tellp()** și **seekp()**. Prompterul unui stream de intrare-ieșire este numit și “get pointer” și “put pointer” după cum îl utilizăm pentru citire sau pentru scriere, poziționarea lui fiind făcută cu funcțiile corespunzătoare întrebunțării. Trebuie reținut că, în acest caz, orice poziționare cu **seekg()**, de exemplu, modifică și valoarea întoarsă de **tellp()**, deoarece stream-ul are numai un singur prompter!

Succesul unei operații de intrare/ieșire depinde de foarte mulți factori care apar în cursul rulării și asupra cărora programatorul nu poate interveni: dacă fișierul există sau nu acolo unde trebuie, dacă utilizatorul are sau nu dreptul să scrie pe disc, dacă este suficient spațiu de stocare, etc. Programatorul trebuie să testeze mereu, prin program, dacă operația I/O efectuată a eșuat sau nu.

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

Funcția	Valoare returnată: FALSE / TRUE ( zero / diferit de zero)
<b>bad()</b>	întoarce <b>TRUE</b> dacă a apărut o eroare I/O nerecuperabilă, de exemplu: o tentativă de scriere/citire dincolo de sfârșitul fișierului.
<b>fail()</b>	întoarce <b>TRUE</b> dacă a apărut o eroare I/O nerecuperabilă sau dacă o operație de deschidere nu a avut succes.
<b>eof()</b>	întoarce <b>TRUE</b> dacă a fost atins sfârșitul de fișier .
<b>good()</b>	întoarce <b>TRUE</b> dacă nici una din condițiile de mai sus nu este adevărată.

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

Imediat după deschiderea unui fișier, de exemplu, trebuie să testăm succesul operației printr-o instrucțiune 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 opoziție cu operatorul !, deci

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

este echivalentă cu

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

dar nu este echivalentă cu

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

deoarece `xout` nu testează și indicatorul `eof`. Recomandăm utilizarea acestei ultime forme pentru testarea “sănătății” stream-ului.

Pentru diagnosticarea erorilor de intrare/ieșire apărute și întreruperea executării programului vom folosi, în exemplele care urmează, fișierul header `assert.h`, fișier în care este definit macroul `assert()`. Acest macro, după expandare (adică după înlocuirea sa în faza de precompilare cu o secvență de cod corespunzătoare scopului urmărit) se comportă ca și cum ar fi o funcție declarată astfel:

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

La execuție, `assert()` verifică dacă expresia este adevărată, iar dacă da nu se întâmplă nimic și execuția trece mai departe, iar dacă nu înseamnă că aserțiunea a picat și atunci sistemul de operare oprește programul și afișează 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 fișier pe disc. Am ales, pentru simplitate, lucrul în modul text cu acces secvențial.

Este ușor de arătat că șirul  $(x_n)$ , dat de recurența

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

rămâne în intervalul  $[0,1]$  dacă  $x_0$  este din  $[0,1]$ . Comportarea șirului este foarte complexă și se schimbă foarte mult de la o valoare la alta a datei inițiale  $x_0$ , 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 șirului, pornind de la un  $x_0$  precizat în `main()`, și îi scriem în fișierul `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
```

În etapa a doua afișăm pe monitor fișierul obținut, pentru verificare. În final citim din nou fișierul `numere.txt` și numărăm câte valori ale șirului sunt în fiecare dintre intervalele  $[0, 0.1)$ ,  $[0.1, 0.2)$ , ...,  $[0.9, 1)$ , și afișăm 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 fișier text avem două strategii, prima: folosim operatorul de extracție >> care știe să interpreteze o secvență de caractere dintr-un stream de intrare ca o valoare numerică, și a doua: folosim funcțiile de conversie **atof()** și **atoi()** din **stdlib.h** care primesc ca date de intrare câte un **string**, îl citesc ca pe un număr și returnează valoarea găsită, de tip **double** pentru **atof()** și de tip **int** pentru **atoi()**. În program am ales a doua strategie, mai sigură: parcurgem fișierul linie cu linie utilizând **getline()** și analizăm fiecare linie citită.

Prima metodă ar fi presupus să citim fișierul caracter cu caracter până la primul '=', apoi să citim cu >> valoarea numerică următoare, după care iar să citim caracter cu caracter până la primul '=', și așa mai departe. Lăsăm această cale ca exercițiu.