Capitolul 13

11 Ierarhia claselor pentru operațiile de intrare/ieşire

Această ierarhie reprezintă o extensie a noțiunii de `stream` utilizată în limbajul C (structura `FILE`), dar adaptată la paradigma programării orientate pe obiecte. Un `stream` reprezintă un flux de octeți prin care un program se poate conecta la un dispozitiv periferic şi poate comunica cu acesta.

Ca şi structura `FILE` şi funcțiile `printf` şi `scanf` ale limbajului C, ierarhia de stream-uri de intrare/ieşire nu face parte din sintaxa limbajului C++, ci din biblioteca standard a limbajului, fiind de asemenea prevăzută în standardul ANSI C++.

Conform acestui standard, fişierele header utilizate pentru biblioteca limbajului C++ nu mai au extensia "`.h`". Pentru a putea păstra compatibilitatea cu programele vechi C++, compilatoarele acestui limbaj permit utilizarea ambelot tipuri de fişiere header. De exemplu, într-un program C++, se poate scrie atât:

```
#include <iostream>
```

cât şi:

```
#include <iostream.h>
```

Între cele două categorii de fişiere există însă câteva deosebiri importante:

a) Declarațiile din fişierele cu extensia nu sunt incluse "`.h`" în spațiul standard al numelor `std`, astfel încât atunci câd sunt utilizate, ele apar în spațiul global al unui program. Spre deosebire de acestea, toate declarațiile din fişierele header fără extensia "`.h`" sunt include în spațiul `std`.
b) Marea majoritate a declarațiilor din fişierele fără extensia "`.h`" sunt în realitate clase parametrizate prin mecanismul `template`. Pentru a păstra compatibilitatea cu declarațiile din fişierele celuilalt tip, sunt utilizate construcții `typedef` pentru a asocia nume unor instanțe ale acestor clase.

De exemplu, clasele parametrizate `basic_ios` şi `basic_string` sunt definite în fişierele header `<ios>` şi `<string>`. Pentru compatibilitate cu vechile fişiere header, se utilizează următoarele construcții:

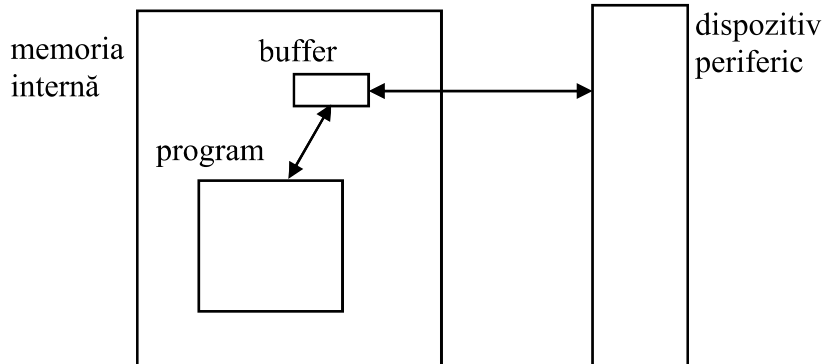
```
typedef basic_ios<wchar_t> wios;
typedef basic_ios<char> ios;
typedef basic_ios<char> string;
```

În acest mod, tipurile de date `ios` şi `string` devin compatibile cu cele definite în vechile fişiere `<ios.h>` şi `<string.h>`.

Un avantaj al utilizării mecanismului `template` în ierarhiile de clase de intrare/ieşire constă în faptul că aceste clase pot fi utilizate atât pentru manipularea stream-urilor în memoria internă, în cadrul fişierelor şi al terminalelor standard de intrare/ieşire.

11.1 Structura ierarhiei de clase de intrare/ieşire

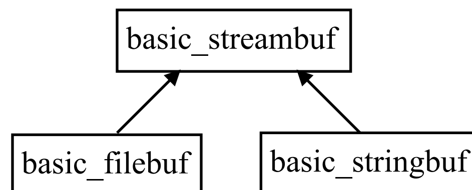
Un ***stream*** este o cale de comunicare între un program şi un dispozitiv periferic. Comunicarea cu dispozitivul periferic se realizează prin intermediul unui ***buffer***, o zonă din memoria internă conectată în mod logic la dipozitiv.



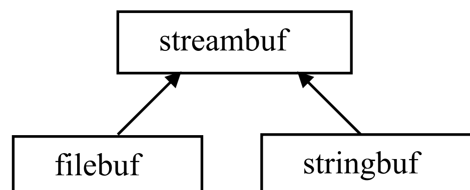
În mod uzual, transferul de date între program şi buffer este vizibil în cadrul programului, pe când cel între buffer şi dispozitivul periferic este transparent, el fiid realizat de task-uri specifice ale sistemului de operare.

Plecând de la acest mod de abordare, limbajul C++ pune la dispoziție două ierarhii de clase : una este asociată operațiilor specifice unui buffer şi este reprezentată de clasa `streambuf`, iar cealaltă este asociată operațiilor de transfer de date între program şi buffer şi este reprezentată de clasa `ios`.

Ierarhia `streambuf` are forma următoare:

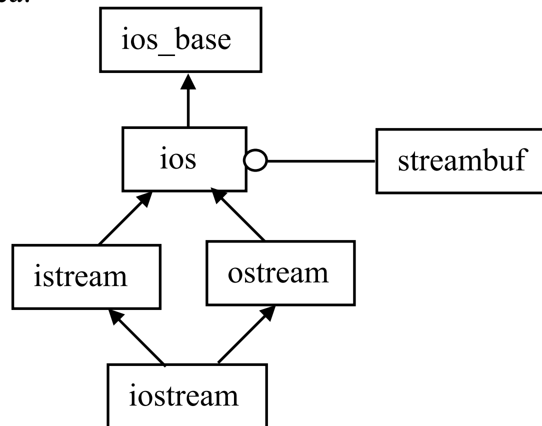


Observație. Clasele `template` din ierarhia de intrare/ieşire prmit definirea stream-urilor privite ca şiruri de caractere, atât de tipul `char`, cât şi de tipul `wchar_t`. Deoarece în mod uzual se utilizează şiruri de caractere de tipul `char`, în continuare se vor utiliza numele claselor în maniera clasică, adică fără prefixul “`basic_`” (prefixul “`basic_`” este utilizat doar pentru definirea claselor `template`):

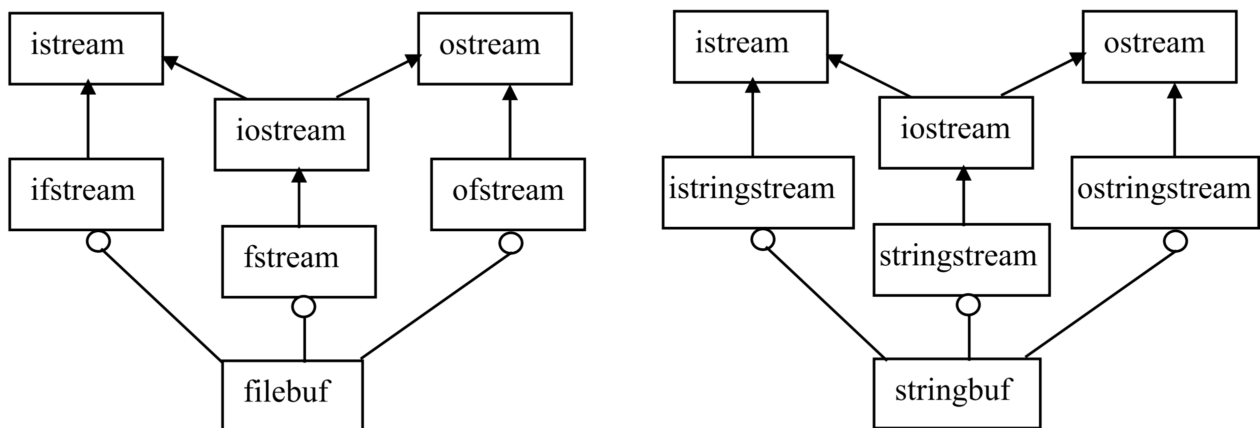


Clasa `streambuf` este asociată bufferelor pentru stream-urile care citesc şi scriu date din/în terminalele standard de intrere/ieşire, clasa `filebuf` este asociată bufferelor pentru stream-urile care citesc şi scriu date din/în fişiere, iar clasa `stringbuf` este asociată bufferelor pentru stream-urile care citesc şi scriu date din/în memorie.

Ierarhia de clase aferentă stream-urilor care citesc şi scriu date din/în terminalele standard de intrere/ieşire este următoarea:



Din ierarhia `iostream` sunt derivate şi celelalte două ierarhii de stream-uri: asociate operaţiilor cu fişiere (ierarhia `fstream`) şi cu string-uri (ierarhia `stringstream`):



Principalele fişiere header ce pot fi utilizate în cadrul programelor sunt următoarele:

- `<istream>` : folosit în cazul în care se utilizează clasa `istream`;
- `<ostream>` : folosit în cazul în care se utilizează clasa `ostream`;
- `<iostream>` : folosit în cazul în care se utilizează clasele `iostream`, `istream` sau `iostream`;
- `<iomanip>` : folosit în cazul în care se utilizează manipulatori;
- `<streambuf>` : folosit în cazul în care se utilizează clasa `streambuf`;
- `<fstream>` : folosit în cazul în care se utilizează clase din ierarhia `fstream`;
- `<sstream>` : folosit în cazul în care se utilizează clase din ierarhia `stringstream`;
- `<iosfwd>` : folosit în cazul în care se utilizează declaraţii de tip `forward` pentru clasele din ierarhiile de intrare/ieşire.

Datorită utilizării claselor parametrizate, declaraţii de forma următoare sunt eronate:

```
class istream;
void f(istream& istr);
```

În locul acesteia se poate utiliza fişierul header `<iosfwd>`, care are o dimensiune mult mai mică decât `<iostream>`:

```
#include <iosfwd>
using namespace std;
void f(istream& istr);
```

Biblioteca C++ conţine, ca şi în cazul limbajului C, câteva obiecte predefinite. Ele sunt definite în fişierul header `<iostream>` şi sunt utilizate pentru operaţiile cu terminalele standard de intrare şi ieşire:

- `cin` : obiect instanţă al clasei `istream`, utilizat pentru operaţiile de citire de la terminalul standard de intrare;
- `cout` : obiect instanţă al clasei `ostream`, utilizat pentru operaţiile de scriere la terminalul standard de ieşire;
- `cerr` : obiect instanţă al clasei `ostream`, utilizat pentru operaţiile de afişare a erorilor din programe;

11.2 Ierarhia `streambuf`

Indiferent de tipul de stream utilizat, comunicarea cu dispozitivul periferic asociat stream-ului se realizează prin intermediul unui buffer. Acest buffer este o instanţă a clasei `streambuf`, sau a uneia derivată din ea.

Obiectele de tip `streambuf` sunt create, utilizate şi distruse în cadrul stream-urilor, ele nefiind accesibile în mod direct în cadrul programelor. Deşi clasa `streambuf` nu are constructori publici, există o serie de funcţii publice ce pot fi accesate în cadrul programelor, dar în mod indirect, prin intermediul stream-urilor.

Principalele funcţii publice sunt prezentate în continare (tipul `streamsize` poate fi considerat ca fiind `unsigned int`).

A) Pentru operaţiile de intrare:

- `streamsize streambuf::in_avail()` : returnează numărul de caractere ce pot fi citite imediat;
- `int streambuf::sbumpc()` : returnează următorul caracter disponibil sau EOF; acest caracter este eliminat din buffer; dacă în buffer nu mai există caractere disponibile, se citesc noi caractere de la dispozitivu periferic;
- `int streambuf::sgetc()` : returnează următorul caracter disponibil sau EOF; caracterul nu este eliminat din buffer;
- `int streambuf::sgetn(char* buff, streamsize n)` : citeşte *n* caractere din bufferul de intrare şi le memorează în zona de memorie indicată de *buff*;
- `int streambuf::snextc()` : elimină caracterul curent din bufferul de intrare şi returnează caracterul imediat următor sau EOF;
- `int streambuf::sputback(char c)` : inserează *c* în bufferul de intrare ca primul caracter ce urmează să fie citit;
- `int streambuf::sungetc()` : returnează ultimul caracter citit din buffer, ca fiind primul caracter ce va fi citi de următoarea operaţie de citire;

B) Pentru operațiile de ieşire:

- `int streambuf::pubsync()` : sincronizează bufferul, prin scrierea informațiilor din buffer în dispozitivul peroferic;
- `int streambuf::sputc(char c)` : inserează caracterul *c* în buffer; dacă bufferul este plin, se scriu caracterele din buffer în dispozitivul periferic;
- `int streambuf::sputn(char* buff, streamsize n)` : inserează *n* caractere din zona de memorie indicată de *buff* în bufferul de ieşire; se returnează numărul de caractere scrise în buffer.

Accesul la bufferul asociat unui stream se poate realiza prin intermediul unor funcții publice ale clasei `ios`, care păstrează un pointer la obiectul `streambuf` aferent. Se pot utiliza două variante supraâncărcate ale funcției `rdbuf`:

- `streambuf* streambuf::rdbuf()` : returnează pointerul la obiectul `streambuf` aferent;
- `streambuf* streambuf::sputn(streambuf* new)` : în acest caz se asociază obiectului `ios` curent un nou buffer specificat ca parametru; funcția returnează un pointer spre bufferul original al obiectului `ios`.

Observație. În cazul ultimei funcții, obiectul `streambuf` inițial al stream-ului nu mai este distrus de către stream când acesta iese din domeniul de vizibilitate; distrugerea lui trebuie realizată în mod explicit de către programator.

În mod normal, ultima variantă a funcției `rdbuf` se utilizează atunci când se doreşte redirectarea unui stream spre alt stream. În acest caz este indicată refacerea inițiale a stream-urilor, aşa încât distrugerea obiectelor `streambuf` asociate să poată fe efectuată automat de către stream-urile respective.

Exemplu. Crearea unui fişier de tip log pentru erorile unui program. Se utilizează un stream de ieşire asociat unui fişier log. Stream-ul predefinit `cerr` este redirecționat spre stream-ul fişierului log, astfel încât toate mesajele de eroare se vor scrie în fişier.

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

void main() {
  ofstream errlog;
  streambuf* cerr_buff = 0;
  // Se deschide fisierul asociat stream-ului errlog
  errlog.open("ferr.log");
  // Redirectionarea lui cerr spre errlog.
  // Memorarea adresei bufferului asociat lui cerr
  cerr_buf = cerr.rdbuf(errlog.rdbuf());
  cerr << "Primul mesaj de eroare\n";
  cerr << "Al doilea mesaj de eroare\n";
  // Refacerea directionarii
  cerr.rdbuf(cerr_buf);
}
```

11.3 Clasele de bază ale ierarhie `iostream`. Operații de formatare.

Clasa de bază a acestei ierarhii este `ios_base`, care permite definirea şi inspectarea stării stream-urilor, precum şi majoritatea facilităților de formatare. Această clasă nu permite crearea în mod direct a obiectelor instanță.

Următoarea clasă este `ios`, iar prin intermediul ei toate celelalte clase ale ierarhiei moştenesc proprietățile clasei `ios_base`. Principalul rol al clasei `ios` este adăugarea şi gestionarea bufferului de tip `streambuf`.

În continuare se prezintă principalele proprietăți ale celor două clase, raportate însă la clasa `ios`.

11.3.1 Starea stream-urilor

În cazul în care o anumită operație de intrare/ieşire a eşuat, toate operațiile următoare sunt suspendate. În aceste cazuri se poate inspecta starea stream-ului respectiv şi eventual repara eroarea.

Starea unui stream este specificată printr-un număr de cifre binare, numite ***indicatori de condiție***. Semnificația lor este următoarea:

- `ios::badbit` – s-a detectat o operație ilegală de intrare/ieşire;
- `ios::eofbit` – s-a detectat sfârşitul de fişier;
- `ios::failbit` – operația curentă a eşuat;
- `ios::goodbit` – operația curentă s-a terminat cu succes;

Pentru determinarea şi gestionarea stării unui stream (a unui obiect `ios` sau derivat din acesta), pot fi utilizate diferite funcții publice ale clasei `ios`. Principalele funcții de inspectare sunt:

- `ios::bad()`
- `ios::eof()`
- `ios::fail()`
- `ios::good()`

Principalele funcții ce pot fi utilizate pentru gestionarea stării unui stream:

- `ios::rdstate()`

Funcția returnează valorile curente ale indicatorilor de condiție. Pentru testarea valorii unui anumit indicator se poate utiliza operatorul '`&`'. De exemplu:

```
if (myStream.rdstate() & ios::good)
{
  // codul in cazul in care nu s-a detectat eroare
}
```

- `ios::setstate(int flags)`

Funcția settează anumite valori dorite ale indicatorilor de condiție.

- `ios::clear()`

Funcția resetează starea de eroare a unui stream. Ea se utilizează în mod uzual după detectarea unei erori, conform algoritmului:

```
dacă *s-a detectat eroare* atunci
|  *se repară eroarea*
|  ios::clear()
└
```

Observație. Obiectele `ios` pot fi utilizate în cadrul expresiilor condiționale. În acest caz se returnează `true` dacă funcția `ios::good()` are o valoare diferită de zero şi `false` altfel. De exemplu, următoarea secvență este corectă, deoarece obiectul `cin` este de asemenea un obiect de tipul `ios`:

```
cin >> x;
if (cin)
  cout << "Citire corecta\n";
else {
  cout << "Citire eronata\n";
```

11.3.2 Formatarea operațiilor de intrare/ieşire

Informația poate fi scrisă în stream-uri sau citită din acestea în două moduri: ***în mod binar*** sau ***formatată*** (în mod text). Formatarea informației presupune anumite conversii între formatul intern de reprezentare şi cel extern, asemănătoare celor realizate de funcțiile `printf` şi `scanf`.

Spre deosebire de funcțiile `printf` şi `scanf`, care utilizează specificatori de format pentru controlul operațiilor de conversie, în cazul stream-urilor formatarea este controlată prin intermediul unor ***indicatori de format***. Modificarea şi gestionarea acestora se poate face în două moduri:

- cu ajutorul unor funcții publice ale clasei `ios`;
- cu ajutorul unor funcții ***manipulator***, care sunt inserate direct în cadrul stream-urilor.

A) Indicatori de format

Conversia fiecărei valori scrise sau citite se realizează în conformitate cu valorile curente ale ***indicatorilor de format***. Aceştia sunt elemente ale unui tip enumerat, `_Fmtflags`, care au valori predefinite. Setarea sau resetarea lor se realizează prin intermediul unor măşti de biți.

Există trei indicatori de tip mască, `ios::adjustfield`, `ios::basefield` şi `ios::floatfield`, care se utilizează în combinație cu alți indiactori. Ceilalți indicatori definesc o singură operație de formatare.

Principalii indicatori de format sunt următorii:

- `ios::adjustfield` – este o valoare de tip mască care se utilizează în combinație cu un indicator ce defineşte modul de aliniere al caracterelor dintr-un câmp de afişare (`ios::right`, `ios::left`, `ios::internal`); este folosit pentru operații de

scriere, în cazul în care dimensiunea valorii de afişat este mai mică decât dimensiunea câmpului; caracterul utilizat pentru completarea zonei libere se numeşte *caracter de umplere* (în mod implict spaţiu);

- `ios::internal` – adaugă caractere de umplere între semnul minus al numerelor negative şi numărul propriu-zis;
- `ios::left` – specifică alinierea la stânga a caracterelor într-un câmp de afişare;
- `ios::right` – specifică alinierea la dreapta a caracterelor într-un câmp de afişare;
- `ios::basefield` – este o valoare de tip mască care se utilizează în combinaţie cu un indicator ce defineşte baza de numeraţie a cifrelor valorilor întregi care se afişează (`ios::dec`, `ios::hex`, `ios::oct`); este folosit pentru operaţii de scriere;
- `ios::dec` – specifică baza 10;
- `ios::hex` – specifică baza 16;
- `ios::oct` – specifică baza 8;
- `ios::floatfield` – este o valoare de tip mască care se utilizează în combinaţie cu un indicator ce defineşte modul de afişare al valorilor reale (`ios::fixed`, `ios::scientific`); este folosit pentru afişarea valorilor reale;
- `ios::fixed` – afişare cu punct zecimal;
- `ios::scientific` – afişare cu format exponenţial;
- `ios::boolalpha` – permite afişarea valorilor booleene sub forma unor şiruri de caractere (`true` şi `false`); în mod implicit acest indicator nu este setat;
- `ios::showpos` – permite afişarea caracterului + în faţa valorilor pozitive;
- `ios::skipws` – este un indicator utilizat pentru operaţii de citire, care în mod implicit este setat; are ca efect ignorarea spaţiilor goale (caracterele `TAB`, `BLANK`, `NL`, etc.) până la începutul următorului câmp de intrare;
- `ios::unitbuf` – permite golirea bufferului de ieşire după fiecare operaţie de scriere;

B) Funcţii publice care permit gestionarea operaţiilor de formatare

Acestea sunt funcţii membru ale clasei ios şi permit inspectarea şi modificarea indicatorilor de format. În continuare se prezintă principalele funcţii ce se pot utiliza.

B1) Funcţii de inspectare

- `ios::fill()` – returnează caracterul curent de umplere;
- `ios::flags()` – returnează mulţimea curentă a indicatorilor de format; pentru inspectarea unui anumit indicator se poate utiliza operatorul '`&`', ca în exemplul următor:

```
if (cout.flags() & ios::hex)
{
  // afisarea valorilor intregi in baza 16
}
```

- `ios::precision()` – returnează o valoare de tip `int`, reprezentând numărul de cifre al părţii fracţionare utilizate pentru afişarea valorilor reale (valoarea implicită este 6);

- `ios::with()` – returnează o valoare de tip `int`, reprezentând numărul de caractere al câmpului de afişare pentru afişarea unei valorilor numerice; valoarea implicită este 0, ceea ce înseamnă că se alocă atâtea caractere câte sunt necesare pentru afişarea valorii respective;

B2) Funcții de modificare

În continuare tipul `fmtflags` poate fi considerat ca fiind `long int` şi reprezintă şablonul pentru indicatorii de format.

- `ios::fill(char padding)` – defineşte un nou caracter de umplere şi îl returnează pe cel vechi;
- `ios::flags(fmtflags flagset)` – modifică valorile curente ale indicatorilor de format şi returnează vechile valori;
- `ios::precision(int signif)` – redefineşte numărul cifrelor părții fracționare pentru afişarea valorilor reale şi returnează vechiul număr;
- `ios::with(int n)` – redefineşte numărul de caractere al câmpului de citire pentru următoarea operație de citire şi returnează lungimea ultimului câmp de afişare; funcția nu are efect decât pentru câmpuri numerice;
- `ios::setf(fmtflags flags)` – setează unul sau mai mulți indicatori de format (folosind operatorul '`|`') şi returnează vechile valori ale tuturor indicatorilor;
- `ios::setf(fmtflags flags, fmtflags mask)` – resetează toți indicatorii de format specificați în parametrul *mask*, setează indicatorii specificați în parametrul *flags* şi returnează vechile valori ale tuturor indicatorilor; funcția se utilizează în mod uzual pentru indicatorii de format de tip mască (`ios::adjustfield`, `ios::basefield` şi `ios::floatfield`), ca în exemplele următoare:

```
// seteaza alinierea la stanga
setf(ios::left, ios::adjustfield);
// seteaza afisarea hexazecimala
setf(ios::hex, ios::basefield);
// seteaza afisarea cu virgula fixa
setf(ios::fixed, ios::floatfield);
```

- `ios::unsetf(fmtflags flags)` – resetează indicatorii de format specificați şi returnează vechile valori ale tuturor indicatorilor;

C) Funcții manipulator

Funcțiile manipulator reprezintă o altă variantă de control a indicatorilor de format. Principala deosebire dintre acestea şi cele descrise anterior o reprezintă modul de utilizare.

O funcție manipulator este inserată direct într-un stream de ieşire sau extrasă dintr-un stream de intrare prin intermediul operatorilor de inserare (<<) şi extragere (>>) despre care se va discuta ulterior. Principalul avantaj al utilizării acestora constă în faptul că nu este necesară o instrucțiune separată pentru controlul următoarelor câmpuri de citire sau scriere.

Ierarhia `iostream` conține o mulțime de manipulatori predefiniți, însă asemenea funcții pot fi definite şi de către programatori.

În general manipulatorii afectează direct indicatorii de format. Cei mai mulți dintre aceştia nu au parametri şi nu este necesar să se specifice parantezele de apel de funție. Dacă se utilizează manipulatori cu parametri, trebuie inclus în fişierul sursă curent fişierul header `<iomanip>`.
Principalii manipulatori sunt următorii:

- Pentru setarea indicatorilor de aliniere:
  `ios::left`
  `ios::right`
  `ios::internal`
- Pentru setarea bazei de numerație a afişării valorilor întregi:
  `ios::dec`
  `ios::hex`
  `ios::oct`
  `ios::setbase(int b)` – manipulator cu parametru ce specifică baza de numerație;
- Pentru setarea modului de afişare a valorilor reale:
  `ios::fixed`
  `ios::scientific`
- Pentru controlul indicatorului `ios::boolalpha`:
  `ios::boolalpha`
  `ios::noboolalpha`
- Pentru controlul indicatorului `ios::showbase`:
  `ios::showbase`
  `ios::noshowbase`
- Pentru controlul indicatorului `ios::skipws`:
  `ios::skipws`
  `ios::noskipws`
- Pentru controlul indicatorului `ios::unitbuf`:
  `ios::unitbuf`
  `ios::nounitbuf`
- `ios::endl` – inserează în bufferul de ieşire caracterul '`\n`' şi goleşte bufferul respectiv;
- `ios::ends` – inserează în bufferul de ieşire caracterul '`\0`';
- `ios::setfill(int ch)` – defineşte caracterul de umplere;
- `ios::setprecision(int width)` – defineşte precizia de afişare a valorilor reale (numărul de caractere al părții fracționare);
- `ios::setw(int width)` – defineşte mărimea următorului câmp de citire sau de scriere; acest manipulator are un efect temporar, doar asupra câmpului imediat următor;
- `ios::setioflags(fmtflags flags)` – manipulatorul apelează funcția `ios::setf(flags)` pentru setarea indicatorilor respectivi;
- `ios::resetioflags(fmtflags flags)` – manipulatorul apelează funcția `ios::resetf(flags)` pentru resetarea indicatorilor respectivi;

Exemplu. Următorul program utilizează diferite setări pentru afişarea unui număr întreg şi a unuia real. S-au folosit funcții de control a indicatorilor de format. Pentru fiecare secvență, s-a scris sub formă de comentariu şirul de caractere afişat la ieşire.

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

void main() {
  int i = 47;
  double f = 2300114.414159;

  cout << i << endl;     // Implicit afisare zecimala
  // 47
  cout.setf(ios::showbase);
  cout.setf(ios::hex, ios::basefield);
  cout << i << endl;
  // 0xf2
  cout.setf(ios::dec, ios::basefield);
  cout.fill('0');
  cout.width(10);
  cout << i << endl;     // Implicit aliniere la dreapta
  // 0000000047

  cout.setf(ios::scientific, ios::floatfield);
  cout << endl << f << endl;  // Implicit precizia 6
  // 2.300114e+006
  cout.setf(ios::fixed, ios::floatfield);
  cout << f << endl;
  // 2300114.414159
  cout.width(30);
  cout.precision(20);
  cout << f << endl;
  // 002300114.41415899990000000000
}
```

O variantă echivalentă utilizând manipulatori este următoarea:

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

void main() {
  int i = 47;
  double f = 2300114.414159;

  cout << i << endl;
  // 47
  cout << showbase << hex << i << endl;
  // 0xf2
  cout << setfill('0') << setw(10) << i << endl;
  // 0000000047

  cout << scientific << f << endl;
```

```
  // 2.300114e+006
  cout << fixed << f << endl;
  // 2300114.414159
  cout << setw(30) << setprecision(20) << f << endl;
  // 002300114.41415899990000000000
}
```

11.4 Clasele `istream` şi `ostream`. Operații de citire şi scriere

Clasele `istream` şi `ostream` definesc principalele operații de citire şi scriere. Fişierele header care conțin declarațiile acestora sunt `<istream>` şi `<ostream>`. Dacă însă se utilizează stream-urile predefinite `cin`, `cout` şi `cerr`, va trebui inclus fişierul `<iostream>`, care în plus cumulează definițiile anterioare.

11.4.1 Clasa `ostream` şi operații de scriere

Clasa `ostream` moşteneşte prin intermediul clasei `ios` toate facilitățile şi operațiile de formatare a datelor. Obiecele `cin`, `cerr` şi `clog` sunt toate instanțe ale acestei clase.

În mod uzual, nu se creează în mod direct obiecte de tipul `ostream` în cadrul programelor, ci se utilizează obiectele predefinite ale acestei clase. Însă este posibil să se creeze obiecte `ostream` pe baza unui stream deja creat, utilizân bufferul acestuia. Sintaxa de creare a unui asemenea stream este:

```
ostream <nume_stream> (streambuf* buffer);
```

Clasa `ostream` permite moduri de scriere a datelor: ***binară*** şi ***cu format***.

Scrierea binară în stream-uri presupune scrierea neformatată a informației, exact aşa cum este reprezentată în memoria internă. Deşi funcțiile ce permit scrierea binaă sunt utilizate în mod special în cazul fişierelor, ele sunt definite în clasa `ostream` ca funcții membru şi pot fi utilizate pentru orice stream-uri de ieşire.

Principalele funcții de scriere binară sunt `put` şi `write`:

- `ostream& ostream::put(char c);`

Funcția scrie un octet în stream-ul asociat. Deoarece un caracter este un octet, această funcție poate fi utilizată ți pentru scrierea unui caracter într-un fişier de tip text.

- `ostream& ostream::write(const char* buffer, int n);`

Funcția scrie *n* octeți din zona de memorie indicată de *buffer* în stream-ul respectiv.

Observație. Ambele funcții returnează stream-ul curent, astfel încât se pot concatena mai multe apeluri consecutive. De exemplu, secvența:

```
cout.put('A');
cout.put('B');
cout.put('C');
```

este echivalentă cu:

```
cout.put('A').put('B').put('C');
```

Pentru ***scrierea formatată*** în stream-uri se utilizează ***operatorul de inserare***, care reprezintă o supraâncărcare a operatorului de deplasare spre stânga al limbajului C. Sintaxa acestuia este:

```
ostream& ostream::operator<<(T& obj);
```

Parametrul operatorului poate fi oricare tip de date predefinit al limbajului C++, astfel încât operatorul << a fost supraâncărcat pentru toate aceste tipuri.

Ca şi în cazul funcţiilor de scriere binare, operatorul de inserare returnează o referintă la stream-ul curent, astfel încât apeluri consecutive ale acestui operator pot fi concatenate. De exemplu, secvenţa:

```
cout << 'A';
cout << 'B';
cout << 'C';
```

este echivalentă cu:

```
cout << 'A' << 'B' << 'C';
```

Datele reprezentând valoarea argumentului operatorului de inserare sunt formatate (adică transformate într-un şir de caractere în conformitate cu indicatorii de format) şi scrise în bufferul obiectului `streambuf` asociat stream-ului. Operaţia următoare, de transfer a informaţiei din buffer la dispozitivul periferic aferent, este gestionată de către obiectul `streambuf`.

11.4.1 Clasa `istream` şi operaţii de citire

Clasa `istream` defineşte principalele operaşii de citire a datelor prin intermediul stream-urilor şi moşteneşte, ca şi `ostream` proprietăţile clasei `ios`. Obiectul predefinit `cin` este de tip `istream`.

Ca şi în cazul clasei `ostream`, în mod uzual obiectele `istream` nu se creează în mod directe, ci prin intermediul unui alt sream creat deja. Sintaxa de definire este următoarea:

```
istream <nume_stream> (streambuf* buffer);
```

Citirea formatată a informaţiei se realizează prin intermediul ***operatorului de extragere***, ce reprezintă o supraâncărcare a operatorului de deplasare spre stânga al limbajului C. În acest caz informaţia externă este un şir de caractere care se converteşte în format intern binar conform valorilor curente ale indicatorilor de format.

Sintaxa acestui operator este:

```
istream& istream::operator>>(T& obj);
```

Şi în acest caz, operatorul de deplasare a fost supraâncărcat pentru toate tipurile de date predefinite. De asemenea, şi în acest caz, operatorul de extragere se poate concatena:

```
cin >> x >> y >> z;
```

Observație. Parametrul operatorului de extragere trebuie să fie o l-valoare, căruia i se va atribui valoarea rezultată după conversie.

Operațiile de ***citire binară*** neformatată realizează citirea unui număr de octeți dintr-un stream de intrare, fără să se mai modifice informația citită.

Observație. Ca şi în cazul scrierii, citirea dintr-un fişier text reprezintă o citire binară, deoarece caracterele citite nu se mai modifică.

Deşi sunt utilizate în mod uzual în cazul fişierelor, următoarele funcții de citire binară sunt definite în calsa `istream` şi pot fi utilzate pentru orice stream de intrare:

- `int istream::get();`

Citeşte următorul caracter din stream-ul de intrare şi îl returnează ca o valoare de tip `int`; în cazul în care nu mai există caractere în stream se returnează `EOF`;

- `istream& istream::get(char* buff,int len[,char delim]);`

Citeşte o secvență de caractere de dimensiune cel mult *len-1* de la stream-ul de intrare în memoria internă la adresa specificată de parametrul *buff*. Delimitatorul implicit de sfârşit al câmpului este '`\n`'. În cazul în care se întâlneşte delimitatorul, citirea se opreşte, dar delimitatorul nu este eliminat din bufferul de intrare. După citirea caracterelor, se adaugă în mod suplimentar un caracter cu codul zero.

- `istream& istream::getline(char* buff,`
  `int len [,char delim]);`

Funcția este asemănătoare ce cea precedentă, cu deosebirea că delimitatorul este eliminat din stream-ul de intrare (dar nu este scris în memorie la adresa *buff*). Dacă delimitatorul nu a fost întâlint în cadrul celor *len-1* caractere, funcțiile `eof()` şi `fail()` vor returna valorea `true`.

- `int istream::peek();`

Funția returnează următorul caracter disponibil din stream-ul de intrare, dar nu-l elimină din stream.

- `istream& istream::putback(char c);`

Caracterul *c* este pus înapoi în stream-ul de intrare, astfel încât el va fi primul caracter citit de următoarea operație de citire. Este obligatoriu ca *c* să fie ultimul caracter citit din stream, altfel operația eşuează.

- `istream& istream::read(char* buff, int len);`

Se citesc cel mult *len* octeți din stream-ul de intrare şi se scriu în memorie la adresa *buff*. Dacă se întâlneşte caracterul `EOF`, citirea se termină şi funcția `eof()` returnează `true`. Funcția este utilizată în mod uzual pentru fişierele binare.

11.4.3 Controlul operațiilor de intrare/ieşire de către utilizator. Supraâncărcarea operatorilor de deplasare

În mod uzual, funcțiile publice ce permit gestiunea indicatorilor de format şi funcțiile manipulator sunt suficiente pentru controlul operațiilor de intrare/ieşire. Limbajul C++ permite în plus scrierea de către programator a funcțiilor proprii de control ale acestor operații.

Există două metode utilizate pentru aceste operații:

- srierea propriilor funcții manipulator,
- supraâncărcarea operatorilor de deplasare.

A) Supraîncărcarea operatorilor de deplasare

Operatorii de inserare şi extragere predeiniți sunt supraîncărcați doar pentru tipurile de date predefinite. Este însă posibil să se creeze operatori proprii, care să permită citirea şi scrierea şi altor tipuri de valori.

Cu alte cuvinte, este posibil să se scrie şi să se citească obiecte instanță ale unor clase definite de utilizator. Acesta însemnă de fapt citirea şi scrierea din/în tream-uri a valorilor datelor membre ale obiectelor acestor clase.

Deoarece un operator de inserare/extragere a datelor în/din stream-uri este asociat întotdeauna unui stream, rezultă că operandul stâng al unui asemenea operator trebuie să fie un stream (sau o referință spre un stream) şi deci operatorii de deplasare trebuie definiți ca funcții globale, nu ca funcții membre ale claselor.

În mod uzual, o asemenea funcție are două argumente: primul este o referință spre stream-ul folosit pentru scriere sau citire, iar al doilea este obiectul care se doreşte să se scrie sau să se citească.

Observație. Pentru citi şi a modifica valorile datelor membre ale clasei respective, este nevoie să se definească funcții de tip accesor.

Notând cu *T* clasa pentru care se doreşte supraâncărcarea unui operator de deplasare, sintaxa prototipului este:

```
istream& operator>>(istream&, T&);
ostream& operator<<(ostream&, T&);
```

Exemplu. Definirea unor operatori globali supraâncărcați de inserare şi de extragere pentru clasa *Punct*.

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

class Punct {
  int x, y;
public:
  Punct (int a=0, int b=0): x(a), y(b) { }
```

```
  int X() const { return x; }
  int Y() const { return y; }
  void SetX(int a) { x = a; }
  void SetY(int b) { y = b; }
};

ostream& operator<<(ostream& os, Punct& p) {
  os << "x= " << p.X() << ", y= " << p.Y() << endl;
  return os;
}

istream& operator>>(istream& is, Punct& p) {
  int a, b;
  if (is >> a >> b) {
    p.SetX(a);
    p.SetY(b);
  }
  return is;
}

void main() {
  punct p;
  cin >> p;
  cout << p;
}
```

B) Crearea funcțiilor manipulator

Un manipulator fără argumente este relativ simplu de scris. Funcția respectivă trebuie să fie definită în afara oricărei clase. Ea primeşte un argument reprezentând o referință la un stream de intrare sau de ieşire, efectuează anumite operații asupra stream-ului respectiv şi îl returnează apoi ca valoare de ieşire.

Exemplu. Crearea unui manipulator propriu pentru afişarea valorilor reale, care stabileşte formatul de afişare:

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

ostream& init(ostream& os) {
  os.width(15);      // campul de afisare de 10 caractere
  os.precision(7);   // precizia de 4 caractere
  os.fill('0');      // caracterul de umplere 0
  return os;
}

void main() {
  cout << init << 174.48954 << endl;
}
```

Crearea manipulatorilor cu argumente reprezintă o operație mai dificilă (chiar şi varianta de implementare a acestora în biblioteca `iostream` este destul de complicată). O soluție la această problemă a fost propusă de Jerrz Schwarz (creatorul bibliotecii standard `iostream`) şi o constituie o categorie de clase numite ***efectori***. Un efector este o clasă ce conține un constructor (de conversie în mod uzual) şi supraâncărcarea operatorului de inserare sau extragere care efectuează operația dorită.

În acest fel, apelul constructorului respectiv poate apare ca argument în cadrul operatorului de inserare/extragere şi este achivalent cu un maniplator cu argumente.

Exemplu. Clasa *ULBin* este o clasă de tip efector, ce permite afişarea unor valor de tip `unsigned long` sub formă de cifre binare. Pentru afişarea cifrelor binare se utilizează o mască (tot de tip `unsigned long`) ce conține un singur bit 1 pe poziția cea mai din stânga şi restul cifrelor 0. Acest bit este deplasat de la stânga la dreapta, selectând astfel toate cifrele numărului respectiv.

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

class ULBin {
  unsigned long n;
public:
  ULBin(unsigned long a): n(a) {}
  unsigned long N() const { return n; }
  void SetN(unsigned long a) { n = a; }
};

ostream& operator<<(ostream& os, ULBin &b) {
  // valoareaa initiala a mastii
  unsigned long mb = 1UL << 31;
  while (mb) {
    os << ((b.N() & mb) ? '1' : '0');   // afisare cifra
     mb >>= 1;      // deplasare spre dreapta
  }
  return os;
}

void main() {
  unsigned long x = 0xa78bc12e, y = 0x1a785ce6;
  cout << "x in baza 2: " << ULBin(x) << endl;
  cout << "y in baza 2: " << ULBin(y) << endl;
}
```

11.5 Ierarhii derivate din `iostream`: `fstream` şi `stringstream`

Modul de proiectare al ierarhiei `iostream` a permis ca operațiile definite în cadrul acesteia să poată fi efectuate şi asupra altor dispozitive în afara terminalelor standard de intrare şi ieşire: asupra fişierelor în cazul stream-urilor din ierarhia `fstream` şi asupra memoriei interne în cazul stream-urilor din ierarhia `stringstream`.

11.5.1 Ierarhia `fstream`

Clasele `ifstream`, `ofstream`, şi `fstream` sunt definite în fişierul header `<fstream>`. Un stream de tipul `ofstream` permite doar operații de scriere în fişiere, unul de tipul `ifstream` permite doar operații de citire, pe când un stream de tipul `fstream` permite ambele categorii de operații.

Spre deosebire de stream-urile asociate perifericelor standard de intrare şi ieşire, în acest caz fiecare stream trebuie asociat în mod explicit la un anumit fişier. Ca şi in cazul limbajului C, operația de asociere a unui stream la un fişier se numeşte ***deschiderea fişierului***, operația inversă numindu-se ***închiderea fişierului***.

Observație. Prin închiderea unui fişier se taie legătura între stream şi fişier, stream-ul respectiv existând în continuare în carul programului până la distrugerea obiectului stream.

A) Crearea şi utilizarea stream-urilor

Există două categorii de constructori pentru aceste tipuri de stream-uri. Prima categorie o formează constructorii implicți şi permit crearea stream-urilor fără ca acestea să fie asociate unui anumit fişier (asocierea putânde-se realiza ulterior prin intermediul unei operații specifice).

De exemplu:

```
ifstream fin;  // crearea unui stream de intrare
ofstream fout; // crearea unui stream de ieşire
ifstream finout; // crearea unui stream de intrare/ieşire
```

În ierarhia `fstream`, bufferul asociat streamurilor este din clasa `filebuf`. Clasa `filebuf`. Este o specializare a clasei `streambuf` şi adaugă la aceasta elemente specifice pentru asocierea fişierelor.

Principala funcție care permite asocierea unui fişier la un obiect `filebuf` este funcția `open`:

```
filebuf* filebuf::open(const char* name,
     ios::openmode mode);
```

Ea deschide un fişier specificat prin nume şi îl asociază la bufferul curent, modul de acces la fişier fiind specificat prin al doilea parametru. Principalele valori posibile pentru parametrul *mode* sunt:

- `ios::in` – deschide un fişier existent pentru citire;
- `ios::out` – deschide un fişier pentru scriere; dacă fişierul nu există, el este creat, altfel conținutul inițial al fişierului se şterge;
- `ios::app` – deschide fişierul pentru scriere, poziționându-se la sfîrşitul fişierului (operația de adăugare); conținutul inițial al fişierului (dacă există) este păstrat;
- `ios::ate` – deschide un fişier existent pentru scriere şi citire, poziționându-se la sfîrşitul acestuia; conținutul fişierului este păstrat;
- `ios::nocreate` – deschide un fişier doar dacă el există deja;

- `ios::noreplace` – deschide un fişier doar dacă el nu există deja;
- `ios::trunc` – deschide un fişier şi şterge fişierul inițial, dacă acesta există deja;
- `ios::binary` – deschide un fişier în mod binar; în mod implicit fişierul este considerat de tip text;

Aceste valori pot fi combinate cu ajutorul operatorilor la nivel de bit. De exmplu, următoarele combinații au semnificații specifice:

- `ios::out | ios::app` - deschide fişierul pentru adăugare (conținutul fişierului este păstrat);
- `ios::out | ios::trunc` - deschide fişierul pentru adăugare (conținutul fişierului este şters);
- `ios::in | ios::out` - deschide fişierul pentru scriere şi citire (fişierul trebuie să existe deja);
- `ios::in | ios::out | ios::trunc` - deschide fişierul pentru scriere şi citire (fişierul, dacă există, este şters);

Observație. Aceste moduri de deschidere nu sunt proprii doar fişierelor, ci tuturor tipurilor de stream-uri.

Funcția opusă lui `open` este `close` şi închide asocierea între buffer şi fişier:

```
filebuf* filebuf::close();
```

Aceste funcții sunt definite şi în clasele `ifstream`, `ofstream` şi `fstream`, apelând funcții cu acelaşi nume din `filebuf`. Funcția `open` deschide un fişier şi îl asociază unui stream:

```
void ofstream::open(const char* name,
       ios::openmode = ios::out | ios::trunc);

void ifstream::open(const char* name,
       ios::openmode = ios::in);

void fstream::open(const char* name,
       ios::openmode = ios::in | ios::out);
```

Funcția `close` închide legătura dintre fişier şi stream:

```
void ofstream::close();
void ifstream::close();
void fstream::close();
```

Observație. Funcția `close` se apelează în mod automat în cadrul destructorului unui stream, aşa încât nu este necesar apelul explicit al ei.

<u>A doua categorie</u> de constructori pentru aceste stream-uri realizează atît crearea stream-urilor, cât şi asociaerea lor la fişiere:

```
explicit ofstream::ofstream(const char* name,
```

```
        ios::openmode = ios::out | ios::trunc);

explicit ifstream::ifstream(const char* name,
        ios::openmode = ios::in);

explicit fstream::fstream(const char* name,
        ios::openmode = ios::in | ios::out);
```

După crearea unui stream şi asocierea lui la un fişier, se pot utiliza funcţiile de scriere şi citire definite în clasele `istream` şi `ostream`.

Exemplu. Programul următor copiază conţinutul unui fişier text de intrare într-un fişier de ieşire, înlocuind spaţiile libere prin caracterul '`*`'. Indicatorul de format `ios::skipws` trebuie dezactivat, prevenind ignorarea spaţiilor. S-a utilizat un manipulaor pentru afişarea unui mesaj de eroare.

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

ostream& err(ostream& os) {
  os << "Eroare deschidere fisier\n";
  return os;
}

void main() {
  ifstream fin;
  ofstream fout("f_out.txt");
  if (!fout) {
    cout << err;
    return;
  }
  fin.open("f_in.txt");
  if (!fout) {
    cout << err;
    return;
  }
  char c;
  fin.unsetf(ios::skipws);
  while (!fin.eof()) {
    fin >> c;
    if (c == ' ')
      c = '*';
    fout << c;
  }
  fin.close();
}
```

B) Poziționarea în cadrul fişierelor

Un element important referitor la scrierea şi citirea în/din stream-uri îl reprezintă poziția din stream unde urmează să fie scrisă sau citită informația. Poziționarea este o problemă comună tuturor stream-urilor, dar utilizată în mod special în cazul celor asociate la fişiere.

Poziția curentă de scriere sau citire este gestionată automat de către funcțiile de citire şi scriere, dar există cazuri în care se doreşte gestionarea explicită de către programator. Toate clasele ierarhiei `iostream` conțin două funcții publice, `seekp` (pentru stream-urile de ieşire) şi `seekg` (pentru stream-urile de ieşire) pentru realizarea acestor operații.

Fiecare dintre cele două funcții are două variante supraâncărcate: într-una dintre acestea se specifică poziția absolută unde va avea lor următoarea operație de citire sau scriare, iar în cealaltă se specifică o poziție relativă. În continuare se poate considera tipul `pos_type` ca fiind o structură ce conține informațiile necesare pentru poziționare, iar `off_type` ca fiind `unsigned int`.

În primul caz, poziția curentă de scriere/citire este gestionată cu ajutorul a doim pointeri, numiți respectiv "put pointer" şi "get pointer":

- `ostream& ostream::seekp(pos_type pos)` – setează poziția curentă pentru următoarea operație de scriere la *pos*;
- `istream& istream::seekg(pos_type pos)` – setează poziția curentă pentru următoarea operație de citire la *pos*;

Pentru a putea utiliza această variantă a funcțiilor `ssekp` şi `seekg`, trebuie în mod uzual apelată anterior una din funcțiile `tellp` (pentru scriere) sau `tellg` (pentru citire):

- `pos_type ostream::tellp();`
- `pos_type istream::tellg();`

Fiecare dintre cele două funcții returnează poziția absolută unde ca avea loc următoarea operație de scriere sau citire. Aceste valori, eventual modificate, vor servi ca argumente funcțiilor `ssekp` şi `seekg`.

În cazul al doilea, poziționarea relativă într-un stream se specifică prin intermediul unui deplasament față de o anumită poziție din stream:

- `ostream& ostream::seekp(off_type offset,`
  `ios::seekdir pos);`
- `istream& istream::seekg(off_type offset,`
  `ios::seekdir pos);`

Tipul enumerat `ios::seekdir` este definit în clasa `ios` şi poate are următoarele valori:

- `ios::beg` – specifică poziția de la începutul stream-ului;
- `ios::end` – specifică poziția de la sfârşitul stream-ului;
- `ios::beg` – specifică poziția curentă a stream-ului;

C) Citirea şi scrierea din/în acelaşi stream asociat unui fişier

Există situații în care este necsar ca un anumit fişier să fie utilizat atât pentru scriere, cât şi pentru citire. În acest caz, stream-ul asociat fişierului trebuie să fie de tipul `fstream`, iar fişierul trebuie deschis pentru ambele tipuri de operații.

În acest caz, bufferele de intrare/ieşire partajează aceeaşi zonă de memorie şi pot fi folosite operații combinate de citire şi scriere.

Deşi pot fi utilizați, operatorii de inserare şi de extragere nu sunt în mod uzual folosiți. În orice caz, nu este permis să se utilizeze într-o singură instrucțiune ambii operatori pentru acelaşi stream. De exemplu, următoarea secvență generează o eroare:

```
string str;
fstream f("f.txt");
f >> str << "Un sir " >> str;
```

Corect este:

```
f >> str;
f << "Un sir ";
f >> str;
```

În mod uzual, pentru citirea şi scrierea informațiilor în acelaşi fişier se utilizează funcțiile de poziționare.

Exemplu. Următorul program permite scrierea şi citirea dintr-un fişier a informațiilor aferente obiectelor unei clase *Punct*. Fişierul ce stochează obiectele de tipul *Punct* este privit ca o secvență de elemente de tipul *Punct*. Funcțiile *Read* şi *Write* ale clasi *Punct* permit citirea şi scrierea în fişier a unui obiect al acestei clase. Al doilea parametru al acestora este un întrag, specificând indicele obiectului ce urmează să fie citit sau scris.

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

class Punct {
  int x, y;
public:
  Punct (int a=0, int b=0): x(a), y(b) { }
  int X() const { return x; }
  int Y() const { return y; }
  void SetX(int a) { x = a; }
  void SetY(int b) { y = b; }
  void Read(fstream& f, int k);
  void Write(fstream& f, int k);
  void Print() const {
    cout << "x= " << x << ", y= " << y << endl;
  }
};

void Punct::Read(fstream& f, int k) {
  f.seekg((k-1)*sizeof(Punct), ios::beg);
```

```
    f.read(reinterpret_cast<char*>(this), sizeof(Punct));
}

void Punct::Write(fstream& f, int k) {
    f.seekp((k-1)*sizeof(Punct), ios::beg);
    f.write(reinterpret_cast<char*>(this), sizeof(Punct));
}

void main() {
    Punct p1(1, 1), p2(2, 2), p3(3, 3), p;
    fstream f("Puncte.dat",
        ios::in | ios::out | ios::trunc);
    f.write(reinterpret_cast<char*>(&p1), sizeof(Punct));
    f.write(reinterpret_cast<char*>(&p2), sizeof(Punct));
    f.write(reinterpret_cast<char*>(&p3), sizeof(Punct));
    p.Read(f,2);
    p.SetX(7);
    p.Write(f, 2);
    p.Read(f, 1);
    p.Print();
    p.Read(f, 2);
    p.Print();
    p.Read(f, 3);
    p.Print();
}
```

11.5.2 Ierarhia `stringstream`

Ierarhia `stringstream` a fost creată pentru a simplifica operațiile de citire şi de scriere a datelor în/din meorie. Anterior s-a utilizat ierarhia `strstream`, dar datorită unor probleme de alocare dinamică a fost creată această nouă ierarhie. Bufferul asociat unui stream din această ierarhie este o instanță a clasei `stringbuf`. Definițiile acestor clase se află în fişierul header `<sstream>`.

În general, toate operațiile specificate la fişiere, cu excepția celor de deschidere şi închidere a fişierelor, se pot aplica şi asupra obiectelor din această ierarhie.

Aceste clase sunt proiectate să lucreze cu elemente din clasa `string`, astfel încât un stream se asociază unui `string`. Crearea unor obiecte din ierarhia `stringstream` se poate face cu ajutorul următoarilor constructori:

```
explicit ostringstream::ostringstream(
     ios::openmode mode = ios::out);

explicit ostringstream::ostringstream(const string& s,
     ios::openmode mode = ios::out);

explicit istringstream::istringstream(
     ios::openmode mode = ios::in);
```

```
explicit istringstream::istringstream(const string& s,
     ios::openmode mode = ios::in);

explicit stringstream::stringstream(
     ios::openmode mode = ios::in | ios::out);

explicit stringstream::stringstream(const string& s,
     ios::openmode mode = ios::in | ios::out);
```

Primul parametru în cazul constructorilor cu doi parametri reprezintă valoarea de inițializare a stream-ului respectiv.

Există variante supraâncărcate ale funcției `str` care permit accesul la string-ul asociat stream-ului curent. De exemplu, pentru clasa `ostringstream`, acestea sunt (pentru celelalte clase funcțiile sunt asemănătoare):

- `string ostringstream::str() const;` - returnează string-ul asociat
- `void ostringstream::str(const string&);` - setează string-ul asociat

Asupra stream-urilor din această ierarhie se pot aplica operațiile de formatare.

Exemplu. Fie programul:

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

void main() {
  ostringstream os;
  cout << os.str() << endl;
  os.setf(ios::showbase);
  os.setf(ios::hex, ios::basefield);
  os << 12345;
  cout << os.str() << endl;
  os << " Un nou sir";
  cout << os.str() << endl;
}
```

Ieşirea programului este următoarea:

```
0x3039
0x3039 Un nou sir
```