# Introducere în programare
# 2013 - 2014

Corina Forăscu

corinfor@info.uaic.ro

http://profs.info.uaic.ro/~corinfor/teach/IntroP/

# Curs 4: conținut

- Crearea de sinonime cu **`typedef`**
- Tipuri enumerative
- Structuri & Structuri alternative
- Operaţii la nivel de bit & Câmpuri de biţi în structuri
- I/O cu fişiere

# Crearea de sinonime cu **`typedef`**

- o declarație de forma

```
double a[20];
```

declară variabila **`a`** ca aparținând tipului **`double[20]`**

- putem asocia tipului **`double[20]`** un sinonim:

```
typedef double TablouDouble20[20];
```

- variabila poate fi declarată acum și așa:

```
TablouDouble20  a;
```

- sintaxa: ⟨decl-typedef⟩ ::= **`typedef`** ⟨definitie-tip⟩ ⟨sinonim⟩

# Exemple de tipuri și sinonime

- tipuri predefinite: `char int long double` etc.

```
typedef int Integer;
```

- tipuri pointer: **T***

```
typedef int *PInt;
```

  - **PInt** este sinonim pentru **int ***

- tablouri

```
typedef double Matrice[MMAX][NMAX];
```

  - **Matrice** este sinonim pentru **double[MMAX][NMAX]**

  - **TablouPInt** este sinonim pentru **int *[NMAX]**

```
typedef int *TablouPInt[NMAX];
```

# Exemple de tipuri și sinonime

- funcţii

```
typedef int Fct1(double);
```

  - **`Fct1`** este sinonim pentru **`int(double)`** = tipul funcţiilor care au un parametru **`double`** şi întorc o valoare de tip **`int`**

```
typedef int* Fct2();
```

  - **`Fct2`** este sinonim pentru **`int *()`** = tipul funcţiilor fără parametru şi care întorc pointer la un **`int`**

- pointer la funcţii

```
typedef int(*Fct3)();
```

  - **`Fct3`** este sinonim pentru **`int (*)()`** = tipul pointerilor la funcţii fără parametru şi care întorc un **`int`**

- Numele unei funcţii este pointer

# Tipuri enumerative

- o declaraţie de forma

```
enum zi { lu, ma, mi, jo, vi, si, du };
```

declară un tip cu numele **enum zi** şi
cu constantele **lu, ma, mi, jo, vi, si, du**

- variabile ale tipului **enum zi**

```
enum zi azi, ieri;
```

- tipul enumerativ este compatibil cu **char** sau cu un tip întreg cu semn sau cu un tip întreg fără semn (depinde de implementare)

- fiecare constantă a tipului are asociată o valoare întreagă

**(int)lu = 0, (int)ma = 1, …, (int)du = 6**

- au sens expresii ca **ieri++** sau **azi + 3**

# Tipuri enumerative

- valorile asociate pot fi precizate explicit

```
enum zi { lu = 1, ma, mi, jo, vi, si, du };
enum roman { i = 1, ii, iii, iv, x = 10, xi, xii };
```

- se poate utiliza în combinaţie cu **typedef**

```
typedef enum zi zi;
zi azi;
enum zi ieri;
```

- care este echivalentă cu

- sau

```
typedef enum zi { lu, ma, ... } zi;
```

- dar se poate şi asa:

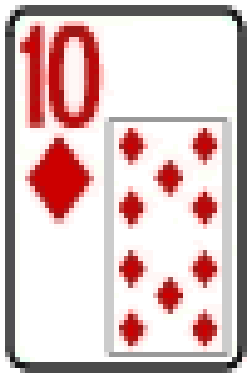
```
typedef enum { lu, ma, ... } zi;
zi azi;
```

```
enum { lu, ma, ... } azi, ieri;
```

# Structuri

- Structură: ansamblu eterogen de variabile numite cîmpuri; structura are un nume şi fiecare câmp are propriul nume şi propriul tip.
- Memoria alocată este o zonă contiguă; elementele sunt memorate în ordinea declarării în structură

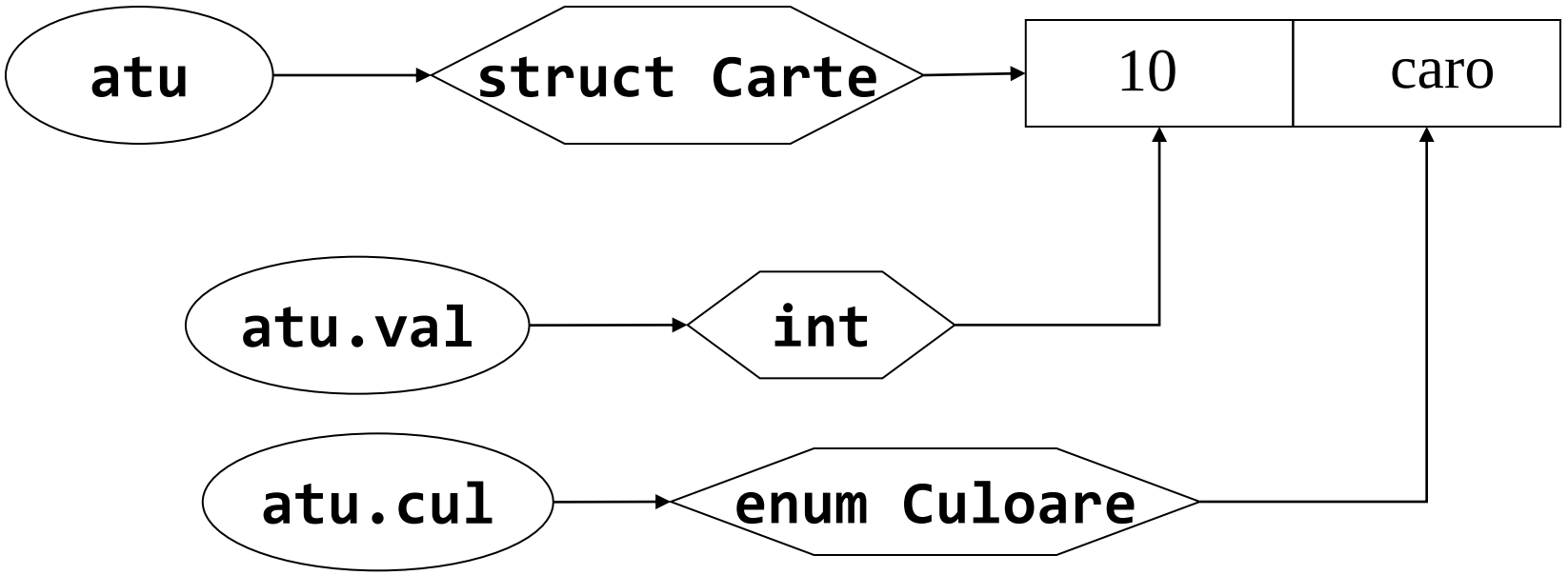
# Structuri simple




```
enum Culoare { trefla, cupa, caro, pica };
typedef enum Culoare Culoare;
struct Carte
{
    int val;
    Culoare cul;
};
```

# Structuri simple

```
struct Carte atu;
atu.val = 10;
atu.cul = caro;
```



# Structuri simple

```
cout << “atuul este : “ << atu.val;
switch (atu.cul)
{
case trefla:
     cout << “ trefla\n”;
     break;
case caro:
     cout << “ caro\n”);
     break;
// ...
}
```

# Asocierea de sinonime pentru structuri

- numele **`struct Carte`** este prea lung
- îi putem asocia un sinonim

```
typedef struct Carte
{
    int val;
    Culoare cul;
} Carte;
```

- acum putem declara o variabilă mult mai simplu

```
Carte atu;
```

- acum **`Carte`** şi **`struct Carte`** sunt sinonime

# Asocierea de sinonime pentru structuri

- cu **`typedef`** structura poate fi şi anonimă

```
typedef struct
{
      int val;
      Culoare cul;
} Carte;
```

- acum poate fi utilizat numai **`Carte`**

# Structuri complexe

- un jucător are nume, o mână de cărţi şi o sumă de bani

```
typedef struct Jucator
{
        char* nume;
        Carte mana[4];
        long suma;
} Jucator;
```

- o masă are un număr şi 4 jucători

```
typedef struct Masa
{
        int nr;
        Jucator jucator[4];
} Masa;
```

# Structuri complexe

- jucătorul j primeşte 8 de treflă ca a doua carte

```
j.mana[1].val = 8;
j.mana[1].cul = trefla;
```

- jucătorul 3 de la masa m primeşte 9 de caro ca prima carte

```
m.jucator[2].mana[0].val = 9;
m.jucator[2].mana[0].cul = caro;
```

# Structuri alternative

- o figura (geometrică) este
  - sau un punct
  - sau un segment
  - sau un cerc
  - sau …

- Întrebare: putem defini un tip “figura”?

- Răspuns: DA
  - utilizând structurile: … dar nu este economic
  - utilizând **structurile alternative**.

# Tipuri **union** (structuri alternative)

- Sintaxa tipului **union** este aceeaşi cu cea a lui **struct**
- Operaţiile sunt aceleaşi
- Un tip uniune defineşte o mulţime de valori alternative care partajează aceeaşi porţiune de memorie

```
union int_or_float{ int i; float x; };
```

- Mărimea (**sizeof**) unei uniuni este mărimea celui mai lung tip dintre membrii uniunii
- Se poate atribui o valoare unui membru, se poate extrage valoarea unui membru, dar:
  - Programatorul este responsabil pentru interpretarea corectă a valorilor memorate în memoria comună

# Structuri alternative

```
typedef struct {
      int x;
      int y;
} punct;

typedef struct {
      punct A;
      punct B;
} segment;

typedef struct {
      punct centru;
      int raza;
} cerc;
```

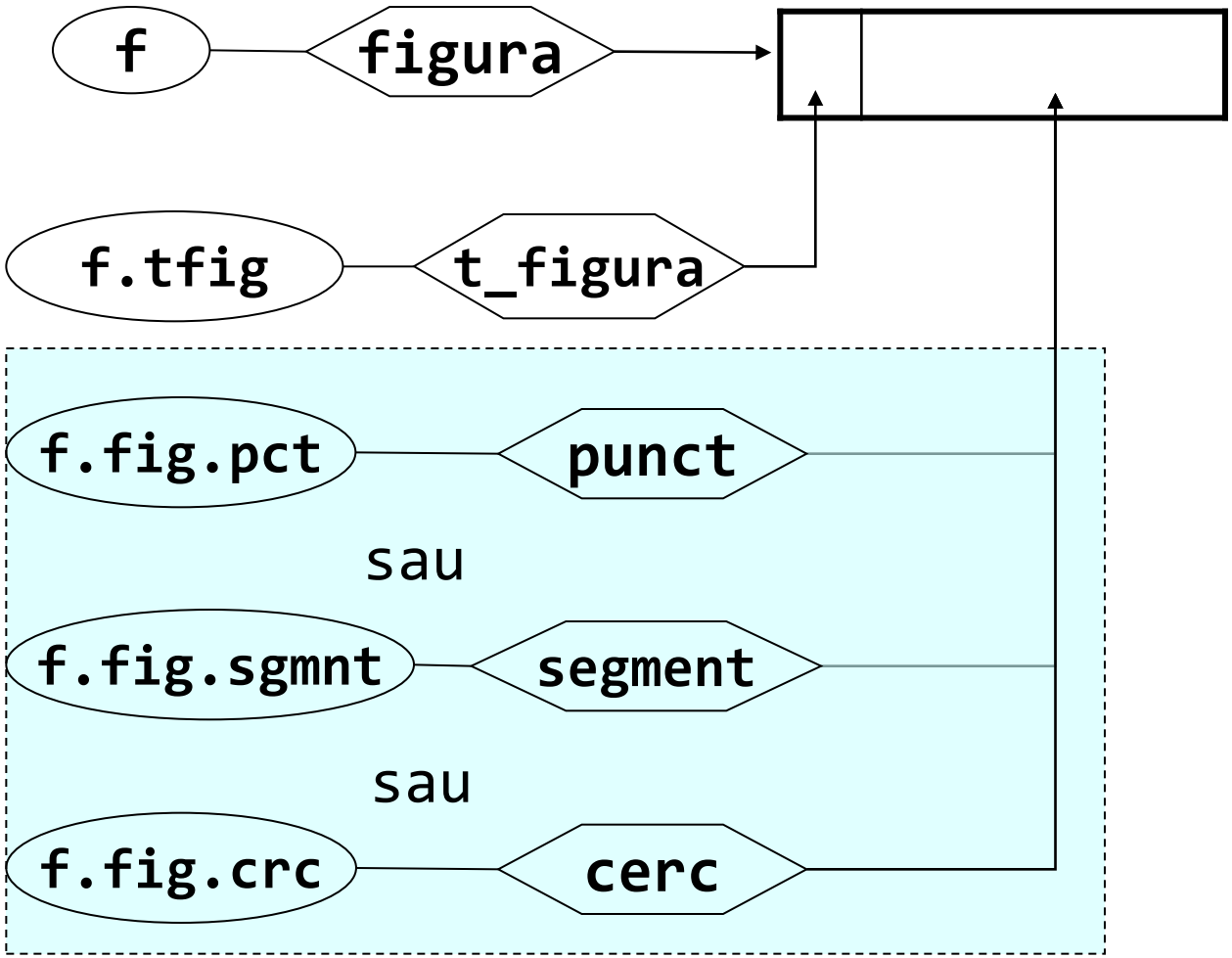
# Structuri alternative

```
typedef enum { t_punct, t_segment, t_cerc }
t_figura;

typedef struct {
      t_figura tfig;
      union {
            punct pct;
            segment sgmnt;
            cerc crc;
      } fig;
} figura;
```

# Structuri alternative - variabile

```
figura f;
```



# Structuri alternative – utilizare variabile

- calculul perimetrului

```
double perim(figura f) {
      switch (f.tfig)
      {
      case t_punct:
            return 0;
            break;
      case t_segment:
            return lung_segm(f.fig.sgmnt);
            break;
      case t_cerc:
            return perim_cerc(f.fig.crc);
            break;
      }
}
```

```
int main(){
    figura o_figura;

    cerc un_cerc = { 5, 5, 100 };

    o_figura.tfig = t_cerc;
    o_figura.fig.crc = un_cerc;  //cerc

    cout << "Figura are perimetrul: ";
    cout << perim(o_figura) << "\n";

    //     segment un_segment = {0,0,3,4};
    //     o_figura.tfig = t_sgmt;
    //     o_figura.tfig.sgmt = un_segment;
    // ...
    return 0;
}
```

# Operaţii bit cu bit

- Se aplică expresiilor întregi
- Complement ~ `b = ~a;`
- Conjuncţie & `c = a & b;`
- Disjuncţie | `c = a | b;`
- Sau exclusiv ^ `c = a ^ b;`
- Deplasare (shift) stânga <<

```
b = a << 5;   x <<= 3;
```

- Deplasare (shift) dreapta >>

```
b = a >> 5;   x >>= 3;
```

- Mască: constantă utilizată pentru a extrage biţii convenabili: `1, 255=2^8-1`

# Operatorii bit cu bit - precedenţa

- **~** are aceeași precedenţă cu **!**, asociativitate dreapta

- **<<** şi **>>** după **+, -** şi
  înainte de **<, <=, >, >=**

- **&, ^, |** în această ordine după **==** şi **!=**, înainte de **&&**

# Operaţii bit cu bit – Exemplul 1.1

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

    void print_bit_cu_bit(int x, const char* s)
    {
        int i;
        int n = sizeof(int)*CHAR_BIT;
        int mask = 1 << (n - 1);
        cout << s;
        for (i = 1; i <= n; i++) {
            cout << ((x & mask) == 0) ? '0' : '1';
            x <<= 1;
            if (i%CHAR_BIT == 0 && i<n)
                cout << ' ';
        }
        cout << "\n";
    }
```

# Operaţii bit cu bit – Exemplul 1.2

```
int main(){
    int a = 0xA5b73, b = 0Xb0c8722;
    int c = ~a, d = a&b, e = a | b, f = a^b;
    print_bit_cu_bit(a, "  a = ");
    print_bit_cu_bit(b, "  b = ");
    print_bit_cu_bit(c, " ~a = ");
    print_bit_cu_bit(d, "a&b = ");
    print_bit_cu_bit(e, "a|b = ");
    print_bit_cu_bit(f, "a^b = ");
    print_bit_cu_bit(a << 3, "a<<3= ");
    print_bit_cu_bit(b >> 6, "b>>6= ");
}
```

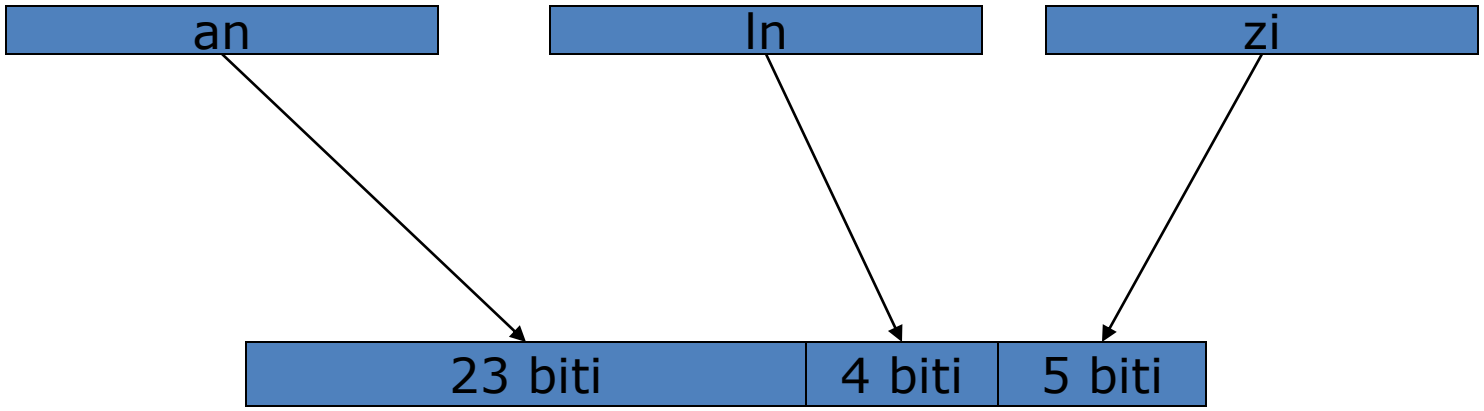
# Operaţii bit cu bit – Exemplul 1.3

```
/*
a = 00000000 00001010 01011011 01110011
b = 00001011 00001100 10000111 00100010

~a = 11111111 11110101 10100100 10001100
a&b = 00000000 00001000 00000011 00100010
a|b = 00001011 00001110 11011111 01110011
a^b = 00001011 00000110 11011100 01010001
a<<3= 00000000 01010010 11011011 10011000
b>>6= 00000000 00101100 00110010 00011100
*/
```

# Exemplul 2 - Împachetare

```
unsgn pack(unsgn an, unsgn ln, unsgn zi, unsgn *id)
{
    *id = an;
    *id = (*id << 4) | ln;
    *id = (*id << 5) | zi;
    return *id;
}
```



# Exemplul 2 - Despachetare

```
void unpack(unsgn *an, unsgn *ln, unsgn *zi, unsgn id)
{
    /*
    *an = (id & (0xFFFF << 9)) >> 9;
    *ln = (id & (15 << 5)) >> 5;
    *zi = id & 31;
    */
    *zi = id & 31;
    id >>= 5;
    *ln = id & 15;
    id >>= 4;
    *an = id;
}
```

# Câmpuri de biţi în structuri

- O declarație, într-o structură, de forma:

  *tip* var:n;  tip ∈ {**int, unsigned**}

  stabilește reprezentarea lui **var** pe n biți:

```
typedef struct {
   unsigned oct0 : 8, oct1 : 8, oct2 : 8, oct3 : 8;
} cuv_oct; /* un cuvant = 4 octeti */
typedef struct {
   unsigned b0 : 1, b1 : 1, ..., b31 : 1;
} cuv_bit; /* un cuvant = 32 biti */
typedef union {
   unsigned i;
   cuv_oct oct;
   cuv_bit bit;
} cuvant;
```

# Câmpuri de biţi

- Mărimea câmpului de biţi nu trebuie să depăşească numărul de biţi într-un cuvânt (32)
- Câmpurile de biţi se declară ca membri consecutivi într-o structură; compilatorul îi împachetează într-un număr minim de cuvinte
- Nu este posibilă declararea de tablouri de câmpuri de biţi
- Nu se poate folosi operatorul adresă **&** pentru câmpuri de biţi
- Se pot folosi câmpuri de biţi, fără nume, pentru aliniere:

```
struct small_int{
      unsigned i1 : 7, i2 : 7, i3 : 7,
      : 11,  // aliniere la cuv urmator
      i4 : 7, i5 : 7, i6 : 7;
};
struct abc{
      unsigned a : 1, : 0, b : 1, : 0, c : 1;
}
```

# Aplicaţie – operaţii cu mulţimi - 1

```
//Multimile reprezentate ca biti

#include  <stdio.h>

typedef struct word {
    unsigned w0 : 1, w1 : 1, w2 : 1, w3 : 1, w4 : 1, w5 : 1,
    w6 : 1, w7 : 1, w8 : 1, w9 : 1, w10 : 1, w11 : 1,
    w12 : 1, w13 : 1, w14 : 1, w15 : 1, w16 : 1, w17 : 1, w18
    : 1, w19 : 1, w20 : 1, w21 : 1, w22 : 1, w23 : 1, w24 : 1,
    w25 : 1, w26 : 1, w27 : 1, w28 : 1, w29 : 1, w30 : 1, w31
    : 1;
}word;

typedef union set {
    word        m;
    unsigned  u;
}set;
```

# Aplicaţie – operaţii cu mulţimi - 2

```
int main(){
   set  x, y;

   x.u = 0x0f100f10;
   cout << "elementul 9 din x = “;
   cout << ((x.m.w9) ? "true" : "false") << "\n";
   y.u = 0x01a1a0a1;
   cout << "elementul 9 din y = ";
   cout << ((y.m.w9) ? "true" : "false") << "\n";
   x.u |= y.u;  /* reuniune de multimi */
   cout << "elementul 9 din x reunit y = ";
   cout << ((x.m.w9) ? "true" : "false");
   return 0;
}
```

# Fişiere

- Un fişier poate fi privit ca un “stream” (flux) de caractere.
- Un fişier are un nume
- Pentru a putea fi accesat un fişier trebuie “deschis”
- Sistemul trebuie să ştie – programatorul îi comunică – ce operaţii pot fi făcute cu un fişier:
  - se deschide pentru citire – fişierul trebuie să existe
  - se deschide pentru scriere – fişierul se crează
  - se deschide pentru adăugare – fişierul există şi se modifică
- După prelucrare fişierul trebuie închis

# fstream.f

- Defineşte clasele:
  - `ifstream`: clasa stream-urilor de intrare
  - `ofstream`: clasa stream-urilor de ieşire
  - `fstream`: clasa stream-urilor de intrare/ieşire
- Declaraţia unui stream:

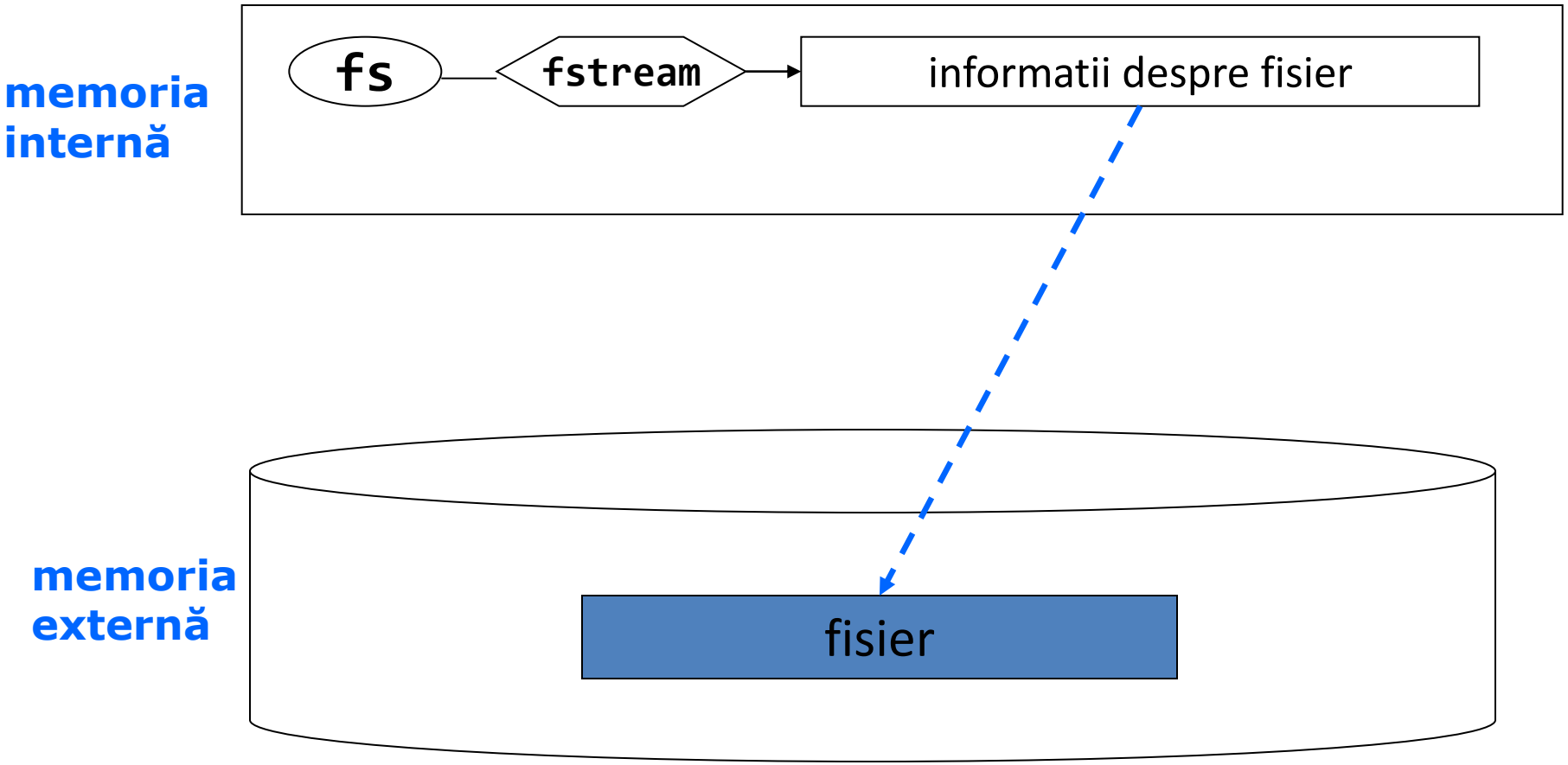
```
ifstream intrare;
```

- Asocierea stream-ului unui fişier fizic (pe disc):

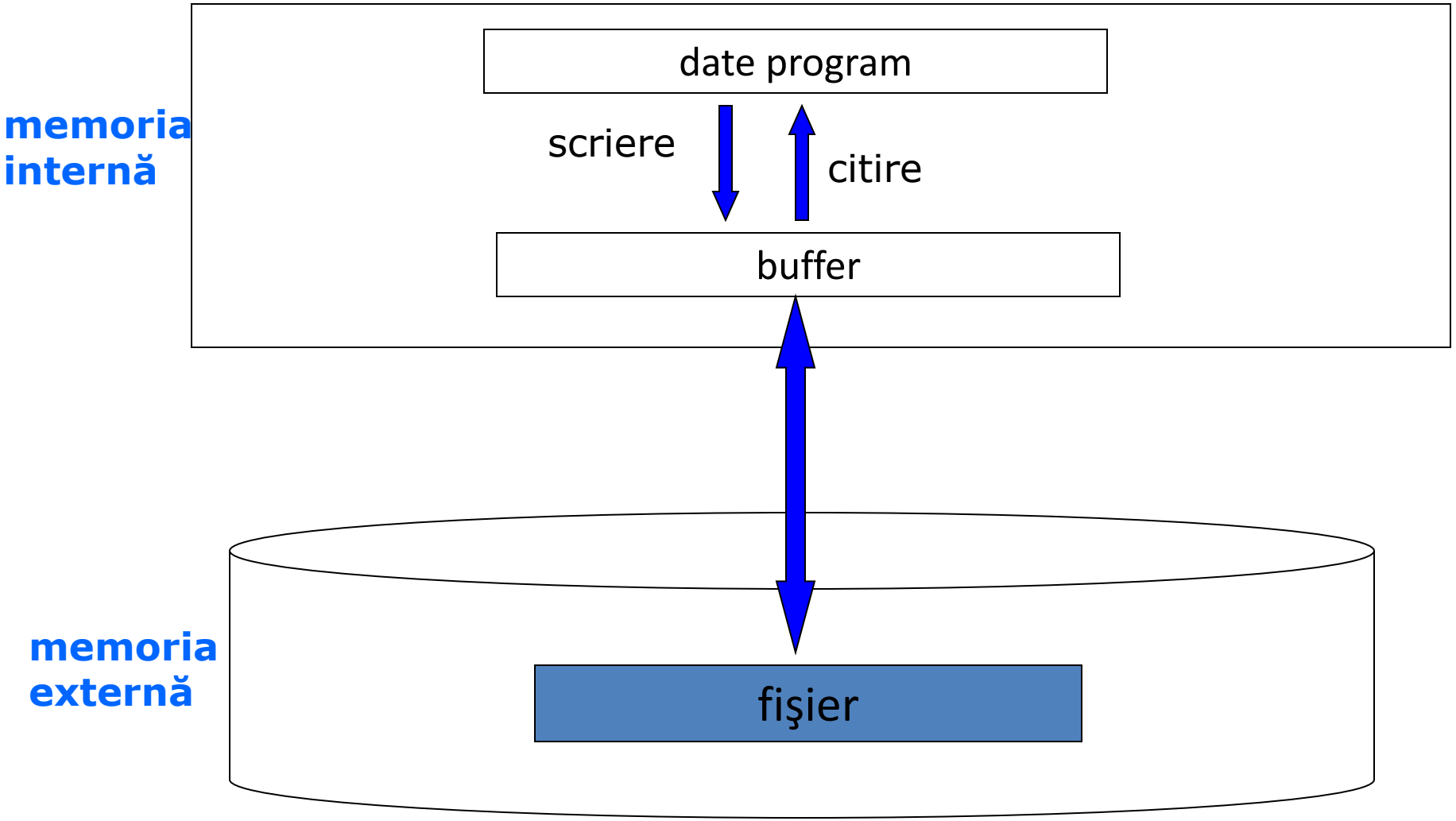
```
void open(const char *filename, int mode);
```

  - *ios::app* – adăugare la sfârşit
  - *ios::ate* – căutarea sfârşitului fişierului
  - *ios::binnary* – deschiderea fişierului în mod binar
  - *ios::in* – fişierul acceptă intrări
  - *ios::out* – fişierul acceptă ieşiri
  - *ios::nocreate, ios::noreplace, ios::trunc*

# Fişiere


memoria internă
fs
fstream
informatii despre fisier
memoria externă
fisier

# Fişiere – citire/scriere



# Funcţiile `close(), flush()`

```
mystream.close();
```

- Realizează cele necesare pentru a închide un fişier: goleşte buffer-ul şi întrerupe orice legătură între fişier şi stream-ul *mystream.*
- Nu preia nici un paramentru şi nu returnează nici o valoare.

```
mystream.flush()
```

- Golirea bufferului: datele din buffer sunt scrise în fişier

# Fişiere de text. Exemplul 1

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

int main(){
    ofstream  out;
    out.open("d:\\rezultate.txt");
    if (!out) {
        cout << "Eroare fisier!\n";  return 1;
    }
    out << "Ionescu " << 7.50 << endl;
    out << "Popescu " << 9.75 << endl;
    out << "Georgescu " << 8.25 << endl;

    out.close();
    return 0;
}
```

# Fişiere de text. Exemplul 2

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

int main(){
    ifstream  in;
    in.open("d:\\rezultate.txt");
    if (!in) {
         cout << "Eroare fisier!\n";  return 1;
    }
    char nume[20]; float nota;
    for (int i = 1; i <= 3; i++){
         in >> nume >> nota;
         cout << nume << " " << nota << '\n';
    }
    in.close();return 0;
}
```

# I/O de tip binar

```
istream & get(char &ch);
ostream & put(char ch);
```

- **`get()`** citeşte un singur caracter din stream-ul asociat şi memorează valoarea în *ch*
- **`put()`** scrie *ch* în stream-ul asociat

# Exemplul 3

```
/*Copiere fisier cu modificarea literelor mici in litere mari*/
#include <iostream>
#include <fstream>
using namespace std;

int main(){
    char   c, file_name[128];
    ifstream  ifs;
    ofstream  ofs;
    cout << "\nIntrodu numele unui fisier:  "; cin >> file_name;
    ifs.open(file_name);
    if (!ifs) {
          cout << "Eroare la deschiderea fisierului\n";
          return 1;
    }
    ofs.open("d:\\copie.out");
    while (ifs.get(c)) {
          if (islower(c)) c = toupper(c);
          ofs.put(c);
    }
    ifs.close(); ofs.close(); return 0;
}
```

# Citirea/scrierea blocurilor de date

```
istream& read(char * buf, int nr);
```

- Se citesc cel mult *nr* octeţi (caractere) din stream-ul asociat şi se pun în buffer-ul indicat de *buf*.

```
istream& write(const char * buf, int
  nr);
```

- Scrie în stream-ul asociat *nr* octeţi citiţi din buffer-ul indicat de *buf*.

# Exemplul 4

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

int main(){
      float tab[3] = { 7.50, 9.75, 8.25 };
      int i;

      ofstream  out;
      out.open("d:\\note.dat", ios::out | ios::binary);
      if (!out) { cout << "Eroare fisier!\n";  return 1; }

      out.write((const char *)&tab, sizeof(tab));
      out.close();
      for (i = 0; i<3; i++) tab[i] = 0.0;

      ifstream  in;
      in.open("d:\\note.dat", ios::in | ios::binary);
      in.read((char *)&tab, sizeof(tab));

      for (i = 0; i<3; i++) cout << tab[i] << " ";

      in.close();
      return 0;
}
```

# Alte funcţii

```
istream &get(char *buf, int nr,
                        char delim ='\n');
```

- Citeşte în buffer-ul *buf* cel mult *nr* octeţi sau până se întâlneşte caracterul *delim*.

```
int get();
```

- Returnează caracterul următor din stream sau EOF dacă se întâlneşte sfârşitul fişierului.

```
istream &getline(char *buf, int nr,
                        char delim ='\n');
```

- Faţă de `get()`, extrage şi delimitatorul din stream

# Alte funcţii

```
int eof();
```

- Returnează nonzero când a fost atins sfârşitul fişierului; altfel returnează zero.

```
int peek();
```

- Returnează caracterul următor din stream sau EOF fără a-l extrage din stream.

```
istream &putback(char ch);
```

- Returnează înapoi în stream caracterul *ch*.

# Funcţii de acces aleator

```
istream &seekg(streamoff offset, seek_dir org);
istream &seekp(streamoff offset, seek_dir org);
```

- Deplasează pointerii *get* şi *put* pentru următoarea operaţie I/O cu *offset* octeţi faţă de *org*.

- Valoare lui *org* poate fi:
  - `ios::beg` – început de fişier
  - `ios::cur` – locaţia curentă
  - `ios::end` – sfârşit de fişier

- Exemple (pentru pointerul *get*):
  - poziţionarea la sfârşitul fişierului
    ```
    mystream.seekg(0, ios::end)
    ```
  - poziţionarea la caracterul precedent
    ```
    mystream.seekg(-1, ios::cur)
    ```
  - poziţionarea la începutul fişierului
    ```
    mystream.seekg(0, ios::beg)
    ```

# Alte funcţii

```
streampos tellg();
streampos tellp();
```

- Determină poziţia curentă a pointerilor *get* şi *put*.

- Starea I/O:
  - **eofbit, failbit, badbit**
  - `int eof(); int fail(); int bad(); int good();`

```
void clear(int indicatori=0);
```

- resetează indicatorii de eroare şi end-of-file

# Exemplul 5

```
#include <iostream>
#include <fstream>
using namespace std;
// Afisarea unui fisier de la sfarsit

int main(){
     char   c, file_name[128];
     ifstream  ifs;

     cout << "\nIntrodu numele unui fisier:  ";  cin >> file_name;
     ifs.open(file_name, ios::in | ios::binary);
     if (!ifs) { cout << "Eroare fisier!\n";  return 1; }

     ifs.seekg(0, ios::end); // pozitionare la sfarsit
     ifs.seekg(-1, ios::cur); // pozitionare la ultimul octet
     while (ifs.tellg() >= ios::beg) {
          ifs.get(c); cout << c;
          ifs.seekg(-2, ios::cur); //octetul anterior
     }
     ifs.close();
     return 0;
}
```