Programare in limbaj C

- Bibliografie
  - Brian W. Kernighan, Dennis M. Ritchie Programarea in limbaj C
  - Al Kelley, Ira Pohl: A Book on C - Programming in C, Addison Wesley, Reading
  - Herbert Schildt: C Manual Complet, Bucuresti, Ed. Teora 1998
  - E. Horowitz, S. Sahni, S. Anderson - Freed: Fundamentals of Data Structures in C, Computer

Primul program C

```
#include <stdio.h>
int main(void)
{
  printf("Salut!\n");
  printf("Iata primul program C!");
  return 0;
}
```

Caracterele limbajului C

- Litere:

A B C D … X Y Z

a b c d … x y z

- Cifre: 0 1 2 3 4 5 6 7 8 9
- Alte caractere:

+ - * / = ( ) { } [ ] < > ‘ “ ! # % & _

|^ ~ \ . , ; : ?

- Caractere spaţiu: blank, newline,tab, etc.

Tipuri standard

- Tipul `char`
- Tipurile standard întregi:
  - 5 tipuri întregi cu semn: `signed char`, `short int`, `int`, `long int`, şi `long long int`.
  - 5 tipuri întregi fără semn: desemnate de cuvântul `unsigned` (tipuri ce ocupă aceeaşi cantitate de memorie)
- Tipuri reale flotante: `float`, `double`, şi `long double`.

Tipuri de date standard

ECHIVALENTE

`signed short int`	≡	`short`
`unsigned short int`	≡	`unsigned short`
`signed int`	≡	`int`
`unsigned int`	≡	`unsigned`
`signed long int`	≡	`long`
`unsigned long int`	≡	`unsigned long`

Declaraţii

- Forma unei declaraţii:

```
tip variabila;
tip var1, var2, …, varn;
tip variabila = expresie_constanta;
```

- Variabile globale: declararea lor se face la începutul programului, în afara oricarei funcţii.
- Variabile locale: declararea se face în corpul funcţiei, la început.

```
char c;
signed char sc;

int i;
int suma = 0;
long j;

float x;
float pi = 3.14;
double y;
```

Tipul întreg

- **int**

  sizeof(int) = 2 sau **4** octeti
- **short int** sau **short**

  sizeof(short)=2, {-32768,…,32767}
- **long int** sau **long**

  sizeof(long) = 4

  {-2 147 483 648, …, 2 147 483 647}
- **signed int, signed short int, signed long int**

Întregi fără semn

- **`unsigned int`**
- **`unsigned short int`**

  {0,…,65535}
- **`unsigned long int`**

- Nu există overflow (depăşire) – calculul se efectuează modulo 2^n, unde n este numărul de biţi

Întregi “foarte scurţi”: **`char`**

- Tipul **`char`** este o submulţime a tipului **`int`**
- **`char`** reprezintă, în funcţie de maşină, domeniul de valori:

  {-128, …, 127} sau {0,…,256}
- **`unsigned char`** {0,…,256}
- **`signed char`** {-128, …, 127}

`sizeof(char) = 1`

Constante întregi în <limits.h>

	16 biti	32 biti
INT_MAX	$2^{15}-1$	$2^{31}-1$
INT_MIN	-2^{15}	-2^{31}
LONG_MAX	$2^{31}-1$	$2^{63}-1$
LONG_MIN	-2^{31}	-2^{63}

Atenţie la reprezentarea circulară!

- INT_MAX + 1 == INT_MIN
- INT_MIN – 1 == INT_MAX
- LONG_MAX + 1 == LONG_MIN
- LONG_MIN – 1 == LONG_MAX

Citiri, afişări

- citirea unui int

```
printf("a: ");
scanf("%d", &a);
```

- afişarea unui int

```
int a = 10;
printf("a = %d", a);
```

- citirea unui char

```
printf("a: ");
scanf("%c", &a);
```

- afişarea unui char

```
char a = 'a';
printf("a = %c", a);
```

Constante - exemplu

```
/* Exemple de constante caracter */
#include <stdio.h>
int main(){
    char a, b, c, d;
    a = 'A'; b = 65; c = '\101'; d = '\x41';
    printf("%c %c %c %c\n", a, b, c, d);
    printf("%c %d %o %x\n", a, a, a, a);
    return 0;
}
/*
A A A A
A 65 101 41
*/
```

Codurile ASCII

```
# include <stdio.h>
int main (void){
   short c;
   for(c=0; c<= 127; c++){
      printf("cod ASCII:%d",c);
      printf(" caracter:%c\n",c);
   }
   return 0;
}
/* for(c='a'; c<='z'; c++) */
```

Macrourile getchar() si putchar()

- Sunt definite in **<stdio.h>**
- Citire caracter de la tastatură
- Scriere caracter pe ecran

```
#include <stdio.h>
int main(void){
  char c;
  while ((c=getchar()) != EOF)
   {putchar(c); putchar(c);}
  return 0;
}
//123456abcd
//112233445566aabbccdd
```

Operaţii, Funcţii în biblioteci

- Operaţii pentru tipurile întregi:

```
+    -    *    /    %
==    !=   <    <=       >      >=
++   --
```

- Funcţii:
  - cele de la tipul flotant
  - cele din biblioteca **<ctype.h>: tolower, toupper, isalpha, isalnum, iscntrl, isdigit, isxdigit, islower, isupper, isgraph, isprint, ispunct, isspace**

Operatorii ++ şi --

- Se aplică doar unei expresii ce desemnează un obiect din memorie (L-value):

Expresie:	++i	i++	--i	i--
Valoare:	i+1	i	i-1	i
i dupa evaluare:	i+1	i+1	i-1	i-1

++5 --(k+1) ++i++ nu au sens

Tipul flotant (real)

- **`float`**
  - Numere reale în simplă precizie
  - **`sizeof(float)`** = 4
  - 10^{-37} <= abs(f) <= 10^{38}
  - 6 cifre semnificative
- **`double`**
  - Numere reale în dublă precizie
  - **`sizeof(double)`** = 8
  - 10^{-307} <= abs(f) <= 10^{308}
  - 15 cifre semnificative

Tipul flotant (real)

- **`long double`**
  - Numere reale în “extra” dublă precizie
  - **`sizeof(float)`** = 12
  - 10^{-4931} <= abs(f) <= 10^{4932}
  - 18 cifre semnificative
- Limitele se găsesc în `<float.h>`
- Operaţii:

```
+    -    *    /
==   !=   <    <=   >    >=
```

Constante reale

- Constantele reale sunt implicit **`double`**

`125.435  1.12E2  123E-2  .45e+6  13.  .56`

- Pentru a fi **`float`** trebuie sa aiba sufixul f sau F

`.56f  23e4f  45.54E-1F`

- Pentru long double trebuie sa aiba sufixul l sau L `123.456e78L`

Funcţii (în biblioteca <math.h>)

sin cos tan asin acos
atan sinh cosh tanh exp
log log10 pow sqrt ceil floor
fabs ldexp frexp modf fmod

Citire, afişare

- citirea unui float

```
printf("x: ");
scanf("%f", &x);
```

- afişarea unui float

```
float pi = 3.14;
printf("pi = %f", pi);
```

- citirea unui double

```
printf("x: ");
scanf("%lf", &x);
```

- afisarea unui double

```
double pi = 3.14L;
printf("pi = %lf", pi);
```

Utilizare `typedef`

- Mecanism prin care se asociază un tip unui identificator:

```
typedef char   litera_mare;
typedef short  varsta;
typedef unsigned long    size_t;
```

- Identificatorul respectiv se poate utiliza pentru a declara variabile sau funcţii:

```
litera_mare    u, v='a';
varsta         v1, v2;
size_t         dim;
```

Date booleene (logice)

- Nu exista un tip special pentru date logice;
- Domeniul de valori: {false, true}
- false = 0
- true = orice întreg nenul
- Operaţii:

! && || == !=

- O declaraţie posibilă:

```
typedef enum {false = 0, true = 1} bool;
bool x, y;
```

Expresii logice

expresie_relationala ::=

expr < *expr* | *expr* > *expr*

| *expr* <= *expr* | *expr* >= *expr*

| *expr* == *expr* | *expr* != *expr*

expresie_logica ::= ! *expr*

| *expr* || *expr*

| *expr* && *expr*

Valoarea expresiilor relaţionale

a-b	a<b	a>b	a<=b	a>=b	a==b	a!=b
pozitiv	0	1	0	1	0	1
zero	0	0	1	1	1	0
negativ	1	0	1	0	0	1

Valoarea expresiilor logice ||

exp1	exp2	exp1 || exp2
$\neq 0$	Nu se evaluează	1
=0	Se evaluează	1 dacă exp2 $\neq 0$ 0 dacă exp2 = 0

Valoarea expresiilor logice &&

exp1	exp2	exp1 && exp2
= 0	Nu se evaluează	0
≠ 0	Se evaluează	1 dacă exp2 ≠ 0 0 dacă exp2 = 0

Exemple

- O condiţie de forma a ≤ x ≤ b se scrie în limbajul C:

  `(x >= a) && (x <= b)` sau

  `a <= x && x <= b`

- O condiţie de forma a > x sau x > b se scrie în limbajul C:

  `x < a || x > b` sau

  `!(x >= a && x <= b)`

Operatorul condiţional `?:`

exp1 `?` *exp2* `:` *exp3*

- Se evaluează *exp1*
- Dacă *exp1* are valoare nenulă (true) atunci valoarea expresiei este valoarea lui *exp2*; *exp3* nu se evaluează
- Dacă exp1 are valoare nulă (false) atunci valoarea expresiei este valoarea lui *exp3*; *exp2* nu se evaluează
- Operatorul `?:` este drept asociativ

Operatorul condiţional ?: Exemple

```
x >= 0 ? x : y
x > y ? x : y
x>y ? x>z ? x :z : y>z ? y : z
```

```
#include <stdio.h>
int main(void){
    int a=1, b=2, c=3;
    int x, y, z;
    x = a?b:c?a:b;
    y = (a?b:c)?a:b; /* asociere stanga */
    z = a?b:(c?a:b); /* asociere dreapta */
    printf("x = %d, y = %d, z = %d\n", x, y, z);
}
 /* x = 2, y = 1, z = 2 */
```

Operatorul “=“ (Expresia de atribuire)

- Expresia de atribuire:

  exp1 = exp2

  - *exp1* este o “L-value” (obiect din memorie: variabilă, variabilă tablou cu indici, etc.)
  - Tipul expresiei este tipul lui *exp1*
  - Se evaluează *exp2* apoi *exp1* capătă valoarea lui *exp2,* eventual convertită.
  - Aşadar, operatorul = modifică valoarea operandului stâng
  - Valoarea expresiei este valoarea lui *exp1* după evaluare
  - Operatorul = este drept asociativ

Operatorul “=“ (Expresia de atribuire)

- Exemple:

  x = sqrt(9)

  a = (b = 2) + (c = 3)

  a = b = c = 0 echivalenta cu

  a = (b = (c = 0))

  while((c = getchar()) != EOF) putchar(c);

- Nu confundaţi *e1* = *e2* cu *e1* == *e2*

```
a = 22;
if ( a == 0)printf(“nul”)
else printf(“nenul”);          /* nenul */

if ( a = 0)printf(“nul”)
else printf(“nenul”);          /* nul */
```

Operatori de atribuire compusă

- O expresie de atribuire compusă are forma:

exp1 op= exp2

unde *op=* este unul din:

+= -= *= /= %= &= |= ^= >>= <<=

- Expresia este echivalentă cu

exp1 = exp1 op (*exp2*)

cu precizarea că *exp1* se evaluează o singură dată.

`j *= k + 3` echivalentă cu: **`j = j*(k+3)`**

`j *= k = m + 5` echivalentă cu:

```
j = (j * (k = (m + 5)))
```

Operatorul virgulă,

expresia_virgula ::= *expresie*, *expresie*

- Se evaluează prima expresie apoi cea de-a doua.
- Valoarea şi tipul întregii expresiii este valoarea şi tipul operandului drept.
- Operatorul virgulă are cea mai mică precedenţă.

- Exemple:

```
a = 1, b = 2
i = 1, j = 2, ++k + 1
k != 1, ++x * 2.0 + 1
for(suma = 0, i = 1; i <= n; suma += i, ++i);
```

Tipul void

- Conversia în tip **`void`** a unei expresii semnifică faptul că valoarea sa este ignorată
- Utilizat pentru tipul pointer; nu se face controlul tipului la un pointer de tip **`void`**
- Utilizat pentru funcţii fără valoare returnată sau pentru funcţii fără parametri
- Este un tip incomplet ce nu poate fi completat

Operatorul `sizeof()`

- Operator unar ce permite găsirea numărului de octeţi pe care se reprezintă un obiect (tip, expresie)

```
sizeof(int), sizeof(double)
sizeof(b*b-4*a*c), sizeof(i)
```

sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)

sizeof(signed)=sizeof(unsigned) = sizeof(int)

sizeof(float)<=sizeof(double)<=sizeof(long double)

Operatorul `sizeof()`

```
#include<stdio.h>
int main(void){
  int x=1; double y=9; long z=0;
  printf("Operatorul sizeof()\n\n\n");
  printf("sizeof(char)          = %2u\n",sizeof(char));
  printf("sizeof(int)           = %2u\n",sizeof(int));
  printf("sizeof(short)         = %2u\n",sizeof(short));
  printf("sizeof(long)          = %2u\n",sizeof(long));
  printf("sizeof(float)         = %2u\n",sizeof(float));
  printf("sizeof(double)        = %2u\n",sizeof(double));
  printf("sizeof(long double)   = %2u\n",sizeof(long
   double));
  printf("sizeof(x +y + z)      = %2u\n",sizeof(x+y+z));
  printf("sizeof(void)          = %2u\n",sizeof(void));
  return 0;
}
```

Operatorul `sizeof()`

Rezultatul executiei Visual C++ (Djgpp):

```
sizeof(char)          =  1
sizeof(int)           =  4
sizeof(short)         =  2
sizeof(long)          =  4
sizeof(float)         =  4
sizeof(double)        =  8
sizeof(long double)   =  8  ( 12 Djgpp)
sizeof(x +y + z)      =  8
sizeof(void)          =  0  (1 Djgpp)
```

Precedenţa operatorilor

OPERATORI	**ASOCIERE**
() ++ -- (postfix)	stânga
++ -- *(prefix)* ! & *(adresa)* * *(deref)* + - *(unari)* sizeof()	dreapta
*** / %**	stânga
+ -	stânga
< <= > >=	stânga
== !=	stânga
&&	stânga
||	stânga
?:	dreapta
= += -= *= /= %=	dreapta
, (operatorul virgulă)	stânga

Forţarea tipului - cast

- Conversia explicită la tipul *numetip*:

  (*numetip*) *expresie*

- Exemple:

```
(long)('A' + 1.0)
(int)(b*b-4*a*c)
(double)(x+y)/z
(float)x*y/z
x/(float)2
```

Exemplu cast

```
#include <stdio.h>
int main(void){
  int i, j; double x, y, z, t;
  i=5/2; x=5/2; y=(double)(5/2);
  j=(double)5/2; z=(double)5/2;
  t=5./2;
  printf("%d, %g, %g, %d, %g, %g\n",
              i,  x,  y,  j,  z,  t);
}
/* 2, 2, 2, 2, 2.5, 2.5  */
```

Fişiere în bibliotecă relative la tipuri

- <limits.h> - pentru tipurile întregi
  - Întregul min/max: INT_MIN, INT_MAX
  - Numărul de biţi pe caracter CHAR_BIT
  - Etc.
- <float.h> - pentru tipurile flotante:
  - Exponentul maxim
  - Precizia zecimală, etc.
- <stdlib.h> - conţine funcţii de conversie:
  - Şir de caractere în **`int`** : **`atoi(const char*)`**
  - Şir de caractere în **`float:atof(const char*)`**

Instrucţiuni

- Expresii: ; *expresie*;
- Compuse (bloc): {*declaraţii instrucţiuni*}
- Condiţionale: *if* *if-else* *switch-case*
- Iterative: *for* *while* *do-while*
- Intreruperea secvenţei:
  `continue; break; return expr;`
- Salt necondiţionat: **`goto`**

Instrucţiunea expresie

instr_expresie ::= {*expresie*}$_{opt}$;

- Exemple:

```
a = b;
a + b + c;
;
printf("a= %d\n", a);
scanf("%d%f%c", &i, &x, &c);
```

Instrucţiunea compusă (bloc)

instr_compusa ::=

{ {*lista_declaratii*}$_{0+}$ {*lista_instructiuni*}$_{0+}$ **}**

- Grupează instrucţiuni într-o unitate executabilă.
- Dacă sunt şi declaraţii la început, instrucţiunea compusă se numeşte şi *bloc.*
- O instrucţiune compusă este ea insăşi o instrucţiune: oriunde poate să apară o instrucţiune, este corect să apară şi o instrucţiune compusă.

Instrucţiunea compusă - Exemple

```
{
    a += b += c;
    printf("a = %d, b = %d, c = %d\n, a, b, c);
}
if(x > y){
    int temp;
    temp = x; x = y; y = temp;
}
{
    int a, b, c;
    {
          b = 2;c = 3;a = b += c;
    }
    printf("a= %d", a);
}
```

Instrucţiunile condiţionale if şi if-else

instr_if ::= **`if (`***expr***`)`** *instructiune*

instr_if-else ::= **`if (`***expr***`)`** *instructiune*
`else` *instructiune*

expr este o condiţie construită cu:

- Expresii aritmetice
- Comparatori: ==, !=, <, <=, >, >=
- Conectori logici: &&, ||, !

Instrucţiunile condiţionale if şi if-else

Exemple:

```
if(b == a) aria = a*a;

if(x < y)
   min = x;
else
   min = y;

if(a%2) if(b%2) p = 1; else p = 2;

if(a%2){
   if(b%2) p = 1;
}else p = 2;
```

Instrucţiunile if şi if-else - Exemple

```
if (i>j)
   if(k>l)
        if(i>k) max = i;
        else max = k;
   else
        if(i>l) max = i;
        else max = l;
else
   if(k>l)
        if(j>k) max = j;
        else max = k;
   else
        if(j>l) max = j;
        else max = l;
```

“Dangling else Problem”

```
if (a == 1)
     if (b == 2)
          printf("*****\n");
else
     printf("ooooo\n");
```

- Nu lăsaţi forma codului să vă ducă în eroare! Regula este: **`else`** este ataşat celui mai apropiat **`if`**.

If-else-exemplu

```
int main(void){
    float x, y, rezultat;
    char operator;
    printf("Expresia:(numar operator numar)\n");
    scanf("%f %c %f", &x, &operator, &y);
    if(operator == '+')
          rezultat = x+y;
    else if(operator == '-')
          rezultat = x-y;
    else if(operator == '*')
          rezultat = x*y;
    else if(operator == '/')
          rezultat = x/y;
    else{
          printf("Eroare in scrierea expresiei!");
          return 1;
    }
    printf("Rezultatul este: %f\n", rezultat);
    return 0;
}
```

Instrucţiunea switch

```
switch(expresie_intreaga){
  case exp_const1: instr1
  case exp_const2: instr2
  …
  case exp_constn: instrn
  default: instructiune
}
```

Instrucţiunea switch

- Valoarea expresiei *expresie_intreaga,* care este de tip **`int`**, se compară cu constantele *exp_const.*
- În caz de egalitate se execută instrucţiunea corespunzătoare si toate cele ce urmează. Există posibilitatea de ieşire cu instrucţiunea **`break`**.
- Dacă valoarea determinată diferă de oricare din constantele specificate, se execută instrucţiunea specificată la **`default`**, care apare o singură dată, nu neaparat la sfârşit. Dacă **`default`** lipseşte se iese din **`switch`**.
- Valorile constantelor trebuie sa fie diferite; ordinea lor nu are importanţă.
- Acoladele ce grupeaza mulţimea **`case`**-urilor sunt obligatorii. După fiecare **`case`** pot apare mai multe instrucţiuni fără a fi grupate în acolade.

Instrucţiunea switch - exemple

```
scanf("%d", &i);
switch(i){
   case 1: printf(" 1");
   case 2: printf(" 2");
   case 3: printf(" 3");
   case 4: printf(" 4");
   default: printf(" blabla! ");
}
```

2

2 3 4 blabla!

Instrucţiunea switch - exemple

```
scanf("%d", &i);
switch(i){
   case 1: printf(" 1"); break;
   case 2: printf(" 2"); break;
   case 3: printf(" 3"); break;
   case 4: printf(" 4"); break;
   default: printf(" blabla! ");
}
```

2

```
   2
```

Instrucţiunea switch - exemple

```
switch (nota)
    {
    case 1:
    case 2:
    case 3:
    case 4:
      printf("Nota nesatisfacatoare.");
      break;
    //...
    }
```

Instrucţiunea while

instructiunea_while ::=
while(*expresie*) *instructiune*

while (*expresie*){
instructiune
}
instructiunea_urmatoare

- Se evaluează *expresie:* dacă valoarea sa este nenulă se execută *instructiune* şi controlul este transferat înapoi, la începutul instrucţiunii while; dacă valoarea este nulă se execută *instructiunea_urmatoare.*

- Aşadar *instructiune* se execută de zero sau mai multe ori.

Instrucţiunea while

```
while (i++ < n)
  factorial *= i;

while((c = getchar()) != EOF){
  if(c >= 'a' && c <= 'z')
     ++ contor_litere_mici;
  ++contor_total;
}
```

Instrucţiunea do..while

instructiunea_do..while ::=
do *instructiune* **while**(*expresie*);

```
do{
        instructiune
} while (expresie);
instructiunea_urmatoare
```

- Se execută *instructiune.*
- Se evaluează *expresie:* dacă valoarea sa este nenulă controlul este transferat înapoi, la începutul instrucţiunii do..while; dacă valoarea este nulă se execută *instructiunea_urmatoare.*
- Aşadar *instructiune* se execută o dată sau de mai multe ori.

Instrucţiunea do..while

```
do{
  c = getchar();
} while(c == ' ');

do{
  printf("Introdu un intreg pozitiv:");
  scanf("%d", &n);
  if(error = (n <= 0))
     printf("\nEroare! Mai incearca!\n");
}while(error);
```

Exemplu - calculator

```
#include <stdio.h>
int main(void){
    float x, y, rezultat;
    char operator, c;
    int ERROR;
    printf("Calculator pentru expresii de forma
                    numar operator numar\n");
    printf("Folositi operatorii + - * / \n");
```

Exemplu - calculator

```
do{
        ERROR = 0;
        printf("Expresia: ");
        scanf("%f %c %f", &x, &operator, &y);
        switch(operator){
          case '+': rezultat = x+y; break;
          case '-': rezultat = x-y; break;
          case '*': rezultat = x*y; break;
          case '/': if(y != 0) rezultat = x/y;
                    else { printf("Impartire prin zero!\n");ERROR = 1;}
                    break;

          default : {printf("Operator necunoscut!\n");ERROR = 1;}
        }
        if(!ERROR)
          printf("%f %c %f = %f \n", x, operator, y, rezultat);
        do{ printf("Continuati (d/n)?"); c = getchar();
        } while (c != 'd' && c != 'n');
  } while (c != 'n');
  printf("La revedere!\n");
  return 0;
}
```

Instrucţiunea for

instructiunea_for ::=

for (*expr1; expr2; expr3*) *instructiune*

for (*expr1; expr2; expr3*){
    instructiune
}
instructiunea_urmatoare

- Una, doua sau toate trei dintre expresii pot lipsi, dar cei doi separatori sunt obligatorii.

Instructiunea for

- Dacă *instructiune* nu conţine **`continue`** şi *expr2* este prezentă, atunci **`for`** este echivalent cu:

```
expr1;
while(expr2){
instructiune
expr3;
}
instructiunea_urmatoare
```

- Dacă există **`continue`** atunci aceasta transferă controlul la *expr3.*

Instrucţiunea for

- Se evaluează *exp1* - în general aceasta se utilizează pentru iniţializarea iteraţiei.
- Se evaluează *exp2* - în general aceasta este o expresie logică ce se utilizează pentru controlul iteraţiei. Dacă valoarea sa este nenulă(true), se execută corpul buclei do (instrucţiune), se evaluează *exp3* si controlul este trecut la începutul buclei do, fără a se mai evalua *exp1*.
- În general *exp3* face trecerea la iteraţia următoare: modifică o variabilă ce intră în componenţa lui *exp2.*
- Procesul continuă până când valoarea *exp2* este nulă (false). Controlul este transferat următoarei instrucţiuni(cea de după for).

Instrucţiunea for - exemplu

s ← 0
for i ← 1 to n do
  s ← s + i
	s = 0;
	for (i = 1; i <= n; ++i)
	  s += i;

a ← 1
for i ← 1 to k do
  a ← a * 2
	a = 1;
	for (i = 1; i <= k; ++i)
	  a *= 2;

s ← 0
for i ← n downto 1 do
  s ← s + i
	s = 0;
	for (i = n; i > 0; --i)
	  s += i;

Instrucţiunea for - exemplu

```
#define N 100
int i, suma=0;
for(i = 1; i<=N; i++)suma+=i;

int suma, i;
for(suma = 0, i=0; i <= N; suma += ++i);

int i = 0;
char c;
for(; (c = getchar()) != '\n'; ++i)
     putchar(c);
```

Instrucţiunea for - exemplu

```
i = 1;
suma = 0;
for(;i <= N;++i) suma += i;

i = 1;
suma = 0;
for(;i <= N;) suma += i++;

i = 1;
suma = 0;
for(;;) suma += i++; // Bucla infinita
```

Instrucţiuni de întrerupere a secvenţei

- **`continue;`**
  - se referă la bucla(for, while, do..while) cea mai apropiată.
  - întrerupe execuţia iteraţiei curente şi trece controlul la iteraţia următoare.

- **`break;`**
  - se referă la bucla(for, while) sau instr. switch cea mai apropiată.
  - produce ieşirea din bucla sau din switch şi trece controlul la instrucţiunea următoare

- **`return expr;`** *sau* **`return;`**
  - în funcţii, întrerupe execuţia şi transferă controlul apelantului, eventual cu transmiterea valorii expresiei *expr.*

Exemplu – for..continue

```
/* Suma numerelor multiple de 3 pana la N */

#include<stdio.h>
#define N 100
int main(){
int i, suma=0;
for(i = 1; i<=N; i++){
   if(i%3 != 0) continue;
   suma+=i;
}
printf("suma = %d", suma);
return 0;
}
/* suma = 1683 */
```

Exemplu – while..continue

```
/* Suma numerelor multiple de 3 pana la N */
#include<stdio.h>
#define N 100
int main(){
int i=0, suma=0;
while(i<=N){
   i++;
   if(i%3 != 0) continue;
   suma+=i;
}
printf("suma = %d", suma);
return 0;
}
/* suma = 1683 */
```

Exemplu – for – continue - break

```
#include <stdio.h>

int main(void){
  for(putchar('1');putchar('2');putchar('3')){
     putchar('4');
     continue;
     // break;
     putchar('5');
  }
  return 0;
}
```

continue vs. break

```
int t;
for(;;){
  scanf("%d",&t);
  if(t==0)continue;
  printf("%d\t",t);
}
```

```
int t;
for(;;){
  scanf("%d",&t);
  if(t==0) break;
  printf("%d\t",t);
}
```

Pointeri

- Declararea unei variabile pointer:

  tip ******nume_var_pointer*;

- *nume_var_pointer* este o variabilă ce poate avea valori adrese din memorie ce conţin valori de tip *tip.*
- Exemple:

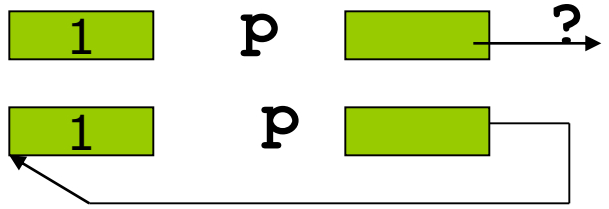
```
int *p, i;  // int *p; int i;
p = 0;
p = NULL;
p = &i;
p = (int*) 232;
```

- p “pointează la i”, “conţine adresa lui i”, “referenţiază la i”.

Pointeri

- Operatorul de dereferenţiere (indirectare) * : **`int *p;`**
  - **`p`** este pointer, **`*p`** este valoarea variabilei ce are adresa **`p`**
  - Valoarea directă a lui **`p`** este adresa unei locaţii iar **`*p`** este valoarea indirectă a lui **`p`** : ceea ce este memorat în locaţie

```
int a = 1, *p;
p = &a;
```



Pointeri

- pentru că memorează adrese, lungimile locaţiilor de memorie nu depind de tipul variabilei asociate

```
sizeof(int*) = sizeof(double*) = ...
```

- afişarea unui pointer:

```
int *px;
px = &x;
printf("%p", px);
```

Pointeri

```
int i = 3, j = 5, *p = &i, *q = &j, *r;
double x;
```

Expresia	Echivalent	Valoare
`p == &i`	`p == (&i)`	1
`**&p`	`*(*(&p))`	3
`r = &x`	`r = (&x)`	eroare!
`3**p/*q+2`	`(((3*(*p)))/(*q))+2`	3
`*(r=&j)*=*p`	`(*(r=(&j)))*=(*p)`	15

Pointeri

```
int *p; float *q; void *v;
```

Expresii corecte	Expresii incorecte
`p = 0;`	`p = 1;`
`p = (int*)1 ;`	`v = 1;`
`p = v = q;`	`p = q;`
`p = (int*)q;`	`&3;`
`q = (float*)v;`	`&(k+8);`
`*((int*)333);`	`*333;`

Pointeri

```
#include <stdio.h>
int main(void){
    int i=5, *p = &i;
    float *q;
    void *v;
    q = (float*)p;
    v = q;
    printf("p = %p, *p = %d\n", p, *p);
    printf("q = %p, *q = %f\n", q, *q);
    printf("v = %p, *v = %f\n", v, *((float*)v));
    printf("(int*)456 = %p, *((int*)456) = %d\n", (int*)456,
                                        *((int*)456));
    return 0;
}
/*
p = 8fb3c, *p = 5
q = 8fb3c, *q = 0.000000
v = 8fb3c, *v = 0.000000
(int*)456 = 1c8, *((int*)456) = 0
*/
```

Funcţii

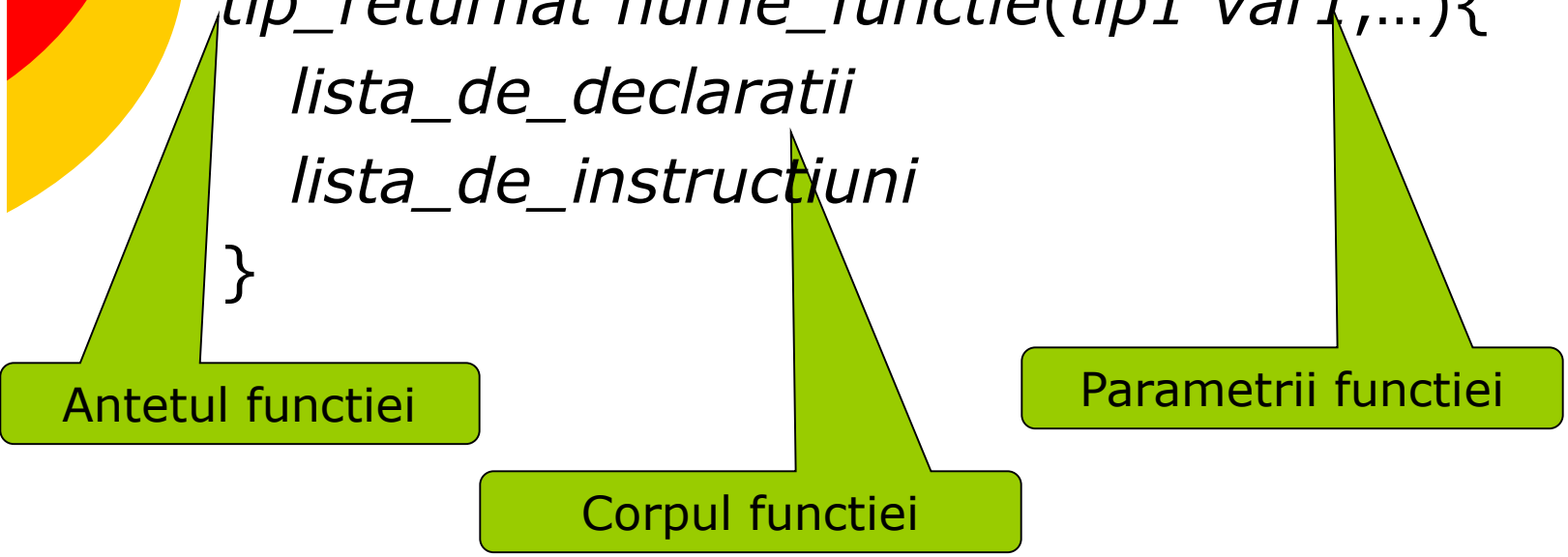
```
function suma(n)
   s ← 0
   for i ← 1 to n do
      s ← s + i
   return s
end
```

```
int suma(int n)
{
   int s = 0;
   int i;
   for(i=1; i<=n; ++i)
      s += i;
   return s;
}
```

Funcţii

- Definiţia unei funcţii:

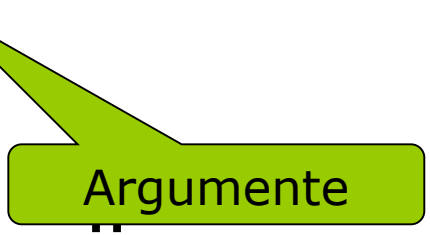
tip_returnat nume_functie(*tip1 var1*,…){
    lista_de_declaratii
    lista_de_instructiuni
}



Funcţii

- Apelul unei funcţii:

nume_functie(*expr1*,...)

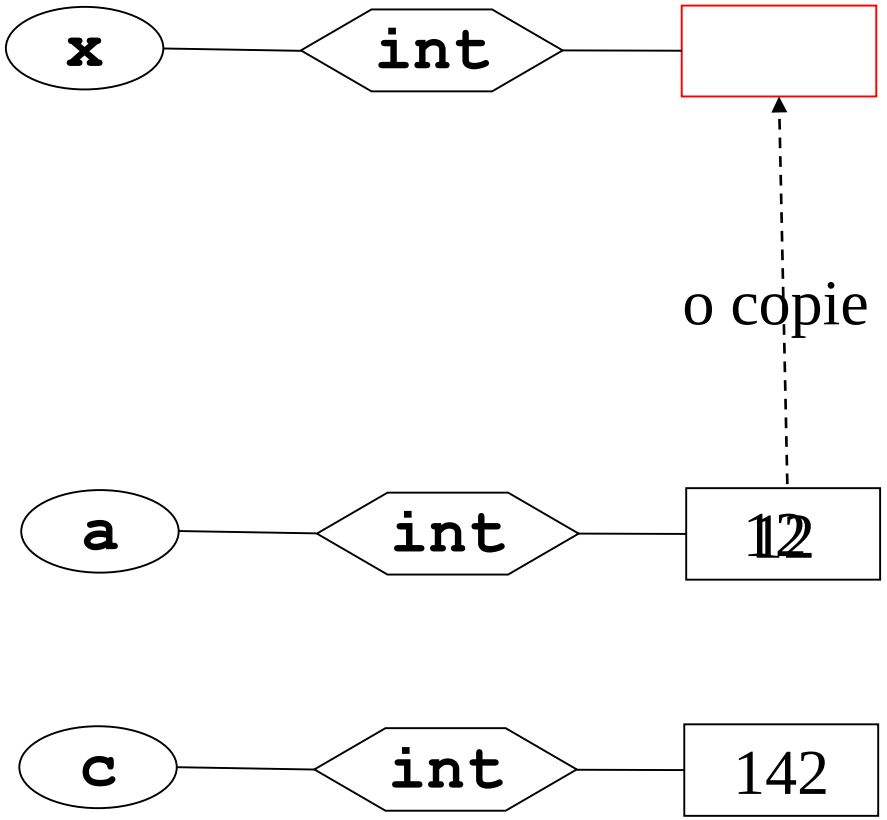


- Argumentele sunt expresii ce substitue parametrii la un apel: parametrii funcţiei sunt iniţializaţi cu valorile argumentelor.

Funcţii: legarea parametrilor (apel prin valoare)

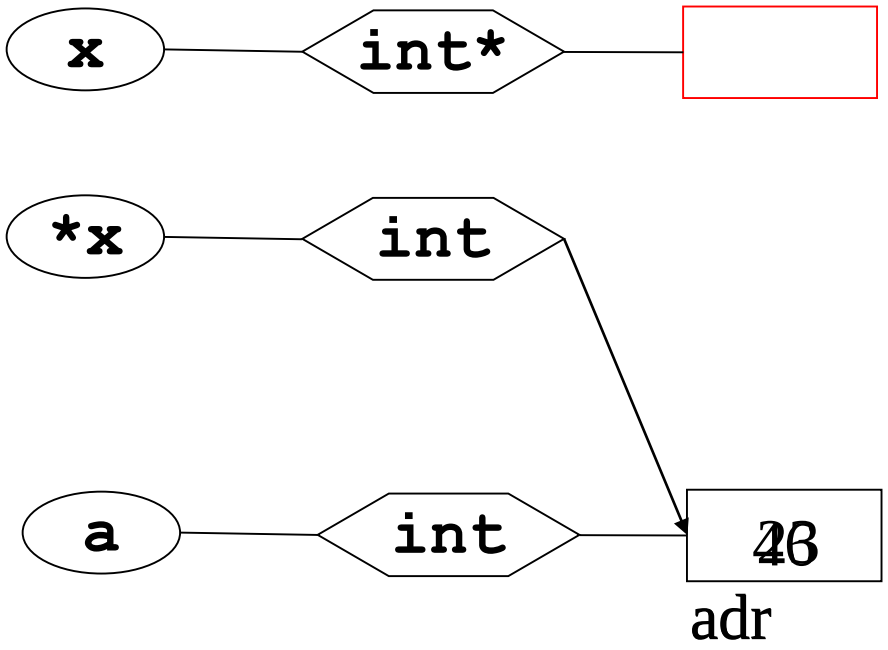
```
int sqr(int x)
{
   return x*x
}
```

```
int a = 12;
b = sqr(a);
c = b-2;
```



Funcţii: parametri pointeri

```
void dubleaza(int *x)
{
   *x += *x;
}
```




```
int a = 23;
dubleaza(&a);
/* a == 46 */
```

Funcţii

```
#include <stdio.h>
void swap(int x, int y){
   int temp = x; x = y; y = temp;
   printf("x = %d, y = %d\n", x, y);
}

int main(void){
    int a = 2, b = 3;
    swap(a, b); // x = 3, y = 2
    printf("a = %d, b = %d\n",a, b);
    // a = 2, b = 3
    return 0;
}
```

Funcţii

```
#include <stdio.h>
void swap(int *x, int *y){
   int temp = *x; *x = *y; *y = temp;
   printf("*x = %d, *y = %d\n", *x, *y);
}

int main(void){
    int a = 2, b = 3;
    swap(&a, &b); // *x = 3, *y = 2
    printf("a = %d, b = %d\n",a, b);
    // a = 3, b = 2
    return 0;
}
```

Funcţii: parametri

- parametri de intrare

```
int sqr(int x) {
  return x*x
}
```

- parametri de iesire

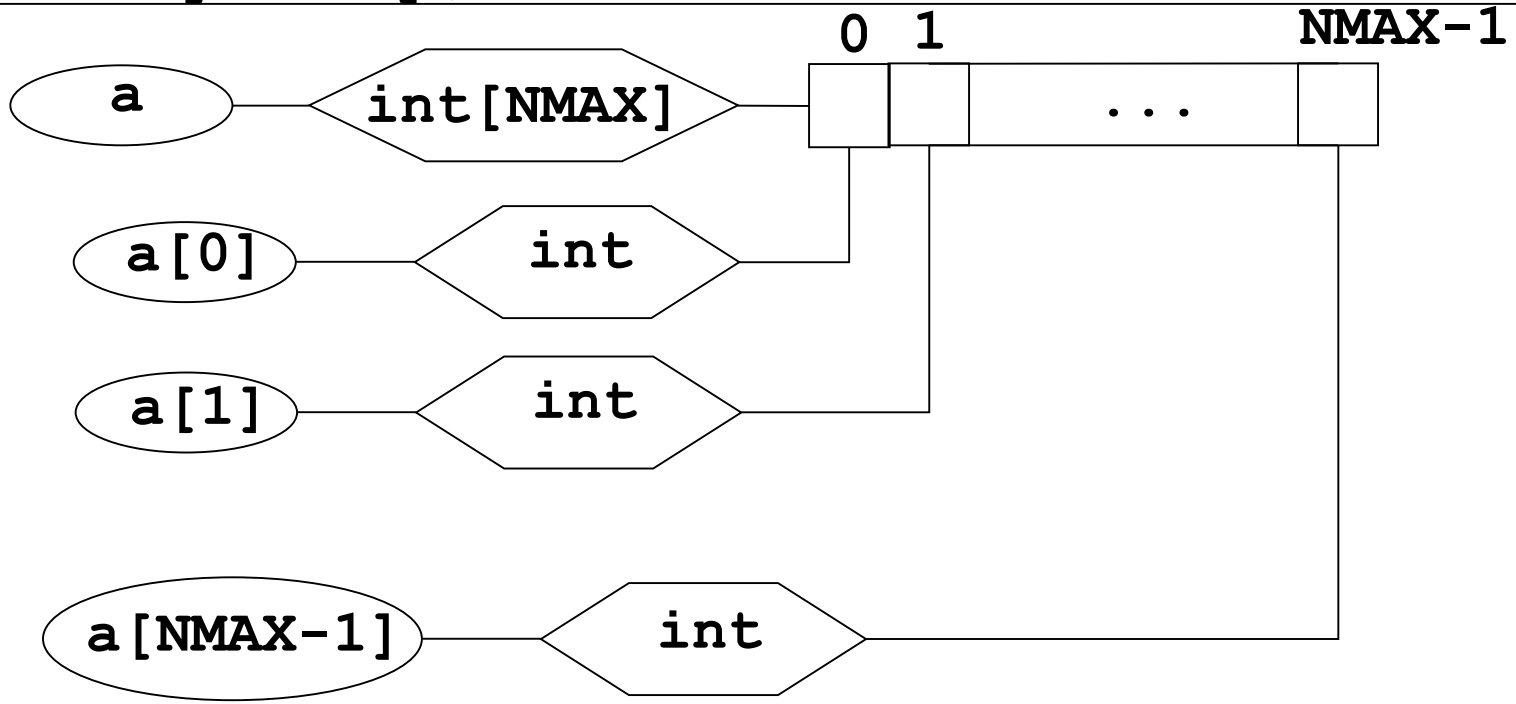

```
int imparte(int x, int y, int *q, int *r) {
  if (!y) return 1;
  *q = x / y;
  *r = x % y;
  return 0;
}
```

- parametri de intrare+iesire

```
void dubleaza(int *x) {
  *x += *x;
}
```

Declarare tablouri unidimensionale

```
#define NMAX 15
int a[NMAX];
```



Numele unui tablou

- Numele unui tablou:
  - nume de variabila
    - ex: **`sizeof(a)`**
  - pointer catre primul element din tablou:

`a` echivalent cu **`&a[0]`**

`*a` echivalent cu **`a[0]`**

`a+1` echivalent cu **`&a[1]`**

`*(a+1)` echivalent cu **`a[1]`**

`a+2` echivalent cu **`&a[2]`**

`*(a+2)` echivalent cu **`a[2]`**

`a+i` echivalent cu **`&a[i]`**

`*(a+i)` echivalent cu **`a[i]`**

Parcurgerea unui tablou

```
/* Varianta 1 */
for (i=0; i < n; ++i)
    suma += a[i];
```

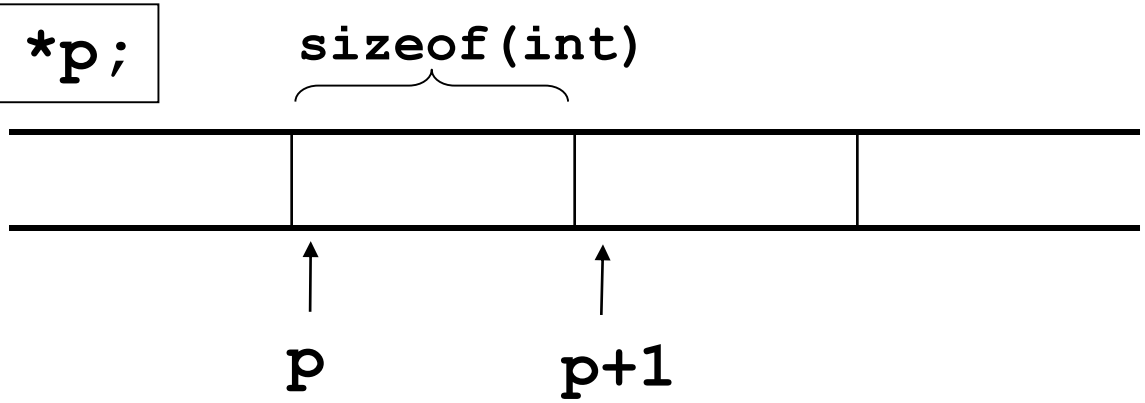
```
/* Varianta 2 */
for (i=0; i < n; ++i)
    suma += *(a+i);
```

```
/* Varianta 3 */
for (p=a; p < &a[n]; ++p)
    suma += *p;
```

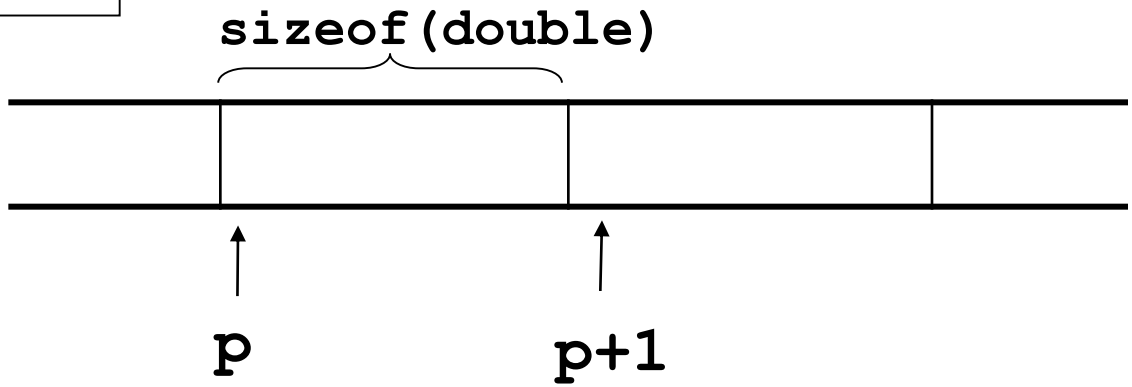
```
/* Varianta 4 */
p=a;
for (i=0; i < n; ++i)
    suma += p[i];
```

Aritmetica pointerilor

```
int *p;
```




```
double *p;
```



Aritmetica pointerilor

```
int    a[2], *p1, *q1;
```

```
p1 = a;
q1 = p1+1;
printf("%d",q1-p1);
printf("%d, %d", sizeof(int), (int)q1 - (int)p1);
```

q1 - p1 = 1

sizeof(int) = 4, (int)q1 - (int)p1 = 4

Aritmetica pointerilor

```
double c[2], *p3, *q3;
```

```
p2 = c;
q3 = p3+1;
printf("%d",q3-p3);
printf("%d, %d", sizeof(double),
                    (int)q3 - (int)p3);
```

q3 - p3 = 1

sizeof(double) = 8, (int)q3 - (int)p3 = 8

Tablourile ca parametri

```
void insert_sort(int a[], int n)
{
  //...
}
```

```
/* utilizare */
int w[100];
...
insert_sort(w, 10);
```

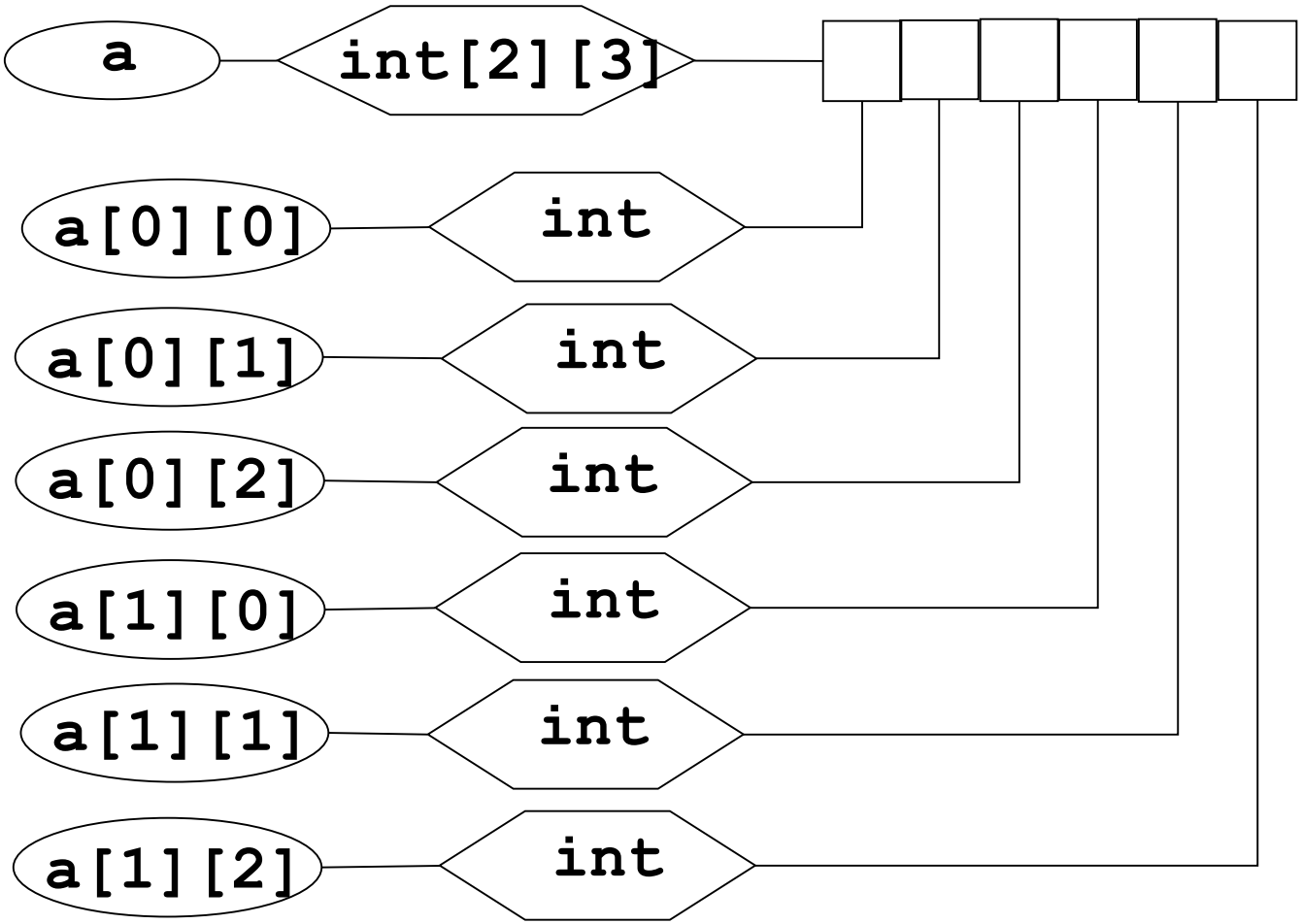
Tablourile ca parametri

```
double suma(double a[], int n);
double suma(double *a, int n);

suma(v, 100);
suma(v, 8);
suma(&v[4], k-6);
suma(v+4, k-6);
```

Tablouri bidimensionale

```
int a[2][3];
```



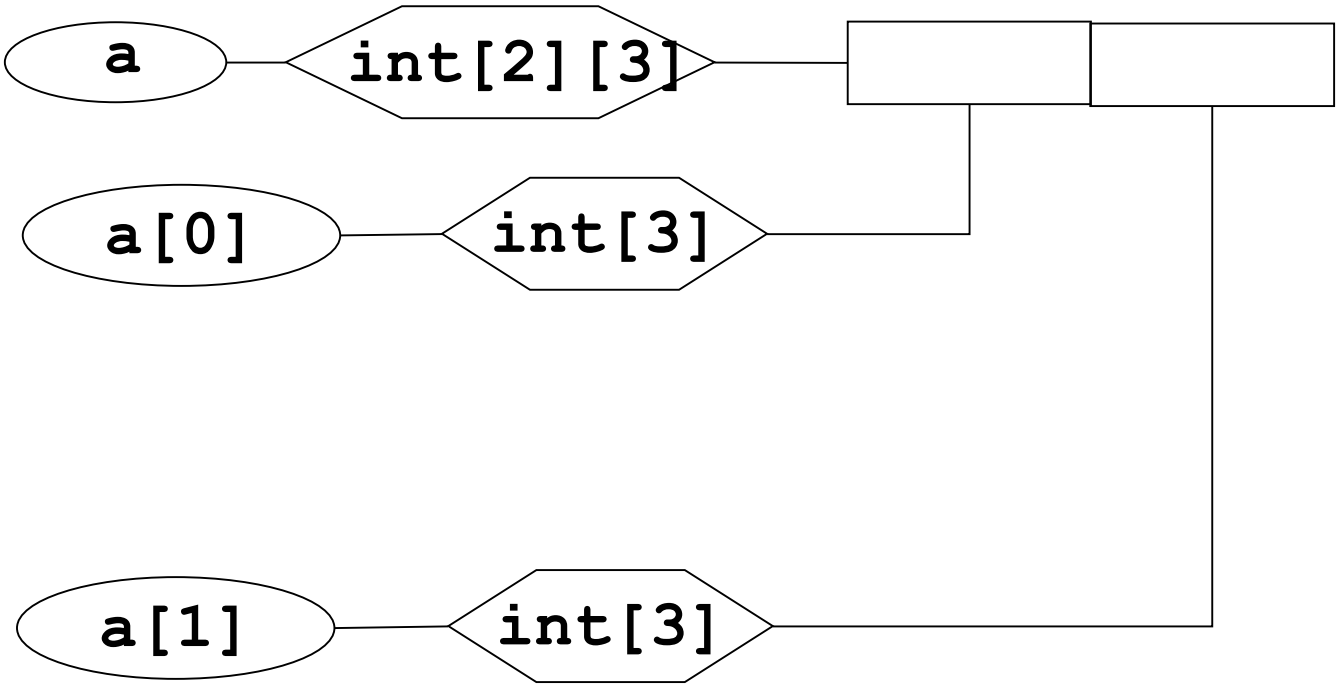
Parcurgerea unui tablou bidimensional

```
double a[MMAX][NMAX]; /* decl. tablou bidim. */
double suma; /* suma elemenetelor din tablou */
```

```
for (i = 0; i < m; i++)
    for(j = 0; j < n; j++)
        fscanf(finp, "%lf", &a[i][j]);
```

```
suma = 0;
for (i = 0; i < m; i++)
    for(j = 0; j < n; j++)
        suma += a[i][j];
```

Tablouri bi-dim. văzute ca tablouri uni-dim.



Funcţia de alocare a memoriei

	coloana 0	coloana 1	. . .
linia 0	`a[0][0]`	`a[0][1]`	. . .
linia 1	`a[1][0]`	`a[1][1]`	. . .
. . .	. . .	. . .	. . .

Expresii echivalente cu `a[i][j]`

`*(a[i] + j)`
`*((*(a + i)) + j)`
`(*(a + i))[j]`
`*(&a[0][0] + NMAX*i +j)`

Tablouri bidimensionale ca parametri

```
int minmax(int t[][NMAX],int i0,int j0,
                         int m,int n)
{
   //...
}
```

```
/* utilizare */
if (minmax(a,i,j,m,n))
{
   // ...
}
```

Iniţializarea tablourilor

```
int a[] = {-1, 0, 4, 7};
 /* echivalent cu */
int a[4] = {- 1, 0, 4, 7};
```

```
char s[] = "un sir"              /* echivalent cu */
char s[7] = {'u', 'n', ' ', 's', 'i', 'r', '\0'}
```

```
int b[2][3] = {1,2,3,4,5,6}  /* echivalent cu */
int b[2][3] = {{1,2,3},{4,5,6}} /*echivalent cu*/
int b[][3] = {{1,2,3},{4,5,6}}
```

Tablouri de char - Şiruri de caractere

- Declarare şiruri:

```
#define MAX_SIR 100
...
char s[MAX_SIR];
```

- Declarare cu iniţializare:

```
char s[] = "un sir"              /* echivalent cu */
char s[7] = {'u', 'n', ' ', 's', 'i', 'r', '\0'}
```

- Citirea unui sir:

```
printf("Sirul: ");
scanf("%s", s);
```

Tablouri de char - Şiruri de caractere

- Determinarea lungimii:

```
lg = 0;
while (s[lg] != '\0') lg++;
```

- Testarea proprietatii de palindrom

```
i=0; j=lg-1;
while (s[i]==s[j] && i<j)
 { i++; j--; }
if (i >= j)
     printf("\nSirul este palindrom.\n");
else printf("\nSirul nu este palindrom.\n");
```

Macrouri si functii pentru siruri

- In fisierul **ctype.h**

```
issspace(c), isdigit(c), islower(c), …
```

- In fisierul **string.h**

```
char *strcat(char *s1, const char *s2);
int strcmp(const char *s1,const char*s2);

char *strcpy(char *s1, const char *s2){
   register char *p = s1;
while(*p++ = *s2++) ;
return s1;
}

size_t strlen(const char *s);
char* strchr(const char* s, int c);
char* strdup(const char* s);
```

Tipuri enumerative

- o declaratie de forma

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

  declara un tip cu numele **`enum zi`** si 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 intreg cu semn sau cu un tip intreg fara semn (depinde de implementare)

- fiecare constanta a tipului are asociata o valoare intreaga

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

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

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 in combinatie cu **typedef**

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

- care este echivalenta cu

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

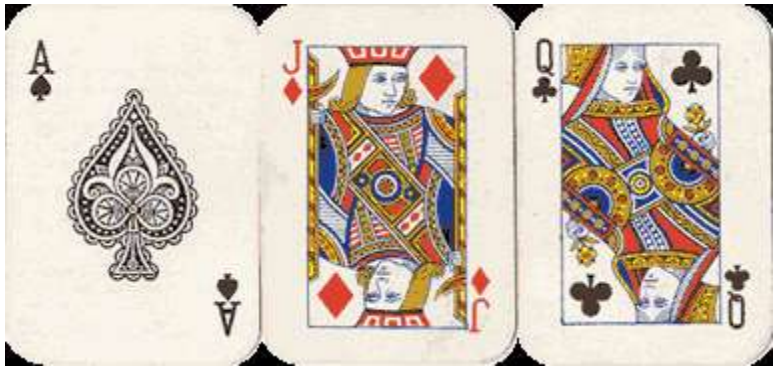
- sau

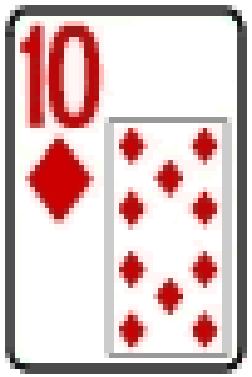
```
typedef enum {lu, ma, ...} zi;
zi azi;
```

- dar se poate si asa:

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

Structuri simple

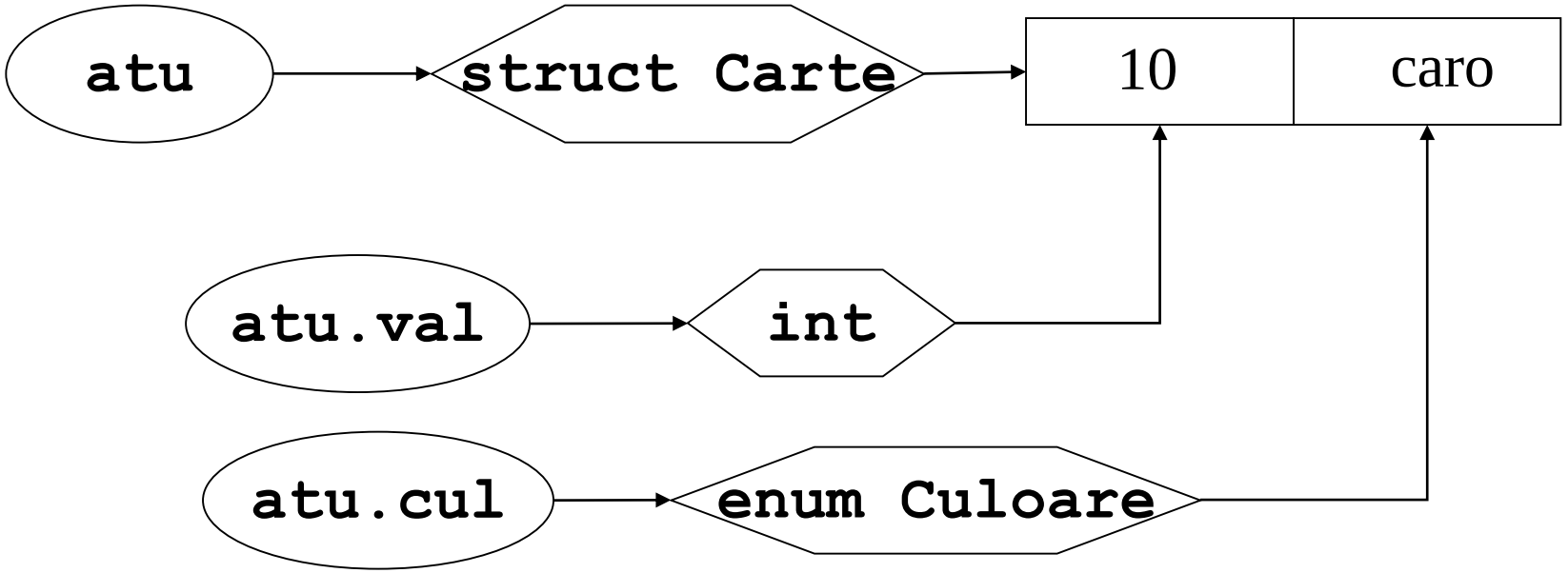





```
enum Culoare {trefla, cupa, caro, pica};
typedef enum Culoare Culoare;
struct Carte
{
   int val;
   Culoare cul;
};
```

Tipuri enumerative si Structuri simple

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



Structuri simple

```
printf("atuul este: %d", atu.val);
switch (atu.cul)
{
case trefla:
   printf(" %s\n", "trefla");
   break;
case caro:
   printf(" %s\n", "caro");
   break;
// ...
}
```

Asocierea de sinonime pentru structuri

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

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

- acum putem declara o variabila mult mai simplu

```
Carte atu;
```

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

Asocierea de sinonime pentru structuri

- cu **typedef** structura poate fi si anonima

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

- acum poate fi utilizat numai **Carte**

Structuri complexe

- un jucator are nume, o mana de carti si o suma de bani

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

- o masa are un numar si 4 jucatori

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

Structuri complexe

- jucatorul j primeste 8 de trefla ca a doua carte

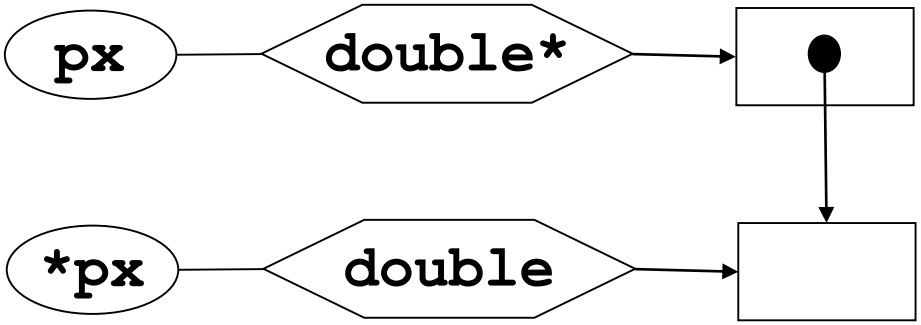
```
j.mana[2].val = 8;
j.mana[2].cul = trefla;
```

- jucatorul 3 de la masa m primeste 9 de caro ca prima carte

```
m.jucator[1].mana[0].val = 9;
m.jucator[1].mana[0].cul = caro;
```

Variabile dinamice - creare

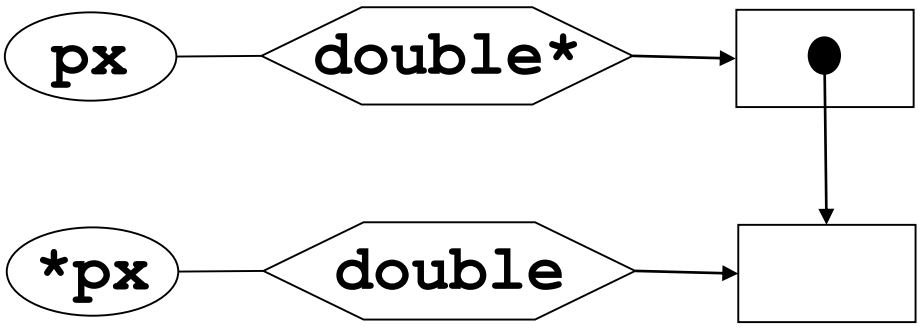
```
double *px;
px = (double*)malloc(sizeof(double));
/* sau */
px = (double*)calloc(1, sizeof(double));
```




```
void *malloc( size_t size );
void *calloc( size_t num, size_t size );
```

Variabile dinamice - distrugere

```
free(px);
```




```
void free( void *memblock );
```

Alocare tablouri 1

- Alocare

```
double **a;
a = (double **)calloc(m,
sizeof(double*));
for (i=0; i<n; i++)
    a[i] = (double *)calloc(n,
sizeof(double));
a[1][2] = 3.14;
printf("%lf", a[1][2]);
```

- Dealocare (eliberare, distrugere)

```
for (i=0; i<n; i++)
    free(a[i]);
free(a);
```

Alocare tablouri 2

- Alocare

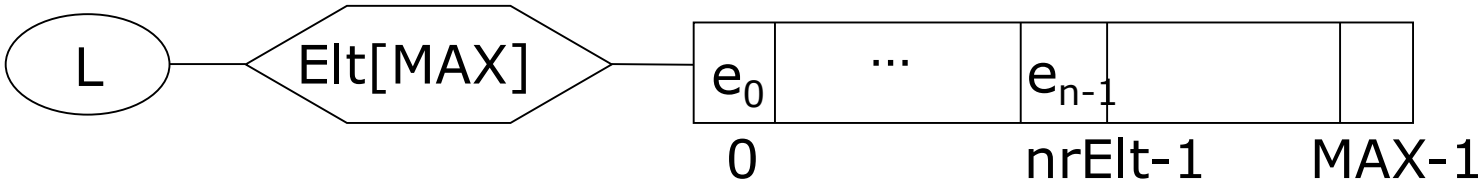
```
double *p;
p=(double *)calloc(m*n, sizeof(double));
a = (double **)calloc(m,
sizeof(double*));
for (i=0; i<m; i++)
   a[i] = p+i*n;
a[1][2] = 3.1415;
```

- Dealocare (eliberare, distrugere)

```
p = (double *)a[0];
free(p);
free(a);
```

Lista liniara: implementare cu tablouri

- L = (e_0,…, e_{n-1})



Lista liniara: implementare cu tablouri

```
#include "elt.h"

#define MAX_LLIN 1000
#define SUCCES 0
#define ERR_LLIN_MEM_INSUF 1
#define ERR_LLIN_INDEX_GRESIT 2

typedef struct Llin {
  Elt tab[];
  int nrElt;
}Llin;
```

Lista liniara: implementare cu tablouri

```
int listaVida(Llin *l){
  /* aloca memorie pentru tablou */
  l->tab = (Elt *)calloc(MAX_LLIN,sizeof(Elt));
  if (l->tab == NULL)
    return ERR_LLIN_MEM_INSUF;

  /* initializeaza numarul de elemente */
  l->nrElt = 0;

  /* operatie terminata cu succes */
  return SUCCES;
}
```

Lista liniara: implementare cu tablouri

```
int insereaza( Llin *l, Elt elt, int k )
{
  int j;

  /* testeaza validitatea indicelui */
  if ((k < 0) || (k > l->nrElt))
    return ERR_LLIN_INDEX_GRESIT;

  /* testeaza daca mai exista loc in tablou */
  if (l->nrElt == MAX_LLIN-1)
    return ERR_LLIN_MEM_INSUF;

  /* deplaseaza elementele la dreapta */
  for (j = l->nrElt-1; j >= k; j--)
    l->tab[j+1] = l->tab[j];
```

Lista liniara: implementare cu tablouri

```
    /* pune pe pozitia k noul element */
    l->tab[k] = elt;

    /* actualizeaza numarul de elemente */
    l->nrElt++;

    /* operatie terminata cu succes */
    return SUCCES;
}
```

Lista liniara: implementare cu tablouri

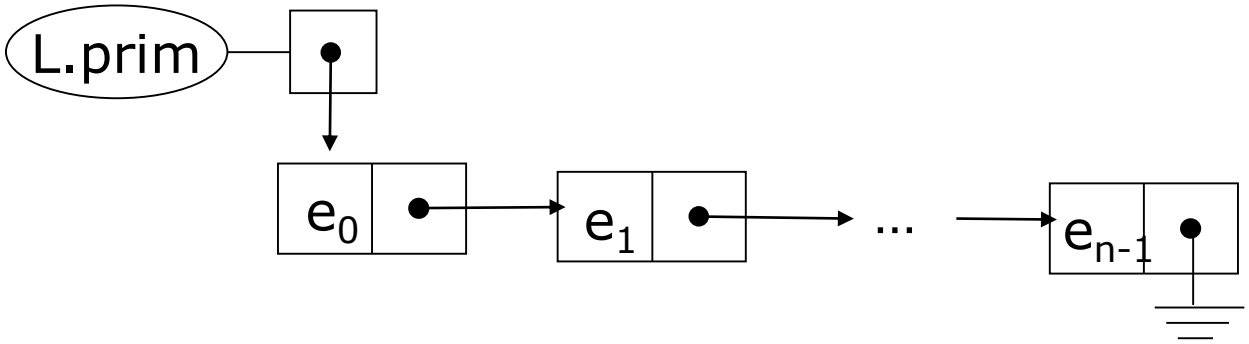
```
void parcurge(Llin *l, void
  viziteaza(Elt))
{
   int i;
   for (i = 0; i < l->nrElt; i++)
     viziteaza(l->tab[i]);
}
```

Lista liniara: implementare cu tablouri

```
Llin lista;
listaVida(&lista);
for (i=1; i<8; i++)
{
    e = i;
    if (coderr = insereaza(&lista, e, i-1))
        printf("\n ERR LLIN: %d\n", coderr);
}
parcurge(&lista, afiseazaInt);
```

Liste liniare: implementarea cu liste inlantuite

- L = (e_0,…, e_{n-1})



Liste liniare: implementarea cu liste inlantuite

```
typedef struct NodLlin
{
  Elt elt;
  struct NodLLin *succ;
} NodLLin;

typedef struct Llin
{
  NodLlin *prim;
  int nrElt;
} Llin;
```

Liste liniare: implementarea cu liste inlantuite

```
int insereaza( Llin *l, Elt elt, int k )
{
  int j;
  NodLlin *p, *q;

  /* testeaza exceptiile */
  if ((k < 0) || (k > l->nrElt))
     return ERR_LLIN_INDEX_GRESIT;

  /* aloca spatiu pentru noul nod */
  q = (NodLlin *)calloc(1, sizeof(NodLlin));
  if (q == NULL)
     return ERR_LLIN_MEM_INSUF;

  /* memoreaza noua informatie */
  q->elt = elt;
```

Liste liniare: implementarea cu liste inlantuite

```
/* stabileste noile legaturi */
  if ((k == 0) || (l->prim == NULL)) {  // primul sau
   lista vida
      q->succ = l->prim;
      l->prim = q;
  }
  else {      // nu-i primul si lista nevida
    for (p = l->prim, j = 0; j < k-1; j++)
       p = p->succ;
    q->succ = p->succ;
   p->succ = q;
  }

  /* actualizeaza numarul de elemente */
  l->nrElt++;

  /* operatie terminata cu succes */
  return SUCCES;
}
```

Liste liniare: implementarea cu liste inlantuite

```
int i, e, coderr;
Llin lista;
listaVida(&lista);
for (i=1; i<8; i++)
{
   e = i;
   if (coderr = insereaza(&lista, e, i-1))
        printf("\n ERR LLIN: %d\n", coderr);
}
parcurge(&lista, afiseazaInt);
```

Domeniul de vizibilitate – “scope”

- Un nume (variabilă, funcţie) poate fi utilizat numai după ce a fost declarat. Declaraţia descrie proprietăţile numelui
- Domeniul de vizibilitate (*scope*) al unui nume este mulţimea instrucţiunilor (liniilor de cod) în care poate fi utilizat acel nume (numele este vizibil)
- Regula de bază: identificatorii sunt accesibili doar în blocul în care au fost declaraţi; ei sunt necunoscuţi în afara acestor blocuri.
- variabile globale – variabile ce sunt declarate în afara oricărui bloc
- variabile locale sunt cele declarate:
  - în funcţii
  - în blocuri
  - ca parametri

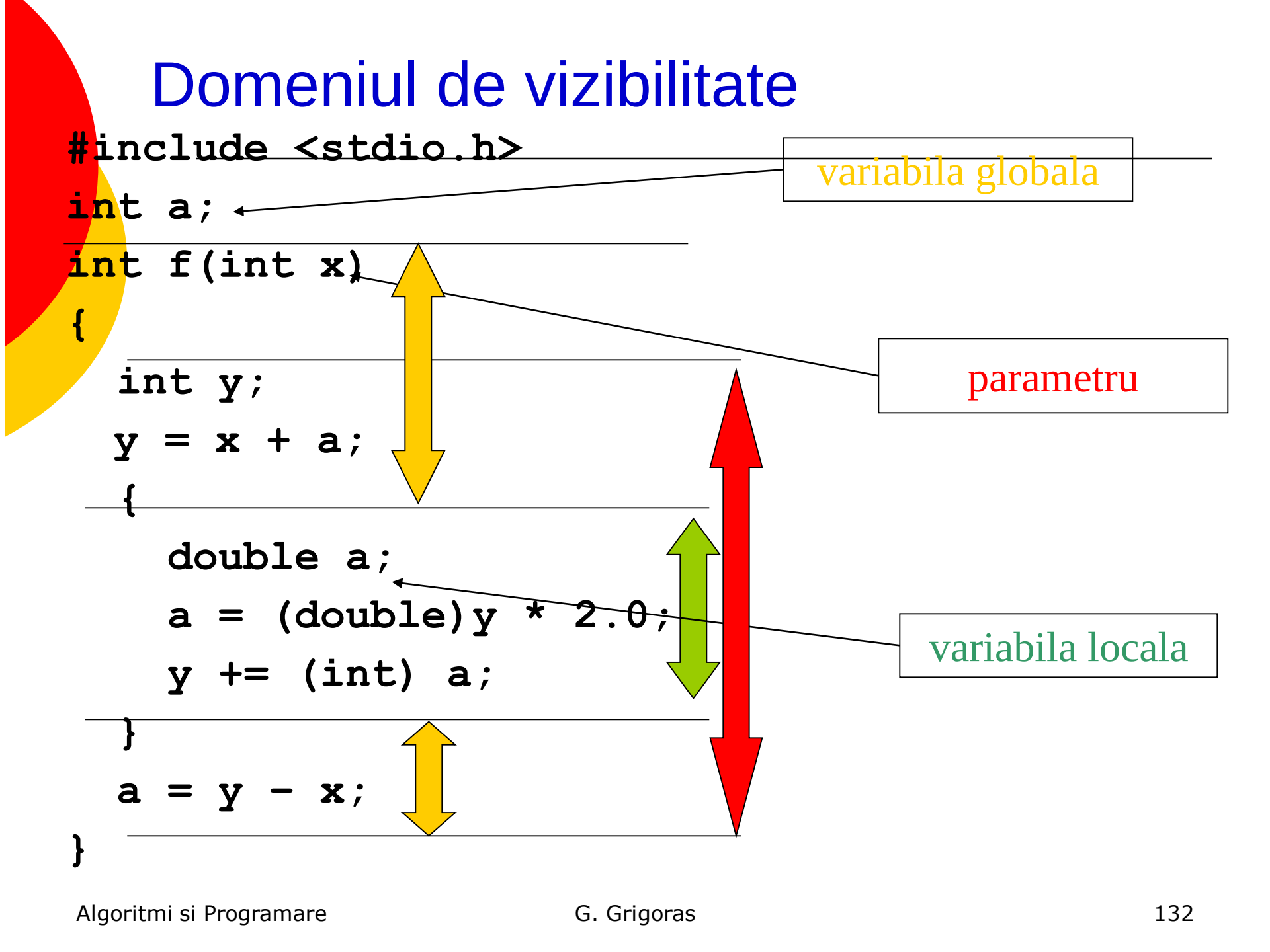
Domeniul de vizibilitate – “scope”

- Blocuri paralele: {…}…{…}. În acest caz cel de-al doilea bloc “nu ştie” nimic de variabilele declarate în primul bloc.
  - Funcţiile sunt declarate “în paralel”

- Blocuri cuibărite: {…{…}…}. Un nume declarat în blocul exterior este vizibil în cel interior dacă nu este redefinit aici; în acest din urmă caz, numele din blocul exterior este “ascuns” sau “mascat”. Spunem că fiecare bloc are “propria nomenclatură” pentru numele variabilelor

Domeniul de vizibilitate

```
#include <stdio.h>
int a;
int f(int x)
{
  int y;
  y = x + a;
  {
    double a;
    a = (double)y * 2.0;
    y += (int) a;
  }
  a = y - x;
}
```



Exemplu

```
{
   int a = 1, b = 2, c = 3;
   printf("%2d%2d%2d\n", a, b, c);          /* 1 2 3 */
   {
        int b = 4; float c = 5.0f;
        printf("%2d%2d%4.1f\n", a, b, c); /* 1 4 5.0 */
        a = b;
        {
           int c; c = b;
           printf("%2d%2d%2d\n", a, b, c);  /* 4 4 4 */
        }
        printf("%2d%2d%4.1f\n", a, b, c); /* 4 4 5.0 */
   }
   printf("%2d%2d%2d\n", a, b, c);          /* 4 2 3 */
}
```

Clase de alocare a memoriei

- Zona de memorie utilizată de un program C cuprinde 4 subzone:
  - Zona text: codul programului
  - Zona de date: variabilele globale
  - Zona stivă: date temporare (variabilele locale)
  - Zona heap: memoria dinamică
- Clasele de alocare a variabilelor:
  - Statică: în zona de date temporare
  - Auto: în stivă
  - Dinamică: în heap , alocate dinamic
  - Register: într-un registru de memorie

Alocarea implicită

- Durata de viaţă vs. domeniu de vizibilitate

	Variabile globale	**Variabile locale**
Alocare	statică la compilare	auto la execuţie bloc
Durata de viaţă	cea a întregului program	cea a blocului în care e declarată
Iniţializare	cu zero	nu

Clase de alocare

- Se poate utiliza cuvântul cheie **`auto`** în declararea variabileor locale:

```
auto int a, b, c;
auto double f;
```

- Clasa de alocare **`extern`**: o variabilă (globală) sau o funcţie declarată extern este vizibilă şi în alt fişier decât cel în care a fost declarată

- Funcţiile au clasa de alocare **`extern`**; cuvântul cheie **`extern`** poate fi utilizat la declararea/definirea funcţiilor:

```
extern double sinus(double);
```

Clase de alocare - exemplu

```
/* fisierul main.c */
int a = 1, b = 2, c = 3; /* variabile globale */
int f(void);  /* prototip */
int main(void){
  printf("a = %d, b = %d, c = %d, f() = %d\n");
}

/* fisierul f.c */
int f(void){
  extern int a; /* cauta a in afara fisierului
  */
  int b, c; /* b, c locale */
  a = b = c = 22;
  return (a + b + c);
}
```

Clasa de alocare static - local

- O variabilă locală declarată **`static`** are durata de viaţă egală cu cea a programului: la intrarea în bloc valoarea sa este cea care a avut-o la ieşire:

```
int f(void){
 static int contor = 0;
 return contor++;
}
f(); f(); f();
```

- Domeniu de vizibilitate vs. Durata de viaţă

Exemplu

```
#include <stdio.h>
int f(void);
int main(void) {
  int i;
  for (i=0; i<10; i++){
     if(!(i%3))
          printf("\nFunctia f() este apelata a %d-a oara.", f());
  }
  return 0;
}
int f(void) {
  static int nr_apeluri=0;
  nr_apeluri++;
  return nr_apeluri;
}
/*
Functia f() este apelata a 1-a oara.
Functia f() este apelata a 2-a oara.
Functia f() este apelata a 3-a oara.
Functia f() este apelata a 4-a oara.
*/
```

Clasa de alocare static - extern

- O variabilă globală declarată **`static`** are domeniul de vizibilitate redus la fişierul sursă în care este declarată, doar după declaraţia sa:

```
int f(void){
 /*variabila v nu este vizibila*/
}
static int v;
void g(void){
 /* v este vizibila aici */
}
```

- O funcţie definită/declarată **`static`** este vizibilă doar în fişierul în care apare definiţia sa

Clasa de alocare static - extern

```
static int g(void); /* prototip */
void f(int a){
  …
  /* g este vizibila aici */
}
static int g(void){
…
}
/* g nu este vizibila in alt
fisier */
```

Clasa de alocare **`register`**

- O variabilă declarată **`register`** solicită sistemului alocarea ei într-un registru maşină, dacă este posibil
- Se utilizează pentru variabile "foarte solicitate", pentru mărirea vitezei de execuţie:

```
{
    register int i;
    for(i = 0; i < N; ++i){
        /*… */
    }
} /* se elibereaza registrul */
```

Domeniul de vizibilitate - rezumat

- Într-un fişier (resp. bloc) un identificator este vizibil după declararea sa până la sfârşitul fişierului (resp. blocului) cu excepţia blocurilor(resp. subblocurilor) în care este redeclarat
- Definiţia unui identificator maschează pe cea a aceluiaşi identificator declarat într-un suprabloc sau în fişier (global)
- Apariţia unui identificator face referinţă la declararea sa în cel mai mic bloc (sau fişier) care conţine această apariţie
- Funcţiile şi variabilele globale ale unei unităţi de program (fişier) sunt implicit *publice*: sunt accesibile din alte unităţi de program.
- **`extern`** indică o declaraţie fără definire: permite referirea unei variabile globale definită în afara unităţii de program.
- **`static`** face ca o variabilă globală sau o funcţie să fie *privată(proprie)* unităţii unde a fost definită: ea devine inaccesibilă altei unităţi, chiar prin folosirea lui **`extern`**.

Preprocesorul
Directivele **#include** si **#define**

#include <*nume_fisier*>
#include "*nume_fisier*"

#define *nume_macrodef*
#define *nume_macrodef macrodef*
#define *nume_macrodef*(*lista_arg*)
#define *nume_macrodef*(*lista_arg*) *macrodef*
nume_macrodef ::= *identificator*
arg ::= *identificator*
lista_arg ::= *arg* |*lista_arg*, *arg*
macrodef ::= *sir_unitati_lexicale_si_arg*

Directivele `#include` si `#define`

```
#define pi 3.14159
#define   egal ==
#define   citeste   scanf
#define patrat(x)   ((x)*(x))
#define cub(x) (patrat(x)*(x))
#define min(x,y) (((x)<(y))?(x):(y))
#define printTablou(a, n, sirControl)    \
          for (i = 0; i < n; i++)        \
          printf(sirControl, a[i]);      \
          putchar('\n')
#define new(X) (X*)malloc(sizeof(X))
```

#define Exemplu

```
#include <stdio.h>
#define swap(t,a,b) {t temp=a;a=b;b=temp;}
int main(){
    int i=10, j=20;
    float x=1.23,y=3.21;
    printf("\ni=%d, j=%d, x=%f, y=%f",i,j,x,y);
    swap(int,i,j);
    swap(float,x,y);
    printf("\ni=%d, j=%d, x=%f, y=%f",i,j,x,y);
    return 0;
}
/*
i=10, j=20, x=1.230000, y=3.210000
i=20, j=10, x=3.210000, y=1.230000
*/
```

Macroul assert() din **assert.h**

- Se utilizează în programe pentru a ne asigura că valoarea unei expresii este cea pe care o anticipăm

- Dacă o aserţiune eşueaza – " condiţia nu este îndeplinită " se va afişa un mesaj şi programul încetează a se executa

```
#include <assert.h>
void f(char *p, int n){
    assert( p != NULL):
    assert(n > 0 && n < 10);
    /*…*/
}

assert(b*b-4.*a*c >= 0);
```

Macrouri în `stdio.h` şi `ctype.h`

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
#define       NULL   ((void*)0)

toupper(c) /* întoarce valoarea "upercase" corespunzătoare lui c */
tolower(c) /* întoarce valoarea "lowercase" corespunzătoare lui c */
toascii(c) /* întoarce valoarea ASCII corespunzătoare lui c */

isalpha(c)      /* întoarce nonzero dacă c este literă */
isdigit(c)      /* întoarce nonzero dacă c este cifră */
isalnum(c)      /* întoarce nonzero dacă c este litera sau cifră*/
islower(c)      /* întoarce nonzero dacă c este litera mică */
isupper(c)      /* întoarce nonzero dacă c este litera mare */
isgraph(c)      /* întoarce nonzero dacă c este printabil, nu spaţiu */
isprint(c)      /* întoarce nonzero dacă c este caracter printabil*/
isxdigit(c) isspace(c) ispunct(c) iscntrl(c) isascii(c)
```

Macrouri predefinite

```
__DATE__        /* şir care conţine data curentă */
__FILE__        /* şir care conţine numele fişierului */
__LINE__        /*  conţine numărul liniei curente */
__STDC__        /* are valoarea nonzero dacă
                   implementarea  este ANSI standard C */
__TIME__        /* şir care conţine timpul curent */
```

Macrouri predefinite

```
#include <stdio.h>
int main(void){
    printf("Macrourile predefinite: \n");
    printf("__DATE__ = %s\n", __DATE__);
    printf("__FILE__ = %s\n", __FILE__);
    printf("__LINE__ = %d\n", __LINE__);
    printf("__STDC__ = %d\n", __STDC__);
    printf("__TIME__ = %s\n", __TIME__);
    return 0;
}
/*
Macrourile predefinite:
__DATE__ = Dec  4 2006
__FILE__ = ../Surse/Macro.c
__LINE__ = 6
__STDC__ = 1
__TIME__ = 17:43:24
*/
```

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 ce se utilizează pentru a extrage biţii convenabili: 1, 255=2^8-1

Operatorii bit cu bit - precedenţa

- ~ are aceeasi precedenţă cu !, asociativitate dreapta
- << şi >> după +, - şi înainte de <, <=, >, >=
- &, ^, | în această ordine după == şi !=, înainte de &&

Precedenţa operatorilor

OPERATORI	ASOCIERE
`()` `[]` `.` `->` `++` `--` (postfix)	stânga
`++` `--` (prefix) `!` `~` `&` (adresa) `*` (dereferentiere) `+` `-` (unari) `sizeof(`***tip***`)`	dreapta
`*` `/` `%`	stânga
`+` `-`	stânga
`<<` `>>`	stânga
`<` `<=` `>` `>=`	stânga
`==` `!=`	stânga
`&`	stânga
`^`	stânga
`|`	stânga
`&&`	stânga
`||`	stânga
`?:`	dreapta
`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	dreapta
`,` (operatorul virgula)	stânga

Operaţii bit cu bit – Exemplul 1

```
#include <stdio.h>
#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);
  printf("%s", s);
  for (i=1; i <= n; i++) {
    putchar(((x & mask) == 0)? '0' : '1');
    x <<= 1;
    if (i%CHAR_BIT == 0 && i<n)
      putchar(' ');
  }
  printf("\n");
}
```

Operaţii bit cu bit – Exemplul 1

```
void main(int x){
   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= ");
}
```

Calificatorul `const`

```
const float pi = 3.14
```

- **pi** este o constanta **float** cu memorie auto
- valoarea variabilei **pi** nu mai poate fi modificata dupa initializare

```
const int a = 5;
int *p = &a;
```

- o atribuire de forma ***p = ...** poate modifica valoarea variabilei **a**
- compilatorul ar trebui sa sesizeze si sa se “plângă”

```
const int a = 5;
const int *p = &a;
```

- **p** este un pointer la o constanta **int** si valoarea sa initiala este adresa lui **a**
- **p** NU este o constanta; o atribuire **p = &b** este OK
- o atribuire de forma ***p = ...** NU mai este posibila

Calificatorul `const`

```
const int a = 5;
const int * const p = &a;
```

- `p` este un pointer constant la o constanta int
- o atribuire `p = &b` NU este OK

```
int f(const int x) {
  return ++x;
}
```

- calificatorul `const` specifica faptul ca parametrul `x` nu poate fi modificat in blocul functiei `f`
- compilatorul ar trebui sa se planga la intalnirea expresiei `++x`

Fişiere. Structura FILE

- 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 sa ştie – programatorul ii spune – 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

Fişiere. Structura FILE

- Starea curentă a unui fişier este descrisă într-o structură numită FILE şi care este definită în **`stdio.h`**
- Programatorul poate folosi fişiere fără să cunoască în detaliu structura FILE

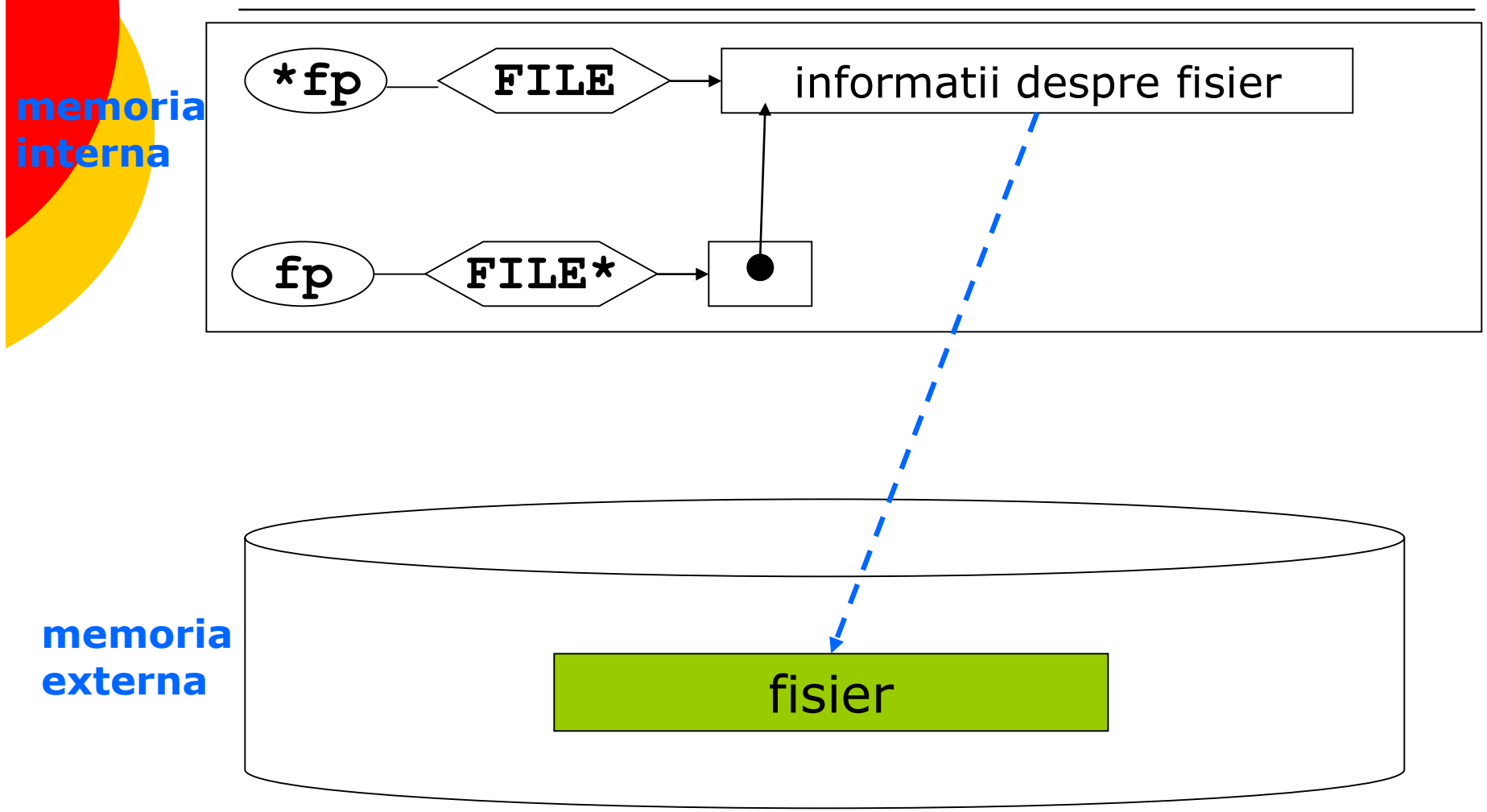
```
typedef struct {
        int   _cnt;
        char *_ptr;
        char *_base;
        int   _bufsiz;
        int   _flag;
        int   _file;
        char *_name_to_remove;
        int   _fillsize;
} FILE;
```

```
struct _iobuf {
        char *_ptr;
        int   _cnt;
        char *_base;
        int   _flag;
        int   _file;
        int   _charbuf;
        int   _bufsiz;
        char *_tmpfname;
        };
typedef struct _iobuf FILE;
```

Fisiere. Structura FILE

- Un obiect de tip FILE înregistrează informaţiile pentru a controla un stream:
  - Indicatorul pentru poziţia in fişier
  - Un pointer la zona buffer asociată
  - Un indicator de eroare care înregistrează dacă se produc erori de citire/scriere (codificat în _flag)
  - Un indicator *end-of-file* ce înregistrează dacă s-a atins sfârşitul de fişier (codificat în _flag)
- Când se deschide un fişier sistemul de operare îl asociază cu un *stream* şi păstrează informaţiile despre acest *stream* într-un obiect de tip FILE
- Un pointer la FILE "face legătura" cu fişierul sau cu stream-ul asociat fişierului

Fişiere


memoria interna
*fp
FILE
informatii despre fisier
fp
FILE*
memoria externa
fisier

Fişiere. Structura FILE

```
FILE *inf, *outf, *f;
```

- În **stdio.h** sunt definiţi pointerii:
  - **stdin:** fişierul standard de intrare
  - **stdout:** fişierul standard de ieşire
  - **stderr:** fişierul standard pentru erori

```
extern FILE __dj_stdin, __dj_stdout, __dj_stderr;
#define stdin       (&__dj_stdin)
#define stdout      (&__dj_stdout)
#define stderr      (&__dj_stderr)
```

- Programatorul nu trebuie să deschidă explicit fişierele standard

Funcţia `fopen()`

```
FILE*  fopen(const char *filename, const char *mode);
```

- Realizează cele necesare gestionării unui fişier:
  - Dacă se execută cu succes, crează un stream şi întoarce pointer la FILE asociat acestui stream
  - Dacă **`filename`** nu pate fi accesat întoarce NULL

```
mode ::= "r"  | "w"  | "a"  |"r+"  |"w+"  |"a+"
       |"rb"  |"wb"  |"ab"  |"r+b" |"w+b" |"a+b"
       |"rb+" |"wb+" |"ab+"
```

- Indicatorul de poziţie este pus la începutul fişierului (în modul "r" sau "w") sau la sfârşit (în modul "a")
- Modul "a+" este pentru actualizare:
  - Scrierea nu poate fi urmată de citire dacă nu s-a ajuns la EOF sau nu s-a intervenit cu o funcţie de poziţionare
  - Citirea nu poate fi urmată de scriere dacă nu se intervine cu apel la flush() sau la o funcţie de poziţionare

Funcţiile `fclose()`, `fflush()`, `freopen()`

```
int fclose(FILE *fp);
```

- Realizează cele necesare pentru a închide un fişier: goleşte buffer-ul şi întrerupe orice legătură între fişier şi pointerul fp
  - Dacă se execută cu succes returnează zero
  - Dacă apare o eroare sau fişierul este deja închis se returneaza EOF

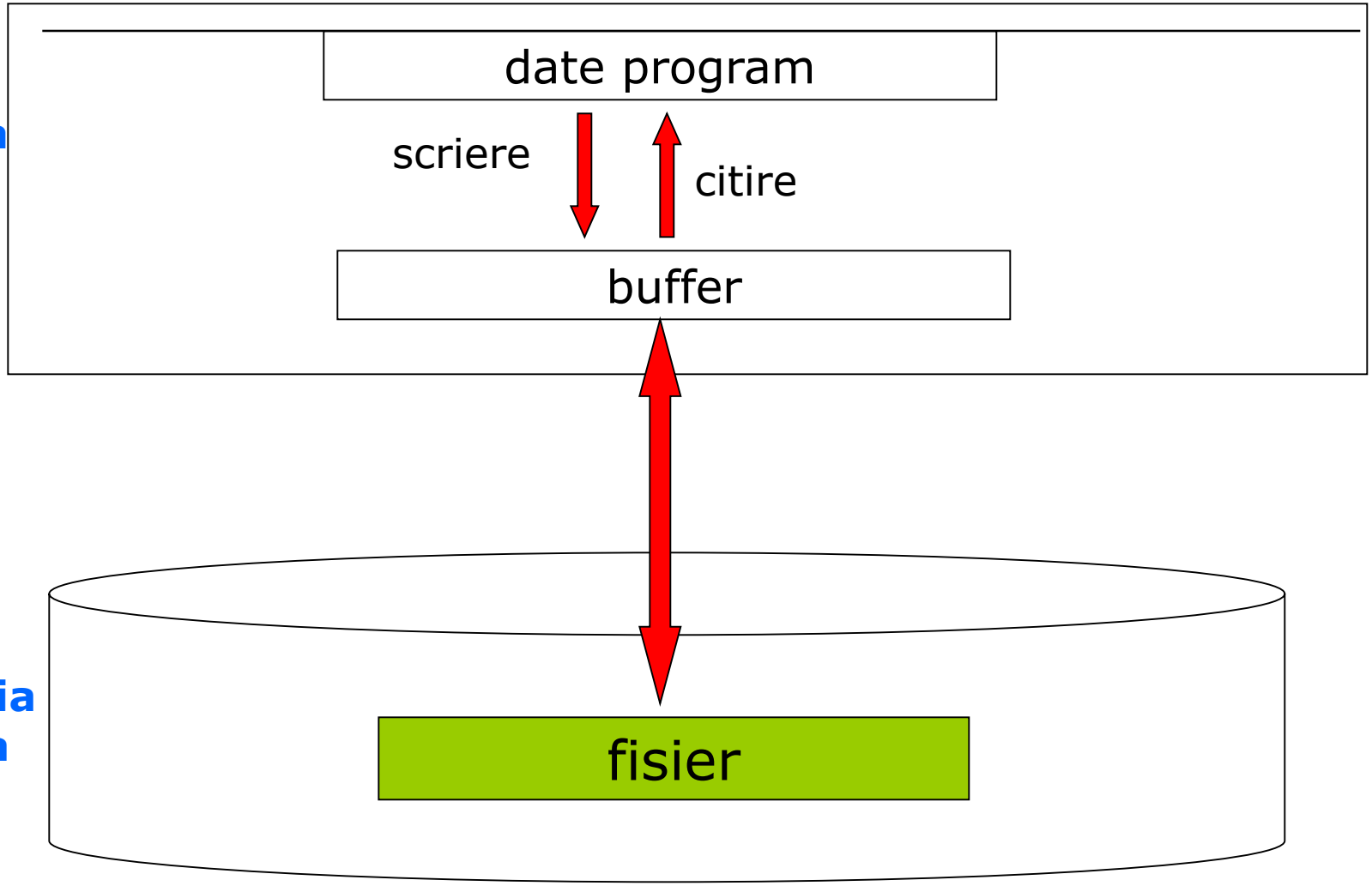
```
int fflush(FILE *fp)
```

- Golirea bufferului: datele din buffer sunt scrise în fişier (daca fp nu este NULL). Se întoarce 0 în caz de succes şi EOF altfel

```
FILE*  freopen(const char *filename,
                       const char *mode, FILE *fp);
```

- Este închis fişierul asociat pointerului fp şi se deschide filename iar `fp` se asociază acestuia

Fisiere – citire/scriere



Funcţiile fprintf(), printf(), sprintf()

```
int fprintf(FILE *pf, const char *format, ...);

int printf(const char *format, ...);

int sprintf(char *s, const char *format, ...);
```

- Apelul returnează numărul de conversii realizate cu succes
- În şirul **format** apar specificatorii de conversie introduşi prin caracterul %
- La apel, corespondenţa argument --- specificator de conversie
- Caracterele ce nu fac parte din specificatorii de conversie sunt scrise in stream-ul de ieşire

```
printf("a = %d, b = %f, c = %s.\n", a, b, c);
```

Funcţiile fprintf(), printf(), sprintf()

specificator_de_conversie ::= %{*modificator*}$_{opt}$
{*marime_camp*}$_{opt}$ {**.***precizie*}$_{opt}$ *caracter_de_conversie*

caracter_de_conversie ::=
c|d|i|u|o|x|X|e|E|f|g|G|s|p|n|%

modificator ::= **h|l|L|-|+|#|0**

marime_camp ::= *numar_intreg_fara_semn*

precizie ::= *numar_intreg_fara_semn*

Funcţiile fprintf(), printf(), sprintf()

- Câmpul „mărime” sau/şi „precizie” poate fi înlocuit prin ***** : valoarea va fi luată dintr-un argument:

```
printf("x= %*.*f\n", m, n, x);
```

Funcţiile fscanf(), scanf(), sscanf()

```
int fscanf(FILE *pf, const char *format, ...);

int scanf(const char *format, ...);

int sscanf(char *s, const char *format, ...);
```

- Apelul returnează numărul de conversii realizate cu succes, respectiv EOF dacă stream-ul de intrare este vid
- În şirul **`format`** apar specificatorii de conversie introduşi prin caracterul %
- La apel, corespondenţa argument --- specificator de conversie. Argumentele trebuie sa fie pointeri sau adrese
- Caracterele ce nu fac parte din specificatorii de conversie trebuie să apară în stream-ul de intrare

Functiile fscanf(), scanf(), sscanf()

```
int   i;
char  c;
char  sir[15];
scanf("%d , %*s   %% %c %7s %s",
                  &i, &c, sir,&sir[7]);
```

- Dacă stream-ul de intrare este:

45 , sir_ce_se_ignora % A string_citit**

- 45 se memoreaza in i
- , se potriveste cu , din format
- Este ignorat sirul **sir_ce_se_ignora**
- % se potriveste cu % din format
- A se memoreaza in c
- string_ se memoreaza in sir[0]..sir[6] iar in sir[6] se pune '\0'
- citit** se memoreaza in sir[7]..sir[13] iar in sir[14] se pune '\0'

Functii de intrare/iesire caracter

```
int  fgetc(FILE *stream);
int  getc(FILE *stream);
int  getchar(void);
char*fgets(char *s, int n, FILE *stream);
char*gets(char *s);
```

- getc() este implementata ca macro
- getchar() este echivalenta cu getc(stdin)
- gets(s) pune in s caracterele citite din `stdin` pana la newline sau EOF. In loc de newline pune la sfarsit '\0'; fgets() pastreaza newline

Functii de intrare/iesire caracter

```
int   fputc(int c, FILE *stream);
int   putc(int c, FILE *stream);
int   putchar(int c);
int   fputs(const char *s, FILE *stream);
int   puts(const char *s);
int   ungetc(int c, FILE *stream);
```

- fputc(c, pf) converteste c la unsigned char, il scrie in pf si intoarce (int)(unsigned char) c sau EOF la eroare
- putc() este macro echivalent cu fputc()
- fputs(s, pf) copie sirul s terminat cu '\0' in ps fara sa puna si '\0'. puts() adauga '\n'
- ungetc(c, pf) pune inapoi valoarea **`(unsigned char) c`** in stream-ul asociat lui pf (c nu este EOF)

Exemplu

```
/* Copiere fisier cu modificare litere mici */
/*..*/
   char   file_name[MAXSTRING];
   int    c;
   FILE   *ifp, *ofp;

   fprintf(stderr, "\nIntrodu numele unui fisier:  ");
   scanf("%s", file_name);
   ifp = fopen(file_name, "r");
   if(!ifp) {
      printf("Eroare la deschiderea fisierului\n");
      return 1;
   }
   ofp = fopen("test.out", "w");
   while ((c = getc(ifp))!=EOF) {
      if(islower(c)) c = toupper(c);
      putc(c, ofp);
   }
```

Functii de citire/scriere fara format

```
size_t   fread(void *ptr, size_t size,
               sizet nelem, FILE *stream);
```

- Se citesc cel mult **`nelem*size`** octeti (caractere) din fisierul asociat cu `stream` in tabloul pointat de `ptr`. Este returnat numarul elementelor transferate in tablou

```
size_t   fwrite(const void *ptr, size_t size,
                size_t nelem, FILE *stream);
```

- Se citesc cel mult **`nelem*size`** octeti (caractere) din tabloul ptr si se scriu in fisierul asociat cu `stream`. Este returnat numarul elementelor din tablou transferate cu succes

Functii de acces aleator

```
int    fseek(FILE *fp, long offset, int place);
```

- Pozitia indicatorului pentru pentru urmatoarea operatie este stabilita la "**offset**" octeti fata de "**place**".
- **Valoare lui place poate fi:**
  - **SEEK_SET sau 0**
  - **SEEK_CUR sau 1**
  - **SEEK_END sau 2**
- Exemple:
  - pozitionarea la sfirsitul fisierului
    ```
    fseek(fp, 0, SEEK_END)
    ```
  - pozitionarea la caracterul precedent
    ```
    fseek(fp, -1, SEEK_CUR)
    ```
  - pozitionarea la inceputul fisierului
    ```
    fseek(fp, 0, SEEK_SET)
    ```

Functii de acces aleator

```
long ftell(FILE *fp);
```

- Returneaza valoarea curenta a indicatorului de pozitie in fisierul fp; la fisierele binare este numarul de octeti de la inceputul fisierului, pentru cele text depinde de sistem

```
int  fsetpos(FILE *fp, const fpos_t *pos);
```

- Seteaza indicatorul de pozitie la valoarea pointata de pos si intoarce 0 daca s-a realizat cu succes

Functii de acces aleator

```
int  fgetpos(FILE *fp, fpos_t *pos);
```

- Indicatorul de pozitie al fisierului fp este memorat la pos si poate fi folosita ulterior
- Este returnat 0 in caz de succes

```
void rewind(FILE *fp);
```

`rewind(fp)` este echivalent cu

```
(void) fseek(fp, 0L, SEEK_SET);
```

```
int   remove(const char *filename);
int   rename(const char *old, const char *new);
```

Exemplu

```
/* Scrierea unui fisier de la sfarsit */
/*..*/
char   file_name[MAXSTRING];
int    c;
FILE   *ifp;

fprintf(stdout, "\nInput a file name:  ");
scanf("%s", file_name);
ifp = fopen(file_name, "rb");
fseek(ifp, 0, 2);   // pozitionare la sfarsit
fseek(ifp, -1, 1);  // pozitionare la ultimul
octet
while (ftell(ifp) > 0) {
          c = getc(ifp);
    putchar(c);
          fseek(ifp, -2, 1); //octetul anterior
}
```

Functii pentru controlul erorilor

```
int    feof(FILE *fp);
```

- Intoarce o valoare nenula daca indicatorul end-of-file este setat pentru fp

```
int    ferror(FILE *fp);
```

- Intoarce o valoare nenula daca indicatorul de eroare este setat pentru fp

```
void   clearerr(FILE *fp);
```

- Reseteaza indicatorii de eroare si end-of-file pentru fp

```
void   perror(const char *_s);
```

- Tipareste un mesaj de eroare la stderr: se scrie sirul s apoi mesajul de eroare. Apelul **`perror(errno)`** scrie doar mesajul de eroare

Parametri in linia de comanda

```
I:\AlgsiProg\Exemple>suma.exe f_in f_out
```

- functia `main()` cu argumente

```
int main(int nr_arg, char *arg[]) { ... }
```

- testarea numarului de argumente

```
if (nr_arg != 3)
{
    printf("Linie de comanda gresita.\n%s%s%s",
      "Trebuie sa introduceti", arg[0],
      "fisier_intrare fisier_iesire.\n");
    exit(1);
}
```

- utilizarea argumentelor

```
finp = fopen(arg[1],"r");
fout = fopen(arg[2],"w");
```