Declaraţii. Instrucţiuni

18 octombrie 2004

Structura programului: declaraţii şi definiţii

Un program C: compus din ≥ 1 *unităţi de compilare* (fişiere). Fiecare: un şir de *declaraţii* (de tipuri, variabile, funcţii) sau *definiţii de funcţii*.

translation-unit ::= *external-declaration* | *translation-unit external-declaration*

external-definition ::= *declaration* | *function-definition*

O *declaraţie* specifică interpretarea şi atributele unui *identificator*
– pentru o variabilă, numele şi tipul
– pentru o funcţie, numele, tipul, şi tipul parametrilor

O *definiţie* e o declaraţie care specifică *complet* identificatorul respectiv
– pentru o variabilă, în plus, are ca efect alocarea memoriei
– pentru o funcţie, include corpul funcţiei

Un identificator nu poate fi folosit înainte de a fi *declarat*.
– e necesară o *declaraţie*, dacă obiectul e folosit înainte de *definiţie*

ex. `printf` e *declarată* în `stdio.h` şi *definită* într-o bibliotecă standard

Sintaxa declaraţiilor

Forma generală: listă de obiecte cu acelaşi tip de bază, evtl. iniţializate:

`int i = 1, n, tab[20], f(double, int);`

Sintaxa cu tipul de bază în faţă e similară cu folosirea în expresii:

`tab[ceva]` este un `int` `f(ceva1, ceva2)` este un `int`

declaratie ::= *specificatori*$_{opt}$ *tip* *lista-declaratori-init* ;

lista-declaratori-init ::= *declarator-init*

| *lista-declaratori-init* , *declarator-init*

declarator-init ::= *declarator*

| *declarator* = *iniţializator*

declarator ::= *identificator*

| *declarator* [*expresie*] pt. tablouri

| *declarator* (*parametri*) pt. funcţii

| * *declarator* pt. pointeri

specificatori: `extern`, `static`, `const`, `typedef`, `inline` etc.

Domeniul de vizibilitate al identificatorilor

Pt. orice identificator, compilatorul trebuie să-i decidă semnificaţia
Identificatorii obişnuiţi: variabile, tipuri, funcţii, constante enumerare au un *spaţiu de nume* comun (NU: variabilă şi funcţie cu acelaşi nume)

Q1: Un identificator poate fi folosit într-un punct de program ?
R: *Domeniul de vizibilitate* (al unei declaraţii / al unui identificator)
– domeniu de vizibilitate la nivel de *fişier* (*file scope*)
pentru identificatori declaraţi în afara oricărui bloc (oricărei funcţii)
din punctul de declaraţie până la sfârşitul fişierului compilat
– domeniu de vizibilitate la nivel de *bloc* (*block scope*)
pentru identificatori declaraţi într-un bloc { } (corp de funcţie, instrucţiune compusă) şi pentru parametrii unei funcţii
din punctul de declaraţie până la acolada } care închide blocul

Un identificator poate fi *redeclarat* într-un bloc interior şi îsi recapătă vechea semnificaţie când blocul ia sfârşit.

Domeniu de vizibilitate: Exemplu

```
int m, n, p; float x, y, z;       /* m1, n1, p1, x1, y1, z1 */
void f(int n, int x) {            /* n2, x2: alt n, alt x */
  int i; float y = 1;             /* i1, y2 */
  m = p; p = n;                   /* m1 = p1; p1 = n2; */
  for (i = 0; i < 10; ++i) {
    float x = i*i;                /* x3 = i1 * i1; */
    z += x;                       /* z1 += x3; */
  }
  z += x + y;                     /* z1 += x2 + y2 */
}
void main(void) {
  int i=0, m=3, x=2;              /* i2, m2, x4 */
  z = f(m, x);                    /* z1 = f(m2, x4); */
  x = f(i, y);                    /* x4 = f(i2, y1); */
}
```

Legătura dintre identificatori (linkage)

Q2: Două declarații ale unui identificator se referă la aceeași entitate?
R: Tipul de legătură (*linkage*) al unui identificator (obiect/funcție)

– *extern*: toate declarațiile identificatorului din toate fișierele care compun un program se referă la același obiect sau funcție
  pentru declarațiile *la nivel de fișier* fără specificator de memorare sau declarația cu specificatorul `extern` a unui identificator care nu a fost deja declarat cu tipul de legătură *intern*

– *intern*: toate declarațiile identificatorului din fișierul curent se referă la același obiect sau funcție; nu se propagă în exteriorul fișierului
  pt. declarațiile *la nivel de fișier* cu specificatorul de memorare `static`

– *fără legături* (no linkage): fiecare declarație denotă o entitate unică
  pentru declarațiile *la nivel de bloc* fără specificatorul `extern`

Declaraţii, definiţii tentative şi definiţii externe

În general, un identificator poate fi *declarat* de mai multe ori,
dar poate fi *definit* o singură dată.
Pentru funcţii, *declaraţia* şi *definiţia* au forme diferite:
declaraţia conţine doar antetul, definiţia conţine şi corpul funcţiei.

Un identificator fără legături nu va fi declarat de > 1 ori într-un bloc.

O *declaraţie* a unui identificator pt. un obiect e o *definiţie externă*
– dacă declaraţia la nivel de fişier, cu un iniţializator
– dacă declaraţia e la nivel de bloc, fără legături (vezi mai sus)

O declaraţie de obiect la nivel de fişier, fără iniţializare, şi fără specificator de memorare, sau cu specificatorul `static` e o *definiţie tentativă*.

Dacă un identificator are în fişier una sau mai multe definiţii tentative, dar nici o definiţie externă, se comportă ca şi pentru o definiţie externă la nivel de fişier, cu iniţializator nul.

Declaraţii/definiţii, tipuri de linkage. Exemplu

```
int x;                          /* x1, def. tentativă, linkage extern */
extern int y, z;                /* y1, z1, declaratie, linkage extern */
static int m;                   /* m1, def. tentativă, linkage intern */
int y;                          /* y1, def. tentativă, vezi y mai sus */
extern int m;                   /* decl. link. intern, vezi m mai sus */
void f(int x);                  /* declaraţie funcţie, linkage extern */
static int g(double);           /* declaraţie funcţie, linkage intern */
int h(int);                     /* declaraţie funcţie, linkage extern */
void f(int m);                  /* decl. link. extern, vezi f mai sus */
extern int h(int y);            /* decl. link. extern, vezi h mai sus */
void f(int x) { z=x; }          /* definiţia funcţiei f; z1 = x2; */
void main(void)                 /* definiţia funcţiei main */
{
  static int m;                 /* m2, definiţie, no linkage */
  int x;                        /* x3, definiţie, no linkage */
  { extern int x; }             /* declaraţie, linkage extern, x1 sus */
}                               /* g, h, z au definiţii în alt fişier */
```

Durata de memorare a obiectelor

Q3: Ce timp de viață/durată de memorare are un obiect în program?
R: 3 feluri diferite: *static*, *automatic* și *alocat* (discutat ulterior)
Pe întreaga durată de viață, un obiect are o *adresă constantă*
și își *păstrează ultima valoare* memorată.

Durată de memorare *statică*:
pentru obiecte declarate cu tipul de legătură *extern* sau *intern*,
sau declarate cu specificatorul de memorare `static`
– timp de viață: *întreaga execuție* a programului.
– obiectul e *inițializat o singură dată*, înainte de lansarea în execuție.

Durată de memorare *automată*: pentru obiecte fără legătură
– timp de viață: de la intrarea în blocul asociat până la încheierea sa
– la fiecare apel recursiv, se crează o nouă instanță a obiectului
– *valoarea inițială: nedeterminată*;
– o eventuală inițializare în declarație e repetată de câte ori e atinsă

Declaraţii de tablouri

Exemple: `char sir[20];` `double mat[6][5];`

Sintaxa: *specificatori*$_{opt}$ *tip* *ident* `[` *D1* `]` ... `[` *Dn* `]` *iniţializare*$_{opt}$

declară un tablou n-dimensional de D1 $\times \ldots \times$ Dn elemente de *tip*

de fapt: tablou de D1 elem. care sunt tablouri de ... Dn elem. de *tip*

Atenţie: în C, numerotarea elementelor în tablou începe de la zero!

În ANSI C, tablourile se declară doar cu dimensiuni *constante* (pozitive)

În C99, tablourile declarate local pot avea dimensiuni evaluate la rulare

```
void f(int n) { char s[n + 3]; /* prelucrează s */ }
```

Un tablou fără dimensiune dată, neiniţializat (`int a[];`) are 1 element!

Şiruri de caractere: caz particular de tablouri de `char`

– în memorie, sfârşitul unui şir e indicat de caracterul special `'\0'` (nul)

Atenţie: toate funcţiile care lucrează cu şiruri depind de acest lucru !

(dar convenţia nu are legătură cu aspectul în text, de ex. la citire)

– constante şir: cu ghilimele duble (`"test"`), terminate implicit cu `'\0'`

Inițializarea

– variabilele cu durată de memorare *statică* sunt inițializate înainte de execuție: implicit cu zero; explicit pot fi inițializate doar cu constante

– variabilele cu durată *automată* pot fi inițializate cu expresii arbitrare (ori de câte ori inițializarea e atinsă la rulare)

Pentru variabilele de tip tablou, inițializatorii se scriu între acolade
– nivelele de acolade indică sub-obiectele inițializate

```
int m[2][3] = { { 1, 0, 0 }, { 0, 1, 0 } };
```

– dacă nu, inițializatorii se folosesc pe rând, în ordinea indicilor

```
int c[2][2][2] = { { 1, 1, 1 }, { { 1, 0 }, 1 } };
```

– pt. inițializator mai mic ca dimensiunea, restul nu e inițializat explicit (v. `c[0][1][1]`, `c[1][1][1]`); când e mai mare, restul se ignoră

`char msg[4] = "test";` ca și `char msg[4] = { 't','e','s','t' };`

– dacă dimensiunea nu e dată explicit, se deduce din inițializator

`char msg[] = "test";` ca și `char msg[5] = { 't','e','s','t','\0' };`

– când se specifică elementul de inițializat, se continuă apoi în ordine:

```
int t[10] = { 1, 2, 3, [8] = 2, 1 };  /* t[3]-t[7] nespecificate */
```

Declaraţii de funcţii

Declaraţia: prototipul (antetul) funcţiei: tip, nume, tipul parametrilor

declaraţie-funcţie ::= *antet-funcţie* ;

decl-fct ::= *specificatori*$_{opt}$ *tip ident* (*param-type-list*)

param-type-list ::= *param-list* | *param-list* , ...

param-list ::= *param-decl* | *param-list* , *param-decl*

param-decl ::= *specificatori*$_{opt}$ *tip* | *specificatori*$_{opt}$ *tip declarator*

```
int abs(int n);   int getchar(void);   double pow(double, double);
```

– tipul returnat nu poate fi *tablou*; poate fi `void` (nimic)

– *numele* parametrilor nu e relevant în *declaraţie* şi poate lipsi

– o funcţie poate fi declarată repetat, cu declaraţii compatibile

– număr *variabil* de parametri dacă lista se termină în `...` (v. ulterior)

– declaraţia doar cu `()` nu specifică parametrii şi e perimată

– specificatorul `inline` e o indicaţie de optimizare pentru viteză;
se rezumă la fişierul curent; depinde de implementare (vezi standard)

Definiţii de funcţii

Sintaxa: *definiţie-funcţie* ::= *antet-funcţie* *bloc*

– *blocul* conţine declaraţii şi instrucţiuni (corpul funcţiei)

– parametrii specificaţi şi prin nume (vizibilitate în corpul funcţiei)

Transferul parametrilor în C se face *prin valoare*

– *expresiile* date ca argumente în apelul de funcţie sunt evaluate şi atribuite parametrilor formali (cu eventuale conversii ca la atribuire)

  – ordinea de evaluarea a argumentelor nu e specificată

  – dispunerea în memorie a argumentelor (pe stivă) nu e specificată

– se execută corpul funcţiei; se revine la instrucţiunea de după apel

Tablouri şi apeluri de funcţii

O funcţie nu poate returna o valoare de tip tablou.

Pentru un tablou declarat *tip* `t[D1]...[Dn];` compilatorul trebuie să calculeze poziţia unui element `t[i1]...[in]` faţă de începutul tabloului. Numărul de elemente dinaintea acestuia este:

$$i_1 \cdot D_2 \cdot \ldots \cdot D_n + i_2 \cdot D_3 \cdot \ldots \cdot D_n + \ldots + i_{n-1} \cdot D_n + i_n$$

$\Rightarrow$ trebuie cunoscute dimensiunile D2 ... Dn, dar nu şi D1

În ANSI C, o funcţie trebuie să precizeze ca şi constante dimensiunile parametrilor tablou (în afară de prima):

```
void addmat(int a[][5], int b[][5]); /* nu merge pentru c[][4] */
```

$\Rightarrow$ e greu de scris funcţii flexibile (ex. înmulţirea de matrici oarecare)

În C99, se pot specifica parametri tablou de dimensiuni variabile:

```
void addmat(int m, int n, int a[m][n], int b[m][n]);
```

Specificatori de memorare. Definiţii de tip

engl. *storage class specifier*; se indică cel mult unul pe declaraţie.

`extern`, `static`: discutaţi mai sus

– stabilesc atât proprietăţile de *linkage* cât și durata de memorare

`auto`: implicit pentru variabile declarate la nivel de bloc

`register`: indicaţie către compilator de a optimiza pentru viteză accesul la o variabila (alocând pentru ea un registru dacă e posibil)

– nu se poate folosi operatorul adresă pentru variabile `register`

Definiţii de tip: `typedef` *declaraţie*

```
typedef unsigned long size_t;          typedef unsigned char byte;
```

– dacă în *declaraţie*, identificatorul ar fi o *variabilă* de un anumit tip, atunci `typedef` *declaraţie* definește identificatorul ca *numele* acelui tip

Ex: în `int mat3x5[3][5];` `mat3x5` ar fi o matrice de 3x5 întregi.

```
typedef int mat3x5[3][5];    /* mat3x5 e tipul tablou de 3x5 int */
mat3x5 A, B;              /* A, B sunt variabile tablou de 3x5 int */
```

Calificatori de tip (type qualifiers)

const

specifică interzicerea modificării prin program a valorii obiectului
ex. pt. declaraţii de constante: `const int MAX = 100;`
– nu se permite folosirea de operatori de atribuire pt. obiecte `const`
– compilatorul e liber să le aloce în memorie read-only

volatile

– obiectul poate fi modificat în mod necunoscut implementării
(ex. port hardware, întrerupere asincronă, program concurent)
⇒ indicaţie către compilator să citească valoarea curentă din memorie
la fiecare folosire, fără optimizări (cf. `register`)
– combinat cu `const`: obiectul poate fi modificat doar extern:

```
extern const volatile int real_time_clock;
```

restrict

– stabileşte că un obiect poate fi modificat doar prin pointerul indicat
(calificator folosit doar cu pointeri, permite optimizări de compilare)

Instrucţiunile limbajului C

Instrucţiunea *expresie* *expresie*$_{opt}$;

– *orice* expresie, evaluată pentru efectele ei laterale; în particular:

expresii de *atribuire*: `x = y + 1; y *= 2; --z;`

apel de funcţie (ignorând valoarea returnată): `printf("salut!\n");`

instrucţiunea *vidă* ; (expresia lipseşte)

Exemplu: ciclu cu corp vid `while (s[i++]);`

Obs: în C ; nu e separator, ci *face parte* din anumite instrucţiuni

Instrucţiunea *compusă* (bloc) { *lista-declaraţii* *lista-instrucţiuni* }

grupează declaraţiile/instrucţiunile din listă sintactic într-o instrucţiune

poate fi încuibată (conţine alte blocuri); poate fi vidă { }

lista-declaraţii nu poate conţine *definiţii* de funcţii

În C99 (şi în C++) un bloc poate conţine declaraţii şi instrucţiuni în orice ordine.

Instrucţiuni etichetate

identificator `:` *instrucţiune*
`case` *expresie-constantă-int* `:` *instrucţiune*
`default :` *instrucţiune*

Etichetele au *spaţiu de nume* separat de cel al identificatorilor obişnuiţi (putem avea variabile/funcţii/etc. şi etichete cu acelaşi nume)

Domeniul de vizibilitate al etichetei e corpul funcţiei în care se află (numele de etichete trebuie să fie unice în cadrul unei funcţii)

Etichetele `case` şi `default` pot apare doar în instrucţiunea `switch`. Într-o instrucţiune `switch` poate exista cel mult o etichetă `default` iar constantele întregi din etichetele `case` vor fi distincte.

Instrucţiuni de selecţie

Instrucţiunea `if` `if ( expresie ) instrucţiune1`

sau `if ( expresie ) instrucţiune1 else instrucţiune2`

– expresia trebuie să fie de tip scalar (întreg, real, enumerare)

– dacă expresia e nenulă se execută *instrucţiune1*, altfel *instrucţiune2*

– un `else` e asociat întotdeauna cu cel mai apropiat `if`

Instrucţiunea `switch` `switch ( expresie-intreagă ) instrucţiune`

– se evaluează expresia (de tip întreg, posibil limitată la 1023 valori)

– dacă în corpul *instrucţiune* (compusă) există o etichetă `case`
cu valoarea întreagă obţinută, se sare la instrucţiunea respectivă

– dacă nu, şi există o etichetă `default`, se sare la acea instrucţiune

– altfel nu se execută nimic (se trece la instrucţiunea următoare)

– pt. acelaşi cod la mai multe etichete: `case` *val1* `:` `case` *val2* `:` *şir-instr*

Obs: Execuţia nu se opreşte la următorul `case` (e doar o etichetă);
ieşirea din `switch`: doar cu instruct. `break` sau la sfârşitul corpului!

$\Rightarrow$ permite utilizarea de cod comun pe ramuri, dar cu mare atenţie!

Instrucţiunea `switch`: exemplu

```
char c; int a, b, r;
printf("Scrieti o operatie intre doi intregi: ");
if (scanf("%d %c %d", &a, &c, &b) == 3) { /* toate 3 corect */
  switch (c) {
    case '+': r = a + b; break;  /* iese din corpul switch */
    case '-': r = a - b; break;  /* idem */
    default: c = '\0'; break;    /* fanion caracter eronat */
    case 'x': c = '*';    /* 'x' e tot înmulţire, continuă */
    case '*': r = a * b; break;  /* ca şi pt.'*' apoi iese */
    case '/': r = a / b;    /* la sfârşit nu trebuie break */
  }
  if (c) printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);
  else printf("Operaţie necunoscută\n");
} else printf("Format eronat\n");
```

Ciclurile cu test iniţial şi final

Instrucţiunile *while* şi *do* (ciclurile cu test iniţial şi final)

`while (` *expresie* `)` *instrucţiune*

`do` *instrucţiune* `while (` *expresie* `);`

– ambele execută *instrucţiunea* atât timp cât valoarea expresiei (de tip scalar) e nenulă (adevărată).

– diferă momentul de evaluare a expresiei (înainte/după fiecare iteraţie)

Obs: În Pascal, din `repeat ... until` se iese pe condiţie *true* (invers!)

Instrucţiunea `for`

`for` (*exp-init* ; *exp-test* ; *exp-cont*)
instrucţiune

e echivalentă* cu:

exp-init;
`while` (*exp-test*) {
instrucţiune;
exp-cont;
}

* excepţie: instrucţiunea `continue`, vezi ulterior

– oricare din cele 3 expresii poate lipsi (dar cele două ; ramân)
– dacă *exp_test* lipsește, e tot timpul adevărată (ciclu infinit)

În C99 (ca și în C++) se permite ca expresia *exp-init* să fie înlocuită cu o *declarație* de variabile (evtl. inițializate) cu domeniu de vizibilitate întreaga instrucțiune.

```
for (int i = 0; i < 10; ++i) { /* corpul */ }
```

Instrucţiunea `return`

`return` *expresie*$_{opt}$;

– încheie execuţia funcţiei curente

– returnează valoarea expresiei date (dacă este prezentă)

Obs: într-o funcţie care nu are tipul `void`, fiecare cale prin cod trebuie să returneze o valoare (compilatorul avertizează în caz contrar)

```
int pos(char s[], char c)       /* prima pozitie a lui c in s */
{
  int i = 0;
  do
    if (s[i] == c) return i;    /* returnează poziţia găsită */
  while (s[i++]);
  return -1;                    /* -1 ca fanion, nu s-a găsit */
}
```

– funcţia `main` returnează un cod (succes/eroare) sistemului de operare

– uzual, se declară ca `int` şi returnează implicit 0 (succes)

Instrucţiunea `break`

– produce ieşirea din corpul instrucţiunii `while, do, for` sau `switch` *imediat înconjurătoare*; execuţia continuă cu instrucţiunea următoare
– mai convenabilă decât testarea unei variabile boolene la ciclul următor
– mai lizibil, dacă codul peste care se sare e complex

```
const int MAX = 20;
int i, t[MAX], v;
/* caută pe v în tabloul t */
for (i = MAX; --i >= 0; )
  if (t[i] == v) break;
if (i == -1) printf("nu s-a găsit\n");
else printf("găsit la poziţia %d\n", i);
```

Instrucțiunea `continue`

– produce trecerea la sfârșitul iterației într-un ciclu `while`, `do` sau `for` începând cu testul pt. `while` și `do`, și cu *expr3* (actualizare) pt. `for` (controlul trece la punctul din ciclu de după ultima instrucțiune)
– la fel, cod mai lizibil, dacă partea neexecutată din iterație e complexă

```
for (d = 2; ; d++) {      /* descompune n > 1 în factori primi */
  if (n % d != 0) continue;  /* nu se împarte, următorul! */
  exp = 0;
  do                      /* repetă de câte ori d e factor */
    exp++;
  while ((n /= d) % d == 0);
  printf ("%d^%d ", d, exp);      /* scrie factorul curent */
  if (n == 1) break;              /* am terminat */
}
```

Instrucţiunea `goto`

Sintaxa: `goto` *eticheta* `;`
Efectul: se sare la execuţia instrucţiunii cu *eticheta* specificată
Obs: orice instrucţiune poate fi etichetată opţional *etichetă* `:` *instr*

– instrucţiunea `goto` nu corespunde principiilor programării structurate
– de evitat: duce uşor la programe dificil de înţeles şi analizat
– orice program poate fi rescris fără folosirea lui `goto`
(eventual utilizând teste şi/sau variabile boolene suplimentare)
– poate fi totuşi utilă, ex. pentru ieşirea din mai multe cicluri încuibate

```
while (...) { /* scriem într-un fişier, linie cu linie */
  while (...) { /* prelucrăm cuvintele şi spaţiile din linie */
    if (eroare_la_scriere)
      goto eroare; /* abandonează ciclurile */
  }
}
eroare: /* cod pt. tratarea erorii */
```

Raționamente despre programe

Pentru a fi convinși că programul e corect, demonstrăm că execuția sa are efectul dorit $\Rightarrow$ formalism matematic (Floyd'67, Hoare'69)
Corectitudine parțială: $\{P\}\ S\ \{Q\}$ dacă S e executat într-o stare care satisface P, și execuția lui S se termină, starea rezultantă satisface Q

Regulile lui Hoare: definite pentru fiecare tip de instrucțiune în parte; prin combinația lor, se poate raționa despre programe întregi

Atribuire: $\dfrac{}{\{Q[x/E]\}\ x := E\ \{Q\}}$ unde $Q[x/E]$ e substituția lui x cu E în Q

Ex: $\{x = y - 2\}\ x := x + 2\ \{x = y\}$ (în rezultat, $x = y$, substituim x cu expresia atribuită, $x + 2$ și obținem $x + 2 = y$, deci $x = y - 2$)

Secvențiere: $$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Decizie: $$\frac{\{P \wedge E\}\ S_1\ \{Q\} \qquad \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \text{if } E \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

Regulile lui Hoare (cont.)

Ciclul cu test (iniţial): cheia în raţionamentul despre programe
– trebuie găsit un *invariant* I = o proprietate menţinută adevărată de fiecare execuţie a ciclului (exprimată în punctul dintre iteraţii)
– dacă intrăm în ciclu (E), invariantul e menţinut după o iteraţie S
– dacă nu mai intrăm ($\neg E$), invariantul implică concluzia Q

$$\frac{\{I \wedge E\}\ S\ \{I\} \qquad I \wedge \neg E \Rightarrow Q}{\{I\}\ \text{while}\ E\ \text{do}\ S\ \{Q\}}$$

```
while (lo < hi) {   /* căutare binară; I: lo <= n && n <= hi */
  m = (lo + hi) / 2;
  if (n > m)        /* ambele cazuri menţin lo<=n && n<=hi */
    lo = m+1;       /* n > m => n >= m+1 => n >= lo */
  else hi = m;      /* !(n < m) => n <= m => n <= hi */
}                   /* I rămâne adevărat */
n = lo;          /* lo<=n && n<=hi && !(lo<hi) => lo==n && n==hi */
```

Funcţii matematice standard (declarate în `math.h`)

Funcţii de conversie

`double fabs(double x);` valoarea absolută a lui `x`

`double floor(double x);` partea întreagă $\lfloor$`x`$\rfloor$ a lui `x`, ca `double`

`double ceil(double x);` cel mai mic întreg $\lceil$`x`$\rceil$ nu mai mic de `x`

`double trunc(double x);` trunchează argumentul la întreg, înspre 0

Funcţii de rotunjire (Obs: direcţia de rotunjire poate fi controlată cu `fgetround()` şi `fsetround()` din `fenv.h`, detalii în standard)

`double nearbyint(double x);` rotunjesc în direcţia curentă cu/

`double rint(double x)` /fără excepţie de argument *inexact*

(implementarea/tratarea excepţiilor e definită în standard, v. `fenv.h`)

`double round(double x)`: rotunjeşte jumătăţile în direcţia opusă lui zero

`long int lrint(double x);` `long int lround(double x);`

ca şi `rint()`, `round()` dar rezultat întreg; nedefinit în caz de depăşire

Funcţiile din `math.h` au variante cu sufixele `f` şi `l` cu argumente şi rezultate `float` sau `long double`. Exemple: `float fabsf(float);` `long double fabsl(long double);`

Funcții standard din `math.h` (cont.)

Funcții de exponențiere și logaritmice

`double exp(double x);` returnează e^x

`double exp2(double x);` returnează 2^x

`double log(double x);` returnează logaritmul natural $ln\ x$

`double log10(double x);` `double log2(double x);` log. în baza 10 și 2

`double pow(double x);` returnează x^y

`double sqrt(double x);` returnează $\sqrt{x}$

Funcții trigonometrice și hiperbolice

`acos`, `asin`, `atan`, `cos`, `sin`, `tan`, `acosh`, `asinh`, `atanh`, `cosh`, `sinh`, `tanh`

(valori unghiulare în radiani; inversele returnează valori principale)

`double atan2(double y, double x);` returnează $arctg(y/x)$ în intervalul $[-\pi, \pi]$, determină cadranul după semnele ambelor argumente