

Structura programului: declarații și definiții

Un program C: compus din ≥ 1 *unități de compilare* (fișiere). Fiecare: un șir de *declarații* (de tipuri, variabile, funcții) sau *definiții de funcții*.

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

O *declarație* specifică interpretarea și atributele unui *identificator*
 – pentru o variabilă, numele și tipul
 – pentru o funcție, numele, tipul, și tipul parametrilor

O *definiție* e o declarație care specifică *complet* identificatorul respectiv
 – pentru o variabilă, în plus, are ca efect alocarea memoriei
 – pentru o funcție, include corpul funcției

Un identificator nu poate fi folosit înainte de a fi *declarat*.

– e necesară o *declarație*, dacă obiectul e folosit înainte de *definiție*
 ex. `printf` e *declarată* în `stdio.h` și *definită* într-o bibliotecă standard

Programarea calculatoarelor 2. Curs 3

Marius Minea

Declarații. Instrucțiuni

20 octombrie 2003

Programarea calculatoarelor 2. Curs 3

Marius Minea

Declarații. Instrucțiuni

3

Sintaxa declarațiilor

Forma generală: listă de obiecte cu același tip de bază, evtl. inițializate:

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

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

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

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

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

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

declarator-init ::= *declarator*

| *declarator* = *inițializator*

declarator ::= *identificator*

| *declarator* [*expresie*]

pt. tablouri

| *declarator* (*parametri*)

pt. funcții

| * *declarator*

pt. pointeri

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

Programarea calculatoarelor 2. Curs 3

Marius Minea

Declarații. Instrucțiuni

5

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); */
}
```

Programarea calculatoarelor 2. Curs 3

Marius Minea

Declarații. Instrucțiuni

4

Domeniul de vizibilitate al identificatorilor

Pt. orice identificator, compilatorul trebuie să-i decidă semnificația
Identificatorii obișnuiți: variabile, tipuri, funcții, constante enumerare
 au un *spațiu de nume* comun (NU: variabilă și funcție cu același nume)

Q1: Un identificator poate fi folosit într-un punct de program ?

R: *Domeniul de vizibilitate* (al unei declarații / al unui identificator)

– domeniu de vizibilitate la nivel de *fișier* (*file scope*)
 pentru identificatori declarați în afara oricărui bloc (oricărei funcții)
 din punctul de declarație până la sfârșitul fișierului compilat
 – domeniu de vizibilitate la nivel de *bloc* (*block scope*)
 pentru identificatori declarați într-un bloc { } (corp de funcție,
 instrucțiune compusă) și pentru parametrii unei funcții
 din punctul de declarație până la acolada } care închide blocul

Un identificator poate fi *redeclarat* într-un bloc interior și își recapătă
 vechea semnificație când blocul ia sfârșit.

Programarea calculatoarelor 2. Curs 3

Marius Minea

Declarații. Instrucțiuni

6

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`

Programarea calculatoarelor 2. Curs 3

Marius Minea

Declarații, definiții tentative și definiții externe

În general, un identificator poate fi *declarat* de mai multe ori, dar poate fi *definit* o singură dată.

Pentru funcții, *declarația* și *definiția* au forme diferite:

declarația conține doar antetul, definiția conține și corpul funcției.

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

O *declarație* a unui identificator pt. un obiect e o *definiție externă*

– dacă declarația la nivel de fișier, cu un inițializator

– dacă declarația e la nivel de bloc, fără legături (vezi mai sus)

O declarație de obiect la nivel de fișier, fără inițializare, și fără specificator de memorare, sau cu specificatorul *static* e o *definiție tentativă*.

Dacă un identificator are în fișier una sau mai multe definiții tentative, dar nici o definiție externă, se comportă ca și pentru o definiție externă la nivel de fișier, cu inițializator nul.

Declarații/definiții, tipuri de linkage. Exemplu

```
int x;                /* x1, def. tentativă, linkage extern */
extern int y, z;     /* y1, z1, declarație, 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);      /* declarație funcție, linkage extern */
static int g(double); /* declarație funcție, linkage intern */
int h(int);         /* declarație funcție, 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; } /* definiția funcției f; z1 = x2; */
void main(void)     /* definiția funcției main */
{
    static int m;    /* m2, definiție, no linkage */
    int x;           /* x3, definiție, no linkage */
    { extern int x; } /* declarație, linkage extern, x1 sus */
}                  /* g, h, z au definiții în alt fișier */
```

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

Declarații de tablouri

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

Sintaxa: `specificatoriopt tip ident [D1] ... [Dn] inițializareopt`

declară un tablou n-dimensional de D1 × ... × Dn elemente de tip de fapt: tablou de D1 elem. care sunt tablouri de ... Dn elem. de tip **Atenție**: 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, neinițializat (`int a[];`) are 1 element!

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

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

Atenție: toate funcțiile care lucrează cu șiruri depind de acest lucru !

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

– constante șir: 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 */

Declarații de funcții

Declarația: prototipul (antetul) funcției: tip, nume, tipul parametrilor

declarație-funcție ::= antet-funcție ;

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 *declarație* și poate lipsi

– o funcție poate fi declarată repetat, cu declarații compatibile

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

– declarația doar cu () nu specifică parametrul și e perimată

– specificatorul *inline* e o indicație de optimizare pentru viteză;

se rezumă la fișierul curent; depinde de implementare (vezi standard)

Definiții de funcții

Sintaxa: *definiție-funcție ::= antet-funcție bloc*
 – *blocul* conține declarații și instrucțiuni (corpul funcției)
 – parametrii specificați și prin nume (vizibilitate în corpul funcției)

Transferul parametrilor În C se face *prin valoare*

– *expresiile* date ca argumente în apelul de funcție sunt evaluate și 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 funcției; se revine la instrucțiunea de după apel

Tablouri și apeluri de funcții

O funcție nu poate returna o valoare de tip tablou.

Pentru un tablou declarat *tip* $t[D_1] \dots [D_n]$; compilatorul trebuie să calculeze poziția unui element $t[i_1] \dots [i_n]$ față de începutul tabloului. Numărul de elemente dinaintea acestuia este:

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

⇒ trebuie cunoscute dimensiunile $D_2 \dots D_n$, dar nu și D_1

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

`void addmat(int a[][5], int b[][5]); /* nu merge pentru c[][4] */`
 ⇒ e greu de scris funcții flexibile (ex. înmulțirea 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. Definiții de tip

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

extern, static: discutați mai sus
 – stabilesc atât proprietățile de *linkage* cât și durata de memorare
auto: implicit pentru variabile declarate la nivel de bloc
register: indicație către compilator de a optimiza pentru viteză accesul la o variabilă (alocând pentru ea un registru dacă e posibil)
 – nu se poate folosi operatorul adresă pentru variabile **register**

Definiții de tip: *typedef declarație*

`typedef unsigned long size_t;` `typedef unsigned char byte;`
 – dacă în *declarație*, identificatorul ar fi o *variabilă* de un anumit tip, atunci *typedef declarație* 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. declarații 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)
 ⇒ indicație 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

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

Instrucțiunile limbajului C

Instrucțiunea *expresie* *expresie*_{opt} ;

– orice expresie, evaluată pentru efectele ei laterale; în particular:
 expresii de *atribuire*: `x = y + 1; y *= 2; --z;`
apel de funcție (ignorând valoarea returnată): `printf("salut!\n");`
 instrucțiunea *vidă* ; (expresia lipsește)

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

Obs: În C ; nu e separator, ci *face parte* din anumite instrucțiuni

Instrucțiunea *compusă* (bloc) `{ lista-declarații lista-instrucțiuni }`
 grupează declarațiile/instrucțiunile din listă sintactic într-o instrucțiune
 poate fi încuibată (conține alte blocuri); poate fi vidă `{ }`
lista-declarații nu poate conține *definiții* de funcții

În C99 (și în C++) un bloc poate conține declarații și instrucțiuni în orice ordine.

Instrucțiuni etichetate

identificator : *instrucțiune*

case *expresie-constantă-int* : *instrucțiune*

default : *instrucțiune*

Etichetele au *spațiu de nume* separat de cel al identificatorilor obișnuiți (putem avea variabile/funcții/etc. și etichete cu același nume)

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

Etichetele **case** și **default** pot apărea doar în instrucțiunea **switch**.
 Într-o instrucțiune **switch** poate exista cel mult o etichetă **default** iar constantele întregi din etichetele **case** vor fi distincte.

Instructiunea **if** `if (expresie) instructiune1`
sau `if (expresie) instructiune1 else instructiune2`
– expresia trebuie să fie de tip scalar (întreg, real, enumerare)
– dacă expresia e nenulă se execută *instructiune1*, altfel *instructiune2*
– un `else` e asociat întotdeauna cu cel mai apropiat `if`

Instructiunea **switch** `switch (expresie-intreaga) instructiune`
– se evaluează expresia (de tip întreg, posibil limitată la 1023 valori)
– dacă în corpul *instructiune* (compusă) există o etichetă `case` cu valoarea întreagă obținută, se sare la instructiunea respectivă
– dacă nu, și există o etichetă `default`, se sare la acea instructiune
– altfel nu se execută nimic (se trece la instructiunea următoare)
– pt. același cod la mai multe etichete: `case val1: case val2: șir-instr`
Obs: Execuția nu se oprește la următorul `case` (e doar o etichetă);
ieșirea din `switch`: doar cu `break` sau la sfârșitul corpului!
⇒ permite utilizarea de cod comun pe ramuri, dar cu mare atenție!

```
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 înmulțire, continuă */
        case '*': r = a * b; break; /* ca și pt. '*' apoi iese */
        case '/': r = a / b; /* la sfârșit nu trebuie break */
    }
    if (c) printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);
    else printf("Operație necunoscută\n");
} else printf("Format eronat\n");
```

Ciclurile cu test inițial și final

Instructiunile **while** și **do** (ciclurile cu test inițial și final)
`while (expresie) instructiune`
`do instructiune while (expresie);`
– ambele execută *instructiunea* atât timp cât valoarea expresiei (de tip scalar) e nenulă (adevărată).
– diferă momentul de evaluare a expresiei (înainte/după fiecare iterație)
Obs: În Pascal, din `repeat ... until` se iese pe condiție `true` (invers!)

Instructiunea for

```
for (exp-init ; exp-test ; exp-cont)
    instructiune
    e echivalentă* cu:
    exp-init;
    while (exp-test) {
        instructiune;
        exp-cont;
    }
* excepție: instructiunea 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 instructiune.
for (int i = 0; i < 10; ++i) { /* corpul */ }
```

Instructiunea return

`return expresie_opt ;`
– încheie execuția funcției curente
– returnează valoarea expresiei date (dacă este prezentă)
Obs: într-o funcție 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 poziția găsită */
 while (s[i++]);
 return -1; /* -1 ca fanion, nu s-a găsit */
}
– funcția `main` returnează un cod (succes/eroare) sistemului de operare
– uzual, se declară ca `int` și returnează implicit 0 (succes)

Instructiunea break

– produce ieșirea din corpul instrucțiunii `while`, `do`, `for` sau `switch`
imediat înconjurătoare; execuția continuă cu instrucțiunea 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 poziția %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 */
}
```

Programarea calculatoarelor 2. Curs 3

Marius Minea

Raționamente despre programe

Pentru a fi convingși 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: $\frac{\{Q[x/E]\} x := E \{Q\}}{\{Q[x/E]\} x := E \{Q\}}$ 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: $\frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}}$

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

Programarea calculatoarelor 2. Curs 3

Marius Minea

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

Funcții 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);` trunchiază argumentul la întreg, înspre 0
Funcții de rotunjire (Obs: direcția de rotunjire poate fi controlată cu `fgetround()` și `fsetround()` din `fcntl.h`, detalii în standard)
`double nearbyint(double x);` rotunjesc în direcția curentă cu/
`double rint(double x)` /fără excepție de argument *inexact* (implementarea/tratarea excepțiilor e definită în standard, v. `fcntl.h`)
`double round(double x);` rotunjește jumătățile în direcția opusă lui zero
`long int lrint(double x);` `long int lround(double x);`
ca și `rint()`, `round()` dar rezultat întreg; nedefinit în caz de depășire

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

Programarea calculatoarelor 2. Curs 3

Marius Minea

Instrucțiunea goto

Sintaxa: `goto eticheta ;`

Efectul: se sare la execuția instrucțiunii cu *eticheta* specificată

Obs: orice instrucțiune poate fi etichetată opțional *etichetă* : *instr*

– instrucțiunea `goto` nu corespunde principiilor programării structurate
– de evitat: duce ușor la programe dificil de înțeles și analizat
– orice program poate fi rescris fără folosirea lui `goto` (eventual utilizând teste și/sau variabile boolene suplimentare)
– poate fi totuși utilă, ex. pentru ieșirea din mai multe cicluri încuibate

```
while (...) { /* scriem într-un fișier, linie cu linie */
    while (...) { /* prelucrăm cuvintele și spațiile din linie */
        if (eroare_la_scriere)
            goto eroare; /* abandonează ciclurile */
    }
}
```

`eroare: /* cod pt. tratarea erorii */`

Programarea calculatoarelor 2. Curs 3

Marius Minea

Regulile lui Hoare (cont.)

Ciclul cu test (inițial): cheia în raționamentul despre programe

– trebuie găsit un *invariant* $I =$ o proprietate menținută adevărată de fiecare execuție a ciclului (exprimată în punctul dintre iterații)
– dacă intrăm în ciclu (E), invariantul e menținut după o iterație S
– dacă nu mai intrăm ($\neg E$), invariantul implică concluzia Q

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

```
while (lo < hi) { /* căutare binară; I: lo <= n && n <= hi */
    m = (lo + hi) / 2;
    if (n > m) /* ambele cazuri mențin 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 */
```

Programarea calculatoarelor 2. Curs 3

Marius Minea

Funcții standard din `math.h` (cont.)

Funcții de exponențiere și logaritmică

`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 cadrantul după semnele ambelor argumente

Programarea calculatoarelor 2. Curs 3

Marius Minea