

Declarații de variabile, tipuri, funcții

2 noiembrie 2004

Puțină teorie

- *sintaxa*: regulile gramaticale care descriu un limbaj
un șir de simboluri (text) face parte din limbaj ? (e bine format ?)
- *semantica*: înțelesul (semnificația) unui obiect din limbaj
rezultă din semnificația fiecărui element de program în parte
determină rezultatul execuției programului

Definim sintaxa elementelor de limbaj folosind anumite notații:

$::=$ pentru definiție | pentru alternative etc.

Convenție: *cursiv* pentru simboluri neterminale (definite la rândul lor)
tipărit pentru simboluri terminale (elemente lexicale)

instructiune_while ::= while (*condiție*)
 instructiune

BNF (Backus-Naur Form): notație formală pt. gramatica unui limbaj

Elemente lexicale

Prima fază de compilare: analiza lexicală = separarea în *atomi lexicali*:
unitățile elementare de limbaj care au o semnificație:

- *cuvinte cheie*: `int`, `void`, `while`, etc.
- *identificatori*: secvență de litere, cifre și `'_'` începând cu literă sau `'_'` folosiți pt. nume de variabile, funcții, tipuri, etichete, etc.

ATENȚIE ! În C se face distincție între majuscule și minuscule !!!

Lungimea *semnificativă* a identificatorilor: 31 (externi)/63 (interni)

(porțiunea suplimentară poate fi ignorată de unele compilatoare!)

- *constante*: `123`, `3.14`, `'\0'`, `"salut!\n"` etc.
- *semne de punctuație*

operatori: `+` `-` `=` `++` `&&` etc.

separatori: `{` `}` `(` `)` `;` etc.

Spațiile: necesare doar unde trebuie separați doi atomi lexicali alăturați
ex. `void main`, NU `voidmain`; NU `floatx=3.14`; ne semnificative în rest.

Indentati programele pt. citire ușoară ! (automat în editoarele bune)

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* (toate informațiile necesare pentru a-l folosi)

- 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

Declarații: forma generală

Întâlnite până acum: `float x; int a, b = 1; char t[20];`

Dar se pot declara deodată și mai multe obiecte cu același tip de bază:

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

declară un întreg inițializat cu 1, alt întreg neinițializat, un tablou de 20 de întregi, și o funcție întreagă cu doi parametri (double și 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`

declaratie ::= specificatori tip lista-decl-init ;

lista-decl-init ::= declarator-init | lista-decl-init , declarator-init

declarator-init ::= declarator

| declarator = inițializator

declarator ::= identificador

| declarator [expresie]

pt. tablouri

| declarator (parametri)

pt. funcții

*| * declarator*

pt. pointeri

Domeniul de vizibilitate al identificatorilor

Pt. orice identificador, 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 identificador poate fi folosit într-un punct de program ?

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

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

Domeniu de vizibilitate: Exemplu

```
int m, n, p; float x, y, z;      /* m1, n1, p1, x1, y1, z1 */
int 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; */
    }
    return 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); */
}
```

Variabile globale și locale

Dacă în declarația de variabile nu apar alți specificatori înainte de tip:

Variabile globale

= o variabilă declarată în afara oricărei funcții

- are spațiu de memorie alocat pe întreaga execuție a programului
- e inițializată o singură dată (cu valoarea dată explicit în declarație, sau implicit cu zero)
- e vizibilă în întreg textul programului începând cu declarația ei

Variabile locale (interne)

= o variabilă declarată în interiorul unui bloc (inclusiv de funcție)

- există doar atât timp cât programul execută blocul respectiv
- sunt inițializate cu valoarea dată la orice intrare în blocul respectiv (sau au o valoare nedefinită dacă declarația nu specifică inițializare)
- sunt vizibile doar în interiorul blocului respectiv

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 identicatorului 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 identicatorului 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*

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: *specificatori_{opt} tip ident [D1] ... [Dn] inițializare_{opt}*

declară un tablou n-dimensional de $D1 \times \dots \times Dn$ 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

(vezi `c[0][1][1]`, `c[1][1][1]`); când inițializatorul 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 */
```

Definiții de constante și tipuri

Definiții de tip: `typedef` *declarație*

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

- sintaxa: ca și declarația de variabile, prefixată cu `typedef`
- dacă în *declarație*, identificatorul ar fi o *variabilă* de un anumit tip, atunci `typedef` *declarație* definește identificatorul ca *numele* aceluiași 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 */
```

Declarații de constante

- cu *calificatorul de tip* `const`: `const int LEN = 10;`
- folosit pt. declararea de constante; constuie eroare modificarea lor
- nu se permite folosirea de operatori de atribuire pt. obiecte `const` (compilatorul e liber de exemplu să le aloce în memorie read-only)

Declarații și definiții de funcții

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

decl-fct ::= tip nume-fct (lista-decl-param) ;

lista-decl-param ::= void | decl-param , ... , decl-param

decl-param ::= tip | tip nume-param

```
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 parametrii 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: $\text{definiție-funcție} ::= \text{antet-funcție} \text{ bloc}$

- *blocul* conține declarații și instrucțiuni (corpul funcției)
- parametri 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

Transmiterea parametrilor: exemple

```
int a = 1, b = 2, m = 3; // primul a, m: a1, m1
int f (int a, int p, int n) // alt a: a2
{
    a = 2; // a2 = 2
    m = 5; n = 0; // m1 = 5
}
void main(void)
{
    int m = 4, n = 5, p = 6; // alt m: m2
    f (b+2, n, p); // f(4, 5 ,6);
    /* a = 1, m1 = 5, m2 = 4 */
}
```


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);` trunchează argumentul la întreg, înspre 0

Funcții de rotunjire (Obs: direcția de rotunjire poate fi controlată cu `fgetround()` și `fsetround()` din `fenv.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. `fenv.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);`

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