Declaraţii de variabile, tipuri, funcţii

23 octombrie 2002

Puţină teorie

– *sintaxa*: regulile gramaticale care descriu un limbaj
un şir de simboluri (text) face parte din limbaj ? (e bine format ?)
– *semantica*: înţelesul (semnificaţia) unui obiect din limbaj
rezultă din semnificaţia fiecărui element de program în parte
determină rezultatul execuţiei programului

Definim sintaxa elementelor de limbaj folosind anumite notaţii:
::= pentru definiţie | pentru alternative etc.
Convenţie: *cursiv* pentru simboluri neterminale (definite la rândul lor)
`tipărit` pentru simboluri terminale (elemente lexicale)
instructiune_while ::= `while` (*condiţie*)
instrucţiune

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

Elemente lexicale

Prima fază de compilare: analiza lexicală = separarea în *atomi lexicali*:
unităţile elementare de limbaj care au o semnificaţie:
– *cuvinte cheie*: `int`, `void`, `while`, etc.
– *identificatori*: secvenţă de litere, cifre şi '_' începând cu literă sau '_'
folosiţi pt. nume de variabile, funcţii, tipuri, etichete, etc.
ATENŢIE ! În C se face distincţie între majuscule şi minuscule !!!
Lungimea *semnificativă* a identificatorilor: 31 (externi)/63 (interni)
(porţiunea suplimentară poate fi ignorată de unele compilatoare!)
– *constante*: `123`, `3.14`, `'\0'`, `"salut!\n"` etc.
– *semne de punctuaţie*
operatori: `+` `-` `=` `++` `&&` etc.
separatori: `{` `}` `(` `)` `;` etc.
Spaţiile: necesare doar unde trebuie separaţi doi atomi lexicali alăturaţi
ex. `void main`, nu `voidmain`; nu `floatx=3.14;` nesemnificative în rest.
Indentaţi programele pt. citire uşoară ! (automat în editoarele bune)

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*
(toate informaţiile necesare pentru a-l folosi)
– 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

Declaraţii: forma generală

Întâlnite până acum: `float x;` `int a, b = 1;` `char t[20];`
Dar se pot declara deodată şi mai multe obiecte cu acelaşi tip de bază:
Ex. `int i = 1, n, tab[20], f(double, int);`
declară un întreg iniţializat cu 1, alt intreg neiniţializat, un tablou de 20 de întregi, şi o funcţie întreagă cu doi parametri (double şi 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 tip lista-decl-init* ;
lista-decl-init ::= *declarator-init* | *lista-decl-init* , *declarator-init*
declarator-init ::= *declarator*
| *declarator* = *iniţializator*
declarator ::= *identificator*
| *declarator* [*expresie*] pt. tablouri
| *declarator* (*parametri*) pt. funcţii
| * *declarator* pt. pointeri

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); */
}
```


Variabile globale și locale

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

Variabile globale
= o variabilă declarată in 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 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`

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 × ... × 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 */`

Definiţii de constante şi tipuri

Definiţii de tip: `typedef` *declaraţie*

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

– sintaxa: ca şi declaraţia de variabile, prefixată cu `typedef`

– dacă în *declaraţie*, identificatorul ar fi o *variabilă* de un anumit tip, atunci `typedef` *declaraţie* defineşte 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 */
```

Declaraţii 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)

Declaraţii şi definiţii de funcţii

Declaraţia: prototipul (antetul) funcţiei: 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 *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

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);`

Funcţii standard din `math.h` (cont.)

Funcţii de exponenţiere şi 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 şi 2

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

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

Funcţii trigonometrice şi 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