Declaraţii

18 octombrie 2005

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*.

Reprezentam sintaxa limbajului ca *gramatica* in BNF (Backus-Naur-Form)

`::=` reprezinta definitii `|` reprezinta alternative

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

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

O *declarație* specifică interpretarea și atributele unui *identificator*
pt. variabilă: numele, tipul; pt. funcție: numele, tipul, 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

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* — un declarator

| *lista-declaratori-init* , *declarator-init* — sau mai multi

declarator-init ::= *declarator* — neinitializat

| *declarator* = *iniţializator* — initializat

declarator ::= *identificator* — variabila simpla

| *declarator* [*expresie*] — tablou

| *declarator* (*parametri*) — funcţie

| * *declarator* — pointer

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

Variabile globale si locale

Privire de ansamblu (detalii in cursul viitor):

Variabile locale

– declarate in interiorul unui bloc `{ }` (de ex. in functii)
– vizibile doar in interiorul blocului respectiv
– memorate doar pe durata executiei blocului (exceptie: `static`)
– nu sunt initializate automat

Variabile globale

– declarate in exteriorul functiilor
– vizibile in intreg programul (eventual la nivel de fisier)
– memorate pe toata durata programului
– initializate la inceputul executiei (explicit / implicit pe zero)

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, numele de tablou reprezintă *adresa* acestuia, si nu grupul de elemente din tablou (nu se fac atribuiri de tablouri în bloc, etc.)

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

ANSI C permite doar definitii cu dimensiuni *constante* (pozitive)

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

```
void f(int n) { char s[n + 3]; /* n e cunoscut la apel */ }
```

NU se pot defini tablouri fără a le preciza dimensiunea !

`int a[];` e o *declaratie*, nu o *definitie*, tipul tablou e incomplet.

Daca nu apare nici o definitie cu dimensiune, aceasta se considera 1 !!

Siruri de caractere

= caz particular de tablouri de `char`

Pentru a reprezenta un sir, trebuie stabilita prin conventie *lungimea* lui

– in Pascal, prin octet care memoreaza lungimea (in tipul `string`)

– in C, printr-un delimitator de sfarsit: caracterul special `'\0'` (nul)

⇒ e suficient sa reprezentam sirul prin *adresa* sa de inceput

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

Atenţie:

– o constanta sir (`"test"`) are in program *tipul* adresa de caracter

– sirul `"a"` (o adresa) si caracterul `'a'` (un intreg) au tipuri diferite !

– nu exista operatori predefiniti pe siruri (comparatie, concatenare, ...)

– toate funcţiile care lucrează cu şiruri presupun terminarea lor in `'\0'`

dar numai pt. reprezentarea in memorie! (nu si la citire/in texte/fisiere)

Inițializarea

– variabilele cu durată de memorare *statică* (ex. globale) sunt inițializate înainte de execuție: implicit cu zero; explicit doar cu constante

– variabilele cu durată *automată* (ex. locale) 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* , ... se poate termina cu ...

param-list ::= *param-decl* | *param-list* , *param-decl* unul sau mai multi

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)

Apelurile de functie

– argumentele pot fi orice *expresii*: `f(2*x+1)`, incl. variabile/constante

– se evalueaza *expresiile* date ca argumente şi se atribuie parametrilor formali *valorile* obtinute (cu eventuale conversii ca la atribuire)

⇒ *Transferul parametrilor* în C se face *prin valoare*

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

In executia corpului functiei NU exista nici o legatura intre un parametru formal si eventuala variabila a carei *valoare* a fost transmisa parametru

Ex. daca apelam `void f(int n)` cu `f(x)`, cand in `f` se executa `n = n+1`

NU se modifica `x` !!! NU se atribuie inapoi `x = n` la retur !!!

NU sunt specificate de standard:

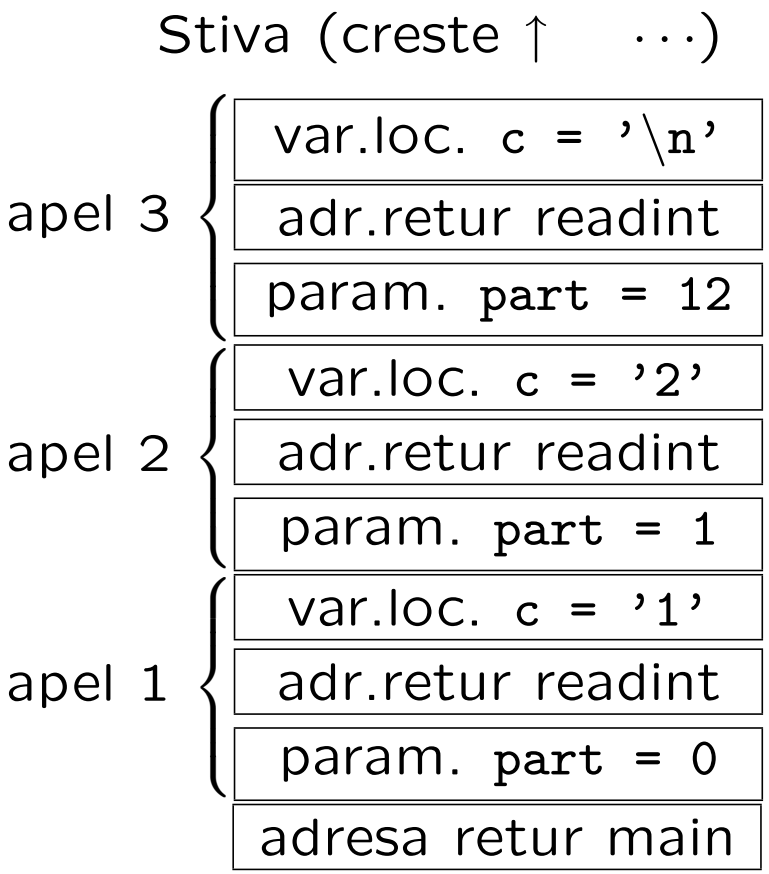
– ordinea de evaluarea a argumentelor (si a adresei functiei apelate)

– dispunerea în memorie a argumentelor (pe stivă).

In mod tipic: ordine inversa pentru ambele (primul argument sus).

Apelul de functie si stiva

```
int readint(int part)
{
  int c = getchar();
  return isdigit(c) ?
    readint(10*part+c-'0') : part;
}
int main(void)
{
  printf("%d\n", readint(0));
  return 0;
}
// rulam cu 12\n de la intrare
```



Tablouri și apeluri de funcții

O funcție nu poate returna o valoare de tip tablou.
Un nume de tablou reprezinta de fapt adresa tabloului (v. ulterior) ⇒
O functie cu parametru tablou primeste adresa, nu blocul de elemente.

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

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

⇒ trebuie cunoscute dimensiunile D2 ... Dn, dar nu și D1

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]);
```