

Declarații

18 octombrie 2005

Programarea calculatoarelor 2. Curs 3a

Marius Minea

Declarații

Structura programului: declarații și definiții

2

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

Reprezentăm sintaxa limbajului ca *gramatică* în BNF (Backus-Naur-Form)

`::=` reprezintă definiții | reprezintă 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

Programarea calculatoarelor 2. Curs 3a

Marius Minea

Declarații

Sintaxa declarațiilor

3

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 ::= specificatoriopt tip lista-declaratori-init ;`

`lista-declaratori-init ::= declarator-init` un declarator

| `lista-declaratori-init , declarator-init` sau mai multi

`declarator-init ::= declarator` neinițializat

| `declarator = inițializator` inițializat

`declarator ::= identificator` variabila simplă

| `declarator [expresie]` tablou

| `declarator (parametri)` funcție

| `* declarator` pointer

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

Programarea calculatoarelor 2. Curs 3a

Marius Minea

Declarații

4

Variabile globale și locale

Privire de ansamblu (detalii în cursul viitor):

Variabile locale

- declarate în interiorul unui bloc `{ }` (de ex. în funcții)
- vizibile doar în interiorul blocului respectiv
- memorate doar pe durata execuției blocului (excepție: `static`)
- nu sunt inițializate automat

Variabile globale

- declarate în exteriorul funcțiilor
- vizibile în întreg programul (eventual la nivel de fișier)
- memorate pe toată durata programului
- inițializate la începutul execuției (explicit / implicit pe zero)

Programarea calculatoarelor 2. Curs 3a

Marius Minea

Declarații

5

Declarații de tablouri

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

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

declară un tablou n-dimensional de $D1 \times \dots \times Dn$ elemente de *tip* de fapt: tablou de $D1$ elem. care sunt tablouri de ... Dn elem. de *tip*

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

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

ANSI C permite doar definiții 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 *declarație*, nu o *definiție*, tipul tablou e incomplet.

Dacă nu apare nici o definiție cu dimensiune, aceasta se considera 1 !!

Programarea calculatoarelor 2. Curs 3a

Marius Minea

Declarații

6

Siruri de caractere

= caz particular de tablouri de `char`

Pentru a reprezenta un șir, trebuie stabilită prin convenție *lungimea* lui

– în Pascal, prin octet care memorează lungimea (în tipul `string`)

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

⇒ e suficient să reprezentăm șirul prin *adresa* sa de început

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

Atenție:

– o constantă șir (`"test"`) are în program *tipul* adresa de caracter

– șirul `"a"` (o adresă) și caracterul `'a'` (un întreg) au tipuri diferite !

– nu există operatori predefiniți pe șiruri (comparație, concatenare, ...)

– toate funcțiile care lucrează cu șiruri presupun terminarea lor în `'\0'`

dar numai pt. reprezentarea în memorie! (nu și la citire/in texte/fișiere)

Programarea calculatoarelor 2. Curs 3a

Marius Minea

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 */
```

Programarea calculatoarelor 2. Curs 3a Marius Minea

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 , ... 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 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)
```

Programarea calculatoarelor 2. Curs 3a Marius Minea

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)

Apelurile de funcție

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

– se evaluează *expresiile* date ca argumente și se atribuie parametrilor formali *valorile* obținute (cu eventuale conversii ca la atribuire)

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

– se execută corpul funcției; se revine la instrucțiunea de după apel

In execuția corpului funcției NU există nici o legătură între un parametru formal și eventuala variabilă a cărei *valoare* a fost transmisă parametrului.

Ex. dacă apelăm `void f(int n)` cu `f(x)`, când în `f` se execută `n = n+1` NU se modifică `x` !!! NU se atribuie înapoi `x = n` la retur !!!

NU sunt specificate de standard:

– ordinea de evaluare a argumentelor (și a adresei funcției apelate)

– dispunerea în memorie a argumentelor (pe stivă).

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

Programarea calculatoarelor 2. Curs 3a Marius Minea

Apelul de funcție și 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
```

Stiva (crește ↑ ...)

apel 3	var.loc. c = 'n'
	adr.retur readint
	param. part = 12
apel 2	var.loc. c = '2'
	adr.retur readint
	param. part = 1
apel 1	var.loc. c = '1'
	adr.retur readint
	param. part = 0
	adresa retur main

Tablouri și apeluri de funcții

O funcție nu poate returna o valoare de tip tablou.

Un nume de tablou reprezintă de fapt adresa tabloului (v. ulterior) ⇒

O funcție cu parametru tablou primește 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:

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

⇒ 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]);
```