Recapitulare. Erori frecvente. Exerciţii

18 ianuarie 2004

Declararea, definirea și apelul funcţiilor

O funcţie se *definește* cu *antetul* urmat de `corpul` său:

tip_rezultat nume_funcţie (*lista_parametri*)

```
{ /* declaraţii şi instrucţiuni din corpul funcţiei */ }
```

unde lista de parametri e fie `void` (dacă nu sunt parametri), fie

$tip\_par_1\ nume\_par_1$, $tip\_par_2\ nume\_par_2$, $tip\_par_n\ nume\_par_n$

O *declarație* de funcţie e doar *antetul* urmat de `;`

– dacă vrem să folosim (apelăm) funcţia fără ca ea să fie încă definită

– dacă e definită altundeva (ex. bibliotecă) – declaraţii din fișiere `.h`

```
int abs(int n);   int getchar(void);   void exit(int status);
double pow(double x, double y);   int main(int argc, char *argv[]);
```

Declaraţia începe cu `void` doar dacă funcţia nu returnează *nimic*!

O funcţie se apelează similar ca și în matematică: *nume*(*argumente*)

Argumentele pot fi *expresii arbitrare* (incl. variabile sau constante)

```
printf("%d",abs(x+2)); c = getchar(); exit(1); z = 2 + pow(x-3, 5);
```

Parametrii de funcţii

În limbajul C, *parametrii se transmit prin valoare*. Adică:
– înainte de apel se calculează valoarea expresilor date ca argument
– la începutul execuţiei funcţiei, fiecare parametru din antet primeşte valoarea argumentului corespunzător
– parametrii din declaraţia funcţiei se comportă ca şi variabile locale: nu sunt vizibili după ieşirea din funcţie; valorile *nu se transmit înapoi*

```
void f(int x) { x = 5; } void main(void) { int y = 3; f(y); }
```

In interiorul lui `f`, `x` se modifică din 3 în 5. Dar `y` nu se schimbă !!!
Dacă apelăm `f(4)`, nu putem să-l facem pe 4 egal cu 5 !!!
Dacă apelăm `f(y*y + 2)`, programul NU rezolvă ecuaţia $y^2 + 2 = 5$!!!

Pentru a returna o valoare folosim instrucţiunea **`return`** *expresie* **`;`**

Parametrii de funcţii (cont.)

La începutul funcţiei, fiecare parametru are o valoare *cunoscută*: valoarea expresiei transmisă ca argument. E greşit să scriem:

```
void f(int x) {
  printf("Introduceţi valoarea lui x"); // x e deja cunoscut !!!
  scanf("%d", &x); // dacă x e dat, de ce vrem să citim altul ???
}
```

Când apelăm `f(4)` înseamnă că vrem să lucrăm cu 4, nu să-l citim!!!
Dacă funcţia citeşte `x`, nu-l are parametru (nu poate să-l modifice!)
NU: `void citeste(int x);` De ce îi *dăm* funcţiei valoarea lui `x` ca argument când noi *cerem* o valoare ? Ce înseamnă `citeşte(3)` ??

Funcţia trebuie să ia ca parametru o *adresă* de întreg, pentru a scrie rezultatul la acea adresă: `void citeşte(int *px) { scanf("%d", px); }` sau *returnează* valoarea (fără parametri, o variabilă locală pt. rezultat)
`int citeşte(void) { int x; scanf("%d", &x); return x; }`

Pointeri

O declaraţie de pointer: `tip *ptr;` spune: *voi avea* un obiect (sau tablou) de tipul `tip`, dar încă *nu există*, *n-am memorie* pentru el

⇒ nu-l putem folosi înainte de a-i atribui o zonă de memorie !
(adresa unei variabile existente, sau zonă alocată dinamic)

– *Alocăm static*: când cunoaştem dinainte dimensiunea.

`char s[80];` NU ne complicăm: `char *s; s = malloc(80); if (!s) ...`

– *Folosim* `malloc`: când ştim dimensiunea în momentul apelului.

```
printf("Câte numere"); scanf("%d", &n"); tab=malloc(n*sizeof(int));
l=strlen(s); if (p=malloc(l+1)) strcpy(p, s); else ...
```

– *Folosim* realloc: când iniţial nu am alocat cât trebuie *întotdeauna* folosim pointerul *nou* returnat (poate muta memoria)

Pointeri si tablouri

Numele unui tablou e *adresa* sa de început (o *constantă* !)

⇒ numele unui tablou (incl. şir de caractere) e un *pointer* (constant)

⇒ `tablou[indice]` sau `pointer[indice]` e acelaşi lucru

⇒ `char a[10], b[10]; a = b;` NU copiază tablouri, ci atribuie adrese !
(şi dă eroare de compilare, pentru că `a` e constantă !)

`s1==s2` compară pointerii (se suprapun?), nu conţinutul: `strcmp(s1, s2)`

⇒ NU are sens să scriem `void f(char s[20])`.
scriem: `void f(char tab[])` sau `void f(char *tab)`
(NU se transmit 20 de caractere, se transmite adresa tabloului)

Tablouri de şiruri de caractere:

`char tab[NUM][LEN];` (dacă cunoaştem lungimea maximă a şirului)

`char *tab[NUM];` fiecare element (adresă) trebuie atribuit (*alocat)* !

Pointeri ca parametri şi rezultate

Orice parametru transmis trebuie să aibă o valoare validă, utilizabilă !
⇒ un pointer transmis trebuie să indice o zonă de memorie validă!
– zona respectivă e folosită la citire sau scriere, depinzând de funcţie
NU: `char *p; strcpy(p, "un sir");` `p` neiniţializat/nealocat !
NU: `char **endptr; l=strtol(sir, endptr, 10);` `endptr` e nealocat!
DA: `char *endptr; l=strtol(sir, &endptr, 10);` scrie valoare la `&endptr`

O funcţie nu poate întoarce adresa unei variabile *locale* (ex. tablou).
– e alocată pe stivă ⇒ va *dispare* odată cu ieşirea din corpul funcţiei
⇒ un pointer returnat de o funcţie provine din a) un parametru;
b) o variabilă globală (problematic: suprascriere); c) alocare dinamică

Un pointer returnat de o funcţie trebuie să fie *valid* sau `NULL`.