

## Declararea, definirea și apelul funcțiilor

O funcție se **definește** cu *antetul* urmat de corpul său:

```
tip_rezultat nume_funcție ( lista_parametri )
{ /* declarații și instrucțiuni din corpul funcției */
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 funcție e doar *antetul* urmat de ;

– dacă vrem să folosim (apelăm) funcția fără ca ea să fie încă definită – dacă e definită altundeva (ex. bibliotecă) – declarații 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[]);
Declarația începe cu void doar dacă funcția nu returnează nimic!
O funcție 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);
```

Utilizarea și programarea calculatoarelor. Curs 14

Marius Minea

## Parametrii de funcții

În limbajul C, **parametrii se transmit prin valoare**. Adică:

– Înainte de apel se calculează valoarea expresiilor date ca argument  
– la începutul execuției funcției, fiecare parametru din antet primește valoarea argumentului corespunzător

– parametrii din declarația funcției se comportă ca și variabile locale: nu sunt vizibili după ieșirea din funcție; valorile **nu se transmit înapoi**

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

În 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 ecuația  $y^2 + 2 = 5$  !!!

Pentru a returna o valoare folosim instrucțiunea **return expresie** ;

Utilizarea și programarea calculatoarelor. Curs 14

Marius Minea

## Parametrii de funcții (cont.)

La începutul funcției, fiecare parametru are o valoare **cunoscută**:

valoarea expresiei transmise ca argument. E greșit să scriem:

```
void f(int x) {
printf("Introduceți 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 funcția citește x, nu-l are parametru (nu poate să-l modifice!)

NU: void citește(int x); De ce îi **dăm** funcției valoarea lui x ca argument când noi **cerem** o valoare ? Ce înseamnă citește(3) ??

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

```
int citește(void) { int x; scanf("%d", &x); return x; }
```

Utilizarea și programarea calculatoarelor. Curs 14

Marius Minea

## Pointeri

O declarație 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 cunoaștem dinainte dimensiunea.

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

– **Folosim malloc**: când știm 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 inițial nu am alocat cât trebuie

**întotdeauna** folosim pointerul **nou** returnat (poate muta memoria)

Utilizarea și programarea calculatoarelor. Curs 14

Marius Minea

## Pointeri si tablouri

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

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

⇒ tablou[indice] sau pointer[indice] e același lucru

⇒ char a[10], b[10]; a = b; NU copiază tablouri, ci atribuie adrese !

(și dă eroare de compilare, pentru că a e constantă !)

s1==s2 compară pointerii (se suprapun?), nu conținutul: 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 șiruri de caractere:**

```
char tab[NUM][LEN]; (dacă cunoaștem lungimea maximă a șirului)
```

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

Utilizarea și programarea calculatoarelor. Curs 14

Marius Minea

## Pointeri ca parametri și 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 funcție

NU: char \*p; strcpy(p, "un sir"); p neinițializat/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 funcție nu poate întoarce adresa unei variabile **locale** (ex. tablou).

– e alocată pe stivă ⇒ va **dispare** odată cu ieșirea din corpul funcției

⇒ un pointer returnat de o funcție provine din a) un parametru;

b) o variabilă globală (problematic: suprascriere); c) alocare dinamică

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

Utilizarea și programarea calculatoarelor. Curs 14

Marius Minea