# Programare modulară. Funcţii în limbajul C. Dezvoltarea algoritmilor folosind funcţii

**Responsabili:**

- Darius Neațu (CA 2019-2020) [mailto:neatudarius@gmail.com]
- Dorinel Filip (CA 2019-2020) [mailto:ion_dorinel.filip@cti.pub.ro]
- Andrei Pârvu [mailto:andrei.parvu@cti.pub.ro]

## Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil:

- să declare şi să definească o funcţie în limbajul C
- să apeleze funcţii definite în acelaşi fişier sursă, cât şi funcţii din alte fişiere sursă sau biblioteci
- să distingă între parametrii formali şi cei efectivi, între cei transmişi prin valoare şi cei transmişi prin adresa de memorie
- să explice rolul funcţiei `main()` într-un program
- să folosească clasele de stocare în declaraţiile unor funcţii

## Noţiuni teoretice

Funcţiile împart taskuri complexe în bucăţi mici mai uşor de înţeles şi de programat. Acestea pot fi refolosite cu alte ocazii, în loc să fie rescrise de la zero. De asemenea, funcţiile sunt utile pentru a ascunde detalii de funcţionare ale anumitor părţi ale programului, ajutând la modul de lucru al acestuia. Utilizând funcţii, care reprezintă unitatea fundamentală de execuţie a programelor C, se obţine o divizare logică a programelor mari şi complexe.

Împărţirea programelor în funcţii este arbitrară şi depinde de modul de gândire a celui care le creează. De obicei, funcţiile cuprind o serie de instrucţiuni care efectuează un calcul, realizează o acţiune, implementează un algoritm, etc. Crearea funcţiilor trebuie să se bazeze pe următoarele principii: `claritate`, `lizibilitate`, `uşurinţă în întreţinere`, `reutilizabilitate`.

## Definirea şi apelul unei funcţii în C

Caracteristicile definitorii ale unei funcţii în C sunt: numele, parametrii de apel şi valorea returnată. Sintaxa standard de **declarare** a unei funcţii este:

```
tip_returnat nume_functie (tip_param1 nume_param1 , tip_param2 nume_param2, ...);
```

Această declarare poartă numele de **antetul funcţiei** (**function signature** sau simplu **signature**). Lista de parametri **poate** lipsi.

Odată declarată, o funcţie trebuie **definită**, în sensul că trebuie expandat corpul acesteia cu instrucţiunile pe care trebuie să le execute.

Definirea unei funcţii are forma:

```
tip_returnat nume_functie(tip_param1 nume_param1, tip_param2 nume_param2, ...) {
  declaratii de variabile si instructiuni;

  return expresie;
}
```

Limbajul C permite separarea declaraţiei unei funcţii de definiţia acesteia (codul care o implementează). Pentru ca funcţia să poată fi folosită, este obligatorie doar declararea acesteia înainte de codul care o apelează. Definiţia poate apărea mai departe în fişierul sursă, sau chiar într-un alt fişier sursă sau bibliotecă.

Diferite părţi din definirea unei funcţii pot lipsi. Astfel, o funcţie minimală este:

```
dummy() {}
```

Funcţia de mai sus nu face absolut nimic, nu întoarce nici o valoare şi nu primeşte nici un argument, însă din punct de vedere al limbajului C este perfect validă.

Tipul returnat de o funcţie poate fi orice tip **standard** sau **definit** de utilizator (**struct**-uri - acoperite într-un laborator următor), inclusiv tipul **void** (care înseamnă că funcția nu returnează nimic).

Orice funcţie care întoare un rezultat trebuie să conţină instrucţiunea:

```
return expression;
```

Expresia este evaluată şi convertită la tipul de date care trebuie returnat de funcţie. Această instrucţiune termină şi execuţia funcţiei, **indiferent** dacă după aceasta mai urmează sau nu alte instrucţiuni. Dacă este cazul, se pot folosi mai multe instrucţiuni **return** pentru a determina mai multe puncte de ieşire din funcţie, în raport cu evoluţia funcţiei.

Exemplu:

declarare.c

```
int min(int x, int y);
```

definire.c

```
int min(int x, int y) {
  if (x < y) {
    return x;
  }

  return y;
}
```

Apelul unei funcţii se face specificând **parametrii efectivi** (parametrii care apar în declararea funcţiei se numesc **parametri formali**).

```
int main() {
  int a, b, minimum;
  //..........
  x = 2;
  y = 5;
  minimum = min(x, 4);
  printf("Minimul dintre %d si 4 este: %d", x, minimum);
  printf("Minimul dintre %d si %d este: %d", x, y, min(x, y));
}
```

# Transmiterea parametrilor

Apelul unei funcţii se face specificând parametrii care se transmit acesteia. În limbajul C, dar şi în alte limbaje de programare există **2 moduri de transmitere a parametrilor**. Deoarece nu avem încă cunoștințele necesare pentru a înțelege ambele moduri, astăzi vom studia doar unul, urmând ca în laboratorul 8 să revenim și să îl explicăm și pe al doilea.

## Transmiterea parametrilor prin valoare

Funcţia va lucra cu **o copie** a variabilei pe care a primit-o şi **orice** modificare din cadrul funcţiei va opera asupra aceste copii. La sfârşitul execuţiei funcţiei, copia va fi **distrusă** şi astfel se va pierde orice modificare efectuată.

Pentru a nu pierde modificările făcute se foloseşte instrucţiunea **return**, care poate întoarce, la terminarea funcţiei, noua valoare a variabilei. Problema apare în cazul în care funcţia modifică mai multe variabile şi se doreşte ca rezultatul lor să fie disponibil şi la terminarea execuţiei funcţiei.

Exemplu de transmitere a parametrilor prin valoare:

```
min(x, 4);  // se face o copie lui x
```

Până acum aţi folosit în programele voastre funcţii care trimit valorile atât prin valoare (de exemplu `printf()`) cât şi prin intermediul adresei de memorie (de exemplu `scanf()`). Mecanismul de transfer al valorilor prin intermediul adresei de memorie unde sunt stocate va fi complet „elucidat” în laboratorul de pointeri.

## Funcţii recursive

O funcţie poate să apeleze la rândul ei alte funcţii. Dacă o funcţie se apelează pe sine **însăşi**, atunci funcţia este **recursivă**. Pentru a evita un număr infinit de apeluri recursive, trebuie ca funcţia să includă în corpul ei o **condiţie de oprire**, astfel ca, la un moment dat, recurenţa să se oprească şi să se revină succesiv din apeluri.

Condiţia trebuie să fie una generică, şi să oprească recurenţa în orice situaţie. Această condiţie se referă în general a parametrii de intrare, pentru care la un anumit moment, răspunsul poate fi returnat direct, fără a mai fi necesar un apel recursiv suplimentar.

`Exemplu`: Calculul recursiv al factorialului

```
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

sau, într-o formă mai compactă:

```
int fact(int n) {
  return (n >= 1) ? n * fact(n - 1) : 1;
}
```

Întotdeauna trebuie avut grijă în lucrul cu funcţii recursive deoarece, la fiecare apel recursiv, contextul este salvat pe stivă pentru a putea fi refăcut la revenirea din recursivitate. În acest fel, în funcţie de numărul apelurilor recursive şi de dimensiunea contextului (variabile, descriptori de fişier, etc.) stiva se poate umple foarte rapid, generând o eroare de tip stack overflow [http://en.wikipedia.org/wiki/Stack_overflow] (vezi şi Infinite recursion pe Wikipedia [http://en.wikipedia.org/wiki/Infinite_recursion]).

## Funcţia main

Orice program C conţine cel puţin o funcţie, şi anume cea principală, numită `main()`. Aceasta are un format special de definire:

```
int main(int argc, char *argv[])
{
    // some code
    return 0;
}
```

Primul parametru, `argc`, reprezintă numărul de argumente primite de către program la linia de comandă, incluzând numele cu care a fost apelat programul. Al doilea parametru, `argv`, este un pointer către conţinutul listei de parametri al căror număr este dat de `argc`. Lucrul cu parametrii liniei de comandă va fi reluat într-un laborator viitor.

Atunci când nu este necesară procesarea parametrilor de la linia de comandă, se poate folosi forma prescurtată a definiţiei funcţiei `main`, şi anume:

```
int main(void)
{
    // some code
    return 0;
}
```

În ambele cazuri, standardul impune ca `main` să întoarcă o valoare de tip întreg, care să reprezinte codul execuţiei programului şi care va fi pasată înapoi sistemului de operare, la încheierea execuţiei programului. Astfel, instrucţiunea `return` în funcţia `main` va însemna şi terminarea execuţiei programului.

În mod normal, orice program care se execută corect va întoarce 0, şi o valoare diferită de 0 în cazul în care apar erori. Aceste coduri ar trebui documentate pentru ca apelantul programului să ştie cum să adreseze eroarea respectivă.

## Tipul de date void

Tipul de date `void` are mai multe întrebuinţări.

Atunci când este folosit ca tip returnat de o funcţie, specifică faptul că funcţia nu întoarce nici o valoare. Exemplu:

```
void print_nr(int number) {
  printf("Numarul este %d", number);
}
```

Atunci când este folosit în declaraţia unei funcţii, `void` semnifică faptul că funcţia nu primeşte nici un parametru. Exemplu:

```
int init(void) {
  return 1;
}
```

Această declaraţie nu este similară cu următorul caz:

```
int init() {
  return 1;
}
```

În cel de-al doilea caz, compilatorul nu verifică dacă funcţia este într-adevăr apelată fără nici un parametru. Apelul celei de-a doua funcţii cu un număr arbitrar de parametri nu va produce nici o eroare, în schimb apelul primei funcţii cu un număr de parametri diferit de zero va produce o eroare de tipul:

`too many arguments to function`.

## Clase de stocare. Fişiere antet vs. biblioteci

Această secţiune este importantă pentru înţelegerea modului de lucru cu mai multe fişiere sursă şi cu bibliotecile oferite de GCC. Deşi în continuare sunt discutate în contextul funcţiilor, lucrurile se comportă aproximativ la fel şi în cazul variabilelor globale (a căror utilizare este, oricum, descurajată).

După cum se ştie, într-un fişier sursă (.c) pot fi definite un număr oarecare de funcţii. În momentul în care programul este compilat, din fiecare fişier sursă se generează un fişier obiect (.o), care conţine codul compilat al funcţiilor respective. Aceste funcţii pot apela la rândul lor alte funcţii, care pot fi definite în acelaşi fişier sursă, sau în alt fişier sursă. În orice caz, compilatorul nu are nevoie să ştie care este definiţia funcţiilor apelate, ci numai semnătura acestora (cu alte cuvinte, `declaraţia` lor), pentru a şti cum să realizeze instrucţiunile de apel din fişierul obiect. Acest lucru explică de ce, pentru a putea folosi o funcţie, trebuie `declarată` înaintea codului în care este folosită.

`Fişierele antet` conţin o colecţie de declaraţii de funcţii, grupate după funcţionalitatea pe care acestea o oferă. Atunci când includem un fişier antet (.h) într-un fişier sursă (.c), compilatorul va cunoaşte toate semnăturile funcţiilor de care are nevoie, şi va fi în stare să genereze codul obiect pentru fiecare fişier sursă în parte. (NOTĂ: Astfel nu are sens includerea unui fişier .c în alt fişier .c; se vor genera două fişiere obiect care vor conţine definiţii comune, şi astfel va apărea un conflict de nume la editarea legăturilor).

Cu toate acestea, pentru a realiza un fişier executabil, trebuie ca fiecare funcţie să fie `definită`. Acest lucru este realizat de către editorul de legături; cu alte cuvinte, fiecare funcţie folosită în program trebuie să fie conţinută în fişierul executabil. Acesta caută în fişierele obiect ale programului definiţiile funcţiilor de care are nevoie fiecare funcţie care le apelează, şi construieşte un singur fişier executabil care conţine toate aceste informaţii. `Bibliotecile` sunt fişiere obiect speciale, al căror unic scop este să conţină definiţiile funcţiilor oferite de către compilator, pentru a fi integrate în executabil de către editorul de legături.

`Clasele de stocare` intervin în acest pas al editării de legături. O clasă de stocare aplicată unei funcţii indică dacă funcţia respectivă poate fi folosită şi de către alte fişiere obiect (adică este `externă`), sau numai în cadrul fişierului obiect generat din fişierul sursă în care este definită (în acest caz funcţia este `statică`). Dacă nu este specificată nici o clasă de stocare, o funcţie este implicit externă.

Cuvintele cheie `extern` şi `static`, puse în faţa definiţiei funcţiei, îi specifică clasa de stocare. De exemplu, pentru a defini o funcţie internă, se poate scrie:

```
static int compute_internally(int, int);
```

Funcţia `compute_internally` nu va putea fi folosită decât de către funcţiile definite în acelaşi fişier sursă şi nu va fi vizibilă de către alte fişiere sursă, în momentul editării legăturilor.

## Exerciții Laborator CB/CD

1. Primul exercitiu presupune modificarea/adaugarea de instructiuni unui cod existent pentru a realiza anumite lucruri. In momentul actual programul afiseaza suma cifrelor unui numar.
    - Nu uitati ca trebuie sa utilizam un coding style [http://ocw.cs.pub.ro/courses/programare/coding-style] adecvat atunci cand scriem sursele.

ex1.c

```
#include <stdio.h>

int sum_recursive(int n)
{
    if (n == 0) {
        return 0;
    }

    return n % 10 + sum_recursive(n / 10);
}


int main(void)
{
    int nr;
```

```
    scanf("%d", &nr);

    printf("%d\n", sum_recursive(nr));

    return 0;
}
```

Cerinte:

- Scrieti o functie care realizeaza tot suma cifrelor, insa intr-un mod nerecursiv.
- Modificati functia recursiva astfel incat sa realizeze doar suma cifrelor impare.
- Modificati functia recursiva astfel incat la prima cifra impara sa nu mai mearga in recursivitate si sa intoarca suma realizata pana in acel moment.
- Luati urmatoarea arhiva [http://swarm.cs.pub.ro/~gmuraru/lab4-example.tar] si observati cum sunt structurate fisierele. Rulati **make** pentru a crea executabilul si **make clean** pentru stergerea fisierelor generate.

**Următoarele două probleme vă vor fi date de asistent în cadrul laboratorului.**

Checker si teste laborator 4 [https://drive.google.com/file/d/1BH7Z__2eg5g5sIAMm5AETW4ityNexqk6/view?usp=sharing]

Pentru utilizarea checkerului:

- Se va scrie cate un fisier sursa pentru fiecare problema;
- La finalul fiecarui printf utilizat pentru afisarea rezultatului trebuie sa existe un newline;
- Sursa nu trebuie sa contina alte printf-uri in afara de cele care scriu rezultatul asteptat la stdout.
- Se va dezarhiva arhiva specifica exercitiului;
- In directorul curent se afla checkerul, executabilul generat, folderele de input si output specifice problemei;
- Se va rula “bash checker.sh <executabil>” unde <executabil> este numele executabilului generat;

## Exerciţii de Laborator

1. [1p] Analizaţi programul de mai jos. Modificaţi sursa astfel încât programul să funcţioneze corect, fara a utiliza transmitere prin adresă de memorie.

```
#include<stdio.h>


void sum(int a, int b, int s) {
  s = a + b;
}


int main() {
  int s;
  sum(2, 3, s);
  printf("Suma este %d\n", s);


  return 0;
}
```

2. [1.5p] Scrieți o funcție recursivă care să ridice un număr x la o putere dată y pozitivă.

```
power(2, 3); // rezultat 8
power(7, 3); // rezultat 343
```

3. [1.5p] Scrieți o funcție recursivă care să returneze numărul de cifre al unui numar întreg.

```
number_of_digits(34); // rezultat 2
number_of_digits(2533); // rezultat 4
```

4. [2p] Folosindu-vă de funcțiile scrise anterior scrieți o funcție recursivă ce inversează ordinea cifrelor unui numar întreg pozitiv.

```
reverse_number(23); // rezultat 32
reverse_number(3523); // rezultat 3253
```

5. [2p] Pentru un număr dat, determinați cel mai mic număr palindrom mai mare sau egal decât acel numar. Un palindrom este un număr care citit de la stânga la dreapta sau de la dreapta la stânga rezultă același numâr.

```
next_palindrome(120); // rezultat 121
```

6. [2p] De la tastatură se citeşte o listă de numere pozitive. Pentru fiecare element citit se va afişa numărul prim cel mai apropiat de acesta. Dacă există două numere prime la fel de apropiate de elementul listei, se vor afişa amândouă. Dacă numărul este prim, nu se mai afişează nimic. Programul se încheie în momentul în care este citit un număr negativ. De exemplu:

```
27
* 29
13
*
68
* 67
69
* 67 71
-1
```

### Bonus

1. [2p] Folosind declarații si definiri de variabile și funcții, creați două fișiere `f1.c` și `f2.c`, și apleați din funcția `main` din fișierul `f1.c` o funcție `f` definită în `f2.c`, iar în funcția `f` o variabilă `g` definită in fișierul `f1.c`. Compilați fișierele împreună și executați programul rezultat.

```
gcc f1.c f2.c; ./a.out
```

Probleme laborator cu checker [https://drive.google.com/file/d/10O8he377znoQlojR2p6UWKQCIIHv8FSz/view?usp=sharing]

## Referinţe

- C - Using Functions [http://www.tutorialspoint.com/ansi_c/c_using_functions.htm]

- Cheatsheat Functii [https://github.com/cs-pub-ro/ComputerProgramming/blob/master/Laboratories/Lab4/Lab4.pdf]

programare/laboratoare/lab04.txt · Last modified: 2020/11/03 12:34 by oana.balan