1 Introducere în programarea în C

1.1 Funcţii în limbajul C

Calcule şi funcţii La origine, rolul programelor e de a efectua în principal *calcule* matematice. Discutăm de aceea structura programelor făcând o paralelă cu noţiunile din matematică, care stau şi la baza studiului teoretic al limbajelor de programare.

În matematică, noţiunea de calcul e strâns legată de cea de *funcţie*: în calcule se *folosesc* funcţii cunoscute, se *definesc* altele noi şi se aplică într-o anumit *succesiune*. La fel, codul dintr-un fişier de program C e structurat în funcţii, care sunt definite similar celor din matematică.

O funcţie simplă Fie ca exemplu funcţia care ridică la pătrat un număr întreg. Matematic, scriem:

$$sqr : \mathbb{Z} \to \mathbb{Z} \qquad sqr(x) = x \cdot x$$

Această *definiţie de funcţie* are două părţi: prima specifică *numele* funcţiei (sqr), *domeniul său de definiţie* (mulţimea numerelor întregi, $\mathbb{Z}$) şi *domeniul său de valori*, de asemenea $\mathbb{Z}$. A doua parte specifică modul de calcul al valorii funcţiei *sqr(x)* pornind de la valoarea argumentului x.

În limbajul C, *definiţia* aceleiaşi funcţii are aspectul următor:

```
int sqr(int x)
{
  return x * x;
}
```

Primul rând reprezintă *antetul funcţiei*, cu acelaşi rol ca şi declaraţia $sqr : \mathbb{Z} \to \mathbb{Z}$: el specifică numele funcţiei, domeniul de valori (cuvântul `int` dinaintea numelui funcţiei), şi între paranteze `( )` după numele funcţiei, numele parametrilor, precedaţi de tipul lor (un singur parametru `x`, tot întreg). După antet urmează *corpul funcţiei*, scris între acolade `{ }`. În interior, cuvântul `return` indică expresia care dă valoarea funcţiei, folosind parametrul `x`; semnul `*` denotă produsul.

O a doua funcţie Să exprimăm acum ridicarea la pătrat a unui număr real. Matematic, scriem:

$$sqrf : \mathbb{R} \to \mathbb{R} \qquad sqrf(x) = x \cdot x$$

Funcţia $sqrf$ e *diferită* de funcţia sqr scrisă anterior, deoarece are alt domeniu de definiţie şi de valori. Strict vorbind, şi operaţia de înmulţire e diferită, fiind definită pe altă mulţime, chiar dacă se foloseşte aceeaşi notaţie. La fel şi în limbajul C, nu putem folosi funcţia definită anterior pe întregi pentru a calcula pătratul unui număr real, ci trebuie să definim altă funcţie:

```
float sqrf(float x)
{
  return x * x;
}
```

Cuvântul `float` folosit pentru domeniul de definiţie şi de valori indică numerele reale.

1.2 Tipuri şi operatori

Tipuri În termeni de limbaje de programare, spunem că `int` şi `float` denotă *tipuri*. Un *tip* (sau tip de date) desemnează o *mulţime de valori* împreună cu o serie de *operaţii* definite pe aceste valori. Deci nu putem echivala pur şi simplu tipul `int` cu multimea $\mathbb{Z}$ şi tipul `float` cu $\mathbb{R}$. E mai apropiată o analogie între un tip şi o structură algebrică, care de asemenea grupează o mulţime–suport cu un set de operaţii, de exemplu inelul $(\mathbb{Z}, +, \cdot)$ sau corpul $(\mathbb{R}, +, \cdot)$. După cum înmulţirea nu e inversabilă pe $\mathbb{Z}$ dar este pe $\mathbb{R}$, tot aşa există diferenţe între unii operatori pentru întregi şi reali în C.

O altă diferenţă faţă de matematică e că în C, atât întregii şi realii au domeniu de valori finit, şi implicit realii au precizie finită. Detalii, şi ce implică aceasta în calcule vom discuta ulterior.

Constante întregi şi reale Constantele numerice pot avea tip întreg, sau tip real, dacă au punct zecimal. Deci, `2.0` şi `2` reprezintă acelaşi număr, dar au tip diferit. Se pot scrie constante şi în format cu exponent de 10; acestea sunt de tip real, chiar dacă nu au punct zecimal: `5e2` sau `.5E3` sau `500.` reprezintă acelaşi număr. Conform standardului, constantele nu au semn; `-3` sau `+4.1` sunt *expresii* cu operatori unari.

Operatori aritmetici Operatorii aritmetici `+ - * ` şi `/` au aceeaşi semnificaţie, *asociativitate* şi *precedenţă* ca în matematică. Parantezele `( )` se pot folosi pentru a grupa subexpresii în modul dorit. De exemplu, pentru $\frac{x}{m \cdot n}$ trebuie scris `x / (m * n)` şi nu `x / m * n`, care înseamnă $\frac{x}{m} \cdot n$.

Două aspecte diferă între întregi şi reali: împărţirea `/` se efectuează *exact* pentru reali şi *cu rest* pentru întregi, iar pentru operanzi întregi există şi operatorul `%` (modulo) care pentru reali nu e definit.

Astfel, `7.0/2.0` e `3.5`, dar `7/2` dă valoarea `3` ! La împărţirea întregilor cu semn, rezultatul are valoarea absolută ca şi pentru întregi pozitivi, iar semnul e dat de regula uzuală, deci `7/-2` este `-3`. Semnul restului (obţinut cu operatorul `%`) e acelaşi cu semnul deîmpărţitului. Deci `a/b * b + a%b` e egal cu `a` (ecuaţia împărţirii cu rest). Exemple fără semn: `9/5` este `1` şi `9%5` este `4` , iar cu semn: `9/-5` este `-1` şi `9%-5` este `4` ; `-9/5` este `-1` şi `-9%5` este `-4` ; `-9/-5` este `1` şi `-9%-5` este `-4` .

Operatorii `+` şi `-` există şi ca operatori *unari*, cu precedenţa mai mare decât operatorii aritmetici binari, deci putem scrie expresii de forma `-x + -2` sau 2 * +3.

Orice *expresie* are şi ea un *tip*, determinat de tipul operanzilor şi de operatori: astfel, operatiile aritmetice pe întregi produc întregi, iar cele între reali produc reali; conversii de tip discutăm ulterior.

1.3 Terminologie: sintaxă şi semantică

Sintaxă Termenii folosiţi pentru a discuta structura programelor scrise în limbaje de programare sunt similari cu cei folosiţi pentru textele scrise în limbaj natural. Un program e alcătuit dintr-o înşiruire de *elemente lexicale*, cele mai mici unităţi cu înţeles de sine stătător:

- *cuvinte cheie* (`int`, `return`): au înţeles predefinit de standardul limbajului (nu se pot redefini)
- *identificatori*: nume date de programator pentru *funcţii*, *parametri*, etc. Un identificator e o secvenţă de caractere formată din litere mari şi mici, liniuţă de subliniere `_` şi cifre, care nu începe cu o cifră şi nu este un cuvânt cheie. Exemple: `sqr`, `x`, `main`, `printf`, `N_1`, `_exit` . Numele funcţiilor de bibliotecă sunt identificatori şi ar putea fi refolosite în alt scop, dar generând confuzii.
- *constante* (întregi: `-12`, reali: `3.14`, vom discuta ulterior caractere şi şiruri)
- *semne de punctuaţie*: operatori, separatori (ex. `,` şi `;`), parantezele `( )`, acoladele `{ }` etc.

În C se face distincţie între litere mari şi litere mici. Astfel, `nr`, `NR`, `Nr` şi `nR` sunt identificatori diferiţi! Toate cuvintele cheie sunt scrise cu minuscule, ca şi marea majoritate a funcţiilor standard.

Într-un program, elementele lexicale sunt grupate în *construcţii de limbaj* mai complexe, după anumite *reguli* de scriere care formează *sintaxa* limbajului. Am dat ca exemplu *definiţia* unei funcţii, formată din *antet* şi *corp*. *Antetul* unei funcţii (în general cu mai mulţi parametri) are sintaxa (forma):
tip-rezultat nume-funcţie `(` *tip-parametru nume-parametru,* $\cdots$ *, tip-parametru nume-parametru* `)`
Corpul unei funcţii conţine între acolade *instrucţiuni*: elemente de limbaj ce specifică *acţiuni* care să fie executate de program. Am folosit până acum doar instrucţiunea `return`, cu sintaxa:

`return` *expresie* `;`

Expresiile întâlnite până acum sunt alcătuite sintactic după reguli similare celor din matematică. Toţi operatorii sunt indicaţi explicit: în loc de $2n$ sau $n(n+1)$ trebuie scris `2*n` şi respectiv `n*(n+1)`

Semantică Sintaxa limbajului descrie doar felul cum arată un program (sau fragment) corect, dar nu şi ce înseamnă. *Semantica* unui limbaj (de programare, a unei constructii de limbaj, sau a unui a program) descrie înţelesul său (efectul pe care îl are). Instrucţiunea `return` are ca efect *evaluarea* (calculul valorii) expresiei precizate, care e returnată ca valoare a funcţiei pentru argumentele primite. Execuţia funcţiei se încheie şi se revine în locul de unde a fost apelată (aspecte detaliate ulterior).

Punerea în pagină a programelor Într-un program C, *spaţiile albe* (fără reprezentare pe ecran, cum ar fi caracterul spaţiu, cel de tabulare, cel de linie nouă) sunt *separatori* între elemente lexicale. Ele nu sunt necesare decât dacă prin omiterea lor, din două elemente lexicale adiacente s-ar crea alt element lexical valid, schimbând înţelesul programului. Astfel, funcţia `sqr` definită anterior poate fi scrisă `int sqr(int x){return x*x;}` prin eliminarea spaţiilor. Cele 3 spaţii rămase sunt necesare ca separatori, în acest caz, între cuvinte cheie şi identificatori. Spaţiul nu e necesar în expresia `3+-5` dar e necesar în `3- -5` pentru că `--` ar avea alt înţeles (decrementare) în limbajul C.

Se recomandă respectarea unor convenţii pentru mai buna lizibilitate a programelor C, ca de exemplu alinierea una sub alta a instrucţiunilor succesive, şi *indentarea* (de regulă cu două spaţii la dreapta) a unei secvenţe de instrucţiuni faţă de acoladele `{ }` inconjurătoare.

1.4 Apelarea funcţiilor

Definim o funcţie pentru calculul discriminantului unei ecuaţii de gradul II, $a \cdot x^2 + b \cdot x + c = 0$. Ea are trei parametri, separaţi prin virgulă, cu tipul indicat pentru fiecare parametru în parte.

```
float discrim(float a, float b, float c)
{
  return sqrf(b) - 4 * a * c;
}
```

Expresia apel În expresia de calcul a valorii funcţiei apare *expresia* `sqrf(b)` : e *apelată* (folosită) funcţia `sqrf` pentru a calcula pătratul parametrului `b`. Sintaxa *apelului de funcţie* este aceeaşi ca şi în matematică: *funcţie* `(` *expresie-argument* `,` *expresie-argument* `,` ... `,` *expresie-argument* `)`
în care argumentele funcţiei, separate cu virgule, pot fi expresii arbitrare, spre deosebire de *parametrii formali* din antet, care sunt *identificatori* (nume). Tot spre deosebire de antet (dar ca în matematică), *tipul* fiecărui argument nu se indică explicit, dar trebuie să corespundă cu tipul parametrului corespunzător, declarat în antet. Astfel, am folosit funcţia `sqrf`, şi nu `sqr`, deoarece argumentul `b` e real şi nu întreg. Expresiile date ca argument sunt evaluate pentru a determina valorile argumentelor pentru care se calculează valoarea funcţiei. Ca argument al unui apel de funcţie putem folosi orice expresie, inclusiv conţinând un alt apel de funcţie. De exemplu, pornind de la $x^6 = (x \cdot x^2)^2$ definim: `float pow6(float x) { return sqrf(x * sqrf(x)); }`

Declaraţie şi utilizare Funcţia `discrim` e corect definită dacă se cunoaşte semnificaţia identificatorului `sqrf`, de exemplu dacă definiţia lui `sqrf` apare în textul programului *înainte* de definiţia funcţiei `discrim` (la fel pentru `pow6`). În limbajul C, semnificaţia oricărui identificator trebuie precizată înainte de a fi folosit (trebuie *declarat*, aspect tratat în detaliu ulterior), tot după cum într-un text matematic trebuie definit întâi fiecare simbol folosit. De exemplu, parametrii unei funcţii sunt *declaraţi* în antetul acesteia, unde li se precizeaza numele şi tipul, şi ei pot fi apoi *folosiţi* în corpul funcţiei.

1.5 Programul principal

Până acum am definit şi apelat doar funcţii individuale, fără a scrie un program complet. Standardul C precizează că într-un *mediu de execuţie* (de exemplu sub un sistem de operare), la începutul rulării programului se apelează funcţia numită în mod convenţional `main`. Această funcţie are tipul `int`, şi returnează mediului de execuţie o valoare întreagă pe care acesta o poate testa pentru a determina dacă rularea programului s-a încheiat cu succes. Convenţional, terminarea cu succes se semnalează prin returnarea valorii 0 (zero) din `main`. Terminarea cu eroare se semnalează prin returnarea unei valori nenule, care poate reprezenta un cod ce identifică, prin diverse convenţii, tipul sau cauza erorii. Cu aceste precizări, cel mai simplu program C poate fi scris:

```
int main(void)
{
  return 0;
}
```

Programul returnează valoarea 0 (terminare cu succes), şi atât. Cuvântul cheie `void` reprezintă tipul vid (cu multime vidă de valori). El indică faptul că funcţia nu ia nici un parametru. Vom vedea mai târziu că funcţia `main` poate să aibă parametri transmişi de mediul de execuţie.

1.6 Tipărirea de texte

Un program are un efect observabil doar dacă provoacă o schimbare (numită efect lateral) în starea mediului de execuţie (mediului inconjurător), de exemplu prin tipărirea (scrierea) unui mesaj. În limbajul C, scrierea (şi citirea) se face tot prin intermediul unor funcţii. Funcţia `printf` permite tipărirea într-o varietate de moduri, prin specificarea unui *format* sau tipar. În cel mai simplu mod de folosire, apelul `printf("salut!")` are ca efect tipărirea textului `salut!` pe ecran.

Din punct de vedere sintactic, acesta este un *apel de funcţie*, la fel ca şi `sqr(4)`, având ca argument un *şir de caractere*. În C, o *constantă* şir de caractere se scrie între ghilimele `" "` care nu fac parte din şir ci au doar rolul de a-l identifica şi a-l separa de restul textului sursă al programului. Funcţia `printf` returnează un `int`, şi anume numărul de caractere scris, deci *expresia* `printf("salut!")` are o valoare, 6. De regulă nu ne interesează aceasta, ci doar efectul tipăririi.

Funcţii de bibliotecă şi fişiere antet Funcţia `printf` este o *funcţie de bibliotecă* specificată de standardul limbajului C. *Declaraţia* ei (precizând numele, tipul şi parametrii) e dată într-un *fişier antet* (*header file*) numit `stdio.h` (de la standard input/output) prezent în fiecare implementare conformă cu standardul. Declaraţia funcţiei `printf`, obligatorie înainte de *folosire*, se include cu secvenţa
`#include <stdio.h>`
Aceasta este o *directivă de preprocesare*, indicată prin caracterul `#` la început de linie, adresată *preprocesorului C*, care prelucrează sursa programului înainte de compilarea propriu-zisă. Efectul e ca şi cum conţinutul fişierului indicat, `stdio.h`, (cu toate declaraţiile) ar fi inclus la acel punct din program.

1.7 Secvenţierea

Putem scrie acum un prim program complet care are un efect vizibil şi tipăreşte un text pe ecran:

```
#include <stdio.h>

int main(void)
{
  printf("hello, world!");
  return 0;
}
```

Acest program simplu evidenţiază un aspect *fundamental* în programare, şi anume *secvenţierea*. În general, corpul oricărei funcţii (în acest caz, `main`) contine o *secvenţă* de *instrucţiuni*, fiecare reprezentând acţiuni care trebuie efectuate de program. Instrucţiunile se execută *secvenţial*, în ordinea dată de textul programului, mai puţin excepţiile (discutate ulterior) care modifică explicit acest comportament. De exemplu, instrucţiunea `return` încheie execuţia unei funcţii, indiferent dacă în corpul funcţiei mai urmează alte instrucţiuni. Programul dat are două instrucţiuni, al căror efect este:
– apelul la funcţia `printf` are ca efect afişarea textului `hello, world!`
– se încheie execuţia funcţiei `main` (şi a programului), returnând 0 (succes) mediului de execuţie.

Instrucţiunea expresie Prima instrucţiune din programul dat e un apel de funcţie. În general, o instrucţiune cu sintaxa *expresie* `;` are ca efect evaluarea expresiei date. În cazul de mai sus, esenţial e *efectul lateral* al evaluării expresiei, şi anume tipărirea; rezultatul expresiei (13, numărul de caractere scris de `printf`) e ignorat şi nu e folosit mai departe. Sunt corecte şi instructiunile `sqr(4);` (dacă e declarată funcţia `sqr`), sau `2 + 3;` sau chiar `3.14;` însă ele nu au nici un efect: expresiile sunt evaluate, dar nu sunt folosite şi nici nu are loc tipărirea lor sau alt efect vizibil.

Semnul `;` nu separă instrucţiunile, ci face parte integrantă din ele (fiecare, inclusiv ultima, ar fi incorectă fără prezenţa sa). Deci în limbajul C `;` nu e separator, ci terminator de instrucţiuni.

1.8 Comentarii

Structura unui program C e supusă unor reguli stricte (sintaxa limbajului). Totuşi, e utilă anotarea textului programului cu explicaţii în format liber. Limbajul C admite două feluri de *comentarii*:
– începând cu caracterele `/*` şi terminat cu caracterele `*/` (se poate întinde pe oricâte rânduri)
– începând cu caracterele `//` şi până la următorul caracter de linie nouă (neincluzându-l pe acesta)

Secventele de început de comentariu nu sunt interpretate în interiorul constantelor şir de caractere (comentariile nu pot începe într-un şir). Compilatorul ignoră conţinutul unui comentariu, şi caută doar sfârşitul său. În particular, într-un comentariu nu se interpretează caracterele de început de comentariu, şi deci nu se pot încadra comentarii în alte comentarii. În secvenţa `/* 1 /* 2 */ 3 */` comentariul se încheie la primul grup `*/` iar restul `3 */` nu e o secvenţă validă de cod C.

Comentariile sunt tratate ca şi cum ar fi spaţii albe, ele au efect de separator între două elemente lexicale succesive: `int/* un intreg */x` e echivalent cu `int x` .

În practica dezvoltării software, comentarea programelor e absolut necesară:
– ajută la înţelegerea programului de către alţii, sau chiar de autor la o citire ulterioară
– conţin informaţii despre evolutia programului: versiuni, revizii, adăugiri, erori corectate, etc.
– servesc ca bază pentru documentaţie (uneori extrasă automat din comentarii cu structură precizată)

Ca un minim, e bine ca programul să fie documentat la nivelul funcţiilor, precizând pentru fiecare:
– succint, ce prelucrare efectuează şi ce scop are
– semnificaţia fiecărui parametru, restricţii asupra valorilor valide, şi eventuale relaţii între parametri
– specificaţia funcţiei (relaţia între parametri şi valoarea returnată sau alte valori calculate de funcţie)
– eventuale efecte laterale (citiri, scrieri, variabile globale modificate, alocare de memorie)
– restricţii privind apelarea funcţiei (de exemplu înainte/după alte apeluri sau prelucrări în program)

1.9 Tipărirea de numere

Programe cu mai multe funcţii Un exemplu de program care defineşte şi apoi apelează o funcţie:

```
#include <stdio.h>
int sqr (int x) { return x * x; }       // definita inainte de folosire

int main(void)
{
  printf("3 ori 2 la patrat e ");       // tipareste un sir
  printf("%d", 3 * sqr(2));             // tipareste un intreg: 12
  return 0;
}
```

Un program poate conţine succesiv mai multe definiţii de funcţii (şi alte declaraţii, vom vedea ulterior). Funcţia `sqr` e definită înainte de `main`, pentru a permite apelarea ei din programul principal.

Tipărirea unui întreg Pentru a tipări un întreg, funcţia `printf` e apelată cu *două* argumente. Mecanismul funcţiilor cu număr variabil de argumente va fi detaliat ulterior; în esenţă, primul argument al lui `printf` este *întotdeauna* un *şir de caractere*, numit *format* sau *tipar* pentru că examinându-l, funcţia determină câte argumente urmează şi ce tip au. Pentru a tipări o valoare întreagă (în baza 10), se dă ca prim argument formatul (şirul de caractere) `"%d"` (de la *decimal*), şi ca al doilea argument *expresia* cu valoare întreagă care va fi tipărită.

Tipărirea unui real Următorul exemplu afişează valoarea dată de o funcţie de bibliotecă:

```
#include <math.h>               // inclus pentru declaratia functiei asin
#include <stdio.h>
int main(void)
{
  printf("Valoarea lui pi este 2 * asin(1): ");         // text
  printf("%f", 2 * asin(1.0));          // afiseaza valoarea 3.141593
  return 0;
}
```

Declaraţia funcţiei `asin` (arcsinus) e dată împreună cu alte funcţii matematice în fişierul `math.h` :

```
double asin(double x);
```

Parametrul şi rezultatul au tip real, dar cu precizie mai mare decât `float`, de unde şi numele (*double precision*). Tipul `double` (cuvânt cheie al limbajului C) e folosit de majoritatea funcţiilor matematice de bibliotecă, precizia de 6 zecimale a lui `float` fiind adesea insuficientă chiar pentru calcule uzuale.

Tipărirea are loc similar ca şi pentru întregi, dar folosind ca parametru alt *format* pentru tipărire, şirul `"%f"` (de la *floating point*). Implicit, tipărirea pentru reali se face cu 6 cifre după punctul zecimal.

Distingem faptul că prima apariţie a lui `2 * asin(1)` în program e înăuntrul unui şir de caractere: este un simplu text, afişat ca atare de `printf`. A doua apariţie este o expresie, dată ca argument la `printf`: ea este *evaluată* prin *apelarea* funcţiei `asin`, iar *valoarea* reală rezultantă e tipărită.

Conversii de tip Expresia `2 * asin(1.0)` înnmulţeşte un întreg cu un real. La operaţii între cele două tipuri, întregul e convertit automat (implicit) la real. Şi pentru argumente de functie are loc conversia valorii la tipul corespunzător parametrului, dacă acesta e cunoscut din declaraţia funcţiei; puteam scrie astfel şi `asin(1)`. La `printf`, formatul `%f` poate fi folosit şi pentru `float`, acesta fiind convertit automat la `double`. Alte reguli de conversie vor fi discutate şi sistematizate ulterior.

Sfârşitul de linie Pentru apeluri succesive la funcţia `printf`, afişarea se continuă de unde s-a oprit. Trecerea la linie nouă trebuie specificată explicit. O linie nouă într-un text e marcată de un caracter special (de control), a cărui tipărire are ca efect avansul cu o linie. Deoarece acest caracter nu are aspect vizibil, ci e caracterizat prin efectul său, el nu poate fi reprezentat direct în program (o constantă şir de caractere între `" "` nu poate fi despărţită pe mai multe linii). Convenţional, caracterul de linie nouă se reprezintă în program (de exemplu într-un şir de caractere) prin secvenţa de două caractere `\n` (de la *newline*). Secvenţa `printf("unu\n"); printf("doi");` tipăreşte cele două cuvinte pe linii succesive, şi cursorul rămâne pe linia a doua. Acelaşi efect îl are `printf("unu\ndoi");` .

Caracterul `\` folosit pentru introducerea secventelor speciale `\n` şi altele trebuie dublat pentru a se reprezenta pe sine însuşi într-un şir. Astfel, `printf("c:\\windows")` va tipări `c:\windows` .

Formatul general în `printf` Se poate combina în acelaşi apel la `printf` scrierea valorilor de mai multe tipuri. Funcţia `printf` parcurge şirul de caractere dat ca prim argument, şi tipăreşte fiecare caracter reprezentat direct sau prin secvenţe speciale (`\n`) . La întâlnirea unui *specificator de format*, cum ar fi `%d` sau `%f`, `printf` ia următorul argument din cele primite şi îl tipăreşte corespunzător cu formatul indicat (întreg, real, etc.). De exemplu, `printf("un intreg %d si un real %f\n", 2, 3.14)` va tipări: `un intreg 2 si un real 3.140000` trecând apoi la linie nouă. Pentru a fi interpretat ca un caracter obişnuit, caracterul `%` trebuie dublat în şirul de format: `printf("7%%")` va tipări `7%` .

Corespondenţa între numărul de specificatori de format cu `%` din şirul dat ca prim argument şi numărul argumentelor următoare, precum şi corespondenţa tipurilor indicate cu cele ale argumentelor trebuie verificată de programator. În caz de nepotrivire, comportamentul funcţiei `printf` e nedefinit!

1.10 Operatorul condiţional. Expresia condiţională

Adeseori, valoarea unei funcţii e calculată cu formule (expresii) diferite pe fragmente ale domeniului de definitie. Un exemplu e funcţia valoare absolută pentru numere întregi:

$$abs : \mathbb{Z} \to \mathbb{Z} \qquad abs(x) = \begin{cases} x & x \geq 0 \\ -x & \text{altfel} \end{cases}$$

O asemenea funcţie nu poate fi definită cu elementele de limbaj discutate până acum: instrucţiunea `return` acceptă o *singură* expresie pentru a calcula valoarea funcţiei. E nevoie de o expresie a cărei valoare depinde de rezultatul unei *decizii* (condiţii), ceea ce se poate scrie în C cu *operatorul condiţional* ternar `? :` (cu trei operanzi). Sintaxa expresiei condiţionale este: *condiţie* `?` *expresie1* `:` *expresie2* . Pentru calculul valorii unei expresii condiţionale se evaluează întâi condiţia. Dacă ea este adevărată, se evaluează doar *expresie1*, iar dacă e falsă, se evaluează doar *expresie2*. În ambele cazuri, valoarea expresiei evaluate devine rezultatul întregii expresii condiţionale. Tipurile celor două expresii trebuie deci să corespundă. Cu aceste explicaţii, putem scrie functia valoare absolută în C după cum urmează:

```
int abs(int x)
{
  return x >= 0 ? x : -x;       // operator minus unar
}
```

În definiţia cu acoladă am scris condiţia "altfel" în loc de $x < 0$, pentru a ţine cont de faptul că operatorul `? :` evaluează (testează) condiţia o singură data, obţinând o valoare adevărată sau falsă. Putem deci traduce direct expresia în C dacă cele două cazuri din definiţie sunt unul opusul celuilalt.

Funcţia `abs` pentru întregi există ca funcţie de bibliotecă declarată în fişierul antet `stdlib.h` . Pentru reali, există funcţia `fabs` cu parametru şi rezultat `double`, declarată în `math.h` .

Operatori logici de comparare Ca prim operand în expresia condiţională avem nevoie de o subexpresie (condiţia) cu valoare logică adevărată sau falsă. Pentru aceasta, putem folosi:
- *operatorii relaţionali* `<`, `>`, `<=`, `>=` (mai mic, mai mare, mai mic sau egal, mai mare sau egal)
- *operatorii de egalitate* `==` (egal) şi `!=` (diferit)

Alţi operatori, precum şi detalii despre noţiunea de valoare logică în C vor fi discutaţi ulterior. Ca exemple, putem defini similar cu funcţia valoare absolută funcţii simple pentru maxim şi minim:

```
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }
```

Atenţie! În C, testul de egalitate se scrie `==` cu două caractere consecutive. Un singur `=` are alt înţeles (atribuire), şi confuzia sau neatentia pot duce la erori uneori greu de detectat! În text, vom distinge în continuare după scris relaţiile *matematice* ($x^2 = y$) de fragmentele de `program` (`x*x == y`).

Cei 6 operatori prezentaţi sunt binari, adică au doi operanzi. Scrierea informală $2 < x < 5$ din matematică are alt înţeles în C (nedorit aici). Folosim *două* teste *separate*, ca în următorul exemplu.

Condiţii multiple Întreaga construcţie sintactică *condiţie* `?` *expresie1* `:` *expresie2* reprezintă o *singură expresie*, care poate fi folosită oriunde sintaxa limbajului cere o expresie, inclusiv în cele două subexpresii de pe ramurile expresiei conditionale. Ilustrăm aceasta pornind de la funcţia semn:

$$sgn : \mathbb{Z} \to \{-1, 0, 1\} \qquad sgn(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

Scrisă astfel, funcţia nu poate fi tradusă direct în program, deoarece operatorul condiţional permite doar o decizie binară, pe două cazuri, şi nu pe mai multe. Putem reformula problema în felul următor: efectuăm o primă decizie (test), de exemplu după condiţia $x < 0$. Dacă aceasta e adevărată, am identificat primul caz. Dacă nu, am redus totuşi problema, numărul de cazuri posibile rămase pentru x fiind cu unul mai mic, adică două, şi putem decide valoarea funcţiei cu încă un test:

$$sgn(x) = \begin{cases} \text{dacă } x < 0 & -1 \\ \text{altfel } (x \geq 0) & \begin{cases} \text{dacă } x = 0 & 0 \\ \text{altfel } (x > 0) & 1 \end{cases} \end{cases}$$

Aşa cum a fost rescrisă, funcţia necesită în continuare mai multe decizii pentru stabilirea valorii, dar fiecare din ele e o decizie logică cu două ramuri, după valoarea de adevăr a unei condiţii, şi poate fi exprimată în C cu operatorul condiţional. O altă diferenţă e că prima variantă are fiecare ramură identificată printr-o condiţie, iar ordinea dintre ele nu e relevantă, condiţiile fiind exclusive între ele. În varianta a doua, ordinea de evaluare a conditiilor contează, rezultatul unei decizii dând informaţie utilă (notată între paranteze) pentru deciziile următoare. Astfel, o valoare falsă pentru condiţia $x = 0$ permite să deducem că $x > 0$ doar datorită faptului că pe această ramură a fost stabilit deja că $x \geq 0$. Cu aceste observaţii, funcţia poate fi transcrisă direct după cum urmează:

```
int sgn(int x)
{
  return x < 0 ? -1
               : x == 0 ? 0 : 1;
}
```

În scrierea codului am aliniat vertical cele două ramuri ale primei decizii, pentru a evidenţia faptul că subexpresia de pe a doua ramură a primei expresii conditionale e ea însăşi o expresie condiţională. Desigur că descompunerea aleasa în decizii individuale nu este unică. Putem scrie de exemplu şi

```
  return x >= 0 ? x > 0 ? 1 : 0
                : -1;
```

În acest caz, expresia de pe prima ramură (adevărat) a primei decizii este la rândul ei o expresie condiţională. Ramurile unei expresii condiţionale complexe şi gruparea lor pot fi identificate unic, şi nu depind de punerea în pagină a codului: orice `?` corespunde unui unic `:` după cum orice decizie are o ramură pentru *adevărat* şi una pentru *fals*.

1.11 Descompunerea în subprobleme

În conceperea unui program (rezolvarea unei probleme), dacă identificăm o prelucrare (un calcul) care reprezintă rezolvarea unei subprobleme, putem defini o funcţie în acest scop. Ea poate fi atunci apelată, cu parametrii corespunzători, ori de câte ori calculul (subproblema) respectivă apare în soluţie.

De exemplul, să scriem o funcţie care calculează minimul a trei numere a, b şi c, de exemplu, întregi. Comparăm întâi două dintre numere. Dacă $a < b$, atunci sigur b nu poate fi minimul, deci rămâne să decidem între a şi al treilea număr c. Altfel, dacă $a \geq b$, e suficient să examinăm ca şi candidaţi pe b şi c. În ambele cazuri, am redus problema la una mai simplă: calculul minimului a două numere.

Presupunând existenţa unei funcţii `min2` care face acest lucru, scriem pentru problema iniţială:

```
int min3(int a, int b, int c) { return a < b ? min2(a, c) : min2(b, c); }
```

Apoi putem rezolva imediat subproblema mai simplă rămasă:

```
int min2(int a, int b) { return a < b ? a : b; }
```

Dacă nu am fi definit o funcţie pentru subproblema mai simplă, minimul a două numere, ar fi trebuit să scriem, dezvoltând:

```
int min3lung(int a, int b, int c)
{
  return a < b ? (a < c ? a : c) : (b < c ? b : c);
}
```

Observăm că expresiile din paranteze urmează acelaşi tipar, doar cu alte nume de variabile. Spunem că avem *duplicare de cod*. Acest aspect e nedorit în programare. Codul devine mai lung, şi mai greu de citit. Mai mult, apar probleme când codul duplicat trebuie modificat (corectat sau adăugit): efortul trebuie repetat pentru fiecare copie a fragmentului de cod, şi există riscul de a uita pe alocuri modificările, ducând la programe inconsistente sau eronate. Gruparea prelucrărilor cu anumit scop într-o funcţie conduce în schimb la cod mai bine structurat şi mai inteligibil, iar în caz de modificări, ele trebuie operate într-un singur loc.

În programul C, punem definiţia lui `min2` *înainte* de `min3`, deoarece aceasta din urmă o apelează. În rezolvarea problemei însă, am conceput funcţia `min3` chiar înainte de a fi stabilit cum rezolvăm subproblema rezultată (funcţia `min2`). Spunem că am proiectat soluţia *de sus în jos* (*top-down*), prin descompunere în probleme mai simple. Invers, ne putem gândi întâi la funcţiile elementare de care avem nevoie, şi apoi să le combinăm în funcţii mai complexe până ajungem la soluţia problemei (dezvoltare *de jos în sus*, sau *bottom-up*). Pornind de sus în jos suntem ghidaţi adesea după structura problemei, ceea ce poate fi mai natural. Abordarea de jos în sus poate conduce în schimb la definirea de funcţii simple, de utilitate generală, pe care le putem *refolosi* în rezolvarea altor probleme.

Ca un alt exemplu, scriem o funcţie care calculează *mediana* a trei numere, adică numărul care se află în poziţia centrală din şir dacă le ordonăm după mărime, între celelalte două.

Începem prin a compara două din numere, a şi b. Obtinem o informaţie suplimentară: trebuie să calculăm mediana a trei numere, din care ştim cum sunt ordonate primele două. Scriem funcţia `mediana`, urmând să definim apoi o funcţie, să-i spunem `med3ord` pentru această subproblemă:

```
int mediana(int a, int b, int c)
{
  return a < b ? med3ord(a, b, c) : med3ord(b, a, c);
}
```

Continuăm scriind o funcţie pentru subproblema obţinută, ştiind că primele două argumente sunt ordonate crescător. Dacă al treilea număr c e în stânga intervalului $[a, b]$, mediana este a, alfel, cel mai mic număr este a, şi mediana e minimul dintre b şi c, funcţie pe care am mai scris-o deja (`min2`).

```
int med3ord(int a, int b, int c)        // mediana, stiind ca a <= b
{
  return a > c ? a : min2(b, c);
}
```

Într-un program complet scriem funcţiile în ordinea: `min2`, `med3ord`, `mediana`, deoarece fiecare o foloseşte pe cea dinainte.