

1 Introducere în programarea în C

1.1 Funcții în limbajul C

Calculare și funcții La origine, rolul programelor e de a efectua în principal *calculare* matematice. Discutăm de aceea structura programelor făcând o paralelă cu noțiunile din matematică, care stau și la baza studiului teoretic al limbajelor de programare.

În matematică, noțiunea de calcul e strâns legată de cea de *funcție*: în calcule se *folosesc* funcții cunoscute, se *definesc* altele noi și se aplică într-o anumită *succesiune*. La fel, codul dintr-un fișier de program C e structurat în funcții, care sunt definite similar celor din matematică.

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

$$sqr : \mathbb{Z} \rightarrow \mathbb{Z} \quad sqr(x) = x \cdot x$$

Această *definiție de funcție* are două părți: prima specifică numele funcției (*sqr*), domeniul său de definiție (mulțimea numerelor întregi, \mathbb{Z}) și domeniul său de valori, de asemenea \mathbb{Z} . A doua parte specifică modul de calcul al valorii funcției *sqr(x)* pornind de la valoarea argumentului *x*.

În limbajul C, *definiția* aceleiași funcții are aspectul următor:

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

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

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

$$sqr f : \mathbb{R} \rightarrow \mathbb{R} \quad sqr f(x) = x \cdot x$$

Funcția *sqr f* e *diferită* de funcția *sqr* scrisă anterior, deoarece are alt domeniu de definiție și de valori. Strict vorbind, și operația de înmulțire e diferită, fiind definită pe altă mulțime, chiar dacă se folosește aceeași notație. La fel și în limbajul C, nu putem folosi funcția definită anterior pe întregi pentru a calcula pătratul unui număr real, ci trebuie să definim altă funcție:

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

Cuvântul `float` folosit pentru domeniul de definiție și de valori indică numerele reale.

1.2 Tipuri și operatori

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

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

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

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

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

Astfel, $7.0/2.0$ e 3.5 , dar $7/2$ dă valoarea 3 ! La împărțirea întregilor cu semn, rezultatul are valoarea absolută ca și pentru întregi pozitivi, iar semnul e dat de regula uzuală, deci $7/-2$ este -3 . Semnul restului (obținut cu operatorul $\%$) e același cu semnul deîmpărțitului. Deci $a/b * b + a\%b$ e egal cu a (ecuația împărțirii cu rest). Exemple fără semn: $9/5$ este 1 și $9\%5$ este 4 , iar cu semn: $9/-5$ este -1 și $9\%-5$ este 4 ; $-9/5$ este -1 și $-9\%5$ este -4 ; $-9/-5$ este 1 și $-9\%-5$ este -4 .

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

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

1.3 Terminologie: sintaxă și semantică

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

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

În C se face distincție între litere mari și litere mici. Astfel, `nr`, `NR`, `Nr` și `nR` sunt identificatori diferiți! Toate cuvintele cheie sunt scrise cu minuscule, ca și marea majoritate a funcțiilor standard.

Într-un program, elementele lexicale sunt grupate în *construcții de limbaj* mai complexe, după anumite *reguli* de scriere care formează *sintaxa* limbajului. Am dat ca exemplu *definiția* unei funcții, formată din *antet* și *corp*. *Antetul* unei funcții (în general cu mai mulți parametri) are sintaxa (forma): *tip-rezultat nume-funcție* (*tip-parametru nume-parametru*, \dots , *tip-parametru nume-parametru*)

Corpul unei funcții conține între acolade *instrucțiuni*: elemente de limbaj ce specifică *acțiuni* care să fie executate de program. Am folosit până acum doar instrucțiunea `return`, cu sintaxa:

```
return expresie ;
```

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

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

Punerea în pagină a programelor Într-un program C , *spațiile albe* (fără reprezentare pe ecran, cum ar fi caracterul spațiu, 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 înțelesul programului. Astfel, funcția `sqr` definită anterior poate fi scrisă `int sqr(int x){return x*x;}` prin eliminarea spațiilor. Cele 3 spații rămase sunt necesare ca separatori, în acest caz, între cuvinte cheie și identificatori. Spațiul nu e necesar în expresia `3+-5` dar e necesar în `3- -5` pentru că `--` ar avea alt înțeles (decrementare) în limbajul C .

Se recomandă respectarea unor convenții pentru mai buna lizibilitate a programelor C , ca de exemplu alinierea una sub alta a instrucțiunilor succesive, și *indentarea* (de regulă cu două spații la dreapta) a unei secvențe de instrucțiuni față de acoladele $\{ \}$ inconjurătoare.

1.4 Apelarea funcțiilor

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

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

Expresia apel În expresia de calcul a valorii funcției apare *expresia sqrtf(b)*: e *apelată* (folosită) funcția `sqrtf` pentru a calcula pătratul parametrului `b`. Sintaxa *apelului de funcție* este aceeași ca și în matematică: *funcție* (*expresie-argument* , *expresie-argument* , ... , *expresie-argument*)

în care argumentele funcției, 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 funcția `sqrtf`, și nu `sqr`, deoarece argumentul `b` e real și nu întreg. Expresiile date ca argument sunt evaluate pentru a determina valorile argumentelor pentru care se calculează valoarea funcției. Ca argument al unui apel de funcție putem folosi orice expresie, inclusiv conținând un alt apel de funcție. De exemplu, pornind de la $x^6 = (x \cdot x^2)^2$ definim: `float pow6(float x) { return sqrtf(x * sqrtf(x)); }`

Declarație și utilizare Funcția `discrim` e corect definită dacă se cunoaște semnificația identificatorului `sqrtf`, de exemplu dacă definiția lui `sqrtf` apare în textul programului *înainte* de definiția funcției `discrim` (la fel pentru `pow6`). În limbajul C, semnificația 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 funcții sunt *declarați* în antetul acesteia, unde li se precizează numele și tipul, și ei pot fi apoi *folosiți* în corpul funcției.

1.5 Programul principal

Până acum am definit și apelat doar funcții individuale, fără a scrie un program complet. Standardul C precizează că într-un *mediu de execuție* (de exemplu sub un sistem de operare), la începutul rulării programului se apelează funcția numită în mod convențional `main`. Această funcție are tipul `int`, și returnează mediului de execuție o valoare întreagă pe care acesta o poate testa pentru a determina dacă rularea programului s-a încheiat cu succes. Convențional, 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 convenții, 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), și atât. Cuvântul cheie `void` reprezintă tipul `void` (cu multime `void` de valori). El indică faptul că funcția nu ia nici un parametru. Vom vedea mai târziu că funcția `main` poate să aibă parametri transmiși de mediul de execuție.

1.6 Tipărirea de texte

Un program are un efect observabil doar dacă provoacă o schimbare (numită efect lateral) în starea mediului de execuție (mediului inconjurător), de exemplu prin tipărirea (scrierea) unui mesaj. În limbajul C, scrierea (și citirea) se face tot prin intermediul unor funcții. Funcția `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 funcție*, la fel ca și `sqr(4)`, având ca argument un *șir de caractere*. În C, o *constantă* șir de caractere se scrie între ghilimele " " care nu fac parte din șir ci au doar rolul de a-l identifica și a-l separa de restul textului sursă al programului. Funcția `printf` returnează un `int`, și anume numărul de caractere scris, deci *expresia* `printf("salut!")` are o valoare, 6. De regulă nu ne interesează aceasta, ci doar efectul tipăririi.

Funcții de bibliotecă și fișiere antet Funcția `printf` este o *funcție de bibliotecă* specificată de standardul limbajului C. *Declarația* ei (precizând numele, tipul și parametrii) e dată într-un *fișier antet* (*header file*) numit `stdio.h` (de la standard input/output) prezent în fiecare implementare conformă cu standardul. Declarația funcției `printf`, obligatorie înainte de *folosire*, se include cu secvența `#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 și cum conținutul fișierului indicat, `stdio.h`, (cu toate declarațiile) ar fi inclus la acel punct din program.

1.7 Secvențierea

Putem scrie acum un prim program complet care are un efect vizibil și tipărește un text pe ecran:

```
#include <stdio.h>

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

Acest program simplu evidențiază un aspect *fundamental* în programare, și anume *secvențierea*. În general, corpul oricărei funcții (în acest caz, `main`) conține o *secvență* de *instrucțiuni*, fiecare reprezentând acțiuni care trebuie efectuate de program. Instrucțiunile se execută *secvențial*, în ordinea dată de textul programului, mai puțin excepțiile (discutate ulterior) care modifică explicit acest comportament. De exemplu, instrucțiunea `return` încheie execuția unei funcții, indiferent dacă în corpul funcției mai urmează alte instrucțiuni. Programul dat are două instrucțiuni, al căror efect este:

- apelul la funcția `printf` are ca efect afișarea textului `hello, world!`
- se încheie execuția funcției `main` (și a programului), returnând 0 (succes) mediului de execuție.

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

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

1.8 Comentarii

Structura unui program C e supusă unor reguli stricte (sintaxa limbajului). Totuși, e utilă anotarea textului programului cu explicații în format liber. Limbajul C admite două feluri de *comentarii*:

- începând cu caracterele `/*` și terminat cu caracterele `*/` (se poate întinde pe oricâte rânduri)
- începând cu caracterele `//` și până la următorul caracter de linie nouă (neincluzându-l pe acesta)

Secvențele de început de comentariu nu sunt interpretate în interiorul constantelor șir de caractere (comentariile nu pot începe într-un șir). Compilatorul ignoră conținutul unui comentariu, și caută doar sfârșitul său. În particular, într-un comentariu nu se interpretează caracterele de început de comentariu, și deci nu se pot încadra comentarii în alte comentarii. În secvența `/* 1 /* 2 */ 3 */` comentariul se încheie la primul grup `*/` iar restul `3 */` nu e o secvență validă de cod C.

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

În practica dezvoltării software, comentarea programelor e absolut necesară:

- ajută la înțelegerea programului de către alții, sau chiar de autor la o citire ulterioară
- conțin informații despre evoluția programului: versiuni, revizii, adăugiri, erori corectate, etc.
- servesc ca bază pentru documentație (uneori extrasă automat din comentarii cu structură precizată)

Ca un minim, e bine ca programul să fie documentat la nivelul funcțiilor, precizând pentru fiecare:

- succint, ce prelucrare efectuează și ce scop are
- semnificația fiecărui parametru, restricții asupra valorilor valide, și eventuale relații între parametri
- specificația funcției (relația între parametri și valoarea returnată sau alte valori calculate de funcție)
- eventuale efecte laterale (citiri, scrieri, variabile globale modificate, alocare de memorie)
- restricții privind apelarea funcției (de exemplu înainte/după alte apeluri sau prelucrări în program)

1.9 Tipărirea de numere

Programe cu mai multe funcții Un exemplu de program care definește și apoi apelează o funcție:

```
#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 conține succesiv mai multe definiții de funcții (și alte declarații, vom vedea ulterior). Funcția `sqr` e definită înainte de `main`, pentru a permite apelarea ei din programul principal.

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

Tipărirea unui real Următorul exemplu afișează valoarea dată de o funcție 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;
}
```

Declarația funcției `asin` (arcsinus) e dată împreună cu alte funcții matematice în fișierul `math.h`:

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

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

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

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

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

Sfârșitul de linie Pentru apeluri succesive la funcția `printf`, afișarea 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 șir de caractere între " " nu poate fi despărțită pe mai multe linii). Convențional, caracterul de linie nouă se reprezintă în program (de exemplu într-un șir de caractere) prin secvența de două caractere `\n` (de la *newline*). Secvența `printf("unu\n"); printf("doi");` tipărește cele două cuvinte pe linii succesive, și cursorul rămâne pe linia a doua. Același efect îl are `printf("unu\ndoi");` .

Caracterul `\` folosit pentru introducerea secvențelor speciale `\n` și altele trebuie dublat pentru a se reprezenta pe sine însuși într-un șir. Astfel, `printf("c:\\windows")` va tipări `c:\windows` .

Formatul general în printf Se poate combina în același apel la `printf` scrierea valorilor de mai multe tipuri. Funcția `printf` parcurge șirul de caractere dat ca prim argument, și tipărește fiecare caracter reprezentat direct sau prin secvențe speciale (`\n`) . La întâlnirea unui *specificator de format*, cum ar fi `%d` sau `%f`, `printf` ia următorul argument din cele primite și îl tipărește 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 obișnuit, caracterul `%` trebuie dublat în șirul de format: `printf("7%%")` va tipări `7%` .

Corespondența între numărul de specificatori de format cu `%` din șirul dat ca prim argument și numărul argumentelor următoare, precum și corespondența tipurilor indicate cu cele ale argumentelor trebuie verificată de programator. În caz de nepotrivire, comportamentul funcției `printf` e nedefinit!

1.10 Operatorul condițional. Expresia condițională

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

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

O asemenea funcție nu poate fi definită cu elementele de limbaj discutate până acum: instrucțiunea `return` acceptă o *singură* expresie pentru a calcula valoarea funcției. E nevoie de o expresie a cărei valoare depinde de rezultatul unei *decizii* (condiții), ceea ce se poate scrie în C cu *operatorul condițional* ternar `? :` (cu trei operanzi). Sintaxa expresiei condiționale este: *condiție ? expresie1 : expresie2* . Pentru calculul valorii unei expresii condiționale se evaluează întâi condiția. 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 condiționale. Tipurile celor două expresii trebuie deci să corespundă. Cu aceste explicații, putem scrie funcția valoare absolută în C după cum urmează:

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

În definiția cu acoladă am scris condiția "altfel" în loc de `x < 0`, pentru a ține cont de faptul că operatorul `? :` evaluează (testează) condiția o singură dată, obținând o valoare adevărată sau falsă. Putem deci traduce direct expresia în C dacă cele două cazuri din definiție sunt unul opusul celuilalt.

Funcția `abs` pentru întregi există ca funcție de bibliotecă declarată în fișierul antet `stdlib.h` . Pentru reali, există funcția `fabs` cu parametru și rezultat `double`, declarată în `math.h` .

Operatori logici de comparare Ca prim operand în expresia condițională avem nevoie de o subexpresie (condiția) cu valoare logică adevărată sau falsă. Pentru aceasta, putem folosi:

- *operatorii relaționali* <, >, <=, >= (mai mic, mai mare, mai mic sau egal, mai mare sau egal)
- *operatorii de egalitate* == (egal) și != (diferit)

Alți operatori, precum și detalii despre noțiunea de valoare logică în C vor fi discutați ulterior.

Ca exemple, putem defini similar cu funcția valoare absolută funcții simple pentru maxim și minim:

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

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

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

Condiții multiple Întreaga construcție sintactică *condiție ? 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 condiționale. Ilustrăm aceasta pornind de la funcția semn:

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

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

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

Așa cum a fost rescrisă, funcția 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 condiții, și poate fi exprimată în C cu operatorul condițional. O altă diferență e că prima variantă are fiecare ramură identificată printr-o condiție, iar ordinea dintre ele nu e relevantă, condițiile fiind exclusive între ele. În varianta a doua, ordinea de evaluare a condițiilor contează, rezultatul unei decizii dând informație utilă (notată între paranteze) pentru deciziile următoare. Astfel, o valoare falsă pentru condiția $x = 0$ permite să deducem că $x > 0$ doar datorită faptului că pe această ramură a fost stabilit deja că $x \geq 0$. Cu aceste observații, funcția 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 evidenția faptul că subexpresia de pe a doua ramură a primei expresii condiționale e ea însăși o expresie condițională. Desigur că descompunerea aleasă în decizii individuale nu este unică. Putem scrie de exemplu și

```
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 condițională. Ramurile unei expresii condiționale complexe și gruparea lor pot fi identificate unic, și nu depind de punerea în pagină a codului: orice ? corespunde unui unic : după cum orice decizie are o ramură pentru *adevărat* și 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 funcție în acest scop. Ea poate fi atunci apelată, cu parametrii corespunzători, ori de câte ori calculul (subproblema) respectivă apare în soluție.

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

Presupunând existența unei funcții `min2` care face acest lucru, scriem pentru problema inițială:

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

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

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

Începem prin a compara două din numere, a și b . Obținem o informație suplimentară: trebuie să calculăm mediana a trei numere, din care știm cum sunt ordonate primele două. Scriem funcția `mediana`, urmând să definim apoi o funcție, 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 funcție pentru subproblema obținută, știind 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 , și mediana e minimumul dintre b și c , funcție 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 funcțiile în ordinea: `min2`, `med3ord`, `mediana`, deoarece fiecare o folosește pe cea dinainte.