3 Caractere şi citirea lor

În general, un program operează asupra unor date de intrare, efectuează prelucrări (calcule) şi produce ca rezultat nişte date de ieşire.

În exemplele de până acum, rezultatele au fost scrise la ieşire, folosind funcţia `printf`, dar valorile de intrare prelucrate au fost specificate direct în program, de obicei ca argumente ale funcţiilor care efectuează prelucrarea. Aceasta necesită însă modificarea şi recompilarea programului ori de câte ori dorim să calculăm o nouă valoare, de exemplu `fact(4)` în loc de `fact(6)` .

Discutăm în continuare cum se pot scrie programe cu care utilizatorul poate să interacţioneze direct, introducând (la tastatură) datele pe care programul să le prelucreze. De exemplu, cum scriem un program care să calculeze şi tipărească factorialul unui număr tastat de utilizator?

3.1 Reprezentarea caracterelor

Exprimându-ne precis, utilizatorul nu introduce de fapt un *număr* (care e o noţiune abstractă, matematică), ci *reprezentarea* sa ca o secvenţă de cifre (aici, în baza 10). Cifrele zecimale, împreună cu literele mari şi mici, operatori, semne de punctuaţie, etc. formează un *alfabet* de *caractere* (simboluri) care pot fi introduse de la tastatură. Pentru ca programele să poată opera cu caracterele introduse, trebuie să definim o altă *reprezentare* a caracterelor, în calculator. Reprezentarea printr-o matrice de pixeli e potrivită pentru afişarea pe ecran, dar nu şi pentru memorarea eficientă sau prelucrările de cuvinte (şiruri de caractere) care apar frecvent în programe. Calculatoarele fiind concepute să lucreze în principal cu *numere*, vom reprezenta caracterele prin numere întregi: dispunem toate simbolurile alfabetului într-o anumită ordine şi reprezentăm un caracter prin numărul de ordine (poziţia) în acest şir.

Cea mai utilizată reprezentare de acest tip este *codul ASCII* (American Standard Code for Information Interchange), care foloseşte pentru codificare întregii de la 0 la 127 (care se pot reprezenta în baza 2 pe 7 biţi). Codul ASCII defineşte 95 de caractere tipăribile, de la codul 32 (pentru *spaţiu*) până la 126; codul 127 (Delete) şi cele de la 0 la 31 sunt folosite pentru *caractere de control*, netipăribile, care pot avea înţeles special la afişare sau transmiterea datelor. Alte tabele derivate, standardizate de ISO folosesc coduri până la 255 pentru a reprezenta caractere suplimentare (cu diacritice, alte simboluri, etc.)

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
-------------------------------------------------------
0x0    \0                    \a \b \t \n \v \f \r
0x10:
0x20:      !  "  #  $  %  &  ’  (  )  *  +  ,  -  .  /
0x30:   0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
0x40:   @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
0x50:   P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
0x60:   ‘  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
0x70:   p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

În tabelul de mai sus, am dispus caracterele pe 8 linii şi 16 coloane, pentru a obţine uşor codul ASCII al fiecărui caracter ca valoare *în baza 16*.

Convenţional, *cifrele hexazecimale* cu valori de la 10 la 15 se reprezintă prin literele de la `A` la `F` (fie mari, fie mici). În limbajul C, numerele hexazecimale se disting prin prefixul `0x` sau `0X`. Astfel, din tabel vedem că litera A are codul ASCII `0x41`, adică 65. Litera z are codul `0x70 + 0xA`, adică $7 \cdot 16 + 10 = 122$.

Observăm că în tabela ASCII, cifrele, litere mari şi literele mici sunt reprezentate în trei grupuri contigue, dar separate.

Caracterele fiind reprezentate în calculator ca întregi, când scriem programe care prelucrează caractere, ar trebui să folosim codurile lor. Conform tabelului de mai sus, pentru a verifica dacă un caracter dintr-un şir este `C` ar trebui să-l comparăm cu numărul `0x43` (67). Scrierea numerică ar fi atât un inconvenient, cât şi o posibilă sursă de erori şi dependenţe de implementare.

Strict vorbind, standardul C99 nu prevede folosirea codificării ASCII, ci doar cerinţa de a putea reprezenta cifrele, literele mari şi mici, majoritatea celorlalte semne grafice uzuale, spaţiul şi sfârşitul de linie, ca reprezentarea fiecărui caracter să încapă pe un octet, iar cifrele să aibă valori succesive în reprezentare.

În limbajul C, ne putem referi la *valoarea* unui caracter (codul său numeric) incluzând caracterul între apostroafe, de exemplu: `'c'`, `'7'`, `':'`. Aceste valori se numesc *constante caracter*, al treilea tip de constante întâlnit până acum, pe lângă cele întregi şi reale. Între apostroafe putem încadra orice caracter tipăribil (inclusiv spaţiul), dar apostroful însuşi trebuie prefixat cu caracterul `\` (*backslash*): `'\''` pentru a-l deosebi de cele două apostroafe delimitator.

Caracterele de control (netipăribile) uzuale sunt reprezentate prin secvenţe speciale formate din `\` urmat de o literă care sugerează numele caracterului:

`'\a'`	alert	`'\n'`	newline	`'\r'`	carriage return
`'\b'`	backspace	`'\v'`	vertical tab	`'\''`	apostrof
`'\t'`	tab	`'\f'`	form feed	`'\\'`	backslash

Caracterul `\` trebuie dublat pentru a fi interpretat ca atare şi nu ca început de secvenţă specială, o convenţie uzuală, folosită şi pentru `%` în `printf`.

Ca aplicaţie, scriem o funcţie simplă care returnează valoarea (codul) cifrei hexazecimale care reprezintă un număr între 0 şi 15. Funcţia va returna deci in cod de caracter fie între `'0'` şi `'9'`, fie între `'A'` şi `'F'`. Cifrele zecimale fiind reprezentate consecutiv, codul pentru cifra de valoare `n` este `'0' + n` (cu `n` poziţii mai mare decât codul lui `0`). Pentru litere, codul va fi cu `n - 10` poziţii mai mare decât cel al lui `A` (de exemplu, `'A' + (14 - 10) == 'E'`).

```
// returneaza cifra hexazecimala pentru 0 <= n < 16, altfel '?'
int xdigit(unsigned n)
{
  return n < 10 ? '0' + n
                : n < 16 ? 'A' + n - 10
                         : '?';
}
```

Am tratat cazul de eroare (apelul cu un parametru mai mare decât 15) prin returnarea valorii `'?'`, care ar fi evidentă la tipărire. O altă variantă ar fi returnarea valorii `-1` care nu reprezintă un caracter. Decizia e la latitudinea proiectantului funcţiei, dar *cazurile de eroare trebuie tratate* şi documentate, împreună cu funcţionalitatea, în comentarii.

3.2 Scrierea de caractere

Caracterele fiind reprezentate intern ca întregi, se pune întrebarea e cum se poate produce *reprezentarea fizică* a unui caracter pornind de la codul său, adică cum se poate scrie (tipări). Acest lucru e realizat în C de funcţia standard

```
int putchar(int c);            // declarata in stdio.h
```

care scrie caracterul cu valoarea (codificarea numerică) dată la ieşirea standard (în mod implicit, ecranul). Funcţia `putchar` returnează chiar valoarea parametrului (caracterul scris), convenţie uzuală la unele funcţii standard care, deşi au scopul principal de a produce un *efect vizibil* (sau *efect lateral*, noţiune discutată ulterior), returnează şi un rezultat utilizabil la nevoie în program.

Apelul `putchar('0')` va tipări deci cifra `0` (fără apostroafe, folosite doar la reprezentarea constantelor caracter în program). Nu vom scrie `putchar(48)`, deoarece e mai greu de înţeles şi depinde de tablelul de codificare folosit. Apelul `putchar(xdigit(12))` (cu funcţia `xdigit` de mai sus) va scrie litera (cifra hexazecimală) `C`.

Reamintim că o constantă caracter *trebuie încadrată între apostroafe*; astfel `putchar(9)` nu scrie cifra `9`, ci caracterul cu valoare 9 (în codul ASCII: *tab*); `'a' + 5` este `'f'`, dar în expresia `a + 5` se presupune că `a` este un *identificator* (declarat anterior), a cărui valoare se adună cu 5, etc.

3.3 Citirea de caractere

Într-un program C, citirea unui caracter se poate face cu funcţia standard

```
int getchar(void);            // declarata in stdio.h
```

Funcţia nu are nevoie de parametri (citirea se face de la *intrarea standard*, implicit tastatura). Ea returnează valoarea caracterului citit (codul acestuia).

Orice funcţie care interacţionează cu mediul extern (în acest caz, intrarea furnizată de utilizator) este supusă la potenţiale *situaţii de eroare* pe care nu le poate controla, şi trebuie să poată raporta aceste erori. La citire, e posibil ca intrarea să nu poată furniza un caracter: de exemplu, dacă utilizatorul a încheiat introducerea datelor, sau dacă programul este rulat cu intrarea standard redirectată, citind datele dintr-un fişier în locul tastaturii, şi s-a ajuns cu citirea la sfârşitul fişierului.

În caz de eroare, funcţia `getchar` returnează o valoare specială identificată prin numele `EOF` (*end of file*). Valoarea `EOF` e specificată de standard ca un întreg negativ (pentru a fi diferită de valorile caracterelor, care sunt nenegative), şi e definită în fişierul antet `stdio.h` care conţine şi declaraţia funcţiei `getchar`, şi a celorlalte funcţii standard de intrare/ieşire (de ex. `printf`).

În multe implementări, valoarea `EOF` este `-1`, dar nu trebuie să ne bazăm pe această presupunere. Ne referim la această valoare specială doar prin nume.

3.4 Funcţii pentru clasificarea caracterelor

În multe probleme dorim să prelucrăm doar caractere de un anumit fel. De exemplu, dacă citim de la intrare un cuvânt, caracter cu caracter, trebuie să ne oprim când caracterul citit nu mai este o literă.

Limbajul C oferă o serie de funcţii care testează din ce categorie face parte un caracter. Toate aceste funcţii sunt declarate în fişierul antet `ctype.h`. Ele iau ca parametru un întreg (codul caracterului de testat), şi returnează un întreg care e: nenul dacă caracterul se încadrează în categoria dorită (corespunzătoare cu numele funcţiei), sau zero în caz contrar. De exemplu, funcţia

```
int isdigit(int c);           // declarata in ctype.h
```

returnează o valoare nenulă dacă parametru este un caracter de la `'0'` la `'9'` şi zero pentru orice alt caracter.

Dăm ca exemplu o funcţie care returnează valoarea (de la 0 la 9) a unui caracter cifră zecimală, sau -1 (ca şi cod de eroare) pentru orice alt caracter:

```
int digitvalue(int c)
{
  return isdigit(c) ? c - '0' : -1;
}
```

Expresia `c - '0'` numără câte cifre sunt între `c` şi `0` în tabela de caractere, ceea ce e tocmai valoarea cifrei `c` (de exemplu, `'7' - '0' == 7`). Este tocmai calculul invers celui făcut în funcţia `xdigit`.

Acest exemplu necesită o clarificare privind valorile logice în limbajul C. Ca şi condiţie pentru operatorul `? :` am folosit până acum doar expresii (cum ar fi comparaţiile) care pot da rezultatele logice *adevărat* sau *fals*. În limbajul C, nu există tip special pentru valori logice. Operatorii care *furnizează* valori logice (cum ar fi cei relaţionali şi de comparaţie, `<`, `!=`, etc.) produc de fapt o valoare întreagă: 1 pentru rezultat adevărat şi 0 pentru fals. Acolo unde se *solicită* o valoare logică (de exemplu în expresia condiţională), se poate folosi mai general *orice* expresie numerică: orice valoare *nenulă* (nu doar 1) semnifică *adevărat*, iar valoarea 0 înseamnă *fals*.

Lista funcţiilor declarate în `ctype.h` cuprinde:

```
int isalnum(int c);   // este litera sau cifra?
int isalpha(int c);   // este litera? (mare sau mica)
int isdigit(int c);   // este cifra?
int islower(int c);   // este litera mica?
int isspace(int c);   // spatiu alb? (spatiu + \t\n\v\f\r)
int isupper(int c);   // este litera mare?
int isxdigit(int c);  // este cifra hexa? (0-9, A-F, a-f)

int iscntrl(int c);   // este caracter de control?
int isgraph(int c);   // caracter tiparibil? (excl. spatiu)
int isprint(int c);   // caracter tiparibil? (incl. spatiu)
int ispunct(int c);   // este semn de punctuatie?
```

Toate aceste funcţii acceptă ca parametru orice întreg din domeniul de valori care reprezintă caractere, sau valoarea `EOF` (e nedefinit însă comportamentul dacă parametrul depăşeşte acest domeniu de valori). În consecinţă, rezultatul returnat de `getchar` poate fi testat direct cu aceste funcţii fără a trebui verificată separat condiţia de eroare/sfârşit de fişier. Evident, `EOF`, nefiind caracter nu intră în nicicare din clasele de mai sus.

Atenţie! Funcţiile de mai sus au ca rezultat 0 sau o valoare nenulă, nu 0 sau 1 ca şi rezultatul comparaţiilor. Ele se folosesc *direct* în condiţii: `isalnum(...) ? ...` după cum sugerează şi numele (este caracter alfanumeric?) şi *nu* comparând cu 1: `isalnum(...) == 1 ? ...` deoarece rezultatul *adevărat* ar putea să fie reprezentat prin altă valoare nenulă.

Se recomandă folosirea acestor funcţii standard în locul testelor bazate pe compararea valorii caracterelor (cum ar fi verificarea că un caracter e literă mare prin încadrarea între `'A'` şi `'Z'`). Un motiv e eficienţa implementării (discutată ulterior), dar mai importantă e corectitudinea. Funcţiile din `ctype.h` sunt concepute să ţină cont de definirea diferită a alfabetului de litere în funcţie de mediul internaţionalizat în care e rulat programul, pe când comparaţia limitată la literele latine ar putea da erori prin omisiune în acest caz.

Antetul `ctype.h` declară şi *funcţii de conversie* între litere mici şi mari:

```
int tolower(int c);   // returnează litera mica corespondenta
int toupper(int c);   // returnează litera mare corespondenta
```

Dacă parametrul este literă mare (resp. mică) şi are în alfabetul mediului de rulare un corespondent literă mică (resp. mare), se returnează acesta, altfel se returnează chiar valoarea neschimbată a argumentului.

3.5 Citirea unui număr

Scriem o funcţie care citeşte de la intrare un număr natural reprezentat în baza 10. Calculăm valoarea numărului pas cu pas, odată cu citirea fiecărei cifre. Dacă numărul are n cifre $c_1, c_2, \ldots, c_n$ (de la stânga la dreapta) putem forma succesiv numerele: $r_0 = 0$, $r_k = r_{k-1} \cdot 10 + c_k$, pentru $k = 1, 2, \ldots, n$. De exemplu, pentru numărul 247 cu $n = 3$ cifre, avem: $r_0 = 0$, $r_1 = 0 \cdot 10 + 2 = 2$, $r_2 = 2 \cdot 10 + 4 = 24$, $r_3 = 24 \cdot 10 + 7 = 247$ care e valoarea dorită.

Transcriem relaţia recursivă de mai sus într-o funcţie care ia doi parametri: rezultatul parţial deja calculat (r_{k-1}) şi ultimul caracter citit (care dă cifra c_k). Ramura de oprire corespunde unui caracter care nu e cifră, caz în care se returnează rezultatul parţial deja calculat. Altfel se returnează rezultatul apelului recursiv cu rezultatul parţial actualizat cu cifra citită, şi un caracter nou.

Dispunem deja de funcţia `getchar` pentru citirea unui caracter, şi am văzut că valoarea numerică reprezentată de caracterul cifră zecimală `c` este `c - '0'` (deplasamentul faţă de cifra `0` în tabela de caracatere). Cum nu ştim dinainte câte cifre va avea numărul introdus de utilizator, folosim funcţia `isdigit` pentru a testa când caracterul citit nu mai este cifră şi a ne opri.

```
unsigned readnat_rc(unsigned r, int c)
{
  return isdigit(c) ? readnat_rc(r*10 + (c-'0'), getchar()) : r;
}
```

Pentru utilizare, această funcţie trebuie apelată cu valori iniţiale corespunzătoare pentru parametri. Rezultatul parţial iniţial este $r_0 = 0$, iar primul caracter e citit cu `getchar`, deci apelul se scrie `readnat_rc(0, getchar())` .

Din punct de vedere al utilizatorului, citirea unui număr natural de la intrarea standard nu ar avea nevoie de informaţii suplimentare: funcţia ar trebui să fie fără parametri, la fel ca şi `getchar` pentru citirea unui caracter. Scriem de aceea o funcţie care încapsulează apelul cu valorile iniţiale necesare stabilite şi este astfel mai natural şi uşor de folosit:

```
unsigned readnat(void) { return readnat_rc(0, getchar()); }
```

Continuăm implementând citirea unui întreg precedat opţional de semn.

```
int readint_c(int c)    // tine cont de semn; c: primul caracter
{
  return c == '-' ? - readnat() :
         c == '+' ? readnat() : readnat_rc(0, c);
}
```

Dacă primul caracter citit este semn, funcţia apelează `readnat`, fără parametri, care va începe citirea cu un caracter nou. Altfel, se apelează `readnat_rc` cu valoarea iniţială 0, şi caracterul tocmai citit. Şi pentru această funcţie scriem o versiune fără parametri, care citeşte singură primul caracter necesar:

```
int readint(void) { return readint_c(getchar()); }
```

3.6 Refacerea unui caracter din intrare

Funcţiile scrise mai sus se opresc din citirea numărului la primul caracter necorespunzător. Acesta este însă consumat din intrare, şi se pierde, nemaifiind disponibil după revenirea din funcţie. Acest aspect nedorit trebuie evitat; funcţia ar trebui să citească doar cât are nevoie, fără a afecta citirile ulterioare.

Limbajul C pune la dispoziţie o funcţie standard pentru a "pune înapoi" un caracter care a fost citit dintr-un flux de intrare. Declaraţia ei este:

```
int ungetc(int c, FILE *stream);      // in stdio.h
```

Vom prezenta ulterior tipul `FILE *` al parametrului 2, reprezentând un fişier, în acest caz de intrare. Pentru moment dorim să punem înapoi un caracter în *intrarea standard*, de unde am efectuat citirea. Pentru aceasta, argumentul al doilea va fi `stdin`, un identificator declarat în `stdio.h` care reprezintă fişierul standard de intrare. Deci, pentru a pune înapoi în intrarea standard un caracter, reprezentat prin valoarea sa întreagă `int c`, vom folosi apelul

```
ungetc(c, stdin);     // pune inapoi caracterul c
```

Funcţia returnează valoarea caracterului dat în caz de succes, sau `EOF` pentru a semnala eroare. Caracterul pus înapoi devine următorul care va fi citit din fluxul de intrare. Limbajul C garantează că *un* caracter poate fi pus înapoi în acest fel înainte de a fi consumat printr-un apel de citire, cum ar fi `getchar()`.

3.7 Secvenţierea pentru expresii

Reexaminând funcţia de citire a unui număr natural, caracterul trebuie pus înapoi în intrare atunci când nu e cifră (`isdigit(c)` indică "fals"), deci pe a doua ramură a expresiei condiţionale. În scrierea curentă, pe această ramură expresia ia valoarea rezultatului parţial `r` primit ca parametru.

```
isdigit(c) ? readnat_rc(r*10 + (c-'0'), getchar()) : r;
```

Cu elementele de limbaj prezentate pâna acum, în locul unei expresii nu se pot efectua *două* acţiuni: întâi apelul funcţiei `ungetc` şi apoi să-i dăm expresiei valoarea `r`. Limbajul C permite însă execuţia succesivă a două *instrucţiuni*. E normal să existe noţiunea de *secvenţiere* şi pentru expresii, mai ales că instrucţiunea cea mai simplă este chiar o evaluare de expresie.

În C, evaluarea secvenţială a două expresii se face cu operatorul virgulă ,

expresie1 , *expresie2*

Se evaluează întâi *expresie1*, iar rezultatul e ignorat: construcţia este deci utilă dacă expresia are un *efect lateral* (un apel de intrare-ieşire sau o atribuire). Rezultatul şi tipul întregii expresii se obţin evaluând *expresie2*.

Operatorul virgulă are cea mai scăzută precedenţă dintre operatori; pentru a-l folosi în ultimul operand al expresiei condiţionale grupăm deci expresia între paranteze. Rescriem astfel citirea unui număr natural obţinând o funcţie care nu consumă caractere suplimentare din intrare.

```
unsigned readnat_rc(unsigned r, int c)
{
  return isdigit(c) ? readnat_rc(r*10 + (c-'0'), getchar())
                    : (ungetc(c, stdin), r);
}
```