„*Theory without practice is useless*; *practice without theory is blind*”
Roger Bacon

Acest manual este structurat astfel încît elementele limbajului C să fie prezentate într-o manieră unitară. Primul capitol face o scurtă introducere şi prezintă şase programe C.

Următoarele cinci capitole descriu elementele primare ale limbajului C: tipurile fundamentale de date, definiții de constante şi variabile, instrucțiuni, funcții. Capitolul şapte descrie strategii de organizare a activității de programare pentru elaborarea programelor mari, compuse din mai multe module. Tot în acest capitol sînt prezentate şi directivele de compilare cele mai folosite. Capitolele opt şi nouă descriu elemente avansate ale limbajului C: pointeri şi structuri.

Capitolele zece şi unsprezece trec în revistă funcțiile cele mai des utilizate definite în biblioteca standard, împreună cu cîteva programe demonstrative. Am selectat doar funcțiile definite de mai multe standarde (în primul rînd ANSI C), pentru a garanta o portabilitate cît mai mare.

Acest manual a fost conceput pentru a servi ca document care să poată fi consultat de programatori în elaborarea proiectelor, şi nu pentru a fi memorat. Manualul nu este o prezentare exhaustivă a limbajului C; am selectat strictul necesar care să permită unui programator să elaboreze programe eficiente şi uşor de întreținut.

Deoarece avem convingerea că cea mai bună explicație este un program funcțional, exemplele din acest manual pot fi rulate pe orice mediu de programare C şi sub orice sistem de operare.

Ca o ultimă observație amintim recomandarea făcută de înşişi creatorii limbajului: cea mai bună metodă de învățare este practica.

„*Dacă constructorii ar fi construit clădiri aşa cum scriu programatorii programe, atunci prima ciocănitoare care ar fi venit ar fi distrus civilizația*”
(din Legile lui Murphy)

1. Generalităţi asupra limbajului C

1.1. Introducere

Limbajul C a fost creat la începutul anilor '70 de către Brian W Kernigham şi Dennis M Ritchie de la Bell Laboratories New Jersey, fiind iniţial destinat scrierii unei părţi din sistemul de operare Unix. Lucrarea „The C Programming Language" a celor doi autori, apărută în mai multe versiuni, a rămas cartea de referinţă în domeniu, impunînd un standard minimal pentru orice implementare.

Caracteristicile distinctive ale limbajului au fost clar definite de la început, ele păstrîndu-se în toate dezvoltările ulterioare:
– portabilitate maximă;
– structurare;
– posibilitatea efectuării operaţiilor la nivelul maşinii cu păstrarea caracteristicilor unui limbaj evoluat.

Limbajul C este un limbaj de programare universal, caracterizat printr-o exprimare concisă, un control modern al fluxului execuţiei, structuri de date, şi un bogat set de operatori.

Limbajul C nu este un limbaj de „nivel foarte înalt" şi nu este specializat pentru un anumit domeniu de aplicaţii. Absenţa restricţiilor şi generalitatea sa îl fac un limbaj mai convenabil şi mai eficient decît multe alte limbaje mai puternice.

Limbajul C permite scrierea de programe bine structurate, datorită construcţiilor sale de control al fluxului: grupări de instrucţiuni, luări de decizii (**`if`**), cicluri cu testul de terminare înaintea ciclului (**`while`**, **`for`**) sau după ciclu (**`do`**) şi selecţia unui caz dintr-o mulţime de cazuri (**`switch`**).

Limbajul C permite lucrul cu pointeri şi are o aritmetică de adrese puternică. Permite de asemenea definirea de către programator a unor tipuri structurate de date, care pot fi oricît de complexe.

Limbajul C nu are operaţii care prelucrează direct obiectele compuse cum sînt şirurile de caractere, mulţimile, listele sau masivele, considerate fiecare ca o entitate. Limbajul C nu prezintă

facilități de alocare a memoriei altele decît definiția statică sau disciplina de stivă relativă la variabilele locale ale funcțiilor. În sfîrşit, limbajul C nu are facilități de intrare / ieşire şi nici metode directe de acces la fişiere. Toate aceste mecanisme de nivel înalt sînt realizate prin funcții explicite.

Deşi limbajul C este, aşadar, un limbaj de nivel relativ scăzut, el este un limbaj agreabil, expresiv şi elastic, care se pretează la o gamă largă de programe. C este un limbaj restrîns şi se învață relativ uşor, iar subtilitățile se rețin pe măsură ce creşte experiența în programare.

1.2. Reprezentarea valorilor numerice

O valoare întreagă se reprezintă în baza doi:

$105_{(10)} = 01101001_{(2)}$

$32750_{(10)} = 0111111111101110_{(2)}$

O valoare întreagă negativă se reprezintă în complement față de doi astfel: dacă pentru reprezentare se folosesc w biți, $-a$ ($0<a<2^{w-1}$) se reprezintă ca şi valoarea 2^w-a, din care se păstrează cele mai puțin semnificative w poziții. Astfel operațiile de adunare (cele mai frecvente operații elementare) se efectuează într-un timp foarte scurt, fără a fi nevoie de alte operații auxiliare.

$-105_{(10)} = 10010111_{(2)}$

$32750_{(10)} = 1000000000010010_{(2)}$

Codificarea ASCII a caracterelor foloseşte 8 biți (un octet):

– codul caracterului spațiu este 32;
– codurile cifrelor sînt valori între 48 şi 57;
– codurile literelor mari sînt valori între 65 şi 90;
– codurile literelor mici sînt valori între 97 şi 122.

Există două convenții de memorare a valorilor întregi care necesită mai mulți octeți:

– „Big Endian", octeții apar în ordinea cifrelor semnificative: valoarea $127AC450_{(16)}$ se memorează astfel: 12, 7A, C4, 50;

– „Little Endian", octeții apar în ordine inversă: valoarea $127AC450_{(16)}$ se memorează astfel: 50, C4, 7A, 12.

Calculatoarele bazate pe procesoare Intel folosesc a doua convenție de reprezentare. Detalii despre acest subiect se află la:

http://www.cs.umass.edu/~verts/cs32/endian.html

O valoare reală nenulă se reprezintă normalizată în virgulă mobilă, descompusă în două elemente: o parte subunitară în intervalul [0.5,1) şi o putere a bazei doi. Pentru semn se foloseşte o poziţie distinctă (0 pozitiv, 1 negativ).

***Exemplu*:** $4.625 = 100.101_{(2)} = 0.100101_{(2)} \cdot 2^3$

Numărul de poziţii rezervate pentru partea fracţionară, respectiv pentru exponent depind de precizia aleasă, care poate fi simplă, dublă sau extinsă.

1.3. Primele programe

În această secţiune sînt prezentate şi explicate şase programe cu scopul de a asigura un suport de bază pentru prezentările din capitolele următoare.

Prin tradiţie primul program C este un mic exemplu din lucrarea devenită clasică – „*The C programming language*", de Brian W Kernigham şi Dennis M Ritchie.

```
#include <stdio.h>
int main() {
printf("Hello, world\n");
return 0;
}
```

Acest program afişează un mesaj de salut.

Prima linie indică faptul că se folosesc funcţii de intrare / ieşire, şi descrierea modului de utilizare (numele, tipul argumentelor, tipul valorii returnate etc) a acestora se află în fişierul cu numele **`stdio.h`** .

A doua linie defineşte funcţia **`main`** care va conţine instrucţiunile programului. În acest caz singura instrucţiune este un apel al funcţiei **`printf`** care afişează un mesaj la terminal. Mesajul este dat între ghilimele şi se termină cu un caracter special new-line (**`\n`**).

Instrucţiunea **`return`** predă controlul sistemului de operare la terminarea programului şi comunică acestuia codul 0 pentru

terminare. Prin convenție această valoare semnifică terminarea normală a programului – adică nu au apărut erori în prelucrarea datelor.

Corpul funcției **main** apare între acolade. Orice program C trebuie să aibă o funcție **main**.

Al doilea program aşteaptă de la terminal introducerea unor numere întregi nenule şi determină suma lor. În momentul în care se introduce o valoare zero, programul afişează suma calculată.

```
#include <stdio.h>
int main() {
int s,n;
s = 0;
do {
   scanf("%d",&n);
   s += n;
   } while (n!=0);
printf("%d\n",s);
return 0;
}
```

În cadrul funcției **main** se definesc două variabile **s** şi **n** care vor memora valori întregi. Variabila **s** (care va păstra suma numerelor introduse) este inițializată cu valoarea 0.

În continuare se repetă (**do**) o secvență de două instrucțiuni, prima fiind o operație de intrare şi a doua o adunare.

Primul argument al funcției **scanf** – formatul de introducere **"%d"** – indică faptul că se aşteaptă introducerea unei valori întregi în format zecimal de la terminal (consolă). Al doilea argument indică *unde* se va depune în memorie valoarea citită; de aceea este necesar să se precizeze *adresa* variabilei **n** (cu ajutorul operatorului **&**).

În a doua instrucțiune la valoarea variabilei **s** se adună valoarea variabilei **n**. Operatorul **+=** are semnificația *adună la*.

Această secvență se repetă (**do**) cît timp (**while**) valoarea introdusă (**n**) este nenulă. Operatorul != are semnificația *diferit de*.

În final funcția **printf** afişează pe terminal *valoarea* variabilei **s** în format zecimal.

Al treilea program aşteaptă de la terminal introducerea unei valori naturale **n**, după care mai aşteaptă introducerea a **n** valori reale (dublă precizie): a_0, a_1, ..., a_{n-1}. În continuare se afişează media aritmetică a acestor valori, calculată în cadrul unei funcții care se defineşte în acest scop.

```
#include <stdio.h>
double media(double v[], int n) {
double s;
int i;
for (s=i=0; i<n; i++)
    s += v[i];
return s/n;
}
int main() {
int n,i;
double a[100];
scanf("%d",&n);
for (i=0; i<n; i++)
    scanf("%lf",&a[i]);
printf("%.3lf\n",produs(a,n));
return 0;
}
```

În cadrul funcției **main** se definesc două variabile **n** şi **i** care vor memora valori întregi. Variabila **n** păstrează numărul actual de valori din masivul **a**. Se definesc de asemenea un masiv unidimensional `a` care va putea memora 100 de valori de tip real (dublă precizie).

Se citeşte de la terminal o valoare `n`. În continuare se introduc valorile reale a_i (**i = 0**, **1**, ..., **n−1**); se observă modul de referire la fiecare element din masiv: **a[i]**. Formatul de introducere **"%lf"** indică faptul că se aşteaptă introducerea unei valori în dublă precizie de la terminal, care va fi depusă la locația de memorie asociată variabilei a_i. În locul construcției **&a[i]** se poate folosi forma echivalentă **a+i**.

Pentru a introduce toate valorile a_i se efectuează un ciclu **for**, în cadrul căruia variabila **i** (care controlează ciclul) ia toate valorile între **0** (inclusiv) şi **n** (exclusiv) cu pasul **1**. Trecerea la următoarea valoare a variabilei **i** se face cu ajutorul operatorului **++**.

Media aritmetică a acestor valori, calculată cu ajutorul funcției **media**, este afişată cu o precizie de trei cifre după punctul zecimal, conform formatului de afişare **".3lf"**.

Să analizăm acum definiția funcției **media**. Aceasta calculează o valoare de tip **double**, operînd asupra unui masiv **v** care are elemente de tip **double**. Cîte elemente are masivul, aflăm din al doilea parametru al funcției: **n**.

Declarația **double v[]**
nu defineşte un masiv: în momentul apelului din **main**, argumentul **v** va indica spre masivul **a** cu care este pus în corespondență.

Declarația **int n**
defineşte o variabilă locală **n**, care în momentul apelului din **main** va primi valoarea argumentului **n**.

Trebuie să subliniem faptul că cele două variabile, deşi au acelaşi nume, sînt distincte: fiecare este *locală* funcției în care este definită. Aceeaşi observație este valabilă şi pentru variabilele **i**.

Ciclul **for** inițializează variabilele **s** şi **i** cu zero. Variabila **s** (de asemenea locală) va memora suma valorilor din masivul **v**. Cu ajutorul variabilei **i** se parcurge masivul **v**, şi valoarea fiecărui element este cumulată în **s**. În final instrucțiunea **return** comunică funcției principale valoarea **s/n**.

Acelaşi program poate fi scris şi altfel, definind o parte din variabile în afara celor două funcții.

```
#include <stdio.h>
int n;
double a[100];
double media() {
double s;
int i;
for (s=i=0; i<n; i++)
    s += a[i];
```

```
return s/n;
}
int main() {
int i;
scanf("%d",&n);
for (i=0; i<n; i++)
    scanf("%lf",&a[i]);
printf("%.3lf\n",produs());
return 0;
}
```

În acest caz ambele funcţii se referă la acelaşi masiv **a** şi la aceeaşi variabilă **n**; spunem că variabilele **a** şi **n** sînt *globale*. Funcţia **media** defineşte două variabile locale: **s** şi **i**, funcţia **main** defineşte o variabilă locală: **i**.

Există o diferenţă importantă între cele două funcţii **media** din cele două programe. Prima definiţie este mai generală, deoarece poate opera pe orice masiv de elemente de tip **double**, avînd oricîte elemente. Dacă, de exemplu, în funcţia principală am avea încă un masiv **b** cu **m** elemente, apelul ar fi: **media(b,m)**.

A doua definiţie nu permite acest lucru. Funcţia principală ar trebui să copieze mai întîi toate elementele masivului **b** în masivul **a**, şi valoarea **m** în variabila **n**. Abia după aceea ar putea fi apelată funcţia **media**. Dar astfel am distruge vechile valori din **a** şi **n**.

Al doilea stil de programare trebuie evitat, deoarece utilizarea funcţiilor este foarte greoaie pentru argumente distincte la apeluri distincte ale acestora. Există o singură justificare pentru acest stil: cînd ştim cu certitudine că o funcţie pe care o definim va opera întotdeauna cu argumente memorate tot timpul în aceleaşi variabile.

Al cincilea program este o ilustrare a unor probleme legate de capacitatea reprezentărilor valorilor de tip întreg şi virgulă mobilă.

```
#include <stdio.h>
int main() {
short k,i;
float a,b,c,u,v,w;
i = 240;  k = i * i;
```

```
printf("%hd\n",k);
a = 60.1;  k = a * 100;
printf("%hd\n",k);
a = 12345679;  b = 12345678;
c = a * a – b * b;
u = a * a;  v = b * b;  w = u - v;
printf("%f %f\n",c,w);
if (c==w) return 0;
   else return 1;
}
```

Variabila **k**, care ar trebui să memoreze valoarea 57600, are tipul întreg scurt (**short**), pentru care domeniul de valori este restrîns la –32768 ÷ 32767. Astfel că valoarea $1110000100000000_{(2)}$ (în zecimal 57600), în reprezentare întreagă cu semn este de fapt –7936.

În continuare variabila **a** primeşte valoarea 60.1, care *nu poate fi reprezentată exact în virgulă mobilă*. De aceea variabila **k** primeşte valoarea 6009 datorită erorilor de aproximare.

Al treilea set de operaţii necesită o analiză mai atentă; explicaţiile sînt valabile pentru programe care rulează pe arhitecturi Intel. Variabila **c**, care ar trebui să memoreze valoarea 24691357 (rezultatul corect), va avea valoarea 24691356, deoarece tipul **float** are rezervate pentru mantisă doar 24 de cifre binare. Rezultatul este foarte apropiat de cel corect deoarece rezultatele intermediare se păstrează în regiştrii coprocesorului matematic cu precizie maximă. Abia la memorare se efectuează trunchierea, de unde rezultă valoarea afişată.

Cu totul altfel stau lucrurile în cazul celui de al patrulea set de operaţii. Aici rezultatele intermediare sînt memorate de fiecare dată cu trunchiere în variabile de tip **float**. În final se calculează şi diferenţa dintre cele două valori trunchiate, de unde rezultă valoarea 16777216.

Înainte de terminare se verifică dacă valorile **c** şi **w** sînt egale. În caz afirmativ se comunică sistemului de operare un cod 0 (terminare normală). În caz contrar se comunică un cod 1 (terminare anormală).

Rulaţi acest program pe diferite sisteme de calcul şi observaţi care este rezultatul.

Ultimul program citeşte cîte o linie de la intrarea standard şi o afişează la ieşirea standard.

```
#include <stdio.h>
char lin[80];
int main() {
while (gets(lin)) puts(lin);
return 0;
}
```

Linia citită cu ajutorul funcției de bibliotecă **gets** este memorată în masivul **lin**, ultimul caracter valid al liniei fiind urmat de o valoare zero, care marchează astfel sfîrşitul şirului. Citirea se încheie atunci cînd intrarea standard nu mai furnizează nimic. Funcția **gets** returnează o valoare care are semnificația *True* dacă intrarea standard a furnizat o linie, sau *False* dacă nu s-a mai furnizat nimic. Dacă linia se termină cu new-line, acest caracter este preluat de la intrarea standard, dar nu este memorat în şirul **lin**.

Am presupus că fiecare linie introdusă are mai puțin de 80 de caractere, altfel programul nu va funcționa corect. De altfel funcția **gets** este folosită în această lucrare doar în scop didactic, pentru simplificarea expunerilor.

Fiecare linie citită de la intrarea standard este afişată cu ajutorul funcției **puts**. Aceasta afişează automat la sfîrşit caracterul new-line.

Dacă programul este rulat în mod obişnuit nu se întîmplă nimic spectaculos: fiecare linie introdusă de la terminal va fi reafişată. Programul poate fi totuşi folosit pentru a afişa conținutul unui fişier text, pentru a crea un fişier text, sau pentru a copia un fişier text. În acest scop programul este rulat în mod linie de comandă astfel:

Prog **<***Fişier-intrare* **>***Fişier-ieşire*

Prog este numele programului executabil, obținut în urma compilării programului sursă. *Fişier-intrare* este numele fişierului al cărui conținut dorim să-l afişăm sau să-l copiem. *Fişier-ieşire* este numele fişierului pe care dorim să-l creăm.

Oricare din cele două argumente poate lipsi. Dacă primul argument este prezent, programul va citi datele din fişierul de intrare

specificat, altfel va aştepta introducerea datelor de la terminal. În al doilea caz trebuie marcată într-un mod special terminarea introducerii, şi acest mod depinde de sistemul de operare sub care se lucrează. Dacă al doilea argument este prezent, programul va scrie datele în fişierul de ieşire specificat, care va fi creat în acest scop. Dacă ieşirea nu este specificată, datele vor fi afişate la terminal.

Observație generală. În cazul fiecărui program, *fiecare variabilă sau funcție este definită înainte de utilizare*, pentru ca în momentul utilizării să existe toate informațiile despre contextul în care poate fi utilizat identificatorul respectiv.

2. Unităţile lexicale ale limbajului C

Setul de caractere al limbajului C este un subset al setului de caractere ASCII, format din:

– 26 litere mici

a b c d e f g h i j k l m n o p q r s t u v w x y z

– 26 litere mari

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

– 10 cifre

0 1 2 3 4 5 6 7 8 9

– 30 simboluri speciale

Blanc ! " # % & ' ( ) * + , – . / : ; < = > ? [\ ] ^ _ ~ { | }

– 6 simboluri negrafice

\n, \t, \b, \r, \f, \a

În limbajul C există şase tipuri de unităţi lexicale: identificatori, cuvinte-cheie, constante, şiruri, operatori şi separatori.

2.1. Identificatori; cuvinte cheie

Un identificator este o succesiune de litere şi cifre dintre care primul caracter este în mod obligatoriu o literă. Se admit şi litere mari şi litere mici dar ele se consideră caractere distincte. Liniuţa de subliniere _ este considerată ca fiind literă.

Cuvintele cheie sînt identificatori rezervaţi limbajului. Ei au o semnificaţie bine determinată şi nu pot fi utilizaţi decît aşa cum cere sintaxa limbajului. Cuvintele cheie se scriu obligatoriu cu litere mici. Cuvintele cheie definite de standardul ANSI sînt:

auto	**do**	**goto**	**signed**	**unsigned**
break	**double**	**if**	**sizeof**	**void**
case	**else**	**int**	**static**	**volatile**
char	**enum**	**long**	**struct**	**while**
const	**extern**	**register**	**switch**	
continue	**float**	**return**	**typedef**	
default	**for**	**short**	**union**	

2.2. Constante

În limbajul C există următoarele tipuri de constante: întreg (zecimal, octal, hexazecimal), întreg lung explicit, flotant, caracter, simbolic.

Constante întregi

O constantă întreagă constă dintr-o succesiune de cifre.

O constantă octală este o constantă întreagă care începe cu `0` (cifra zero), şi este formată cu cifre de la **`0`** la **`7`**.

O constantă hexazecimală este o constantă întreagă precedată de **`0x`** sau **`0X`** (cifra **`0`** şi litera **`x`**). Cifrele hexazecimale includ literele de la **`A`** la **`F`** şi de la **`a`** la **`f`** cu valori de la **`10`** la **`15`**.

În orice alt caz, constanta întreagă este o constantă zecimală.

Exemplu: constanta zecimală **`31`** poate fi scrisă ca **`037`** în octal şi **`0x1f`** sau **`0X1F`** în hexazecimal.

O constantă întreagă este generată pe un cuvînt (doi sau patru octeți, dacă sistemul de calcul este pe 16 sau 32 de biți).

O constantă zecimală a cărei valoare depăşeşte pe cel mai mare întreg cu semn reprezentabil pe un cuvînt scurt (16 biți) se consideră de tip **`long`** şi este generată pe 4 octeți.

O constantă octală sau hexazecimală care depăşeşte pe cel mai mare întreg fără semn reprezentabil pe un cuvînt scurt se consideră de asemenea de tip **`long`**.

O constantă întreagă devine negativă dacă i se aplică operatorul unar de negativare **`-`**.

Constante de tip explicit

O constantă întreagă zecimală, octală sau hexazecimală, urmată imediat de litera **`l`** sau **`L`** este o constantă lungă. Aceasta va fi generată în calculator pe 4 octeți.

***Exemplu*: `123L`**

O constantă întreagă zecimală urmată imediat de litera **u** sau **U** este o constantă de tip întreg fără semn. Litera **u** sau **U** poate fi urmată de litera **l** sau **L**.

***Exemplu*: `123ul`**

Constante flotante

O constantă flotantă constă dintr-o parte întreagă, un punct zecimal, o parte fracționară, litera **e** sau **E**, şi opțional un exponent care este un întreg cu semn. Partea întreagă şi partea fracționară sînt constituite din cîte o succesiune de cifre. Într-o constantă flotantă, atît partea întreagă cît şi partea fracționară pot lipsi, dar nu ambele; de asemenea poate lipsi punctul zecimal sau litera **e** şi exponentul, dar nu deodată (şi punctul şi litera e şi exponentul).

***Exemplu*: `123.456e-7`** sau **`0.12e-3`**

Orice constantă flotantă se consideră a fi în precizie extinsă.

Constante caracter

O constantă caracter constă dintr-un singur caracter scris între apostrofuri, de exemplu **`'x'`**. Valoarea unei constante caracter este valoarea numerică a caracterului, în setul de caractere al calculatorului. De exemplu în setul de caractere ASCII caracterul zero sau **`'0'`** are valoarea **48** în zecimal, total diferită de valoarea numerică zero.

Constantele caracter participă la operațiile aritmetice ca şi oricare alte numere. De exemplu, dacă variabila **c** conține valoarea ASCII a unei cifre, atunci prin instrucțiunea:

```
c = c - '0' ;
```

această valoare se transformă în valoarea efectivă a cifrei.

Anumite caractere negrafice şi caractere grafice **'** (apostrof) şi **** (backslash) pot fi reprezentate ca şi constante caracter cu ajutorul unor *secvențe de evitare*. Acestea oferă de altfel şi un mecanism general pentru reprezentarea caracterelor mai dificil de introdus în calculator şi a oricăror configurații de biți. Aceste secvențe de evitare sînt:

\n new-line	**\r** carriage return	**** backslash
\t tab orizontal	**\f** form feed	**\'** apostrof
\b backspace	**\a** semnal sonor	**\"** ghilimele

ooo configurație de biți precizată în baza 8
\x*hh* configurație de biți precizată în baza 16

Fiecare din aceste secvențe, deşi e formată din mai multe caractere, reprezintă în realitate un singur caracter.

***Exemplu*:** secvența **'\040'** va genera caracterul spațiu.

Constante simbolice

O constantă simbolică este un identificator cu valoare de constantă. Valoarea constantei poate fi orice şir de caractere introdus prin construcția **#define** (capitolul şapte).

Exemplu: **#define MAX 1000**

După întîlnirea acestei construcții compilatorul va înlocui toate aparițiile constantei simbolice **MAX** cu valoarea **1000**.

Numele constantelor simbolice se scriu de obicei cu litere mari (fără a fi obligatoriu).

2.3. Şiruri

Un şir este o succesiune de caractere scrise între ghilimele, de exemplu **"ABCD"**.

Ghilimelele nu fac parte din şir; ele servesc numai pentru delimitarea şirului. Caracterul **"** (ghilimele) poate apărea într-un şir dacă se utilizează secvența de evitare **\"**. În interiorul unui şir pot fi folosite şi alte secvențe de evitare pentru constante caracter, de asemenea poate fi folosit caracterul **** (backslash) la sfîrşitul unui rînd pentru a da posibilitatea continuării unui şir pe mai multe linii, situație în care caracterul **** însuşi va fi ignorat.

Pentru şirul de caractere se mai foloseşte denumirea constantă şir sau constantă de tip şir.

Cînd un şir apare într-un program C, compilatorul creează un masiv de caractere care conține caracterele şirului şi plasează

automat caracterul null (0) la sfîrşitul şirului, astfel ca programele care operează asupra şirurilor să poată detecta sfîrşitul acestora. Această reprezentare înseamnă că, teoretic, nu există o limită a lungimii unui şir, iar programele trebuie să parcurgă şirul, analizîndu-l pentru a-i determina lungimea. Se admit şi şiruri de lungime zero.

Tehnic, un şir este un masiv ale cărui elemente sînt caractere. El are tipul masiv de caractere şi clasa de memorie **`static`** (capitolul trei). Un şir este iniţializat cu caracterele date.

La alocare, memoria fizică cerută este cu un octet mai mare decît numărul de caractere scrise între ghilimele, datorită adăugării automate a caracterului null la sfîrşitul fiecărui şir.

Exemplu. Secvenţa următoare determină lungimea şirului de caractere `s`, excluzînd caracterul terminal null.

```
for (n=0; s[i]; n++) ;
```

Atragem atenţia asupra diferenţei dintre o constantă caracter şi un şir care conţine un singur caracter. **`"x"`** nu este acelaşi lucru cu **`'x'`**. **`'x'`** este un singur caracter, folosit pentru a genera pe un octet valoarea numerică a literei *x*, din setul de caractere al calculatorului. **`"x"`** este un şir de caractere, care în calculator se reprezintă pe doi octeţi, dintre care primul conţine un caracter (litera *x*), iar al doilea caracterul null care indică sfîrşitul de şir.

2.4. Operatori

Limbajul C prezintă un număr mare de operatori care pot fi clasificaţi după diverse criterii. Există operatori unari, binari şi ternari, operatori aritmetici, logici, operatori pe biţi etc. Capitolul patru este rezervat în exclusivitate descrierii operatorilor definiţi în limbajul C.

2.5. Separatori

Un separator este un caracter sau un şir de caractere care separă unităţile lexicale într-un program scris în C.

Separatorul cel mai frecvent este aşa numitul spaţiu alb (blanc) care conţine unul sau mai multe spaţii, tab-uri, new-line-uri sau comentarii.

Aceste construcţii sînt eliminate în faza de analiza lexicală a compilării.

Dăm mai jos lista separatorilor admişi în limbajul C.

`( )` Parantezele mici – încadrează lista de argumente ale unei funcţii sau delimitează anumite părţi în cadrul expresiilor aritmetice etc

`{ }` Acoladele – încadrează instrucţiunile compuse, care constituie corpul unor instrucţiuni sau corpul funcţiilor

`[ ]` Parantezele mari – încadrează dimensiunile de masiv sau indicii elementelor de masiv

`" "` Ghilimelele – încadrează un şir de caractere

`' '` Apostrofurile – încadrează un singur caracter sau o secvenţă de evitare

`;` Punct şi virgula – termină o instrucţiune

`/*` Slash asterisc – început de comentariu

`*/` Asterisc slash – sfîrşit de comentariu

Un comentariu este un şir de caractere care începe cu caracterele `/*` şi se termină cu caracterele `*/`.

Un comentariu poate să apară oriunde într-un program unde poate apărea un blanc şi are rol de separator; el nu influenţează cu nimic semnificaţia programului, scopul lui fiind doar o documentare a acestuia.

Nu se admit comentarii imbricate.

3. Variabile

Ca şi constantele, variabilele sînt elemente de bază cu care operează un program scris în C. O variabilă este un obiect de programare căruia i se atribuie un nume şi i se asociază o zonă de memorie.

Variabilele se deosebesc după nume şi pot primi diferite valori. Numele variabilelor sînt identificatori. Numele de variabile se scriu de obicei cu litere mici (fără a fi obligatoriu).

În limbajul C, variabilele sînt caracterizate prin două atribute: clasă de memorie şi tip. Acestea îi sînt atribuite unei variabile prin intermediul unei declaraţii. Declaraţiile enumeră variabilele care urmează a fi folosite, stabilesc clasa de memorie, tipul variabilelor şi eventual valorile iniţiale. Sintaxa unei declaraţii este:

clasă tip listă-variabile ;

Lista de variabile poate avea un element sau mai multe, în al doilea caz ele fiind separate prin virgulă.

Clasa de memorie precizează care funcţii pot vedea variabilele declarate în cadrul unui modul:

– doar funcţia / blocul unde acestea sînt definite: variabile automatice şi variabile statice interne;

– toate funcţiile din cadrul modulului care urmează declaraţiei: variabile globale statice;

– toate funcţiile din toate modulele care cunosc declaraţia: variabile globale externe.

Clasa de memorie precizează în plus şi durata de viaţă a variabilelor declarate. Variabilele automatice există doar pe durata execuţiei funcţiei sau blocului unde au fost definite. Variabilele statice şi externe există pe toată durata execuţiei programului.

Tipul unei variabile precizează domeniul de valori pe care le poate lua şi operaţiile care se pot efectua cu aceste valori.

Declaraţiile se folosesc pentru a specifica interpretarea pe care compilatorul trebuie să o dea fiecărui identificator, şi pot apărea în

afara oricărei funcţii sau la începutul unei funcţii înaintea oricărei instrucţiuni. Nu orice declaraţie rezervă şi memorie pentru un anumit identificator, de aceea deosebim:

– declaraţia de definiţie a unei variabile, care creează variabila şi i se alocă memorie;

– declaraţia de utilizare a unei variabile, care doar anunţă proprietăţile variabilei care urmează a fi folosită.

3.1. Clase de memorie

Limbajul C prezintă patru clase de memorie: automatică, externă, statică, registru.

Variabile automatice

Variabilele automatice sînt variabile locale fiecărui bloc sau funcţii. Ele se declară prin specificatorul de clasă de memorie **`auto`** sau implicit prin context. O variabilă care apare în corpul unei funcţii sau al unui bloc pentru care nu s-a făcut nici o declaraţie de clasă de memorie se consideră implicit de clasă **`auto`**.

O variabilă **`auto`** este actualizată la fiecare intrare în bloc şi se distruge în momentul cînd controlul a părăsit blocul. Ele nu îşi reţin valorile de la un apel la altul al funcţiei sau blocului şi trebuie iniţializate la fiecare intrare. Dacă nu sînt iniţializate, conţin valori reziduale. Nici o funcţie nu are acces la variabilele **`auto`** din altă funcţie. În funcţii diferite pot exista variabile locale cu aceleaşi nume, fără ca variabilele să aibă vreo legătură între ele.

Exemple. În fiecare program din primul capitol, variabilele locale sînt automatice.

Variabile registru

O variabilă registru se declară prin specificatorul de clasă de memorie **`register`**. Ca şi variabilele **`auto`** ele sînt locale unui bloc sau funcţii şi valorile lor se pierd la ieşirea din blocul sau funcţia respectivă. Variabilele declarate **`register`** indică compilatorului că

variabilele respective vor fi folosite foarte des. Dacă este posibil, variabilele **register** vor li plasate de către compilator în regiştrii rapizi ai calculatorului, ceea ce conduce la programe mai compacte şi mai rapide.

Variabile **register** pot fi numai variabilele automatice sau parametrii formali ai unei funcţii. Practic există cîteva restricţii asupra variabilelor **register** care reflectă realitatea hardware-ului de bază. Astfel:

– numai cîteva variabile din fiecare funcţie pot fi păstrate în regiştri (de obicei două sau trei); declaraţia **register** este ignorată pentru celelalte variabile;

– numai tipurile de date întregi şi **pointer** sînt admise;

– nu este posibilă referirea la adresa unei variabile **register**.

Variabile statice

Variabilele statice se declară prin specificatorul de clasă de memorie **static**. Aceste variabile sînt la rîndul lor de două feluri: interne şi externe.

Variabilele statice interne sînt locale unei funcţii şi se definesc în interiorul unei funcţii; spre deosebire de variabilele **auto**, ele îşi păstrează valorile tot timpul execuţiei programului. Variabilele statice interne nu sînt create şi distruse de fiecare dată cînd funcţia este activată sau părăsită; ele oferă în cadrul unei funcţii o memorie particulară permanentă pentru funcţia respectivă.

Alte funcţii nu au acces la variabilele statice interne proprii unei funcţii.

Ele pot fi declarate şi implicit prin context; de exemplu şirurile de caractere care apar în interiorul unei funcţii – cum ar fi argumentele funcţiei **printf** – sînt variabile statice interne.

Variabilele statice externe se definesc în afara oricărei funcţii şi orice funcţie are acces la ele. Aceste variabile sînt însă globale numai pentru fişierul sursă în care ele au fost definite. Nu sînt recunoscute în alte fişiere.

În concluzie, variabila statică este externă dacă este definită în afara oricărei funcţii şi este static internă dacă este definită în interiorul unei funcţii.

În general, funcţiile sînt considerate obiecte externe. Există însă şi posibilitatea să declarăm o funcţie de clasă **`static`**. Aceasta face ca numele funcţiei să nu fie recunoscut în afara fişierului în care a fost declarată.

În secţiunea rezervată problemei iniţializărilor de la sfîrşitul acestui capitol vom da un exemplu care ilustrează comportamentul variabilelor automatice şi statice.

Variabile externe

Variabilele externe sînt variabile cu caracter global. Ele se definesc în afara oricărei funcţii şi pot fi apelate prin nume din oricare modul (fişier sursă) care intră în alcătuirea programului.

În declaraţia de definiţie aceste variabile nu necesită specificarea nici unei clase de memorie. La întîlnirea unei definiţii de variabilă externă compilatorul alocă şi memorie pentru această variabilă.

Într-un fişier sursă domeniul de definiţie şi acţiune al unei variabile externe este de la locul de declaraţie pînă la sfîrşitul fişierului. Aceste variabile există şi îşi păstrează valorile de-a lungul execuţiei întregului program.

Pentru ca o funcţie să poată utiliza o variabilă externă, numele variabilei trebuie făcut cunoscut funcţiei printr-o declaraţie.

Detalii despre utilizarea variabilelor externe vor fi date în capitolul şapte.

3.2. Tipuri de variabile

Limbajul C admite numai cîteva tipuri fundamentale de variabile: caracter, întreg, virgulă mobilă.

Pentru fiecare din tipurile prezentate în continuare se precizează şi descriptorii de format cei mai utilizaţi în cadrul funcţiilor de intrare / ieşire. În capitolul zece va fi detaliată tema formatării datelor de intrare şi de ieşire.

Tipul caracter

O variabilă de tip caracter se declară prin specificatorul de tip **char**. Zona de memorie alocată unei variabile de tip **char** este de un octet. Ea este suficient de mare pentru a putea memora orice caracter al setului de caractere implementate pe calculator.

Dacă un caracter din setul de caractere este memorat într-o variabilă de tip **char**, atunci valoarea sa este egală cu codul întreg al caracterului respectiv. Şi alte cantităţi pot fi memorate în variabile de tip **char**, dar implementarea este dependentă de sistemul de calcul.

Domeniul valorilor variabilelor caracter este între **-128** şi **127**. Caracterele setului ASCII sînt toate pozitive, dar o constantă caracter specificată printr-o secvenţă de evitare poate fi şi negativă, de exemplu **'\377'** are valoarea **-1**. Acest lucru se întîmplă atunci cînd această constantă apare într-o expresie, moment în care se converteşte la tipul **int** prin extensia bitului cel mai din stînga din octet (datorită modului de funcţionare a instrucţiunilor calculatorului).

Domeniul valorilor variabilelor caracter fără semn (**unsigned char**) este între **0** şi **255**.

Descriptori de format:

– **%c** pentru o variabilă sau o valoare de tip **char**;
– **%s** pentru o variabilă sau o expresie de tip şir de caractere.

Tipul întreg

Variabilele întregi pozitive sau negative pot fi declarate prin specificatorul de tip **int**. Zona de memorie alocată unei variabile întregi poate fi de cel mult trei dimensiuni.

Relaţii despre dimensiune sînt furnizate de calificatorii **short**, **long** şi **unsigned**, care pot fi aplicaţi tipului **int**.

Calificatorul **short** se referă totdeauna la numărul minim de octeţi pe care se reprezintă un întreg, de obicei 2.

Calificatorul **long** se referă la numărul maxim de octeţi pe care poate fi reprezentat un întreg, de obicei 4.

Tipul **int** are dimensiunea naturală sugerată de sistemul de calcul. Domeniul numerelor întregi reprezentabile în maşină depinde de sistemul de calcul: un întreg poate lua valori între **-32768** şi **32767** (sisteme de calcul pe 16 biţi) sau între **-2147483648** şi **2147483647** (sisteme de calcul pe 32 de biţi).

Calificatorul **unsigned** alături de declaraţia de tip **int** determină ca valorile variabilelor astfel declarate să fie considerate întregi fără semn. Un întreg fără semn poate lua valori între **0** şi **65535** (sisteme de calcul pe 16 biţi) sau între **0** şi **4294967295** (sisteme de calcul pe 32 de biţi).

Valorile de tip **unsigned** respectă legile aritmeticii modulo 2^w, unde *w* este numărul de biţi din reprezentarea unei variabile de tip **int**. Numerele de tipul **unsigned** sînt totdeauna pozitive.

Declaraţiile pentru calificatori sînt de forma:

```
short int x;
long int y;
unsigned int z;
unsigned long int u;
```

Cuvîntul **int** poate fi omis în aceste situaţii.

Descriptori de format:

– **%d** pentru o variabilă sau o valoare de tip **int**;
– **%u** pentru o variabilă sau o valoare de tip **unsigned**;
– **%ld** pentru o variabilă sau o valoare de tip **long**;
– **%lu** pentru o variabilă sau o valoare de tip **unsigned long**;
– **%hd** pentru o variabilă sau o valoare de tip **short**;
– **%hu** pentru o variabilă sau o valoare de tip **unsigned short**.

Tipul flotant (virgulă mobilă)

Variabilele flotante pot fi în simplă precizie şi atunci se declară prin specificatorul de tip **float** sau în dublă precizie şi atunci se declară prin specificatorul **double**. Majoritatea sistemelor de calcul admit şi reprezentarea în precizie extinsă; o variabilă în precizie extinsă se declară prin specificatorul **long double**.

Descriptori de format:

– **`%f`** pentru o variabilă sau o valoare de tip **`float`**;
– **`%lf`** pentru o variabilă sau o valoare de tip **`double`**;
– **`%Lf`** pentru o variabilă sau o valoare de tip **`long double`**.

Tipuri derivate

În afară de tipurile aritmetice fundamentale, există, în principiu, o clasă infinită de tipuri derivate, construite din tipurile fundamentale în următoarele moduri:

– *masive de T* pentru masive de obiecte de un tip dat *T*, unde *T* este unul dintre tipurile admise;
– *funcţii care returnează T* pentru funcţii care returnează obiecte de un tip dat *T* ;
– *pointer la T* pentru pointeri la obiecte de un tip dat *T* ;
– *structuri* pentru un şir de obiecte de tipuri diferite;
– *reuniuni* care pot conţine obiecte de tipuri diferite, tratate într-o singură zonă de memorie.

În general aceste metode de construire de noi tipuri de obiecte pot fi aplicate recursiv. Tipurile derivate vor fi detaliate în capitolele opt şi nouă.

3.3. Obiecte şi valori-stînga

Alte două noţiuni folosite în descrierea limbajului C sînt *obiectul* şi *valoarea-stînga*.

Un *obiect* este conţinutul unei zone de memorie.

O *valoare-stînga* este o expresie care se referă la un obiect. Un exemplu evident de *valoare-stînga* este un identificator. Există operatori care produc *valori-stînga*: de exemplu, dacă *E* este o expresie de tip pointer, atunci **E* este o expresie *valoare-stînga* care se referă la obiectul pe care-l indică *E*. Numele *valoare-stînga* (în limba engleză *left value*) a fost sugerat din faptul că în expresia de atribuire *E*1 = *E*2 operandul stîng *E*1 apare în stînga operatorului de atribuire. În paragraful de descriere a operatorilor se va indica dacă

operanzii sînt *valori-stînga* sau dacă rezultatul operației este o *valoare-stînga*.

Exemple.

```
s = a;
```

În partea dreaptă a operatorului de atribuire, **a** reprezintă *valoarea variabilei respective*. Operația **s = *V*;** semnifică: valoarea ***V*** se atribuie variabilei **s**.

```
*p = *q;
```

În partea dreaptă a operatorului de atribuire, ***q** reprezintă *valoarea aflată la adresa indicată de* q. Operația ***p = *V*;** semnifică: valoarea ***V*** se memorează la adresa indicată de **p**.

3.4. Conversii de tip

Un număr mare de operatori pot cauza conversia valorilor unui operand de la un tip la altul. Dacă într-o expresie apar operanzi de diferite tipuri, ei sînt convertiți la un tip comun după un mic număr de reguli. În general se fac automat numai conversiile care an sens, de exemplu: din întreg în flotant, într-o expresie de forma **f+i**. Expresii care nu au sens, ca de exemplu un număr flotant ca indice, nu sînt admise.

Caractere şi întregi

Un caracter poate apărea oriunde unde un întreg este admis. În toate cazurile valoarea caracterului este convertită automat într-un întreg. Deci într-o expresie aritmetică tipul **char** şi **int** pot apărea împreună. Aceasta permite o flexibilitate considerabilă în anumite tipuri de transformări de caractere. Un astfel de exemplu este funcția **atoi** descrisă în capitolul şase care converteşte un şir de cifre în echivalentul lor numeric.

Expresia următoare produce valoarea numerică a caracterului (cifră) memorat în ASCII:

```
s[i] - '0'
```

Atragem atenția că atunci cînd o variabilă de tip **char** este convertită la tipul **int**, se poate produce un întreg negativ, dacă bitul

cel mai din stînga al octetului conține 1. Caracterele din setul de caractere ASCII nu devin niciodată negative, dar anumite configurații de biți memorate în variabile de tip caracter pot apărea ca negative prin extensia la tipul **int**.

Conversia tipului **int** în **char** se face cu pierderea biților de ordin superior.

Întregii de tip **short** sînt convertiți automat la **int**. Conversia întregilor se face cu extensie de semn; întregii sînt totdeauna cantități cu semn.

Un întreg **long** este convertit la un întreg **short** sau **char** prin trunchiere la stînga; surplusul de biți de ordin superior se pierde.

Conversii flotante

Toate operațiile aritmetice în virgulă mobilă se execută în precizie extinsă. Conversia de la virgulă mobilă la întreg se face prin trunchierea părții fracționare. Conversia de la întreg la virgulă mobilă este acceptată.

Întregi fără semn

Într-o expresie în care apar doi operanzi, dintre care unul **unsigned** iar celălalt un întreg de orice alt tip, întregul cu semn este convertit în întreg fără semn şi rezultatul este un întreg fără semn.

Cînd un **int** trece în **unsigned**, valoarea sa este cel mai mic întreg fără semn congruent cu întregul cu semn (modulo 2^{16} sau 2^{32}). Într-o reprezentare la complementul față de 2, conversia este conceptuală, nu există nici o schimbare reală a configurației de biți.

Cînd un întreg fără semn este convertit la **long**, valoarea rezultatului este numeric aceeaşi ca şi a întregului fără semn, astfel conversia nu face altceva decît să adauge zerouri la stînga.

Conversii aritmetice

Dacă un operator aritmetic binar are doi operanzi de tipuri diferite, atunci tipul de nivel mai scăzut este convertit la tipul de

nivel mai înalt înainte de operație. Rezultatul este de tipul de nivel mai înalt. Ierarhia tipurilor este următoarea:

– **char < short < int < long**;

– **float < double < long double**;

– tip întreg cu semn **<** tip întreg fără semn;

– tip întreg **<** tip virgulă mobilă.

Conversii prin atribuire

Conversiile de tip se pot face prin atribuire; valoarea membrului drept este convertită la tipul membrului stîng, care este tipul rezultatului.

Conversii logice

Expresiile relaționale de forma **i<j** şi expresiile logice legate prin operatorii **&&** şi **||** sînt definite ca avînd valoarea 1 dacă sînt adevărate şi 0 dacă sînt false.

Astfel atribuirea:

```
d = (c>='0') && (c<='9');
```

îl face pe **d** egal cu 1 dacă **c** este cifră şi egal cu 0 în caz contrar.

Conversii explicite

Dacă conversiile de pînă aici le-am putea considera implicite, există şi conversii explicite de tipuri pentru orice expresie. Aceste conversii se fac prin construcția specială numită *cast* de forma:

(*nume-tip***)** *expresie*

În această construcție *expresie* este convertită la tipul specificat după regulile precizate mai sus. Mai precis aceasta este echivalentă cu atribuirea expresiei respective unei variabile de un tip specificat, şi această nouă variabilă este apoi folosită în locul întregii expresii. De exemplu, în expresia:

```
sqrt((double)n)
```

se converteşte **n** la **double** înainte de a se transmite funcției **sqrt**. Variabila **n** nu-şi modifică valoarea.

Expresia constantă

O expresie constantă este o expresie care conține numai constante. Aceste expresii sînt evaluate în momentul compilării şi nu în timpul execuției; ele pot fi astfel utilizate în orice loc unde sintaxa cere o constantă, ca de exemplu:

```
#define NMAX 1000
char lin[NMAX+1];
```

3.5. Masive

În limbajul C se pot defini masive unidimensionale, bidimensionale, tridimensionale etc. Un masiv se compune din mai multe elemente de acelaşi tip; un element se identifică prin indice (poziția relativă în masiv), sau prin indici (dacă masivul este multidimensional).

Exemple:

```
char s[100];
int a[10][15];
```

Prima linie declară un şir de caractere: **`s[0]`**, **`s[1]`**, ..., **`s[99]`**. A doua linie declară o matrice de întregi: **`a[0][0]`**, ..., **`a[0][14]`**, **`a[1][0]`**, ..., **`a[1][14]`**, ..., **`a[9][0]`**, ..., **`a[9][14]`**.

Acest subiect va fi detaliat în capitolul opt.

3.6. Inițializări

Într-o declarație se poate specifica o valoare inițială pentru identificatorul care se declară. Inițializatorul este precedat de semnul = şi constă dintr-o expresie (variabile simple) sau o listă de valori incluse în acolade (masive sau structuri).

Toate expresiile dintr-un *inițializator* pentru variabile statice sau externe trebuie să fie expresii constante sau expresii care se reduc la adresa unei variabile declarate anterior. Variabilele de clasă **`auto`** sau **`register`** pot fi inițializate cu expresii oarecare, nu neapărat expresii constante, care implică constante sau variabile declarate anterior sau chiar funcții.

În absența inițializării explicite, variabilele statice şi externe sînt inițializate implicit cu valoarea 0. Variabilele **auto** şi **register** au valori inițiale nedefinite (reziduale).

Pentru variabilele statice şi externe, inițializarea se face o singură dată, în principiu înainte ca programul să înceapă să se execute.

Pentru variabilele **auto** şi **register**, inițializarea este făcută la fiecare intrare în funcție sau bloc.

Problema inițializării masivelor şi masivelor de pointeri este detaliată în capitolul opt. Problema inițializării structurilor este detaliată în capitolul nouă.

```
#include <stdio.h>
int e = 1;
int f() {
int a = 2;
a++;  e++;
return a + e;
}
int g() {
static int s = 2;
s++;  e++;
return s + e;
}
int main() {
int v1,v2,v3,v4;
v1 = f();  v2 = g();
v3 = f();  v4 = g();
printf("%d %d %d %d %d\n",v1,v2,v3,v4,e);
return 0;
}
```

Să executăm pas cu pas acest program.

v1 = f(): a = 3; e = 2; v1 = 5;

v2 = g(): s = 3; e = 3; v2 = 6;

v3 = f(): a = 3; e = 4; v3 = 7;

variabila **a** este reinițializată la intrarea în funcția **f**;

v4 = g(): s = 4; e = 5; v1 = 9;

variabila **s** îşi păstrează valoarea la intrarea în funcția **g**;

Se vor afişa valorile: **5 6 7 9 5**

3.7. Calificatorul `const`

Valoarea unei variabile declarate cu acest calificator nu poate fi modificată.

Sintaxa:

***const** tip nume-variabilă = valoare ;*
***nume-funcție*(..., const *tip* **nume-variabilă*, ...);**

În prima variantă (declararea unei variabile) se atribuie o valoare inițială unei variabile care nu mai poate fi ulterior modificată de program. De exemplu,

```
const int virsta = 39;
```

Orice atribuire pentru variabila **virsta** va genera o eroare de compilare.

În a doua variantă (declararea unei funcții) calificatorul **const** este folosit împreună cu un parametru pointer într-o listă de parametri ai unei funcții. Funcția nu poate modifica variabila pe care o indică pointerul:

```
int printf (const char *format, ...);
```

Atenție! O variabilă declarată cu **const** poate fi indirect modificată prin intermediul unui pointer:

```
int *p = &virsta;
*p = 35;
```

4. Operatori şi expresii

Limbajul C prezintă un număr mare de operatori, caracterizaţi prin diferite nivele de prioritate sau precedenţă.

În acest capitol descriem operatorii în ordinea descrescătoare a precedenţei lor. Vom preciza de fiecare dată dacă asociativitatea este la stînga sau la dreapta.

Expresiile combină variabile şi constante pentru a produce valori noi şi le vom introduce pe măsură ce vom prezenţa operatorii.

4.1. Expresii primare

constantă

O constantă este o *expresie-primară*. Tipul său poate fi **`int`**, **`long`** sau **`double`**. Constantele caracter sînt de tip **`int`**, constantele flotante sînt de tip **`long double`**.

identificator

Un identificator este o *expresie-primară*, cu condiţia că el să fi fost declarat corespunzător. Tipul său este specificat în declaraţia sa.

Dacă tipul unui identificator este *masiv de T*, atunci valoarea *expresiei-identificator* este un pointer la primul obiect al masivului, iar tipul expresiei este *pointer la T*. Mai mult, un identificator de masiv nu este o expresie *valoare-stînga* (detalii în capitolul opt).

La fel, un identificator declarat de tip *funcţie care returnează T*, care nu apare pe poziţie de apel de funcţie este convertit la *pointer la funcţie care returnează T* (detalii în capitolul opt).

şir

Un şir este o *expresie-primară*. Tipul său original este *masiv de caractere*, dar urmînd aceleaşi reguli descrise mai sus pentru identificatori, acesta este modificat în *pointer la caracter* şi rezultatul este un pointer la primul caracter al şirului.

(*expresie***)**

O expresie între paranteze rotunde este o *expresie-primară*, al cărei tip şi valoare sînt identice cu cele ale expresiei din interiorul parantezelor (expresia din paranteze poate fi şi o *valoare-stînga*).

***expresie-primară* [*expresie-indice*]**

O *expresie-primară* urmată de o expresie între paranteze pătrate este o *expresie-primară*. Sensul intuitiv este de indexare. De obicei *expresia-primară* are tipul *pointer la T*, *expresia-indice* are tipul **`int`**, iar rezultatul are tipul *T*. O expresie de forma ***E***1**`[`***E***2**`]`** este identică (prin definiție) cu **`*((`***E***1**`)+(`***E***2**`))`**, unde * este operatorul de indirectare (detalii în capitolul opt).

***expresie-primară* (*listă-expresii*)**

Un apel de funcție este o *expresie-primară*. Ea constă dintr-o *expresie-primară* urmată de o pereche de paranteze rotunde, care conțin o *listă-expresii* separate prin virgule. *Lista-expresii* constituie argumentele reale ale funcției; această listă poate fi şi vidă. *Expresia-primară* trebuie să fie de tipul *funcție care returnează T*, iar rezultatul apelului de funcție va fi de tipul *T* (detalii în capitolul şase).

Înaintea apelului, oricare argument de tip **`float`** este convertit la tipul **`double`**, oricare argument de tip **`char`** sau **`short`** este convertit la tipul **`int`**. Numele de masive sînt convertite în pointeri la începutul masivului. Nici o altă conversie nu se efectuează automat.

Dacă este necesar ca tipul unui argument actual să coincidă cu cel al argumentului formal, se va folosi un *cast*.

Sînt permise apeluri recursive la orice funcție.

Despre definirea şi apelul funcțiilor, detalii în capitolul şase.

valoare-stînga* . *identificator

O *valoare-stînga* urmată de un punct şi un identificator este o *expresie-primară*. *Valoarea-stînga* denumeşte o structură sau o reuniune (capitolul nouă) iar identificatorul denumeşte un membru din structură sau reuniune. Rezultatul este o *valoare-stînga* care se referă la membrul denumit din structură sau reuniune.

expresie-primară -> identificator

O *expresie-primară* urmată de o săgeată (constituită dintr-o liniuță şi semnul **>** urmată de un identificator) este o *expresie-primară*. Prima expresie trebuie să fie un pointer la o structură sau reuniune, iar identificatorul trebuie să fie numele unui membru din structura sau reuniunea respectivă. Rezultatul este o *valoare-stînga* care se referă la membrul denumit din structura sau reuniunea către care indică expresia pointer.

Expresia ***E***1**->***E**2 este identică din punctul de vedere al rezultatului cu **(*** ***E***1**) .** ***E***2 (detalii în capitolul nouă).

Listă-expresii

O listă de expresii este considerată de asemenea expresie, dacă acestea sînt separate prin virgulă.

4.2. Operatori unari

Expresiile unare se grupează de la dreapta la stînga.

***** *expresie***

Operatorul unar ***** este operatorul de indirectare. Expresia care-l urmează trebuie să fie un pointer, iar rezultatul este o valoare-stînga care se referă la obiectul către care indică expresia. Dacă tipul expresiei este *pointer la T* atunci tipul rezultatului este *T*. Acest operator tratează operandul său ca o adresă, face acces la ea şi îi obține conținutul (detalii în capitolul opt).

& *valoare-stînga*

Operatorul unar **&** este operatorul de obținere a adresei unui obiect sau de obținere a unui pointer la obiectul respectiv. Operandul este o *valoare-stînga* iar rezultatul este un pointer la obiectul referit de *valoarea-stînga*. Dacă tipul *valorii-stînga* este *T* atunci tipul rezultatului este *pointer la T* (detalii în capitolul opt).

– *expresie*

Operatorul unar **–** este operatorul de negativare. Operandul său este o expresie, iar rezultatul este negativarea operandului. În acest

caz sînt aplicate conversiile aritmetice obişnuite. Negativarea unui întreg de tip **unsigned** se face scăzînd valoarea sa din 2^w, unde *w* este numărul de biţi rezervaţi tipului **int**.

! *expresie*

Operatorul unar **!** este operatorul de negare logică. Operandul său este o expresie, iar rezultatul său este 1 sau 0 după cum valoarea operandului este 0 sau diferită de zero. Tipul rezultatului este **int**. Acest operator este aplicabil la orice expresie de tip aritmetic sau la pointeri.

~ *expresie*

Operatorul unar **~** (tilda) este operatorul de complementare la unu. El converteşte fiecare bit 1 la 0 şi invers. El este un operator logic pe biţi.

Operandul său trebuie să fie de tip întreg. Se aplică conversiile aritmetice obişnuite.

++ *valoare-stînga*
valoare-stînga ++

Operatorul unar **++** este operatorul de incrementare. Operandul său este o *valoare-stînga*. Operatorul produce incrementarea operandului cu 1. Acest operator prezintă un aspect deosebit deoarece el poate fi folosit ca un operator prefix (înaintea variabilei: **++n**) sau ca un operator postfix (după variabilă: **n++**). În ambele cazuri efectul este incrementarea lui **n**. Dar expresia **++n** incrementează pe **n** înainte de folosirea valorii sale, în timp ce **n++** incrementează pe **n** după ce valoarea sa a fost utilizată. Aceasta înseamnă că, în contextul în care se urmăreşte numai incrementarea lui **n**, oricare construcţie poate fi folosită, dar într-un context în care şi valoarea lui **n** este folosită, **++n** şi **n++** furnizează două valori distincte.

***Exemplu*:** dacă `n` este 5, atunci

```
x = n++;        atribuie lui x valoarea 5
x = ++n;        atribuie lui x valoarea 6
```

În ambele cazuri **n** devine 6.

Rezultatul operației nu este o *valoare-stînga*, dar tipul său este tipul *valorii-stînga*.

-- valoare-stînga

valoare-stînga --

Operatorul unar **--** este operatorul de decrementare. Acest operator este analog cu operatorul **++** doar că produce decrementarea cu 1 a operandului.

(nume-tip) expresie

Operatorul **(***nume-tip***)** este operatorul de conversie de tip. Prin *nume-tip* înțelegem unul dintre tipurile fundamentale admise în C inclusiv tipul pointer. Operandul acestui operator este o *expresie*. Operatorul produce conversia valorii *expresiei* la tipul denumit. Această construcție se numeşte *cast*.

sizeof(*operand***)**

Operatorul **sizeof** furnizează dimensiunea în octeți a operandului său. Aplicat unui masiv sau structuri, rezultatul este numărul total de octeți din masiv sau structură. Dimensiunea se determină în momentul compilării, din declarațiile obiectelor din expresie. Semantic, această expresie este o constantă întreagă care se poate folosi în orice loc în care se cere o constantă. Cea mai frecventă utilizare este în comunicarea cu rutinele de alocare a memoriei sau cele de intrare / ieşire.

Operatorul **sizeof** poate fi aplicat şi unui *nume-tip*. În acest caz el furnizează dimensiunea în octeți a unui obiect de tipul indicat.

4.3. Operatori aritmetici

***Expresie-multiplicativă*:**

expresie * expresie

expresie / expresie

expresie % expresie

Operatorii multiplicativi *** / %** şi sînt operatori aritmetici binari şi se grupează de la stînga la dreapta.

Operatorul binar **`*`** indică înmulţirea. Operatorul este asociativ, dar în expresiile în care apar mai mulţi operatori de înmulţire, ordinea de evaluare nu se specifică. Compilatorul rearanjează chiar şi un calcul cu paranteze. Astfel **`a*(b*c)`** poate fi evaluat ca **`(a*b)*c`**. Aceasta nu implică diferenţe, dar dacă totuşi se doreşte o anumită ordine, atunci se vor introduce variabile temporare.

Operatorul binar **`/`** indică împărţirea. Cînd se împart două numere întregi pozitive, trunchierea se face spre zero; dacă unul dintre operanzi este negativ atunci trunchierea depinde de sistemul de calcul.

Operatorul binar **`%`** furnizează restul împărţirii primei expresii la cea de a doua. Operanzii trebuie să fie de tip întreg. Restul are totdeauna semnul deîmpărţitului. Totdeauna **`(a/b)*b+a%b`** este egal cu **`a`** (dacă **`b`** este diferit de 0). Sînt executate conversiile aritmetice obişnuite.

***Expresie-aditivă*:**

expresie* + *expresie

expresie* − *expresie

Operatorii aditivi **`+`** şi **`-`** sînt operatori aritmetici binari şi se grupează de la stînga la dreapta. Se execută conversiile aritmetice obişnuite.

Operatorul binar **`+`** produce suma operanzilor săi. El este asociativ şi expresiile care conţin mai mulţi operatori pot fi rearanjate la fel ca în cazul operatorului de înmulţire.

Operatorul binar **`-`** produce diferenţa operanzilor săi.

4.4. Operatori de comparare

***Expresie-relaţională*:**

expresie* < *expresie

expresie* > *expresie

expresie* <= *expresie

expresie* >= *expresie

Operatorii relaţionali **<**, **>**, **<=**, **>=** se grupează de la stînga la dreapta.

Operatorii **<** (mai mic), **>** (mai mare), **<=** (mai mic sau egal) şi **>=** (mai mare sau egal) produc valoarea 0 dacă relaţia specificată este falsă şi 1 dacă ea este adevărată.

Tipul rezultatului este **int**. Se execută conversiile aritmetice obişnuite. Aceşti operatori au precedenţa mai mică decît operatorii aritmetici.

***Expresie-egalitate*:**

expresie* == *expresie

expresie* != *expresie

Operatorii **==** (egal cu) şi **!=** (diferit de) sînt analogi cu operatorii relaţionali, dar precedenţa lor este mai mică. Astfel că **a<b == c<d** este 1 dacă **a<b** şi **c<d** au aceeaşi valoare de adevăr.

4.5. Operatori logici pe biţi

***Expresie-deplasare*:**

expresie* << *expresie

expresie* >> *expresie

Operatorii de deplasare **<<** şi **>>** sînt operatori logici pe biţi. Ei se grupează de la stînga la dreapta.

Operatorul **<<** produce deplasarea la stînga a operandului din stînga cu un număr de poziţii binare dat de operandul din dreapta.

Operatorul **>>** produce deplasarea la dreapta a operandului din stînga cu un număr de poziţii binare dat de operandul din dreapta.

În ambele cazuri se execută conversiile aritmetice obişnuite asupra operanzilor, fiecare dintre ei trebuind să fie de tip întreg. Operandul din dreapta este convertit la **int**; tipul rezultatului este cel al operandului din stînga. Rezultatul este nedefinit dacă operandul din dreapta este negativ sau mai mare sau egal cu lungimea obiectului, în biţi. Astfel valoarea expresiei ***E*1<<*E*2** este ***E*1** (interpretată ca şi configuraţie de biţi) deplasată la stînga cu ***E*2**

poziții bit; biții eliberați devin zero. Expresia ***E*1>>*E*2** este ***E***1 deplasată la dreapta cu ***E***2 poziții binare. Deplasarea la dreapta este logică (biții eliberați devin 0) dacă ***E***1 este de tip **`unsigned`**, altfel ea este aritmetică (biții eliberați devin copii ale bitului semn).

Exemplu: **`x<<2`** deplasează pe **`x`** la stînga cu 2 poziții, biții eliberați devin 0; aceasta este echivalent cu multiplicarea lui **`x`** cu 4.

Expresie-ŞI:
expresie **`&`** *expresie*

Operatorul **`&`** este operatorul *ŞI* logic pe biți. El este asociativ şi expresiile care conțin operatorul **`&`** pot fi rearanjate. Rezultatul este funcția logică *ŞI* pe biți aplicată operanzilor săi. Operatorul se aplică numai la operanzi de tipuri întregi. Legea după care funcționează este:

`&`	0	1
0	0	0
1	0	1

Operatorul **`&`** este deseori folosit pentru a masca o anumită mulțime de biți: de exemplu:

```
c = n & 0177;
```

pune pe zero toți biții afară de ultimii 7 biți de ordin inferior ai lui **`n`**, fără a afecta conținutul lui **`n`**.

Expresie-SAU-exclusiv:
expresie **`^`** *expresie*

Operatorul **`^`** este operatorul *SAU-exclusiv* logic pe biți. El este asociativ şi expresiile care-l conțin pot fi rearanjate. Rezultatul este funcția logică *SAU-exclusiv* pe biți aplicată operanzilor săi. Operatorul se aplică numai la operanzi de tipuri întregi. Legea după care funcționează este:

`^`	0	1
0	0	1
1	1	0

***Expresie-SAU-inclusiv*:**
expresie* `|` *expresie

Operatorul `|` este operatorul *SAU-inclusiv* logic pe biți. El este asociativ şi expresiile care-l conțin pot fi rearanjate. Rezultatul este funcția logică *SAU-inclusiv* pe biți aplicată operanzilor săi. Operatorul se aplică numai la operanzi de tipuri întregi. Legea după care funcționează este:

`|`	0	1
0	0	1
1	1	1

Operatorul `|` este folosit pentru a poziționa biți; de exemplu:

```
x = x | MASK;
```

pune pe 1 toți biții din `x` care corespund la biți poziționați pe 1 din `MASK`. Se efectuează conversiile aritmetice obişnuite.

4.6. Operatori pentru expresii logice

***Expresie-ŞI-logic*:**
expresie* `&&` *expresie

Operatorul `&&` este operatorul *ŞI-logic* şi el se grupează de la stînga la dreapta. Rezultatul este 1 dacă ambii operanzi sînt diferiți de zero şi 0 în rest. Spre deosebire de operatorul *ŞI* pe biți `&`, operatorul *ŞI-logic* `&&` garantează o evaluare de la stînga la dreapta; mai mult, al doilea operand nu este evaluat dacă primul operand este 0.

Operanzii nu trebuie să aibă în mod obligatoriu acelaşi tip, dar fiecare trebuie să aibă unul dintre tipurile fundamentale sau pointer. Rezultatul este totdeauna de tip `int`.

***Expresie-SAU-logic*:**
expresie* `||` *expresie

Operatorul `||` este operatorul *SAU-logic* şi el se grupează de la stînga la dreapta. Rezultatul este 1 dacă cel puțin unul dintre operanzi este diferit de zero şi 0 în rest.

Spre deosebire de operatorul *SAU-inclusiv* pe biți **|**, operatorul *SAU-logic* **||** garantează o evaluare de la stînga la dreapta; mai mult, al doilea operand nu este evaluat dacă valoarea primului operand este diferită de zero.

Operanzii nu trebuie să aibă în mod obligatoriu acelaşi tip, dar fiecare trebuie să aibă unul dintre tipurile fundamentale sau pointer. Rezultatul este totdeauna de tip **int**.

4.7. Operatorul condițional

***Expresie-condițională*:**

expresie* ? *expresie* : *expresie

Operatorul condițional **?:** este un operator ternar. Prima expresie se evaluează şi dacă ea este diferită de zero sau adevărată, rezultatul este valoarea celei de-a doua expresii, altfel rezultatul este valoarea expresiei a treia. De exemplu expresia:

```
z = (a>b) ? a : b;
```

calculează maximul dintre **a** şi **b** şi îl atribuie lui **z**. Se evaluează mai întîi prima expresie **a>b**. Dacă ea este adevărată se evaluează a doua expresie şi valoarea ei este rezultatul operației, această valoare se atribuie lui **z**. Dacă prima expresie nu este adevărată atunci **z** ia valoarea lui **b**.

Expresia condițională poate fi folosită peste tot unde sintaxa cere o expresie.

Dacă este posibil, se execută conversiile aritmetice obişnuite pentru a aduce expresia a doua şi a treia la un tip comun; dacă ambele expresii sînt pointeri de acelaşi tip, rezultatul are şi el acelaşi tip; dacă numai o expresie este un pointer, cealaltă trebuie sa fie constanta 0, iar rezultatul este de tipul pointerului. Întotdeauna numai una dintre expresiile a doua şi a treia este evaluată.

Dacă **f** este flotant şi n întreg, atunci expresia

```
(h>0) ? f: n
```

este de tip **double** indiferent dacă **n** este pozitiv sau negativ. Parantezele nu sînt necesare deoarece precedența operatorului **?:**

este mai mică, dar ele pot fi folosite pentru a face expresia condițională mai lizibilă.

4.8. Operatori de atribuire

Există mai mulți operatori de atribuire, care se grupează toți de la dreapta la stînga. Operandul stîng este o valoare-stînga, operandul drept este o expresie. Tipul expresiei de atribuire este tipul operandului stîng.

Rezultatul este valoarea memorată în operandul stîng după ce atribuirea a avut loc. Cele două părți ale unui operator de atribuire compus sînt unități lexicale (simboluri) distincte.

***Expresie-atribuire*:**
valoare-stînga* = *expresie
valoare-stînga op= expresie

unde ***op*** poate fi unul din operatorii **+**, **-**, *****, **/**, **%**, **<<**, **>>**, **&**, **^**, **|**.

Într-o atribuire simplă cu **=**, valoarea expresiei înlocuieşte pe cea a obiectului referit de *valoare-stînga*. Dacă ambii operanzi au tip aritmetic, atunci operandul drept este convertit la tipul operandului stîng înainte de atribuire.

Expresiile de forma ***E***1 ***op*= *E***2 se interpretează ca fiind echivalente cu expresiile de forma ***E***1 **=** ***E***1 ***op*** ***E***2; totuşi ***E***1 este evaluată o singură dată.

***Exemplu*:** expresia **x *= y+1** este echivalentă cu

```
x = x * (y+1)
```

şi nu cu

```
x = x * y + 1
```

Pentru operatorii **+=** şi **-=**, operandul stîng poate fi şi un pointer, în care caz operandul din dreapta este convertit la întreg (capitolul opt). Toți operanzii din dreapta şi toți operanzii din stînga care nu sînt pointeri trebuie să fie de tip aritmetic.

Atribuirea prescurtată este avantajoasă în cazul cînd în membrul stîng avem expresii complicate, deoarece ele se evaluează o singură dată.

4.9. Operatorul virgulă

***Expresie-virgulă*:**

expresie*, *expresie

O pereche de expresii separate prin virgulă se evaluează de la stînga la dreapta şi valoarea expresiei din stînga se neglijează. Tipul şi valoarea rezultatului sînt cele ale operandului din dreapta. Aceşti operatori se grupează de la stînga la dreapta. În contextele în care virgula are un sens special, (de exemplu într-o listă de argumente reale ale unei funcţii şi lista de iniţializare), operatorul virgulă descris aici poate apărea numai în paranteze. De exemplu funcţia:

```
f(a,(t=3,t+2),c)
```

are trei argumente, dintre care al doilea are valoarea 5. Expresia acestui argument este o expresie virgulă. În calculul valorii lui se evaluează întîi expresia din stînga şi se obţine valoarea 3 pentru **t**, apoi cu această valoare se evaluează a doua expresie şi se obţine **t=**5. Prima valoare a lui **t** se pierde.

4.10. Precedenţa şi ordinea de evaluare

Tabelul de la sfîrşitul acestei secţiuni constituie un rezumat al regulilor de precedenţă şi asociativitate ale tuturor operatorilor.

Operatorii din aceeaşi linie au aceeaşi precedenţă; liniile sînt scrise în ordinea descrescătoare a precedenţei, astfel de exemplu operatorii *****, **/** şi **%** au toţi aceeaşi precedenţă, care este mai mare decît aceea a operatorilor **+** şi **-**.

După cum s-a menţionat deja, expresiile care conţin unul dintre operatorii asociativi sau comutativi (*****, **+**, **&**, **^**, **|**) pot fi rearanjate de compilator chiar dacă conţin paranteze. În cele mai multe cazuri aceasta nu produce nici o diferenţă; în cazurile în care o asemenea diferenţă ar putea apărea pot fi utilizate variabile temporare explicite, pentru a forţa ordinea de evaluare.

Limbajul C, ca şi multe alte limbaje, nu specifică în ce ordine sînt evaluaţi operanzii unui operator. Dăm în continuare două exemple:

```
x = f() + g();
a[i] = i++;
```

În asemenea situații trebuie memorate rezultate intermediare în variabile temporare pentru a garanta o execuție în ordinea dorită.

Singurele cazuri în care se garantează evaluarea de la stînga la dreapta se referă la operatorii pentru expresii logice **`&&`** şi **`||`**. În aceste cazuri se garantează în plus un număr minim de evaluări.

Operator	Asociativitate
`() [] -> .`	stînga la dreapta
`! ++ -- - (`*`tip`*`) * & sizeof`	dreapta la stînga
`* / %`	stînga la dreapta
`+ -`	stînga la dreapta
`<< >>`	stînga la dreapta
`< <= > >=`	stînga la dreapta
`== !=`	stînga la dreapta
`&`	stînga la dreapta
`^`	stînga la dreapta
`|`	stînga la dreapta
`&&`	stînga la dreapta
`||`	stînga la dreapta
`?:`	dreapta la stînga
`=` *`op`*`=`	dreapta la stînga
`,`	stînga la dreapta

5. Instrucțiuni

Într-un program scris în limbajul C instrucțiunile se execută secvențial, în afară de cazul în care se indică altfel.

Instrucțiunile pot fi scrise cîte una pe o linie pentru o lizibilitate mai bună, dar nu este obligatoriu.

5.1. Instrucțiunea *expresie*

Cele mai multe instrucțiuni sînt instrucțiuni *expresie*. O expresie devine instrucțiune dacă ea este urmată de punct şi virgulă.

```
a = b + c;
n++;
scanf(...);
```

În limbajul C punct şi virgula este un terminator de instrucțiune şi este obligatoriu.

O instrucțiune de atribuire poate apărea pe post de expresie într-o altă instrucțiune. Funcția **strcpy** copiază şirul **s** peste şirul **t**.

```
void strcpy(char t[], const char s[]) {
int i = 0;
while (t[i]=s[i]) i++;
}
```

Expresia **t[i]=s[i]** este o atribuire şi are valoarea *True* sau *False* în funcție de valoarea **s[i]**.

5.2. Instrucțiunea compusă sau blocul

Instrucțiunea compusă este o grupare de declarații şi instrucțiuni închise între acolade. Aceasta a fost introdusă cu scopul de a folosi mai multe instrucțiuni acolo unde sintaxa cere o instrucțiune. Instrucțiunea compusă sau blocul este echivalentă sintactic cu o singură instrucțiune.

Format:

{
listă-declaratori
listă-instrucțiuni
}

Dacă anumiți identificatori din *lista-declaratori* au fost declarați anterior, atunci declarația exterioară este ignorată pe durata blocului, după care își reia sensul său.

Orice inițializare pentru variabilele **auto** şi **register** se efectuează la fiecare intrare în bloc. Inițializările pentru variabilele **static** se execută numai o singură dată cînd programul începe să se execute.

Un bloc se termină cu o acoladă dreaptă care nu este urmată niciodată de caracterul punct şi virgulă.

5.3. Instrucțiunea `if`

Sintaxa instrucțiunii condiționale **if** admite două formate:

if (*expresie***)**	**if (***expresie***)** *instrucțiune*-1
instrucțiune-1	**else** *instrucțiune*-2

Instrucțiunea **if** se foloseşte pentru a lua decizii. În ambele cazuri se evaluează expresia şi dacă ea este *adevărată* (deci diferită de zero) se execută *instrucțiune*-1. Dacă expresia este *falsă* (are valoarea zero) şi instrucțiunea **if** are şi parte de **else** atunci se execută *instrucțiune*-2.

În al doilea caz una şi numai una dintre cele două instrucțiuni se execută. Deoarece un **if** testează pur şi simplu valoarea numerică a unei expresii, se admite o prescurtare şi anume:

if (*expresie***)**

în loc de:

if (*expresie* != 0**)**

Deoarece partea **else** a unei instrucțiuni **if** este opțională, există o ambiguitate cînd un **else** este omis dintr-o secvență de **if**

imbricată. Aceasta se rezolvă asociind **else** cu ultimul **if** care nu are **else**.

```
if (n>0)                    if (n>0) {
   if (a>b)                    if (a>b)
      z = a;                      z = a;
   else                        }
      z = b;                else
                                 z = b;
```

În exemplul de mai sus, în primul caz partea **else** se asociază **if**-ului din interior. Dacă nu dorim acest lucru atunci folosim acoladele pentru a forța asocierea, ca în al doilea caz.

Instrucțiunea condițională admite şi construcția **else-if**:

if (expresie-1) ***instrucțiune*-1**
else
if (expresie-2) ***instrucțiune*-2**
else
if (expresie-3) ***instrucțiune*-3**
else ***instrucțiune*-4**

Această cascadă de **if**-uri se foloseşte frecvent în programe, ca mod de a exprima o decizie multiplă.

Expresiile se evaluează în ordinea în care apar; dacă se întîlneşte o expresie adevărată, atunci se execută instrucțiunea asociată cu ea şi astfel se termină întregul lanț.

Oricare instrucțiune poate fi o instrucțiune simplă sau un grup de instrucțiuni între acolade.

Instrucțiunea după ultimul **else** se execută în cazul în care nici o expresie nu a fost adevărată.

Dacă în acest caz nu există nici o acțiune explicită de făcut, atunci ultima parte poate să lipsească:

else ***instrucțiune*-4**

Pot exista un număr arbitrar de construcții:

else if (*expresie***)** ***instrucțiune***

grupate între un **if** inițial şi un **else** final.

Întotdeauna un **else** se asociază cu ultimul **if** întîlnit.

5.4. Instrucțiunile `while` şi `do`

Formatul instrucțiunii **`while`**:

`while (`*expresie***`)`** *instrucțiune*

Instrucțiunea se execută repetat atîta timp cît valoarea expresiei este diferită de zero. Testul are loc *înaintea* fiecărei execuții a instrucțiunii. Prin urmare ciclul este următorul: se testează condiția din paranteze dacă ea este adevărată (expresia din paranteze are o valoare diferită de zero) se execută corpul instrucțiunii **`while`**, se verifică din nou condiția, dacă ea este adevărată se execută din nou corpul instrucțiunii. Cînd condiția devine falsă (valoarea expresiei din paranteze este zero) se continuă execuția cu instrucțiunea de după corpul instrucțiunii **`while`**, deci instrucțiunea **`while`** se termină.

Formatul instrucțiunii **`do`**:

`do` *instrucțiune*
`while (`*expresie***`);`**

Instrucțiunea se execută repetat pînă cînd valoarea expresiei devine zero. Testul are loc *după* fiecare execuție a instrucțiunii.

```
unsigned n, c = 0;
scanf("%u",&n);
while (n>0) {
    n /= 10;  c++;
    }
printf("%u",c);
```

```
unsigned n, c = 0;
scanf("%u",&n);
do {
   c++;  n /= 10;
   } while (n>0);
printf("%u",c);
```

Secvența din stînga afişează numărul de cifre zecimale ale valorii lui **`n`** numai dacă **`n >`** 0, altfel afişează zero. Secvența din dreapta afişează numărul de cifre zecimale ale valorii lui **`n`** în orice condiții.

5.5. Instrucțiunea `for`

Format:

`for (`*expresie*-1**`;`** *expresie*-2**`;`** *expresie*-3**`)`** *instrucțiune*

Această instrucțiune este echivalentă cu secvența:

expresie-1**`;`**

```
while (expresie-2) {
    instrucțiune;
    expresie-3;
    }
```

Expresie-1 constituie inițializarea ciclului şi se execută o singură dată înaintea ciclului. *Expresie*-2 specifică testul care controlează ciclul. El se execută înaintea fiecărei iterații. Dacă condiția din test este adevărată atunci se execută corpul ciclului, după care se execută *expresie*-3, care constă de cele mai multe ori în modificarea valorii variabilei de control al ciclului. Se revine apoi la reevaluarea condiției. Ciclul se termină cînd condiția devine falsă.

Oricare dintre expresiile instrucțiunii **for** sau chiar toate pot lipsi.

Dacă lipseşte *expresie*-2, aceasta implică faptul că clauza **while** este echivalentă cu **while (1)**, ceea ce înseamnă o condiție totdeauna adevărată. Alte omisiuni de expresii sînt pur şi simplu eliminate din expandarea de mai sus.

Instrucțiunile **while** şi **for** permit un lucru demn de observat, şi anume: ele execută testul de control la începutul ciclului şi înaintea intrării în corpul instrucțiunii.

Dacă nu este nimic de făcut, nu se face nimic, cu riscul de a nu intra niciodată în corpul instrucțiunii.

Exemple. 1) În primul capitol, al treilea şi al patrulea program folosesc instrucțiunea **for** pentru a parcurge un şir de valori. Să notăm faptul că fiecare expresie poate fi oricît de complexă. Secvența următoare determină suma primelor n numere naturale.

```
unsigned s,i;
scanf("%u",&n);
for (s=0,i=1; i<=n; i++)
    s += i;
printf("%u",s);
```

2) Funcția **strlen(s)** returnează lungimea şirului de caractere **s**, excluzînd caracterul terminal null.

```
int strlen(const char s[]) {
int n;
```

```
for (n=0; s[i]; ++n) ;
return n;
}
```

3) Instrucțiunea de inițializare (*expresie*-1) trebuie să fie una singură, de aceea, dacă avem mai multe operații de făcut, ele trebuie separate prin virgulă şi nu prin punct şi virgulă. Aceeaşi observație este valabilă şi pentru *expresie*-3. Funcția **strrev** inversează un şir de caractere:

```
void strrev(char s[]) {
int i,j;
for (i=0,j=strlen(s)-1; i<j; i++,j--) {
    char c = s[i];
    s[i] = s[j];
    s[j] = c;
    }
}
```

4) Reluăm problema din secțiunea precedentă: dîndu-se o valoare naturală **n**, să se determine numărul de cifre zecimale.

```
unsigned n,c;
scanf("%u",&n);
for (c=!n; n; c++)
    n /= 10;
printf("%u",c);
```

5) Funcția **strcpy** poate fi implementată şi astfel.

```
void strcpy(char t[], const char s[]) {
int i;
for (i=0; t[i]=s[i]; i++) ;
}
```

5.6. Instrucțiunea switch

Instrucțiunea **switch** este o decizie multiplă specială şi determină transferul controlului unei instrucțiuni sau unui bloc de instrucțiuni dintr-un şir de instrucțiuni în funcție de valoarea unei expresii.

Format:

switch (*expresie***)** *instrucțiune-bloc*

Expresia este supusă la conversiile aritmetice obişnuite dar rezultatul evaluării trebuie să fie de tip întreg.

Fiecare instrucțiune din corpul instrucțiunii **switch** poate fi etichetată cu una sau mai multe prefixe **case** astfel:

case *expresie-constantă***:**

unde *expresie-constantă* trebuie să fie de tip întreg.

Poate exista de asemenea cel mult o instrucțiune etichetată cu

default:

Cînd o instrucțiune **switch** se execută, se evaluează expresia din paranteze şi valoarea ei se compară cu fiecare constantă din fiecare **case**.

Dacă se găseşte o constantă **case** egală cu valoarea expresiei, atunci se execută secvența care urmează după **case**-ul respectiv.

Dacă nici o constantă **case** nu este egală cu valoarea expresiei şi dacă există un prefix **default**, atunci se execută secvența de după el, altfel nici o instrucțiune din **switch** nu se execută.

Prefixele **case** şi **default** nu alterează fluxul de control, care continuă printre astfel de prefixe.

Pentru ieşirea din **switch** se foloseşte instrucțiunea **break** sau **return**. Dacă o secvență de instrucțiuni este urmată de un nou **case** fără să fie încheiată cu **break** sau **return**, *nu se iese din* **switch**. În concluzie, ieşirea din **switch** trebuie să fie explicită.

Exemplu. Un meniu poate fi implementat astfel:

```
char op[80];
do {
   gets(op);
   switch (op[0]) {
      case 'i':  case 'I':
         Initializare(...);  break;
      case 'a':  case 'A':
         Adaugare(...);  break;
      case 'm':  case 'M':
         Modificare(...);  break;
      case 's':  case 'S':
```

```
      Stergere(...);  break;
   case 'c':  case 'C':
      Consultare(...);  break;
   case 'l':  case 'L':
      Listare(...);  break;
   default:
      Eroare(...);  break;
   }
} while (op!='t' && op!='T');
```

În funcție de opțiunea dorită (primul caracter din șirul **op**) se selectează acțiunea asociată valorii respective. Dacă opțiunea nu este recunoscută, se selectează acțiunea care semnalează o eroare. Toate aceste acțiuni se execută în cadrul unui ciclu **do**. Acesta se încheie în momentul în care se introduce opțiunea **'t'** sau **'T'**.

5.7. Instrucțiunile `break` și `continue`

Formatul instrucțiunii **break**:

```
   break;
```

Această instrucțiune determină terminarea celei mai interioare instrucțiuni **while**, **do**, **for** sau **switch** care o conține. Controlul trece la instrucțiunea care urmează după instrucțiunea astfel terminată.

Exemplu. Fie un șir de caractere **s**; să se elimine spațiile de la sfîrșitul șirului.

```
for (i=strlen(s)-1; i>=0; i--)
   if (s[i]!=' ') break;
s[i+1] = 0;
```

Formatul instrucțiunii **continue**:

```
   continue;
```

Această instrucțiune determină trecerea controlului la porțiunea de continuare a ciclului celei mai interioare instrucțiuni **while**, **do** sau **for** care o conține, adică la sfîrșitul ciclului și reluarea următoarei iterații a ciclului. În **while** și **do** se continuă cu testul, iar în **for** se continuă cu *expresie*-3.

Mai precis în fiecare dintre instrucțiunile:

`while (...) {`
`  ...`
`  contin:;`
`}`
	`for (...) {`
	`  ...`
	`  contin:;`
	`}`
		`do {`
		`  ...`
		`  contin:;`
		`} while (…);`

dacă apare o instrucțiune **`continue`** aceasta este echivalentă cu un salt la eticheta **`contin`**. După **`contin:`** urmează o instrucțiune vidă.

Secvența următoare prelucrează numai elementele pozitive ale unui masiv.

```
for (i=0; i<n; i++) {
   if (a[i]<0)      /* sare peste elementele negative */
      continue;
     ...       /* prelucrează elementele pozitive */
   }
```

Un exemplu de utilizare a instrucțiunii **`goto`**

Pentru a efectua un salt explicit în interiorul unui program C se poate folosi şi instrucțiunea **`goto`**, deşi aceasta conduce la programe greu de verificat.

Fie două masive, **`a`** cu **`n`** elemente şi **`b`** cu **`m`** elemente. Să se determine dacă există cel puțin o valoare comună. O secvență care utilizează instrucțiunea **`goto`** se poate scrie astfel:

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      if (a[i]==b[j])
         goto egale;
/* Nu s-a găsit nici un element; prelucrări necesare acestui caz */
goto contin;
egale:
/* Un element comun pe pozițiile i şi j; prelucrări necesare acestui caz */
contin:
/* Continuă execuția după finalizarea prelucrărilor pentru cazul depistat */
```

Dacă scriem o funcție în acest scop, nu avem nevoie să folosim instrucțiunea **`goto`**. Funcția va memora pozițiile **`i`** şi **`j`** în variabile

ale căror adrese sînt comunicate de o funcție apelantă. Dacă nu există o valoare comună atunci aceste variabile vor conține valori negative.

```
void Comun(double a[], int n, double b[], int m,
           int *pi, int *pj) {
int i,j;
*pi = *pj = -1;
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      if (a[i]==b[j]) {
         *pi = i; *pj = j;
         return;
         }
}
```

Această funcție poate fi apelată astfel:

```
Comun(a,n,b,m,&k,&l);
```

5.8. Instrucțiunea `return`

O instrucțiune **`return`** permite ieşirea dintr-o funcție şi transmiterea controlului apelantului funcției. O funcție poate returna valori apelantului său, prin intermediul unei instrucțiuni **`return`**.

Formate:

`return;`

`return` *expresie*`;`

Primul format se foloseşte pentru funcții de tip **`void`**, care nu returnează în mod explicit o valoare. În al doilea format valoarea expresiei este returnată funcției apelante. Dacă este nevoie, expresia este convertită, ca într-o atribuire, la tipul funcției în care ea apare.

Detalii despre instrucțiunea **`return`** vor fi prezentate în capitolul următor.

5.11. Instrucțiunea vidă

Format:

`;`

Instrucțiunea vidă este utilă într-o instrucțiune compusă, sau pentru a introduce un corp nul într-o instrucțiune de ciclare care cere corp al instrucțiunii, ca de exemplu **`while`** sau **`for`**.

***Exemplu*:**

```
for (nc=0; s[nc]; ++nc) ;
```

În această instrucțiune, care numără caracterele unui şir, corpul lui `for` este vid, deoarece toată activitatea se face în partea de test şi actualizare dar sintaxa lui `for` cere un corp al instrucțiunii. Instrucțiunea vidă satisface acest lucru.

Am precizat anterior că o instrucțiune compusă nu este urmată niciodată de caracterul punct şi virgulă.

```
if (…) {
   ...
   };
else ...
```

Această secvență este greşită, deoarece după instrucțiunea compusă urmează o instrucțiune vidă, şi clauza **`else`** apare izolată fără a avea nici o instrucțiune **`if`** asociată.

6. Funcţii

Funcţiile sînt elementele de bază ale unui program scris în C: orice program, de orice dimensiune, constă din una sau mai multe funcţii, care specifică operaţiile care trebuie efectuate.

O funcţie oferă un mod convenabil de încapsulare a anumitor calcule într-o cutie neagră care poate fi utilizată apoi fără a avea grija conţinutului ei. Funcţiile sînt într-adevăr singurul mod de a face faţă complexităţii programelor mari, permit desfacerea programelor mari în module mai mici, şi dau utilizatorului posibilitatea de a dezvolta programe, folosind ceea ce alţii au făcut deja, în loc să o ia de la început.

Limbajul C a fost conceput să permită definirea de funcţii eficiente şi uşor de mînuit. În general e bine să concepem programe constituite din mai multe funcţii mici decît din puţine funcţii de dimensiuni mari. Un program poate fi împărţit în mai multe fişiere sursă în mod convenabil, iar fişierele sursă pot fi compilate separat.

Un program C constă dintr-o secvenţă de definiţii externe de funcţii şi de date. În fiecare program trebuie să existe o funcţie cu numele impus `main`; orice program îşi începe execuţia cu funcţia `main`. Celelalte funcţii sînt apelate din interiorul funcţiei `main`. Unele dintre funcţiile apelate sînt definite în acelaşi program, altele sînt conţinute într-o bibliotecă de funcţii.

6.1. Definiţia unei funcţii

O funcţie se defineşte astfel:

clasă tip-funcţie nume-funcţie **(***listă-parametri***) {**
listă-declaraţii
listă-instrucţiuni
}

Într-o declaraţie, numele funcţiei este urmat de paranteze chiar dacă lista parametrilor este vidă. Dacă funcţia are mai mulţi

parametri, declarațiile acestora sînt separate prin virgulă. Un declarator de funcție care nu o şi defineşte nu conține corpul funcției, acesta fiind înlocuit cu caracterul punct şi virgulă.

După cum se observă, diferitele părți pot să lipsească; o funcție minimă este:

```
dummy() {}
```

funcția care nu face nimic. Această funcție poate fi utilă în programe, ținînd locul unei alte funcții în dezvoltarea ulterioară a programului.

Numele funcției poate fi precedat de un tip, dacă funcția returnează o valoare de alt tip decît **int**, sau dacă tipul acesteia este **void**. O funcție de tip **void** nu returnează o valoare în mod explicit, execută doar o secvență de instrucțiuni.

Numele funcției poate fi de asemenea precedat de clasa de memorie **extern** sau **static**, alți specificatori de clasă nu sînt admişi. Dacă nici un specificator de clasă de memorie nu este prezent, atunci funcția este automat declarată externă, deoarece în C nu se admite ca o funcție să fie definită în interiorul altei funcții. Dacă o funcție este declarată **static**, ea poate fi apelată numai din fişierul unde a fost definită.

Comunicarea între funcții se face prin argumente şi valori returnate de funcții. Comunicarea între funcții poate fi făcută şi prin intermediul variabilelor globale.

Se recomandă ca definiția unei funcții sau cel puțin declararea ei să se facă înainte de orice apelare. Astfel compilatorul poate depista imediat neconcordanțe între tipurile de parametri.

Să ilustrăm mecanismul definirii unei funcții scriind funcția factorial **fact(n)**. Această funcție este folosită într-un program care calculează valoarea expresiei $C_n^k = \frac{n!}{k!(n-k)!}$

```
#include <stdio.h>
int fact(int n) {
int i,p;
for (p=i=1; i<=n; i++)
   p *= i;
return p;
}
```

```
int main() {
int n,k,i;
puts("Valori n si k: ");
scanf("%d %d",&n,&k);
printf("Combinari:%d\n",fact(n)/fact(k)/fact(n-k));
return 0;
}
```

Funcția **fact** este apelată în programul principal de trei ori, fiecare apel transmite funcției un argument.

O secvență mai eficientă – din punctul de vedere al volumului de calcul şi al corectitudinii acestora – se bazează pe următoarea formulă: $C_n^k = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \ldots \cdot \frac{n-k+1}{k}$

Funcția **fact** produce rezultate eronate pentru valori mari ale lui **n** ($n \geq 8$, sisteme de calcul pe 16 biți; $n \geq 12$, sisteme de calcul pe 32 de biți). Funcția **comb**, bazată pe relația de mai sus, permite operarea cu valori mai mari ale parametrilor **n** şi **k**.

```
int comb(int n, int k) {
int i,p;
for (p=i=1; i<=k; i++)
   p = p * (n-i+1) / i;
return p;
}
```

Atenție! În acest caz nu putem folosi forma compactă

```
   p *= (n-i+1) / i;
```

deoarece se va evalua mai întîi expresia din dreapta şi rezultatul va fi înmulțit cu valoarea lui **p**. Astfel se vor obține rezultate eronate pentru orice valoare **n** pară.

6.2. Apelul funcțiilor; funcții recursive

În limbajul C orice funcție este apelată prin numele ei, urmat de lista reală a argumentelor, închisă între paranteze.

Dacă într-o expresie numele funcției nu este urmat imediat de o paranteză stînga **(**, adică nu se apelează funcția respectivă, se

generează un pointer la funcție. Detalii despre acest subiect vor fi prezentate în capitolul opt.

Funcțiile în C pot fi definite recursiv, deci o funcție se poate apela pe ea însăşi fie direct, fie indirect. În general însă recursivitatea nu face economie de memorie, deoarece trebuie menținută o stivă cu valorile de prelucrat.

Exemple. 1) Funcția **rcomb** implementează formula recursivă de calcul a valorii C_n^k :

```
int rcomb(int n, int k) {
if (n==k || k==0) return 1;
else return rcomb(n,k-1) + rcomb(n-1,k-1);
}
```

2) Funcția **rfibo** implementează procedura recursivă de calcul a termenului de rang **n** din şirul lui Fibonacci:

$$F_n := \begin{cases} n, n \leq 1 \\ F_{n-1} + F_{n-2}, n > 1 \end{cases}$$

```
int rfibo(int n) {
if (n<=1) return 1;
else return fibo(n-1) + fibo(n-2);
}
```

Simplitatea cu care sînt definite aceste funcții ar putea cuceri imediat pe un programator neexperimentat. Trebuie să atragem atenția asupra faptului că cele două definiții au o complexitate exponențială: pentru o valoare oarecare a lui **n**, numărul de operații elementare efectuate (aritmetice şi logice) este aproximativ proporțional cu 2^n.

De aceea recursivitatea este justificată numai în situația în care ştim cu certitudine că se obține o definiție avînd complexitate acceptabilă. Un exemplu în acest sens va fi dat în capitolul nouă: gestionarea unei structuri de date de tip arbore.

Am văzut în secțiunea precedentă o definiție simplă şi eficientă a unei funcții care calculează valoarea C_n^k. În continuare prezentăm o definiție simplă şi eficientă pentru determinarea unui termen din şirul lui Fibonacci:

```
int fibo(int n) {
int a,b,c,i;
a = 0;  b = 1;  c = n;
for (i=2; i<=n; i++) {
   c = a + b;
   a = b;  b = c;
   }
return p;
}
```

Funcția foloseşte variabilele locale **a** şi **b** pentru a memora termenii F_{k-2} şi F_{k-1}. La un anumit pas se calculează $F_k = F_{k-2} + F_{k-1}$ în variabila **c**, şi apoi variabilele a şi b memorează termenii F_{k-1} şi F_k.

6.3. Revenirea din funcții

Revenirea dintr-o funcție se face prin intermediul unei instrucțiuni **return**.

Valoarea pe care o funcție o calculează poate fi returnată prin instrucțiunea **return**, care după cum am văzut are două formate. Ca argument al acestei instrucțiuni poate apărea orice expresie admisă în C. Funcția returnează valoarea acestei expresii funcției apelante.

O funcție de tip **void** nu returnează în mod explicit o valoare. O instrucțiune **return**, fără expresie ca parametru, cauzează doar transferul controlului funcției apelante.

La rîndul său funcția apelantă poate ignora valoarea returnată.

Dacă nu se precizează tipul funcției, aceasta returnează o valoare de tip întreg. Dacă se doreşte ca funcția să returneze un alt tip, atunci numele tipului trebuie să preceadă numele funcției.

Pentru a evita orice confuzie recomandăm ca tipul valorii returnate de funcție să fie întotdeauna precizat, iar dacă dorim în mod expres ca funcția să nu returneze o valoare în mod explicit să o declarăm de tip **void**.

De exemplu, funcţia **atof(s)** din biblioteca asociată limbajului C converteşte şirul **s** de cifre în valoarea sa în dublă precizie. Putem declara funcţia sub forma:

```
double atof(char s[]);
```

Dacă o funcţie returnează o valoare de tip **char**, nu este nevoie de nici o declaraţie de tip datorită conversiilor implicite. Totdeauna tipul **char** este convertit la **int** în expresii.

6.4. Argumentele funcţiei şi transmiterea parametrilor

O metodă de a comunica datele între funcţii este prin argumente. Parantezele care urmează imediat după numele funcţiei închid lista argumentelor.

În limbajul C argumentele funcţiilor sînt transmise prin valoare. Aceasta înseamnă că în C funcţia apelată primeşte valorile argumentelor sale într-o copie particulară de variabile locale (concret pe stivă).

Apelul prin valoare este o posibilitate, dar nu şi o obligativitate. Apelul prin valoare conduce la programe mai compacte cu mai puţine variabile externe, deoarece argumentele pot fi tratate ca variabile locale, şi care pot fi modificate convenabil în rutina apelată.

Ca exemplu, prezentăm o versiune a funcţiei factorial care face uz de acest fapt:

```
int fact(int n) {
int p;
for (p=1; n>0; --n)
   p *= n;
return p;
}
```

Argumentul n este utilizat ca o variabilă automatică şi este decrementat pînă devine zero; astfel nu este nevoie de încă o variabilă **i**. Orice operaţii s-ar face asupra lui **n** în interiorul funcţiei, ele nu au efect asupra argumentului pentru care funcţia a fost apelată.

Dacă totuşi se doreşte alterarea efectivă a unui argument al funcţiei apelante, acest lucru se realizează cu ajutorul pointerilor sau al variabilelor declarate externe. În cazul pointerilor, funcţia apelantă trebuie să furnizeze adresa variabilei care trebuie modificată (tehnic printr-un pointer la această variabilă), iar funcţia apelată trebuie să declare argumentul corespunzător ca fiind un pointer. Referirea la variabila care trebuie modificată se face prin adresare indirectă (capitolul opt).

Printre argumentele funcţiei pot apărea şi nume de masive. În acest caz valoarea transmisă funcţiei este în realitate adresa de început a masivului (elementele masivului nu sînt copiate). Prin indexarea acestei valori funcţia poate avea acces şi poate modifica orice element din masiv. Dacă a fost precizat calificatorul **`const`**, modificarea nu este permisă.

Pentru funcţiile cu număr variabil de parametri standardul ANSI defineşte o construcţie specială **`...`** (trei puncte) numită elipsă. Acesta este de exemplu cazul funcţiilor de citire şi scriere cu format (familiile *...scanf* şi *...printf* – capitolul zece). În acest caz, parametrii ficşi sînt verificaţi la compilare, iar cei variabili sînt transmişi fără nici o verificare.

6.5. Exemple de funcţii şi programe

1. Acest exemplu este un program de căutare şi afişare a tuturor liniilor dintr-un text care conţin o anumită configuraţie de caractere, de exemplu cuvîntul englezesc "*the*". Structura de bază a acestui program este:

```
while (mai există linii sursă)
    if (linia conţine configuraţia de caractere)
        afişează linia
```

Pentru a citi cîte o linie din fişierul de intrare vom folosi funcţia **`gets`** din bibliotecă, cunoscută din primul capitol unde a fost folosită într-un exemplu. Această funcţie citeşte o linie de la intrarea standard şi o depune într-un şir de caractere precizat prin argumentul acesteia. Dacă la intrarea standard mai există linii (se mai poate citi) funcţia

returnează adresa şirului unde a fost memorată linia, în caz contrar funcţia returnează valoarea NULL (zero).

Funcţia **index** caută configuraţia de caractere dorită în interiorul liniei şi furnizează poziţia sau indexul în şirul **s**, unde începe şirul **t**, sau −1 dacă linia nu conţine şirul **t**. Argumentele funcţiei sînt: **s** care reprezintă adresa şirului de studiat, şi **t** care reprezintă configuraţia căutată.

Pentru afişarea unei linii folosim funcţia **puts** din bibliotecă, de asemenea cunoscută din primul capitol.

Programul care realizează structura de mai sus este:

```
#include <stdio.h>
int index(char s[], char t[]) {
          /* returnează poziţia din şirul s unde începe şirul t, sau –1 */
int i,j,k;
for (i=0; s[i]!=0; i++) {
   for (j=i, k=0; (t[k]!=0) && (s[j]==t[k]);
        j++, k++) ;
   if (t[k]==0)
      return i;
   }
return -1;
}
int main() {     /* afişează toate liniile care conţin cuvîntul "the" */
char lin[80];
while (gets(lin)>0)
   if (index(lin,"the")>=0)
      puts(lin);
}
```

2. Funcţia **atof** converteşte un şir de cifre din formatul ASCII într-un număr flotant în precizie dublă. Această funcţie acceptă un şir care conţine reprezentarea unui număr zecimal (eventual cu semn), compus din parte întreagă şi parte fracţionară (oricare putînd lipsi), separate prin punct zecimal.

```
double atof(char s[]) {
double val,fr;
int i,sn;
```

```
sn = 1; i = 0;
if (s[0]=='+' || s[0]=='-')
   sn = (s[i++]=='+') ? 1 : -1;
for (val=0; s[i]>='0' && s[i]<='9'; i++)
   val = 10 * val + s[i] - '0';
if (s[i]== '.')
   i++;
for (fr=1; s[i]>='0' && s[i]<='9'; i++) {
   val = 10 * val +s[i] - '0':
   fr *= 10;
   }
return sn * val / fr;
}
```

3. Funcția **atoi** converteşte un şir de cifre în echivalentul lor numeric întreg.

```
int atoi(char s[]) {
int i,sn,v;
i = 0; sn = 1;
if (s[i]=='+' || s[i]=='-')
   sn = (s[i++]=='+') ? 1 : -1;
for (v=i=0; s[i]>='0' && s[i]<='9'; i++)
   v = 10 * v + s[i] - '0';
return sn * v;
}
```

4. Funcția **lower** converteşte literele mari din setul de caractere ASCII în litere mici. Dacă lower primeşte un caracter care nu este o literă mare atunci îl returnează neschimbat.

```
int lower(int c) {
if (c>='A' && c<='Z')
   return c + 'a' - 'A';
else
   return c;
}
```

5. Funcția **binary** realizează căutarea valorii **x** într-un masiv sortat **v**, care are **n** elemente.

```
int binary(int x, int v[], int n) {
int l,r,m;
l = 0;  r = n - 1;
while (l<=r) {
   m = (l + r) / 2;
   if (v[m]==x) return m;
   if (v[m]<x) l = m + 1;
      else r = m - 1;
   }
return -1;
}
```

Funcția returnează poziția lui **x** (un număr între 0 și **n**−1), dacă **x** apare în **v** sau −1 altfel.

7. Dezvoltarea programelor mari

În multe situații este mai convenabil să fragmentăm un program mare în module mai mici, pe care le putem dezvolta mai uşor independent unele de altele, şi în final să le asamblăm într-un singur program. De altfel acesta este principiul de bază care se aplică la proiectele realizate în colective de programatori.

Dintre modulele (fişierele sursă) care intră în compunerea programului, unul singur conține funcția main. Acest modul apelează funcții şi utilizează variabile care pot fi definite în alte module, cu alte cuvinte sînt *externe*. Deoarece aceste module sînt compilate în mod independent, este necesar un mecanism care să permită o declarare unitară a elementelor puse în comun. Astfel compilatorul poate şti în orice moment cel fel de parametri aşteaptă o anumită funcție, sau în ce context poate fi utilizată o variabilă.

Limbajul C oferă o facilitate de extensie cu ajutorul unui preprocesor de macro-operații simple. Folosirea directivelor de compilare **`#define`** şi **`#include`** este cea mai uzuală metodă pentru extensia limbajului, caracterul # (diez) indicînd faptul că aceste linii vor fi tratate de către preprocesor. Liniile precedate de caracterul # au o sintaxă independentă de restul limbajului şi pot apărea oriunde în fişierul sursă, avînd efect de la punctul de definiție pînă la sfîrşitul fişierului.

7.1. Includerea fişierelor

`#include "`*`nume-fişier`***`"`**

Această linie realizează înlocuirea liniei respective cu întregul conținut al fişierului *nume-fişier*. Fişierul denumit este căutat în primul rînd în directorul fişierului sursă curent şi apoi într-o succesiune de directoare standard, cum ar fi biblioteca I/O standard asociată compilatorului.

Linia următoare caută fişierul *nume-fişier* numai în biblioteca standard şi nu în directorul fişierului sursă:

#include <*nume-fişier***>**

Deseori, o linie sau mai multe linii, de una sau ambele forme apar la începutul fiecărui fişier sursă pentru a include definiţii comune (prin declaraţii **#define** şi declaraţii externe pentru variabilele globale).

Facilitatea de includere a unor fişiere într-un text sursă este deosebit de utilă pentru gruparea declaraţiilor unui program mare. Ea va asigura faptul că toate fişierele sursă vor primi aceleaşi definiţii şi declaraţii de variabile, în felul acesta eliminîndu-se un tip particular de erori. Dacă se modifică un fişier inclus printr-o linie **#include**, toate fişierele care depind de el trebuie recompilate.

7.2. Înlocuirea simbolurilor; substituţii macro

O definiţie de forma:

#define *identificator şir-simboluri*

determină ca preprocesorul să înlocuiască toate apariţiile ulterioare ale identificatorului cu *şir-simboluri* dat (*şir-simboluri* nu se termină cu **;** (punct şi virgulă) deoarece în urma substituţiei identificatorului cu acest şir ar apărea prea multe caractere **;** – unele situaţii conduc chiar la erori de sintaxă.

Şir-simboluri sau textul de înlocuire este arbitrar. În mod normal el este tot restul liniei ce urmează după identificator. O definiţie însă poate fi continuată pe mai multe linii, introducînd ca ultim caracter în linia de continuat caracterul **** (backslash).

Un identificator care este subiectul unei linii **#define** poate fi redefinit ulterior în program printr-o altă linie **#define**. În acest caz, prima definiţie are efect numai pînă la definiţia următoare. Substituţiile nu se realizează în cazul în care identificatorii sînt încadraţi între ghilimele. De exemplu fie definiţia:

```
#define ALFA 1
```

Oriunde va apărea în programul sursă identificatorul **ALFA** el va fi înlocuit cu constanta **1**, cu excepţia unei situaţii de forma:

```
printf("ALFA");
```

în care se va afişa chiar textul *ALFA* şi nu constanta **1**, deoarece identificatorul este încadrat între ghilimele.

Această facilitate este deosebit de valoroasă pentru definirea constantelor simbolice ca în:

```
#define NMAX 100
int tab[NMAX];
```

deoarece într-o eventuală modificare a dimensiunii tabelului **tab** se va modifica doar o singură linie în fişierul sursă. O linie de forma:

#define *identif***(***identif*-1**,**…**,***identif-n***)** *şir-simboluri*

în care nu există spaţiu între primul identificator şi caracterul **(** (paranteză stînga) este o definiţie pentru o macro-operaţie cu argumente, în care textul de înlocuire (*şir-simboluri*) depinde de modul în care se apelează macro-ul respectiv. Ca un exemplu să definim o macro-operaţie numită **max** în felul următor:

```
#define Max(a,b) ((a)>(b) ? (a) : (b))
```

atunci linia dintr-un program sursă:

```
x = Max(p+q,r+s);
```

va fi înlocuită cu:

```
x = ((p+q)>(r+s) ? (p+q) : (r+s));
```

Această macro-definiţie furnizează o „*funcţie maximum*” care se expandează în cod, în loc să se realizeze un apel de funcţie. Acest macro va servi pentru orice tip de date, nefiind nevoie de diferite tipuri de funcţii maximum pentru diferite tipuri de date, aşa cum este necesar în cazul funcţiilor propriu-zise.

Dacă se examinează atent expandarea lui **Max** se pot observa anumite probleme ce pot genera erori, şi anume: expresiile fiind evaluate de două ori, în cazul în care ele conţin operaţii ce generează efecte colaterale (apelurile de funcţii, operatorii de incrementare) se pot obţine rezultate total eronate.

De asemenea, trebuie avută mare grijă la folosirea parantezelor pentru a face sigură ordinea evaluării dorite. De exemplu macro-operaţia **Square(x)** definită prin:

```
#define Square(x) x*x
```

care se apelează în programul sursă sub forma:

```
z = Square(z+1);
```

va produce un rezultat, altul decît cel scontat, datorită priorităţii mai mari a operatorului ***** faţă de cea a operatorului **+**.

7.3. Compilarea condiționată

O linie de control a compilatorului de forma următoare verifică dacă expresia constantă este evaluată la o valoare diferită de zero:

#if *expresie-constantă*

O linie de control de forma următoare verifică dacă identificatorul a fost deja definit:

#ifdef *identificator*

O linie de control de forma următoare verifică dacă identificatorul este nedefinit:

#ifndef *identificator*

Toate cele trei forme de linii de control precedente pot fi urmate de un număr arbitrar de linii care, eventual, pot să conțină o linie de control forma:

#else

şi apoi de o linie de control de forma:

#endif

Dacă condiția supusă verificării este adevărată, atunci orice linie între **#else** şi **#endif** este ignorată. Dacă condiția este falsă atunci toate liniile între testul de verificare şi un **#else** sau în lipsa unui **#else** pînă la **#endif** sînt ignorate.

Toate aceste construcții pot fi imbricate.

7.4. Exemple

1) Se citeşte de la tastatură o pereche de numere naturale **p** şi **q**. Să se determine dacă fiecare din cele două numere este prim sau nu, şi să se calculeze cel mai mare divizor comun.

Să rezolvăm această problemă folosind două fişiere sursă. Primul conține funcția **main** şi apelează două funcții: **eprim** şi **cmmdc**. Al doilea fişier implementează cele două funcții.

Prezentăm mai întîi un fişier header (**numere.h**) care conține declarațiile celor două funcții.

```
#ifndef _Numere_H
#define _Numere_H
unsigned eprim(unsigned n);
```

```
unsigned cmmdc(unsigned p, unsigned q);
#endif
```

Fişierul sursă **princn.c** conţine funcţiile **main** şi **citire**. Să observăm că funcţia **citire** este declarată **static**, de aceea nu poate fi apelată de alte module.

```
#include <stdio.h>
#include "numere.h"
static void citire(unsigned *n) {
scanf("%u",n);
if (eprim(*n))
   printf("%u e prim\n",*n);
else
   printf("%u nu e prim\n",*n);
}
int main() {
unsigned p,q,k;
citire(&p); citire(&q);
k = cmmdc(p,q);
printf("Cmmdc: %u\n",k);
return 0;
}
```

Fişierul sursă **numere.c** este proiectat pentru a putea fi folosit în mai multe aplicaţii care au nevoie de astfel de funcţii. Prin includerea fişierului **numere.h** se garantează că definiţia funcţiilor este conformă cu aşteptările modulului principal, în care este inclus acelaşi fişier header.

```
#include "numere.h"
unsigned eprim(unsigned n) {
unsigned i,a,r;
if (n==0) return 0;
if (n<4) return 1;
if ((n&1)==0) return 0;
for (i=3; ; i+=2) {
   r = n%i;
   if (r==0) return 0;
   a = n/i;
   if (a<=i) return 1;
```

```
   }
}
unsigned cmmdc(unsigned p, unsigned q) {
while ((p>0) && (q>0))
   if (p>q) p %= q;
      else q %= p;
if (p==0) return q;
   else return p;
}
```

Pentru a obține un program executabil din aceste două fișiere se poate utiliza o singură comandă de compilare:

Cc **`princ.c numere.c`** *opțiuni-de-compilare*

unde ***Cc*** este numele compilatorului folosit (exemplu: **`bcc`**, **`gcc`**). Opțiunile de compilare (atunci cînd sînt necesare) sînt specifice mediului de programare folosit.

2) Al doilea proiect gestionează o listă de valori numerice asupra căreia se efectuează următoarele operații:
– inițializare, operație realizată automat la lansarea în execuție a programului;
– adăugarea unei valori în listă, dacă aceasta nu există deja;
– interogare: o anumită valoare se află în listă?
– ştergerea unei valori din listă;
– menținerea în permanență a unei relații de ordine crescătoare.

Masivul **`Lis[]`** memorează lista de valori, care are un număr de **`ne`** elemente, inițial zero.

Prezentăm mai întîi un fişier header (**`lista.h`**) care conține declarațiile celor trei funcții, precum şi a variabilelor externe **`Lis`** şi **`ne`**.

```
#ifndef _Lista_H
#define _Lista_H
extern double Lis[];
extern int ne;
int Adaugare(double v);
int Stergere(double v);
int Interogare(double v);
```

```
#endif
```

Fişierul sursă **princl.c** conține funcția **main**. Acesta aşteaptă de la terminal opțiunea utilizatorului, urmată de un spațiu şi de valoarea care se doreşte a fi prelucrată. În funcție de opțiune se apelează funcția corespunzătoare sau se afişează un mesaj de eroare. După fiecare operație se afişează lista valorilor.

```
#include <stdio.h>
#include "lista.h"
int main() {
double atof(char s[]);
double v;
int i;
char o[80];
do {
   gets(o);  v = atof(op+2);
   switch (o[0]) {
      case 'a':
         i = Adaugare(v);
         break;
      case 's':
         if (Stergere(v))
            puts("S-a sters");
         else puts("Nu exista");
         break;
      case 'i':
         i = Interogare(v);
         if (i>=0)
            printf("Pozitia: %d\n",i);
         else puts("Nu exista");
         break;
      default:
         puts("Optiune eronata");
         break;
      }
   puts("Lista:");
   for (i=0; i<ne; i++)
      printf(" %.3lf",Lis[i]);
   puts("");
   } while (o!='t');
```

```
return 0;
}
```

Fişierul sursă **lista.c** defineşte variabilele **Lis[]** şi **ne**, şi funcţiile **Adaugare**, **Interogare**, **Stergere**.

Funcţia **Adaugare** adaugă o valoare **v** în listă dacă aceasta nu există. Valorile mai mari sînt deplasate spre dreapta cu o poziţie pentru a face loc noii valori. Returnează poziţia valorii în listă după (o eventuală) adăugare.

Funcţia **Stergere** şterge o valoare **v** din listă dacă aceasta există. Valorile mai mari sînt deplasate spre stînga cu o poziţie pentru a păstra lista compactă. Returnează *True* dacă operaţia a reuşit şi *False* în caz contrar.

Funcţia **Interogare** returnează poziţia pe care se află valoarea **v** în listă, dacă aceasta există, sau **-1** în caz contrar. Deoarece lista este ordonată se face o căutare binară. Această funcţie este apelată şi de funcţia **Adaugare**.

```
#include "lista.h"
double Lis[100];
int ne;

int Adaugare(double v) {
int k;
k = Interogare(v);
if (k<0) {
   for (k=ne-1; k>=0; k--)
       if (Lis[k]>v)
          Lis[k+1] = Lis[k];
      else break;
   Lis[k+1] = v;  ne++;
   }
return k;
}

int Stergere(double v) {
int k;
k = Interogare(v);
if (k<0) return 0;
for (; k<ne; k++)
```

```
    Lis[k] = Lis[k+1];
ne--;
return 1;
}

int Interogare(double v) {
int l,r,m;
double d;
l = 0; r = ne - 1;
while (l<=r) {
    m = (l + r) / 2;  d = Lis[m] - v;
    if (d==0) return m;
    if (d<0) l = m + 1;
       else r = m - 1;
    }
return -1;
}
```

Proiectele – programe de dimensiuni mari – sînt compuse de cele mai multe ori din module care se elaborează şi se pun la punct în mod independent. Uneori pot fi identificate funcţii care prezintă un interes general mai mare, şi care pot fi utilizate în elaborarea unor tipuri de aplicaţii foarte diverse. Acestea sînt dezvoltate separat şi, după ce au fost puse la punct în totalitate, se depun într-o bibliotecă de funcţii în format obiect. În momentul în care acestea sînt incluse în proiecte nu se mai pierde timp cu compilarea modulelor sursă. În schimb modulele care le apelează au nevoie de modul cum sînt definite aceste funcţii, şi acesta se păstrează ca şi declaraţii în fişiere header.

Fişierele header păstrează şi alte informaţii comune mai multor module, cum sînt de exemplu tipurile definite de programator, cel mai adesea tipuri structurate (detalii în capitolul nouă).

8. Pointeri şi masive

Un pointer este o variabilă care conţine adresa unei alte variabile. Pointerii sînt foarte mult utilizaţi în programe scrise în C, pe de o parte pentru că uneori sînt unicul mijloc de a exprima un calcul, iar pe de altă parte pentru că oferă posibilitatea scrierii unui program mai compact şi mai eficient decît ar putea fi obţinut prin alte căi.

8.1. Pointeri şi adrese

Deoarece un pointer conţine adresa unui obiect, cu ajutorul lui putem avea acces, în mod indirect, la acea variabilă (obiect).

Să presupunem că **x** este o variabilă de tip întreg şi **px** un pointer la un întreg. Atunci aplicînd operatorul unar **&** lui **x**, instrucţiunea:

```
px = &x;
```

atribuie variabilei **px** adresa variabilei **x**; în acest fel spunem că **px** indică (pointează) spre **x**.

Invers, dacă **px** conţine adresa variabilei **x**, atunci instrucţiunea:

```
y = *px;
```

atribuie variabilei y conţinutul locaţiei pe care o indică **px**.

Variabila **x** are asociată o zonă de memorie, şi această zonă are (presupunem) adresa 8A54. După atribuirea **px = &x**, variabila **px** va avea valoarea 8A54.

Evident toate variabilele care sînt implicate în aceste instrucţiuni trebuie declarate. Aceste declaraţii sînt:

```
int x,y,*px;
```

Declaraţiile variabilelor **x** şi **y** sînt deja cunoscute. Declaraţia pointerului **px** este o noutate. Aceasta indică faptul că o combinaţie de forma ***px** este un întreg, iar variabila **px** care apare în contextul ***px** este echivalentă cu un pointer la o variabilă de tip întreg. În locul tipului întreg poate apărea oricare dintre tipurile admise în limbaj şi se referă la obiectele pe care le indică **px**.

Pointerii pot apărea şi în expresii; în expresia următoare variabilei **y** i se atribuie valoare variabilei **x** plus **1**:

```
y = *px + 1;
```

Instrucţiunea următoare are ca efect convertirea valorii variabilei **x** pe care o indică **px** în tip **double** şi apoi depunerea rădăcinii pătrate a valorii astfel convertite în variabila **d**:

```
d = sqrt((double)*px);
```

Referiri la pointeri pot apărea de asemenea şi în partea stîngă a atribuirilor. Dacă, de exemplu, **px** indică spre **x**, atunci următoarea instrucţiune atribuie variabilei **x** valoarea zero:

```
*px = 0;
```

Instrucţiunile următoare sînt echivalente, şi incrementează valoarea variabilei **x** cu **1**:

```
*px += 1;
(*px)++;
```

În acest ultim exemplu parantezele sînt obligatorii deoarece, în lipsa lor, expresia ar incrementa pe **px** în loc de valoarea variabilei pe care o indică (operatorii unari *****, **++** au aceeaşi precedenţă şi sînt evaluaţi de la dreapta spre stînga).

Pointeri de tip `void` (tip generic de pointer)

Un pointer poate fi declarat de tip **void**. În această situaţie pointerul nu poate fi folosit pentru indirectare (dereferenţiere) fără un *cast* explicit, şi aceasta deoarece compilatorul nu poate cunoaşte tipul obiectului pe care pointerul îl indică.

```
int i;
float f;
void *p;
p = &i;              /* p indică spre i */
*(int *)p = 2;
p = &f;              /* p indică spre f */
*(float *)p = 1.5;
```

Pointerii generici sînt foarte utilizaţi de unele funcţii de bibliotecă, atunci cînd este nevoie să se opereze cu zone de memorie care pot conţine informaţii de orice natură: operaţii de intrare / ieşire

fără format (capitolul zece), alocări de memorie, sortare şi căutare (capitolul unsprezece). Aceste funcţii nu cunosc tipul datelor cu care operează, pentru aceste funcţii zonele de memorie *nu au un tip precizat*. Acest fapt este evidenţiat prin pointeri de tip **void**.

8.2 Pointeri şi argumente de funcţii

Dacă transmiterea argumentelor la funcţii se face prin valoare (şi nu prin referinţă), funcţia apelată nu are posibilitatea de a altera o variabilă din funcţia apelantă. Problema care se pune este cum procedăm dacă dorim să modificăm valoarea unui argument?

De exemplu, o rutină de sortare trebuie să schimbe între ele două elemente care nu respectă ordinea dorită, cu ajutorul unei funcţii **swap**. Fie funcţia **swap** definită astfel:

```
void swap(int x, int y) {       /* greşit */
int tmp;
tmp = x;
x = y;
y = tmp;
}
```

Funcţia **swap** apelată prin **swap(a,b)** nu va realiza acţiunea dorită deoarece ea nu poate afecta argumentele **a** şi **b** din rutina apelantă.

Există însă o posibilitate de a obţine efectul dorit, dacă funcţia apelantă transmite ca argumente pointeri la valorile ce se doresc interschimbate. Atunci în funcţia apelantă apelul va fi:

```
swap(&a,&b);
```

iar forma corectă a lui **swap** este:

```
void swap(int *px, int *py) {/* interschimbă *px şi *py */
int tmp;
tmp = *px;
*px = *py;
*py = tmp;
}
```

Modificarea nu este permisă dacă parametrul de tip pointer este precedat de declaratorul **const**.

8.3. Pointeri şi masive

În limbajul C există o strînsă legătură între pointeri şi masive. Orice operaţie care poate fi realizată prin indicarea masivului poate fi de asemenea făcută prin pointeri, care conduce în plus la o accelerare a operaţiei. Declaraţia:

```
int a[10];
```

defineşte un masiv de dimensiune 10, care reprezintă un bloc de 10 obiecte consecutive numite **a[0]**, ... **a[9]**. Notaţia **a[i]** reprezintă al **i**-lea element al masivului sau elementul din poziţia **i+1**, începînd cu primul element. Dacă **pa** este un pointer la un întreg, şi **a** este un masiv de întregi, atunci instrucţiunea de atribuire încarcă variabila **pa** cu adresa primului element al masivului **a**.

```
int *pa,a[10];
pa = &a[0];
```

Atribuirea următoare copiază conţinutul lui **a[0]** în **x**:

```
x = *pa;
```

Dacă **pa** indică un element particular al unui masiv **a**, atunci prin definiţie **pa+i** indică un element cu **i** poziţii după elementul pe care îl indică **pa**, după cum **pa-i** indică un element cu **i** poziţii înainte de cel pe care indică **pa**. Astfel, dacă variabila **pa** indică pe **a[0]** atunci ***(pa+i)** se referă la conţinutul lui **a[i]**.

Aceste observaţii sînt adevărate indiferent de tipul variabilelor din masivul **a**.

Întreaga aritmetică cu pointeri are în vedere faptul că expresia **pa+i** înseamnă de fapt înmulţirea lui **i** cu lungimea elementului pe care îl indică **pa** şi adunarea apoi la **pa**, obţinîndu-se astfel adresa elementului de indice **i** al masivului.

Corespondenţa dintre indexarea într-un masiv şi aritmetica de pointeri este foarte strînsă. De fapt, o referire la un masiv este convertită de compilator într-un pointer la începutul masivului. Efectul este că un nume de masiv este o expresie pointer, deoarece numele unui masiv este identic cu numele elementului de indice zero din masiv.

Cele două instrucţiuni de atribuire sînt identice:

```
pa = &a[1];
```

```
pa = a + 1;
```

De asemenea, expresiile **a[i]** şi ***(a+i)** sînt identice. Aplicînd operatorul **&** la ambele părţi obţinem **&a[i]** identic cu **a+i**. Pe de altă parte, dacă **pa** este un pointer, expresiile pot folosi acest pointer ca un indice: **pa[i]** este identic cu ***(pa+i)**. Pe scurt orice expresie de masiv şi indice poate fi scrisă ca un pointer şi un deplasament şi invers, chiar în aceeaşi instrucţiune.

Există însă o singură diferenţă între un nume de masiv şi un pointer la începutul masivului. Un pointer este o variabilă, deci următoarele instrucţiuni sînt corecte:

```
pa = a;
pa++;
```

Dar un nume de masiv este o constantă şi deci următoarele construcţii nu sînt permise:

```
a = pa;
a++;
p = &a;
```

Cînd se transmite unei funcţii un nume de masiv, ceea ce se transmite de fapt este adresa primului element al masivului. Aşadar, un nume de masiv, argument al unei funcţii, este în realitate un pointer, adică o variabilă care conţine o adresă. Fie de exemplu funcţia **strlen** care determină lungimea şirului **s**:

```
int strlen(const char *s) {
int n;
for (n=0; *s; s++) n++;
return n;
}
```

Incrementarea lui **s** este legală deoarece **s** este o variabilă pointer. **s++** nu afectează şirul de caractere din funcţia care apelează pe **strlen**, ci numai copia adresei şirului din funcţia **strlen**.

Este posibil să se transmită unei funcţii, ca argument, numai o parte a unui masiv, printr-un pointer la începutul sub-masivului respectiv. De exemplu, dacă `a` este un masiv, atunci:

```
f(&a[2])
f(a+2)
```

transmit funcției **f** adresa elementului **a[2]**, deoarece **&a[2]** şi **a+2** sînt expresii pointer care, ambele, se referă la al treilea element al masivului **a**. În cadrul funcției **f** argumentul se poate declara astfel:

```
f(int arr[]) {...}
f(int *arr) {...}
```

Declarațiile **int arr[]** şi **int *arr** sînt echivalente.

8.4. Aritmetica de adrese

Dacă **p** este un pointer, atunci următoarea instrucțiune incrementează pe **p** pentru a indica cu **i** elemente după elementul pe care îl indică în prealabil **p**:

```
p += i;
```

Această construcție şi altele similare sînt cele mai simple şi comune formule ale aritmeticii de adrese, care constituie o caracteristică puternică a limbajului C.

Pentru exemplificare reluăm inversarea caracterelor unui şir:

```
void strrev(char s[]) {
int i,j;
char c;
for (i=0,j=strlen(s)-1; i<j; i++,j--) {
   c = s[i];
   s[i] = s[j];
   s[j] = c;
   }
}
```

Versiunea cu pointeri a acestei funcții este:

```
void strrev(char s[]) {
char *p,*q,c;
for (p=s,q=s+strlen(s)-1; p<q; p++,q--) {
   c = *p;
   *p = *q;
   *q = c;
   }
}
```

Exemplul de mai sus demonstrează cîteva din facilităţile aritmeticii de adrese (pointeri). În primul rînd, pointerii pot fi comparaţi în anumite situaţii. Relaţiile **<**, **<=**, **>**, **>=**, **==**, **!=** sînt valide *numai dacă* **p** *şi* **q** *sînt pointeri la elementele aceluiaşi masiv*. Relaţia **p<q**, de exemplu, este adevărată dacă **p** indică un element mai apropiat de începutul masivului decît elementul indicat de pointerul **q**. Comparările între pointeri pot duce însă la rezultate imprevizibile dacă ei se referă la elemente aparţinînd la masive diferite.

Se observă că pointerii şi întregii pot fi adunaţi sau scăzuţi. Construcţia de forma **p+n** înseamnă adresa celui de-al **n**-lea element după cel indicat de **p**, indiferent de tipul elementului pe care îl indică **p**. Compilatorul C aliniază valoarea lui **n** conform dimensiunii elementelor pe care le indică **p**, dimensiunea fiind determinată din declaraţia lui **p** (scara de aliniere este 1 pentru **char**, 2 pentru **short int** etc).

Dacă **p** şi **q** indică elemente ale aceluiaşi masiv, **p-q** este numărul elementelor dintre cele pe care le indică **p** şi **q**. Să scriem alte versiuni ale funcţiei **strlen** folosind această ultimă observaţie:

```
int strlen(char *s) {
char *p;
p = s;
while (*p) p++;
return p-s;
}
```

```
int strlen(char *s) {
char *p;
for (p=s; *p; p++) ;
return p-s;
}
```

În acest exemplu **s** rămîne constant cu adresa de început a şirului, în timp ce **p** avansează la următorul caracter de fiecare dată. Diferenţa **p-s** dintre adresa ultimului element al şirului şi adresa primului element al şirului indică numărul de elemente.

> Sînt admise de asemenea incrementările şi decrementările precum şi alte combinaţii ca de exemplu ***++p** şi ***--p**.

În afară de operaţiile binare menţionate (adunarea sau scăderea pointerilor cu întregi şi scăderea sau compararea a doi pointeri), celelalte operaţii cu pointeri sînt ilegale. Nu este permisă adunarea,

înmulțirea, împărțirea sau deplasarea pointerilor, după cum nici adunarea lor cu constante de tip **double** sau **float**.

8.5. Pointeri la caracter şi funcţii

O constantă şir este un masiv de caractere, care în reprezentarea internă este terminat cu caracterul **0**, astfel încît programul poate depista sfîrşitul lui. Lungimea acestui şir în memorie este astfel cu 1 mai mare decît numărul de caractere ce apar efectiv între ghilimelele de început şi sfîrşit de şir.

Cea mai frecventă apariție a unei constante şir este ca argument la funcții, caz în care accesul la ea se realizează prin intermediul unui pointer.

În exemplul:

```
puts("Buna dimineata");
```

funcția **puts** primeşte de fapt un pointer la masivul de caractere.

În prelucrarea unui şir de caractere sînt implicați numai pointeri, limbajul C neoferind nici un operator care să trateze şirul de caractere ca o unitate de informație.

Vom prezenta cîteva aspecte legate de pointeri şi masive analizînd două exemple. Să considerăm pentru început funcția **strcpy(t,s)** care copiază şirul **s** (source) peste şirul **t** (target). O primă versiune a programului ar fi următoarea:

```
void strcpy(char t[], const char s[]) {
int i;
i = 0;
while (t[i]=s[i]) i++;
}
```

O a doua versiune cu ajutorul pointerilor este următoarea:

```
void strcpy(char *t, const char *s) {
while (*t++=*s++) ;
}
```

Această versiune cu pointeri modifică prin incrementare pe **t** şi **s** în partea de test. Valoarea lui ***s++** este caracterul indicat de

pointerul **s**, înainte de incrementare. Notația postfix **++** asigură că **s** va fi modificat după preluarea conținutului indicat de el, de la vechea poziție a lui **s**, după care **s** se incrementează. La fel, **t** va fi modificat după memorarea caracterului la vechea poziție a lui **t**, după care şi **t** se incrementează. Efectul este că se copiază caracterele şirului **s** în şirul **t** pînă la caracterul terminal **0** inclusiv.

Am preferat să evităm redundanța comparării cu caracterul **0**, facilitate care rezultă din structura instrucțiunii **while**. Am fi putut scrie şi astfel:

```
while ((*t++=*s++)!=0) ;
```

Să considerăm, ca al doilea exemplu, funcția **strcmp(t,s)** care compară caracterele şirurilor **t** şi **s** şi returnează o valoare negativă, zero sau pozitivă, după cum şirul **t** este lexicografic mai mic, egal sau mai mare ca şirul **s**. Valoarea returnată se obține prin scăderea caracterelor primei poziții în care **t** şi **s** diferă.

O primă versiune a funcției **strcmp(t,s)** este următoarea:

```
int strcmp(const char t, const char s) {
int i;
i = 0;
while (t[i]==s[i])
   if (t[i++]==0)
      return 0;
return t[i]-s[i];
}
```

O versiune cu pointeri a aceleiaşi funcții este:

```
int strcmp(const char *t, const char *s) {
for (; *t==*s; t++,s++)
   if (*t==0)
      return 0;
return *t-*s;
}
```

8.6. Masive multidimensionale

Limbajul C oferă facilitatea utilizării masivelor multi-dimensionale, deşi în practică ele sînt folosite mai rar decît masivele de pointeri.

Masivele multi-dimensionale sînt folosite mai ales de programatorii începători în scop didactic.

Să considerăm problema conversiei datei, de la zi din lună, la zi din an şi invers, ţinînd cont de faptul că anul poate să fie bisect sau nu. Definim două funcţii care să realizeze cele două conversii.

Funcţia **NrZile** converteşte ziua şi luna în numărul de zile de la începutul anului, şi funcţia **LunaZi** converteşte numărul de zile în lună şi zi.

Ambele funcţii au nevoie de aceeaşi informaţie şi anume un tabel cu numărul zilelor din fiecare lună. Deoarece numărul zilelor din lună diferă pentru anii bisecţi de cele pentru anii nebisecţi este mai uşor să considerăm un tabel bidimensional în care prima linie să corespundă numărului de zile ale lunilor pentru anii nebisecţi, iar a doua linie să corespundă numărului de zile pentru anii bisecţi. În felul acesta nu trebuie să ţinem o evidenţă în timpul calculului a ceea ce se întîmplă cu luna februarie. Atunci masivul bidimensional care conţine informaţiile pentru cele două funcţii este următorul:

```
static int Zile[2][13] = {
       {0,31,28,31,30,31,30,31,31,30,31,30,31},
       {0,31,29,31,30,31,30,31,31,30,31,30,31}
       };
```

Masivul **Zile** trebuie să fie declarat global pentru a putea fi folosit de ambele funcţii.

În limbajul C, prin definiţie, un masiv cu două dimensiuni este în realitate un masiv cu o dimensiune ale cărui elemente sînt masive. De aceea indicii se scriu sub forma **[i][j]** în loc de **[i,j]**, cum se procedează în multe alte limbaje. Într-un masiv bidimensional elementele sînt memorate pe linie, adică indicele cel mai din dreapta variază cel mai rapid.

Un masiv se iniţializează cu ajutorul unei liste de iniţializatori închişi între acolade; fiecare linie a unui masiv bidimensional se iniţializează cu ajutorul unei subliste de iniţializatori. În cazul

exemplului nostru, masivul **Zile** începe cu o coloană zero, pentru ca numerele lunilor să fie între 1 şi 12 şi nu între 0 şi 11, aceasta pentru a nu face modificări în calculul indicilor.

Şi atunci funcţiile care realizează conversiile cerute de exemplul nostru sînt:

```
int NrZile (int an, int luna, int zi) {
int i,bis;
bis = !(an%4) && (an%100) || !(an%400);
for (i=1; i<luna; i++)
   zi += Zile[bis][i];
return zi;
}
```

Deoarece variabila **bis** poate lua ca valori numai zero sau unu, după cum expresia (scrisă de data aceasta explicit):

```
(an%4==0) && (an%100!=0) || (an%400==0)
```

este falsă sau adevărată, ea poate fi folosită ca indice de linie în tabelul **Zile** care are doar două linii în exemplul nostru.

```
void LunaZi(int an, int zian, int *plun, int *pzi) {
int i,bis;
bis = !(an%4) && (an%100) || !(an%400);
for (i=1; zian>Zile[bis][i]; i++)
   zian -= Zile[bis][i];
*plun = i;  *pzi = zian;
}
```

Deoarece această ultimă funcţie returnează două valori, argumentele care corespund lunii şi zilei sînt pointeri.

Exemplu. **LunaZi(1984,61,&m,&d)** va încărca pe **m** cu 3, iar pe **d** cu 1 (adică 1 martie).

Dacă un masiv bidimensional trebuie transmis unei funcţii, declaraţia argumentelor funcţiei trebuie să includă dimensiunea coloanei. Dimensiunea liniei nu este necesar să apară în mod obligatoriu, deoarece ceea ce se transmite de fapt este un pointer la masive de cîte 13 întregi, în cazul exemplului nostru. Astfel, dacă masivul **Zile** trebuie transmis unei funcţii **f**, atunci declaraţia lui **f** poate fi:

```
f(int (*Zile)[13]);
```

```
f(int Zile[][13]);
```

unde declarația **(*Zile)[13])** indică faptul că argumentul lui **f** este un pointer la un masiv de 13 întregi.

În general, un masiv *d*-dimensional **a[i][j]…[l]** de rangul **i*j*…*l** este un masiv (*d*–1)-dimensional de rangul **j*k*…*l** ale cărui elemente, fiecare, sînt masive (*d*–2)-dimensionale de rang **k*…*l** ale cărui elemente, fiecare, sînt masive (*d*–3)-dimensionale ş.a.m.d. Oricare dintre expresiile următoare: **a[i]**, **a[i][j]**..., **a[i][j]…[l]** pot apărea în expresii. Prima are tipul masiv, ultima are tipul **int**, de exemplu, dacă masivul este de tipul **int**. Vom mai reveni asupra acestei probleme cu detalii.

8.7. Masive de pointeri şi pointeri la pointeri

Deoarece pointerii sînt variabile, are sens noțiunea de masiv de pointeri. Vom ilustra modul de lucru cu masive de pointeri pe un exemplu.

Să scriem un program care să sorteze lexicografic liniile de lungimi diferite ale unui text, linii care spre deosebire de întregi nu pot fi comparate sau schimbate printr-o singură operație.

Citirea textului decurge astfel. Se citește o linie din textul introdus de la intrarea standard, şirul de caractere fiind depus într-un masiv **lin**. Deoarece şi următoarele linii vor fi depuse tot aici, este nevoie să se salveze conținutul acestui masiv. În acest scop se cere sistemului de operare să se creeze un duplicat al acestui şir prin apelul funcției **strdup**. Aceasta va rezerva o zonă de memorie, va copia şirul dat în zona respectivă şi ne va comunica adresa acestei zone.

Funcțiile care operează cu şiruri de caractere sînt declarate cu:

```
#include <string.h>
```

Adresele liniilor sînt memorate într-un masiv de pointeri la caracter. Astfel două linii de text pot fi comparate transmițînd pointerii lor funcției **strcmp**. Dacă două linii care nu respectă ordinea trebuie să fie schimbate, se schimbă doar pointerii lor din masivul de pointeri şi nu textul efectiv al liniilor.

În primul pas al procesului de sortare se citesc toate liniile textului de la intrare. În al doilea pas se sortează liniile în ordine lexicografică. În ultimul pas se afişează liniile sortate în noua ordine.

Vom scrie programul prin funcţiile sale, fiecare funcţie realizînd unul din cei trei paşi de mai sus. O rutină principală va controla cele trei funcţii. Ea are următorul cod:

```
int main() {          /* program care sortează liniile de la intrare */
#define NLIN 100              /* nr maxim de linii de sortat */
char *linptr[NLIN];           /* pointeri la linii */
int nrl;                      /* nr linii intrare citite */
nrl = ReadLines(linptr,NLIN);
if (nrl>=0) {
   Sort(linptr,nrl);
   WriteLines(linptr,nrl);
   }
else puts("Intrare prea mare pentru sort");
return 0;
}
```

Cele 3 funcţii care realizează întregul proces sînt: **ReadLines**, **Sort** şi **WriteLines**.

Rutina de intrare **ReadLines** trebuie să memoreze caracterele fiecărei linii şi să construiască un masiv de pointeri la liniile citite. Trebuie, de asemenea, să numere liniile din textul de la intrare, deoarece această informaţie este necesară în procesul de sortare şi de afişare. Întrucît funcţia de intrare poate prelucra numai un număr finit de linii de intrare, ea poate returna un număr ilegal, cum ar fi −1, spre a semnala că numărul liniilor de intrare este prea mare pentru capacitatea de care dispune.

Atunci funcţia **ReadLines** care citeşte liniile textului de la intrare este următoarea:

```
int ReadLines(char *linptr[], int nmax) {
int nrl = 0;
char *p,lin[80];
while (gets(lin)) {
   if (nrl>=nmax) return -1;
   p = strdup(lin);
```

```
   if (!p) return -1;
   linptr[nrl++] = p;
   }
return nrl;
}
```

Rutina care afişează liniile în noua lor ordine este **WriteLines** şi are următorul cod:

```
void WriteLines(char *linptr[], int nrl) {
int i;
for (i=0; i<nrl; i++)
   puts(linptr[i]);
}
```

Declaraţia nouă care apare în aceste programe este:

```
   char *linptr[NLIN];
```

care indică faptul că **linptr** este un masiv de **NLIN** elemente, fiecare element al masivului fiind un pointer la un caracter. Astfel **linptr[i]** este un pointer la un caracter, iar ***linptr[i]** permite accesul la caracterul respectiv.

Deoarece **linptr** este el însuşi un masiv, care se transmite ca argument funcţiei **WriteLines**, el va fi tratat ca un pointer şi atunci funcţia **WriteLines** mai poate fi scrisă şi astfel:

```
void WriteLines(char *linptr[], int nrl) {
while (--nrl>=0)
   puts(*linptr++);
}
```

În funcţia **puts**, **linptr** indică iniţial prima linie de afişat; fiecare incrementare avansează pe ***linptr** la următoarea linie de afişat, în timp ce **nrl** se micşorează după fiecare afişare a unei linii cu 1.

Funcţia care realizează sortarea efectivă a liniilor se bazează pe algoritmul de înjumătăţire şi are următorul cod:

```
void Sort(char *v[], int n) {
int gap,i,j;
char *tmp;
for (gap=n/2; gap>0; gap/=2)
```

```
   for (i=gap; i<n; i++)
      for (j=i-gap; j>=0; j-=gap) {
         if (strcmp(v[j],v[j+gap])<=0)
            break;
         tmp = v[j];
         v[j] = v[j+gap];
         v[j+gap] = tmp;
         }
}
```

Deoarece fiecare element al masivului **v** (care este de fapt masivul **linptr** din funcția **main**) este un pointer la primul caracter al unei linii, variabila **tmp** va fi şi ea un pointer la un caracter, deci operațiile de atribuire din ciclu după variabila **j** sînt admise şi ele realizează reinversarea pointerilor la linii dacă ele nu sînt în ordinea cerută.

Structura programului este următoarea:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int ReadLines(...) {...}
void WriteLines(...) {...}
void Sort(...) {...}
int main(...) {...}
```

Să reținem următoarele lucruri legate de masive şi pointeri. De cîte ori apare într-o expresie un identificator de tip masiv el este convertit într-un pointer la primul element al masivului. Prin definiție, operatorul de indexare **[]** este interpretat astfel încît ***E***1**[*****E***2**]** este identic cu ***((*****E**1**)+(*****E**2**))**. Dacă ***E***1 este un masiv, iar ***E***2 un întreg, atunci ***E***1**[*****E***2**]** se referă la elementul de indice ***E***2 al masivului ***E***1.

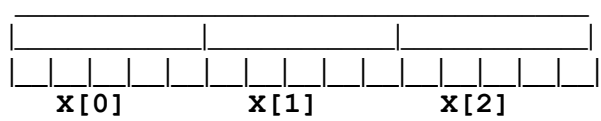
O regulă corespunzătoare se aplică şi masivelor multi-dimensionale. Dacă ***E***1 este un masiv *d*-dimensional, de rangul **i*j*...*k**, atunci ori de cîte ori e1 apare într-o expresie, e1 va fi convertit într-un pointer la un masiv (*d*–1)-dimensional de rangul **j*...*k**, ale cărui elemente sînt masive. Dacă operatorul ***** se aplică

acestui pointer, rezultatul este masivul (d–1)-dimensional, care se va converti imediat într-un pointer la un masiv (d–2)-dimensional ş.a.m.d. Raţionamentul se poate aplica în mod inductiv pînă cînd, în final, ca urmare a aplicării operatorului ***** se obţine ca rezultat un întreg, de exemplu, dacă masivul a fost declarat de tipul **int**.

Să considerăm, de exemplu, masivul:

```
int X[3][5];
```

X este un masiv de întregi, de rangul 3×5. Cînd **X** apare într-o expresie, el este convertit într-un pointer la (primul din cele trei) masive de 5 întregi.



În expresia **X[i]**, care este echivalentă cu expresia ***(X+i)**, **X** este convertit într-un pointer la un masiv, ale cărui elemente sînt la rîndul lor masive de 5 elemente; apoi **i** se converteşte la tipul **X**, adică indicele **i** se înmulţeşte cu lungimea elementului pe care îl indică **X** (adică 5 întregi) şi apoi rezultatele se adună. Se aplică operatorul ***** pentru obţinerea masivului **i** (de 5 întregi) care la rîndul lui este convertit într-un pointer la primul întreg din cei cinci.

Se observă deci că primul indice din declaraţia unui masiv nu joacă rol în calculul adresei.

8.8. Iniţializarea masivelor şi masivelor de pointeri

Iniţializatorul unei variabile declarate masiv constă dintr-o listă de iniţializatori separaţi prin virgulă şi închişi între acolade, corespunzători tuturor elementelor masivului. Ei sînt scrişi în ordinea crescătoare a indicilor masivului. Dacă masivul conţine sub-masive atunci regula se aplică recursiv membrilor masivului. Dacă în lista de iniţializare există mai puţini iniţializatori decît elementele masivului,

restul elementelor neinițializate se inițializează cu zero. Nu se admite inițializarea unui masiv de clasă automatic.

Acoladele **{** şi **}** se pot omite în următoarele situații:

– dacă inițializatorul începe cu acoladă stînga **{**, atunci lista de inițializatori, separați prin virgulă, va inițializa elementele masivului; nu se acceptă să existe mai mulți inițializatori decît numărul elementelor masivului;

– dacă însă inițializatorul nu începe cu acoladă stînga **{**, atunci se iau din lista de inițializatori atîția inițializatori cîți corespund numărului de elemente ale masivului, restul inițializatorilor vor inițializa următorul membru al masivului, care are ca parte (sub-masiv) masivul deja inițializat.

Un masiv de caractere poate fi inițializat cu un şir, caz în care caracterele succesive ale şirului inițializează elementele masivului.

***Exemple*:**

1) **`int x[] = {1,3,5};`**

Această declarație definește şi inițializează pe **`x`** ca un masiv unidimensional cu trei elemente, chiar dacă nu s-a specificat dimensiunea masivului. Prezența inițializatorilor închişi între acolade determină dimensiunea masivului.

2) Declarația

```
int y[4][3]={
      {1,3,5},
      {2,4,6},
      {3,5,7}
      };
```

este o inițializare complet închisă între acolade. Valorile 1,3,5 inițializează prima linie a masivului **`y[0]`** şi anume pe **`y[0][0]`**, **`y[0][1]`**, **`y[0][2]`**. În mod analog următoarele două linii inițializează pe **`y[1]`** şi **`y[2]`**. Deoarece inițializatorii sînt mai putini decît numărul elementelor masivului, linia **`y[3]`** se va inițializa cu zero, respectiv elementele **`y[3][0]`**, **`y[3][1]`**, **`y[3][2]`** vor avea valorile zero.

3) Acelaşi efect se poate obține din declarația:

```
int y[4][3] = {1,3,5,2,4,6,3,5,7};
```

unde inițializatorul masivului **y** începe cu acolada stîngă în timp ce inițializatorul pentru masivul **y[0]** nu, fapt pentru care primii trei inițializatori sînt folosiți pentru inițializarea lui **y[0]**, restul inițializatorilor fiind folosiți pentru inițializarea masivelor **y[1]** şi respectiv **y[2]**.

4) Declarația:

```
int y[4][3] = {
      {1},{2,},{3,},{4}
      };
```

inițializează masivul **y[0]** cu (1,0,0), masivul **y[1]** cu (2,0,0), masivul **y[2]** cu (3,0,0) şi masivul **y[4]** cu (4,0,0).

5) Declarația:

```
static char msg[] = "Eroare de sintaxa";
```

inițializează elementele masivului de caractere **msg** cu caracterele succesive ale şirului dat.

În ceea ce priveşte inițializarea unui masiv de pointeri să considerăm următorul exemplu.

Fie funcția **NumeLuna** care returnează un pointer la un şir de caractere care indică numele unei luni a anului. Funcția dată conține un masiv de şiruri de caractere şi returnează un pointer la un astfel de şir, cînd ea este apelată.

Codul funcției este următorul:

```
char *NumeLuna(int n) { /* returnează numele lunii a n-a */
static char *NL[] = {
   "luna ilegala", "ianuarie", "februarie",
   "martie", "aprilie", "mai", "iunie", "iulie",
   "august", "septembrie", "octombrie",
   "noiembrie", "decembrie"
   }
return (n<1) || (n>12) ? NL[0] : NL[n] ;
}
```

În acest exemplu, **NL** este un masiv de pointeri la caracter, al cărui inițializator este o listă de şiruri de caractere. Compilatorul alocă o zonă de memorie pentru memorarea acestor şiruri şi

generează cîte un pointer la fiecare din ele pe care apoi îi introduce în masivul name. Deci **NL[i]** va conține un pointer la şirul de caractere avînd indice **i** al inițializatorului. Nu este necesar să se specifice dimensiunea masivului **NL** deoarece compilatorul o determină numărînd inițializatorii furnizați şi o completează în declarația masivului.

8.9. Masive de pointeri şi masive multidimensionale

Adesea se creează confuzii în ceea ce priveşte diferența dintre un masiv bidimensional şi un masiv de pointeri. Fie date declarațiile:

```
int a[10][10],*b[10];
```

În acest caz **a** este un masiv de întregi căruia i se alocă spațiu pentru toate cele 100 de elemente, iar calculul indicilor se face în mod obişnuit pentru a avea acces la oricare element al masivului.

Pentru masivul **b** declarația alocă spațiu numai pentru zece pointeri, fiecare trebuind să fie încărcat cu adresa unui masiv de întregi.

Presupunînd că fiecare pointer indică un masiv de zece elemente înseamnă că ar trebui alocate încă o sută de locații de memorie pentru elementele masivelor.

În această accepțiune, folosirea masivelor **a** şi **b** poate fi similară în sensul că **a[5][5]** şi **b[5][5]**, de exemplu, se referă ambele la unul şi acelaşi întreg (dacă fiecare element **b[i]** este inițializat cu adresa masivului **a[i]**).

Astfel, masivul de pointeri utilizează mai mult spațiu de memorie decît masivele bidimensionale şi pot cere un pas de inițializare explicit. Dar masivele de pointeri prezintă două avantaje, şi anume: accesul la un element se face cu adresare indirectă, prin intermediul unui pointer, în loc de procedura obişnuită folosind înmulțirea şi apoi adunarea, iar al doilea avantaj constă în aceea că dimensiunea masivelor de pointeri poate fi variabilă. Acest lucru înseamnă că un element al masivului de pointeri **b** poate indica un masiv de zece

elemente, altul un masiv de două elemente şi altul poate să nu indice nici un masiv.

Cu toate că problema prezentată în acest paragraf a fost descrisă în termenii întregilor, ea este cel mai frecvent utilizată în memorarea şirurilor de caractere de lungimi diferite (ca în funcţia **NumeLuna** prezentată mai sus).

Prezentăm în continuare un model de rezolvare a alocării de memorie pentru o matrice de dimensiuni care sînt cunoscute abia în momentul execuţiei. În biblioteca limbajului C există funcţii care ne oferă posibilitatea de a aloca memorie necesară pentru memorarea unor date. După ce datele au fost prelucrate, memoria ocupată nu mai este necesară şi poate fi eliberată.

Declaraţiile acestor funcţii se obţin cu:

```
#include <stdlib.h>
```

Pentru alocarea unei zone de memorie apelăm funcţia **malloc**:

```
void *malloc(unsigned n);
```

Funcţia alocă o zonă de memorie de **n** octeţi şi returnează adresa acesteia.

O zonă de memorie avînd adresa **p** (**p** fiind un pointer) poate fi eliberată după ce nu mai este necesară:

```
void free(void *p);
```

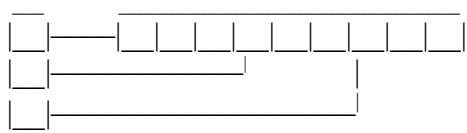
Presupunem că trebuie să alocăm memorie pentru o matrice de elemente întregi avînd **n** linii şi **m** coloane.

```
#include <stdlib.h>
int **IntMatr(int n, int m) {
int i,**p;
p = (int **)malloc(n*sizeof(int *));
p[0] = (int *)malloc(nm*sizeof(int));
for (i=1; i<n; i++)
   p[i] = p[i-1] + m;
return p;
}
```

Iniţial se alocă memorie unde se vor depune adresele celor **n** linii, de aceea primul apel **malloc** cere o zonă de memorie pentru **n** pointeri. În continuare trebuie alocată memorie şi pentru matricea

propriu-zisă. Pentru simplificare am preferat să alocăm o zonă continuă de **n×m** întregi (al doilea apel **malloc**).

În acest moment putem determina adresa fiecărei linii a matricei. Prima linie are determinată adresa din momentul alocării. Pentru o linie oarecare **i** (**i >** 0) adresa acesteia se obține din adresa liniei precedente la care se adaugă **m** (numărul de elemente ale liniei); a se vedea şi figura de mai jos.



Este foarte important să se respecte aceşti paşi în alocarea memoriei pentru matrice. În caz contrar variabilele pointer conțin valori necunoscute şi nu există garanții că indică spre zone de memorie sigure: este foarte probabil că programul nu are acces la zonele respective.

Funcția principală va fi scrisă astfel:

```
int main() {
int **Mat,n,m;
/* alte definiții de variabile */
/* Citeşte valorile n şi m */
Mat = IntMatr(n,m);
/* Completează matricea cu valori Mat[i][j] ... */
/* Prelucrează matricea */
free(Mat[0]); free(Mat);          /* Eliberează memoria */
}
```

Este foarte important să se respecte şi paşii precizați pentru eliberarea memoriei ocupate: mai întîi zona ocupată de matrice şi apoi şi zona ocupată de pointeri. Dacă se eliberează mai întîi zona ocupată de pointeri, programul nu mai are acces la zona respectivă, şi un apel **free(Mat[0])** va fi respins de sistemul de operare.

Să reluăm problema alocării de memorie pentru o matrice în cazul general, unde cunoaştem:
– **n**: numărul liniilor matricei;

– **M[]**: masiv care indică numărul elementelor din fiecare linie;
– **s**: mărimea în octeți a unui element al matricei (se obține cu operatorul **sizeof**); toate elementele matricei sînt de acelaşi tip.

```
void **Genmatr(int n, int M[], int s) {
int i;
void **p;
p = (void **)malloc(n*sizeof(void *));
for (i=0; i<n; i++)
   p[i] = malloc(M[i]*s);
return p;
}
```

Spre deosebire de exemplul precedent, unde toată memoria a fost alocată într-o singură operație (al doilea apel **malloc**), aici am alocat separat memorie pentru fiecare linie.

Apelul acestei funcții necesită anumite pregătiri:

```
double **Dm;      /* pentru o matrice cu elemente de tip double */
int n,*M;
/* alte definiții de variabile */
/* Citeşte valoarea n */
M = (int *)malloc(n*sizeof(int));
/* precizează pentru fiecare linie numărul de elemente */
Dm = (double **)Genmatr(n,M,sizeof(double));
/* Operații cu elementele matricei */
```

Memoria ocupată se eliberează astfel:

```
for (i=0; i<n; i++)
   free(Dm[i]);
free(Dm); free(M);
```

Este posibil să alocăm o zonă compactă pentru memorarea matricei, trebuie mai întîi să determinăm numărul total de elemente. A doua variantă a funcției **GenMatr** este:

```
void **Genmatr(int n, int M[], int s) {
int i,l;
char **p;
p = (char **)malloc(n*sizeof(void *));
for (l=i=0; i<n; i++)
   l += M[i];
p[0] = (char *)malloc(l*s);
```

```
for (i=1; i<n; i++)
   p[i] = p[i-1] + M[i] * s;
return p;
}
```

Memoria ocupată de matrice se eliberează cu două apeluri **free**, aşa cum am arătat mai sus în cazul matricei de întregi.

Considerăm că acest stil de programare, bazat pe alocarea compactă, este mai avantajos deoarece se evită o fragmentare excesivă a memoriei. Sistemul de operare memorează informaţii suplimentare (ascunse programatorului) care îi permit ulterior să elibereze o zonă de memorie la cerere.

8.10. Argumentele unei linii de comandă

În sistemul de calcul care admite limbajul C trebuie să existe posibilitatea ca în momentul execuţiei unui program scris în acest limbaj să i se transmită acestuia argumente sau parametri prin linia de comandă. Cînd un program este lansat în execuţie şi funcţia **main** este apelată, apelul va conţine două argumente.

În literatura de specialitate declaraţia funcţiei **main** este:

```
int main(int argc, char *argv[]);
```

Pentru a obţine descrieri compacte ale programelor am preferat să folosim numele **ac** şi **av** pentru argumentele funcţiei **main**. Primul argument reprezintă numărul de argumente din linia de comandă care a lansat programul. Al doilea argument este un pointer la un masiv de pointeri la şiruri de caractere care conţin argumentele, cîte unul pe şir.

Să ilustrăm acest mod dinamic de comunicare între utilizator şi programul său printr-un exemplu.

Fie programul numit **pri** care dorim să afişeze la terminal argumentele lui luate din linia de comandă, afişarea făcîndu-se pe o linie, iar argumentele afişate să fie separate prin spaţii.

Comanda:

```
pri succes colegi
```

va avea ca rezultat afişarea la terminal a textului

succes colegi

Prin convenţie, **av[0]** este un pointer la numele **pri** al programului apelat, astfel că **ac**, care specifică numărul de argumente din linia de comandă, este cel puţin 1.

În exemplul nostru, **ac** este 3, iar **av[0]**, **av[1]** şi **av[2]** sînt pointeri la **pri**, **succes** şi respectiv **colegi**. Primul argument efectiv este **av[1]** iar ultimul este **av[ac-1]**. Dacă **ac** este 1, înseamnă că linia de comandă nu are nici un argument după numele programului.

Atunci programul **pri** are următorul cod:

```
int main(int ac, char *av[]) {    /* afişează argumentele */
int i;
for (i=1; i<ac; i++)
    printf("%s%%c",av[i],(i<ac-1)?' ':'\n');
return 0;
}
```

Deoarece **av** este un pointer la un masiv de pointeri, există mai multe posibilităţi de a scrie acest program. Să mai scriem o versiune a acestui program.

```
int main(int ac, char *av[]) {    /* versiunea a doua */
while (--ac>0)
   printf("%s%c",*++av,(av>1) ? ' ' : '\n');
return 0;
}
```

Deoarece **av** este un pointer la un masiv de pointeri, prin incrementare ***++av** va indica spre **av[1]** în loc de **av[0]**. Fiecare incrementare succesivă poziţionează pe **av** la următorul argument, iar ***av** este pointerul la argumentul şirului respectiv. În acelaşi timp **ac** este decrementat pînă devine zero, moment în care nu mai sînt argumente de afişat.

Această versiune arată că argumentul funcţiei **printf** poate fi o expresie ca oricare alta.

Ca un al doilea exemplu, să reconsiderăm un program anterior care afişează fiecare linie a unui text care conţine un şir specificat de caractere (şablon).

Dorim acum ca acest şablon să poată fi precizat printr-un argument în linia de comandă la momentul execuţiei.

Şi atunci funcţia principală a programului care caută şablonul dat de primul argument al liniei de comandă este:

```
int main(int ac, char *av[ ]) {
char lin[80];
if (ac!=2)
   puts("Linia de comanda eronata");
else
   while (gets(lin))
      if (index(lin,av[1])>=0)
         puts(lin);
return 0;
}
```

unde linia de comandă este de exemplu: **find limbaj** în care **find** este numele programului, iar **limbaj** este şablonul căutat. Rezultatul va fi afişarea tuturor liniilor textului de intrare care conţin cuvîntul **limbaj**.

Să elaborăm acum modelul de bază, legat de linia de comandă şi argumentele ei.

Să presupunem că dorim să introducem în linia de comandă două argumente opţionale: unul care să afişeze toate liniile cu excepţia acelora care conţin şablonul, şi al doilea care să preceadă fiecare linie afişată cu numărul ei de linie.

O convenţie pentru programele scrise în limbajul C este ca argumentele dintr-o linie de comandă care încep cu un caracter **-** să introducă un parametru opţional. Dacă alegem, de exemplu, **-x** pentru a indica *cu excepţia* şi **-n** pentru a cere *numărarea liniilor*, atunci comanda:

```
find -x -n la
```

avînd intrarea:

la miezul stinselor lumini
s-ajung victorios,
la temelii, la rădăcini,
la măduvă, la os.

va produce afişarea liniei a doua, precedată de numărul ei, deoarece această linie nu conţine şablonul **la**.

Argumentele opţionale sînt permise în orice ordine în linia de comandă. Analizarea şi prelucrarea argumentelor unei linii de comandă trebuie efectuată în funcţia principală **main**, iniţializînd în mod corespunzător anumite variabile. Celelalte funcţii ale programului nu vor mai ţine evidenţa acestor argumente.

Este mai comod pentru utilizator dacă argumentele opţionale sînt concatenate, ca în comanda:

```
find -xn la
```

Caracterele **x** respectiv **n** indică doar absenţa sau prezenţa acestor opţiuni (*switch*) şi nu sînt tratate din punct de vedere al valorii lor.

Fie programul care caută şablonul **la** în liniile de la intrare şi le afişează pe acelea, care nu conţin şablonul, precedate de numărul lor de linie. Programul tratează corect atît prima formă a liniei de comandă cît şi a doua. În continuare este prezentată doar funcţia principală a acestui program.

```
int main(int ac, char *av[]) {        /* caută un şablon */
char lin[80],*s;
long nl;
int exc,num,tst;
nl = num = 0;
while (--ac>0 && (*++av)[0]=='-')
   for (s=av[0]+1; *s!=0; s++)
      switch(*s) {
         case 'x': exc = 1; break;
         case 'n': num = 1; break;
         default:
           printf("find: optiune ilegala %c\n",*s);
           ac = 0;
           break;
            }
if (ac!=1)
   puts("Nu exista argumente sau sablon");
else
   while (gets(lin) {
      nl++;
```

```
        tst = index(lin,*av)>=0;
        if (tst!=except) {
           if (num)
              printf("%4d:",nl);
           puts(lin);
           }
        }
}
```

Dacă nu există erori în linia de comandă, atunci la sfîrşitul primului ciclu while **ac** trebuie să fie 1, iar ***av** conține adresa şablonului. ***++av** este un pointer la un şir argument, iar **(*++av)[0]** este primul caracter al şirului. În această ultimă expresie parantezele sînt necesare deoarece fără ele expresia înseamnă ***++(av[0])** ceea ce este cu totul altceva (şi greşit): al doilea caracter din numele programului. O alternativă corectă pentru **(*++av)[0]** este ****++av**.

8.11. Pointeri la funcții

În limbajul C o funcție nu este o variabilă, dar putem defini un pointer la o funcție, care apoi poate fi prelucrat, transmis unor alte funcții, introdus într-un masiv şi aşa mai departe. Relativ la o funcție se pot face doar două operații: apelul ei şi considerarea adresei ei. Dacă numele unei funcții apare într-o expresie, fără a fi urmat imediat de o paranteză stîngă, deci nu pe poziția unui apel la ea, atunci se generează un pointer la această funcție. Pentru a transmite o funcție unei alte funcții, ca argument, se poate proceda în felul următor:

```
int f();
g(f);
```

unde funcția **f** este un argument pentru funcția **g**. Definiția funcției **g** va fi:

```
g(int (*funcpt)()) {
funcpt();
}
```

În expresia **g(f)**, **f** nu este un apel de funcție. În acest caz, pentru argumentul funcției **g** se generează un pointer la funcția **f**. Deci **g** apelează funcția **f** printr-un pointer la ea.

Declarațiile din funcția **g** trebuie studiate cu grijă.

```
int (*funcpt)();
```

spune că **funcpt** este un pointer la o funcție care returnează un întreg. Primul set de paranteze este necesar, deoarece fără el

```
int *funcpt();
```

ar însemna că **funcpt** este o funcție care returnează un pointer la un întreg, ceea ce este cu totul diferit față de sensul primei expresii. Folosirea lui **funcpt** în expresia:

```
funcpt();
```

indică apelul funcției a cărei adresă este dată de pointerul **funcpt**.

Ca un exemplu, să considerăm procedura de sortare a liniilor de la intrare, modificată în sensul ca dacă argumentul opțional **-n** apare în linia de comandă, atunci liniile se vor sorta nu lexicografic ci numeric, liniile conținînd grupe de numere.

O sortare constă de regulă din trei componente: o comparare care determină ordinea oricărei perechi de elemente, un schimb pentru a inversa ordinea elementelor implicate, şi un algoritm de sortare care face comparările şi inversările pînă cînd elementele sînt aduse în ordinea cerută. Algoritmul de sortare este independent de operațiile de comparare astfel încît, transmițînd diferite funcții de comparare funcției de sortare, elementele de intrare se pot aranja după diferite criterii.

Compararea lexicografică a două linii se realizează prin funcția **strcmp**. Mai avem nevoie de o rutină **numcmp** care să compare două linii pe baza valorilor numerice şi care să returneze aceiaşi indicatori ca şi rutina **strcmp**.

Declarăm aceste trei funcții în funcția principală **main**, iar pointerii la aceste funcții îi transmitem ca argumente funcției **sort**, care la rîndul ei va apela aceste funcții prin intermediul pointerilor respectivi.

Funcția principală **main** va avea atunci următorul cod:

```
#define NLIN 100        /* număr maxim de linii de sortat */
```

```
int main (int ac, char *av[]) {
char *linptr[NLIN];   /* pointeri la linii text */
int nrl;              /* număr de linii citite */
int num = 0;          /* 1 dacă sort numeric */
if (ac>1 && av[1][0]=='-' && av[1][1]=='n')
   num = 1;
nrl = ReadLines(linptr,NLIN);
if (nrl>=0) {
   if (num) Sort(linptr,nrl,numcmp);
      else Sort(linptr,nrl,strcmp);
   WriteLines(linptr,nrl);
   }
else puts("Intrare prea mare pentru Sort");
}
```

În apelul funcției **Sort**, argumentele **strcmp** şi **numcmp** sînt adresele funcțiilor respective. Deoarece ele sînt declarate funcții care returnează un întreg (a se vedea mai jos structura programului), operatorul & nu trebuie să preceadă numele funcțiilor, compilatorul fiind cel care gestionează transmiterea adreselor funcțiilor.

Să sintetizăm într-o declarație elementele comune ale celor două funcții:

```
typedef int (*P_func)(const char *, const char *);
```

Această declarație definește identificatorul **P_func** ca fiind un tip pointer la o funcție care returnează un întreg, şi are ca argumente doi pointeri la caracter. Elementele celor două masive nu pot fi modificate. Tipul **P_func** poate fi folosit în continuare pentru a construi noi declarații.

Funcția **Sort** care aranjează liniile în ordinea crescătoare se va modifica astfel:

```
void Sort(char *v[], int n, P_func comp) {
int gap,i,j;
char *tmp;
for (gap=n/2; gap>0; gap/=2)
   for (i=gap; i<n; i++)
      for (j=i-gap; j>=0; j-=gap) {
         if (comp(v[j],v[j+gap])<=0)
            break;
```

```
            tmp = v[j];
            v[j] = v[j+gap];
            v[j+gap] = tmp;
            }
}
```

Funcția **numcmp** este definită astfel:

```
int numcmp(const char *s1, const char *s2) {
double atof(char *s),v1,v2;
v1 = atof(s1);
v2 = atof(s2);
if (v1<v2) return -1;
if (v1>v2) return 1;
return 0;
}
```

Structura programului este următoarea:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int ReadLines(...) {...}
void WriteLines(...) {...}
int numcmp(...) {...}
typedef int (*P_func)(const char *, const char *);
void Sort(...) {...}
int main(...) {...}
```

În funcția **main**, în locul variabilei **num** de tip întreg putem folosi un pointer de tip **P_func** astfel:

```
P_func Fcomp = strcmp;
if (ac>1 && av[1][0]=='-' && av[1][1]=='n')
   Fcomp = numcmp;
nrl = ReadLines(linptr,NLIN);
if (nrl>=0) {
   Sort(linptr,nrl,Fcomp);
   WriteLines(linptr,nrl);
   }
```

Dacă programul este compus din mai multe module, se utilizează un fişier header care conţine declaraţiile tuturor funcţiilor utilizate. Acest fişier trebuie inclus în fiecare modul.

9. Structuri şi reuniuni

O structură este o colecţie de una sau mai multe variabile, de acelaşi tip sau de tipuri diferite, grupate împreună sub un singur nume pentru a putea fi tratate împreună.

Structurile ajută la organizarea datelor complexe, în special în programele mari, deoarece permit unui grup de variabile legate să fie tratate ca o singură entitate. În acest capitol vom ilustra modul de utilizare a structurilor.

9.1. Elemente de bază

Sintaxa unei declaraţii de structură este:

struct *nume-şablon-structură* **{**
listă-declaraţii-membri
} *listă-variabile***;**

Numele şablonului structurii poate lipsi, în acest caz şablonul nu va avea un nume, şi lista variabilelor este obligatorie.

Lista variabilelor poate lipsi, în acest caz şablonul de tip structură trebuie să aibă un nume. Astfel se defineşte un şablon pentru variabile care vor fi definite ulterior, avînd acest tip structurat:

struct *nume-şablon-structură listă-variabile***;**

Fiecare variabilă din listă va fi compusă din aceleaşi elemente (membri), aşa cum a fost definit şablonul.

Să revenim asupra rutinei de conversie a datei care a fost prezentată în capitolul anterior. O dată calendaristică se compune din zi, lună şi an. Aceste trei variabile pot fi grupate într-o singură structură astfel:

```
struct S_data {
   int zi,luna,an;
   } ;
```

Cuvîntul cheie **struct** introduce o declaraţie de structură care este o listă de declaraţii închise între acolade. Acest cuvînt poate fi

urmat opțional de un nume, numit marcaj de structură sau etichetă de structură, cum este în exemplul nostru numele **S_data**. Marcajul denumeşte acest tip de structură şi poate fi folosit în continuare ca o prescurtare pentru declarația de structură detaliată căreia îi este asociat.

Elementele sau variabilele menționate într-o structură se numesc membri ai structurii. Un membru al structurii sau o etichetă şi o variabilă oarecare, nemembru, pot avea acelaşi nume fără a genera conflicte, deoarece ele vor fi întotdeauna deosebite una de alta din context.

Declarația următoare defineşte variabila **d** ca o structură având acelaşi şablon ca structura **S_data**:

```
struct S_data d;
```

O structură externă sau statică poate fi inițializată, ataşînd după definiția ei o listă de inițializatori pentru componente, de exemplu:

```
struct S_data d = {4,7,1984};
```

Un membru al unei structuri este referit printr-o expresie de forma:

nume-structură`.`*membru*

în care operatorul membru de structură **.** leagă numele membrului de numele structurii. Ca exemplu fie atribuirea:

```
bis = !(d.an%4) && (d.an%100) || !(d.an%400);
```

Structurile pot fi imbricate: o înregistrare de stat de plată, de exemplu, poate fi de următoarea formă:

```
struct S_pers {
   long nrl;
   char num[LNUM],adr[LADR];
   long sal;
   struct S_data dnast,dang;
   } ;
```

Tipul structură **S_pers** conține două structuri de şablon **S_data**. Declarația următoare defineşte o variabilă de tip structură cu numele **ang**, având acelaşi şablon ca structura **S_pers**:

```
struct S_pers ang;
```

În acest caz:

ang.dnast.luna se referă la luna de naştere

ang.dang.an se referă la anul angajării

Singurele operaţii pe care le vom aplica unei structuri vor fi accesul la un membru al structurii şi considerarea adresei ei cu ajutorul operatorului **&**. Structurile de clasă automatic, ca şi masivele de aceeaşi clasă, nu pot fi iniţializate; pot fi iniţializate numai structurile externe şi statice, regulile de iniţializare fiind aceleaşi ca pentru masive.

Structurile şi funcţiile conlucrează foarte eficient prin intermediul pointerilor. Ca un exemplu, să rescriem funcţia de conversie a datei, care determină numărul zilei din an.

```
int NrZile(const struct S_data *pd) {
int i,zi,bis;
zi = pd->zi;
bis = !(pd->an%4) && (pd->an%100) || !(pd->an%400);
for (i=1; i<pd->luna; i++)
   zi += Zile[bis][i];
return zi;
}
```

Declaraţia din lista de argumente indică faptul că **pd** este un pointer la o structură de şablonul **S_data**. Notaţia **pd->an** indică faptul că se referă la membrul **an** al acestei structuri. În general, dacă **p** este un pointer la o structură, **p->***membru-structură* se referă la un membru particular (operatorul **->** se formează din semnul minus urmat de semnul mai mare).

Deoarece **pd** este pointer la o structură, membrul `an` poate fi de asemenea referit prin: **(*pd).an**, dar notaţia **->** se impune ca un mod convenabil de prescurtare. În notaţia **(*pd).an**, parantezele sînt necesare deoarece precedenţa operatorului **.** (membru de structură) este mai mare decît cea a operatorului ***** (indirectare).

Operatorii **->** şi **.** ai structurilor, împreună cu **()** (modificare priorităţi, liste de argumente) şi **[]** (indexare) se găsesc în vîrful listei de precedenţă, fiind din acest punct de vedere foarte apropiaţi. Astfel, fie declaraţia următoare care defineşte un pointer `p` la o structură:

```
struct {
   int x,*y;
   } *p;
```

Semnificaţiile următoarelor expresii sînt:

++p->x

Incrementează membrul **x**, nu pointerul **p**, deoarece operatorul **->** are o precedenţă mai mare decît **++**. Parantezele pot fi folosite pentru a modifica ordinea operatorilor dată de precedenţa.

(++p)->x

Incrementează mai întîi pointerul **p**, apoi accesează membrul **x**, din structura indicată de noua valoare a lui **p**.

(p++)->x

Accesează mai întîi membrul **x**, apoi incrementează pointerul **p**.

***p->y**

Indică conţinutul de la adresa memorată la **y**.

***p->y++**

Accesează mai întîi ceea ce indică **y**, apoi incrementează pe **y**.

(*p->y)++

Incrementează ceea ce indică **y**.

***p++->y**

Accesează mai întîi ceea ce indică **y**, apoi incrementează pointerul **p**.

9.2. typedef

Stilul de utilizare a tipurilor de date structurate prezentat mai sus este foarte frecvent, şi se întîlneşte mai ales în programe care apelează funcţii ale sistemelor de operare Linux.

Considerăm totuşi că a scrie declaraţii de funcţii care au ca şi parametri date structurate, sau a defini variabile de tip structură, este destul de neplăcut datorită cerinţei de a preciza numele tipului

precedat de cuvîntul rezervat **struct**. De aceea, pe parcursul lucrării vom folosi o facilitate (**typedef**) de a defini sinonime pentru tipuri de date existente, care pot fi fundamentale (predefinite) sau definite de programator.

Într-o declarație **typedef** fiecare identificator care apare ca parte a unui declarator devine sintactic echivalent cu cuvîntul cheie rezervat pentru tipul asociat cu identificatorul. De exemplu, declarația următoare îl face pe **LENGTH** sinonim cu **int**:

```
typedef int LENGTH;
```

„Tipul" **LENGTH** poate fi folosit ulterior în declarații în acelaşi mod ca şi tipul **int**:

```
LENGTH len,maxlen,*lgm[];
```

În mod similar, declarația următoare îl face pe **STRING** sinonim cu **char***, adică pointer la caracter:

```
typedef char *STRING;
```

„Tipul" **STRING** poate fi folosit în declarații ulterioare:

```
STRING p,linptr[NLIN];
```

Observăm că tipul care se declară prin **typedef** apare pe poziția numelui de variabilă, nu imediat după cuvîntul rezervat **typedef**.

Reluăm definiția tipului **P_func** din capitolul precedent:

```
typedef int (*P_func)(const char *, const char *);
```

Această declarație defineşte identificatorul **P_func** ca fiind un tip pointer la o funcție care returnează un întreg, şi are ca argumente doi pointeri la caracter. Elementele celor două masive nu pot fi modificate. Tipul **P_func** poate fi folosit în continuare pentru a construi noi declarații.

Rescriem definițiile de tipuri din secțiunea precedentă:

```
typedef struct {
   int zi,luna,an;
   } T_data;
```

O structură externă sau statică poate fi inițializată, ataşînd după definiția ei o listă de inițializatori pentru componente, de exemplu:

```
T_data d = {4,7,1984};
```

Structurile pot fi imbricate: un tip de înregistrare de stat de plată, de exemplu, poate fi definit astfel:

```
typedef struct {
   long nrl;
   char num[LNUM],adr[LADR];
   long sal;
   T_data dnast,dang;
   } T_pers;
```

Funcția **NrZile** poate fi declarată astfel:

```
int NrZile(const T_data *pd);
```

Există două motive principale care impun folosirea declarațiilor **typedef**. Primul este legat de problemele de portabilitate. Cînd se folosesc declarații **typedef** pentru tipuri de date care sînt dependente de maşină, atunci pentru o compilare pe un alt sistem de calcul este necesară modificarea doar a acestor declarații nu şi a datelor din program. La sfîrşitul acestui capitol vom da un exemplu în acest sens reluînd definirea tipului structurat dată calendaristică.

Al doilea constă în faptul că prin crearea de noi nume de tipuri se oferă posibilitatea folosirii unor nume mai sugestive în program, deci o mai rapidă înțelegere a programului.

În continuare pe parcursul lucrării vom folosi facilitatea **typedef** pentru a defini tipuri complexe de date şi în special structuri.

9.3. Masive de structuri

Structurile sînt în mod special utile pentru tratarea masivelor de variabile legate prin context. Pentru exemplificare vom considera un program care numără intrările fiecărui cuvînt cheie dintr-un text. Pentru aceasta avem nevoie de un masiv de şiruri de caractere pentru păstrarea cuvintelor cheie şi un masiv de întregi pentru a contoriza aparițiile acestora. O posibilitate este de a folosi două masive paralele **keyw** şi **keyc** declarate prin:

```
char *keyw[NKEYS];
int keyc[NKEYS];
```

respectiv unul de pointeri la şiruri de caractere şi celălalt de întregi.

Fiecărui cuvînt cheie îi corespunde perechea (**keyw**,**keyc**) astfel încît putem considera cele două masive ca fiind un masiv de perechi. Următoarea declarație definește un tip structură **T_key**:

```
typedef struct {
   char *keyw;
   int keyc;
   } T_key;
```

Folosim acest tip structură pentru a defini un masiv **keym**, unde **NKEYS** este o constantă definită cu ajutorul directivei **#define**:

```
T_key keym[NKEYS];
```

Fiecare element al masivului este o structură de acelaşi şablon ca şi structura **T_key**.

Deoarece masivul de structuri **keym** conține, în cazul nostru, o mulțime constantă de cuvinte cheie, este mai uşor de inițializat o dată pentru totdeauna chiar în locul unde este definit. Inițializarea structurilor este o operație analogă cu inițializarea unui masiv în sensul că definiția este urmată de o listă de inițializatori închişi în acolade.

Atunci masivul de structuri **keym** va fi inițializat astfel:

```
T_key keym[] = {
      "break",0, "case",0, "char",0,
      /* … */  "while",0
      };
```

Inițializatorii sînt perechi care corespund la membrii structurii. Inițializarea ar putea fi făcută şi incluzînd inițializatorii fiecărei structuri din masiv în acolade ca în:

```
T_key keym[] = { {"break",0},{"case",0},...}
```

dar parantezele interioare nu mai sînt necesare dacă inițializatorii sînt variabile simple sau şiruri de caractere şi dacă toți inițializatorii sînt prezenți.

Dacă masivul **keym** este global, eventualii membri pentru care nu se precizează o valoare inițială vor primi valoarea zero, dar în acest caz parantezele interioare sînt necesare:

```
T_key keym[] = { {"break"},{"case"},...}
```

Compilatorul va determina, pe baza inițializatorilor, dimensiunea masivului de structuri **keym**, motiv pentru care nu este necesară indicarea dimensiunii masivului la inițializare.

Programul de numărare a cuvintelor cheie începe cu definirea masivului de structuri **keym**. Funcția principală **main** citeşte cîte un cuvînt (şir de caractere) din textul de la intrarea standard prin apelul repetat al funcției **scanf**. Fiecare şir este apoi căutat în tabelul **keym** cu ajutorul unei funcții de căutare **binary**. Lista cuvintelor cheie trebuie să fie în ordine crescătoare pentru ca funcția **binary** să lucreze corect. Dacă şirul cercetat este un cuvînt cheie atunci funcția **binary** returnează numărul de ordine al acestuia în tabelul cuvintelor cheie, altfel returnează –1.

```
int binary(char *word, const T_key kt[], int n) {
int l,r,m,c;
l = 0;  r = n - 1;
while (l<=r) {
   m =(l + r) / 2;
   c = strcmp(word,kt[mid].keyw);
   if (cond==0) return m;
   if (cond<0) r = m - 1;
      else l = m + 1;
   }
return -1;
}

int main() {                          /* numără cuvintele cheie */
int n;
char word[36];
while (scanf("%s",word)!=EOF) {
   n = binary(word,keym,NKEYS);
   if (n>=0) keym[n].keyc++;
for (n=0; n<NKEYS; n++)
   if (keym[n].keyc>0)
      printf("%4d %s\n",keym[n].keyc,
                        keym[n].keyw);
return 0;
}
```

Important! Funcția **scanf** cu descriptorul de format **"%s"** furnizează în masivul **word** un şir de caractere *care nu conține spații*. Dacă şirul are cel puțin 36 de caractere, comportamentul programului este imprevizibil. Dacă dorim să preluăm cel mult 35 de caractere folosim formatul **"%35s"**. Citirea şi scrierea cu format vor fi detaliate în capitolul 10.

Valoarea **NKEYS** este numărul cuvintelor cheie din **keym** (dimensiunea masivului de structuri). Deşi putem calcula acest număr manual, este mai simplu şi mai sigur s-o facem prin program, mai ales dacă lista cuvintelor cheie este supusă modificărilor. O posibilitate de a calcula **NKEYS** prin program este de a termina lista inițializatorilor cu un pointer **NULL** şi apoi prin ciclare pe **keym** să detectăm sfîrşitul lui. Acest lucru nu este necesar deoarece dimensiunea masivului de structuri este perfect determinată în momentul compilării. Numărul de intrări se determină împărțind dimensiunea masivului la dimensiunea structurii **key**.

Operatorul **sizeof** furnizează dimensiunea în octeți a argumentului său.

În cazul nostru, numărul cuvintelor cheie este dimensiunea masivului **keym** împărțită la dimensiunea unui element de masiv. Acest calcul este făcut într-o linie **#define** pentru a da o valoare identificatorului **NKEYS**:

```
#define NKEYS (sizeof(keym)/sizeof(T_key))
```

Structura programului este următoarea:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
typedef struct {...} T_key;
T_key keym[] = {...} ;
int binary(...) {...}
int main(...) {...}
```

9.4. Pointeri la structuri

Pentru a ilustra modul de corelare dintre pointeri şi masivele de structuri, să rescriem programul de numărare a cuvintelor cheie dintr-un text, de data aceasta folosind pointeri în loc de indici de masiv.

Declarația externă a masivului de structuri **keym** nu necesită modificări, în timp ce funcțiile **main** şi **binary** da. Prezentăm în continuare aceste funcții modificate.

```
T_key *binary(char *word, T_key kt[],int n) {
int c;
struct key *m;
struct key *l = kt;
struct key *r = kt + n - 1;
l = kt;  r = kt + n - 1;
while (l<=r) {
   m = l + (r - l) / 2;
   c = strcmp(word,m->keyw);
   if (c==0) return m;
   if (c>0) l = m + 1;
      else r = m - 1;
   }
return NULL;
}

int main() {     /* numără cuvintele cheie, versiune cu pointeri */
int t;
char word[36];
T_key *p;
while (scanf("%s",word)!=EOF)
   if (p=binary(word,keym,NKEYS))
      p->keyc++;
for (p=keym; p<keym+NKEYS; p++)
   if (p->keyc>0)
      printf("%4d %s\n",p->keyc,p->keyw);
}
```

Să observăm cîteva lucruri importante în acest exemplu. În primul rînd, declarația funcției **binary** trebuie să indice că ea returnează un pointer la o structură de acelaşi şablon cu structura

T_key, în loc de un întreg. Dacă **binary** găseşte un cuvînt în structura **T_key**, atunci returnează un pointer la el; dacă nu-l găseşte returnează **NULL**. În funcţie de aceste două valori returnate, funcţia **main** semnalează găsirea cuvîntului prin incrementarea cîmpului **keyc** corespunzător cuvîntului sau trece direct la citirea următorului cuvînt.

În al doilea rînd, toate operaţiile de acces la elementele masivului de structuri **keym** se fac prin intermediul pointerilor. Acest lucru determină o modificare semnificativă în funcţia **binary**. Calculul elementului mijlociu nu se mai poate face simplu prin:

```
m = (l + r) / 2
```

deoarece adunarea a doi pointeri este o operaţie ilegală, nedefinită. Această instrucţiune trebuie modificată în:

```
m = l + (r - l) / 2
```

care face ca adresa memorată la `m` să indice elementul de la jumătatea distanţei dintre **l** şi **r**.

Să mai observăm iniţializarea pointerilor **l** şi **r**, care este perfect legală, deoarece este posibilă iniţializarea unui pointer cu o adresă a unui element deja definit.

În funcţia **main** avem următorul ciclu:

```
for(p=keym; p<keym+NKEYS; p++)...
```

Dacă **p** este un pointer la un masiv de structuri, orice operaţie asupra lui **p** ţine cont de dimensiunea unei structuri, astfel încît **p++** incrementează pointerul **p** la următoarea structură din masiv, adunînd la **p** dimensiunea corespunzătoare a structurii. Nu întotdeauna dimensiunea structurii este egală cu suma dimensiunilor membrilor ei, deoarece din cerinţe de aliniere a unor membri pot apărea „goluri" în interiorul acesteia.

De aceea, dacă dorim să economisim cît mai multă memorie în cazul masivelor de structuri, recomandăm gruparea membrilor astfel încît să eliminăm pe cît posibil apariţia unor astfel de goluri. În principiu, fiecare tip de date are anumite cerinţe de aliniere:

– **short** la multiplu de 2;

– **long**, **float**, **double**, **long double** la multiplu de 4.

Dimensiunea unei structuri este rotunjită la un multiplu întreg corespunzător acestor cerințe.

9.5. Structuri auto-referite

O altă problemă legată de definirea şi utilizarea structurilor este căutarea în tabele. Cînd se întîlneşte o linie de forma:

```
#define YES 1
```

simbolul **YES** şi textul de substituție **1** se memorează într-o tabelă. Mai tîrziu, ori de cîte ori textul **YES** va apărea în instrucțiuni, el va fi înlocuit cu constanta **1**.

Crearea şi gestionarea tabelelor de simboluri este o problemă de bază în procesul de compilare. Deoarece nu ştim dinainte cîte simboluri vor apărea în program, nu putem defini un masiv care să memoreze aceste informații: compilatorul trebuie să cunoască numărul de elemente în momentul compilării. De aceea vom aloca spațiu pentru fiecare simbol nou, şi aceste zone vor fi legate între ele.

Definim două funcții care gestionează simbolurile şi textele lor de substituție. Prima, **Install(s,t)** înregistrează simbolul **s** şi textul de substituție **t** într-o tabelă, **s** şi **t** fiind şiruri de caractere. A doua, **Lookup(s)** caută şirul **s** în tabelă şi returnează fie un pointer la locul unde a fost găsit, fie **NULL** dacă şirul **s** nu figurează în tabel.

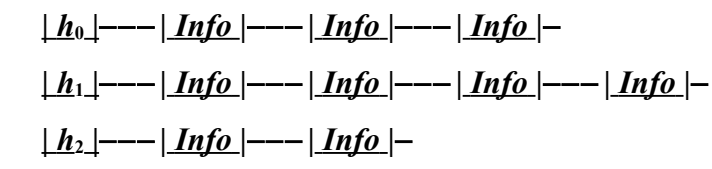
Algoritmul folosit pentru crearea şi gestionarea tabelei de simboluri este o căutare pe bază de hashing (dispersie). Fiecărui simbol i se calculează un cod hash astfel: se adună codurile ASCII ale caracterelor simbolului şi se ia restul provenit din împărțirea numărului obținut din adunare şi dimensiunea tabelului. Astfel, fiecărui simbol i se asociază un cod hash `H` care verifică relația:

0<=H<0x100 (în hexazecimal)

Codul hash astfel obținut va fi folosit apoi ca un indice într-o tabelă de pointeri. Un element al acestei tabele (masiv) indică începutul unui lanț de blocuri care descriu simboluri cu acelaşi cod hash. Dacă un element al tabelei este **NULL** înseamnă că nici un simbol nu are valoarea respectivă de hashing.

Un bloc dintr-un lanț indicat de un element al tabelei este o structură care conține un pointer la simbol, un pointer la textul de

substituție și un pointer la următorul bloc din lanț. Un pointer **NULL** la următorul bloc din lanț indică sfîrșitul lanțului.



Șablonul unei structuri (nod) este următorul:

```
typedef struct S_list {
   char *num,*def;
   struct S_list *nxt;   / * următoarea intrare în lanț */
   } T_list, *P_list;
```

Această declarație „recursivă" a unui nod este perfect legală, deoarece o structură nu poate conține ca și componentă o intrare a ei însăși, dar poate conține un pointer la o structură de același șablon cu ea.

Declarația de mai sus definește un tip structurat cu numele **T_list**, sinonim cu tipul **struct S_list**. Tipul **P_list** este sinonim cu **T_list *** și cu **struct S_list ***.

Masivul de pointeri care indică începuturile lanțului de blocuri ce descriu simboluri de același cod hash este:

```
#define NHASH 0x100
static T_list *hasht[NHASH];
```

Algoritmul de hashing pe care-l prezentăm nu este cel mai bun posibil, dar are meritul de a fi extrem de simplu:

```
int hashf(char *s) {  /* formează valoarea hash pentru șirul s */
int hv;
for (hv=0; *s;) hv += *s++;
return hv % NHASH;
}
```

Acesta produce un indice în masivul de pointeri **hasht**. În procesul de căutare a unui simbol, dacă acesta există trebuie să se afle în lanțul de blocuri care începe la adresa conținută de elementul din **hasht** cu indicele respectiv.

Căutarea în tabela de simboluri **hasht** se realizează cu funcția **Lookup**. Dacă simbolul căutat este prezent undeva în lanț, funcția returnează un pointer la el, altfel returnează **NULL**.

```
T_list *Lookup(char *s) {       /* caută şirul s în hasht */
T_list *np;
for (np=hasht[hashf(s)]; np; np=np->nxt)
   if (strcmp(s,np->num)==0)
      return np;          /* s-a găsit s */
return NULL;              /* nu s-a găsit s */
}
```

Dacă dorim să eliminăm un nod din listă, avem nevoie să ştim *unde este memorată adresa acestuia*. Dacă nodul nedorit este primul din listă, adresa acestuia este memorată într-un element din masivul **hast**, altfel această adresă este memorată în membrul **nxt** al nodului precedent din listă. Variabila pointer **pp** va reține unde se află memorată adresa care ne interesează. În parcurgerea listei, valoarea variabilei **pp** este tot timpul cu un pas în urma lui **np**, care indică nodul curent.

```
P_list *pp;
T_list *Lookup(char *s) {       /* caută şirul s în hasht */
T_list *np;
pp = hasht + hashf(s);
while (np=*pp) {
   if (strcmp(s,np->num)==0)
      return np;
   pp = &np->nxt;
   }
return pp = NULL;
}
```

Fiecare nod din listă are tipul **T_list** şi conține, printre altele, şi adresa următorului nod. Dar adresa primului nod e memorată într-un element care are *alt tip* decît **T_list**. De aceea avem nevoie să memorăm *adresa zonei* unde este memorată adresa care ne interesează, de unde necesitatea definiției globale

```
P_list *pp;
```

Această definiție este echivalentă cu

```
    T_list **pp;
```

Dacă şi elementele masivului ar avea tipul **T_list**, nu am avea nevoie de două nivele de indirectare, ar fi suficient să memorăm adresa nodului precedent într-o variabilă pointer la **T_list**.

Rutina **Install** foloseşte funcţia **Lookup** pentru a determina dacă simbolul nou care trebuie introdus în lanţ este deja prezent sau nu. Dacă mai există o definiţie anterioară pentru acest simbol, ea trebuie înlocuită cu definiţia nouă; altfel se creează o intrare nouă pentru acest simbol, care se introduce la începutul lanţului. Funcţia **Install** returnează **NULL** dacă din anumite motive nu există suficient spaţiu pentru crearea unui bloc nou.

```
T_list *Install(char *num, char *def) {
                                    /* scrie (num, def) în htab */
T_list *np;
int hv;
np = Lookup(num);
if (np==NULL) {     /* nu s-a găsit */
   np = (T_list*)calloc(1,sizeof(*np));
   if (np==NULL) return NULL;      /* nu există spaţiu */
   np->num = strdup(num);
   if (np==NULL) return NULL;
   hv = hashf(np->num);
   np->nxt = hasht[hv];
   hasht[hv] = np;
   }
else                    /* nodul există deja */
   free(np->def); /* eliberează definiţia veche */
np->def = strdup(def);
if (np->def==NULL) return NULL;
return np;
}
```

Rutina **Remove** elimină din listă ultimul element consultat.

```
void Remove() {
T_list *np;
if (!pp) return;
free(np=(*pp)->nxt);
*pp = np; pp = NULL;
```

```
}
```

Cum eliberăm toate zonele ocupate de o listă? Trebuie să fim foarte atenți deoarece, înainte de a elibera memoria ocupată de nodul curent, trebuie eliberată memoria ocupată de nodul care urmează celui curent. Prezentăm mai întîi varianta recursivă:

```
void FreeList(T_list *np) {
if (np) {
   FreeList(np->nxt);
   free(np);
   }
}
```

Definiția recursivă a funcției **FreeList** consumă destul de multă memorie din stivă, de aceea redăm în continuare şi varianta nerecursivă, mai eficientă.

```
void FreeList(T_list *np) {
T_list *nx;
while (np) {
   nx = np->nxt;
   free(np);
   np = nx;
   }
}
```

Funcția **main**, care utilizează aceste rutine, nu are nevoie să inițializeze elementele masivului **hasht** cu **NULL**, deoarece acesta este declarat global şi este automat inițializat corespunzător. De la intrarea standard se aşteaptă introducerea unor comenzi:

+ *nume valoare*

– *nume*

? *nume*

Comanda **+** inserează în listă simbolul *nume* cu valoarea dată. Dacă numele introdus nu există în listele **hasht** atunci se afişează un mesaj corespunzător, altfel se afişează vechea valoare care este apoi înlocuită de noua valoare.

Comanda **?** afişează valoarea simbolului *nume*, dacă acesta există în listele **hasht**.

Comanda **–** elimină simbolul *nume*, dacă acesta există în liste.

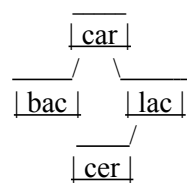
După epuizarea datelor introduse memoria ocupată de cele **NHASH** liste este eliberată.

```
int main() {
char num[36],def[36],op;
T_list *np;
int i;
while (scanf(" %c",&op)!=EOF)
   switch (op) {
      case '+':
         scanf("%s %s",num,def);
         np = Lookup(num);
         if (!np)
            puts("New name\n");
         else
            printf("Old value: %s\n",np->def);
         Install(num,def);
         break;
      case '-': case '?':
         scanf("%s",num);
         np = Lookup(num);
         if (!np)
            puts("Not in list");
         else
            switch (op) {
               case '-':
                  Remove(); puts("Removed");
                  break;
               case '?':
                  printf("Value: %s\n",np->def);
                  break;
               }
      }
for (i=0; i<NHASH; i++)
   FreeList(hasht[i]);
return 0;
}
```

9.6. Arbori binari de căutare

Să considerăm problema numărului aparițiilor tuturor cuvintelor dintr-un text de intrare. Deoarece lista cuvintelor nu este cunoscută dinainte, n-o putem sorta folosind algoritmul prezentat la sfîrşitul capitolului precedent, altfel contorizarea ar fi fost foarte simplă pentru o listă deja ordonată. Nu putem utiliza nici liste înlănțuite pentru a vedea dacă un cuvînt citit există sau nu în liste, pentru că timpul de execuție al programului ar creşte pătratic cu numărul cuvintelor de la intrare.

Un mod de a organiza datele pentru a lucra eficient cu o listă de cuvinte arbitrare este de a păstra mulțimea de cuvinte tot timpul sortată, plasînd fiecare nou cuvînt din intrare pe o poziție corespunzătoare, relativ la intrările anterioare. Dacă am realiza acest lucru prin deplasarea cuvintelor într-un masiv liniar, programul ar dura, de asemenea, foarte mult. De aceea, pentru rezolvarea eficientă a acestei probleme vom folosi o structură de date numită arbore binar:



Fiecare nod al arborelui va reprezenta un cuvînt distinct din intrare şi va conține următoarea informație:
– un pointer la un şir de caractere;
– un contor pentru numărul de apariții;
– un pointer la descendentul stîng al nodului;
– un pointer la descendentul drept al nodului; nici un nod al arborelui nu va avea mai mult decît doi descendenți dar poate avea un descendent sau chiar nici unul.

Arborele se construieşte astfel încît pentru orice nod, sub-arborele stîng al său conține numai cuvintele care sînt mai mici decît cuvîntul din nod, iar sub-arborele drept conține numai cuvintele care sînt mai mari decît cuvîntul din nod, compararea făcîndu-se din punct de vedere lexicografic.

Pentru a şti dacă un cuvînt nou din intrare există deja în arbore, se porneşte de la nodul rădăcină şi se compară noul cuvînt cu cuvîntul memorat în nodul rădăcină. Dacă ele coincid se incrementează contorul de numărare a aparițiilor pentru nodul rădăcină şi se va citi un nou cuvînt din intrare.

Dacă noul cuvînt din intrare este mai mic decît cuvîntul memorat în nodul rădăcină, căutarea continuă cu descendentul stîng, altfel se investighează descendentul drept. Dacă nu există nici un descendent pe direcția cerută, noul cuvînt nu există în arbore şi va fi inserat pe poziția descendentului corespunzător. Se observă că acest proces de căutare este recursiv, deoarece căutarea din fiecare nod utilizează o căutare într-unul dintre descendenții săi.

Prin urmare se impune de la sine ca rutinele de inserare în arbore şi de afişare să fie recursive.

Revenind la descrierea unui nod, el apare ca fiind o structură cu patru componente:

```
typedef struct S_node {      /* nodul de bază */
   char *w;                  /* pointer la cuvînt */
   int c;                    /* numărător de apariții */
   struct S_node *l;         /* descendent stîng (left) */
   struct S_node *r;         /* descendent drept (right) */
   } T_node,*P_node;
```

Declarația de mai sus defineşte un tip structurat cu numele **T_node**, sinonim cu tipul **struct S_node**. Tipul **P_node** este sinonim cu **T_node *** şi cu **struct S_node ***.

Funcția principală **main** citeşte cîte un cuvînt din textul de la intrarea standard, şi îl plasează în arbore prin rutina **TreeInsert**.

```
int main() {          /* contorizare apariții cuvinte */
char word[36];
P_node root;
root = NULL;
while (scanf("%s",word)!=EOF)
   TreeInsert(&root,word);
TreePrint(root);
TreeFree(root);
}
```

Funcția **TreeInsert** gestionează fiecare cuvînt din intrare începînd cu cel mai înalt nivel al arborelui (rădăcina). La fiecare pas, cuvîntul din intrare este comparat cu cuvîntul asociat rădăcinii şi este apoi transmis în jos, fie descendentului stîng, fie celui drept, printr-un apel recursiv la rutina **TreeInsert**. În acest proces, cuvîntul fie există deja, undeva în arbore, caz în care contorul lui de numărare a aparițiilor se incrementează, fie căutarea continuă pînă la întîlnirea unui pointer **NULL**, caz în care nodul trebuie creat şi adăugat arborelui. Cînd se creează un nod nou, rutina **TreeInsert** îl leagă nodul al cărui descendent este noul nod, în cîmpul **l** sau **r**, după cum noul cuvînt este mai mic sau mai mare față de cuvîntul origine. Tocmai din acest motiv rutina **TreeInsert** primeşte *adresa* variabilei pointer care va indica spre viitorul nod fiu.

```
void TreeInsert(P_node *p, char *w) {
int c;
if (*p) {
   c = strcmp(w,(*p)->w);
   if (c==0) (*p)->c++;
   else
   if (c<0)       /* noul cuvînt mai mic, mergem spre stînga */
      TreeInsert(&(*p)->l,w);
   else        /* noul cuvînt mai mare, mergem spre dreapta */
      TreeInsert(&(*p)->r,w);
   }
else {
   *p = (P_node)calloc(1,sizeof(T_node));
   (*p)->w = strdup(w);
   (*p)->c = 1;
}
```

Am considerat că este mai comod să solicităm alocare de memorie pentru un nod apelînd funcția **calloc**:

```
void calloc(unsigned ne, unsigned sz);
```

Argumentele acestei funcții sînt **ne**: numărul de elemente pe care le va avea zona, şi **sz**: mărimea în octeți a unui element, care se poate obține cu operatorul **sizeof**. Conținutul zonei de memorie

este pus la zero, spre deosebire de funcţia **malloc** care păstrează neschimbat acest conţinut (valori reziduale).

La adăugarea unui nod nou, funcţia **calloc** iniţializează pointerii spre cei doi descendenţi cu NULL. Rămîne să creăm o copie a şirului dat cu funcţia **strdup**, deja cunoscută, şi să iniţializăm contorul cu 1.

Rutina **TreePrint** afişează arborele astfel încît pentru fiecare nod se afişează sub-arborele lui stîng, adică toate cuvintele mai mici decît cuvîntul curent, apoi cuvîntul curent şi la sfîrşit sub-arborele drept, adică toate cuvintele mai mari decît cuvîntul curent. Funcţia **TreePrint** este una din cele mai tipice rutine recursive.

```
void TreePrint(P_node p) {
if (p) {
   Treeprint(p->l);
   printf("%5d %s\n",p->c,p->w);
   TreePrint(p->r);
   }
}
```

Este important de reţinut faptul că în algoritmul de căutare în arbore, pentru a ajunge la un anumit nod, se parcurg toate nodurile precedente pe ramura respectivă (stîngă sau dreaptă), începînd întotdeauna cu nodul rădăcină. După fiecare ieşire din rutina **TreeInsert**, datorită recursivităţii se parcurge acelaşi drum, de data aceasta de la nodul găsit spre rădăcina arborelui, refăcîndu-se toţi pointerii drumului parcurs.

O observaţie legată de acest exemplu: dacă arborele este „nebalansat“, adică cuvintele nu sosesc în ordine aleatoare din punct de vedere lexicografic, atunci timpul de execuţie al programului poate deveni foarte mare. Cazul limită în acest sens este acela în care cuvintele de la intrare sînt deja în ordine (crescătoare sau descrescătoare), caz în care programul nostru simulează o căutare liniară într-un mod destul de costisitor.

Rutina **TreeFree** eliberează memoria ocupată de nodurile arborelui. Ordinea în care se face eliberarea este următoarea: mai întîi sub-arborele stîng, apoi sub-arborele drept, şi în final nodul curent. Şi

aici eliberarea memoriei ocupate de nod trebuie să respecte o ordine: mai întîi memoria ocupată de şirul de caractere, apoi memoria ocupată de nodul propriu-zis.

```
void TreeFree(P_node p) {
if (p) {
   TreeFree(p->l);
   TreeFree(p->r);
   free(p->w); free(p);
                                                    }
}
```

9.7. Cîmpuri

Un cîmp se defineşte ca fiind o mulţime de biţi consecutivi dintr-un cuvînt sau întreg. Concret, din motive de economie a spaţiului de memorie, este utilă împachetarea unor obiecte într-un singur cuvînt maşină. Un caz frecvent de acest tip este utilizarea unui set de flaguri, fiecare pe un bit, pentru tabela de simboluri a unui compilator.

Fiecare simbol dintr-un program are anumite informaţii asociate lui, cum sînt de exemplu clasa de memorie, tipul, dacă este sau nu cuvînt cheie ş.a.m.d. Cel mai compact mod de a codifica aceste informaţii este folosirea unui set de flaguri, de cîte un bit, într-un singur întreg sau caracter.

Modul cel mai uzual pentru a face acest lucru este de a defini un set de măşti, fiecare mască fiind corespunzătoare poziţiei bitului în interiorul caracterului sau cuvîntului. De exemplu:

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC 04
```

definesc măştile **KEYWORD**, **EXTERNAL** şi **STATIC** care se referă la biţii 0, 1 şi respectiv 2 din caracter sau cuvînt. Atunci accesarea acestor biţi se realizează cu ajutorul operaţiilor de deplasare, mascare şi complementare, descrişi într-un capitol anterior. Numerele trebuie să fie puteri ale lui 2.

Expresii de forma următoare apar frecvent şi ele setează biţii 1 şi 2 din caracterul sau întregul `flags` (în exemplul nostru):

```
flags | = EXTERNAL | STATIC;
```

Expresia următoare selectează biţii 1 şi 2 din flags:

```
flags &= EXTERNAL | STATIC;
```

Expresia următoare este adevărată cînd cel puţin unul din biţii 1 sau 2 din flags este unu:

```
if (flags & (EXTERNAL | STATIC))...
```

Expresia următoare este adevărată cînd biţii 1 şi 2 din flags sînt ambii zero:

```
if (!(flags & (EXTERNAL | STATIC)))...
```

Limbajul C oferă aceste expresii ca o alternativă pentru posibilitatea de a defini şi de a accesa biţii dintr-un cuvînt în mod direct, folosind operatorii logici pe biţi.

Sintaxa definiţiei cîmpului şi a accesului la el se bazează pe structuri. De exemplu construcţiile **`#define`** din exemplul de mai sus pot fi înlocuite prin definirea a trei cîmpuri:

```
struct {
   unsigned is_keyword:1;
   unsigned is_external:1;
   unsigned is_static:1;
   } flags;
```

Această construcţie defineşte variabila **`flags`** care conţine 3 cîmpuri, fiecare de cîte un bit. Numărul care urmează după **`:`** (două puncte) reprezintă lungimea cîmpului în biţi. Cîmpurile sînt declarate **`unsigned`** pentru a sublinia că ele sînt cantităţi fără semn. Pentru a ne referi la un cîmp individual din variabila **`flags`** folosim o notaţie similară cu notaţia folosită pentru membrii structurilor.

```
flags.is_keyword
flags.is_static
```

Cîmpurile se comportă ca nişte întregi mici fără semn şi pot participa în expresii aritmetice ca orice alţi întregi. Astfel, expresiile anterioare pot fi scrise mai natural sub forma următoare:

```
flags.is_extern = flags.is_static = 1;
```

pentru setarea biţilor 1 şi 2 din variabila **`flags`**,

```
flags.is_extern = flags.is_static = 0;
```
pentru ştergerea biţilor, iar:
```
if (flags.is_extern==0 && flags.is_static==0)
```
pentru testarea lor.

Un cîmp nu trebuie să depăşească limitele unui cuvînt (16 sau 32 de biţi, în funcţie de sistemul de calcul). În caz contrar, cîmpul se aliniază la limita următorului cuvînt. Cîmpurile nu necesită să fie denumite. Un cîmp fără nume, descris numai prin caracterul **:** şi lungimea lui în biţi, este folosit pentru a rezerva spaţiu în vederea alinierii următorului cîmp. Lungimea zero a unui cîmp poate fi folosită pentru forţarea alinierii următorului cîmp la limita unui nou cuvînt, el fiind presupus a conţine tot cîmpuri şi nu un membru obişnuit al structuri, deoarece în acest ultim caz alinierea se face în mod automat. Nici un cîmp nu poate fi mai lung decît un cuvînt. Cîmpurile se atribuie de la dreapta la stînga, de la poziţiile cele mai puţin semnificative la cele mai semnificative.

Cîmpurile nu pot constitui masive, nu au adrese, astfel încît operatorul & nu se poate aplica asupra lor.

9.8. Reuniuni

O reuniune este o variabilă care poate conţine, la momente diferite, obiecte de diferite tipuri şi dimensiuni; compilatorul este cel care ţine evidenţa dimensiunilor şi aliniamentului.

Reuniunile oferă posibilitatea ca mai multe tipuri diferite de date să fie tratate într-o singură zonă de memorie.

Să reluăm exemplul tabelei de simboluri a unui compilator, presupunînd că se gestionează constante de tip **int**, **float** sau şiruri de caractere.

Valoarea unei constante particulare ar putea fi memorată într-o variabilă de tip corespunzător, dar este mai convenabil, pentru gestiunea tabelei de simboluri, ca valoarea să fie memorată în aceeaşi zonă de memorie, indiferent de tipul ei. Acesta este scopul unei reuniuni: de a furniza o singură variabilă care să poată conţine oricare dintre valorile unor tipuri de date. Ca şi în cazul cîmpurilor, sintaxa

definiției şi accesului la o reuniune se bazează pe structuri. Fie definiția:

```
union u_tag {
   int ival;
   float fval;
   char *pval;
   } uval;
```

Variabila **uval** va fi suficient de mare ca să poată păstra pe cea mai mare dintre cele trei tipuri de componente. Oricare dintre tipurile de mai sus poate fi atribuit variabilei **uval** şi apoi folosit în expresii în mod corespunzător, adică tipul în **uval** este tipul ultim atribuit. Utilizatorul este cel care ține evidența tipului curent memorat într-o reuniune.

Sintactic, membrii unei reuniuni sînt accesibili printr-o construcție de forma:

nume-reuniune **.** *membru*

pointer-la-reuniune **->** *membru*

Dacă variabila **utype** este utilizată pentru a ține evidența tipului curent memorat în **uval**, atunci putem avea următorul cod:

```
switch (utype) {
   case INT:
      printf ("%d\n",uval.ival); break;
   case FLOAT:
      printf("%f\n",uval.fval); break;
   case STRING:
      printf("%s\n",uval.pval); break;
   default:
      printf("tip incorect %d in utype\n",utype);
      break;
   }
```

Reuniunile pot apărea în structuri şi masive şi invers. Sintaxa pentru accesarea unui membru al unei reuniuni dintr-o structură (sau invers) este identică cu cea pentru structurile imbricate. De exemplu, în masivul de structuri **symtab[NSYM]** definit de:

```
struct {
   char *name;
   int flags,utype;
   union {
      int ival;
      float fval;
      char *pval;
      } uval;
   } symtab[NSYM];
```

variabila **ival** se referă prin:

```
symtab[i].uval.ival
```

iar primul caracter al şirului indicat de **pval** prin:

```
*symtab[i].uval.pval
```

Tehnic, o reuniune este o structură în care toți membrii au deplasamentul zero, structura fiind suficient de mare pentru a putea păstra pe cel mai mare membru. Alinierea este corespunzătoare pentru toate tipurile reuniunii. Pointerii la reuniuni pot fi folosiți în mod similar cu pointerii la structuri.

Următorul exemplu trebuie studiat cu atenție, deoarece implementarea este dependentă de arhitectura sistemului de calcul. De aceea vom prezenta două declaraţii de tip, corespunzătoare celor două moduri de reprezentare a valorilor întregi:

– pentru arhitecturi *Big Endian*:

```
typedef struct {
   short an;
   char luna, zi;
   } T_struc;
```

– pentru arhitecturi *Little Endian*:

```
typedef struct {
   char luna, zi;
   short an;
   } T_struc;
```

În continuare putem defini un tip dată calendaristică:

```
typedef union {
   T_struc ds;
   long dn;
   } T_data;
```

Fie următoarele definiții de variabile:

```
T_data dc1, dc2;
```

În continuare putem interpreta conținutul variabilelor **dc1** şi **dc2** fie ca structuri fie ca întregi lungi.

O variabilă de acest tip se inițializează astfel:

```
dc1.ds.an = ...;
dc1.ds.luna = ...;
dc1.ds.zi = ...;
```

Două date calendaristice se compară cît se poate de natural:

```
dc1.dn < dc2.dn
```

Cum putem depista automat ce fel de arhitectură are sistemul de calcul?

```
union {
   long l;
   char c[4];
   } tst = 0x12345678;
if (tst.c[0]==0x12) puts("Big Endian");
   else puts("Little Endian");
```

10. Operații de intrare / ieşire

Întrucît limbajul C nu a fost dezvoltat pentru un sistem particular de operare, şi datorită faptului că s-a dorit realizarea unei portabilități cît mai mari, atît a unui compilator C, cît şi a programelor scrise în acest limbaj, el nu posedă facilități de intrare / ieşire.

Există totuşi un sistem de intrare / ieşire constituit dintr-un număr de subprograme care realizează funcții de intrare / ieşire pentru programe scrise în C, dar care nu fac parte din limbajul C. Aceste subprograme se găsesc în biblioteca C.

Scopul acestui capitol este de a descrie cele mai utilizate subprograme de intrare / ieşire şi interfața lor cu programele scrise în limbajul C.

10.1. Intrări şi ieşiri standard; fişiere

Sistemul I/O oferă utilizatorului trei „*fişiere*" standard de lucru. Cuvîntul fişier a fost pus între ghilimele, deoarece limbajul nu defineşte acest tip de dată şi pentru că fişierele reprezintă mai degrabă nişte fluxuri de intrare / ieşire standard puse la dispoziția utilizatorului. Aceste fişiere sînt:

– fişierul standard de intrare (**`stdin`**);
– fişierul standard de ieşire (**`stdout`**);
– fişierul standard de afişare a mesajelor (**`stderr`**).

Toate aceste trei fişiere sînt secvențiale şi în momentul execuției unui program C sînt implicit definite şi deschise.

`stdin` şi **`stdout`** sînt asociate în mod normal terminalului de la care a fost lansat programul în execuție. Sistemul I/O permite redirectarea acestor fişiere pe alte periferice şi închiderea lor la încheierea execuției programului. Redirectarea fişierului **`stdin`** se specifică prin construcția:

`<`*specificator-fişier-intrare*

în linia de comandă prin care a fost lansat programul.

Redirectarea fişierului **stdout** se specifică prin construcţia:

>*specificator-fişier-ieşire*

în linia de comandă prin care a fost lansat programul.

Redirectarea fişierului **stdout** pe un alt periferic, în scopul efectuării unei operaţii de adăugare (append) se specifică prin construcţia:

>>*specificator-fişier-ieşire*

stderr este întotdeauna asociat terminalului de la care a fost lansat programul în execuţie şi nu poate fi redirectat.

Astfel că o linie de comandă a unui program poate arăta astfel:

Prog **<***intrare-standard* **>***ieşire-standard alţi parametri*

Prog **<** *intrare-standard* **>>** *ieşire-standard alţi parametri*

Fiecare din parametri poate lipsi; în lipsa unui specificator de intrare sau ieşire standard se foloseşte terminalul curent. Asocierea unui fişier cu intrarea sau ieşirea standard este făcută de sistemul de operare, programul primind doar informaţii despre ceilalţi parametri din linia de comandă.

Pentru a se putea face o referire la fişiere orice program C trebuie să conţină fişierul **stdio.h**, care se include printr-o linie de forma:

```
#include <stdio.h>
```

Pentru claritatea şi lizibilitatea programelor scrise în C, cît şi pentru crearea unei imagini sugestive asupra lucrului cu fişiere, în fişierul de definiţii standard **stdio.h** s-a definit un nou nume de tip de dată şi anume **FILE** care este o structură. Pentru a referi un fişier, este necesară o declaraţie de forma:

```
FILE *fp;
```

unde ***fp*** va fi numele de dată cu care se va referi fişierul în orice operaţie de intrare / ieşire asociată. Iată cîteva informaţii păstrate de structura **FILE**:

– un identificator de fişier pe care sistemul de operare îl asociază fluxului pe durata prelucrării; acesta poate fi aflat cu ajutorul funcţiei **fileno**;

– adresele zonelor tampon asociate; poziţia curentă în aceste zone;

– indicatorii de sfîrşit de fişier şi de eroare;

– alte informaţii.

10.2. Accesul la fişiere; deschidere şi închidere

Nume

fopen - deschide un flux

Declaraţie

```
FILE *fopen(const char *num, const char *mod);
```

Descriere

Funcţia **fopen** deschide fişierul al cărui nume este un şir indicat de **num** şi îi asociază un flux.

Argumentul **mod** indică un şir care începe cu una din secvenţele următoare:

r deschide un fişier pentru citire;

r+ deschide pentru citire şi scriere;

w trunchiază fişierul la lungime zero sau creează un fişier pentru scriere;

w+ deschide pentru adăugare la sfîrşit, în citire şi scriere; fişierul este creat dacă nu există, altfel este trunchiat;

a deschide pentru adăugare la sfîrşit, în scriere; fişierul este creat dacă nu există;

a+ deschide pentru adăugare la sfîrşit, în citire şi scriere; fişierul este creat dacă nu există;

După deschidere, în primele patru cazuri indicatorul poziţiei în flux este la începutul fişierului, în ultimele două la sfîrşitul acestuia.

Şirul **mod** include de asemenea litera **b** (deschide un fişier binar) sau **t** (deschide un fişier text) fie pe ultima poziţie fie pe cea din mijloc.

Operaţiile de citire şi scriere pot alterna în cazul fluxurilor read / write în orice ordine. Să reţinem că standardul ANSI C cere să existe o funcţie de poziţionare între o operaţie de intrare şi una de ieşire, sau între o operaţie de ieşire şi una de intrare, cu excepţia cazului cînd o operaţie de citire detectează sfîrşitul de fişier. Această operaţie poate fi inefectivă – cum ar fi **fseek(flux, 0L, SEEK_CUR)** apelată cu scop de sincronizare.

Valori returnate

În caz de succes se returnează un pointer de tip **FILE**. În caz de eroare se returnează NULL şi variabila globală **errno** indică codul erorii.

Nume

fclose - închide un flux

Declarație

```
int fclose( FILE *flux);
```

Descriere

Funcția **fclose** închide fişierul asociat fluxului **flux**. Dacă **flux** a fost deschis pentru ieşire, orice date aflate în zone tampon sînt scrise în fişier în prealabil cu un apel **fflush**.

Valori returnate

În caz de succes se returnează **0**. În caz de eroare se returnează **EOF** şi variabila globală **errno** indică codul erorii.

Nume

tmpfile - creează un fişier temporar

Declarație

```
FILE *tmpfile();
```

Descriere

Funcția **tmpfile** generează un nume unic de fişier temporar. Acesta este deschis în mod binar pentru scriere / citire (**"wb+"**). Fişierul va fi şters automat la închidere sau la terminarea programului.

Valoare returnată

Funcția returnează un descriptor de flux în caz de succes, sau **NULL** dacă nu poate fi generat un nume unic de fişier sau dacă fişierul nu poate fi deschis. În caz de eroare variabila globală **errno** indică codul erorii.

Nume

fflush - forțează scrierea în flux

Declarație

```
int fflush(FILE *flux);
```

Descriere

Funcția **fflush** forțează o scriere a tuturor datelor aflate în zone tampon ale fluxului **flux**. Fluxul rămîne deschis.

Valori returnate

În caz de succes se returnează **0**. În caz de eroare se returnează **EOF** şi variabila globală **errno** indică codul erorii.

Nume

fseek, **ftell**, **rewind** - repoziționează un flux

Declarație

```
int fseek(FILE *flux, long ofs, int reper);
long ftell(FILE *flux);
void rewind(FILE *flux);
```

Descriere

Funcția **fseek** setează indicatorul de poziție pentru fişierul asociat fluxului **flux**. Noua poziție, dată în octeți, se obține adunînd **ofs** octeți (offset) la poziția specificată de **reper**. Dacă **reper** este **SEEK_SET**, **SEEK_CUR**, sau **SEEK_END**, **ofs** este relativ la începutul fişierului, poziția curentă a indicatorului, respectiv sfîrşitul fişierului. Funcția **fseek** şterge indicatorul de sfîrşit de fişier.

Funcția **ftell** obține valoarea curentă a indicatorului de poziție pentru fişierul asociat fluxului **flux**.

Funcția **rewind** poziționează indicatorul de poziție pentru fişierul asociat fluxului **flux** la începutul fişierului. Este echivalentă cu:

```
(void)fseek(flux, 0L, SEEK_SET)
```

cu completarea că funcția **rewind** şterge şi indicatorul de eroare al fluxului.

Valori returnate

Funcția **rewind** nu returnează nici o valoare. În caz de succes, **fseek** returnează **0**, şi **ftell** returnează offset-ul curent. În caz de eroare se returnează **EOF** şi variabila globală **errno** indică codul erorii.

Nume

fileno – returnează descriptorul asociat fluxului

Declarație

```
int fileno(FILE *flux);
```

Descriere

Funcția **fileno** examinează argumentul **flux** şi returnează descriptorul asociat de sistemul de operare acestui flux.

10.3. Citire şi scriere fără format

Nume

fgets - citeşte un şir de caractere dintr-un flux text

Declarație

```
char *fgets(char *s, int size, FILE *flux);
```

Descriere

Funcția **fgets** citeşte cel mult **size-1** caractere din **flux** şi le memorează în zona indicată de **s**. Citirea se opreşte la detectarea sfîrşitului de fişier sau new-line. Dacă se citeşte caracterul new-line acesta este memorat în **s**. După ultimul caracter se memorează null.

Apeluri ale acestei funcții pot fi combinate cu orice apeluri ale altor funcții de intrare din bibliotecă (**fscanf**, de exemplu) pentru un acelaşi flux de intrare.

Valori returnate

Funcția returnează adresa **s** în caz de succes, sau **NULL** în caz de eroare sau la întîlnirea sfîrşitului de fişier dacă nu s-a citit nici un caracter.

Nume

fputs - scrie un şir de caractere într-un flux text

Declarație

```
int fputs(const char *s, FILE *flux);
```

Descriere

Funcția **fputs** scrie şirul **s** în flux fără caracterul terminator null.

Apeluri ale acestei funcții pot fi combinate cu orice apeluri ale altor funcții de ieşire din bibliotecă (**fprintf**, de exemplu) pentru un acelaşi flux de ieşire.

Valori returnate

Funcția returnează o valoare non-negativă în caz de succes, sau **EOF** în caz de eroare.

Nume

fread, **fwrite** - intrări / ieşiri pentru fluxuri binare

Declarație

```
unsigned fread(void *ptr, unsigned size,
         unsigned nel, FILE *flux);
unsigned fwrite(const void *ptr, unsigned
         size, unsigned nel, FILE *flux);
```

Descriere

Funcția **fread** citeşte **nel** elemente, fiecare avînd mărimea **size** octeți, din fluxul indicat de **flux**, şi le memorează în zona indicată de **ptr**.

Funcția **fwrite** scrie **nel** elemente, fiecare avînd mărimea **size** octeți, din fluxul indicat de **flux**, pe care le ia din zona indicată de **ptr**.

Valori returnate

Funcțiile returnează numărul de elemente citite sau scrise cu succes (şi nu numărul de caractere). Dacă apare o eroare sau se întîlneşte sfîrşitul de fişier, valoarea returnată este mai mică decît **nel** (posibil zero).

10.4. Citire cu format

Nume

scanf, **fscanf**, **sscanf** - citire cu format

Declarație

```
int scanf(const char *fmt, ...);
int fscanf(FILE *flux, const char *fmt, ...);
int sscanf(char *str, const char *fmt, ...);
```

Descriere

Funcțiile din familia *...scanf* scanează intrarea în concordanță cu şirul de caractere **fmt** după cum se descrie mai jos. Acest format poate conține specificatori de conversie; rezultatele unor astfel de conversii (dacă se efectuează) se memorează prin intermediul argumentelor pointer. Funcția **scanf** citeşte şirul de intrare din fluxul standard **stdin**, **fscanf** din **flux**, şi **sscanf** din şirul indicat de **str**.

Fiecare argument pointer trebuie să corespundă în ordine ca tip cu fiecare specificator de conversie (dar a se vedea *suprimarea* mai jos). Dacă argumentele nu sînt suficiente comportamentul programului este imprevizibil. Toate conversiile sînt introduse de caracterul `%`. Şirul format poate conține şi alte caractere. Spații albe (blanc, tab, sau new-line) din şirul format se potrivesc cu orice spațiu alb în orice număr (inclusiv nici unul) din şirul de intrare. Orice alte caractere trebuie să se potrivească exact. Scanarea se opreşte atunci cînd un caracter din şirul de intrare nu se potriveşte cu cel din format. Scanarea se opreşte de asemenea atunci cînd o conversie nu se mai poate efectua (a se vedea mai jos).

Conversii

După caracterul `%` care introduce o conversie poate urma un număr de caractere indicatori, după cum urmează:

`*` Suprimă atribuirea. Conversia care urmează se face în mod obişnuit, dar nu se foloseşte nici un argument pointer; rezultatul conversiei este pur şi simplu abandonat.

h Conversia este de tip **d**, **i**, **o**, **u**, **x** sau **n** şi argumentul asociat este un pointer la **short** (în loc de **int**).

l Conversia este de tip **d**, **i**, **o**, **u**, **x** sau **n** şi argumentul asociat este un pointer la **long** (în loc de **int**), sau conversia este de tip **e**, **f**, **g** şi argumentul asociat este un pointer la **double** (în loc de **float**).

L Conversia este de tip **e**, **f**, **g** şi argumentul asociat este un pointer la **long double**.

În completare la aceşti indicatori poate exista o mărime *w* maximă opțională pentru cîmp, exprimată ca un întreg zecimal, între caracterul **%** şi cel de conversie, şi înaintea indicatorului. Dacă nu este dată o mărime maximă se foloseşte mărimea implicită „infinit” (cu o excepție la conversia de tip **c**); în caz contrar se scanează cel mult un număr de *w* caractere în timpul conversiei. Înainte de a începe o conversie, majoritatea conversiilor ignoră spațiile albe; acestea nu sînt contorizate în mărimea cîmpului.

Sînt disponibile următoarele conversii:

% Potrivire cu un caracter **%**. Cu alte cuvinte, **%%** în şirul format trebuie să se potrivească cu un caracter **%**. Nu se efectuează nici o conversie şi nici o atribuire.

d Potrivire cu un întreg zecimal (eventual cu semn); argumentul asociat trebuie să fie un pointer la **int**.

i Potrivire cu un întreg (eventual cu semn); argumentul asociat trebuie să fie un pointer la **int**. Valoarea întreagă este citită în baza 16 dacă începe cu **0x** sau **0X**, în baza 8 dacă începe cu **0**, şi în baza 10 în caz contrar. Sînt folosite numai caracterele care corespund bazei respective.

o Potrivire cu un întreg octal fără semn; argumentul asociat trebuie să fie un pointer la **unsigned**.

u Potrivire cu un întreg zecimal fără semn; argumentul asociat trebuie să fie un pointer la **unsigned**.

x Potrivire cu un întreg hexazecimal fără semn; argumentul asociat trebuie să fie un pointer la **unsigned**.

f Potrivire cu un număr în virgulă mobilă (eventual cu semn); argumentul asociat trebuie să fie un pointer la **float**.

e,**g** Echivalent cu **f**.

s Potrivire cu o secvență de caractere diferite de spațiu alb; argumentul asociat trebuie să fie un pointer la **char**, şi zona trebuie să fie suficient de mare pentru a putea primi toată secvența şi caracterul terminator null. Şirul de intrare se termină la un spațiu alb sau la atingerea mărimii maxime a cîmpului (prima condiție întîlnită).

c Potrivire cu o secvență de caractere de mărime *w* (dacă aceasta este specificată; prin lipsă se ia *w*=1); argumentul asociat trebuie să fie un pointer la **char**, şi zona trebuie să fie suficient de mare pentru a putea primi toată secvența (nu se adaugă terminator null). Nu se ignoră ca de obicei spațiile albe din față. Pentru a ignora mai întîi spațiile albe se indică un spațiu explicit în format.

[Potrivire cu o secvență nevidă de caractere din setul specificat de caractere acceptate; argumentul asociat trebuie să fie un pointer la **char**, şi zona trebuie să fie suficient de mare pentru a putea primi toată secvența şi caracterul terminator null. Nu se ignoră ca de obicei spațiile albe din față. Şirul de intrare va fi format din caractere aflate în (sau care nu se află în) setul specificat în format; setul este definit de caracterele aflate între **[** şi **]**. Setul exclude acele caractere dacă primul caracter după **[** este **^**. Pentru a include caracterul **]** în set, acesta trebuie să fie primul caracter după **[** sau **^**; caracterul **]** aflat în orice altă poziție închide setul. Caracterul **-** are şi el un rol special: plasat între două alte caractere adaugă toate celelalte caractere aflate în intervalul respectiv la set. Pentru a include caracterul **-** acesta trebuie să fie ultimul caracter înainte de **]**. De exemplu, **"%[^]0-9-]"** semnifică setul *orice caracter cu excepția* **]**, **0**

pînă la **9**, *şi* **-**. Şirul se termină la apariţia unui caracter care nu se află (sau, dacă se precizează **^**, care se află) în set sau dacă se atinge mărimea maximă specificată.

n Nu se prelucrează nimic din şirul de intrare; în schimb, numărul de caractere consumate pînă la acest punct din şirul de intrare este memorat la argumentul asociat, care trebuie să fie un pointer la **int**.

Valori returnate

Funcţiile returnează numărul de valori atribuite, care poate fi mai mic decît numărul de argumente pointer, sau chiar zero, în cazul în care apar nepotriviri între format şi şirul de intrare. Zero indică faptul că, chiar dacă avem un şir de intrare disponibil, nu s-a efectuat nici o conversie (şi atribuire); această situaţie apare atunci cînd un caracter din şirul de intrare este invalid, cum ar fi un caracter alfabetic pentru o conversie **%d**. Valoarea **EOF** este returnată dacă apare o eroare înainte de prima conversie, cum ar fi detectarea sfîrşitului de fişier. Dacă o eroare sau un sfîrşit de fişier apare după ce o conversie a început, se returnează numărul de conversii efectuate cu succes, şi se poziţionează bitul corespunzător din structura **FILE**, care poate fi testat.

10.5. Scriere cu format

Nume

printf, **fprintf**, **sprintf** - scriere cu format

Declaraţie

```
int printf(const char *fmt, ...);
int fprintf(FILE *flux, const char *fmt, ...);
int sprintf(char *str, const char *fmt, ...);
```

Descriere

Funcţiile din familia *...printf* generează o ieşire în concordanţă cu format după cum se descrie mai jos. Funcţia **printf** afişează ieşirea la fluxul standard **stdout**; **fprintf** scrie ieşirea la **flux**; **sprintf** scrie ieşirea în şirul de caractere **str**.

Aceste funcții generează ieşirea sub controlul şirului format care specifică cum se convertesc argumentele pentru ieşire.

Şirul de formatare

Şirul **fmt** este un şir de caractere, printre care se pot afla zero sau mai multe directive: caractere obişnuite (diferite de **%**) care sînt copiate aşa cum sînt în fluxul de ieşire, şi specificații de conversie, fiecare dintre ele rezultînd din încărcarea a zero sau mai multe argumente. Fiecare specificație de conversie este introdusă de caracterul **%** şi se termină cu un specificator de conversie. Între acestea pot fi (în această ordine) zero sau mai mulți indicatori, o mărime minimă a cîmpului opțională, o precizie opțională şi un modificator opțional de lungime.

Argumentele trebuie să corespundă în ordine ca tip cu specificatorii de conversie. Acestea sînt folosite în ordinea dată, unde fiecare caracter ***** şi fiecare specificator de conversie solicită următorul argument. Dacă argumentele nu sînt suficiente comportamentul programului este imprevizibil.

Caractere indicatori

Caracterul **%** este urmat de zero, unul sau mai mulți indicatori:

0 Valoarea numerică este convertită cu zerouri la stînga. Pentru conversii de tip **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **F**, **g** şi **G**, valoarea convertită este completată cu zerouri la stînga în loc de blanc. Dacă apar indicatorii **0** şi **-** împreună, indicatorul **0** este ignorat. Dacă pentru o conversie numerică (**d**, **i**, **o**, **u**, **x**, **X**) este dată o precizie, indicatorul **0** este ignorat. Pentru alte conversii rezultatul este nedefinit.

- Valoarea convertită este aliniată la stînga (implicit alinierea se face la dreapta). Cu excepția conversiilor de tip **n**, valoarea convertită este completată la dreapta cu blanc, în loc să fie completată la stînga cu blanc sau zero. Dacă apar indicatorii **0** şi **-** împreună, indicatorul **0** este ignorat.

Sp (spațiu) În cazul unui rezultat al unei conversii cu semn, înaintea unui număr pozitiv sau şir vid se pune un blanc.

+ Semnul (**+** sau **-**) este plasat înaintea numărului generat de o conversie cu semn. Implicit semnul este folosit numai pentru numere negative. Dacă apar indicatorii **+** şi *Sp* împreună, indicatorul *Sp* este ignorat.

Lăţimea cîmpului

Un şir de cifre zecimale (cu prima cifră nenulă) specifică o lăţime minimă pentru cîmp. Dacă valoarea convertită are mai puţine caractere decît lăţimea specificată, va fi completată cu spaţii la stînga (sau dreapta, dacă s-a specificat aliniere la stînga). În locul unui număr zecimal se poate folosi * pentru a specifica faptul că lăţimea cîmpului este dată de argumentul corespondent, care trebuie să fie de tip **int**. O valoare negativă pentru lăţime este considerată un indicator **-** urmat de o valoare pozitivă pentru lăţime. În nici un caz nu se va trunchia cîmpul; dacă rezultatul conversiei este mai mare decît lăţimea cîmpului, cîmpul este expandat pentru a conţine rezultatul conversiei.

Precizia

Precizia (opţională) este dată de caracterul **.** urmat de un şir de cifre zecimale. În locul şirului de cifre zecimale se poate scrie ***** pentru a specifica faptul că precizia este dată de argumentul corespondent, care trebuie să fie de tip **int**. Dacă precizia este dată doar de caracterul **.** sau dacă precizia este negativă, atunci aceasta se consideră zero. Precizia dă numărul minim de cifre care apar pentru conversii de tip **d**, **i**, **o**, **u**, **x**, **X**, numărul de cifre care apar după punctul zecimal pentru conversii de tip **e**, **E**, **f**, **F**, numărul maxim de cifre semnificative pentru conversii de tip **g** şi **G**, sau numărul maxim de caractere generate pentru conversii de tip **s**.

Dacă se foloseşte ***** pentru lăţime sau precizie (sau ambele), argumentele se iau în ordine: lăţime, precizie, valoare de scris.

Modificator de lungime

În acest caz prin conversie întreagă înţelegem conversie de tip **d**, **i**, **o**, **u**, **x**, **X**.

h Conversia întreagă care urmează corespunde unui argument **short** sau **unsigned short**, sau următoarea conversie de tip **n** corespunde unui argument de tip pointer la **short**.

l Conversia care urmează corespunde unui argument de tip:
– **long** sau **unsigned long**, dacă este o conversie de tip **d**, **i**, **o**, **u**, **x**, **X** sau **n**;
– **double**, dacă este o conversie de tip **f**.

L Următoarea conversie de tip **e**, **E**, **f**, **F**, **g**, **G** corespunde unui argument **long double**.

Specificator de conversie

Un caracter care specifică tipul conversiei care se va face. Specificatorii de conversie şi semnificația lor sînt:

d,**i**

Argumentul de tip **int** este convertit la notația zecimală cu semn. Precizia, dacă este dată, dă numărul minim de cifre care trebuie să apară; dacă valoarea convertită necesită mai puține cifre, aceasta este completată la stînga cu zerouri. Precizia implicită este 1. Dacă valoarea 0 este afişată cu precizie explicită 0, ieşirea este vidă.

o,**u**,**x**,**X**

Argumentul de tip **unsigned** este convertit la notație octală fără semn (**o**), zecimală fără semn (**u**), sau hexazecimală fără semn (**x** şi **X**). Literele **abcdef** se folosesc pentru conversii de tip **x**, literele **ABCDEF** pentru conversii de tip **X**. Precizia, dacă este dată, dă numărul minim de cifre care trebuie să apară; dacă valoarea convertită necesită mai puține cifre, aceasta este completată la stînga cu zerouri. Precizia implicită este 1. Dacă valoarea 0 este afişată cu precizie explicită 0, ieşirea este vidă.

e,**E**

Argumentul de tip flotant este rotunjit şi convertit în stil [**-**]*d.ddd***e±***dd* unde avem o cifră înainte de punctul zecimal şi numărul de cifre după acesta este egal cu precizia; dacă aceasta

lipseşte se consideră 6; dacă precizia este zero, punctul zecimal nu apare. O conversie de tip **E** foloseşte litera **E** (în loc de **e**) pentru a introduce exponentul. Exponentul are întotdeauna cel puţin două cifre; dacă valoarea este zero, exponentul este **00**.

f,**F**
Argumentul de tip flotant este rotunjit şi convertit în notaţie zecimală în stil [**-**]*ddd*.*ddd*, unde numărul de cifre după punctul zecimal este egal cu precizia specificată. Dacă precizia lipseşte se consideră 6; dacă precizia este explicit zero, punctul zecimal nu apare. Dacă punctul zecimal apare, cel puţin o cifră apare înaintea acestuia.

g,**G**
Argumentul de tip flotant este convertit în stil **f** sau **e** (sau **E** pentru conversii de tip **G**). Precizia specifică numărul de cifre semnificative. Dacă precizia lipseşte se consideră 6; dacă precizia este zero se consideră 1. Stilul **e** este folosit dacă exponentul rezultat în urma conversiei este mai mic decît –4 ori mai mare sau egal cu precizia. Zerourile finale sînt eliminate din partea fracţionară a rezultatului; punctul zecimal apare numai dacă este urmat de cel puţin o cifră.

c Argumentul de tip **int** este convertit la **unsigned char** şi se scrie caracterul rezultat.

s Argumentul de tip **const char *** este un pointer la un şir de caractere. Caracterele din şir sînt scrise pînă la (fără a include) caracterul terminator null; dacă precizia este specificată, nu se scrie un număr mai mare decît cel specificat. Dacă precizia este dată, nu e nevoie de caracterul null; dacă precizia nu este specificată, şirul trebuie să conţină un caracter terminator null.

p Argumentul de tip pointer este scris în hexazecimal; formatul este specific sistemului de calcul.

n Numărul de caractere scrise pînă în acest moment este memorat la argumentul de tip **int ***. Nu se face nici o conversie.

`%` Se scrie un caracter `%`. Nu se face nici o conversie. Specificația completă este `%%`.

Valoare returnată

Funcțiile returnează numărul de caractere generate (nu se include caracterul terminator null pentru **`sprintf`**).

10.6. Tratarea erorilor

Nume

`perror` - afişează un mesaj de eroare sistem

Declarație

```
void perror(const char *s);

#include <errno.h>
const char *sys_errlist[];
int sys_nerr;
```

Descriere

Rutina **`perror`** afişează un mesaj la ieşirea standard de eroare, care descrie ultima eroare întîlnită la ultimul apel sistem sau funcție de bibliotecă. Mai întîi se afişează argumentul **`s`**, apoi virgula şi blanc, şi în final mesajul de eroare şi new-line. Se recomandă (mai ales pentru depanare) ca argumentul **`s`** să includă numele funcției în care a apărut eroarea. Codul erorii se ia din variabila externă **`errno`**.

Lista globală de erori **`sys_errlist[]`** indexată cu **`errno`** poate fi folosită pentru a obține mesajul de eroare fără new-line. Ultimul indice de mesaj din listă este **`sys_nerr-1`**. Se recomandă o atenție deosebită în cazul accesului direct la listă deoarece unele coduri noi de eroare pot lipsi din **`sys_errlist[]`**.

Dacă un apel sistem eşuează variabila **`errno`** indică codul erorii. Aceste valori pot fi găsite în **`<errno.h>`**. Funcția **`perror`** serveşte la afişarea acestui cod de eroare într-o formă lizibilă. Dacă un apel terminat cu eroare nu este imediat urmat de un apel **`perror`**, valoarea variabilei **`errno`** se poate pierde dacă nu e salvată.

Nume

clearerr, **feof**, **ferror** - verifică şi resetează starea fluxului

Declarație

```
void clearerr(FILE *flux);
int feof(FILE *flux);
int ferror(FILE *flux);
```

Descriere

Funcția **clearerr** şterge indicatorii de sfîrşit de fişier şi eroare ai fluxului.

Funcția **feof** testează indicatorul de sfîrşit de fişier al fluxului, şi returnează non-zero dacă este setat. Acesta este setat dacă o operație de citire a detectat sfîrşitul de fişier.

Funcția **ferror** testează indicatorul de eroare al fluxului, şi returnează non-zero dacă este setat. Acesta este setat dacă o operație de citire sau scriere a detectat o eroare (datorată de exemplu hardware-ului).

Funcțiile de citire (cu sau fără format) nu fac distincție între sfîrşit de fişier şi eroare, astfel că trebuie apelate funcțiile **feof** şi **ferror** pentru a determina cauza terminării.

Atenție! Este foarte frecventă folosirea incorectă a funcției **feof** pentru a testa dacă s-a ajuns la sfîrşitul fişierului. Nu se recomandă în nici un caz acest stil de programare:

```
#define LSIR 80
char lin[LSIR];
FILE *fi,*fo;
fi=fopen(nume-fişier-intrare,"rt");
fo=fopen(nume-fişier-ieşire,"wt");
while (!feof(fi)) {          /* greşit! */
   fgets(lin,LSIR,fi);
   fputs(lin,fo);
   }
fclose(fi); fclose(fo);
```

În această secvență, dacă și ultima linie a fișierului text de intrare este terminată cu new-line, aceasta va fi scrisă de două ori în fișierul de ieșire. De ce? După ce se citește ultima linie încă nu este poziționat indicatorul de sfîrșit de fișier, deci funcția **fgets** încă returnează succes. La reluarea ciclului se încearcă un nou **fgets** și abia acum se depistează sfîrșitul de fișier, fapt marcat în zona rezervată fluxului **fi**. Astfel conținutul șirului **lin** rămîne nemodificat și este scris a doua oară în fișierul de ieșire. Abia la o nouă reluare a ciclului funcția **feof** ne spune că s-a depistat sfîrșitul de fișier.

În acest manual sînt prezentate mai multe programe care efectuează diferite prelucrări asupra unor fișiere text. Pentru simplitate toate programele presupun că nu apar erori la citire sau la scriere.

10.7. Operații cu directoare

Funcțiile de parcurgere a cataloagelor de fișiere descrise în această secțiune (**opendir**, **readdir**, **closedir**) sînt definite de mai multe medii de programare C (printre care și GNU C pentru sistemul de operare Linux), precum și de standardul POSIX. Aceste funcții sînt declarate în **<dirent.h>**.

Funcțiile de redenumire și ștergere a unor fișiere sînt descrise în **<stdio.h>**.

Nume

opendir - deschide un director

Declarație

```
DIR *opendir(const char *nume);
```

Descriere

Funcția **opendir** deschide un flux pentru directorul cu numele **nume**, și returnează un pointer la fluxul deschis. Fluxul este poziționat pe prima intrare din director.

Valoare returnată

Funcția returnează un pointer la flux în caz de succes, sau NULL în caz de eroare şi variabila globală **errno** indică codul erorii.

Cîteva erori posibile

EACCES Acces interzis

ENOTDIR **nume** nu este un director

Nume

readdir - citeşte dintr-un director

Declarație

```
struct dirent *readdir(DIR *dir);
```

Descriere

Funcția **readdir** returnează un pointer la o structură de tip **dirent** care reprezintă următoarea intrare din directorul indicat de fluxul **dir**. Returnează NULL dacă s-a depistat sfîrşitul de director sau dacă a apărut o eroare.

Structura de tip **dirent** conține un cîmp **char d_name[]**. Utilizarea altor cîmpuri din structură reduce portabilitatea programelor.

Valoare returnată

Funcția returnează un pointer la o structură de tip **dirent**, sau NULL dacă s-a depistat sfîrşitul de director sau dacă a apărut o eroare.

Nume

closedir - închide un director

Declarație

```
int closedir(DIR *dir);
```

Descriere

Funcția **closedir** închide fluxul **dir**.

Valoare returnată

Funcția returnează **0** în caz de succes sau **EOF** în caz de eroare.

Nume

rename - redenumeşte un fişier

remove - şterge un fişier

Declarație

```
int rename(const char *old, const char *new);
int remove(const char *name);
```

Descriere

Funcția **rename** schimbă numele unui fişier din **old** în **new**. Dacă a fost precizat un periferic în **new**, acesta trebuie să coincidă cu cel din **old**. Directoarele din **old** şi **new** pot să fie diferite, astfel că **rename** poate fi folosită pentru a muta un fişier dintr-un director în altul. Nu se permit specificatori generici (wildcards).

Funcția **remove** şterge fişierul specificat prin **name**.

Valoare returnată

În caz de succes se returnează **0**. În caz de eroare se returnează **EOF** şi variabila globală **errno** indică codul erorii.

10.8. Programe demonstrative

Primele trei programe primesc ca parametri în linia de comandă numele fişierelor pe care le vor prelucra. Ultimul program primeşte ca parametru în linia de comandă numele directorului al cărui conținut va fi afişat.

1) Determinarea mărimii unui fişier

```
#include <stdio.h>
FILE *f;
int main(int ac, char **av) {
if (ac!=2) {
   fputs("Un argument!\n",stderr);
   return 1;
   }
f = fopen(av[1],"rb");
if (!f) {
   perror("Eroare la deschidere");
```

```
   return 1;
   }
fseek(f,0,SEEK_END);
fprintf(stderr,"File %s, size %ld\n",ftell(f));
fclose(f);
return 0;
}
```

2) Copierea unui fişier

Funcţiile **fgets** şi **fputs** se folosesc pentru fluxuri deschise în mod text. Cum se utilizează pentru copierea unui fişier text?

```
#include <stdio.h>
#define LSIR 80
char lin[LSIR];
FILE *fi, *fo;
int main(int ac, char **av) {
if (ac!=3) {
   fputs("Doua argumente!\n",stderr);
   }
fi=fopen(av[1],"rt"); fo=fopen(av[2],"wt");
if (!fi || !fo) {
   perror("Eroare la deschidere");
   return 1;
   }
while (fgets(lin,LSIR,fi))
   fputs(lin,fo);
fclose(fi); fclose(fo);
return 0;
}
```

Funcţiile **fread** şi **fwrite** se folosesc pentru fluxuri deschise în mod binar. Cum se utilizează pentru copierea unui fişier binar?

```
#include <stdio.h>
#define LZON 4096
char zon[LZON];
FILE *fi, *fo;
unsigned k;
int main(int ac, char **av) {
if (ac!=3) {
```

```
   fputs("Doua argumente!\n",stderr);
   return 1;
   }
fi=fopen(av[1],"rb"); fo=fopen(av[2],"wb");
if (!fi || !fo) {
   perror("Eroare la deschidere");
   return 1;
   }
while (k=fread(zon,1,LZON,fi))
   fwrite(zon,1,k,fo);
fclose(fi); fclose(fo);
return 0;
}
```

3) Prelucrarea unui fişier text

Programul prezentat în continuare citeşte un fişier text care conține pe fiecare linie un şir de caractere (fără spații) şi trei valori întregi, şi afişează pe terminal numele pe 12 poziții aliniat la stînga şi media aritmetică a celor trei valori întregi.

```
#include <stdio.h>
FILE *fi;
char num[10];
int a,b,c;
double m;
int main(int ac, char **av) {
if (ac!=2) {
   fputs("Un argument!\n",stderr);
   return 1;
   }
fi=fopen(av[1],"rt");
if (!fi) {
   perror("Eroare la deschidere");
   return 1;
   }
while (fscanf(fi,"%s %d %d %d",num,&a,&b,&c)!=EOF)
   {
   m=(a+b+c)/3.0;
   printf("%-12s%6.2lf\n",num,m);
   }
```

```
fclose(fi);
return 0;
}
```

4) Afişarea conţinutului unui director

```
#include <dirent.h>
#include <stdio.h>
DIR *dir;
struct dirent *ent;
int main(int ac, char **av) {
if (ac!=2) {
   printf("Un parametru\n");
   return 1;
   }
dir = opendir(av[1]);
if (!dir) {
   perror("Eroare open dir");
   return 1;
   }
while (ent=readdir(dir))
   printf("%s\n",ent->d_name);
return 0;
}
```

11. Alte rutine din biblioteca standard

În acest capitol sînt descrise funcţii care rezolvă probleme legate de alocarea dinamică a memoriei, sortare şi căutare, clasificare, operaţii cu blocuri de memorie şi şiruri de caractere, funcţii matematice.

11.1. Alocarea dinamică a memoriei

Nume

calloc, **malloc**, **realloc** - alocă memoria în mod dinamic

free - eliberează memoria alocată în mod dinamic

Declaraţie

```
#include <stdlib.h>
void *calloc(unsigned nel, unsigned size);
void *malloc(unsigned size);
void *realloc(void *ptr, unsigned size);
void free(void *ptr);
```

Descriere

Funcţia **calloc** alocă memorie pentru un masiv de **nel** elemente, fiecare de mărime **size** octeţi şi returnează un pointer la memoria alocată. Conţinutul memoriei este pus la zero.

Funcţia **malloc** alocă **size** octeţi şi returnează un pointer la memoria alocată. Conţinutul memoriei nu este şters.

Funcţia **free** eliberează spaţiul de memorie indicat de **ptr**, care trebuie să fi fost returnat de un apel anterior **malloc**, **calloc** sau **realloc**. În caz contrar, sau dacă a existat deja un apel anterior **free(ptr)**, comportamentul programului este imprevizibil.

Funcţia **realloc** modifică mărimea blocului de memorie indicat de **ptr** la **size** octeţi. Conţinutul rămîne neschimbat la mărimea minimă dintre mărimea veche şi cea nouă; noul spaţiu de

memorie care este eventual alocat este neinițializat. Dacă **ptr** este **NULL** apelul este echivalent cu **malloc(size)**; dacă **size** este egal cu zero apelul este echivalent cu **free(ptr)**. Cu excepția cazului cînd **ptr** este **NULL**, acesta trebuie să fi fost returnat de un apel precedent **malloc**, **calloc** sau **realloc**.

Valori returnate

Pentru **calloc** şi **malloc** valoarea returnată este un pointer la memoria alocată, aliniată în mod corespunzător pentru orice tip de variabile, sau **NULL** dacă nu există suficientă memorie continuă.

Funcția **free** nu returnează nimic.

Funcția **realloc** returnează un pointer la noua zonă de memorie alocată, aliniată în mod corespunzător pentru orice tip de variabile. Valoarea acestuia poate fi diferită de **ptr**, poate fi **NULL** dacă nu există suficientă memorie continuă sau dacă valoarea **size** este egală cu **0**. Dacă **realloc** eşuează, blocul original rămîne neatins – nu este nici eliberat nici mutat.

11.2. Sortare şi căutare

Nume

qsort - sortează un masiv

bsearch - căutare binară într-un masiv sortat

Declarație

```
#include <stdlib.h>
void qsort(void *base, unsigned nel,
  unsigned size, int (*comp)
  (const void *, const void *));
void *bsearch(const void *key, const void
  *base, unsigned nel, unsigned size, int
  (*comp)(const void *, const void *));
```

Descriere

Funcția **qsort** sortează un masiv de **nel** elemente, fiecare de mărime **size**. Argumentul **base** indică spre începutul masivului.

Elementele masivului sînt sortate în ordine crescătoare în concordanță cu funcția de comparare referită de **comp**, apelată cu două argumente care indică spre obiectele ce se compară. Funcția de comparare trebuie să returneze un întreg mai mic decît, egal cu, sau mai mare decît zero dacă primul argument este considerat a fi mai mic decît, egal cu, respectiv mai mare decît al doilea. Dacă cele două elemente comparate sînt egale, ordinea în masivul sortat este nedefinită.

Funcția **bsearch** caută într-un masiv de **nel** elemente, fiecare de mărime **size**, un membru care coincide cu obiectul indicat de **key**. Argumentul **base** indică spre începutul masivului.

Conținutul masivului trebuie să fie sortat crescător în concordanță cu funcția de comparare referită de **comp**, apelată cu două argumente care indică spre obiectele ce se compară. Funcția de comparare se definește ca în cazul **qsort**. Primul argument este adresa cheii, al doilea este adresa unui element din masiv.

Valoare returnată

Funcția **bsearch** returnează un pointer la un membru al masivului care coincide cu obiectul indicat de **key**, sau **NULL** dacă nu se găsește nici un membru. Dacă există mai multe elemente care coincid cu **key**, poate fi returnat oricare element cu această proprietate.

11.3. Rutine de clasificare

Nume

isalnum, **isalpha**, **isascii**, **iscntrl**, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit** - rutine de clasificare

tolower - conversie în literă mică

toupper - conversie în literă mare

Declarație

```
#include <ctype.h>
int isalnum(int c);          int isalpha(int c);
```

```
int isascii(int c);          int isprint(int c);
int iscntrl(int c);          int ispunct(int c);
int isdigit(int c);          int isspace(int c);
int isgraph(int c);          int isupper(int c);
int islower(int c);          int isxdigit(int c);

int tolower(int c);          int toupper(int c);
```

Descriere

Primele 12 funcții verifică dacă **c**, care trebuie să fie o valoare de tip **unsigned char** sau **EOF**, se află în una din clasele de caractere enumerate mai sus.

isalnum

Verifică dacă **c** este alfanumeric (literă sau cifră).

isalpha

Verifică dacă **c** este alfabetic (literă mare sau mică).

isascii

Verifică dacă **c** este o valoare pe 7 biți din setul de caractere ASCII.

iscntrl

Verifică dacă **c** este un caracter de control.

isdigit

Verifică dacă **c** este o cifră (între 0 şi 9).

isgraph

Verifică dacă **c** este un caracter afişabil cu excepția spațiului.

islower

Verifică dacă **c** este o literă mică.

isprint

Verifică dacă **c** este un caracter afişabil inclusiv spațiu.

ispunct

Verifică dacă **c** este un caracter diferit de spațiu şi non-alfanumeric.

isspace
Verifică dacă **c** este un spațiu alb.

isupper
Verifică dacă **c** este o literă mare.

isxdigit
Verifică dacă **c** este o cifră hexazecimală din setul:
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

tolower
Converteşte caracterul **c**, dacă este o literă, la litera mică corespunzătoare.

toupper
Converteşte caracterul **c**, dacă este o literă, la litera mare corespunzătoare.

Valoare returnată
Valoarea returnată de funcțiile *is*... este nenulă dacă caracterul **c** se află în clasa testată, şi zero în caz contrar.
Valoarea returnată de funcțiile *to*... este litera convertită dacă caracterul **c** este o literă, şi nedefinită în caz contrar.

11.4. Operații cu blocuri de memorie

Pentru majoritatea funcțiilor din această categorie compilatorul expandează codul acestora folosind instrucțiuni pe şiruri de caractere. Aceste funcții sînt declarate în **<string.h>**

Nume
memcpy - copiază o zonă de memorie

Declarație
```
void *memcpy(void *dest, const void *src,
     unsigned n);
void *memmove(void *dest, const void *src,
     unsigned n);
```

Descriere

Funcția **memcpy** copiază **n** octeți din zona de memorie **src** în zona de memorie **dest**. Zonele de memorie nu trebuie să se suprapună. Dacă există acest risc se utilizează **memmove**.

Valoare returnată

Funcțiile returnează un pointer la **dest**.

Nume

memcmp - compară două zone de memorie

Declarație

```
int memcmp(const void *s1, const void *s2,
    unsigned n);
```

Descriere

Funcția **memcmp** compară primii **n** octeți ai zonelor de memorie **s1** şi **s2**.

Valoare returnată

Returnează un întreg mai mic decît, egal cu, sau mai mare decît zero dacă **s1** este mai mic decît, coincide, respectiv este mai mare decît **s2**.

Nume

memset - umple o zonă de memorie cu o constantă pe un octet

Declarație

```
void *memset(void *s, int c, unsigned n);
```

Descriere

Funcția **memset** umple primii **n** octeți ai zonei de memorie indicată de **s** cu constanta **c** pe un octet.

Valoare returnată

Funcția returnează un pointer la zona de memorie **s**.

Nume

memchr - caută în memorie un caracter

Declarație

```
void *memchr(const void *s, int c, unsigned n);
```

Descriere

Funcția **memchr** caută caracterul **c** în primii **n** octeți de memorie indicați de **s**. Căutarea se opreşte la primul octet care are valoarea **c** (interpretată ca **unsigned char**).

Valoare returnată

Funcția returnează un pointer la octetul găsit sau NULL dacă valoarea nu există în zona de memorie.

11.5. Operații cu şiruri de caractere

Pentru majoritatea funcțiilor din această categorie compilatorul expandează codul acestora folosind instrucțiuni pe şiruri de caractere. Aceste funcții sînt declarate în **<string.h>**

Nume

strlen - calculează lungimea unui şir

Declarație

```
unsigned strlen(const char *s);
```

Descriere

Funcția **strlen** calculează lungimea şirului **s**, fără a include caracterul terminator null.

Valoare returnată

Funcția returnează numărul de caractere din **s**.

Nume

strcpy, **strncpy** - copiază un şir de caractere

Declarație

```
char *strcpy(char *dest, const char *src);
```

```
char *strncpy(char *dest, const char *src,
     unsigned n);
```

Descriere

Funcția **strcpy** copiază şirul indicat de **src** (inclusiv caracterul terminator null) în zona indicată de **dest**. Şirurile nu trebuie să se suprapună, şi în plus zona **dest** trebuie să fie suficient de mare pentru a primi copia.

Funcția **strncpy** este similară, cu excepția faptului că nu se copiază mai mult de **n** octeți din **src**. Astfel, dacă caracterul terminator null nu se află în primii **n** octeți din **src**, rezultatul nu va fi terminat cu null. În cazul în care lungimea lui **src** este mai mică decît **n**, restul octeților din **dest** primesc valoarea null.

Valoare returnată

Funcțiile returnează un pointer la şirul **dest**.

Nume

strdup - duplică un şir

Declarație

```
char *strdup(const char *s);
```

Descriere

Funcția **strdup** returnează un pointer la un nou şir care este un duplicat al şirului **s**. Memoria pentru noul şir se obține cu **malloc**, şi poate fi eliberată cu **free**.

Valoare returnată

Funcția returnează un pointer la şirul duplicat, sau NULL dacă nu există memorie suficientă disponibilă.

Nume

strcat, **strncat** - concatenează două şiruri

Declarație

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src,
     unsigned n);
```

Descriere

Funcția **strcat** adaugă şirul **src** la şirul **dest** suprascriind caracterul null de la sfîrşitul lui **dest**, şi la sfîrşit adaugă un caracter terminator null. Şirurile nu trebuie să se suprapună, şi în plus şirul **dest** trebuie să aibă suficient spațiu pentru a păstra rezultatul.

Funcția **strncat** este similară, cu excepția faptului că numai primele **n** caractere din **src** se adaugă la **dest**.

Valoare returnată

Funcțiile returnează un pointer la şirul rezultat **dest**.

Nume

strcmp - compară două şiruri de caractere

Declarație

```
int strcmp(const char *s1, const char *s2);
```

Descriere

Funcția **strcmp** compară cele două şiruri **s1** şi **s2**.

Valoare returnată

Funcția returnează un întreg mai mic decît, egal cu, sau mai mare decît zero dacă **s1** este mai mic decît, coincide, respectiv este mai mare decît **s2**.

Nume

strchr, **strrchr** - localizează un caracter

Declarație

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

Descriere

Funcția **strchr** returnează un pointer la prima apariție a caracterului **c** în şirul **s**.

Funcția **strrchr** returnează un pointer la ultima apariție a caracterului **c** în şirul **s**.

Valoare returnată

Funcțiile returnează un pointer la caracterul găsit sau NULL dacă valoarea nu a fost găsită.

Nume

strstr - localizează un subşir

Declarație

```
char *strstr(const char *sir, const char *subs);
```

Descriere

Funcția **strstr** găseşte prima apariție a subşirului **subs** în şirul **sir**. Caracterul terminator null nu este luat în considerare.

Valoare returnată

Funcția returnează un pointer la începutul subşirului, sau NULL dacă subşirul nu este găsit.

Nume

strspn, **strcspn** - caută un set de caractere într-un şir

Declarație

```
unsigned strspn(const char *s, const char *acc);
unsigned strcspn(const char *s, const char *rej);
```

Descriere

Funcția **strspn** determină lungimea segmentului inițial din **s** format în întregime numai cu caractere din **acc**.

Funcția **strcspn** determină lungimea segmentului inițial din **s** format în întregime numai cu caractere care nu se găsesc în **rej**.

Valori returnate

Funcția **strspn** returnează poziția primului caracter din **s** care nu se află în **acc**.

Funcția **strcspn** returnează poziția primului caracter din **s** care se află în **rej**.

11.6. Biblioteca matematică

1) Funcțiile din prima categorie sînt declarate în **<stdlib.h>**

Nume

rand, **srand** - generarea numerelor pseudo-aleatoare

Declarație

```
int rand(void);
void srand(unsigned int seed);
```

Descriere

Funcția **rand** returnează un întreg pseudo-aleator între **0** şi **RAND_MAX** (pentru majoritatea mediilor de programare C această constantă este egală cu valoarea maximă cu semn reprezentabilă pe un cuvînt al sistemului de calcul).

Funcția **srand** inițializează generatorul cu valoarea **seed** pentru o nouă secvență de valori întregi pseudo-aleatoare care vor fi returnate de **rand**. Aceste secvențe se repetă dacă **srand** se apelează cu aceeaşi valoare **seed**.

Se obişnuieşte ca generatorul să fie inițializat cu o valoare dată de ceasul sistemului de calcul, ca în exemplul de mai jos:

```
#include <time.h>
srand(time(NULL));
```

Valoare returnată

Funcția **rand** returnează o valoare între **0** şi **RAND_MAX**.

Observație

În lucrarea *Numerical Recipes in C: The Art of Scientific Computing* - William H Press, Brian P Flannery, Saul A Teukolsky, William T Vetterling / New York: Cambridge University Press, 1990 (1st ed, p 207), se face următorul comentariu:

„*Dacă doriți să generați o valoare aleatoare întreagă între 1 şi 10, se recomandă să folosiți secvența*

```
j=1+(int)(10.0*rand()/(RAND_MAX+1.0));
```

şi nu o secvență de tipul

```
j=1+(int)(1000000.0*rand())%10;
```

care foloseşte biții de rang inferior."

Tot în fişierul **<stdlib.h>** sînt descrise şi următoarele funcţii:

```
int abs(int i);          valoare absolută
long labs(long i);       valoare absolută
int atoi(char *s);       conversie din ASCII în întreg
long atol(char *s);      conversie din ASCII în întreg lung
double atof(char *s);    conversie din ASCII în dublă precizie
```

Următoarele funcţii de conversie necesită o discuţie detaliată:

```
double strtod(char *str, char **end);
```

conversie din ASCII în dublă precizie

```
long strtol(char *str, char **end, int base);
```

conversie din ASCII în întreg lung

```
unsigned long strtoul(char *str, char **end, int base);
```

conversie din ASCII în întreg lung fără semn

Şirul de caractere **str** este convertit în reprezentare binară, ca şi în cazul funcţiilor **atoi**, **atol**, **atof**. Pentru conversii de tip întreg se precizează şi baza de numeraţie folosită, care poate fi între 2 şi 36. Pentru baze mai mari decît zece se folosesc litere mici sau mari. În plus, dacă parametrul **end** indică o adresă validă (nu este null), la ieşire primeşte adresa primului caracter invalid din **str**, care nu respectă regulile de scriere a unei valori numerice de tipul aşteptat.

2) Funcţiile din a doua categorie sînt declarate în **<math.h>**

```
double fabs(double x);    valoare absolută
double floor(double x);   parte întreagă inferioară
double ceil(double x);    parte întreagă superioară
double sqrt(double x);    √x
double sin(double x);     sin(x)
double cos(double x);     cos(x)
double tan(double x);     tg(x)
double asin(double x);    arcsin(x)
double acos(double x);    arccos(x)
double atan(double x);    arctg(x)  în [-π/2,π/2]
double atan2(double y, double x);
                          arctg(y/x)  în [–π,π]
```

```
double exp(double x);
```
e^x
```
double log(double x);
```
$\ln(x)$
```
double pow(double x, double y);
```
x^y
```
double sinh(double x);
```
$\sinh(x)$
```
double cosh(double x);
```
$\cosh(x)$
```
double tanh(double x);
```
$\tanh(x)$
```
double ldexp(double x, int e);
```
$x \cdot 2^e$
```
double fmod(double x, double y);
```
x modulo y

Funcția **fmod** returnează o valoare f definită astfel: $x = a \cdot y + f$ a este o valoare întreagă (dată de x/y) şi $0 \le |f| < y$; f are semnul lui x.

Următoarele două funcții returnează două valori: una este valoarea returnată de funcție (de tip **double**), şi cealaltă returnată prin intermediul unui argument de tip pointer la **int** respectiv **double**.

```
double frexp(double x, int *e);
```
Funcția **frexp** desparte valoarea x în două părți: o parte fracționară normalizată ($f \in [0.5,1)$) şi un exponent e. Dacă x este 0 atunci f=0 şi e=0. Valoarea returnată este f.

```
double modf(double x, double *n);
```
Funcția **modf** desparte valoarea x în două părți: o parte fracționară subunitară f şi o parte întreagă n. Valorile f şi n au acelaşi semn ca şi x. Valoarea returnată este f.

11.7. Funcții pentru timp şi dată calendaristică

Funcțiile din această categorie, împreună cu tipurile şi structurile de date folosite, sînt declarate în **<time.h>**

Funcția time

Data şi ora calendaristică se reprezintă în format compact printr-o valoare de tip **time_t**. Majoritatea mediilor de programare definesc

acest tip ca un sinonim pentru **unsigned long**. Această valoare se obține prin apelul funcției **time**.

```
time_t time(time_t *tims);
```

Funcția returnează data şi ora curentă în coordonate universale (UTC), care reprezintă numărul de secunde de la începutul erei Unix (1 ianuarie 1970, ora 0:00:00).

Valoarea returnată este memorată la adresa indicată de argumentul **tims**, dacă acesta nu este NULL. Dacă argumentul **tims** este NULL, trebuie să memorăm valoarea returnată.

Structura tm (dată calendaristică defalcată)
Funcția gmtime

O valoare de tip **time_t** poate fi defalcată pe componente şi memorată într-o structură:

```
struct tm {
        int tm_sec;     /* secunde, între 0 şi 59 */
        int tm_min;     /* minute, între 0 şi 59 */
        int tm_hour;    /* ore, între 0 şi 23 */
        int tm_mday;    /* ziua din lună, între 1 şi 31 */
        int tm_mon;     /* luna, între 0 şi 11 */
        int tm_year;    /* anul curent minus 1900 */
        int tm_wday;    /* ziua din săptămînă, duminica: 0 */
        int tm_yday;    /* ziua din an, între 0 şi 365 */
        int tm_isdst;   /* opțiunea daylight saving time */
        };
```

Observație. Alegerea numelui **tm** pentru această structură a fost cît se poate de neinspirată. Programatorii trebuie să fie deosebit de atenți ca, atunci cînd folosesc funcții din această categorie, să nu folosească în nici un caz numele **tm** pentru definiții proprii.

Funcția **gmtime** efectuează această defalcare. Această funcție returnează adresa unei zone unde se află data calendaristică defalcată.

Informațiile din această zonă trebuie salvate în cazul în care se efectuează prelucrări ulterioare. Un apel ulterior al funcției **gmtime** le va suprascrie, zona fiind alocată static în interiorul acesteia.

```
struct tm *gmtime(const time_t *tims);
```

Funcția mktime

```
time_t mktime(struct tm *datc);
```

Această funcție efectuează conversia inversă, de la o structură defalcată la o valoare compactă. Este suficient să completăm valorile **tm_year**, **tm_mon**, **tm_mday**, **tm_hour**, **tm_min**, **tm_sec**. Dacă o valoare se află în afara limitelor permise, acestea sînt automat normalizate. De exemplu, ora 25:00:00 este echivalată cu ora 1:00:00 a zilei următoare. Funcția corectează (dacă e nevoie) toate valorile structurii. Dacă data calendaristică nu poate fi reprezentată ca o valoare de tip **time_t**, funcşia returnează valoarea (**time_t**)(–1).

Funcțiile ctime şi asctime

```
char *ctime(const time_t *tims);
char *asctime(const struct *datc);
```

Aceste funcții convertesc data calendaristică în reprezentare ASCII. Aceste funcții returnează adresa unui şir de forma:

```
"Sun Jun 05 10:00:00 2005"
```

Informațiile din această zonă trebuie salvate în cazul în care se efectuează prelucrări ulterioare. Un apel ulterior al funcției le va suprascrie, zona fiind alocată static în interiorul acesteia.

Funcția clock

```
clock_t clock(void);
```

Funcția **clock** returnează o aproximare a timpului pe care procesorul l-a folosit pentru program pînă la momentul apelului. Tipul **clock_t** este definit de majoritatea mediilor de programare

ca un sinonim pentru **unsigned long**. Numărul de secunde de la lansarea programului se obține astfel:

```
(double)clock()/CLOCKS_PER_SECOND
```

Constanta **CLOCKS_PER_SECOND** este specifică mediului de programare folosit. De exemplu, biblioteca GNU C Compiler pentru sistemul de operare Linux defineşte valoarea 1000000. O consecință a acestui fapt este că pe un sistem de calcul pe 32 de biți funcția **clock** va returna aceeaşi valoare la aproximativ fiecare 72 de minute.

Observație. Standardul C nu impune ca această cronometrare să înceapă de la zero la lansarea programului. De aceea, pentru a garanta un maxim de portabilitate, cronometrarea se face astfel:

```
main() {
double stm;
stm = clock();
...
printf("Time:%.3lf\n",(clock()-stm)/
                    CLOCKS_PER_SECOND);
}
```

Unele sisteme de operare, de exemplu Linux, permit cronometrarea exactă a timpului de rulare a unui program (evident, cu limitarea la 72 de minute), incluzînd aici şi timpul ocupat de sistemul de operare pentru efectuarea unor operații cerute de program. Nu este cronometrat timpul cît procesorul este ocupat cu rularea altui program.

11.7. Executarea unei comenzi sistem

Funcția **system**, declarată în **<stdlib.h>**, oferă posibilitatea de a executa o comandă sub controlul programului.

Nume

system – execută o comandă a interpretorului de comenzi

Declarație

```
system(char *com);
```

Descriere

Funcția **system** execută comanda **com** prin lansarea interpretorului de comenzi cu acest parametru.

Se recomandă folosirea cu atenție a acestei funcții sub unele sisteme de operare (de exemplu Linux), dacă interpretorul de comenzi permite precizarea mai multor comenzi într-o singură linie.

11.8. Programe demonstrative

1) Programul prezentat în continuare generează un şir de **n** valori întregi aleatoare în intervalul [0,**M**−1] pe care le depune în masivul **X** (alocat dinamic), şi apoi le sortează crescător. În continuare se generează **k** valori întregi aleatoare pe care le caută în masivul **X**. Pentru fiecare căutare cu succes se afişează pe terminal valoarea căutată şi poziția în masiv.

Valorile **n**, **k** şi **M** se iau în această ordine din linia de comandă.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int cmp(const void *A, const void *B) {
return *(int *)A-*(int *)B;
}
int main(int ac, int **av) {
int *X,*p,M,n,k,i,v;
if (ac!=4) {
   fputs("Trei argumente!\n",stderr);
   return 1;
   }
n=atoi(av[1]); k=atoi(av[2]);
M=atoi(av[3]);
X=(int *)malloc(n*sizeof(int));
if (!X) return 1;
srand(time(NULL));
for (i=0; i<n; i++)
   X[i]=rand()%M;
qsort(X,n,sizeof(int),cmp);
for (i=0; i<k; i++) {
```

```
   v=rand()%M;
   p=(int *)bsearch(&v,X,n,sizeof(int),cmp);
   if (p)
      printf("Val: %d   Pos: %d\n",v,p-X);
   }
free(X);
return 0;
}
```

2) Să reluăm al treilea exemplu din capitolul precedent. Se citeşte un fişier text care conţine pe fiecare linie un nume (şir de caractere fără spaţiu) şi trei valori reale (note). Pentru fiecare linie se calculează media aritmetică a celor trei valori şi se determină dacă elementul este admis (fiecare notă este minimum 5) sau respins (cel puţin o notă este sub 5). În final se afişează liniile în ordinea următoare: mai întîi elementele admise în ordinea descrescătoare a mediei, şi apoi elementele respinse în ordine alfabetică după nume. Se afişează doar numele, situaţia (**A/R**) şi media.

În acest exemplu punem în evidenţă o modalitate comodă de selectare a membrilor unei structuri cu ajutorul macrourilor. Macroul **Mbr** selectează din zona referită de pointerul **P** membrul **f**. Deoarece pointerul **P** (care poate fi argumentul **A** sau **B** al funcţiei **comp**) referă o zonă de tip **void**, este necesar mai întîi un *cast* pentru a preciza tipul concret al acesteia. Membrul **f** poate fi: **nm**, **ar**, **md**, definiţi în cadrul structurii de tip **StEl**.

Deoarece nu ştim de la început cîte linii are fişierul de intrare, sîntem nevoiţi să folosim următoarea strategie. La început alocăm pentru masivul **El** o zonă care să memoreze **NA** elemente. Pe măsură ce această zonă se completează, la un moment dat numărul de linii citite coincide cu **NA**. În acest moment se alocă o zonă nouă care să poată memora un număr mai mare de elemente. Desigur, de fiecare dată se va actualiza mărimea spaţiului alocat.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define NA 32
typedef struct {
```

```
   char nm[11], ar;
   float na, nb, nc, md;
   } StEl;
#define Mbr(P,f) ((StEl *)P)->f
int comp(const void *A, const void *B) {
float w;
int d;
if (d=Mbr(A,ar)-Mbr(B,ar))
   return d;
if (Mbr(A,ar)=='A') {
   w=Mbr(B,md)-Mbr(A,md);
   if (w>0) return 1;
   if (w<0) return -1;
   }
return strcmp(Mbr(A,nm),Mbr(B,nm));
}
int main(int ac, char **av) {
int na,ne,i;
StEl *El;
FILE *fi;
if (ac!=2) {
   fputs("Un argument!\n",stderr);
   return 1;
   }
fi=fopen(av[1],"rt");
if (!fi) {
   perror("Eroare la deschidere");
   return 1;
   }
na=NA;  ne=0;
El=(StEl *)malloc(na*sizeof(StEl));
while (fscanf(fi,"%s %f %f %f",El[ne].nm,
       &El[ne].na,&El[ne].nb, &El[ne].nc)
       !=EOF) {
   if ((El[ne].na>=5) && (El[ne].nb>=5) &&
       (El[ne].nc>=5)) El[ne].ar='A';
   else El[ne].ar='R';
   El[ne].md=(El[ne].na+El[ne].nb+El[ne].nc)/3.0;
   ne++;
   if (ne==na) {
```

```
      na+=NA;
      El=(StEl *)realloc(El,na*sizeof(StEl));
      }
   }
fclose(fi);
qsort(El,ne,sizeof(StEl),comp);
for (i=0; i<ne; i++)
   printf("%-12s %c%6.2lf\n",El[i].nm,
          El[i].ar,El[i].md);
free(El);
return 0;
}
```

3) Se citeşte dintr-un fişier text o valoare naturală **n**. Următoarele linii conţin **n** cuvinte, fiecare cuvînt avînd acelaşi număr de litere (cel mult 10). Să se afişeze cuvintele din fişier ordonate alfabetic.

Pentru a memora lista de cuvinte folosim următoarea strategie. În loc să alocăm pentru fiecare cuvînt (şir de caractere) citit o zonă nouă de memorie, alocăm de la început o zonă în care să putem memora toate cuvintele din listă. Această zonă va avea mărimea de **(l+1)*n** octeţi, unde **l** este lungimea fiecărui cuvînt (numărul de litere). De ce **(l+1)**: trebuie să memorăm şi caracterul terminator null.

Avantaje: memoria este utilizată mai eficient dacă se alocă de la început o zonă contiguă de dimensiune mai mare, şi se reduce foarte mult fragmentarea memoriei.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int comp(const void *A, const void *B) {
return strcmp((char *)A,(char *)B);
}
int main(int ac, char **av) {
char *C,s[11];
int n,l,i;
FILE *fi;
if (ac!=2) {
   fputs("Un argument!\n",stderr);
   return 1;
```

```
   }
fi=fopen(av[1],"rt");
if (!fi) {
   perror("Eroare la deschidere");
   return 1;
   }
fscanf(fi,"%d %s",n,s);
l=strlen(s);
C=(char *)malloc((l+1)*n);
strcpy(C,s);
for (i=1; i<n; i++)
   fscanf(fi,"%s",C+(l+1)*i);
fclose(fi);
qsort(C,n,l+1,comp);
for (i=0; i<n; i++)
   printf("%s\n",C+(l+1)*i);
free(C);
return 0;
}
```

Bibliografie (inclusiv manuale electronice)

Pentru o descriere detaliată a limbajului C recomandăm:

Brian W Kernigham, Dennis M Ritchie - *The C Programming Language*
Prentice-Hall Software Series, 1988
http://freebooks.by.ru/CandCpp.html

Herbert Schildt - *Manual C complet*
Editura Teora, 1998

Brian Brown - *C Programming*
http://www.xploiter.com/mirrors/cprogram/cstart.htm

Steve Summit - *Introductory C Programming Class Notes*
http://www.eskimo.com/~scs/cclass/cclass.html

Steve Summit - *Intermediate C Programming Class Notes*
http://www.eskimo.com/~scs/cclass/cclass.html

Cuprins