Cuprins

1. Generalități asupra limbajului C 4
1.1. Introducere 4
1.2. Primele programe 5
1.3. Meta-limbajul şi setul de caractere 9

2. Unitățile lexicale ale limbajului C 10
2.1. Identificatori 10
2.2. Cuvinte cheie 10
2.3. Constante 11
2.4. Şiruri 14
2.5. Operatori 15
2.6. Separatori 15

3. Variabile 17
3.1. Clase de memorie 17
3.2. Tipuri de variabile 20
3.3. Obiecte şi valori-stînga 23
3.4. Conversii de tip 23

4. Operatori şi expresii 27
4.1. Expresii primare 27
4.2. Operatori unari 29
4.3. Operatori multiplicativi 32
4.4. Operatori aditivi 33
4.5. Operatori de deplasare 33
4.6. Operatori relaționali 34
4.7. Operatori de egalitate 35
4.8. Operatorul ŞI pe biți 35
4.9. Operatorul SAU-exclusiv pe biți 35
4.10. Operatorul SAU-inclusiv pe biți 36
4.11. Operatorul ŞI-logic 36
4.12. Operatorul SAU-logic 37
4.13. Operatorul condițional 37
4.14. Operatori de atribuire 38

4.15. Operatorul virgulă 39
4.16. Precedența și ordinea de evaluare 40

5. Declarații 42
5.1. Specificatori de clasă de memorie 43
5.2. Specificatori de tip 44
5.3. Declaratori 45
5.4. Modificatorul `const` 47
5.5. Inițializare 48
5.6. *Nume-tip* 49

6. Instrucțiuni 51
6.1. Instrucțiunea *expresie* 51
6.2. Instrucțiunea compusă sau blocul 51
6.3. Instrucțiunea condițională 52
6.4. Instrucțiunea `while` 54
6.5. Instrucțiunea `do` 54
6.6. Instrucțiunea `for` 55
6.7. Instrucțiunea `switch` 56
6.8. Instrucțiunea `break` 58
6.9. Instrucțiunea `continue` 58
6.10. Instrucțiunea `return` 59
6.11. Instrucțiunea vidă 59

7. Funcțiile și structura unui program 60
7.1. Definiția funcțiilor 60
7.2. Apelul funcțiilor 63
7.3. Revenirea din funcții 63
7.4. Argumentele funcției și transmiterea parametrilor 64
7.5. Funcții cu număr variabil de parametri 65
7.6. Exemple de funcții și programe 66

8. Liniile de control ale compilatorului 70
8.l. Înlocuirea simbolurilor, substituții macro 70
8.2. Includerea fișierelor 72
8.3. Compilarea condiționată 72
8.4. Utilizarea directivelor de compilare 73

9. Pointeri şi masive 76
9.1. Pointeri şi adrese 76
9.2 Pointeri şi argumente de funcţii 77
9.3. Pointeri şi masive 78
9.4. Aritmetica de adrese 80
9.5. Pointeri la caracter şi funcţii 83
9.6. Masive multidimensionale 86
9.7. Masive de pointeri şi pointeri la pointeri 88
9.8. Iniţializarea masivelor şi masivelor de pointeri 93
9.9. Masive de pointeri şi masive multidimensionale 96
9.10. Argumentele unei linii de comandă 97
9.11. Pointeri la funcţii 102

10. Structuri şi reuniuni 106
10.1. Elemente de bază 106
10.2. Structuri şi funcţii 108
10.3. Masive de structuri 110
10.4. Pointeri la structuri 115
10.5. Structuri auto-referite 117
10.6. Căutare în tabele 123
10.7. Cîmpuri 127
10.8. Reuniuni 129
10.9. Declaraţii de structuri, reuniuni şi cîmpuri 131
10.10. `Typedef` 133

11. Intrări / ieşiri 136
11.1. Intrări şi ieşiri standard; fişiere 136
11.2. Accesul la fişiere; deschidere şi închidere 138
11.3. Citire şi scriere fără format 141
11.4. Citire cu format 143
11.5. Scriere cu format 147
11.6. Tratarea erorilor 152
11.7. Operaţii cu directoare 154
11.8. Programe demonstrative 156

12. Alte rutine din biblioteca standard 160
12.1. Alocarea dinamică a memoriei 160

12.2. Sortare şi căutare 161
12.3. Rutine de clasificare 162
12.4. Operaţii cu blocuri de memorie 164
12.5. Operaţii cu şiruri de caractere 166
12.6. Biblioteca matematică 170
12.7. Programe demonstrative 173

Bibliografie 178

1. Generalităţi asupra limbajului C

1.1. Introducere

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.

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 facilităţi de alocare a memoriei altele decît definiţia statică sau disciplina de stivă relativă la variabilele locale ale funcţiilor. În sfîrşit, limbajul C nu are facilităţi de intrare-ieşire şi nici metode directe de acces la fişiere. Toate aceste mecanisme de nivel înalt sînt realizate prin funcţii 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 învaţă relativ uşor, iar subtilităţile se reţin pe măsură ce experienţa în programare creşte.

1.2. Primele programe

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

Prin tradiție 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>
main() {
printf("Hello, world\n");
return 0;
}
```

Acest program afişează un mesaj de salut.

Prima linie indică faptul că se folosesc funcții 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 declară funcția `main` care va conține instrucțiunile programului. În acest caz singura instrucțiune este un apel al funcției `printf` care afişează un mesaj la terminal. Mesajul este dat între ghilimele şi se termină cu un caracter special new-line (`'\n'`).

Instrucțiunea `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.

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>
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 declară 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 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 parcurge această listă şi se determină produsul valorilor strict pozitive. În final programul afişează produsul calculat.

```
#include <stdio.h>
main() {
int n,i;
double a[100], p;
```

```
scanf("%d",&n);
for (i=0; i<n; i++)
    scanf("%lf",&a[i]);
p = 1;
for (i=0; i<n; i++)
    if (a[i]>0)
       p *= a[i];
printf("%lf\n",p);
return 0;
}
```

În cadrul funcției `main` se declară două variabile `n` şi `i` care vor memora valori întregi. Variabila `n` păstrează numărul de valori reale din lista `a`. Se declară de asemenea un tablou unidimensional `a` care va memora 100 de valori de tip real (dublă precizie), şi o variabilă `p` care va memora produsul cerut.

Se citeşte de la terminal o valoare `n`. În continuare se introduc valorile reale a_i (`i` = 0, 1, ..., `n`−1). Formatul de introducere `"%lf"` indică faptul că se aşteaptă introducerea unei valori reale 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 ++.

În continuare variabila `p`, care va memora produsul valorilor cerute, se inițializează cu `1`. Fiecare valoare a_i este verificată (instrucțiunea `if`) dacă este strict pozitivă şi în caz afirmativ este înmulțită cu valoarea `p`. Operatorul `*=` are semnificația *înmulțeşte cu*.

Al patrulea 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=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.

Al doilea set de operații necesită o analiză mai atentă; explicațiile sînt valabile pentru programe care rulează pe arhitecturi Intel. Variabila `c`, care ar trebui să memoreze valoarea 2461357 (rezultatul corect), va avea valoarea 2461356, 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 treilea set de operații. Aici rezultatele intermediare sînt memorate de fiecare dată cu trunchiere în variabile de tip `float`. În final se calculează şi diferența 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).

Rulați acest program pe diferite sisteme de calcul şi observați care este rezultatul.

1.3. Meta-limbajul şi setul de caractere

Meta-limbajul care serveşte la descrierea formală a sintaxei limbajului C este simplu.

Categoriile sintactice sau noţiunile care trebuie definite sînt urmate de simbolul '**:**'.

Definiţiile alternative de categorii sînt listate pe linii separate. Dacă o linie nu este suficientă, se trece la linia următoare, aliniată la un tab faţă de linia precedentă.

Un simbol opţional, terminal sau neterminal este indicat prin adăugarea imediată după el a configuraţiei de caractere „<opt>”.

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

2. Unităţile lexicale ale limbajului C

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

2.1. Identificatori

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. Deci alfabetul peste care sînt definiţi identificatorii este următorul:

$A = <a,...,z,A,...,Z,0,...,9,_>$

2.2. Cuvinte cheie

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. Aceste cuvinte sînt (fiecare mediu de programare C poate folosi şi alte cuvinte rezervate):

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

2.3. 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 precedată de litera `l` sau `L`.

Exemplu: `123lu`.

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 aşa numitor secvențe de evitare. Secvențele de evitare oferă de altfel şi un mecanism general pentru reprezentarea caracterelor mai greu 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

`\ddd` configurație de biți (*ddd*)

Aceste secvențe, deşi sînt formate din mai multe caractere, ele reprezintă în realitate un singur caracter. Secvența `'\ddd'` unde ddd este un şir de 1 pînă la 3 cifre octale, generează pe un octet valoarea caracterului dorit sau a configurației de biți dorite, date de şirul *ddd*.

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

Un caz special al acestei construcții este secvența `'\0'` care indică caracterul `NULL`, care este caracterul cu valoarea zero. `'\0'` este scris deseori în locul lui `0` pentru a sublinia natura de caracter a unei anumite expresii.

Cînd caracterul care urmează după un backslash nu este unul dintre cele specificate, backslash-ul este ignorat. Atragem atenția că toate caracterele setului ASCII sînt pozitive, dar o constantă caracter specificată printr-o secvență de evitare poate fi şi negativă, de exemplu `'\377'` are valoarea `-1`.

Constante simbolice

O constantă simbolică este un identificator cu valoare de constantă. Valoarea constantei poate fi orice şir de caractere introdus prin construcția `#define` (vezi capitolul 8).

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).

2.4. 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 secvenţa de evitare `\"`. În interiorul unui şir pot fi folosite şi alte secvenţe 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, situaţie 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 conţine 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` (vezi secţiunea 3.1). Un şir este iniţializat cu caracterele date (vezi secţiunea 5.4).

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. Funcţia `strlen(s)` returnează lungimea şirului de caractere `s`, excluzînd caracterul terminal null.

```
int strlen(char s[]) {
                          /* returnează lungimea şirului */
int i;
i=0;
while (s[i]!='\0')
   ++i;
```

```
return i;
}
```

Atragem atenția asupra diferenței dintre o constantă caracter şi un şir care conține 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 octeți, dintre care primul conține un caracter (litera *x*), iar al doilea caracterul `NULL` care indică sfîrşitul de şir.

2.5. Operatori

Limbajul C prezintă un număr mare de operatori care pot fi clasificați după diverse criterii. Există operatori unari, binari şi ternari, operatori aritmetici, logici, operatori pe biți etc.

Într-un capitol separat vom prezenta clasele de operatori care corespund la diferite nivele de prioritate.

2.6. Separatori

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

Separatorul cel mai frecvent este aşa numitul spațiu alb (blanc) care conține unul sau mai multe spații, tab-uri, new-line-uri sau comentarii.

Aceste construcții 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 funcții sau delimitează anumite părți în cadrul expresiilor aritmetice etc

`{ }` Acoladele – încadrează instrucțiunile compuse, care constituie corpul unor instrucțiuni sau corpul funcțiilor

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

indicii elementelor de masiv

`" "` Ghilimelele – încadrează un şir de caractere

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

`;` Punct şi virgula – termină o instrucțiune

`/*` 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 influențează cu nimic semnificația programului, scopul lui fiind doar o documentare a programului.

Nu se admit comentarii imbricate.

3. Variabile

O variabilă este un obiect de programare căruia i se atribuie un nume.

Variabilele ca şi constantele sînt elemente de bază cu care operează un program scris în C.

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

Variabilele în limbajul C sînt caracterizate prin două atribute: clasă de memorie şi tip.

Aceste două atribute îi sînt atribuite unei variabile prin intermediul unei declarații. Declarațiile listează variabilele care urmează a fi folosite, stabilesc clasa de memorie, tipul variabilelor şi eventual valorile inițiale, dar despre declarații vom discuta în capitolul 5.

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 (secțiunea 6.2) sau funcții (capitolul 7). Ele se declară prin specificatorul de clasă de memorie auto sau implicit prin context. O variabilă care apare în corpul unei funcții sau al unui bloc pentru care nu s-a făcut nici o declarație 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 reține valorile de la un apel la altul al funcției sau blocului şi trebuie inițializate la fiecare intrare. Dacă nu sînt inițializate, conțin valori

reziduale. Nici o funcție nu are acces la variabilele `auto` din altă funcție. În funcții diferite pot exista variabile locale cu aceleaşi nume, fără ca variabilele să aibă vreo legătură între ele.

Variabile externe

Variabilele externe sînt variabile cu caracter global. Ele se definesc în afara oricărei funcții şi pot fi apelate prin nume din oricare funcție care intră în alcătuirea programului.

În declarația de definiție aceste variabile nu necesită specificarea nici unei clase de memorie.

La întîlnirea unei definiții de variabilă externă compilatorul alocă şi memorie pentru această variabilă.

Într-un fişier sursă domeniul de definiție şi acțiune al unei variabile externe este de la locul de declarație pînă la sfîrşitul fişierului.

Aceste variabile există şi îşi păstrează valorile de-a lungul execuției întregului program.

Pentru ca o funcție să poată utiliza o variabilă externă, numele variabilei trebuie făcut cunoscut funcției printr-o declarație. Declarația poate fi făcută fie explicit prin utilizarea specificatorului extern, fie implicit prin context.

Dacă definiția unei variabile externe apare în fişierul sursă înaintea folosirii ei într-o funcție particulară, atunci nici o declarație ulterioară nu este necesară, dar poate fi făcută.

Dacă o variabilă externă este referită într-o funcție înainte ca ea să fie definită, sau dacă este definită într-un fişier sursă diferit de fişierul în care este folosită, atunci este obligatorie o declarație extern pentru a lega aparițiile variabilelor respective.

Dacă o variabilă externă este definită într-un fişier sursă diferit de cel în care ea este referită, atunci o singură declarație extern dată în afara oricărei funcții este suficientă pentru toate funcțiile care urmează declarației.

Funcțiile sînt considerate în general variabile externe afară de cazul cînd se specifică altfel.

Variabilele externe se folosesc adeseori în locul listelor de argumente pentru a comunica date între funcții, chiar dacă funcțiile sînt compilate separat.

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

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

Ele pot fi declarate şi implicit prin context; de exemplu şirurile de caractere care apar în interiorul unei funcții cum ar fi argumentele funcției `printf` (vezi capitolul 11) sînt variabile statice interne.

Variabilele statice externe se definesc în afara oricărei funcții şi orice funcție 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 funcții şi este static internă dacă este definită în interiorul unei funcții.

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

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 2 sau 3); declaraţia register este ignorată pentru celelalte variabile;

– numai tipurile de date `int`, `char` şi `pointer` sînt admise;

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

3.2. Tipuri de variabile

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

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.

Ordinul de mărime al variabilelor caracter este între `-128` şi `127`. Caracterele setului ASCII sînt toate pozitive, dar o constantă caracter specificată printr-o secvență 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 funcționare a instrucțiunilor calculatorului).

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.

Relații despre dimensiune sînt furnizate de calificatorii `short`, `long` şi `unsigned`, care pot fi aplicați tipului `int`.

Calificatorul `short` se referă totdeauna la numărul minim de octeți pe care poate fi reprezentat un întreg, în cazul nostru 2.

Calificatorul `long` se referă la numărul maxim de octeți pe care poate fi reprezentat un întreg, în cazul nostru 4.

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

Calificatorul `unsigned` alături de declarația de tip `int` determină ca valorile variabilelor astfel declarate să fie considerate întregi fără semn.

Numerele de tipul unsigned respectă legile aritmeticii modulo 2^n, unde n este numărul de biți din reprezentarea unei variabile de tip `int`. Numerele de tipul `unsigned` sînt totdeauna pozitive.

Declarațiile pentru calificatori sînt de forma:

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

Cuvîntul `int` poate fi omis în aceste situații.

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`.

Tipul `void` (tip neprecizat)

Un pointer poate fi declarat de tip `void`. În această situație pointerul nu poate fi folosit pentru indirectare (dereferențiere) fără un *cast* explicit, şi aceasta deoarece compilatorul nu poate determina mărimea obiectului pe care pointerul îl indică.

```
int x;
float r;
void *p = &x;          /* p indică pe x */
int main() {
*(int *) p = 2;
p = &r;                /* p indică pe r */
*(float *)p = 1.1;
}
```

Dacă o funcție este declarată de tip `void`, aceasta nu va returna o valoare:

```
void hello(char *name) {
printf("Hello, %s.",name);
}
```

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;
– „*funcții care returnează T*” pentru funcții 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 conține 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. Amănunte despre tipurile derivate sînt date în secțiunea 5.3.

3.3. Obiecte şi valori-stînga

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

Un obiect este conținutul 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 $E1 = E2$ operandul stîng $E1$ trebuie să fie o expresie *valoare-stînga*. În paragraful de descriere a operatorilor se va indica dacă operanzii sînt *valori-stînga* sau dacă rezultatul operației este o *valoare-stînga*.

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 secțiunea 7.5 care converteşte un şir de cifre în echivalentul lor numeric.

Expresia:

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

produce valoarea numerică a caracterului (cifră) memorat în ASCII.

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 `float` la `int` se face prin trunchierea părții fracționare. Conversia de la `int` la `float` 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 (deci pentru numere negative), 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 < 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`. Notăm însă că, conținutul real al lui `n` nu este alterat. Operatorul `cast` are aceeaşi precedență ca şi oricare operator unar.

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 MAXLINE 1000
char line[MAXLINE+1];
```

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. Operatorii descrişi în acelaşi paragraf au aceeaşi precedenţă. Vom specifica 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

Expresie-primară:
    identificator
    constantă
    şir
    (*expresie*)
    expresie-primară [*expresie*]
    expresie-primară (*listă-expresii*<opt>)
    valoare-stînga . *identificator*
    expresie-primară -> *identificator*
Listă-expresii:
    expresie
    listă-expresii , *expresie*

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 ...*”, atunci valoarea *expresiei-identificator* este un pointer la primul obiect al masivului, iar tipul expresiei este „*pointer la ...*”. Mai mult, un identificator de masiv nu este o expresie *valoare-stînga*.

La fel, un identificator declarat de tip „*funcție care returnează* ...”, care nu apare pe poziție de apel de funcție este convertit la „*pointer la funcție care returnează* ...”.

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`.

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. Există cîteva excepții în anumite inițializări (vezi paragraful 5.4).

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*).

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* ...”, *expresia-indice* are tipul `int`, iar rezultatul are tipul „...”. O expresie de forma *E*1`[`*E*2`]` este identică (prin definiție) cu `*(`(*E*1`)+(`*E*2`))`, unde `*` este operatorul de indirectare.

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* ...”, iar rezultatul apelului de funcție va fi de tipul „...”.

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 pentru ca tipul unui argument actual să coincidă cu cel al argumentului formal, se va folosi un cast (vezi secțiunea 3.4).

Sînt permise apeluri recursive la orice funcție.

O *valoare-stînga* urmată de un punct şi un identificator este o *expresie-primară*. *Valoarea-stînga* denumeşte o structură sau o reuniune (vezi capitolul 10) iar identificatorul denumeşte un membru din structură sau reuniune. Rezultatul este o *valoare-stînga* care se referă la membrul denumit din structură sau reuniune.

O *expresie-primară* urmată de o săgeată (constituită dintr-o liniuţă 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

Descriem în continuare operatorii limbajului C împreună cu expresiile care se pot constitui cu aceşti operatori.

4.2. Operatori unari

Toţi operatorii unari au aceeaşi precedenţă, iar expresiile unare se grupează de la dreapta la stînga.

Expresie-unară:

`*` *expresie*
`&` *valoare-stînga*
– *expresie*
`!` *expresie*
`~` *expresie*
`++` *valoare-stînga*
`--` *valoare-stînga*
valoare-stînga `++`
valoare-stînga `--`
`(`*nume-tip*`)` *expresie*
`sizeof (`*nume-tip*`)`

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 ...*” atunci tipul rezultatului este „...”. Acest operator tratează operandul său ca o adresă, face acces la ea şi îi obține conținutul.

Exemplu: instrucțiunea `y = *px;` atribuie lui `y` conținutul adresei către care indică `px`.

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 „...” atunci tipul rezultatului este „*pointer la ...*”.

Exemplu. Fie `x` o variabilă de tip `int` şi `px` un pointer creat într-un anumit fel (vezi capitolul 9). Atunci prin instrucțiunea

```
px = &x;
```

se atribuie variabilei de tip „*pointer la* `int`” `px` adresa variabilei `x`; putem spune acum că `px` indică spre `x`. Secvența:

```
px = &x;  y = *px;
```

este echivalentă cu

```
y = x;
```

Operatorul `&` poate fi aplicat numai la variabile şi la elemente de masiv. Construcții de forma `&(x+1)` şi `&3` nu sînt admise. De asemenea nu se admite ca variabila să fie de clasă `register`.

Operatorul unar `&` ajută la transmiterea argumentelor de tip adresă în funcții.

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^n, unde n este numărul de biți rezervați tipului `int`.

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.

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

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

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 construcție 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*.

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

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

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 o are în comunicarea cu rutinele de alocare a memoriei sau rutinele I/O sistem.

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

Construcția `sizeof(`*nume-tip*`)` este luată ca o unitate, astfel că expresia

```
sizeof(nume-tip)-2
```

este acelaşi lucru cu

```
(sizeof(nume-tip))-2
```

4.3. Operatori multiplicativi

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

Expresie-multiplicativă:
expresie `*` *expresie*
expresie `/` *expresie*
expresie `%` *expresie*

Operatorul binar `*` indică înmulțirea. Operatorul este asociativ, dar în expresiile în care apar mai mulți operatori de înmulțire, 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 diferențe, dar dacă totuşi se doreşte o anumită ordine, atunci se vor introduce variabile temporare.

Operatorul binar `/` indică împărțirea. 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 împărțirii primei expresii la cea de a doua. Operanzii nu pot fi de tip `float`. Restul are totdeauna semnul deîmpărțitului. Totdeauna `(a/b)*b+a%b` este egal cu `a` (dacă `b` este diferit de 0). Sînt executate conversiile aritmetice obişnuite.

4.4. Operatori aditivi

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

Expresie-aditivă:
expresie + *expresie*
expresie − *expresie*

Operatorul binar + produce suma operanzilor săi. El este asociativ şi expresiile care conțin mai mulți operatori pot fi rearanjate la fel ca în cazul operatorului de înmulțire.

Operatorul binar − produce diferența operanzilor săi.

4.5. Operatori de deplasare

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

Expresie-deplasare:
expresie << *expresie*
expresie >> *expresie*

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

Operatorul >> produce deplasarea la dreapta a operandului din stînga cu un număr de poziții 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 biți. Astfel valoarea expresiei *E*1<<*E*2 este *E*1 (interpretată ca şi configurație de biți) 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.

4.6. Operatori relaționali

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

Expresie-relațională:
expresie < *expresie*
expresie > *expresie*
expresie <= *expresie*
expresie >= *expresie*

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

Tipul rezultatului este `int`. Se execută conversiile aritmetice obişnuite. Aceşti operatori au precedența mai mică decît operatorii aritmetici, astfel că expresia `i<x-1` se consideră `i<(x-1)` aşa după cum ne aşteptam.

4.7. Operatori de egalitate

Expresie-egalitate:
expresie == *expresie*
expresie != *expresie*

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

4.8. Operatorul *ŞI* pe biţi

Expresie-ŞI:
expresie & *expresie*

Operatorul & este operatorul „*ŞI*” logic pe biţi. El este asociativ şi expresiile care conţin operatorul & pot fi rearanjate. Rezultatul este funcţia logică „*ŞI*” pe biţi aplicată operanzilor săi. Operatorul se aplică numai la operanzi de tipuri întregi. Legea după care funcţionează este:

&	0	1
0	0	0
1	0	1

Operatorul & este deseori folosit pentru a masca o anumită mulţime de biţi: de exemplu:

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

pune pe zero toţi biţii afară de ultimii 7 biţi de ordin inferior ai lui n, fără a afecta conţinutul lui n.

4.9. Operatorul *SAU-exclusiv* pe biţi

Expresie-SAU-exclusiv:
expresie ^ *expresie*

Operatorul ^ este operatorul „*SAU-exclusiv*” logic pe biţi. El este asociativ şi expresiile care-l conţin 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

4.10. Operatorul *SAU-inclusiv* pe biți

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.

4.11. Operatorul *ŞI-logic*

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`.

4.12. Operatorul *SAU-logic*

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

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

Spre deosebire de operatorul *SAU-inclusiv* pe biţi `|`, 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`.

4.13. Operatorul condiţional

Expresie-condiţională:
expresie `?` *expresie* `:` *expresie*

Operatorul condiţional `?` 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 operaţiei, această valoare se

atribuie lui `z`. Dacă prima expresie nu este adevărată atunci `z` ia valoarea lui `b`.

Expresia condiţională 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 precedenţa operatorului `?:` este mai mică, dar ele pot fi folosite pentru a face expresia condiţională mai vizibilă.

4.14. Operatori de atribuire

Există mai mulţi operatori de atribuire, care se grupează toţi 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 părţi ale unui operator de atribuire compus sînt unităţi 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 (vezi capitolul 9). Toţi operanzii din dreapta şi toţi 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.

4.15. 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.

4.16. Precedența şi ordinea de evaluare

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

Operatorii din aceeaşi linie au aceeaşi precedență; liniile sînt scrise în ordinea descrescătoare a precedenței, astfel de exemplu operatorii `*`, `/` şi `%` au toți aceeaşi precedență, care este mai mare decît aceea a operatorilor + şi −.

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

Limbajul C, ca şi multe alte limbaje, nu specifică în ce ordine sînt evaluați operanzii unui operator. De exemplu într-o instrucțiune de forma:

```
x = f() + g();
```

`f` poate fi evaluată înainte sau după evaluarea lui `g`; dacă `f` sau `g` alterează o variabilă externă de care cealaltă depinde, `x` poate depinde de ordinea de evaluare. Din nou rezultate intermediare trebuie memorate în variabile temporare pentru a asigura o secvență particulară.

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. Declarații

Declarațiile se folosesc pentru a specifica interpretarea pe care compilatorul trebuie să o dea fiecărui identificator.

Declarație:
    specificator-declarație lista-declarator<opt>;
Specificator-declarație:
    specificator-tip specificator-declarație<opt>
    specificator-clasă-memorie specificator-declarație<opt>
Specificator-tip:
    `char`
    `short`
    `int`
    `long`
    `unsigned`
    `float`
    `double`
    `void`
    specificator-structură-sau-reuniune
    `typedef`-*nume*
Specificator-clasă-memorie:
    `auto`
    `static`
    `extern`
    `register`
    `const`
    `typedef`
Lista-declarator:
    declarator-cu-inițializare
    declarator-cu-inițializare, *lista-declarator*
Declarator-cu-inițializare:
    declarator inițializator<opt>
Declarator:

identificator
(*declarator*)
* *declarator*
declarator ()
declarator [*expresie-constantă*<opt>]

Declarațiile listează toate variabilele care urmează a fi folosite într-un program. O declarație specifică tipul, clasa de memorie şi eventual valorile inițiale pentru una sau mai multe variabile de acelaşi tip. Declarațiile se fac sau în afara oricărei funcții sau la începutul unei funcții înaintea oricărei instrucțiuni.

Nu orice declarație rezervă şi memorie pentru un anumit identificator, de aceea deosebim:

– declarația de definiție a unei variabile, care se referă la locul unde este creată variabila şi unde i se alocă memorie;

– declarația de utilizare a unei variabile, care se referă la locul unde numele variabilei este declarat pentru a anunța proprietățile variabilei care urmează a fi folosită.

5.1. Specificatori de clasă de memorie

Specificatorii de clasă de memorie sînt:

```
auto    static    extern    register
const   typedef
```

Specificatorul `typedef` nu rezervă memorie şi este denumit „*specificator de clasă de memorie*” numai din motive sintactice; el va fi discutat în capitolul 10.

Semnificația diferitelor clase de memorie a fost discutată deja în paragraful 3.1.

Declarațiile cu specificatorii `auto`, `static` şi `register` determină şi rezervarea unei zone de memorie corespunzătoare. Declarația cu specificatorul `extern` presupune o definiție externă pentru identificatorii dați, undeva în afara funcției sau fişierului în care ei sînt declarați.

Într-o declarație poate să apară cel mult un specificator de clasă de memorie. Dacă specificatorul de clasă lipseşte din declarație, el se consideră implicit `auto` în interiorul unei funcții şi definiție extern în afara funcției. Excepție fac funcțiile care nu sînt niciodată automatice. De exemplu liniile:

```
int sp;
double val[MAXVAL];
```

care apar într-un program în afara oricărei funcții, definesc variabilele externe `sp` de tip `int` şi `val` de tip masiv de `double`. Ele determină alocarea memoriei şi servesc de asemenea ca declarații ale acestor variabile în tot restul fişierului sursă. Pe de altă parte liniile:

```
extern int sp;
extern double val[];
```

declară pentru restul fişierului sursă că variabilele `sp` şi `val` sînt externe, `sp` este de tip `int` şi `val` este un masiv de `double` şi că ele au fost definite în altă parte, unde li s-a alocat şi memorie. Deci aceste declarații nu creează aceste variabile şi nici nu le alocă memorie.

5.2. Specificatori de tip

Specificatorii de tip sînt:

`char` `short` `int` `long` `unsigned`
`float` `double` `void`
specificator-structură-sau-reuniune
`typedef`-*nume*

Cuvintele `long`, `short` şi `unsigned` pot fi considerate şi ca adjective; următoarele combinații sînt acceptate:

```
short int
long int
unsigned int
unsigned long int
long double
```

Într-o declarație se admite cel mult un specificator de tip, cu excepția combinațiilor amintite mai sus. Dacă specificatorul de tip lipsește din declarație, el se consideră implicit `int`.

Specificatorii de structuri şi reuniuni sînt prezentați în secțiunea 10.9, iar declarațiile cu `typedef` în secțiunea 10.10.

5.3. Declaratori

Lista-declarator care apare într-o declarație este o succesiune de declaratori separați prin virgule, fiecare dintre ei putînd avea un inițializator.

Declaratorii din *lista-declarator* sînt identificatorii care trebuie declarați.

Fiecare declarator este considerat ca o afirmație care, atunci cînd apare o construcție de aceeaşi formă cu declaratorul, produce un obiect de tipul şi de clasa de memorie indicată. Fiecare declarator conține un singur identificator. Gruparea declaratorilor este la fel ca şi la expresii.

Dacă declaratorul este un identificator simplu, atunci el are tipul indicat de specificatorul din declarație.

Un declarator între paranteze este tot un declarator, dar legătura declaratorilor complecşi poate fi alterată de paranteze.

Să considerăm acum o declarație de forma:

*T D*1

unde *T* este un specificator de tip (ca de exemplu `int`) şi *D*1 un declarator. Să presupunem că această declarație face ca identificatorul să aibă tipul „...*T*” unde „...” este vid dacă *D*1 este un identificator simplu (aşa cum tipul lui `x` în int `x` este `int`). Dacă *D*1 are forma:

`*`*D*

atunci tipul identificatorului pe care-l conține acest declarator este „*pointer la T*”.

Dacă *D*1 are forma:

D `()`

atunci identificatorul pe care-l conține are tipul „*funcție care returnează T*”.

Dacă *D*1 are forma:

D`[`*expresie-constantă*`]` sau *D*`[]`

atunci identificatorul pe care-l conține are tipul „*masiv de T*”.

În primul caz expresia constantă este o expresie a cărei valoare este determinabilă la compilare şi al cărei tip este `int`. Cînd mai mulți identificatori „*masiv de T*” sînt adiacenți, se creează un masiv multidimensional; expresiile constante care specifică marginile masivelor pot lipsi numai pentru primul membru din secvență. Această omisiune este utilă cînd masivul este extern şi definiția reală care alocă memoria este în altă parte (vezi secțiunea 5.1). Prima expresie constantă poate lipsi de asemenea cînd declaratorul este urmat de inițializare. În acest caz dimensiunea este calculată la compilare din numărul elementelor inițiale furnizate.

Un masiv poate fi construit din obiecte de unul dintre tipurile de bază, din pointeri, din reuniuni sau structuri, sau din alte masive (pentru a genera un masiv multidimensional).

Nu toate posibilitățile admise de sintaxa de mai sus sînt permise. Restricțiile sînt următoarele: funcțiile nu pot returna masive, structuri, reuniuni sau funcții, deşi ele pot returna pointeri la astfel de obiecte; nu există masive de funcții, dar pot fi masive de pointeri la funcții. De asemenea, o structură sau reuniune nu poate conține o funcție, dar ea poate conține un pointer la funcție. De exemplu, declarația

```
int i, *ip, f(), *fip(), (*pfi)();
```

declară un întreg `i`, un pointer `ip` la un întreg, o funcție `f` care returnează un întreg, o funcție `fip` care returnează un pointer la un întreg, un pointer `pfi` la o funcție care returnează un întreg. Prezintă interes compararea ultimilor doi declaratori.

Construcția `*fip()` este `*(fip())`, astfel că declarația sugerează apelul funcției `fip` şi apoi utilizînd indirectarea prin intermediul pointerului se obține un întreg.

În declaratorul `(*pfi)()`, parantezele externe sînt necesare pentru arăta că indirectarea printr-un pointer la o funcție furnizează o funcție, care este apoi apelată; ea returnează un întreg.

Declarațiile de variabile pot fi explicite sau implicite prin context. De exemplu declarațiile:

```
int a,b,c;
char d, m[100];
```

specifică un tip şi o listă de variabile. Aici clasa de memorie nu este declarată explicit, ea se deduce din context. Dacă declarația este făcută în afara oricărei funcții atunci clasa de memorie este `extern`; dacă declarația este făcută în interiorul unei funcții atunci implicit clasa de memorie este `auto`.

Variabilele pot fi distribuite în declarații în orice mod; astfel listele le mai sus pot fi scrise şi sub forma:

```
int a;
int b;
int c;
char d;
char m[100];
```

Aceasta ultimă formă ocupă mai mult spațiu dar este mai convenabilă pentru adăugarea unui comentariu pentru fiecare declarație sau pentru modificări ulterioare.

5.4. Modificatorul `const`

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

Sintaxa:

`const` *nume-variabilă* = *valoare* `;`

nume-funcție (..., `const` tip `*`*nume-variabilă*, ...)

În prima variantă, modificatorul 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.

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

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

```
*p = 35;
```

În a doua variantă modificatorul `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, ...);
```

5.5. Inițializare

Un declarator poate specifica o valoare inițială pentru identificatorul care se declară. Inițializatorul este precedat de semnul = şi constă dintr-o expresie sau o listă de valori incluse în acolade.

Inițializator:
    expresie
    { *listă-inițializare* }
Listă-inițializare:
    expresie
    listă-inițializare, *listă-inițializare*
    { *listă-inițializare* }

Toate expresiile dintr-un *inițializator* pentru variabile statice sau externe trebuie să fie expresii constante (vezi secțiunea 3.4) sau expresii care se reduc la adresa unei variabile declarate anterior, posibil offset-ul unei expresii constante. 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.

Dacă un inițializator se aplică unui „*scalar*” (un pointer sau un obiect de tip aritmetic) el constă dintr-o singură expresie, eventual în

acolade. Valoarea inițială a obiectului este luată din expresie; se efectuează aceleaşi operaţii ca în cazul atribuirii.

Pentru inițializarea masivelor şi masivelor de pointeri vezi secțiunea 9.8. Pentru inițializarea structurilor vezi secțiunea 10.3.

Dacă masivul sau structura conține sub-masive sau sub-structuri regula de inițializare se aplică recursiv la membrii masivului sau structuri.

5.6. *Nume-tip*

În cele expuse mai sus furnizarea unui *nume-tip* a fost necesar în două contexte:

– pentru a specifica conversii explicite de tip prin intermediul unui cast (vezi secțiunea 3.4);

– ca argument al lui `sizeof` (vezi secțiunea 4.2).

Un *nume-tip* este în esență o declarație pentru un obiect de acest tip, dar care omite numele obiectului.

Nume-tip:
    specificator-tip declarator-abstract
Declarator-abstract:
    vid
    `(`*declarator-abstract*`)`
    `*`*declarator-abstract*
    declarator-abstract `()`
    declarator-abstract `[`*expresie-constantă*<opt>`]`

Pentru a evita ambiguitatea, în construcția:

    `(`*declarator-abstract*`)`

declaratorul abstract se presupune a nu fi vid. Cu această restricție, este posibil să identificăm în mod unic locul într-un *declarator-abstract*, unde ar putea apărea un identificator, dacă această construcție a fost un declarator într-o declarație. Atunci tipul denumit este acelaşi ca şi tipul identificatorului ipotetic. De exemplu:

```
int
int*
```

```
int *[3]
int(*)[3]
int *( )
int(*)()
```

denumeşte respectiv tipurile `int`, „*pointer la întreg*”, „*masiv de* 3 *pointeri la întregi*”, „*pointer la un masiv de* 3 *întregi*”, „*funcție care returnează pointer la întreg*” şi „*pointer la o funcție care returnează întreg*”.

6. 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.

6.1. Instrucțiunea *expresie*

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

Format:

expresie`;`

De obicei instrucțiunile *expresie* sînt atribuiri sau apeluri de funcție; de exemplu:

```
x = 0;
printf(...);
```

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

6.2. Instrucțiunea compusă sau blocul

Instrucțiunea compusă este o grupare de declarații şi instrucțiuni închise între acolade. Ele au fost introduse cu scopul de a folosi mai multe instrucțiuni acolo unde sintaxa cere o instrucțiune. Instrucțiunea compusă sau blocul sînt echivalente sintactic cu o singură instrucțiune.

Format:

Instrucțiune-compusă:

{ *listă-declaratori*<opt> *listă-instrucțiuni*<opt> }

Listă-declaratori:

declarație

declarație listă-declaratori

Listă-instrucțiuni:
instrucțiune
instrucțiune *listă-instrucțiuni*

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

Orice inițializare pentru variabile `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 punct şi virgulă.

6.3. Instrucțiunea condițională

Sintaxa instrucțiunii condiționale admite două formate:

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

Instrucțiunea condițională 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.

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`.

Exemplu:

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

Partea `else` aparține `if`-ului din interior. Dacă nu dorim acest lucru atunci folosim acoladele pentru a forța asocierea:

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

Instrucțiunea condițională admite şi construcția else-if de forma:

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

Această secvență de `if` 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 partea

```
else instrucțiune-4
```

poate să lipsească.

Funcția `binary` din secțiunea 7.5 este un exemplu de decizie multiplă de ordinul 3.

Pot exista un număr arbitrar de construcții:

```
else if (expresie)
        instrucțiune
```

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

Întotdeauna un `else` se leagă cu ultimul `if` întîlnit.

6.4. Instrucțiunea `while`

Format:

```
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, deci 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, adică valoarea expresiei din paranteze este zero, se face un salt la instrucțiunea de după corpul instrucțiunii `while`, deci instrucțiunea `while` se termină.

6.5. Instrucțiunea `do`

Format:

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

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

6.6. Instrucțiunea `for`

Format:

```
for (expresie-1<opt>; expresie-2<opt>; expresie-3<opt>)
      instrucțiune
```

Această instrucțiune este echivalentă cu:

```
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.

6.7. 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*

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

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 `int`.

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 instrucțiunea 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 instrucțiunea 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` (vezi secțiunea 6.8) sau `return` (vezi secțiunea 6.10).

De obicei instrucțiunea care constituie corpul unui `switch` este o instrucțiune compusă. La începutul acestei instrucțiuni pot apărea şi declarații, dar inițializarea variabilelor automatice şi registru este inefectivă.

```
na = nb = nc = 0;
while (c=s[i++])
```

```
      switch (c) {
         case '0':
         case '1':
         case '2':
         case '3':
         case '4':
         case '5':
         case '6':
         case 'T':
         case '8':
         case '9':
            nc[c-'0']++;
         break;
      case ' ':
      case '\r':
      case '\t':
         nb++;
         break;
      default:
         na++;
         break;
      }
   printf("cifre: ");
   for (i=0; i<10; i++)
      printf(" %d",nc[i]);
   printf("\nspatii albe: %d, altele: %d\n",
          nb,na);
```

În acest exemplu se parcurg toate caracterele dintr-un şir, se numără cifrele, spaţiile albe şi alte caractere şi se afişează aceste numere însoţite de comentarii.

Instrucţiunea `while` este cea care asigură parcurgerea şirului pînă la sfîrşit. Pentru fiecare caracter se execută corpul instrucţiunii `while` care constă dintr-o singură instrucţiune `switch`.

Se evaluează expresia întreagă din paranteze (în cazul nostru caracterul `c`) şi se compară valoarea sa cu toate constantele-case. În

momentul cînd avem egalitate se începe execuţia de la case-ul respectiv.

Afişarea rezultatelor se face prin intermediul instrucţiunii `for` şi a funcţiei `printf` (vezi capitolul 11).

6.8. Instrucţiunea `break`

Format:

```
break;
```

Această instrucţiune determină terminarea celei mai interioare instrucţiuni `while`, `do`, `for` sau `switch` care o conţine. Controlul trece la instrucţiunea care urmează după instrucţiunea astfel terminată.

6.9. Instrucţiunea `continue`

Format:

```
continue;
```

Această instrucţiune determină trecerea controlului la porţiunea de continuare a ciclului celei mai interioare instrucţiuni `while`, `do` sau `for` care o conţine, adică la sfîrşitul ciclului şi reluarea următoarei iteraţii a ciclului. În `while` şi `do` se continuă cu testul, iar în `for` se continuă cu *expresie*-3.

Mai precis în fiecare dintre instrucţiunile:

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

dacă apare o instrucţiune `continue` aceasta este echivalentă cu un salt la eticheta `contin`. După `contin:` urmează o instrucţiune vidă (vezi secţiunea 6.11).

Porţiunea de program din exemplul următor 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 */
   }
```

6.10. 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;
```

În primul caz valoarea returnată nu este definită. În al doilea caz valoarea expresiei este returnată apelantului funcției. Dacă se cere, expresia este convertită, ca într-o atribuire, la tipul funcției în care ea apare.

6.11. Instrucțiunea vidă

Format:

```
;
```

Instrucțiunea vidă este utilă pentru a introduce o etichetă înaintea unei acolade drepte, 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]!=0; ++nc) ;
```

Această instrucțiune numără caracterele unui şir. Corpul lui `for` este vid, deoarece tot lucrul se face în partea de test şi actualizare dar sintaxa lui `for` cere un corp al instrucțiunii. Instrucțiunea vidă satisface acest lucru.

7. Funcţiile şi structura unui program

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. Funcţiile permit desfacerea programelor mari în unele 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.

Mai precis, un program C constă dintr-o secvenţă de definiţii externe. Acestea sînt definiţii de funcţii şi de date. Sintaxa definiţiilor externe este aceeaşi ca şi a tuturor declaraţiilor.

7.1. Definiţia funcţiilor

Sintaxa definiţiei unei funcţii este următoarea:

Definiţia-funcţiei:
    tip-funcţie<opt> *nume-funcţie* (*lista-parametri*) *corp-funcţie*
Lista-parametri:
    declaraţie-parametru
    declaraţie-parametru, *lista-parametri*
Corpul-funcţiei:
    instrucţiune-compusă

Dintre specificatorii de clasă de memorie numai `extern` şi `static` sînt admişi. Un declarator de funcţie este similar cu un

declarator pentru „*funcție care returnează ...*” cu excepția că el listează parametrii formali ai funcției care se defineşte.

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 altceva decît o valoare întreagă (vezi secțiunea 7.2).

Numele funcției poate fi de asemenea precedat de clasa de memorie `extern` sau `static`. 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.

Declarația unei funcții se face implicit la apariția ei. Se interpretează ca funcție orice identificator nedeclarat anterior şi urmat de o paranteză `'('`. Pentru a evita surprize neplăcute datorită neconcordanței dintre tipurile de parametri această practică nu este recomandată.

Dăm mai jos un exemplu de funcție complet definită:

```
int max(int a, int b, int c) {
int m;
m = (a>b) ? a : b;
return (m>c) ? m : c;
}
```

Această funcție determină maximul dintre 3 numere date:

– `int` este un specificator de tip care indică faptul că funcția max returnează o valoare întreagă;

– `max(int a, int b, int c)` este declarația funcției şi a parametrilor formali;

– `{ ... }` este corpul funcției.

Să ilustrăm mecanismul definirii unei funcții scriind funcția putere `power(m,n)` care ridică un întreg `m` la o putere întreagă pozitivă `n`. Astfel valoarea lui `power(2,5)` este 32. Această funcție

desigur nu este completă, deoarece calculează numai puteri pozitive de întregi mici.

Prezentăm mai jos funcția `power` şi un program principal care o apelează, pentru a putea vedea şi structura unui program.

```
power(int x, int n) {
int i, p;
p = 1;
for (i=1; i<=n; ++i)
   p = p * x;
return p;
}

main() {
int i;
for (i=0; i<10; ++i)
   printf(" %d %d\n", i, power(2,i),
                          power(-3,i));
}
```

Funcția `power` este apelată în programul principal de două ori. Fiecare apel transmite funcției două argumente.

Rezultatul este afişat de funcția de bibliotecă `printf` (vezi capitolul 11).

Numele funcției este urmat obligatoriu de paranteze, chiar dacă lista parametrilor este vidă.

În general numele funcției este ales după dorința utilizatorului. Totuşi în fiecare program trebuie să existe o funcție cu numele impus `main`; orice program îşi începe execuția cu funcția `main`. Celelalte funcții sînt apelate din interiorul funcției `main`. Unele dintre funcțiile apelate sînt definite în acelaşi program, altele sînt conținute într-o bibliotecă de funcții.

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

O funcție poate fi declarată şi `static`. În acest caz ea poate fi apelată numai din fişierul unde a fost definită.

Funcțiile în C pot fi folosite 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.

7.2. Apelul funcțiilor

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 '(', adică funcția nu apare pe poziția de apel de funcție atunci nu se realizează imediat apelul funcției respective ci se generează un pointer la funcție (vezi secțiunea 9.11).

7.3. 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:

```
return;
return expresie;
```

Ca argument poate apărea orice expresie admisă în C. Funcția returnează valoarea acestei expresii funcției apelante.

O funcție nu returnează în mod obligatoriu o valoare. O instrucțiune `return`, fără expresie ca parametru, cauzează numai transferul controlului funcției apelante nu şi o valoare utilă.

La rîndul său funcția apelantă poate ignora valoarea returnată.

De obicei o funcție returnează o valoare de tip întreg. Dacă se dorește ca funcția să returneze un alt tip, atunci numele tipului trebuie să preceadă numele funcției, iar programul trebuie să conțină o declarație a acestei funcții atît în fişierul în care funcția este definită cît şi în fişierul unde funcția este apelată.

Pentru a evita orice confuzie se recomandă ca tipul valorii returnate de funcție să fie întotdeauna precizat, iar dacă dorim în mod expres ca funcția să nu returneze o valoare să folosim tipul `void`.

De exemplu, funcția `atof(s)` din biblioteca asociată compilatorului converteşte şirul `s` de cifre în valoarea sa în dublă precizie. Vom declara funcția sub forma:

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

sau împreună cu alte variabile de tip `double`:

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

Funcțiile nu pot returna masive, structuri, reuniuni sau funcții.

Dacă o funcție returnează o valoare de tip `char`, nu este nevoie de nici o declarație de tip din cauza conversiilor implicite. Totdeauna tipul `char` este convertit la `int` în expresii.

7.4. Argumentele funcției şi transmiterea parametrilor

O metodă de a comunica datele între funcții este prin argumente. Parantezele mici care urmează după numele funcției închid lista argumentelor.

În limbajul C argumentele funcțiilor sînt transmise prin valoare. Aceasta înseamnă că în C funcția apelată primeşte valorile argumentelor sale într-o copie particulară de variabile temporare (în realitate pe stivă).

Funcția apelată nu poate altera decît variabilele sale particulare, adică copiile temporare.

Apelul prin valoare este o posibilitate, dar nu şi o obligativitate. Apelul prin valoare conduce la programe mai compacte cu mai puține 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 funcției putere care face uz de acest fapt:

```
power(int x, int n) {
int p;
```

```
for (p=1; n>0; --n)
   p = p * x;
return p;
}
```

Argumentul `n` este utilizat ca o variabilă temporară 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 a variabilelor declarate externe.

În cazul pointerilor, funcţia apelantă trebuie să furnizeze adresa variabilei de modificat (tehnic printr-un pointer la această variabilă), iar funcţia apelată trebuie să declare argumentul corespunzător ca fiind un pointer. Referirea la variabila de modificat se face prin adresare indirectă (vezi capitolul 9).

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.

7.5. Funcţii cu număr variabil de parametri

Pentru funcţiile cu număr variabil de parametri standardul ANSI defineşte o construcţie specială `...` , numită elipsă. Acesta este de exemplu cazul funcţiilor de citire şi scriere cu format (familiile *...scanf* şi *...printf* - a se vedea capitolul 11). În acest caz, parametrii ficşi sînt verificaţi la compilare, iar cei variabili sînt transmişi ca şi cînd nu ar exista prototip de funcţie.

Pentru simplitatea expunerii am păstrat convenţiile definite de standardele mai vechi ale limbajului, păstrate şi de standardul ANSI C. O funcţie poate fi declarată fără nici un parametru, compilatorul deducînd de aici că funcţia respectivă poate avea oricîţi parametri de orice tip. Nu se recomandă totuşi o asemenea practică deoarece este

posibil să se creeze confuzii care conduc la erori în execuția programelor, erori care se corectează foarte greu în cazul programelor mari.

7.6. Exemple de funcții şi programe

1. Acest exemplu este un program de căutare şi imprimare a tuturor liniilor dintr-un text care conțin o anumită configurație de caractere, de exemplu cuvîntul englezesc `"the"`. Structura de bază a acestui program este:

```
while (mai există linii sursă)
    if (linia conține configurația de caractere)
        imprimă linia
```

În scopul scrierii acestui program vom scrie trei funcții. Funcția `getline` citeşte o linie din textul sursă şi returnează lungimea ei. Această funcție are două argumente. Argumentul `s` indică numele masivului unde se va citi linia, iar argumentul `lim` reprezintă lungimea maximă a unei linii care poate fi citită.

Funcția `index` caută configurația de caractere dorită în interiorul liniei şi furnizează poziția sau indexul în şirul `s`, unde începe şirul `t`, sau −1 dacă linia nu conține şirul `t`. Argumentele funcției sînt `s` care reprezintă adresa şirului de studiat şi `t` care reprezintă configurația căutată.

Funcția care realizează afişarea este `printf` din bibliotecă.

Programul care realizează structura de mai sus este:

```
#define MAXLINE 1000
#define EOF -1

getline(char s[], int lim) {
         /* citeşte o linie în s; returnează lungimea */
int c, i;
i = 0;
while (--lim>0 && (c=getchar())!= EOF &&
       c!='\n')
```

```
      s[i++]=c;
if (c=='\n') s[i++]=c;
s[i]='\0';
return i;
}

index(char s[], char t[]) {
   /* returnează poziția 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;
}

main() {
        /* imprimă toate liniile care conţin cuvîntul „the” */
char line [MAXLINE];
while (getline(line, MAXLINE)>0)
   if (index(line,"the")>=0)
      printf("%s",line);
}
```

2. Funcția `atof` converteşte un şir de cifre din formatul ASCII într-un număr flotant în precizie dublă.

```
double atof(char s[]) {
                    /* converteşte şirul s în dublă precizie */
double val, power;
int i, sign;
for (i=0; s[i]==' ' || s[i]=='\n' ||
   s[i]=='\t'; i++) ;    /* sare peste spații albe */
```

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

3. Funcția `atoi` convertește un șir de cifre în echivalentul lor numeric întreg.

```
atoi(char s[]) {    /* convertește șirul s la un întreg */
int i,n;
n = 0;
for (i=0; s[i]>='0' && s[i]<='9'; i++)
   n = 10 * n +s[i] - '0';
return n;
}
```

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

```
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.

```
binary(int x, int v[], int n) {
                  /* caută x în v0, v1, ..., vn-1 */
int low, high, mid;
low = 0;
high = n - 1;
while (low<=high) {
   mid = (low + high) / 2;
   if (x < v[mid])
      high = mid - 1;
   else if (x > v[mid])
            low = mid + 1;
         else     /* s-a găsit o intrare */
            return(mid);
   }
return -1;
}
```

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

Exemplul ilustrează o decizie triplă, dacă `x` este mai mic, mai mare sau egal cu elementul din mijlocul șirului `v[mid]`.

8. Liniile de control ale compilatorului

Limbajul C oferă o facilitate de extensie a limbajului cu ajutorul unui preprocesor de macro-operații simple. Folosirea liniilor de forma:

```
#define
#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.

8.l. Înlocuirea simbolurilor; substituții macro

O definiție de forma:

`#define` *identificator şir-simboluri*

determină ca preprocesorul să înlocuiască toate aparițiile ulterioare ale identificatorului cu *şir-simboluri* dat (*şir-simboluri* nu se termină cu '`;`' deoarece în urma substituției identificatorului cu acest şir ar apărea prea multe caractere '`;`').

Şir-simboluri sau textul de înlocuire este arbitrar. În mod normal el este tot restul liniei ce urmează după identificator. O definiție 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 definiție are efect numai pînă la definiția următoare. Substituțiile nu se realizează în cazul în care identificatorii sînt încadrați între ghilimele. De exemplu fie definiția:

```
#define ALFA 1
```

Oriunde va apărea în programul sursă identificatorul ALFA el va fi înlocuit cu constanta 1, cu excepția unei situații de forma:

```
printf("ALFA");
```
în care se va tipări 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 TABSIZE 100
int tab[TABSIZE];
```
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 '`(`' 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 priorității mai mari a operatorului * față de cea a operatorului +.

8.2. Includerea fişierelor

O linie de forma:

`#include "`*nume-fişier*`"`

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 locuri standard, cum ar fi biblioteca I/O standard asociată compilatorului. Alternativ, o linie de forma:

`#include <`*nume-fişier*`>`

caută fişierul nume-fişier numai în biblioteca standard şi nu în directorul fişierului sursă.

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

Facilitatea de includere a unor fişiere într-un text sursă este deosebit de utilă pentru gruparea declarațiilor unui program mare. Ea va asigura faptul că toate fişierele sursă vor primi aceleaşi definiții şi declarații 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.

8.3. Compilarea condiționată

O linie de control a compilatorului de forma:

`#if` *expresie-constantă*

verifică dacă expresia constantă este evaluată la o valoare diferită de zero.

O linie de control de forma:

`#ifdef` *identificator*

verifică dacă identificatorul a fost subiectul unei linii de control de forma `#define`.

O linie de control de forma:

```
#ifndef identificator
```

verifică dacă identificatorul este nedefinit în preprocesor.

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.

8.4. Utilizarea dirctivelor de compilare

Prezentăm în continuare un exemplu didactic de utilizare a directivelor de compilare în dezvoltarea unor proiecte.

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 fițier implementează cele două funcții.

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

```
#ifndef _Numere_H
#define _Numere_H
unsigned eprim(unsigned n);
unsigned cmmdc(unsigned p, unsigned q);
#endif
```

Fişierul sursă `princ.c` este foarte scurt.

```
#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 prezentat în continuare.

```
#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 fişiere 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.

Proiectele – programe de dimensiuni mari – sînt compuse de cele mai multe ori din module care se elaborează şi se pun la punct separat. Uneori pot fi identificate funcții care prezintă un interes general mai mare, şi care pot fi utilizate în elaborarea unor tipuri de aplicații foarte diverse. Acestea sînt dezvoltate separat şi, după ce au fost puse la punct în totalitate, se depun într-o bibliotecă de funcții 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 descrise aceste funcții, şi acesta se păstrează în fişiere header.

9. Pointeri şi masive

Un pointer este o variabilă care conține adresa unei alte variabile. Pointerii sînt foarte mult utilizați 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 obținut prin alte căi.

9.1. Pointeri şi adrese

Deoarece un pointer conține 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 această variabilă. Atunci aplicînd operatorul unar `&` lui `x`, instrucțiunea:

```
px = &x;
```

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

Invers, dacă `px` conține adresa variabilei `x`, atunci instrucțiunea:

```
y = *px;
```

atribuie variabilei `y` conținutul locației pe care o indică `px`.

Evident toate variabilele care sînt implicate în aceste instrucțiuni trebuie declarate. Aceste declarații sînt:

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

Declarațiile variabilelor `x` şi `y` sînt deja cunoscute. Declarația pointerului `px` este o noutate. A doua declarație indică faptul că o combinație 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, ca de exemplu în expresia următoare:

```
y = *px + 1;
```
unde variabilei y i se atribuie conținutul variabilei x plus 1.

Instrucțiunea:
```
d = sqrt((double)*px);
```
are ca efect convertirea conținutului variabilei x pe care o indică px în tip double şi apoi depunerea rădăcinii pătrate a valorii astfel convertite în variabila d.

Referiri la pointeri pot apărea de asemenea şi în partea stîngă a atribuirilor. Dacă, de exemplu, px indică spre x, atunci:
```
*px = 0;
```
atribuie variabilei x valoarea zero, iar:
```
*px += 1;
```
incrementează conținutul variabilei x cu 1, ca şi în expresia:
```
(*px)++;
```
În acest ultim exemplu parantezele sînt obligatorii deoarece, în lipsa lor, expresia ar incrementa pe px în loc de conținutul variabilei pe care o indică (operatorii unari *, ++ au aceeaşi precedență şi sînt evaluați de la dreapta spre stînga).

9.2 Pointeri şi argumente de funcții

Deoarece în limbajul C transmiterea argumentelor la funcții se face „prin valoare” (şi nu prin referință), funcția apelată nu are posibilitatea de a altera o variabilă din funcția apelantă. Problema care se pune este cum procedăm dacă totuşi dorim să schimbăm un argument?

De exemplu, o rutină de sortare poate schimba între ele două elemente care nu respectă ordinea dorită, cu ajutorul unei funcții swap. Fie funcția swap definită astfel:

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

Funcția `swap` apelată prin `swap(a,b)` nu va realiza acțiunea dorită deoarece ea nu poate afecta argumentele `a` şi `b` din rutina apelantă.

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

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

iar forma corectă a lui `swap` este:

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

9.3. Pointeri şi masive

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

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

defineşte un masiv de dimensiune 10, care reprezintă un bloc de 10 obiecte consecutive numite `a[0]`, ... `a[9]`. Notația `a[i]` reprezintă al `i`-lea element al masivului sau elementul din poziția `i+1`, începînd cu primul element. Dacă `pa` este un pointer la un întreg declarat sub forma:

```
int *pa;
```

atunci atribuirea:

```
pa = &a[0];
```

încarcă variabila `pa` cu adresa primului element al masivului `a`.

Atribuirea:

```
x = *pa;
```

copiază conținutul lui `a[0]` în `x`.

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

Aceste observații sînt adevărate indiferent de tipul variabilelor din masivul a.

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

Corespondența 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.

Atribuirea:

```
pa = &a[0];
```

este identică cu:

```
pa = a;
```

De asemenea, expresiile a[i] şi *(a+i) sînt identice. Aplicînd operatorul & la ambele părți obținem &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 instrucțiune.

Există însă o singură diferență între un nume de masiv şi un pointer la începutul masivului. Un pointer este o variabilă, deci pa = a şi pa++ sînt instrucțiuni corecte. Dar un nume de masiv este o constantă şi deci construcții de forma a = pa, a++ sau p = &a sînt ilegale.

Cînd se transmite unei funcții un nume de masiv, ceea ce se transmite de fapt este adresa primului element al masivului. Aşadar, un nume de masiv, argument al unei funcții, este în realitate un

pointer, adică o variabilă care conține o adresă. Fie de exemplu funcția `strlen` care calculează lungimea şirului `s`:

```
strlen(char *s) {       /* returnează lungimea şirului */
int n;
for (n=0; *s!='\0'; s++)
   n++;
return n;
}
```

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

Este posibil să se transmită unei funcții, 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[]) { }
```

sau

```
f(int *arr) { }
```

Declarațiile `int arr[]` şi `int *arr` sînt echivalente, opțiunea pentru una din aceste forme depinzînd de modul în care vor fi scrise expresiile în interiorul funcției.

9.4. Aritmetica de adrese

Dacă `p` este un pointer, atunci `p += i` incrementează pe `p` pentru a indica cu `i` elemente după elementul pe care îl indică în prealabil `p`. Această construcție şi altele similare sînt cele mai simple şi comune formule ale aritmeticii de adrese, care constituie o caracteristică puternică a limbajului C. Să ilustrăm cîteva din

proprietățile aritmeticii de adrese scriind un alocator rudimentar de memorie. Fie rutina `alloc(n)` care returnează un pointer `p` la o zonă de `n` caractere consecutive care vor fi folosite de rutina apelantă pentru memorarea unui şir de caractere. Fie rutina `free(p)` care eliberează o zonă începînd cu adresa indicată de pointerul `p` pentru a putea fi refolosită mai tîrziu. Zona de memorie folosită de rutinele `alloc` şi `free` este o stivă funcționînd pe principiul ultimul intrat – primul ieşit, iar apelul la `free` trebuie făcut în ordine inversă cu apelul la `alloc`. Să considerăm că funcția `alloc` va gestiona stiva ca pe elementele unui masiv pe care îl vom numi `allocbuf`. Vom mai folosi un pointer la următorul element liber din masiv, pe care-l vom numi `allocp`. Cînd se apelează rutina `alloc` pentru `n` caractere, se verifică dacă există suficient spațiu liber în masivul `allocbuf`. Dacă da, `alloc` va returna valoarea curentă a lui `allocp`, adică adresa de început a blocului cerut, după care va incrementa pe `allocp` cu n pentru a indica următoarea zonă liberă. `free(p)` actualizează `allocp` cu valoarea `p`, dacă `p` indică în interiorul lui `allocbuf`.

```
#define NULL 0
        /* valoarea pointerului pentru semnalizarea erorii */
#define ALLOCSIZE 1000
        /* dimensiunea spațiului disponibil */
static char allocbuf [ALLOCSIZE];
        /* memoria pentru alloc */
static char *allocp = allocbuf;
        /* următoarea poziție liberă */

char *alloc(int n) {
        /* returnează pointer la n caractere */
if (allocp+n<=allocbuf+ALLOCSIZE) {
   allocp += n;      /* dimensiunea satisfăcută */
   return allocp-n;    /* vechea valoare */
   }
else
   return NULL;      /* nu este spațiu suficient */
```

```
}

free(char *p) {  /* eliberează memoria indicată de p */
if (p>=allocbuf && p<allocbuf+ALLOCSIZE)
   allocp = p;
}
```

Testul `if (allocp+n<=allocbuf+ALLOCSIZE)` verifică dacă există spațiu suficient pentru satisfacerea cererii de alocare a `n` caractere. Dacă cererea poate fi satisfăcută, alloc revine cu un pointer la zona de `n` caractere consecutive. Dacă nu, alloc trebuie să semnaleze lipsa de spațiu pe care o face returnînd valoarea constantei simbolice `NULL`. Limbajul C garantează că nici un pointer care indică corect o dată nu va conține zero, prin urmare o revenire cu valoarea zero poate fi folosită pentru semnalarea unui eveniment anormal (în cazul nostru, lipsa de spațiu). Atribuirea valorii zero unui pointer este deci un caz special.

Observăm de asemenea că variabilele `allocbuf` şi `allocp` sînt declarate static cu scopul ca ele să fie locale numai fişierului sursă care conține funcțiile `alloc` şi `free`.

Exemplul de mai sus demonstrează cîteva din facilitățile aritmeticii de adrese (pointeri). În primul rînd, pointerii pot fi comparați în anumite situații. Dacă `p` şi `q` sînt pointeri la membri unui acelaşi masiv, atunci relațiile `<`, `<=`, `>`, `>=`, `==`, `!=` sînt valide. Relația `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 aparținînd la masive diferite.

Se observă că pointerii şi întregii pot fi adunați sau scăzuți. Construcția 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 declarația lui p (scara de aliniere este 1 pentru char, 2 pentru 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 o altă versiune a funcției strlen folosind această ultimă observație:

```
strlen(char *s) {   /* returnează lungimea unui şir */
char *p;
p = s;
while (*p != '\0')
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. Diferența p-s dintre adresa ultimului element al şirului şi adresa primului element al şirului indică numărul de elemente.

În afară de operațiile binare menționate (adunarea sau scăderea pointerilor cu întregi şi scăderea sau compararea a doi pointeri), celelalte operații 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.

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

9.5. Pointeri la caracter şi funcții

O constantă şir, de exemplu:

```
"Buna dimineata"
```

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:

```
printf("Buna dimineata\n");
```

funcția `printf` 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(s,t)` care copiază şirul `t` peste şirul `s`. O primă versiune a programului ar fi următoarea:

```
strcpy(char s[], char t[]) {/* copiază t peste s */
int t;
i = 0;
while ((s[i]=t[i]) != '\0')
   i++;
}
```

O a doua versiune cu ajutorul pointerilor este următoarea:

```
strcpy(char *s, char *t) {
                              /* versiune cu pointeri */
while ((*s++=*t++) != '\0') ;
}
```

Această versiune cu pointeri modifică prin incrementare pe `s` şi `t` în partea de test. Valoarea lui `*t++` este caracterul indicat de pointerul `t`, înainte de incrementare. Notația postfix `++` asigură că `t` va fi modificat după depunerea conținutului indicat de el, la vechea poziție a lui `s`, după care şi `s` se incrementează. Efectul este că se copiază caracterele şirului `t` în şirul `s` pînă la caracterul terminal `'\0'` inclusiv.

Am mai putea face o observație legată de redundanța comparării cu caracterul '\0', redundanță care rezultă din structura instrucțiunii while.

Şi atunci forma cea mai prescurtată a funcției strcpy(s,t) este:

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

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

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

```
strcmp(char s, char t) {/* compară şirurile s şi t */
int i;
i = 0;
while (s[i]==t[i])
if (s[i++]=='\0')
   return 0;
return s[i]-t[i];
}
```

Versiunea cu pointeri a aceleiaşi funcții este:

```
strcmp(char *s, char *t) {
for (; *s==*t; s++,t++)
if (*s=='\0')
   return 0;
return *s-*t;
}
```

În final prezentăm funcția strsav care copiază un şir dat prin argumentul ei într-o zonă obținută printr-un apel la funcția alloc. Ea returnează un pointer la şirul copiat sau NULL, dacă nu mai există suficient spațiu pentru memorarea şirului.

```
char *strsav(char *s) {  /* copiază şirul s */
char *p;
p = alloc(strlen(s)+1);
if (p!=NULL)
   strcpy(p,s);
return p;
}
```

9.6. Masive multidimensionale

Limbajul C oferă facilitatea utilizării masivelor multidimensionale, deşi în practică ele sînt folosite mai puţin decît masivele de pointeri.

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 `day_of_year` converteşte ziua şi luna în ziua anului şi funcţia `month_day` converteşte ziua anului 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 day_tab[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 `day_tab` trebuie să fie declarat extern pentru a putea fi folosit de ambele funcţii.

În limbajul C, prin definiție, 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 cele mai multe limbaje. Un masiv bidimensional poate fi tratat în acelaşi fel ca şi în celelalte limbaje, în sensul că elementele sînt memorate pe linie, adică indicele cel mai din dreapta variază cel mai rapid.

Un masiv se inițializează cu ajutorul unei liste de inițializatori închişi între acolade; fiecare linie a unui masiv bidimensional se inițializează cu ajutorul unei subliste de inițializatori. În cazul exemplului nostru, masivul `day_tab` 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 funcțiile care realizează conversiile cerute de exemplul nostru sînt:

```
day_of_year (int year, int month, int day)
{                          /* ziua anului din lună şi zi */
int i, leap;
leap = (year%4==0) && (year%100!=0) ||
       (year%400==0);
for (i=1; i<month; i++)
   day += day_tab[leap][i];
return day;
}
```

Deoarece variabila `leap` poate lua ca valori numai zero sau unu după cum expresia:

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

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

```
month_day(int year, int yearday,
          int *pmonth, int *pday) {
int i,leap;
leap = (year%4==0) && (year%100!=0) ||
```

```
        (year%400==0);
for (i=1; yearday>day_tab[leap][i]; i++)
   yearday -= day_tab[leap][i];
*pmonth = i;
*pday = yearday;
}
```

Deoarece această ultimă funcție returnează două valori, argumentele lună şi zi vor fi pointeri.

Exemplu: `month_day(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 funcții, declarația argumentelor funcției 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 `day_tab` trebuie transmis unei funcții `f`, atunci declarația lui `f` poate fi:

```
f(int (*day_tab)[13])
```

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

În general deci, un masiv *d*-dimensional `a[i][j]...[p]` de rangul `i*j*...*p` este un masiv *d*–1 - dimensional de rangul `j*k*...*p` ale cărui elemente, fiecare, sînt masive *d*–2 - dimensionale de rang `k*...*p` ale cărui elemente, fiecare, sînt masive *d*–3 - dimensionale ş.a.m.d. Oricare dintre expresiile `a[i]`, `a[i][j]...`, `a[i][j]... [p]` 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.

9.7. 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.

Dacă memorăm liniile textului una după alta într-un masiv lung de caractere (gestionat de funcția `alloc`), atunci fiecare linie poate fi accesibilă cu ajutorul unui pointer la primul ei caracter. Pointerii tuturor liniilor, la rîndul lor, pot fi memorați sub forma unui masiv. Atunci 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.

Procesul de sortare îl vom realiza în trei paşi:
1) se citesc toate liniile textului de la intrare;
2) se sortează liniile în ordine lexicografică;
3) se tipăresc liniile sortate în noua ordine.

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

```
#define LINES 100       /* nr maxim de linii de sortat */
main() {              /* sortează liniile de la intrare */
char *lineptr[LINES]; /* pointeri la linii */
int nlines;           /* nr linii intrare citite */
if ((nlines=readlines(lineptr,LINES))>=0)
   {
   sort(lineptr,nlines);
   writelines(lineptr,nlines);
   }
else printf
     ("Intrarea prea mare pentru sort\n");
}
```

Cele 3 funcții 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 imprimare. 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:

```
#define MAXLEN 1000
#define NULL 0
#define EOF -1
readlines(char *lineptr[], int maxlines) {
                                    /* citeşte liniile */
int len,nlines;
char *p,*alloc(),line[MAXLEN];
nlines = 0;
while ((len=getline(line,MAXLEN))>0)
   if (nlines>=maxlines)
      return -1;
   else if ((p=alloc(len))==NULL)
           return -1;
        else {
           line[len-1] = '\0';
           strcpy(p,line);
           lineptr[nlines++] = p;
           }
return nlines;
}
```

Instrucţiunea `line[len-1] = '\0';` şterge caracterul < LF> de la sfîrşitul fiecărei linii ca să nu afecteze ordinea în care sînt sortate liniile şi depune în locul lui caracterul `'\0'` ca marcă de sfîrşit de şir.

Rutina care tipăreşte liniile în noua lor ordine este `writelines` şi are următorul cod:

```
writelines(char *lineptr[], int nlines) {
                                   /* scrie liniile sortate */
int i;
for (i=0; i<nlines; i++)
   printf("%s\n",lineptr[i]);
}
```

Declarația nouă care apare în aceste programe este:

```
   char *lineptr[LINES];
```

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

Deoarece `lineptr` este el însuşi un masiv, care se transmite ca argument funcției `writelines`, el va fi tratat ca un pointer (vezi secțiunea 9.3) şi atunci funcția `writelines` mai poate fi scrisă şi astfel:

```
writelines(char *lineptr[], int nlines) {
while (--nlines>=0)
   printf("%s\n",*lineptr++);
}
```

În funcția `printf`, `lineptr` indică inițial prima linie de imprimat; fiecare incrementare avansează pe `*lineptr` la următoarea linie de imprimat, în timp ce `nlines` se micşorează după fiecare tipărire a unei linii cu 1.

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

```
#define NULL 0
#define LINES 100       /* nr maxim de linii de sortat */
sort(char *v[], int n) {
   /* sortează şirurile v0, v1, ... vn-1 în ordine crescătoare */
int gap,i,j;
```

```
char *temp;
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;
         temp = v[j];
         v[j] = v[j+gap];
         v[j+gap] = temp;
         }
}
```

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

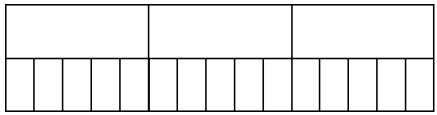
Să reţinem deci următoarele lucruri legate de masive şi pointeri. De cîte ori apare într-o expresie un identificator de tip masiv el este convertit într-un pointer la primul element al masivului. Prin definiţie, operatorul de indexare `[]` este interpretat astfel încît *E*1`[`*E*2`]` este identic cu `*(`(*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.

9.8. 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 neiniţializate se iniţializează cu zero. Nu se admite iniţializarea unui masiv de clasă cu automatic.

Acoladele `{` şi `}` se pot omite în următoarele situaţii:

– dacă iniţializatorul începe cu o acoladă stîngă (`{`), atunci lista de iniţializatori, separaţi prin virgulă, va iniţializa elementele masivului; nu se acceptă să existe mai mulţi iniţializatori decît numărul elementelor masivului;

– dacă însă inițializatorul nu începe cu acoladă stîngă ({), atunci se iau din lista de inițializatori atîția inițializatori cîți corespund numărului de elemente ale masivului, restul inițializatorilor vor iniția liza 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, în ciuda faptului că 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 `month_name` 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 *month_name(int n) {
                      /* returnează numele lunii a n-a */
static char *name[] = {
   "luna ilegala", "ianuarie",
   "februarie",  "martie", "aprilie",
   "mai", "iunie", "iulie", "august",
   "septembrie", "octombrie", "noiembrie",
   "decembrie"
   }
return ((n<1) || (n>12)) ? name[0] :
                           name[n] ;
}
```

În acest exemplu, `name` 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 `name[i]` va conține un pointer la şirul de

caractere avînd indice `i` al inițializatorului. Dimensiunea masivului `name` nu este necesar a fi specificată deoarece compilatorul o calculează numărînd inițializatorii furnizați şi o completează în declarația masivului.

9.9. 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];
int *b[10];
```

În această declarație `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 pointate 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 de exemplu poate să nu indice nici un masiv.

Cu toate că problema prezentată în acest paragraf am descris-o în termenii întregilor, ea este cel mai frecvent utilizată în memorarea şirurilor de caractere de lungimi diferite (ca în funcţia `month_name` prezentată mai sus).

9.10. 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. Primul argument (numit convenţional `argc`) reprezintă numărul de argumente din linia de comandă care a lansat programul. Al doilea argument (`argv`) 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 imprime la terminal argumentele lui luate din linia de comandă, imprimarea făcîndu-se pe o linie, iar argumentele imprimate să fie separate prin spaţii.

Comanda:

```
pri succes colegi
```

va avea ca rezultat imprimarea la terminal a textului

succes colegi

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

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

Atunci programul `pri` are următorul cod:

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

Deoarece `argv` este un pointer la un masiv de pointeri, există mai multe posibilităţi de a scrie acest program. Să mai scriem două versiuni ale acestui program.

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

Deoarece `argv` este un pointer la un masiv de pointeri, incrementîndu-l, `(++argv)`, el va pointa la `argv[1]` în loc de `argv[0]`. Fiecare incrementare succesivă poziţionează pe `argv` la următorul argument, iar `*argv` este pointerul la argumentul şirului respectiv. În acelaşi timp `argc` este decrementat pînă devine zero, moment în care nu mai sînt argumente de imprimat.

Alternativ:

```
main(int argc, char *argv[ ]) {
                                 /* versiunea a treia */
while (--argc>0)
   printf((argc>1)? "%s ":"%s\n",*++argv);
}
```

Această versiune arată că argumentul funcției `printf` poate fi o expresie ca oricare alta, cu toate că acest mod de utilizare nu este foarte frecvent.

Ca un al doilea exemplu, să reconsiderăm programul din secțiunea 7.5, care imprimă fiecare linie a unui text care conține un şir specificat de caractere (schemă).

Dorim acum ca această schemă să poată fi modificată dinamic, de la execuție la execuție. Pentru aceasta o specificăm printr-un argument în linia de comandă.

Şi atunci programul care caută schema dată de primul argument al liniei de comandă este:

```
#define MAXLINE 1000
main(int argc, char *argv[ ]) {
                    /* găseşte schema din primul argument */
char line[MAXLINE];
if (argc!=2)
   printf("Linia de comanda eronata\n");
else
   while (getline(line,MAXLINE)>0)
      if (index(line,argv[1])>=0)
         printf("%s",line);
}
```

unde linia de comandă este de exemplu: `"find limbaj"` în care `"find"` este numele programului, iar `"limbaj"` este schema căutată. Rezultatul va fi imprimarea tuturor liniilor textului de intrare care conțin 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 opționale: unul care să tipărească toate liniile cu excepția acelora care conțin schema, şi al doilea care să preceadă fiecare linie tipărită cu numărul ei de linie.

O convenție pentru programele scrise în limbajul C este ca argumentele dintr-o linie de comandă care încep cu un semn '`-`' să introducă un parametru opțional. Dacă alegem, de exemplu, `-x`

pentru a indica „cu excepția” 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 tipărirea liniei a doua, precedată de numărul ei, deoarece această linie nu conține schema `"la"`.

Argumentele opționale sînt permise în orice ordine în linia de comandă. Analizarea şi prelucrarea argumentelor unei linii de comandă trebuie efectuată în funcția principală `main`, inițializînd în mod corespunzător anumite variabile. Celelalte funcții ale programului nu vor mai ține evidența acestor argumente.

Este mai comod pentru utilizator dacă argumentele opționale sînt concatenate, ca în comanda:

```
find -xn la
```

Caracterele `'x'` respectiv `'n'` indică doar absența sau prezența acestor opțiuni (switch) şi nu sînt tratate din punct de vedere al valorii lor.

Fie programul care caută schema `"la"` în liniile de la intrare şi le tipărește pe acelea, care nu conțin schema, precedate de numărul lor de linie. Programul tratează corect, atît prima formă a liniei de comandă cît şi a doua.

```
#define MAXLINE 1000

main(int argc, char *argv[]) {
                                    /* caută schema */
char line[MAXLINE], *s;
long line0;
int except, number;
line0 = 0;
```

```
number = 0;
while (--argc>0 && (*++argv)[0]=='-')
   for (s=argv[0]+1; *s!='\0'; s++)
      switch(*s) {
         case 'x': except = 1; break;
         case 'n': number = 1; break;
         default:
          printf
             ("find: optiune ilegala %c\n",
             *s);
          argc = 0;
          break;
         }
if (argc!=1)
   printf
     ("Nu exista argumente sau schema\n");
else
   while (getline(line,MAXLINE)>0) {
      line0++;
      if ((index(line,*argv)>=0)!=except)
         {
         if (number)
            printf("%d:",line0);
         printf("%s",line);
         }
      }
}
```

Dacă nu există erori în linia de comandă, atunci la sfîrşitul primului ciclu while `argc` trebuie să fie 1, iar `*argv` conţine adresa schemei. `*++argv` este un pointer la un şir argument, iar `(*++argv)[0]` este primul caracter al şirului. În această ultimă expresie parantezele sînt necesare deoarece fără ele expresia înseamnă `*++(argv[0])` ceea ce este cu totul altceva (şi greşit): al doilea caracter din numele programului. O alternativă corectă pentru `(*++argv[0])` este `**++argv`.

9.11. 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)();
}
```

Funcția `f` trebuie declarată explicit în rutina apelantă (`int f();`), deoarece apariția ei în `g(f)` nu a fost urmată de paranteză stîngă '`(`'. În expresia `g(f) f` nu apare pe poziția de 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();
```

înseamnă 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 faptul că `funcpt` este un pointer la o funcție, `*funcpt` este funcția, iar `(*funcpt)()` este apelul funcției.

O formă echivalentă simplificată de apel este următoarea:

```
funcpt();
```

Ca un exemplu, să considerăm procedura de sortare a liniilor de la intrare, descrisă în secţiunea 9.7, dar modificată în sensul ca dacă argumentul opţional `-n` apare în linia de comandă, atunci liniile se vor sorta nu lexicografic ci numeric, liniile conţinînd grupe de numere.

O sortare constă adesea din trei părţi: o comparare care determină ordinea oricărei perechi de elemente, un schimb care inversează 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 operaţiile de comparare şi inversare, astfel încît transmiţînd diferite funcţii de comparare şi inversare funcţiei de sortare, elementele de intrare se pot aranja după diferite criterii.

Compararea lexicografică a două linii se realizează prin funcţiile `strcmp` şi `swap`. 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 funcţii în funcţia principală `main`, iar pointerii la aceste funcţii îi transmitem ca argumente funcţiei `sort`, care la rîndul ei va apela aceste funcţii prin intermediul pointerilor respectivi.

Funcţia principală `main` va avea atunci următorul cod:

```
#define LINES 100       /* nr maxim de linii de sortat */

main (int argc, char *argv[]) {
char *lineptr[LINES];     /* pointeri la linii text */
int nlines;               /* număr de linii citite */
int strcmp(), numcmp();   /* funcţii de comparare */
int swap ();              /* funcţia de inversare */
int numeric;
numeric = 0;              /* 1 dacă sort numeric */
if (argc>1 && argv[1][0]=='-' &&
    argv[1][1]=='n')
   numeric = 1;
if ((nlines=readlines(lineptr,LINES))>=0)
```

```
    {
    if (numeric)
       sort(lineptr,nlines,numcmp,swap);
    else
       sort(lineptr,nlines,strcmp,swap);
    writelines (lineptr,nlines);
    }
else
 printf
    ("Nr de linii de intrare prea mare\n");
}
```

În apelul funcției `sort`, argumentele `strcmp`, `numcmp` şi `swap` sînt adresele funcțiilor respective. Deoarece ele au fost declarate funcții care returnează un întreg, operatorul '`&`' nu este necesar să preceadă numele funcțiilor, compilatorul fiind cel care gestionează transmiterea adreselor funcțiilor.

Funcția `sort` care aranjează liniile în ordinea crescătoare se va modifica astfel:

```
sort(char *v[], int n, int (*comp)(),
     int (*exch)()) { /* sortează v0, v1, ... , vn-1 */
int gap,i,j;
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;
         exch(v+j,v+j+gap);
         }
}
```

Să studiem declarațiile din această funcție.

```
   int(*comp)(), (*exch)();
```

indică faptul că `comp` şi `exch` sînt pointeri la funcții care returnează un întreg (primul set de paranteze este necesar).

```
      if (comp(v[j],v[j+gap])<=0)
```

înseamnă apelul funcției `comp` (adică `strcmp` sau `numcmp`), deoarece `comp` este un pointer la funcție, `*comp` este funcția, iar

```
comp(v[j],v[j+gap])
```

este apelul funcției.

```
exch(v+j,v+j+gap)
```

este apelul funcției `swap`, de inversare a două linii, inversare care realizează interschimbarea adreselor liniilor implicate (vezi secțiunea 9.2). Funcția `numcmp` este următoarea:

```
numcmp(char *s1, char *s2) {
                          /* compară s1 şi s2 numeric */
double atof(),v1,v2;
v1 = atof(s1);
v2 = atof(s2);
if (v1<v2)
   return -1;
else
   if (v1>v2)
      return 1;
   else
      return 0;
}
```

Pentru ca programul nostru să fie complet să mai prezentăm şi codul funcției `swap`, care schimbă între ei pointerii a două linii.

```
swap(char *px[], char *py[]) {
char *temp;
temp = *px;
*px = *py;
*py = temp;
}
```

10. Structuri şi reuniuni

O structură este o colecție 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 (în alte limbaje, structurile se numesc articole).

Structurile ajută la organizarea datelor complicate, în special în programele mari, deoarece permit unui grup de variabile legate să fie tratate ca o singură entitate. Vom ilustra în acest capitol modul de utilizare a structurilor.

10.1. Elemente de bază

Să revenim asupra rutinei de conversie a datei prezentată în capitolul 9.

O dată constă din zi, lună şi an, eventual numărul zilei din an şi numele lunii. Aceste cinci variabile pot fi grupate într-o singură structură astfel:

```
struct date {
int day;
int month;
int year;
int yearday;
char mon_name[4];
};
```

Cuvîntul cheie `struct` introduce o declarație de structură care este o listă de declarații închise în acolade.

Cuvîntul `struct` poate fi urmat opțional de un nume, numit marcaj de structură sau etichetă de structură, cum este în exemplul nostru numele `date`. Acest marcaj 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.

Acolada dreaptă care încheie o listă de membri ai unei structuri poate fi urmată de o listă de variabile, la fel ca şi în cazul tipurilor de bază. De exemplu:

```
struct {...} x,y,z;
```

este din punct de vedere sintactic analog cu:

```
int x,y,z;
```

în sensul că fiecare declarație declară pe `x`, `y` şi `z` ca variabile de tipul numit (structură în primul caz şi întreg în al doilea) şi cauzează alocarea de spațiu pentru ele.

O declarație de structură care nu este urmată de o listă de variabile nu alocă memorie; ea descrie numai un şablon, o formă de structură. Dacă structura este marcată sau etichetată, atunci marcajul ei poate fi folosit mai tîrziu pentru definirea unor alte variabile de tip structură, cu acelaşi şablon ca structura marcată. De exemplu, fiind dată declarația:

```
struct date d;
```

ea definește variabila `d`, ca o structură de acelaşi fel (şablon) ca structura date.

O structură externă sau statică poate fi inițializată, ataşînd după definiția ei o listă de inițializatori pentru componente, de exemplu:

```
struct date d = {4,7,1984,185,"iulie"};
```

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:

```
leap = (d.year%4==0) && (d.year%100!=0)
       || (d.year%400==0);
```

sau verificarea numelui lunii:

```
if (strcmp(d.mon_name,"august")==0) ...
```

Structurile pot fi imbricate; o înregistrare de stat de plată, de exemplu, poate fi de următoarea formă:

```
struct person {
   char name[NAMESIZE];
   char address[ADRSIZE];
   long zipcode;
   long ss_number;
   double salary;
   struct date birthdate;
   struct date hiredate;
   };
```

Structura person conține două structuri de şablon date. Declarația:

```
struct person emp;
```

defineşte şi alocă o structură cu numele `emp` de acelaşi şablon ca şi `person`. Atunci:

```
emp.birthdate.month
```

se referă la luna de naştere. Operatorul de membru de structură '`.`' este asociativ de la stînga la dreapta.

10.2. Structuri şi funcții

Există un număr de restricții asupra structurilor în limbajul C. Singurele operații care se pot aplica unei structuri sînt accesul la un membru al structurii şi considerarea adresei ei cu ajutorul operatorului `&`. Acest lucru implică faptul că structurile nu pot fi atribuite sau copiate ca entități şi că ele nu pot fi transmise ca argumente la funcții şi nici returnate din funcții. Structurile de clasă automatic, ca şi masivele de aceeaşi clasă, nu pot fi inițializate; pot fi inițializate numai structurile externe şi statice, regulile de inițializare fiind aceleaşi ca pentru masive.

Pointerii la structuri nu se supun însă acestor restricții, motiv pentru care structurile şi funcțiile pot coexista şi conlucra prin intermediul pointerilor.

Ca un exemplu, să rescriem programul de conversie a datei, care calculează ziua anului, din lună şi zi.

```
day_of_year(struct date *pd) {
                                    /* calculul zilei anului */
int i, day, leap;
day = pd->day;
leap = (pd->year%4==0) &&
       (pd->year%100!==0) ||
       (pd->year%400==0);
for (i=1; i<pd->month; i++)
   day += day_tab[leap][i];
return day;
}
```

Declarația:

```
   struct date * pd;
```

indică faptul că `pd` este un pointer la o structură de şablonul lui date.

Notația:

```
   pd->year
```

indică faptul că se referă membrul "`year`" 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 year poate fi de asemenea referit prin:

```
   (*pd).year
```

Notația "`->`" se impune ca un mod convenabil de prescurtare. În notația `(*pd).year`, parantezele sînt necesare deoarece precedența operatorului membru de structură '`.`' este mai mare decît cea a operatorului `*`.

Ambii operatori '`.`' şi '`->`' sînt asociativi de la stînga la dreapta, astfel încît:

```
   p->q->membru
   emp.birthdate.month
```

sînt de fapt:

```
(p->q)->membru
(emp.birthdate).month
```

Operatorii '->' şi '.' ai structurilor, împreună cu `()` pentru listele de argumente şi `[]` pentru indexare se găsesc în vîrful listei de precedență (vezi secțiunea 4.16), fiind din acest punct de vedere foarte apropiați. Astfel, fiind dată declarația:

```
struct {
int x;
int *y;} *p;
```

unde `p` este un pointer la o structură, atunci expresia:

```
++p->x
```

incrementează pe `x`, nu pointerul `p`, deoarece operatorul '->' are o precedență mai mare decît '++'. Parantezele pot fi folosite pentru a modifica ordinea operatorilor dată de precedența. Astfel:

```
(++p)->x
```

incrementează mai întîi pe `p` şi apoi accesează elementul `x`, din structura nou pointată.

În expresia `(p++)->x` se accesează mai întîi `x`, apoi se incrementează pointerul `p`.

În mod analog, `*p->y` indică conținutul adresei pe care o indică `y`. Expresia `*p->y++` accesează mai întîi ceea ce indică `y` şi apoi incrementează pe `y`. Expresia `(*p->y)++` incrementează ceea ce indică `y`. Expresia `*p++->y` accesează ceea ce indică `y` şi apoi incrementează pointerul `p`.

10.3. 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 numelor cuvintelor cheie şi un masiv de întregi pentru a memora numărul aparițiilor. O posibilitate este de a folosi două masive paralele `keyword` şi `keycount` declarate prin:

```
char *keyword[NKEYS];
```

```
int keycount[NKEYS];
```

respectiv unul de pointeri la şiruri de caractere şi celălalt de întregi. Fiecărui cuvînt cheie îi corespunde perechea:

```
char *keyword;
int keycount;
```

astfel încît putem considera cele două masive ca fiind un masiv de perechi. Atunci, declaraţia de structură:

```
struct key {
   char *keyword;
   int keycount;
   } keytab[NKEYS];
```

defineşte un masiv `keytab` de structuri de acest tip şi alocă memorie pentru ele. Fiecare element al masivului `keytab` este o structură de acelaşi şablon ca şi structura `key`.

Definiţia masivului `keytab` poate fi scrisă şi sub forma:

```
struct key {
   char *keyword;
   int keycount;
   };
struct key keytab[NKEYS];
```

Deoarece masivul de structuri `keytab` conţine, în cazul nostru, o mulţime constantă de cuvinte cheie, este mai uşor de iniţializat o dată pentru totdeauna chiar în locul unde este definit. Iniţializarea structurilor este o operaţie analoagă cu iniţializarea unui masiv în sensul că definiţia este urmată de o listă de iniţializatori închişi în acolade.

Atunci iniţializarea masivului de structuri `keytab` va fi următoarea:

```
struct key {
   char * keyword;
   int keycount;
   } keytab[] = {
      "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:

```
    {"break",0},{"case",0}....
```

dar parantezele interioare nu sînt necesare dacă inițializatorii sînt variabile simple sau şiruri de caractere şi dacă toți inițializatorii sînt prezenți.

Compilatorul va calcula, pe baza inițializatorilor, dimensiunea masivului de structuri `keytab` motiv pentru care, la inițializare, nu este necesară indicarea dimensiunii masivului.

Programul de numărare a cuvintelor cheie începe cu definirea masivului de structuri `keytab`. Rutina principală `main` citeşte textul de la intrare prin apel repetat la o funcție `getword`, care extrage din intrare cîte un cuvînt la un apel. Fiecare cuvînt este apoi căutat în tabelul `keytab` cu ajutorul unei funcții de căutare `binary`, descrisă în secțiunea 7.5. Lista cuvintelor cheie trebuie să fie în ordine crescătoare pentru ca funcția `binary` să lucreze corect. Dacă cuvîntul cercetat este un cuvînt cheie atunci funcția `binary` returnează numărul de ordine al cuvîntului în tabelul cuvintelor cheie, altfel returnează –1.

```
#define MAXWORD 20

binary(char *word, struct key tab[],
       int n) {
int low,high,mid,cond;
low = 0;
high = n - 1;
while (low<=high) {
   mid =(low + high) / 2;
   if (
      (cond=strcmp(word,tab[mid].keyword))
       <0)
```

```
      high = mid - 1;
   else
      if (cond>0) low = mid + 1;
      else
         return mid;
   }
return -1;
}

main() {                    /* numără cuvintele cheie */
int n,t;
char word[MAXWORD];
while ((t=getword(word,MAXWORD))!=EOF)
   if (t==LETTER)
      if (
         (n=binary(word,keytab,NKEYS))
          >=0)
         keytab[n].keycount++;
for (n=0; n<NKEYS; n++)
   if (keytab[n].keycount>0)
      printf("%4d %s\n",
             keytab[n].keycount,
             keytab[n].keyword);
}
```

Înainte de a scrie funcția `getword` este suficient să spunem că ea returnează constanta simbolică `LETTER` de fiecare dată cînd găseşte un cuvînt în textul de intrare şi copiază cuvîntul în primul ei argument.

Cantitatea `NKEYS` este numărul cuvintelor cheie din `keytab` (dimensiunea masivului de structuri). Deşi putem calcula acest număr manual, este mai simplu şi mai sigur s-o facem cu calculatorul, mai ales dacă lista cuvintelor cheie este supusă modificărilor. O posibilitate de a calcula `NKEYS` cu calculatorul este de a termina lista inițializatorilor cu un pointer `NULL` şi apoi prin ciclare pe `keytab` să detectăm sfîrşitul lui. Acest lucru este mai mult decît 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` descris în secțiunea 4.2 furnizează dimensiunea în octeți a argumentului său.

În cazul nostru, numărul cuvintelor cheie este dimensiunea masivului `keytab` 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(keytab) /
               sizeof(struct key))
```

Să revenim acum la funcția `getword`. Programul pe care-l vom da pentru această funcție este mai general decît este necesar în aplicația noastră, dar nu este mult mai complicat.

Funcția `getword` citeşte cuvîntul următor din textul de intrare, printr-un cuvînt înțelegîndu-se fie un şir de litere şi cifre, cu primul caracter literă, fie un singur caracter. Funcția returnează constanta simbolică `LETTER` dacă a găsit un cuvînt, `EOF` dacă a detectat sfîrşitul fişierului sau caracterul însuşi, dacă el nu a fost alfabetic.

```
getword(char *w, int lim) {/* citeşte un cuvînt */
int t;
while (--lim>0) {
   t = type(*w++=getchar());
   if (t==EOF)
      return EOF;
   if (t!=LETTER && t!=DIGIT)
      break;
   }
*(w-1) = '\0';
return LETTER;
}
```

Funcția `getword` apelează funcția `type` pentru identificarea tipului fiecărui caracter citit la intrare cu ajutorul funcției `getchar`. Versiunea funcției `type` pentru caractere ASCII este:

```
type(int c) {     /* returnează tipul caracterului */
```

```
if (c>='a' && c<='z' || c>='A' && c<='Z')
   return LETTER;
if (c>='0' && c<='9')
   return DIGIT;
return c;
}
```

Constantele simbolice `LETTER` şi `DIGIT` pot avea orice valoare care nu vine în contradicție cu caracterele nealfabetice şi EOF. Valori posibile pot fi:

```
#define LETTER 'a'
#define DIGIT '0'
```

10.4. 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 `keytab` nu necesită modificări, în timp ce funcțiile `main` şi `binary` da. Prezentăm, în continuare, aceste funcții modificate.

```
#define MAXWORD 20

struct key *binary(char *word, struct key
                   tab[], int n) {
                                /* caută cuvînt */
int cond;
struct key * low;
struct key *high;
struct key *mid;
low = &tab[0];
high = &tab[n-1];
while (low<=high) {
   mid = low + (high-low) / 2;
   if ((cond=strcmp(word,mid->keyword))<0)
      high = mid - 1;
   else if (cond>0)
```

```
            low = mid + 1;
         else return mid;
   }
return NULL;
}

main() {/* numără cuvintele cheie, versiune cu pointeri */
int t;
char word[MAXWORD];
struct key *binary(), *p;
while ((t=getword(word.MAXWORD))!=EOF)
   if (t==LETTER)
      if ((p=binary(word,keytab,NKEYS))
           != NULL)
         p->keycount++;
for (p=keytab; p<keytab+NKEYS; p++)
   if (p->keycount>0)
      printf("%4d %s\n",p->keycount,
      p->keyword);
}
```

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 `key`, în loc de un întreg. Acest lucru este declarat atît în funcția principală `main` cît şi în funcția `binary`. Dacă `binary` găseşte un cuvînt în structura `key`, atunci returnează un pointer la el; dacă nu-l găseşte, returnează `NULL`. În funcție de aceste două valori returnate, funcția `main` semnalează găsirea cuvîntului prin incrementarea cîmpului `keycount` corespunzător cuvîntului sau citeşte următorul cuvînt.

În al doilea rînd, toate operațiile de acces la elementele masivului de structuri `keytab` se fac prin intermediul pointerilor. Acest lucru determină o modificare semnificativă în funcția `binary`. Calculul elementului mijlociu nu se mai poate face simplu prin:

```
mid = (low + high) / 2
```

deoarece adunarea a doi pointeri este o operație ilegală, nedefinită. Această instrucțiune trebuie modificată în:

```
mid = low + (high-low) / 2
```

care face ca `mid` să pointeze elementul de la jumătatea distanței dintre `low` şi `high`.

Să mai observăm inițializarea pointerilor `low` şi `high`, care este perfect legală, deoarece este posibilă inițializarea unui pointer cu o adresă a unui element deja definit.

În funcția `main` avem următorul ciclu:

```
for(p=keytab; p<keytab+NKEYS; p++)...
```

Dacă `p` este un pointer la un masiv de structuri, orice operație asupra lui `p` ține cont de dimensiunea unei structuri, astfel încît `p++` incrementează pointerul `p` la următoarea structură din masiv, adunînd la `p` dimensiunea corespunzătoare a unei structuri. Acest lucru nu înseamnă că dimensiunea structurii este egală cu suma dimensiunilor membrilor ei deoarece din cerințe de aliniere a unor membri se pot genera „goluri" într-o structură.

În sfîrşit, cînd o funcție returnează un tip complicat şi are o listă complicată de argumente, ca în:

```
struct key *binary(char *word, struct key
                   tab, int n)
```

funcția poate fi mai greu vizibilă şi detectabilă cu un editor de texte. Din acest motiv, se poate opta şi pentru următoarea formă:

```
struct key *binary(word,tab,n)
char *word; struct key tab; int n;
```

unde înainte de acolada de deschidere se precizează tipul fiecărui parametru.

Alegeți forma care vă convine şi care vi se pare mai sugestivă.

10.5. Structuri auto-referite

Să considerăm problema mai generală, a numărului aparițiilor tuturor cuvintelor dintr-un text de intrare. Deoarece lista cuvintelor nu este cunoscută dinainte, n-o putem sorta folosind algoritmul de căutare binară, ca în paragraful precedent. Nu putem utiliza nici o

căutare liniară pentru fiecare cuvînt, pe măsura apariţiei lui pentru a vedea dacă a mai fost prezent sau nu, pentru că timpul de execuţie al programelor 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 mulţimea de cuvinte, tot timpul sortată, plasînd fiecare nou cuvînt din intrare pe o poziţie 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 conţine următoarea informaţie:
- un pointer la cuvînt;
- un contor pentru numărul de apariţii;
- un pointer la descendentul stîng al cuvîntului;
- un pointer la descendentul drept al cuvîntului. Nici un nod al arborelui nu va avea mai mult decît doi descendenţi dar poate avea un descendent sau chiar nici unul.

Arborele se construieşte astfel încît pentru orice nod, sub-arborele stîng al său conţine numai cuvintele care sînt mai mici decît cuvîntul din nod, iar sub-arborele drept conţine numai cuvinte, 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 apariţiilor 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 direcţia cerută, noul cuvînt nu există în arbore şi va fi inserat pe poziţia descendentului corespunzător. Se observă că acest proces de căutare este recursiv, deoarece căutarea din fiecare nod utilizează o căutare într-unul dintre descendenţii săi.

Prin urmare se impune de la sine ca rutinele de inserare în arbore şi de imprimare să fie recursive.

Revenind la descrierea unui nod, el apare ca fiind o structură cu patru componente:

```
struct tnode {              /* nodul de bază */
char *word;                 /* pointer la cuvînt */
int count;                /* numărător de apariţii */
struct tnode *left;         /* descendent stîng */
struct tnode *right;        /* descendent drept */
};
```

Această declaraţie „recursivă" a unui nod este perfect legală, deoarece o structură nu poate conţine ca şi componentă o intrare a ei însăşi dar poate conţine un pointer la o structură de acelaşi şablon cu ea.

Declaraţia:

```
   struct tnode *left;
```

declară pe `left` ca fiind un pointer la structură (nod) şi nu o structură însăşi.

În program vom folosi rutinele `getword`, pentru citirea unui cuvînt din intrare, `alloc` pentru rezervarea de spaţiu necesar memorării unui cuvînt şi alte cîteva rutine pe care le cunoaştem deja. Rutina principală `main` citeşte prin intermediul rutinei getword un cuvînt, şi îl plasează în arbore prin rutina `tree`.

```
#define MAXWORD 20

main() {    /* contorizare apariţii cuvinte */
struct tnode *root, *tree();
char word[MAXWORD];
int t;
root = NULL;
while ((t=getword(word,MAXWORD))!=EOF)
   if (t==LETTER)
      root = tree(root,word);
treeprint(root);
}
```

Rutina `main` 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 `tree`. În acest proces, cuvîntul fie există deja, undeva în arbore, caz în care contorul lui de numărare a apariţiilor 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 `tree` returnează un pointer la el, care apoi este introdus în nodul de origine (adică în nodul al cărui descendent este noul nod) în cîmpul `left` sau `right` după cum noul cuvînt este mai mic sau mai mare faţă de cuvîntul origine. Rutina `tree`, care returnează un pointer la o structură de şablon `tnode` are următorul cod:

```
struct tnode *tree(struct tnode *p,
       char *w) {   /* introduce cuvîntul w în nodul p */
struct tnode *talloc(int n);
char *strsav(char *s);
int cond;
if (p==NULL) {      /* a sosit un nou cuvînt */
   p = talloc();    /* creează un nod nou */
   p->word = strsav(w);
   p->count = 1;
   p->left = p->right = NULL;
   }
else
   if ((cond=strcmp(w,p->word))==0)
      p->count++;
   else
      if (cond<0)   /* noul cuvînt mai mic */
         p->left = tree(p->left,w);
      else          /* noul cuvînt mai mare */
         p->right = tree(p->right,w);
         return p;
```

```
}
```

Memoria pentru noul nod se alocă de către rutina `talloc`, care este o adaptare a rutinei `alloc`, pe care am văzut-o deja. Ea returnează un pointer la un spațiu liber, în care se poate înscrie noul nod al arborelui. Vom discuta rutina `talloc` mai tîrziu. Noul cuvînt se copiază în acest spațiu cu ajutorul rutinei `strsav`, care returnează un pointer la începutul cuvîntului, contorul de apariții se inițializează la 1 şi pointerii către cei doi descendenți se fac `NULL`. Această parte de cod se execută numai cînd se adaugă un nou nod.

Rutina `treeprint` tipăreşte arborele astfel încît pentru fiecare nod se imprimă 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. Rutina `treeprint` este una din cele mai tipice rutine recursive.

```
treeprint(struct tnode *p) {
                        /* tipăreşte arborele p recursiv */
if (p!=NULL) {
   treeprint(p->left);
   printf("%5d %s\n",p->count,p->word);
   treeprint(p->right);
   }
}
```

Este important de reținut 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 `tree`, din cauza recursivității, se parcurge acelaşi drum, de data aceasta de la nodul găsit spre rădăcina arborelui, refăcîndu-se toți pointerii drumului parcurs.

Dacă considerați ca nu ați înțeles suficient de bine recursivitatea, desenați-vă un arbore şi imprimați-l cu ajutorul rutinei `treeprint`, avînd grijă să memorați fiecare ieşire din `tree` şi `treeprint`.

O observație legată de acest exemplu: dacă arborele este „nebalansat“, adică cuvintele nu sosesc în ordine aleatoare din punct

de vedere lexicografic, atunci timpul de execuție 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.

Să ne oprim puțin asupra alocării de memorie. Cu toate că se alocă diferite tipuri de obiecte, este de preferat să existe un singur alocator de memorie într-un program. Relativ la acest alocator de memorie se pun doua probleme: în primul rînd cum poate satisface el condițiile de aliniere ale obiectelor de un anumit tip (de exemplu întregii trebuie alocați la adrese pare); în al doilea rînd cum se poate declara că alocatorul returnează pointeri la tipuri diferite de obiecte.

Cerințele de aliniere pot fi în general rezolvate cu uşurință pe seama unui spațiu care se pierde, dar care este nesemnificativ ca dimensiune. De exemplu, alocatorul `alloc` returnează totdeauna un pointer la o adresă pară. În cazul în care cererea de alocare poate fi satisfăcută şi de o adresă impară (pentru şiruri de caractere, de exemplu) se pierde un caracter.

În ceea ce priveşte declararea tipului alocatorului `alloc` (adică a tipului de obiect pe care îl indică pointerul returnat de `alloc`), un foarte bun procedeu în limbajul C este de a declara că funcția `alloc` returnează un pointer la char şi apoi să convertim explicit acest pointer la tipul dorit printr-un cast. Astfel dacă `p` este declarat în forma:

```
char *p;
```

atunci:

```
(struct tnode *)p;
```

converteşte pe `p` dintr-un pointer la `char` într-un pointer la o structură de şablon `tnode`, dacă el apare într-o expresie. Şi atunci, o versiune a alocatorului `talloc` poate fi următoarea:

```
struct tnode *talloc() {
char *alloc();
return (struct tnode *)  alloc
        (sizeof(struct tnode));
}
```

10.6. Căutare în tabele

O altă problemă legată de definirea şi utilizarea structurilor este căutarea în tabele. Cînd se întîlneşte de exemplu, o linie de forma:

```
#define YES 1
```

simbolul `YES` şi textul de substituţie 1 se memorează într-o tabelă. Mai tîrziu, ori de cîte ori textul `YES` va apărea în instrucţiuni, el se va înlocui cu constanta 1.

Crearea şi gestionarea tabelelor de simboluri este o problemă de bază în procesul de compilare. Există două rutine principale care gestionează simbolurile şi textele lor de substituţie. Prima, `install(s,t)` înregistrează simbolul `s` şi textul de substituţie `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. Fiecărui simbol i se calculează un cod hash astfel: se adună codurile ASCII ale caracterelor simbolului şi se ia restul provenit din împărţirea numărului obţinut din adunare şi dimensiunea tabelului. Astfel, fiecărui simbol i se asociază un cod hash `H` care verifică relaţia:

`0<=H<0x100` (în hexazecimal)

Codul hash astfel obţinut va fi folosit apoi ca un indice într-o tabelă de pointeri. Un element al acestei tabele (masiv) indică începutul unui lanţ 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 lanţ indicat de un element al tabelei este o structură care conţine un pointer la simbol, un pointer la textul de substituţie şi un pointer la următorul bloc din lanţ. Un pointer `NULL` la următorul bloc din lanţ indică sfîrşitul lanţului.

Şablonul unei structuri (nod) este următorul:

```
struct nlist {
   char *name;
   char *def;
```

```
   struct nlist *next;/ * următoarea intrare în lanț */
   };
```

Tabelul de pointeri care indică începuturile lanțului de blocuri ce descriu simboluri de acelaşi cod hash este:

```
#define HASHSIZE 0x100
static struct nlist *hashtab[HASHSIZE];
```

Algoritmul de hashing pe care-l prezentăm nu este cel mai bun posibil, dar are meritul de a fi extrem de simplu:

```
hash(char *s) {
                /* formează valoarea hash pentru şirul s */
int hashval;
for (hashval=0; *s!='\0';)
   hashval += *s++;
return hashval % HASHSIZE;
}
```

Algoritmul de hashing produce un indice în masivul de pointeri `hashtab`. În procesul de căutare a unui simbol, dacă el există, el trebuie să fie în lanțul de blocuri care începe la adresa conținută de elementul din `hashtab` cu indicele respectiv.

Căutarea în tabela de simboluri `hashtab` se realizează cu funcția `lookup`. Dacă simbolul căutat este prezent undeva în lanț, funcția returnează un pointer la el; altfel returnează `NULL`.

```
struct nlist *lookup(char *s) {
                        /* caută şirul s în hashtab */
struct nlist *np;
for (np=hashtab[hash(s)]; np!=NULL;
     np=np->next)
    if (strcmp(s,np->name)==0)
       return np;         /* s-a găsit s */
return NULL;              /* nu s-a găsit s */
}
```

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 unu.

```
struct nlist *install(char *name, char
       *def) {      /* scrie (nume, def) în  htab */
struct nlist *np, *lookup();
char *strsav(), *alloc();
int hashval;
if ((np=lookup(name))==NULL) {    /* nu s-a găsit */
   np = (struct nlist*)alloc(sizeof(*np));
   if (np==NULL)
      return NULL;          /* nu există spaţiu */
   if ((np->name=strsav(name))==NULL)
      return NULL;
   hashval = hash(np->name);
   np->next = hashtab[hashval];
   hashtab[hashval] = np;
   }
else                        /* nodul există deja */
   free(np->def);           /* eliberează definiţia veche */
if ((np->def=strsav(def))==NULL)
   return NULL;
return np;
}
```

Deoarece apelurile la funcţiile `alloc` şi `free` pot apărea în orice ordine şi deoarece alinierea contează, versiunea simplă a funcţiei alloc, prezentată în capitolul 9 nu este adecvată aici. În biblioteca standard există funcţii de alocare fără restricţii, care se apelează implicit sau explicit de către utilizator dintr-un program scris în C pentru a obţine spaţiul de memorie necesar. Deoarece şi alte acţiuni dintr-un program pot cere spaţiu de memorie într-o

manieră asincronă, spațiul de memorie gestionat de funcția `alloc` poate să fie necontiguu. Astfel, spațiul liber de memorie este păstrat sub forma unui lanț de blocuri libere, fiecare bloc conținînd o dimensiune, un pointer la următorul bloc şi spațiul liber propriu-zis. Blocurile sînt păstrate în ordinea crescătoare a adreselor iar, ultimul bloc, de adresa cea mai mare, indică primul bloc, prin pointerul lui la blocul următor din lanț, astfel încît lanțul este circular.

Cînd se lansează o cerere, se examinează lista spațiului liber, pînă se găseşte un bloc suficient de mare pentru cererea respectivă. Dacă blocul are exact dimensiunea cerută, el se eliberează din lanțul blocurilor libere şi este returnat utilizatorului. Dacă blocul este mai mare se descompune, astfel încît partea cerută se transmite utilizatorului, iar partea rămasă se introduce înapoi în lista de spațiu liber. Dacă nu se găseşte un bloc suficient de mare pentru cererea lansată se caută un alt bloc de memorie.

Eliberarea unei zone de memorie prin intermediul rutinei `free` cauzează, de asemenea, o căutare în lista de spațiu liber, pentru a găsi locul corespunzător de inserare a blocului de memorie eliberat. Dacă blocul de memorie eliberat este adiacent cu un bloc din lista de spațiu liber la orice parte a sa, el este alipit la acel bloc, creîndu-se un bloc mai mare, astfel ca memoria să nu devină prea fragmentată. Determinarea adiacenței este uşurată de faptul că lista de spațiu liber se păstrează în ordinea crescătoare a adreselor de memorie.

Exemplul de utilizare a acestor funcții inițializează elementele masivului `hashtab` cu `NULL`. În continuare se aşteaptă de la tastatură introducerea unui nume şi a unei definiții pentru acest nume. Dacă numele introdus nu există în lista `hashtab` atunci se afişează un mesaj corespunzător, altfel se afişează vechea definiție care este apoi înlocuită de noua definiție introdusă.

```
main() {
char num[30],def[30];
int i;
struct nlist *np;
for (i=0; i<HASHSIZE; i++)
   hashtab[i] = NULL;
```

```
do {
   getword(num); getword(def);
   if ((np=lookup(num))==NULL)
      printf("New name\n");
   else
      printf("Old definition: %s\n",
             np->def);
   install(num,def);
   } while (1);
}
```

10.7. Cîmpuri

Un cîmp se defineşte ca fiind o mulţime de biţi consecutivi dintr-un cuvînt sau întreg. Adică 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 m 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:

```
flags | = EXTERNAL | STATIC;
```
apar frecvent şi ele setează biţii 1 şi 2 din caracterul sau întregul `flags` (în exemplul nostru)

în timp ce expresia:
```
flags &= (EXTERNAL | STATIC);
```
selectează biţii 1 şi 2 din flags.

Expresia:
```
if (flags & (EXTERNAL | STATIC)) ...
```
este adevărată cînd cel puţin unul din biţii 1 sau 2 din flags este unu.

Expresia:
```
if (!(flags & (EXTERNAL | STATIC))) ...
```
este adevărată cînd biţii 1 şi 2 din flags sînt ambii zero.

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 ’`:`’ 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 alți întregi. Astfel, expresiile anterioare pot fi scrise mai natural sub forma următoare:

```
flags.is_extern = flags.is_static = 1;
```

pentru setarea biților 1 şi 2 din variabila `flags`,

```
flags.is_extern = flags.is_static = 0;
```

pentru ştergerea biților, iar:

```
if (flags.is_extern==0 &&
    flags.is_static==0)
```

pentru testarea lor.

Un cîmp nu trebuie să depăşească limitele unui cuvînt. 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 biți este folosit pentru a rezerva spațiu în vederea alinierii următorului cîmp. Lungimea zero a unui cîmp poate fi folosită pentru forțarea alinierii următorului cîmp la limita unui nou cuvînt, el fiind presupus a conține 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.

Cîmpurile nu pot constitui masive, nu au adrese, astfel încît operatorul '`&`' nu se poate aplica asupra lor.

10.8. Reuniuni

O reuniune este o variabilă care poate conține, la momente diferite, obiecte de diferite tipuri şi dimensiuni; compilatorul este cel care ține evidența dimensiunilor şi aliniamentului.

Reuniunile oferă posibilitatea ca mai multe tipuri diferite de date să fie tratate într-o singură zonă de memorie, fără a folosi în program vreo informație dependentă de maşină.

Să reluăm exemplul tabelei de simboluri a unui compilator, presupunînd că constantele pot fi de tip `int`, `float` sau şiruri de caractere.

Valoarea unei constante particulare trebuie memorată într-o variabilă de tip corespunzător, cu toate că este mai convenabil, pentru gestiunea tabelei de simboluri, ca valoarea să fie memorată în aceeaşi zonă de memorie, indiferent de tipul ei şi să ocupe aceeaşi cantitate de memorie. Acesta este scopul unei reuniuni: de a furniza o singură variabilă care să poată conține 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*

sau

pointer-la-reuniune`->`*membru*

Dacă variabila `utype` este utilizată pentru a ține evidența tipului curent memorat în `uval`, atunci fie următorul cod:

```
if (utype==INT)
   printf ("%d\n",uval.ival);
else if (utype== FLOAT)
   printf("%f\n",uval.fval);
else if (utype==STRING)
   printf("%s\n",uval.pval);
else
   printf("tip incorect %d in utype\n",
          utype);
```

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. Pe exemplu, în masivul de structuri `symtab[NSYM]` definit de:

```
struct {
   char * name;
   int flags;
   int 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 pointat de `pval` prin:

```
*symtab[i].uval.pval
```

De fapt, 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. Ca şi la structuri, singurele operații permise cu reuniuni sînt accesul la un membru al reuniunii şi considerarea adresei ei.

Reuniunile nu pot fi atribuite, transmise la funcții sau returnate de către acestea. Pointerii la reuniuni pot fi folosiți în mod similar cu pointerii la structuri.

10.9. Declarații de structuri, reuniuni şi cîmpuri

Deoarece specificatorii de structuri, reuniuni şi cîmpuri au aceeaşi formă vom prezenta sintaxa lor generală în acest paragraf.

Specificator-structură-sau-reuniune:
struct-sau-union { *lista-declarațiilor* }

struct-sau-union identificator { *lista-declarațiilor* }
struct-sau-union identificator

Struct-sau-union:
`struct`
`union`

Lista-declarațiilor este o secvență de declarații pentru membrii structurii sau reuniunii.

Lista-declarațiilor:
declarație-structură
declarație-structură, *lista-declarațiilor*
Declarație-structură:
specificator-tip, *lista-declarator*;
Lista-declarator:
declarator-structură
declarator-structură, *lista-declarator*

În mod obișnuit, un declarator-structură este chiar un declarator pentru un membru al structurii sau reuniunii. Un membru al structurii poate fi constituit dintr-un număr specificat de biți, caz în care avem de-a face cu un cîmp. Lungimea lui se separă de nume prin caracterul ':' Atunci:

Declarator-structură:
declarator
declarator : *expresie-constantă*
: *expresie-constantă*

Într-o structură fiecare membru care nu este un cîmp începe la o adresă corespunzătoare tipului său. Astfel într-o structură pot exista zone fără nume neutilizate, rezultate din motive de aliniere.

Limbajul C nu introduce restricții privind tipurile obiectelor care pot fi declarate cîmpuri.

Un specificator-structură-sau-reuniune de forma a doua declară un identificator ca fiind eticheta (marcajul) structurii sau reuniunii. Atunci o declarație ulterioară poate folosi forma a treia a unui specificator-structură-sau-reuniune.

Etichetele de structuri permit definirea structurilor auto-referite; de asemenea permit ca partea de declarație a corpului structurii să fie dată o singură dată şi folosită de mai multe ori. Este interzisă declararea recursivă a unei structuri sau reuniuni, dar o structură sau o reuniune poate conține un pointer la ea.

Două structuri pot partaja o secvență inițială comună de membri; adică acelaşi membru poate apărea în două structuri diferite dacă el are acelaşi tip în ambele structuri şi dacă toți membri precedenți lui sînt identici în cele două structuri.

10.10. `Typedef`

Limbajul C oferă o facilitate numită typedef pentru a crea noi nume de tipuri de date. Specificatorul de tip typedef-nume are sintaxa:

typedef-nume:
    declarator

Într-o declarație implicînd `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:

```
typedef int LENGTH;
```

îl face pe `LENGTH` sinonim cu `int`. „Tipul" `LENGTH` poate fi folosit ulterior în declarații în acelaşi mod ca şi tipul `int`.

```
LENGTH len, maxlen;
LENGTH *length[];
```

În mod similar, declarația:

```
typedef char *STRING;
```

îl face pe `STRING` sinonim cu `char*`, adică pointer la caracter, care apoi poate fi utilizat în declarații de tipul:

```
STRING p, lineptr[LINES], alloc();
```

Se observă că tipul care se declară prin `typedef` apare pe poziția numelui de variabilă nu imediat după cuvîntul rezervat `typedef`. Sintactic `typedef` este sinonim cu clasele de memorie

`extern`, `static` etc, dar nu rezervă memorie pentru variabilele respective.

Ca un exemplu mai complicat să reluăm declarația unui nod al unui arbore, de data aceasta folosind `typedef` pentru a crea un nou nume pentru un tip structură (vezi secțiunea 10.5).

```
typedef struct tnode {
   char *word;           /* pointer la text */
   int count;            /* număr apariții */
   struct tnode *left;   /* descendent stîng */
   struct tnode *right;  /* descendent drept */
   } TREENODE, *TREEPTR;
```

Această declarație creează două nume noi de tipuri, numite `TREENODE`, care este o structură şi `TREEPTR`, care este un pointer la o structură. Atunci rutina `talloc` poate fi scrisă sub forma:

```
TREEPTR talloc() {
char *alloc();
return (TREEPTR)alloc(sizeof(TREENODE));
}
```

Trebuie subliniat faptul că declarația `typedef` nu creează noi tipuri în nici un caz; ea adaugă doar sinonime pentru anumite tipuri de date, deja existente. Variabilele declarate în acest fel au exact aceleaşi proprietăți ca şi cele declarate explicit. De fapt, `typedef` se aseamănă cu `#define`, cu excepția faptului că în timp ce `#define` este tratat de preprocesor, `typedef` este tratat de către compilator. De exemplu:

```
typedef int(*PFI)();
```

creează numele `PFI` pentru „*pointer la o funcție care returnează un întreg*”, tip care poate fi folosit ulterior într-un context de tipul:

```
PFI strcmp, numcmp, swap;
```

în programul de sortare din capitolul 9.

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.

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.

11. Intrări / ieşiri

Întrucît limbajul C nu a fost dezvoltat pentru un sistem particular de operare şi datorită faptului că s-a dorit realizarea unei portabilităţi cît mai mari, atît a unui compilator C, cît şi a programelor scrise în acest limbaj, el nu posedă facilităţi de intrare / ieşire.

Există totuşi un sistem de intrare / ieşire (sistemul I/O) constituit dintr-un număr de subprograme care realizează funcţii 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 interfaţa lor cu programele scrise în limbajul C.

11.1. 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 dispoziţia 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 secvenţiale şi în momentul execuţiei 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 execuţie. Sistemul I/O permite redirectarea acestor fişiere pe alte periferice sau închiderea lor după lansarea programului. Redirectarea fişierului `stdin` se specifică prin construcţia:

<*specificator-fişier*

în linia de comandă prin care a fost lansat programul.

Redirectarea fişierului `stdout` se specifică prin construcţia:

>*specificator-fişier*

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*

`stderr` este întotdeauna asociat terminalului de la care a fost lansat programul în execuţie şi nu poate fi redirectat.

Pentru a se putea face o referire la aceste fişiere orice program C trebuie să conţină fişierul `stdio.h`, care se include printr-o linie de forma:

```
#include <stdio.h>
```

dacă acest fişier se află în biblioteca standard.

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.

11.2. Accesul la fişiere; deschidere şi închidere

Nume

fopen - deschide un flux

Declarație

```
FILE *fopen(const char *path,
      const char *mode);
```

Descriere

Funcția `fopen` deschide fişierul al cărui nume este un şir indicat de `path` şi îi asociază un flux.

Argumentul `mode` indică un şir care începe cu una din secvențele 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 poziției în flux este la începutul fişierului, în ultimele două la sfîrşitul acestuia.

Şirul `mode` include de asemenea litera **b** (deschide un fişier binar) sau **t** (deschide un fişier text) fie pe ultima poziție fie pe cea din mijloc.

Operațiile de citire şi scriere pot alterna în cazul fluxurilor read / write în orice ordine. Să reținem că standardul ANSI C cere să existe o funcție de poziționare între o operație de intrare şi una de ieşire, sau între o operație de ieşire şi una de intrare, cu excepția cazului cînd o operație de citire detectează sfîrşitul de fişier. Această operație 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 offset,
    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 `offset` octeți la poziția specificată de `reper`. Dacă `reper` este `SEEK_SET`, `SEEK_CUR`, sau `SEEK_END`, `offset` 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 fișierul asociat fluxului `flux` la începutul fișierului. 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.

11.3. 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` 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 având mărimea `size` octeți, din fluxul indicat de `flux`, şi le memorează în zona indicată de `ptr`.

Funcția `fwrite` scrie `nel` elemente, fiecare având 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).

11.4. Citire cu format

Nume

`scanf`, **`fscanf`**, **`sscanf`** - citire cu format

Declarație

```
int scanf(const char *format, ...);
int fscanf(FILE *flux, const char *format,
    ...);
int sscanf(const char *str, const char
    *format, ...);
```

Descriere

Familia de funcții *scanf* scanează intrarea în concordanță cu şirul de caractere `format` 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 **`dioux`** sau **`n`** şi argumentul asociat este un pointer la `short` (în loc de `int`).

`l` Conversia este de tip **`dioux`** sau **`n`** şi argumentul asociat este un pointer la `long` (în loc de `int`), sau conversia este de tip **`efg`** şi argumentul asociat este un pointer la `double` (în loc de `float`).

`L` Conversia este de tip **`efg`** 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 apariția unui caracter care nu se află (sau, dacă se precizează **^**, care se află) în set sau dacă se atinge mărimea maximă specificată.

p Potrivire cu o valoare pointer (aşa cum se afişează cu **%p** în `printf`); argumentul asociat trebuie să fie un pointer la pointer.

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

Funcțiile 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 situație 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 un 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.

11.5. Scriere cu format

Nume

printf, **fprintf**, **sprintf** - scriere cu format

Declarație

```
int printf(const char *format, ...);
int fprintf(FILE *flux, const char
    *format, ...);
int sprintf(char *str, const char *format,
    ...);
```

Descriere

Funcțiile din familia *printf* generează o ieşire în concordanță cu format după cum se descrie mai jos. Funcția `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 format 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 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 mulţi indicatori:

Valoarea numerică se converteşte în format alternativ. Pentru conversii de tip **o**, primul caracter al şirului de ieşire este zero (prin prefixare cu `0` dacă valoarea nu este zero). Pentru conversii de tip **x** şi **X**, o valoare nenulă este prefixată cu `0x` (sau `0X` pentru conversii de tip **X**). Pentru conversii de tip **e**, **E**, **f**, **F**, **g** şi **G**, rezultatul va conţine întotdeauna punctul zecimal, chiar dacă nu apare partea fracţionară (în mod normal punctul zecimal apare în aceste conversii numai dacă există şi partea fracţionară). Pentru conversii de tip **g** şi **G** zerourile finale nu sînt eliminate aşa cum se procedează în mod normal. Pentru alte conversii rezultatul este nedefinit.

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 excepţia 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 (spaţiu) 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.

Lățimea cîmpului

Un şir de cifre zecimale (cu prima cifră nenulă) specifică o lățime minimă pentru cîmp. Dacă valoarea convertită are mai puține caractere decît lățimea specificată, va fi completată cu spații la stînga (sau dreapta, dacă s-a specificat aliniere la stînga). În locul unui număr zecimal se poate folosi `*` pentru a specifica faptul că lățimea cîmpului este dată de argumentul următor, care trebuie să fie de tip `int`. O valoare negativă pentru lățime este considerată un indicator **-** urmat de o valoare pozitivă pentru lățime. În nici un caz nu se va trunchia cîmpul; dacă rezultatul conversiei este mai mare decît lățimea cîmpului, cîmpul este expandat pentru a conține rezultatul conversiei.

Precizia

Precizia (opțională) 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 următor, care trebuie să fie de tip `int`. Dacă precizia este dată doar de **.**, 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**.

Modificator de lungime

În acest caz prin conversie întreagă înțelegem 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 întreagă care urmează corespunde unui argument `long` sau `unsigned long`, sau următoarea conversie de tip **n** corespunde unui argument de tip pointer la `long`.

L Următoarea conversie de tip **e**, **E**, **f**, **g** sau **G** corespunde unui argument `long double`.

Specificator de conversie

Un caracter care specifică tipul conversiei care se va face. Specificatorii de conversie şi semnificaţia lor sînt:

d,**i**

Argumentul de tip `int` este convertit la notaţia zecimală cu semn. Precizia, dacă este dată, dă numărul minim de cifre care trebuie să apară; dacă valoarea convertită necesită mai puţine 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 notaţie 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 puţine 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 puțin 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 fracționară a rezultatului; punctul zecimal apare numai dacă este urmat de cel puțin 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, sau dacă este mai mare decît mărimea şirului, şirul trebuie să conțină 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`).

11.6. 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);
int fileno( 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.

Funcția `fileno` examinează argumentul `flux` şi returnează descriptorul asociat de sistemul de operare acestui flux.

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 şi ultima linie a fişierului text de intrare este terminată cu new-line, aceasta va fi scrisă de două ori în fişierul de ieşire. De ce? După ce se citeşte ultima linie încă nu este

poziționat indicatorul de sfîrşit de fişier, deci funcția `fgets` returnează succes. La reluarea ciclului se încearcă un nou `fgets` şi abia acum se depistează sfîrşitul de fişier, fapt marcat în zona rezervată fluxului `fi`. Astfel conținutul tabloului `lin` rămîne nemodificat şi este scris a doua oară în fişierul de ieşire. Abia la o nouă reluare a ciclului funcția `feof` ne spune că s-a depistat sfîrşitul de fişier.

În acest manual sînt prezentate mai multe programe care efectuează diferite prelucrări asupra unor fişiere text. Pentru simplitate toate programele presupun că nu apar erori la citire sau la scriere.

11.7. Operații cu directoare

Funcțiile de parcurgere a cataloagelor de fişiere descrise în această secțiune (`opendir`, `readdir`, `closedir`) sînt definite de mai multe medii de programare C (Borland, Watcom, Visual C, GNU Linux), precum şi de standardul POSIX. Aceste funcții sînt descrise în `<dirent.h>`.

Funcțiile de redenumire şi ştergere a unor fişiere 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`, şi 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 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

Declaraţie

```
int rename(const char *old, const char
    *new);
int remove(const char *name);
```

Descriere

Funcţia `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).

Funcţia `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.

11.8. 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 conţinut 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 80
char zon[LZON];
FILE *fi, *fo;
int 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 conținutului 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;
}
```

12. 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.

12.1. 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 tablou 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` schimbă 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, care este 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, care este aliniată în mod corespunzător pentru orice tip de variabile, şi poate fi diferită de `ptr`, sau 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.

12.2. Sortare şi căutare

Nume

`qsort` - sortează un tablou

`bsearch` - căutare binară într-un tablou 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 tablou de `nel` elemente, fiecare de mărime `size`. Argumentul `base` indică spre începutul tabloului.

Elementele tabloului 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 tabloul sortat este nedefinită.

Funcția `bsearch` caută într-un tablou de `nel` elemente, fiecare de mărime `size`, un membru care coincide cu obiectul indicat de `key`. Argumentul `base` indică spre începutul tabloului.

Conținutul tabloului 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 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.

Valoare returnată

Funcția `bsearch` returnează un pointer la un membru al tabloului care coincide cu obiectul indicat de `key`, sau `NULL` dacă nu se găseşte nici un membru. Dacă există mai multe elemente care coincid cu `key`, poate fi returnat oricare element cu această proprietate.

12.3. 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 islower(int c);
int isalpha(int c);    int isprint(int c);
int isascii(int c);    int ispunct(int c);
int iscntrl(int c);    int isspace(int c);
int isdigit(int c);    int isupper(int c);
int isgraph(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; este echivalentă cu `(isalpha(c) || isdigit(c))`.

`isalpha`

Verifică dacă `c` este alfabetic; este echivalent cu `(isupper(c) || islower(c))`.

`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 spaţiu.

`ispunct`
Verifică dacă `c` este un caracter diferit de spaţiu şi non-alfanumeric.

`isspace`
Verifică dacă `c` este un spaţiu 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 funcţiile `is`... este nenulă dacă caracterul `c` se află în clasa testată, şi zero în caz contrar.

Valoarea returnată de funcţiile `to`... este litera convertită dacă caracterul `c` este o literă, şi nedefinită în caz contrar.

12.4. Operaţii cu blocuri de memorie

Pentru majoritatea funcţiilor din această categorie compilatorul expandează codul acestora folosind instrucţiuni pe şiruri de caractere. Declaraţiile acestor funcţii se obţin cu

```
#include <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.

12.5. Operații cu şiruri de caractere

Pentru majoritatea funcțiilor din această categorie compilatorul expandează codul acestora folosind instrucțiuni pe şiruri de caractere. Declarațiile acestor funcții se obțin cu

```
#include <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` calculează lungimea segmentului inițial din `s` format în întregime numai cu caractere din `acc`.

Funcția `strcspn` calculează 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`.

12.6. Biblioteca matematică

1) Funcțiile din prima categorie sînt descrise 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` și `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 aceeași valoare `seed`.

Se obișnuiește 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 funcții:

```
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

2) Funcțiile din a doua categorie sînt descrise în fişierul `<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);
```
$\sqrt{x}$

```
double sin(double x);
```
$\sin(x)$

```
double cos(double x);
```
$\cos(x)$

```
double tan(double x);
```
$\mathrm{tg}(x)$

```
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);  tgh(x)
double ldexp(double x, int e);     x · 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*.

12.7. 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 tabloul `X` (alocat dinamic), şi apoi le sortează crescător. În continuare se generează `k` valori întregi aleatoare pe care le caută în tabloul `X`. Pentru fiecare căutare cu succes se afişează pe terminal valoarea căutată şi poziția în tablou.

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 `Fld` 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 tabloul `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[10], ar;
   float na, nb, nc, md;
   } StEl;
```

```
#define Fld(P,f) ((StEl *)P)->f
int comp(const void *A, const void *B) {
float w;
int d;
if (d=Fld(A,ar)-Fld(B,ar))
   return d;
if (Fld(A,ar)=='A') {
   w=Fld(B,md)-Fld(A,md);
   if (w>0) return 1;
   if (w<0) return -1;
   }
return strcmp(Fld(A,nm),Fld(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 %d %d %d",
    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)`? pentru că 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

Brian W Kernigham, Dennis M Ritchie - *The C Programming Language*
Prentice-Hall Software Series, 1978

* * * - *Limbajul C*; *manual de programare*
Institutul de tehnică de calcul, Cluj-Napoca 1984

Herbert Schildt - *Manual C complet*
Editura Teora, 1998

Manuale electronice

http://www.programmingtutorials.com/c.html

Marshall Brain - *Introduction to C Programming*
http://devcentral.iftech.com/learning/tutorials/c-cpp/c/

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

Brian Brown - *C Programming*
http://www.cit.ac.nz/smac/cprogram/onlinet.htm

Brian Brown - *An Introduction to C Programming*
http://www.cit.ac.nz/smac/cprogram/default.htm

Cuprins

1. Generalități asupra limbajului C 4
1.1. Introducere 4
1.2. Primele programe 5
1.3. Meta-limbajul şi setul de caractere 9

2. Unitățile lexicale ale limbajului C 10
2.1. Identificatori 10
2.2. Cuvinte cheie 10
2.3. Constante 11
2.4. Şiruri 14
2.5. Operatori 15
2.6. Separatori 15

3. Variabile 17
3.1. Clase de memorie 17
3.2. Tipuri de variabile 20
3.3. Obiecte şi valori-stînga 23
3.4. Conversii de tip 23

4. Operatori şi expresii 27
4.1. Expresii primare 27
4.2. Operatori unari 29
4.3. Operatori multiplicativi 32
4.4. Operatori aditivi 33
4.5. Operatori de deplasare 33
4.6. Operatori relaționali 34
4.7. Operatori de egalitate 35
4.8. Operatorul ŞI pe biți 35
4.9. Operatorul SAU-exclusiv pe biți 35
4.10. Operatorul SAU-inclusiv pe biți 36
4.11. Operatorul ŞI-logic 36
4.12. Operatorul SAU-logic 37
4.13. Operatorul condițional 37
4.14. Operatori de atribuire 38

4.15. Operatorul virgulă 39
4.16. Precedența și ordinea de evaluare 40

5. Declarații 42
5.1. Specificatori de clasă de memorie 43
5.2. Specificatori de tip 44
5.3. Declaratori 45
5.4. Modificatorul `const` 47
5.5. Inițializare 48
5.6. *Nume-tip* 49

6. Instrucțiuni 51
6.1. Instrucțiunea *expresie* 51
6.2. Instrucțiunea compusă sau blocul 51
6.3. Instrucțiunea condițională 52
6.4. Instrucțiunea `while` 54
6.5. Instrucțiunea `do` 54
6.6. Instrucțiunea `for` 55
6.7. Instrucțiunea `switch` 56
6.8. Instrucțiunea `break` 58
6.9. Instrucțiunea `continue` 58
6.10. Instrucțiunea `return` 59
6.11. Instrucțiunea vidă 59

7. Funcțiile și structura unui program 60
7.1. Definiția funcțiilor 60
7.2. Apelul funcțiilor 63
7.3. Revenirea din funcții 63
7.4. Argumentele funcției și transmiterea parametrilor 64
7.5. Funcții cu număr variabil de parametri 65
7.6. Exemple de funcții și programe 66

8. Liniile de control ale compilatorului 70
8.l. Înlocuirea simbolurilor, substituții macro 70
8.2. Includerea fișierelor 72
8.3. Compilarea condiționată 72
8.4. Utilizarea directivelor de compilare 73

9. Pointeri şi masive 76
9.1. Pointeri şi adrese 76
9.2 Pointeri şi argumente de funcţii 77
9.3. Pointeri şi masive 78
9.4. Aritmetica de adrese 80
9.5. Pointeri la caracter şi funcţii 83
9.6. Masive multidimensionale 86
9.7. Masive de pointeri şi pointeri la pointeri 88
9.8. Iniţializarea masivelor şi masivelor de pointeri 93
9.9. Masive de pointeri şi masive multidimensionale 96
9.10. Argumentele unei linii de comandă 97
9.11. Pointeri la funcţii 102

10. Structuri şi reuniuni 106
10.1. Elemente de bază 106
10.2. Structuri şi funcţii 108
10.3. Masive de structuri 110
10.4. Pointeri la structuri 115
10.5. Structuri auto-referite 117
10.6. Căutare în tabele 123
10.7. Cîmpuri 127
10.8. Reuniuni 129
10.9. Declaraţii de structuri, reuniuni şi cîmpuri 131
10.10. `Typedef` 133

11. Intrări / ieşiri 136
11.1. Intrări şi ieşiri standard; fişiere 136
11.2. Accesul la fişiere; deschidere şi închidere 138
11.3. Citire şi scriere fără format 141
11.4. Citire cu format 143
11.5. Scriere cu format 147
11.6. Tratarea erorilor 152
11.7. Operaţii cu directoare 154
11.8. Programe demonstrative 156

12. Alte rutine din biblioteca standard 160
12.1. Alocarea dinamică a memoriei 160

12.2. Sortare şi căutare 161
12.3. Rutine de clasificare 162
12.4. Operaţii cu blocuri de memorie 164
12.5. Operaţii cu şiruri de caractere 166
12.6. Biblioteca matematică 170
12.7. Programe demonstrative 173

Bibliografie 178