CAPITOLUL 2. TIPURI, OPERATORI SI EXPRESII

Variabilele si constantele sunt obiectele - date de baza manipulate intr-un program. Declaratiile listeaza variabilele ce se vor folosi si specifica tipul lor si probabil, valorile lor initiale. Operatorii specifica ce trebuie facut cu ele. Expresiile combina variabile si constante pentru a produce valori noi. Toate acestea constituie subiectul acestui capitol.

2.1. Nume de variabile

Exista unele restrictii asupra numelor de constante si variabile. Numele sunt alcatuite din litere si cifre; primul caracter trebuie sa fie o litera. Liniuta de subliniere "_" este considerata litera; ea este utila in usurarea citirii numelor lungi de variabile. Literele mari si mici sunt caractere distincte; practica traditionala in C foloseste literele mici pentru nume de variabile si literele mari pentru constantele simbolice. Numai primele opt caractere ale unui nume intern sunt semnificative, cu toate ca se pot folosi mai multe. Pentru numele externe, de exemplu nume de functii si de variabile externe, numarul de caractere poate sa fie mai mic ca 8, deoarece numele externe sunt folosite de diferite asambloare si incarcatoare. Mai mult, cuvinte cheie ca: **if**, **else**, **int**, etc sunt rezervate: nu pot fi folosite ca nume de variabile (trebuie sa fie scrise cu litere mici). Este intelept sa alegem numele de variabile astfel incit sa insemne ceva, legat de scopul variabilei, si e neplacut sa amestecam litere mari cu mici.

2.2. Tipuri si marimi de date

Exista numai citeva tipuri de date de baza in limbajul C:

char un singur octet, capabil sa pastreze un caracter din setul local de caractere

int un intreg, reflectind tipic marimea efectiva a intregilor pe calculatorul gazda

float numar flotant in simpla precizie

double numar flotant in dubla precizie.

In plus, exista un numar de calificatori care pot fi aplicati tipului "int": short, long si unsigned. short si long se refera la diferite marimi de intregi. Numerele "unsigned" se supun legilor aritmeticii modulo 2^n unde n este numarul de biti dintr-un int; ele sunt intodeauna pozitive. Declaratiile pentru calificatori arata astfel:

```
short int x;

long int y;

unsigned int z;
```

Cuvintul int poate fi omis in astfel de situatii, ceea ce se si intimpla de obicei. Precizia acestor obiecte este data in tabelul urmator:

char 8 biti
int 16 biti
short 16 biti
long 32 biti
float 32 biti
double 64 biti

Intentia e ca short si long sa aiba lungimi diferite de intregi unde e practic; int reflecta normal, cea mai "naturala" lungime pentru un calculator. Ceea ce trebuie sa notati este ca short nu este niciodata mai lung decit long.

2.3. Constante

Constantele int si float au fost deja expuse; notam in plus ca notatia uzuala

123.456e-7

sau notatia stiintifica

0.12E3

pentru numerele flotante sunt ambele legale. Orice constanta flotanta este considerata ca fiind de tipul double, asa ca notatia "e" serveste atit pentru float cit si pentru double. Constantele lungi sunt scrise in stilul 123L. O constanta intreaga normala care este prea lunga pentru un int, este luata deasemenea ca fiind o constanta long.

O constanta caracter este un caracter singur scris intre ghilimele simple ca, de exemplu, 'x'. Valoarea unei constante caracter este valoarea numerica a caracterului in setul de caractere al calculatorului. De exemplu, in setul de caractere ASCII, caracterul zero, sau '0', are valoarea 48. Constantele caracter participa in operatiile numerice la fel ca oricare alte numere, cu toate ca cel mai adesea ele sunt folosite in comparari cu alte caractere.

O expresie constanta este o expresie care implica numai constante. Astfel de expresii sunt evaluate la compilare si nu la executie si ele pot fi folosite in orice loc in care poate apare o constanta, ca in

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

sau

```
seconds = 60 * 60 * hours;
```

O constanta-sir este o secventa compusa din zero sau mai multe caractere intre ghilimele duble, ca

"I am a string"

sau

```
"" /* un sir nul */
```

Ghilimelele duble nu sunt parte a sirului ci servesc doar ca delimitatori. Aceleasi secvente escape folosite pentru constantele caracter se aplica si la siruri; \" reprezinta caracterul dubla ghilimea.

Tehnic, un sir este un tablou ale carui elemente sunt caractere. Compilatorul plaseaza automat un caracter nul \0 la sfirsitul oricarui astfel de sir, astfel ca programele pot determina lesne sfirsitul sirului. Aceasta reprezentare spune ca nu exista o limita reala pentru lungimea unui sir, dar programele trebuie sa parcurga tot sirul pentru a-i determina lungimea. Memoria fizica ceruta este cu o locatie mai mult decit numarul de caractere scrise intre ghilimele duble. Functia urmatoare, strlen(s) returneaza lungimea unui sir de caractere s, exclusiv terminatorul \0.

```
strlen(s) /* returneaza lungimea lui s */
char s[];
{
int i;
i = 0;
while (s[i] != '\0')
++i;
return(i);
}
```

Trebuie distins intre o constanta caracter si un sir care contine un singur caracter: 'x' si "x" nu sunt acelasi lucru. Primul este un caracter, folosit pentru a produce valoarea numerica a caracterului x din setul de caractere al calculatorului; al doilea este un sir de caractere care contine un singur caracter (litera x) si un \0.

2.4. Declaratii

Toate variabilele trebuie declarate inainte de a fi folosite, cu toate ca anumite declaratii pot fi facute implicit de context. O declaratie specifica un tip si este urmata de o lista de una sau mai multe variabile de acel tip, ca in exemplul de mai jos:

```
int lower, upper, step;
char c, line[1000];
```

Variabilele pot apare oricum printre declaratii. Lista de mai sus poate fi scrisa, in mod egal, si astfel:

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

Aceasta ultima forma ocupa mai mult spatiu dar este mai comoda pentru a adauga cite un comentariu la fiecare declaratie sau pentru modificari ulterioare.

Variabilele pot fi, deasemenea, initializate in declaratia lor, cu toate ca exista anumite restrictii. Daca numele este urmat de un semn egal si de o constanta, aceasta serveste la initializare, ca in:

```
char backslash = '\\';
int i = 0;
float eps = 1.0e-5;
```

Daca variabila in chestiune este externa sau statica, initializarea este facuta o singura data, conceptual inainte ca programul sa-si inceapa executia. Variabilele automate initializate explicit sunt initializate la fiecare apel al functiei in care sunt continute. Variabilele automate pentru care nu exista o initializare explicita au valoare nedefinita (adica gunoi). Variabilele externe si statice se initializeaza implicit cu zero dar este un bun stil de programare acela de a declara initializarea lor in orice caz. Vom discuta initializarile mai departe pe masura ce se introduc noi tipuri de date.

2.5. Operatori aritmetici

Operatorii aritmetici binari sunt "+", "-", "*", "/" si operatorul modulo "%". Exista operatorul "-" unar dar nu exista operatorul unar "+". Impartirea intregilor trunchiaza orice parte fractionara. Expresia

```
x % y
```

produce restul cind x se imparte la y si deci este zero cind impartirea este exacta.

Operatorul % nu poate fi aplicat la float sau double.

Operatorii + si - au aceeasi pondere, care este mai mica decit ponderea (identica) a lui *, / si % care la rindul ei este mai mica decit ponderea operatorului unar -.

Operatorii aritmetici se grupeaza de la stinga la dreapta. Ordinea de evaluare nu este specificata pentru operatorii asociativi si comutativi de tipul lui * si +. Compilatorul poate rearanja un calcul cu paranteze implicind unul din acestia. Astfel, a+(b+c) poaate fi evaluat ca (a+b)+c. Acest lucru produce rar diferente dar daca se cere o ordine particulara, trebuie folosite explicit variabilele temporare.

2.6. Operatori relationali si logici

Operatorii relationali sunt **>**, **>=, <**, **<=**. Ei au toti aceasi pondere. Sub ei in tabelul de ponderi se afla operatorii de egalitate **==**, **!=** , care au o aceeasi pondere. Operatorii relationali au ponderea mai mica decit cei aritmetici, asa ca expresii de tipul i < lim-1 se evalueaza ca i < (lim-1), asa cum ar fi de asteptat.

Mai interesanti sunt conectorii logici && si ||. Expresiile care-i contin sunt evaluate de la stinga la dreapta si evaluarea se opreste in clipa in care se cunoaste adevarul sau falsul rezultatului. Aceste proprietati se dovedesc critice in scrierea programelor. De exemplu iata o bucla:

```
for (i=0; i<lim-1 && (c = getchar()) != '\n' && c != EOF;++i)
s[i] = c;
```

In mod clar, inainte de a citi un nou caracter trebuie vazut daca mai exista loc pentru a-l depune in tabloul s , asa ca testul i<lim-1 trebuie facut in primul rind. Nu numai atit dar daca testul esueaza, nu trebuie sa mai citim un nou caracter. Similar, ar fi nepotrivit sa testam daca c este EOF inainte de apelul lui getchar; apelul trebuie sa aiba loc inainte ca sa testam caracterul c. Ponderea lui && este mai mare decit cea a lui || si amindoua sunt mai mici decit cele ale operatorilor relationali si de egalitate, asa ca expresii de tipul

```
i<lim-1 && (c = getchar()) != '\n' && c != EOF
```

nu mai au nevoie de paranteze suplimentare. Dar, deoarece ponderea lui != este mai mare decit cea a asignarii, este nevoie de paranteze in

```
(c = getchar()) != '\n'
```

pentru a obtine rezultatul dorit. Operatorul unar de negatie "!" converteste un operand nonzero sau adevarat in zero si un operand zero sau fals in 1. O utilizare obisnuita a lui ! este in constructii de tipul

```
if (!inword)
```

mai degraba decit

```
if (inword == 0)
```

Este mai greu sa generalizam care forma este mai buna. Constructiile de tipul !inword arata mai frumos ("daca nu e in cuvint"), dar constructiile mai complicate pot fi greu de inteles.

Exercitiul 2.1. Scrieti o bucla echivalenta cu bucla for de mai sus fara a folosi &&.

2.7. Conversii de tip

Cind intr-o expresiie apar operanzi de mai multe tipuri, ei se convertesc intr-un tip comun, dupa un numar mic de reguli. In general, singurele conversii care se fac automat sunt acelea cu sens , de exemplu convertirea unui numar intreg intr-un flotant in expresii de tipul f + i. Expresiile fara sens, de exemplu folosirea unui float ca indice de tablou, nu sunt permise. In primul rind, char-i si int-i pot fi amestecati in expresiile aritmetice: orice char este convertit automat intr-un int. Aceasta permite o flexibilitate remarcabila in anumite tipuri de transformari de caractere. Exemplificam cu functia atoi, care converteste un sir de cifre in echivalentul lui numeric.

```
atoi(s) /* converteste un sir s in intreg */
char s[];
{
int i, n;
n = 0;
for (i = 0; s[i] >= '0' && s[i] <= '9' ; ++i)
n = 10 * n + s[i] - '0';
return(n);
```

```
}
```

Asa cum am vazut in Capitolul 1, expresia

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

reprezinta valoarea numerica a caracterului aflat in s[i] deoarece valorile lui '0','1', etc formeaza un sir crescator pozitiv si contiguu. Un alt exemplu de conversie intre char si int il constituie functia lower care transforma literele mari din setul de caractere ASCII in litere mici. Daca intrarea nu este o litera mare, functia o returneaza neschimbata:

```
lower(c) /* conversie ASCII litere mari in litere mici */
int c;
{
if (c >= 'A' && c <= 'Z')
return(c + 'a' - 'A');
else
return(c)
}
```

Aceasta functie este valabila numai pentru ASCII deoarece pe de o parte intre literele mari si literele mici exista o distanta fixata, ca valoare numerica, iar pe de alta parte ambele alfabete sunt contigue - intre A si Z se gasesc numai litere.

O alta forma utila de conversie de tip automata este aceea ca expresiile relationale de tipul i > j si expresiile logice conectate prin && si || se definesc a avea valoarea 1 pentru adevar si 0 pentru fals. Astfel, o asignare:

```
isdigit = c >= '0' && c <= '9';
```

pune pe isdigit pe 1 daca c este o cifra si pe 0 daca nu. (In partea de test a lui if, while ,for, etc, "adevarat" inseamna "nonzero").

Conversiile aritmetice implicite lucreaza in mare masura cum ne asteptam. In general, daca un operator ca + sau * care are doi operanzi (un "operator binar") are operanzi de tipuri diferite, tipul "inferior" este promovat la tipul "superior" inainte de executia operatiei. Rezultatul insusi este de tipul superior. Mai precis, pentru fiecare operator aritmetic, se aplica urmatoarea secventa de reguli de conversie:

char si short se convertesc in int iar float este convertit in double.

Apoi, daca un operand este double , celalalt este convertit in double iar rezultatul este double.
Altfel, daca un operand este long, celalalt este convertit in long iar rezultatul este long.
Altfel, daca un operand este unsigned, celalalt este convertit inunsigned, iar rezultatul este unsigned.
Altfel, operanzii trebuie sa fie int, iar rezultatul este int.

Sa notam ca toti float dintr-o expresie sunt convertiti in double; orice calcul flotant in C este facut in dubla precizie. Conversiile se fac in asignari; valoarea partii drepte este convertita la tipul din stinga, care este tipul

rezultatului. Un caracter este convertit intr-un int fie cu extensie de semn, fie nu, asa cum s-a descris mai sus. Operatia inversa, int in char, se comporta bine, pur si simplu, bitii de ordin superior in exces sunt eliminati. Astfel, in:

```
int i;
char c;
i = c;
c = i;
```

valoarea lui c este neschimbata. Acesta este adevarat si cind extensia de semn este implicita si cind nu este implicita.
Daca x este float iar i este int, atunci:

```
x = i;
```
si
```
i = x;
```

provoaca amindoua conversii; float in int provoaca trunchierea oricarei parti fractionare. double este convertit in float prin rotunjire. Intregii lungi sunt convertiti in scurti sau in char prin pierderea bitilor de ordin superior in exces.

Deoarece argumentul unei functii este o expresie, conversia de tip are loc deasemenea si cind argumentele sunt pasate functiei in particular, char si short devin int, iar float devine double. Iata de ce am declarat argumentul functiei ca fiind int si double chiar cind functia este apelata cu char si float.

Exercitiul 2.2. Scrieti o functie htoi(s) care converteste un sir de cifre hexazecimale in valoarea sa intreaga echivalenta. Cifrele sunt de la 0 la 9, literele de la a la f si de la A la F.

2.8. Operatori de incrementare si decrementare

Limbajul C ofera doi operatori neuzuali pentru incrementarea si decrementarea variabilelor. Operatorul de incrementare ++ aduna 1 la operandul sau; operatorul de decrementare -- scade 1. Am folosit frecvent ++ pentru a incrementa variabilele, de exemplu:

```
if (c == '\n')
++nl;
```

Aspectul neobisnuit al lui ++ si al lui -- este acela ca ei pot fi folositi atit ca operatori prefix (inaintea variabilei, ca in ++n) cit si ca operatori sufix (dupa variabila, ca in n++). In ambele cazuri, efectul este incrementarea lui n. Dar expresia ++n il incrementeaza pe n inainte de a-i folosi valoarea, in timp ce expresia n++, il incrementeaza pe n dupa ce a fost folosita valoarea lui. Aceasta inseamna ca intr-un context in care valoarea este folosita, si nu numai efectul, ++n si n++ sunt diferiti. Daca n este 5, atunci:

```
x = n++;
```

il face pe x egal cu 5, dar

x = ++n;

il face pe x egal cu 6. In ambele cazuri, n devine 6. Operatorii de incrementare si decrementare se pot aplica numai variabilelor. O expresie de tipul x = (i+j)++ este ilegala. Intr-un context in care valoarea nu este folosita, ci numai efectul de incrementare, ca in

```
if (c == '\n')
nl++;
```

alegeti modul prefix sau sufix dupa cum doriti.

Exercitiul 2.3. Scrieti o alta versiune a lui squeeze(s1, s2) care sterge fiecare caracter din s1 care se potriveste cu vreun caracter din s2.

Exercitiul 2.4. Scrieti functia any(s1, s2) care returneaza prima locatie din sirul s1 in care apare vreun c acter din sirul s2, sau pe -1 daca s1 nu contine nici un caracter din s2.

2.9. Operatori logici pe biti

Limbajul C ofera un numar de operatori pentru manipularea bitilor; acestia nu se pot aplica lui float si double.

2.10. Operatori si expresii de asignare

Expresii de tipul:

```
i = i + 2
```

in care membrul sting este repetat in membrul drept pot fi scrise intr-o forma condensata:

```
i += 2
```

folosind operatorul de asignare +=. Majoritatea operatorilor binari (operatori ca +, care au un operand sting si un operand drept) au un operator de asignare corespunzator "op=", unde op este unul din:

```
+ - * / % << >> & ^ |
```

Daca e1 si e2 sunt doua expresii, atunci:

```
e1 op= e2
```

este echivalent cu

```
e1 = (e1) op (e2)
```

cu exceptia ca e1 este calculat o singura data. Sa remarcam parantezele din jurul lui e2:

x *= y + 1

inseamna de fapt

x = x * (y + 1)

si nu

x = x * y + 1

CAPITOLUL 3. CONTROLUL FLUXULUI

Instructiunile de control al fluxului dintr-un limbaj specifica ordinea in care se fac calculele. Ne-am intilnit deja cu cele mai cunoscute constructii de control al fluxului din limbajul C, in exemplele date in paginile anterioare; in cele ce urmeaza, vom completa setul de instructiuni si vom fi mult mai precisi asupra celor discutate mai sus.

3.1. Instructiuni si blocuri

O expresie ca de exemplu x = 0 sau i++ sau printf(...) devine instructiune cind este urmata de punct si virgula, ca in:

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

In limbajul C, punct-virgula este terminator de instructiune.

Acoladele { si } sunt folosite pentru a grupa impreuna instructiuni si declaratii intr-o instructiune compusa sau bloc, asa ca ele sunt suntactic echivalente cu o singura instructiune. Acoladele ce inchid instructiunile unei functii sau cele pentru instructiunile multiple dupa un if, else, while, for sunt exemple clare pentru aceasta. Nu se pune niciodata punct si virgula dupa acolada inchisa care termina un bloc.

3.2. If-Else

Instructiunea If-Else este folosita pentru luarea de decizii. Formal, sintaxa ei este:

```
if(expresie)
instructiune-1
else
instructiune-2
```

unde partea "else" este optionala. "Expresia" este evaluata; daca este "adevarata" (adica, are o valoare nenula), "instructiune-1" este executata. Daca ea este "falsa" ("expresia" este zero) si daca exista partea cu

"else", se executa in schimb "instructiune-2". Deoarece un "if" testeaza pur si simplu valoarea numerica a unei expresii, sunt posibile anumite prescurtari de cod. Cel mai clar exemplu este scriind

```
if(expresie)
```

in loc de

```
if(expresie != 0)
```

Citeodata, acest lucru este natural si clar. Altadata poate parea cifrat. Deoarece partea cu "else" a unui if-else este optionala, se poate ajunge la o ambiguitate cind se omite un else dintr-o secventa imbricata de if-uri. Aceasta este rezolvata, ca de obicei, asa: else este asociat cu if-ul anterior cel mai apropiat, care nu face pereche cu un "if". De exemplu, in:

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

else face pereche cu if cel mai dinauntru, asa cum am aratat prin tabulare. Daca nu dorim aceasta, trebuie sa folosim acolade pentru a forta asocierea potrivita:

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

Sa notam ca exista un punct si virgula dupa z = a in:

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

Aceasta deoarece, gramatical, dupa if urmeaza o instructiune si o instructiune de asignare de tipul z = a se termina intotdeauna cu punct si virgula.

3.3. Else-If

Constructia

```
if (expresie)
```

```
instructiune
else if (expresie)
instructiune
else if (expresie)
instructiune
else
instructiune
```

apare atit de des incit este demn de purtat o discutie scurta si separata asupra ei. Aceasta secventa de if-uri este calea cea mai generala de a scrie decizii multiple. Expresiile sunt evaluate in ordine; daca o expresie este adevarata, instructiunea asociata cu ea este executata, si aceasta termina intregul lant. Codul pentru fiecare "instructiune" este fie o instructiune, fie un grup intre acolade. Ultima parte de "else" manipuleaza cazul "niciuna din cele mai de sus" sau implicit, in care nici una din conditii nu este indeplinita. Citeodata nu exista nici o actiune explicita pentru cazul implicit; in acest caz,

```
else
instructiune
```

poate fi omisa, sau poate fi utila pentru verificarea de erori, pentru a prinde o conditie "imposibila". Pentru a ilustra o decizie trivalenta, dam o functie binara de cautare, care decide daca o valoare particulara x apare intr-un tablou sortat v. Elementele lui v trebuie sa fie in ordine crescatoare. Functia returneaza pozitia (un numar intre 0 si n-1) daca x apare in v, si -1 daca nu.

```
binary (x, v, n) /* gaseste pe x in v[0], v[1], ..., v[n-1] */
int x, v[], n;
{
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])
         l       ow = mid + 1;
         else /* gasit potrivirea */
         return(mid);
        }
return(-1);
}
```

Decizia fundamentala este aceea daca x este mai mic decit, mai mare decit sau egal cu elementul din mijloc v[mid] la fiecare pas; aceasta este natural pentru un if-else.

3.4. Switch

Instructiunea switch este realizator special de decizii multiple care testeaza daca o expresie se potriveste cu una dintr-un numar de valori constante si ramifica corespunzator programul. In capitolul 1 am scris un program care contorizeaza aparitiile fiecarei cifre, a spatiului, si a tuturor celorlalte caractere, folosind o secventa de if ...else. Dam in continuare acelasi program cu instructiunea switch.

```
main() /* contorizeaza cifre , blancuri , alte caractere */
{
int c, i, nwhite, nother, ndigit[10];
nwhite = nother = 0;
for (i = 0; i < 10; i++)
ndigit[i] = 0;
while ((c = getchar()) != EOF)
switch (c) {
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
ndigit[c-'0']++;
break;
case ' ':
case '\n':
case '\t':
nwhite++;
break;
default:
nother++;
break;
}
printf("digits =");
for (i = 0; i < 10; i++)
printf(" %d", ndigit[i]);
printf("\nwhite space= %d, other= %d\n", nwhite,nother);
}
```

Switch evalueaza expresia intreaga din paranteze (in acest program caracterul c) si compara valoarea ei cu toate cazurile. Fiecare caz trebuie sa fie etichetat cu o constanta intreaga sau caracter sau cu o expresie constanta. Daca un caz se potriveste cu valoarea expresiei, executia incepe la acel caz. Cazul etichetat

"default" este executat daca nici unul din cazuri nu este satisfacut. Un "default" este optional; daca el nu este prezent si nici unul din cazuri nu se potriveste nu se executa nici o actiune. Cazurile si "default" pot apare in orice ordine. Cazurile trebuie sa fie toate diferite.

Instructiunea break declanseaza o iesire imediata din switch. Deoarece cazurile servesc doar ca etichete, dupa ce codul unui caz a fost executat, executia continua spre urmatoarea instructiune daca nu nu luati o actiune explicita spre a iesi. Break si return sunt modurile cele mai uzuale de a parasi o instructiune switch. O instructiune switch poate fi deasemenea folosita si pentru a forta o iesire imediata dintr-o bucla while, for sau do, asa cum vom discuta mai departe in acest capitol.

Exercitiul 3.1. Scrieti o functie expand(s, t) care converteste caracterele de tipul lui "linie noua" si "tab" in secvente escape vizibile de tipul "\n" si "\t" in timp ce se copiaza sirul s in sirul t. Folositi instructiunea switch.

3.5. Bucle - While si For

Am intilnit deja buclele while si for. In

```
while (expresie)
instructiune
```

"expresie" este evaluata. Daca ea este nenula, "instructiune" este executata si "expresie" este reevaluata. Acest ciclu continua atita timp cit "expresie" nu este zero, iar cind ea devine zero executia se reia de dupa "instructiune".

Instructiunea for:

```
for (expr1; expr2; expr3)
instructiune
```

este echivalenta cu

```
expr1;
while (expr2) {
instructiune
expr3;
}
```

Din punct de vedere gramatical, cele trei componente ale unei bucle for sunt expresii. In majoritatea cazurilor, expr1 si expr3 sunt asignari sau apeluri de functii iar expr2 este o expresie relationala. Oricare din cele trei parti poate fi omisa, cu toate ca punct-virgula corespunzatoare trebuie sa ramina. Daca expr1 sau expr3 este lasata afara, i nu mai este incrementat. Daca testul, expr2 nu este prezent, el este luat ca fiind permanent adevarat, asa incit:

```
for (;;) {
...
{
```

este o bucla infinita, si probabil de spart cu alte mijloace (ca de exemplu, un break sau return). Folosirea lui while sau a lui for este in mare masura un subiect de gust. De exemplu, in:

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
; /* sari caracterele de spatiere */
```

nu exista nici o initializare sau reinitializare, asa ca while pare cea mai naturala. Bucla for este clar superioara atunci cind exista o simpla initializare si reinitializare, deoarece ea pastreaza instructiunile de control al buclei impreuna si la loc vizibil in virful buclei. Acest lucru este cel mai evident in:

```
for (i = 0; i < N; i++)
```

care este varianta in C pentru prelucrarea primelor N elemente dintr-un tablou.

Un operator final in limbajul C este virgula "," care isi gaseste adesea utilizare in instructiunea for. O pereche de expresii separate printr-o virgula este evaluata de la stinga spre dreapta si tipul si valoarea rezultatului sunt tipul si valoarea operandului din dreapta. Astfel, intr-o instructiune for este posibil sa plasam expresii multiple in parti variate, de exemplu sa prelucram doi indici in paralel. Acest lucru este ilustrat de functia reverse(s) care inverseaza pe loc un sir s.

```
reverse(s) /* inverseaza pe loc sirul s */
char s[];
{
int c, i, j;
for (i = 0, j = strlen(s) - 1; i < j; i++, j--) {
c = s[i];
s[i] = s[j];
s[j] = c;
}
}
```

Virgulele care separa argumentele functiilor, variabilele din declaratii, etc nici nu sunt operatori "virgula" si nu garanteaza evaluarea de la stinga la dreapta.

Exercitiul 3.2. Scrieti o functie expand(s1, s2) care expandeaza notatiile scurte de tipul a-z in sirul s1 in lista echivalenta si completa abc....xyz in s2. Sint permise litere mari si mici si cifre; sa fiti pregatiti sa tratati si cazuri de tipul a-b-c si a-z0-9 si -a-z. (O conventie utila este aceea ca "-"la inceput este considerat ca atare).

3.6. Bucle Do - While

Buclele while si for impartasesc atributul de testare a conditiei de terminare la inceputul buclei mai degraba decit la sfirsitul ei, asa cum am discutat in Capitolul 1. Al treilea tip de bucle in C - bucla do-while - testeaza conditia la sfirsit, dupa ce a executat intreg corpul buclei; corpul este executat cel putin o data. Sintaxa ei este

```
do
```

```
instructiune
while (expresie);
```

"Instructiune" este executata si apoi "expresie" este evaluata. Daca este adevarata, "instructiune" se executa din nou, s.a.m.d. Daca "expresie" devine falsa, bucla se termina.
Asa cum este de asteptat, bucla "do-while" este folosita mai putin decit while si for, probabil 5% din totalul de folosire a buclelor.

Exercitiul 3.3. In reprezentarea numerelor ca si complemente fata de 2 versiunea noastra pentru itoa nu functioneaza pentru numarul negativ cel mai mic, adica pentru valoarea lui n egala cu -(2 la puterea dimensiune cuvint-1). Explicati de ce. Modificati functia pentru a functiona corect si pentru aceasta valoare, indiferent de calculatorul pe care se executa.

Exercitiul 3.4. Scrieti o functie analoaga itob(n, s) care converteste intregii fara semn n intr-o reprezentare binara pe caracter in s. Scrieti itoh, care converteste un intreg intr-un numar haxazecimal.

Exercitiul 3.5. Scrieti o versiune a lui itoa care accepta trei argumente in loc de doua. Al treilea argument este un cimp de lungime minima; numarul convertit trebuie completat cu blancuri la stinga, daca e necesar, pentru a se inscrie in cimpul dat.

3.7. Break

Adesea este convenabil sa controlam iesirile din bucle altfel decit testind conditia la inceputul sau sfirsitul buclei. Instructiunea break ofera o iesire mai devreme din for, while, do si switch. O instructiune break face ca bucla (sau switch-ul) cea mai din interior sa se termine imediat. Urmatorul program sterge blancurile si taburile de la sfirsitul fiecarei linii de intrare, folosind un break pentru a iesi din bucla la (primul) cel mai din dreapta caracter nonblanc sau nontab

```
#define MAXLINE 1000 ;
main() /* sterge caracterele albe de la sfirsitul liniei */
{
 int n;
 char line[MAXLINE];
 while ((n = getline(line, MAXLINE)) > 0) {
 while( --n > 0)
 if (line[n] != ' ' && line[n] != '\t'
 && line[n] != '\n')
 break;
 line[n+1] = '\0';
 printf("%s\n", line);
 }
}
```

getline returneaza lungimea liniei. Bucla while din interior incepe cu ultimul caracter al lui line (sa ne amintim ca --n decrementeaza pe n inainte de a-i folosi valoarea) si cauta inapoi primul caracter care nu este blanc, tab sau

(newline) linie noua. Bucla este sparta cind este gasit unul din acestea sau cind n devine negativ (adica atunci cind intreaga linie a fost analizata). Ar trebui sa verificati ca este corect si in cazul in care linia este formata numai din caractere albe (de spatiere). O alternativa la break consta in a pune testul chiar in bucla:

```
while ((n = getline(line, MAXLINE)) > 0) {
 while (--n >= 0
 && (line[n] == ' ' || line[n] == '\t' || line[n] == '\n'))
 ;
 ...
}
```

Aceasta este inferioara versiunii precedente, deoarece testul este mai greu de inteles. Testele care necesita un amestec de && ,||,! sau paranteze sunt in general interzise.

3.8. Continue

Instructiunea continue este legata de break, dar mult mai putin folosita; ea face sa inceapa urmatoarea iteratie a buclei (while, for, do). In cazul lui while si do aceasta inseamna ca partea de test se executa imediat; in cazul lui for, controlul se trece la faza de reinitializare. (continue se aplica numai la bucle, nu si la switch. Un continue inauntrul unui switch dintr-o bucla declanseaza urmatoarea iteratie a buclei. Ca exemplu, fragmentul urmator prelucreaza numai elementele pozitive dintr-un tablou a; valorile negative sunt sarite:

```
 for (i = 0; i < N; i++) {
 if (a[i] < 0) /* sari elementele negative */
 continue;
 .../* prelucreaza elementele pozitive */
 }
```

Instructiunea continue este folosita adesea cind partea din bucla care urmeaza este complicata, astfel ca inversind un test si incluzind inca un nivel, ar imbrica programul si mai mult.

Exercitiul 3.6. Scrieti un program care copiaza intrarea in iesire, cu exceptia ca el tipareste o singura data o linie dintr-un grup de linii adiacente identice. (Aceasta este o versiune simpla a utilitarului UNIX uniq.)

3.9. Goto-uri si etichete

Limbajul C ofera instructiunea - de care se poate abuza oricit goto si etichete pentru ramificare. Formal, goto nu este necesara niciodata si in practica este aproape intodeauna usor sa scriem cod fara ea. Cu toate acestea, va sugeram citeva situatii in care goto isi poate gasi locul. Cea mai obisnuita folosire este aceea de a abandona prelucrarea in anumite structuri puternic imbricate, de exemplu de a iesi afara din doua bucle deodata. Instructiunea break nu poate fi folosita deoarece ea paraseste numai bucla cea mai din interior. Astfel:

```
 for (...)
 for (...) {
 ...
```

```
if (dezastru)
goto error;
}
...
error:
descurca problema
```

Aceasta organizare este manevrabila daca codul de minuire a erorii este netrivial si daca erorile pot apare in locuri diferite. O eticheta are aceeasi forma ca si un nume de variabila si este urmata de doua puncte. Ea poate fi atasata oricarei instructiuni dintr-o aceeasi functie ca si goto. Ca un alt exemplu, sa consideram problema gasirii primului element negativ dintr-un tablou bidimensional. (Tablourile multidimensionale sunt discutate in Capitolul 5). O posibilitate este:

```
for (i = 0; i < N; i++)
for (j = 0; j < M; j++)
if (v[i][j] < 0)
goto found;
/* nu s-a gasit */
...
found:
/* s-a gasit la pozitia i,j */
...
```

Codul implicind un goto poate fi scris intodeauna fara goto, chiar daca pretul pentru aceasta este o variabila suplimentara, sau teste repetate. De exemplu, cautarea in tablou devine:

```
found = 0;
for (i = 0; i < N && !found; i++)
for (j = 0; j < M && !found; j++)
found = v[i][j] < 0;
if (found)
/* a fost la i-1, j-1 */
...
else
/* nu a fost gasit */
...
```

Cu toate ca nu suntem dogmatici in privinta subiectului, se pare ca e adevarat ca instructiunea goto ar trebui folosita cu economie, daca nu chiar deloc.