The C Programming Language

Brian W. Kernighan
Dennis M. Ritchie

C O N T I N U T

P R E F A T A

C este un limbaj de programare cu scop general ale carui caracteristici sint economia de expresie, structuri moderne de control al fluxului si de date, precum si un set bogat de operatori. C nu este un limbaj de nivel "foarte inalt", nici "mare", si nu este specializat vreunei arii particulare de aplicatii. Dar absenta in restrictii si generalitatea sa il fac mai convenabil si mai de efect pentru mai multe scopuri decit limbaje presupuse mai puternice.

C a fost la inceput proiectat si implementat pe sistemul de operare UNIX pe DEC PDP11 de catre Dennis Ritchie. Sistemul de operare, compilatorul C si in mod esential, toate programele de aplicatii ale lui UNIX (inclusiv software-ul folosit pentru a pregati cartea aceasta) sint scrise in C. Compilatoare de C exista deasemenea si pe mai multe alte calculatoare, intre care IBM System/370 Honeywell 6000 si Interdata 8/32. C nu este legat de nici un hardware sau calculator anumit si e simplu de scris programe care se pot executa fara nici o modificare pe diferite calculatoare care au limbajul C implementat.

Aceasta carte are drept scop sa-l ajute pe cititor sa invete sa programeze in C. Ea contine o initiere, pentru ca noii utilizatori sa poata incepe cit mai repede posibil, capitole separate pentru fiecare caracteristica majora, si un manual de referinta. Marea parte a textului nu se bazeaza atit pe expunerea de reguli si propozitii cit pe citirea, scrierea si revizuirea de exemple. In cea mai mare parte exemplele sint programe reale si sint complete si nu fragmente izolate. Toate exemplele au fost testate direct din text, care este intr-o forma citibila pe calculator. Pe linga faptul ca am aratat cum se utilizeaza efectiv limbajul, am incercat in plus, acolo unde era posibil, sa-l ilustram cu algoritmi utili si cu principii de bun stil in programare si proiectare sanatoasa.

Aceasta carte nu este un manual introductiv de programare. Ea presupune anumite familiaritati cu conceptele de baza din programare, ca variabile, instructiuni de asignare, bucle, functii. Cu toate acestea, un programator novice va fi in stare sa citeasca cartea si sa-si insuseasca limbajul, chiar daca ajutorul unui coleg cu experienta mai mare i-ar usura munca foarte mult. In experienta noastra, C s-a dovedit un limbaj placut, expresiv si adaptabil pentru o mare varietate de programe. Este usor de invatat si "se poarta bine" pe masura ce experienta in programare cu el creste. Speram ca aceasta carte va va ajuta sa-l folositi bine.

Criticile si sugestiile multor prieteni si colegi au imbogatit

si imbunatatit mult aceasta carte si au marit placerea noastra de a o scrie. In particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin si Larry Rosler au citit cu totii multiplele versiuni cu grija. Sintem asemenea indatorati lui Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Halley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson si Peter Weinberger pentru comentariile pline de ajutor in diferitele stadii si lui Mike Lesk si Joe Ossanna pentru asistenta nepretuita in tiparirea lucrarii.

Brian W. Kernighan

Dennis M. Ritchie

CAPITOLUL 0. I N T R O D U C E R E

C este un limbaj de programare cu scop general. El este puternic legat de sistemul UNIX, deoarece a fost dezvoltat pe acest sistem si deoarece UNIX-ul si software-ul sau sint scrise in C. Cu toate acestea, limbajul nu este legat de un anume sistem de operare sau calculator; si, desi a fost numit "limbaj de programare sistem", deoarece este util in scrierea sistemelor de operare, el a fost folosit la fel de bine in scrierea de programe importante ce trateaza probleme numerice, prelucrari de texte sau baze de date.

C este un limbaj relativ "de nivel inferior". Aceasta caracterizare nu este peiorativa; ea inseamna pur si simplu ca C opereaza cu aceeasi clasa de obiecte cu care lucreaza majoritatea calculatoarelor, si anume caractere, numere si adrese. Acestea pot fi combinate si prelucrate cu operatori aritmetici si logici implementati pe actualele calculatoare.

C nu poseda operatii pentru a prelucra direct obiecte compuse, cum ar fi siruri de caractere, multimi, liste sau tablouri considerate ca un intreg. Nu exista nici o analogie, de exemplu, cu operatiile din PL/1 care manipuleaza un intreg tablou sau sir. Limbajul nu defineste nici o alta facilitate de alocare de memorie in afara de definitiile statice si de lucrul cu stiva folosite de variabilele locale ale functiilor; nu exista colectii reziduale sau de gramezi ca in Algol 68. In sfirsit, limbajul C in sine nu are facilitati de intrare-iesire: nu exista instructiuni READ sau WRITE si nici metode de acces la fisiere, "cablate" in limbaj. Toate aceste mecanisme de nivel inalt trebuiesc facute prin apeluri explicite de functii.

In mod similar, C ofera numai constructii directe, liniare de control al fluxului: teste, bucle, grupari, si subprograme, insa nu multiprogramare, operatii paralele, sincronizari sau corutine.

Cu toate ca absenta acestor caracteristici ar parea o grava deficienta ("Vrei sa spui ca trebuie sa apelez o functie pentru a compara doua siruri de caractere ?"), pastrarea limbajului la o dimensiune modesta a adus beneficii reale. Deoarece C este relativ mic, el poate fi descris intr-un spatiu redus si invatat repede. Un compilator pentru C poate fi simplu si compact. Compilatoarele sint deasemenea usor de scris; folosind tehnologia curenta, ne putem astepta la un timp de citeva luni pentru scrierea unui compilator nou si sa avem surpriza ca

80% din codul noului compilator este comun cu cele existente. Aceasta dovedeste marele grad de mobilitate a limbajului. Deoarece tipurile de date si structurile de control posedate de C sint suportate de majoritatea calculatoarelor existente, biblioteca necesara executiei [run - time] necesara pentru a implementa programele independente este minuscula. Pe PDP-11, de exemplu, ea contine numai rutinele pentru inmultirea si impartirea pe 32 de biti si subrutinele ce realizeaza secventele de inceput si de sfirsit. Desigur, fiecare implementare poseda o biblioteca cuprinzatoare si compatibila de functii pentru a indeplini functiile de I/O, a trata sirurile si operatiile de alocare de memorie dar, deoarece se apeleaza numai explicit, poate fi evitata daca e nevoie; ea poate fi scrisa portabil chiar in C.

Din nou, deoarece limbajul reflecta capacitatile calculatoarelor curente, programele C tind sa fie suficient de eficiente astfel ca nu exista nici o constringere pentru a le scrie in limbajul de asamblare. Cel mai evident exemplu in acest sens este chiar sistemul de operare UNIX, care este scris aproape in intregime in C. Din cele 13000 de linii de cod ale sistemului, numai aproximativ 800 de linii de la nivelul cel mai de jos sint scrise in limbajul de asamblare. In plus, software-ul pentru toate aplicatiile UNIX esentiale este scris in C; marea majoritate a utilizatorilor UNIX (inclusiv unul din autorii acestei carti) nici macar nu cunosc limbajul de asamblare al lui PDP-11.

Cu toate ca C se potriveste cu caracteristicile multor calculatoare, el este independent de arhitectura oricarui calculator particular si astfel, cu putina grija, este usor a scrie programe "portabile", adica programe care pot fi rulate fara modificari pe varietate de calculatoare. In mediul nostru este deja un fapt obisnuit ca programele dezvoltate pe UNIX sa fie transportate pe sistemele Honeywell, IBM si Interdata. In realitate, compilatoarele de C si suportul de executie pentru aceste patru tipuri de calculatoare sint mai compatibile decit versiunile presupuse standard ANSI pentru FORTRAN. Sistemul de operare UNIX insusi se executa acum atit pe PDP-11 cit si pe Interdata 8/32. In afara programelor care, in mod necesar, sint intrucitva dependente de tipul de calculator, ca: asamblorul, compilatorul, depanatorul - software-ul scris in C este identic pe ambele calculatoare. Chiar in cadrul sistemului de operare, 7000 de linii de cod, in afara suportului pentru limbajul de asamblare si handlerelor dispozitivelor de I/O este identic in proportie de 95%.

Pentru programatorii familiari cu alte limbaje, se poate dovedi util sa mentionam citeva aspecte istorice, tehnice si filosofice legate de C, pentru contrast si comparatie.

Multe din cele mai importante idei din C isi au radacina in limbajul -de acum suficient de batrin, dar inca viabil - BCPL, dezvoltat de Martin Richards. Influenta lui BCPL asupra lui C apare indirect prin limbajul B, care a fost scris de Ken Thompson

in 1970 pentru primul sistem UNIX pe un PDP-7.

Cu toate impartaseste multe caracteristici esentiale cu BCPL, limbajul C nu este in nici un sens un dialect al acestuia. Limbajele BCPL si B sint limbaje "fara tipuri"[typeless]: singurul tip de data este cuvintul masina, si accesul la alte tipuri de obiecte se face cu operatori speciali sau apeluri de functii. In C obiectele (datele) fundamentale sint caracterele, intregii de diferite dimensiuni si numerele flotante. In plus, exista o ierarhie de tipuri de date derivate create cu pointeri, tablouri, structuri, uniuni si functii.

Limbajul C poseda constructiile fundamentale pentru controlul fluxului necesare pentru programele bine structurate: grupare de instructiuni; luare de decizii ("if"); buclare cu test de terminare la inceput ("while", "for") sau la sfirsit ("do"), selectare a unui caz dintr-o multime de cazuri posibile ("switch"). (Toate acestea erau valide si in BCPL, chiar daca cu o sintaxa diferita; acest limbaj anticipa voga pentru "programare structurata" cu mai multi ani inainte).

Limbajul C foloseste pointeri si are abilitatea de a face aritmetica cu adrese. Argumentele functiilor sint pasate copiind valoarea argumentului si este imposibil pentru functia apelata sa modifice argumentul real din apelant. Cind se doreste sa se obtina un "apel prin referinta", se trimite explicit un pointer, iar functia poate modifica obiectul la care puncteaza pointerul. Numele de tablouri sint trimise ca locatie a originii tabloului, asa ca argumentele tablouri sint efectiv apeluri prin referinta.

Orice functie poate fi apelata recursiv si variabilele sale sint tipic "automate" sau create nou cu fiecare invocare. Definitiile de functii nu pot fi imbricate dar variabilele pot fi declarate in maniera de bloc structurat. Functiile unui program C pot fi compilate separat. Variabilele pot fi interne unei functii, externe dar cunoscute numai intr-un singur fisier sursa, sau complet globale. Variabilele interne pot fi automate sau statice. Variabilele automate pot fi in registre pentru eficienta marita, dar declaratia de registru este numai interna compilatorului si nu se refera la vreun registru specific al calculatorului.

Limbajul C nu este un limbaj puternic tipizat in sensul lui PASCAL sau Algol - 68. El este relativ liberal in conversia de date, cu toate ca nu converteste automat tipurile de date cum ar fi PL/1. Compilatoarele existente nu poseda verificare la executie a indicilor elementelor de tablouri, tipurilor argumentelor, etc.

Pentru acele situatii in care se cere o puternica verificare a tipului, se foloseste o versiune separata a compilatorului. Acest program se numeste "lint" deoarece triaza bitii dubiosi of fluff dintr-un program. El nu genereaza cod, verifica numai

foarte strict multe aspecte ale programelor asa cum pot fi verificate la compilare si la incarcare. El detecteaza nepotrivirile de tip, folosirea inconsistenta a argumentelor, variabilele nefolosite sau aparent neinitializate, dificultatile potentiale de portabilitate si alte asemenea aspecte. Programele care trec cu bine aceasta verificare, cu citeva exceptii, se elibereaza de erorile de tip la fel de complet ca si, de exemplu, programele scrise in Algol 68. Vom mentiona si alte capacitati ale lui "lint" atunci cind ni se va ivi ocazia.

In fine, limbajul C, ca si alte limbaje, are defectele sale. Unii din operatori au o pondere gresita; o parte sau parti ale sintaxei ar fi putut fi mai bune; exista mai multe versiuni ale limbajului diferind foarte putin una de alta. Cu toate acestea, limbajul C s-a dovedit a fi un limbaj extrem de eficace si expresiv pentru o larga varietate de aplicatii de programare.

Restul cartii este organizat dupa cum urmeaza. Capitolul 1 este o initiere in partea centrala a limbajului C. Scopul lui este sa faca cititorul sa inceapa sa programeze in C cit mai repede posibil, deoarece noi credem puternic ca singurul mod de a invata un limbaj nou este de a se scrie programe in el. "Initierea" presupune o cunoastere a elementelor de baza ale programarii; nu se dau explicatii despre calculatoare, compilatoare si nici despre semnificatia unor expresii ca, de exemplu, n = n + 1. Cu toate ca am incercat pe cit posibil sa aratam tehnici de programare utile, cartea aceasta nu se vrea de referinta in structuri de date si algoritmi; cind am fost fortati sa alegem, ne-am concentrat asupra limbajului.

Capitolele 2-6 discuta diferite aspecte ale limbajului C mai detaliat si mai formal decit o face Capitolul 1, cu toate ca accentul cade tot pe exemple de programe utile complete si nu pe fragmente izolate. Capitolul 2 se ocupa cu tipurile de date fundamentale, operatorii si expresiile. Capitolul 3 trateaza controlul fluxului: if-else, while, for, etc. Capitolul 4 acopera functiile si structura programului -variabile externe, reguli de domeniu, si asa mai departe. Capitolul 5 discuta pointerii si aritmetica adreselor. Capitolul 6 contine detalii despre structuri si uniuni.

Capitolul 7 descrie biblioteca C standard de I/O care asigura o interfata bisnuita cu sistemul de operare. Aceasta biblioteca de I/O este tratata pe oate calculatoarele care suporta limbajul C, asa ca programele care o folosesc entru intrari, iesiri si alte functii sistem pot fi mutate de pe un sistem pe altul in principal fara modificari.

Capitolul 8 descrie interfata intre programele C si sistemul de operare UNIX, concentrindu-se asupra operatiilor de intrare/iesire, sistemului de fisiere si portabilitatii. Cu toate ca acest capitol este oarecum specific pentru UNIX, programatorii care nu folosesc acest sistem pot gasi si aici un material util, inclusiv o privire asupra modului in care

este implementata o versiune a bibliotecii standard, precum si sugestii pentru a obtine un cod portabil.

Anexa A contine manualul de referinta al limbajului C. Acesta este declaratia "oficiala" a sintaxei si semanticii lui C si (exceptind compilatoarele proprii) arbitrul final al oricarui ambiguitati sau omisiuni din capitolele precedente.

Deoarece C este un limbaj in dezvoltare care exista pe o varietate de calculatoare, anumite parti din aceasta carte pot sa nu mai corespunda cu stadiul curent de dezvoltare pentru un sistem particular. Am incercat sa evitam aceste probleme si sa atentionam asupra potentialelor dificultati. Cind am fost in dubiu, am ales in general descrierea situatiei PDP-11 UNIX, care este mediul de lucru al majoritatii programatorilor in C. Anexa A contine deasemenea diferentele de implementare pentru C pe sistemele majore pe care el poate fi gasit.

CAPITOLUL 1. I N I T I E R E

Sa incepem cu o introducere rapida in C. Scopul nostru este sa preezentam elementele esentiale ale limbajului in programe reale, fara insa a ne impotmoli in detalii, reguli formale si exceptii. In acest punct al expunerii nu incercam sa fim completi si nici macar foarte precisi (mentionam totusi ca exemplele vor sa fie corecte). Dorim sa va aducem cit mai repede posibil in punctul in care veti fi capabili sa scrieti programe utile si, pentru aceasta, ne-am concentrat asupra fundamentelor: variabile si constante, aritmetica, controlul fluxului, functii si rudimente de operatii de I/O. Am lasat deoparte intentionat din acest capitol acele caracteristici ale limbajului C care sint de importanta vitala in scrierea programelor mai mari. Acestea includ pointerii, structurile, majoritatea din bogatul set de operatori ai lui C, anumite instructiuni de control al fluxului si o multime de detalii.

Acest mod de abordare are neajunsurile lui, desigur. Cel mai notabil este acela ca povestea completa a caracteristicilor oricarui limbaj de programare nu este gasita intr-un singur loc, iar o initiere in el, fiind scurta, poate induce in eroare. Deoarece exemplele nu pot folosi intreaga putere a lui C, ele nu sint atit de concise si de elegante pe cit ar putea fi. Am incercat sa minimalizam aceste efecte, dar fiti atenti!

Un alt neajuns este acela ca in capitolele urmatoare vom repeta in mod necesar cite ceva din acest capitol. Speram ca aceasta repetitie va va ajuta mai mult decit va va plictisi.

In orice caz, programatorii experimentati vor fi capabili sa extrapoleze din materialul din acest capitol propriile lor nevoi de programare. Incepatorii vor putea scrie mici programe, similare celor prezentate de noi. Ambele grupe pot folosi acest capitol drept cadru pentru descrierile riguroase care incep cu Capitolul 2.

1.1 Sa incepem

Singurul mod de a invata un nou limbaj de programare este de a scrie programe in el. Primul program pe care-l vom scrie este acelasi pentru toate limbajele:

Tipariti cuvintele

```
hello, world
```

Acesta este primul obstacol; pentru a sari peste el, trebuie sa fiti in stare sa creati undeva textul program, sa-l compilati cu succes, sa-l incarcati, sa-l executati si sa aflati textul tiparit acolo unde este iesirea calculatorului dumneavoastra. In C, programul pentru a tipari "hello, world" este:

```
main ()
{
     printf("hello, world\n");
}
```

Cum ruleaza acest program, depinde de sistemul pe care-l folositi, Drept exemplu specific, pe sistemul de operare RSX, trebuie sa creati acest program sursa intr-un fisier al carui nume se termina in ".C", de exemplu "hello.C" apoi sa-l compilati cu comenzile:

```
>cc hello
>as hello
```

Daca n-ati gresit nimic, de exemplu sa fi uitat un caracter sau sa fi inversat doua caractere, compilarea se va desfasura silentios si va produce un fisier obiect numit "hello.obj". Lansindu-l in executie dupa linkeditare cu comenzile

```
>tkb hello=hello,lb:[1,1]clib/lb
>run hello
```

va produce

```
hello, world
```

ca iesire a sa. Pe alte sisteme, regulile vor fi diferite; verificati-le cu expertul local.

Exercitiul 1.1. Executati acest program pe sistemul dumneavoastra. Incercati sa vedeti ce mesaje de eroare obtineti, lasind la o parte parti din program.

Si acum citeva explicatii despre programul insusi. Un program C, oricare i-ar fi marimea, consta din una sau mai multe "functii" care specifica operatiile efective de calculat care trebuiesc

facute. Functiile din C sint similare cu functiile si subrutinele dintr-un program Fortran sau cu procedurile din PL/1, Pascal,etc. In exemplul nostru, "main" este o astfel de functie. In mod normal aveti libertatea de a da functiilor ce nume doriti, dar "main" este un nume special - programul dumneavoastra se va executa de la inceputul lui "main". Aceasta inseamna ca fiecare program trebuie sa aibe un "main" undeva, ca "main" va invoca in mod obisnuit alte functii pentru a-si realiza scopul, unele venind din acelasi program iar altele din biblioteci ce contin functii scrise anterior.

O metoda de a comunica date intre functii este prin argumentele functiilor. Parantezele care urmeaza dupa numele functiei includ lista de argumente. In cazul nostru, "main" este o functie fara argumente ceea ce se indica prin "()". Acoladele "{ }" includ instructiunile care alcatuiesc functia. Ele sint analoage lui "DO-END" din PL/1 sau lui "begin-end" din ALGOL, PASCAL, etc. O functie este apelata prin nume, urmate de o lista de argumente in paranteze.
Nu exista instructiunea CALL ca in FORTRAN sau PL/1. Parantezele trebuie sa fie prezente chiar daca nu exista argumente.

Linia care spune:

```
printf("hello, world\n");
```

este un apel de functie, care cheama o functie numita "printf" cu argumentul ("hello, world\n"). "printf" este o functie din biblioteca care tipareste pe terminal (daca nu este specificata o alta destinatie). In acest caz, ea tipareste sirul de caractere care alcatuiesc argumentul.

O secventa alcatuita din orice numar de caractere cuprinse intre doua ghilimele "..." se numeste sir de caractere sau constanta sir. Pentru moment, singura folosire a sirurilor de caractere va fi ca argumentele pentru "printf" si alte functii.

Secventa "\n" din sir este notatia din C pentru caracterul "linie noua", care, cind este tiparit, avanseaza cursorul terminalului la marginea din stinga a urmatoarei linii. Daca uitati "\n" (un experiment care merita facut), veti observa ca iesirea dumneavoastra nu se termina cu o linie noua. Singurul mod de a avea caracterul "linie noua" in "printf" este "\n" ca argument. Daca incercati ceva de tipul

```
printf("hello, world
");
```

compilatorul C va va tipari un diagnostic neprietenos despre ghilimele absente.

"printf" nu furnizeaza o linie noua in mod automat, asa ca apelurile multiple pot fi folosite pentru a tipari o linie pe etape. Primul nostru program poate fi scris la fel de bine si

astfel:

```
main()
{
        printf("hello, ");
        printf("world");
        printf("\n");
}
```

pentru a produce o iesire identica.
Sa notam ca "\n" reprezinta un singur caracter. O "secventa escape" ca de exemplu "\n" este in general un mecanism extensibil pentru reprezentarea caracterelor greu de obtinut sau invizibile. Printre alte secvente escape, limbajul C poseda: \t pentru tab, \b pentru backspace, \" pentru apostrof dublu si \\ pentru backspace.

Exercitiul 1.2. Experimentati sa vedeti ce se intimpla cind sirul argument din "printf" contine "\x" un x este un caracter oarecare care nu a fost listat mai sus.

1.2. Variabile si aritmetica

Urmatorul program tipareste tabela de temperaturi Fahrenheit si echivalentele lor in centigrade sau grade Celsius, folosind formula: C = (5 / 9) * (F - 32).

```
  0   -17.8
 20    -6.7
 40     4.4
 60    15.6
    ...
260   126.7
280   137.8
300   148.9
```

Iata acum si programul:

```
/* Print Fahrenheit-Celsius table
   for f = 0, 20, ..., 300 */
main()
{
        int lower, upper, step;
        float fahr, celsius;
        lower = 0; /* lower limit of temperature table */
        upper = 300; /* upper limit */
        step  = 20;  /* step size */
        fahr  = lower;
        while (fahr <= upper) {
          celsius = (5.0 / 9.0) * (fahr - 32.0);
          printf("%4.0f %6.1f\n", fahr, celsius);
          fahr = fahr + step;
        }
}
```

Primele doua linii :

```
        /* Print Fahrenheit-Celsius table
           for f = 0, 20, ..., 300 */
```

sint un comentariu, care in acest caz explica pe scurt ce face programul. Orice caractere cuprinse intre "/*" si "*/" sint ignorate de compilator; ele pot fi folosite liber pentru a face programul mai usor de inteles. Comentariile pot apare oriunde poate aparea un spatiu sau o linie noua.

In limbajul C, toate variabilele trebuie declarate inainte de a fi folosite, deobicei la inceputul liniei, inaintea oricarei instructiuni executabile. Daca veti uita o declaratie, veti primi un diagnostic de la compilator. O declaratie consta dintr-un "tip" si o lista de variabile care au acel tip, ca in:

```
int lower, upper, step;
float fahr, celsius;
```

Tipul "int" implica faptul ca variabilele listate sint intregi; "float" denota virgula mobila, adica numere care pot avea parte fractionara. Precizia atit pentru "int" cit si pentru "float" depinde de calculatorul pe care-l folositi. Pentru PDP-11, de exemplu, un "int" este un numar cu semn de 16 biti, adica un numar cuprins intre -32768 si +32767 . Un numar "float" este o cantitate ce se reprezinta pe 32 de biti ceea ce revine la aproximativ sapte cifre semnificative, cu magnitudinea cuprinsa aproximativ intre 10 ^ -38 si 10 ^ +38. Capitolul 2 da olista a acestor marimi pentru alte calculatoare.

Pe linga tipurile "int" si "float", limbajul C poseda si alte tipuri de date fundamentale :

```
"char"   caracter - un singur octet
"short"  intreg scurt
"long"   intreg lung
"double" numar flotant dubla precizie
```

Marimea acestor obiecte este deasemenea dependenta de calculator; detalii se dau in Capitolul 2. Exista deasemenea "tablouri", "structuri" si "uniuni" de asemenea tipuri de baza, "pointeri" la ele si "functii" care le returneaza si cu toate ne vom intilni in aceasta carte.

Calculul efectiv in programul de conversie temperatura incepe cu asignarile:

```
lower = 0;
upper = 300;
step  = 20;
fahr  = lower;
```

care seteaza variabilele pe valorile lor de start. Instructiu-

nile individuale se termina cu punct si virgula.
Fiecare linie a tabelei este calculata in acelasi fel, asa ca vom folosi o bucla care se repeta odata pe linie; acesta este scopul instructiunii "while":

```
while (fahr <= upper) {
      ...
}
```

Este testata conditia din paranteza. Daca ea este adevarata (fahr este mai mic sau egal cu upper), este executat corpul buclei (toate instructiunile incluse intre parantezele { si }). Apoi conditia este retestata si, daca este adevarata, corpul este executat din nou . Cind testul devine fals (fahr este mai mare decit upper), bucla se termina si executia continua cu instructiunea care urmeaza buclei. Deoarece in acest program nu mai exista alte instructiuni care sa succeada bucla, executia lui se termina.

Corpul unei bucle while poate fi alcatuit din mai multe instructiuni incluse intre acolade, ca in programul de mai sus sau dintr-o singura instructiune, fara paranteze, ca in exemplul de mai jos:

```
while(i<j)
        i = 2 * i;
```

In ambele cazuri, instructiunile controlate de "while" sint decalate cu un tab, asa ca se observa de la prima privire ce instructiuni se gasesc in interiorul buclei. Decalarea scoate in evidenta structura logica a programului. Cu toate ca limbajul C este destul de liberal in ceea ce priveste pozitionarea instructiunilor, o tabulare potrivita si folosirea spatiilor albe sint critice in scrierea programelor usor citibile si de catre altii decit autorul lor.

Recomandam scrierea unei singure instructiuni pe un rind si lasarea de spatii (uzual) in jurul operatorilor. Pozitia acoladelor este mai putin importanta; noi am ales unul dintre stilurile cele mai populare. Alegeti-va un stil care va convine, apoi folositi-l consecvent.

Temperatura celsius este calculata si asignata lui "celsius" prin instructiunea:

```
celsius = (5.0 / 9.0) * (fahr - 32.0);
```

Ratiunea pentru folosirea lui (5.0/9.0) in locul mai simplei 5/9 este aceea ca in C, ca si in multe alte limbaje, impartirea intreaga trunchiaza rezultatul, asa ca orice parte fractionara este eliminata. Astfel 5/9 este zero si tot asa ar fi fost toate temperaturile. Un punct zecimal intr-o constanta indica faptul ca aceasta este flotanta si deci 5.0/9.0 este 0.5555... ceea ce am si dorit. Am scris deasemenea 32.0 in loc de 32, chiar daca deoarece "fahr" este "float", 32 ar fi convertit automat in "float" inainte de scadere. Ca o problema de stil, este bine sa scriem

constantele flotante cu punct zecimal explicit chiar cind au valori intregi; aceasta accentueaza natura lor flotanta pentru cititorii umani si ne asigura ca si compilatorul va gindi in acelasi mod ca si noi.

Regulile detaliate pentru conversiile de intregi in flotante sint date in Capitolul 2. Acum sa notam doar ca asignarea

```
fahr = lower;
```

si testul

```
while (fahr <= upper)
```

vor lucra amindoua asa cum ne asteptam "int" este convertit in "float" inainte de executia operatiei.

Acest exemplu arata deasemenea putin mai multe despre modul de lucru al lui "printf". Ea este o functie de conversie de format cu scop general, care va fi descrisa complet in Capitolul 7. Primul sau argument este un sir de caractere ce se vor tipari, fiecare caracter % indicind argumentele (al doilea, al treilea) ce se va substitui si forma in care se vor tipari. De exemplu, in instructiunea

```
printf("%4.0f %6.1f\n", fahr, celsius);
```

specificatia de conversie "%4.0f" spune ca un numar flotant va fi tiparit intr-un spatiu de cel putin patru caractere, cu zero cifre dupa punctul zecimal. "%6.1f" descrie un alt numar care va ocupa cel putin sase spatii, cu o cifra dupa punctul zecimal, analog cu F6.1 din FORTRAN sau F(6,1) din PL/1. Parti din specificator pot fi omise: "%6f" arata ca numarul are o lungime de cel putin 6 caractere; "%.2f" cere doua pozitii dupa punctul zecimal, dar lungimea lui nu este supusa restric-tiilor; "%f" spune doar sa se tipareasca numarul ca flotant. "printf" recunoaste deasemenea: "%d" pentru intregi zecimali, "%o" pentru numere octale,"%x" pentru hexazecimale, "%c" pentru un caracter, "%s" pentru sir de caractere si "%%" pentru insusi semnul "%".

Fiecare constructie cu % in primul argument al lui "printf" face pereche cu al doilea, al treilea,... argument. Aceste perechi trebuie sa corespunda ca numar si tip, altfel veti avea surpriza unor rezultate lipsite de inteles.

Fiindca veni vorba, "printf" NU face parte din limbajul C. Nu exista definite in C intrari si iesiri. Nu e nimic magic in legatura cu "printf" ea este pur si simplu o functie utila care face parte din biblioteca standard de rutine care sint in mod normal accesibile programelor C. Cu scopul de a ne concentra asupra limbajului C, nu vom spune prea multe despre opera-tiile de I/O pina in Capitolul 7. In particular, vom amina descrierea intrarilor cu format pina atunci. Daca aveti de introdus numere, cititi descrierea functiei "scanf" din Capitolul

7, sectiunea 7.4; "scanf" este foarte asemanatoare cu "printf" atit doar ca ea citeste intrari in loc sa scrie iesiri.

Exercitiul 1.3. Modificati programul de conversie temperaturi pentru a scrie un antet la inceputul tabelei de conversie.

Exercitiul 1.4. Scrieti un program care sa tipareasca tabela corespunzatoare Celsius - Fahrenheit.

1.3. Instructiunea For

Asa cum probabil va asteptati, exista o multime de moduri pentru a scrie un program; haideti sa incercam o alta varianta a programului de conversie de temperatura :

```
main() /* Fahrenheit-Celsius table */
{
     int fahr;
     for (fahr = 0; fahr <= 300; fahr = fahr + 20)
     printf("%4d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
}
```

Aceasta va produce aceleasi rezultate dar, cu siguranta, arata altfel decit prima. O modificare esentiala este eliminarea majoritatii variabilelor; a ramas numai "fahr", declarata ca "int" (observati specificatorul "%d" in printf). Limitele inferioara si superioara si marimea pasului apar doar ca si constante in instructiunea "for", ea insasi o constructie noua, iar expresia care calculeaza temperatura Celsius apare acum ca al treilea argument din "printf" in loc de a fi o instructiune de asignare separata.

Aceasta ultima schimbare este un exemplu pentru o regula generala in C - in orice context in care este permisa folosirea valorii unei variabile de un anumit tip, se poate folosi o expresie de acel tip. Deoarece al treilea argument al lui "printf" trebuie sa fie o valoare flotanta pentru a se potrivi cu "%6.1f", orice expresie flotanta poate apare pe locul ei.

Instructiunea "for" este o bucla, o generalizare a lui "while". Daca o comparati cu "while", aceasta afirmatie va va fi clara. Ea contine trei parti separate prin punct si virgula. Prima parte

```
fahr = 0
```

se face o data, inainte ca bucla propriu-zisa sa inceapa. A doua parte este testul sau conditia care controleaza bucla:

```
fahr <= 300
```

Este evaluata aceasta conditie; daca ea este adevarata, este executat corpul buclei (la noi, o singura "printf"). Urmeaza apoi pasul de reinitializare

```
fahr = fahr + 20
```

care este executat si apoi conditia este reevaluata. Bucla se termina atunci cind conditia devine falsa. La fel ca si la instructiunea "while", corpul buclei poate fi alcatuit dintr-o singura instructiune sau dintr-un grup de instructiuni inclus intre acolade. Partile de initializare si reinitializare pot fi o singura expresie.

Alegerea intre "while" si "for" este arbitrara, bazata pe ceea ce ne pare noua a fi mai clar. Instructiunea "for" este potrivita in mod uzual pentru buclele in care initializarea si reinitia-lizarea sint instructiuni unice si logic inrudite deoarece este mai compacta decit "while" si pastreaza instructiunile de control al buclei intr-un singur loc si impreuna.

Exercitiul 1.5. Modificati programul de conversie temperatura pentru a tipari tabela in ordine inversa, adica de la 300 de grade la zero.

1.4. Constante simbolice

Vom face o observatie finala inainte de a parasi pentru todeauna programul de conversie de temperatura. E o practica proasta aceea de a inmorminta "numere magice" ca 300 sau 20, intr-un program; ele transmit putina informatie cuiva care va citi programul mai tirziu si este greu sa le modificam intr-o maniera sistematica. Din fericire, C poseda o modalitate de a evita astfel de numere magice. Cu ajutorul constructiei "#define", se pot defini la inceputul programului nume sau constante simbolice, care sint un sir particular de caractere. Dupa aceea, compilatorul va inlocui toate aparitiile nepuse intre ghilimele ale numelui, prin sirul corespunzator. Inlocui-rea efectiva a numelui poate fi orice text; ea nu se limiteaza la numere.

```
#define LOWER 0     /* lower limit of the table */
#define UPPER 300   /* upper limit */
#define STEP  20    /* step size */
main()       /* Fahrenheit-Celsius table */
{
   int fahr;
   for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
   printf("%4d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
}
```

Cantitatile LOWER, UPPER si STEP sint constante, asa incit ele nu apar in declaratii. Numele simbolice se scriu in mod normal cu litere mari, asa ca ele pot fi usor distinse de numele de variabile care se scriu cu litere mici. Sa notam ca la sfirsitul unei definitii NU se pune punct si virgula. Deoarece intreaga linie de dupa numele definit este substituita, in instructiunea "for" ar exista prea multe punct si virgule.

1.5. O colectie de programe utile

Vom considera in cele ce urmeaza o familie de programe inrudite pentru efectuarea de operatii simple asupra datelor alcatuite din caractere. Vom vedea ca multe programe sint doar versiuni extinse ale prototipurilor pe care le vom discuta aici.

Introducere si extragere de caractere

Biblioteca standard poseda functii pentru citirea si scrierea unui caracter la un moment dat. "getchar()" aduce urmatorul caracter de intrare de fiecare data cind este apelata si returneaza acel caracter ca si valoare a ei. Adica, dupa

```
c=getchar()
```

variabila "c" contine urmatorul caracter de intrare. Caracterele vin in mod normal de la terminal, dar aceasta nu ne intereseaza pina in Capitolul 7.

Functia "putchar(c)" este complementara lui "getchar()":

```
putchar(c)
```

tipareste continutul variabilei "c" pe un mediu de iesire, in mod normal, tot pe terminal. Apelurile la "putchar" si "printf" pot fi intercalate; iesirea va apare in ordinea in care s-au facut apelurile.

Ca si in cazul lui "printf", nu exista nimic special relativ la "getchar" si "putchar". Ele nu sint parti ale limbajului C, dar sint universal disponibile.

Copiere de fisiere

Date "getchar" si "putchar", veti putea scrie o cantitate surprinzatoare de cod util, fara a sti nimic despre operatiile de I/O. Cel mai simplu program este acela care-si copiaza intrarea in iesire, caracter cu caracter. Schitindu-l:

```
citeste_un_caracter
while (caracterul_nu_este_semnal_de_sfirsit_de_fisier)
        tipareste_caracterul_chiar_citit
        citeste_un_nou_caracter
```

Convertind aceasta in limbajul C, obtinem:

```
main()   /* copy input to output; 1st version */
{
        int c;
        c = getchar();
        while (c != EOF) {
                putchar(c);
                c = getchar(c);
        }
```

```
        }
```

Operatorul relational "!=" inseamna "diferit de".

Principala problema este detectarea sfirsitului de intrare. Prin conventie, "getchar" returneaza o valoare care nu este un caracter valid atunci cind intilneste sfirsitul intrarii; in acest mod, programele pot detecta cind s-au terminat intrarile. Singura complicatie, o neplacere serioasa de fapt este aceea ca exista doua conventii ce se folosesc uzual pentru valoarea sfirsitului de fisier. Noi am evitat aceasta folosind deobicei numele simbolic de EOF pentru aceasta valoare, oricare ar fi fost ea. In practica, EOF poate fi sau -1 sau 0, asa ca programul trebuie sa fie precedat de una din declaratiile de mai jos:

```
                #define EOF -1
```

sau

```
                #define EOF  0
```

pentru ca el sa lucreze corect. Folosind constanta simbolica EOF pentru a reprezenta valoarea pe care o ret/neaza getchar cind intilneste sfirsitul de fisier, ne asiguram ca numai un singur lucru din program depinde de valoarea numerica specificata.

De asemenea il declaram pe "c" ca fiind "int", nu "char", pentru ca el sa poata pastra valoarea pe care o returneaza "getchar". Cum vom vedea in Capitolul 2, aceasta valoare este normal un "int", deoarece ea trebuie sa fie capabila sa-l reprezinte si pe EOF in plus fata de toate char-urile posibile.

Programul de copiere ar putea fi de fapt scris mult mai concis de catre un programator experimentat in limbajul C. In C, o asignare ca

```
                c = getchar()
```

poate fi folosita intr-o expresie; valoarea sa este pur si simplu valoarea ce se asigneaza partii stingi. Daca asignarea unui caracter lui c se pune in partea de test a unui "while", programul de copiat fisiere poate fi scris astfel:

```
        main()  /* copy input to output; 2nd version */
        {
                int c
                while ((c = getchar()) != EOF)
                      putchar(c);
        }
```

Programul citeste un caracter, il asigneaza lui "c" si apoi testeaza daca acesta a fost semnalul de sfirsit de fisier. Daca nu a fost, corpul buclei "while" este executat, tiparindu-se caracterul si bucla se repeta. Cind semnalul de sfirsit de fisier este atins in fine, bucla "while" se termina, terminindu-se totodata si programul "main".

Aceasta versiune centralizeaza intrarile - nu mai apare decit

un singur apel la "getchar"- si restringe programul. Plasarea unei asignari intr-un test constituie unul din locurile unde C permite o concizie uluitoare. (Este posibil sa mergeti si mai departe, creind un cod impenetrabil, tendinta pe care noi incercam sa nu o incurajam).

Este important sa recunoastem ca parantezele ce includ asignarea sint absolut necesare. Ponderea lui "!=" este mai mare decit aceea a lui "=" ceea ce inseamna ca, in absenta parantezelor, testul relational "!=" va fi facut inaintea asignarii "=". Asa ca instructiunea

```
c = getchar() != EOF
```

este echivalenta cu

```
c = (getchar() != EOF)
```

Aceasta are un efect nedorit, prin setarea lui "c" pe 0 sau 1, dupa cum apelul lui "getchar " a intilnit sau nu sfirsitul de fisier. (Mai multe despre acestea se vor vedea in Capitolul 2).

Contorizarea caracterelor

Urmatorul program va contoriza caracterele; el este o mica elaborare a programului de copiere.

```
main()  /* count characters in input */
{
        long nc;
        nc = 0;
        while (getchar() != EOF)
             ++nc;
        printf("%ld\n",nc);
}
```

Instructiunea

```
++nc;
```

ne introduce un nou operator "++" care inseamna, increment cu 1. Se putea scrie si "nc = nc+1", dar "++nc" este mai concisa si adesea mai eficienta. Exista un operator corespunzator, "--" pentru decrementare cu 1. Operatorii "++" si "--" pot fi atit operatori prefix, cit si sufix ("nc++"); aceste doua forme au valori diferite in expresii asa cum se va arata in Capitolul 2, dar "++nc" si "nc++" il incrementeaza amindoi pe "nc".

Programul de contorizare de caractere acumuleaza numarul de caractere intr-o variabila "long" in loc de "int". La PDP-11, valoarea maxima pentru un intreg este 32767 si s-ar putea ca sa dam peste o depasire de contor daca-l declaram intreg; pe Honeywell si pe IBM, "long" si "int" sint sinonime si mult mai mari. Specificatorul de conversie "%ld" semnaleaza lui

"printf" ca argumentul corespunzator este un intreg "long".

Pentru a face fata la numere chiar si mai mari, se poate folosi o declaratie de "double" ("float" de lungime dubla). Vom folosi, deasemenea, instructiunea "for" in locul lui "while" pentru a ilustra un alt mod in scrierea buclei.

```
main()  /* count characters in input */
{
        double nc ;
        for (nc = 0; getchar() != EOF; ++nc)
            ;
        printf("%.0f\n", nc);
}
```

"printf" foloseste "%f" atit pentru "float" cit si pentru "double"; "%.0f" suprima tiparirea partii fractionare inexistente.

Corpul buclei "for" este in cazul acesta vid, deoarece toata munca este facuta in partile de test si reinitializare. Dar regulile gramaticale ale limbajului C pretind ca o instructiune "for" sa aiba un corp. Punctul si virgula ce apare izolat pe o linie, in mod tehnic o instructiune nula, este pus tocmai pentru a satisface aceasta cerere. Noi l-am pus pe o linie separata tocmai pentru a-l face mai vizibil.

Inainte de a parasi programul de contorizare caractere, sa observam ca daca intrarea nu contine nici un caracter, testul din "while" sau "for" esueaza la primul apel la getchar si deci rezultatul programului este zero, ceea ce este corect. Aceasta este o observatie importanta. Unul din lucrurile frumoase care se pot spune despre "while" si despre "for" este cela ca ele testeaza la inceputul buclei, inainte de a prelucra corpul buclei. Daca nu este nimic de facut, nimic nu se face, chiar daca aceasta inseamna ca nu se va parcurge corpul buclei nicio-data. Programele vor actiona inteligent atunci cind vor minui intrari de tipul "nici un caracter". Instructiunile "while" si "for" ne asigura ca vor face lucruri rezonabile si in conditii la limita.

Contorizarea liniilor

Urmatorul program contorizeaza liniile pe care le primeste ca intrare. Liniile de intrare se presupun a fi terminate cu un caracter "linie noua" \n adaugat cu sfintenie la fiecare linie scrisa.

```
main()  /* contorizarea liniilor in intrare */
{
        int c, nl;
        nl = 0;
        while ((c = getchar()) != EOF)
             if(c == '\n')
                    ++nl;
        printf("%d\n", nl);
```

```
        }
```

Corpul buclei "while" consta acum dintr-un "if",care la rindul ei controleaza incrementarea ++nl. Instructiunea "if" testeaza conditia din paranteza si, daca este adevarata, se executa instructiunea (sau grupul de instructiuni dintre acolade) care urmeaza. Am aliniat iarasi, ca sa aratam ce este controlat de cine (ce).

Semnul dublu de egal "==" este in C notatia pentru "este egal cu" (ca si .EQ. din FORTRAN). Acest simbol este folosit pentru a distinge testul de egalitate de egal simplu (=) folosit pentru asignare. Deoarece asignarea este cam de doua ori mai frecventa in C decit testul de egalitate, este normal ca si operatorul de asignare sa fie jumatate din cel de egalitate, ca lungime.

Orice caracter singur poate fi scris intre apostrofuri, pentru a produce valoarea numerica a caracterului in codul de carctere al calculatorului; acesta se numeste constanta de caracter. Asa de exemplu, 'A' este o constanta de caracter; in setul de caractere ASCII, valoarea sa este 65, reprezentarea interna a caracterului A. Desigur 'A' este de preferat lui 65: semnificatia lui este evidenta si independenta de orice set particular de caractere.

Secventele escape folosite in sirurile de caractere sint si ele legale in constantele de caracter, asa ca in teste si in expresii aritmetice '\n' tine locul caracterului "linie noua". Sa notam ca '\n' este un singur caracter si, in expresii, este echivalent cu un singur intreg; pe de alta parte, "\n" este un sir de caractere care, intimplator, contine un singur caracter. Subiectul comparatiei intre siruri si caractere este discutat mai departe in Capitolul 2.

Exercitiul 1.6. Scrieti un program care sa numere blankurile, taburile si new-line-urile.

Exercitiul 1.7. Scrieti un program care sa copieze intrarea in iesire,inlocuind fiecare sir de unul sau mai multe blankuri cu un singur blank.

Exercitiul 1.8. Scrieti un program care sa inlocuiasca fiecare tab printr-o secventa >,backspace,- care se va tipari ca "->" s fiecare backspace prin secventa similara "<-". Aceasta face taburile si backspace-urile vizibile.

Contorizarea de cuvinte

Al patrulea program din seria de programe utile va contoriza linii, cuvinte si caractere, un singur cuvint fiind definit ca orice secventa de caractere care nu contine blanc, tab sau linie noua (acesta este de fapt un schelet al programului utilitar "wc" din UNIX).

```
    #define YES  1
    #define NO   0
    main() /*contorizare linii, cuvinte si caractere la intrare*/
    {
       int c, nl, nw, nc, inword;
       inword = NO;
       nl = nw = nc = 0;
       while ((c = getchar()) != EOF) {
                  ++nc;
                  if(c == '\n')
                       ++nl;
                  if(c == ' ' || c == '\n' || c == '\t')
                     inword = NO;
                  else if (inword == NO) {
                        inword = YES;
                        ++nw;
                       }
                 }
       printf("%d %d %d'\n", nl, nw, nc);
 }
```

De fiecare data cind programul intilneste primul caracter al unui cuvint, il contorizeaza. Variabila "inword" inregistreaza de cite ori programul este intr-un cuvint sau nu ; initial el "nu este intr-un cuvint " si variabilei i s-a asignat valoarea NO. Preferam constantele simbolice YES si NO valorilor literale 1 si 0 deoarece ele fac programul mai usor citibil. Desigur ca intr-un program mic ca acesta diferenta este mica, dar intr-un program mai mare cresterea in claritate merita micul efort suplimenar de a-l scrie in acest mod de la inceput. Veti vedea deasemenea ca este mai usor sa efectuati modificari masive in programe in care numerele apar numai ca si constante simbolice.

Linia

```
       nl = nw = nc = 0;
```

seteaza toate cele trei variabile pe zero. Acesta nu este un caz special ci doar o consecinta a faptului ca o asignare asociaza de la dreapta spre stinga. Este ca si cind am fi scris;

```
       nc = (nl = (nw = 0));
```

Operatorul || inseamna SAU, asa ca linia

```
       if(c == ' ' || c == '\n' || c == '\t');
```

spune ca "daca c este un blanc sau c este o linie noua sau c este un tab...". (Secventa escape \t este reprezentarea vizibila a caracterului tab).Exista un operator corespunzator && pentru SI. Expresiile conectate prin && sau || sint evaluate de la stinga la dreapta si evaluarea se opreste atunci cind se cunoaste adevarul sau falsul expresiei. Astfel daca c contine un blanc, nu mai este nevoie sa testam daca el contine

o line noua sau un tab, asa ca testele acestea nu se mai fac. In particular, aceasta nu este important aici, dar este foarte semnificativ in multe situatii complicate, asa cum vom vedea in curind.

Exemplul nostru foloseste deasemenea instructiunea "else", care specifica o actiune alternativa ce trebuie executata daca partea de conditie unei instructiuni "if" este falsa.

Forma generala este:

```
if (expresie)
  instructiune1
else
  instructiune2
```

Una si numai una din instructiunile asociate cu if-else se executa. Daca "expresia" este adevarata, se executa "instructiunea-1"; daca nu, se executa "instructiunea-2". Fiecare "instructiune" poate fi, de fapt, mult mai complicata. In exemplul nostru instructiuea de dupa "else" este un "if" care controleaza doua instructiuni in paranteze.

Exercitiul 1.9. Cum veti testa programul de contorizare cuvinte? Care sint unele dintre limitele lui ?

Exercitiul 1.10. Scrieti un program care sa tipareasca cuvintele introduse,cite unul pe linie.

Exercitiul 1.11. Revizuiti programul de contorizare cuvinte pentru a folosi o mai buna definitie a "cuvintului", de exemplu o secventa de litere, cifre si apostrofuri care incepe cu o litera.

1.6. Tablouri

Vom scrie acum un program care va contoriza aparitiile fiecarei cifre, a fiecarui caracter de spatiere (blanc, tab, linie noua) si a tuturor celorlalte caractere. Desigur, este un program artificial, dar ne va permite sa ilustram mai multe aspecte ale lui C intr-un singur program.

Exista 12 categorii de intrari, asa ca ne este mai convenabil sa folosim un tablou pentru a tine numarul de aparitii a fiecarei cifre, decit sa folosim 10 variabile individuale. Iata acum o versiune a acestui program:

```
main()  /* contorizeaza cifre, spatii albe, alte caractere */
{
      int c, i, nwhite, nother;
      int, ndigit[10];
      nwhite = nother = 0;
      for (i = 0; i < 10; ++i)
            ndigit[i] = 0;
      while ((c = getchar()) != EOF)
```

```
                if (c >= '0' && c <= '9')
                        ++ndigit[c-'0'];
                else if (c == ' ' || c == '\n' || c == '\t')
                        ++nwhite;
                else
                        ++nother;
        printf("digits =");
        for (i = 0; i < 10; ++i)
                printf(" %d", ndigit[i]);
        printf("\nwhite space = %d, other = %d\n",
                nwhite, nother);
  }
```

Declaratia

```
        int ndigit[10];
```

spune ca ndigit este un tablou de 10 intregi. Indicii de tablou intodeauna incep de la zero in C (spre deosebire de FORTRAN sau PL/1 unde incepe de la unu), asa ca elementele tabloului sint ndigit[0], ndigit[1] , ...,ndigit[9]. Acestea se reflecta in buclele "for", care initializeaza si tiparesc tabloul. Un indice poate sa fie orice expresie intreaga, inclusiv desigur variabilele intregi ca "i", sau constantele intregi.

Acest program particular se bazeaza mult pe proprietatile reprezentarii drept caractere a cifrelor. De exemplu, testul

```
        if (c >= '0' && c <= '9') ...
```

determina daca un caracter din c este cifra. Daca el este cifra, valoare numerica a acelei cifre este

```
                c - '0'
```

Acest algoritm functioneaza bine numai daca '0', '1', etc sint pozitive si in ordine crescatoare, si intre '0' si '9' nu se gaseste altceva decit cifre. Din fericire, aceasta este adevarul pentru toate seturile de caractere conventionale.

Prin definitie, calculele aritmetice care implica tipuri "char" si "int", convertesc totul in tipul "int" inainte de prelucrare, asa ca variabilele si constantele de tip "char" sint esential identice cu tipul "int" in contexte aritmetice. Acest fapt este aproape natural si convenabil; de exemplu: c-'0' este o expresie intreaga cu o valoare intre 0 si 9 corespunzatoare caracteruluidintre '0' si '9' depus in c, si deci este un indice valid pentru tabloul ndigit.

Decizia se ia asupra caracterului (daca el este o cifra, un caracter de spatiere sau altceva) in secventa:

```
        if (c >= '0' && c <= '9')
                ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
```

```
                ++nwhite;
        else
                ++nother;
```

Constructia de tipul

```
        if (conditie)
            instructiune
        else if (conditie)
            instructiune
        else
            instructiune
```

apare frecvent in programe ca o modalitate de a exprima deciziile multiple. Codul se citeste simplu de sus in jos pina cind o conditie este indeplinita; in acest punct, se executa partea corespunzatoare de "instructiune" si intreaga constructie este terminata. (Desigur ca "instructiune" pot fi mai multe instructiuni incluse intre paranteze). Daca nici una din conditii nu este indeplinita, instructiunea care urmeaza dupa ultimul "else" este executata daca este prezenta. Daca "else" final si "instructiune" lipsesc (ca in programul nostru), nu are loc nici o actiune. Pot exista un numar arbitrar de constructii de tipul

```
        else if (conditie)
            instructiune
```

grupate intre "if"-ul initial si "else"-ul final. Ca o chestiune de stil, va sfatuim sa formati aceste constructii asa cum le-am facut si noi, astfel incit deciziile lungi sa nu ajunga pe marginea din dreapta a paginii.

Instructiunea "switch", care va fi prezentata in Capitolul 3, reprezinta un alt mod de a scrie o decizie multipla si este potrivita, in particular, cind conditia care se testeaza este simpla sau cind o expresie de caractere sau de intregi se potriveste cu o constanta dintr-un sir dat. Prin contrast, vom prezenta o versiune "switch" a acestui program in Capitolul 3.

Exercitiul 1-12. Scrieti un program pentru a tipari histograma lungimilor cuvintelor care apar la intrare. Este cel mai usor sa desenati histograma orizontal; o orientare verticala este mai laborioasa.

1.7. Functii

In C o functie este echivalenta cu o subrutina sau cu o functie din FORTRAN sau cu o procedura din PL/1 sau PASCAL, etc. O functie reprezinta un mod convenabil de a incapsula anumite calcule intr-o cutie neagra care poate fi apoi folosita fara sa ne mai pese de ce se afla inauntru. Functiile sint singura modalitate de a face fata la complexitatea potentiala a programelor mari. Cu functii scrise asa cum trebuie, este posibil sa ignoram "cum" este facuta o anumita treaba; ne este suficient sa

stim "ce" anumita treaba este facuta. Limbajul C este proiectat pentru a face folosirea lor usoara, convenabila si eficienta; veti observa adesea o functie lunga numai de citeva rinduri, apelata o singura data, numai fiindca ea clarifica anumite portiuni de cod.

Pina acum am folosit numai functii ca printf, getchar si putchar, care au fost scrise de altii pentru noi; este momentul sa ne scriem si noi propriile noastre functii. Deoarece limbajul C nu poseda un operator de exponentiere ca ** din FORTRAN sau PL/1, vom ilustra mecanismul de definire de functii scriind o functie putere "power(m,n)" care va ridica un intreg la o putere intreaga pozitiva in n. Adica, valoarea lui power(2,5) este 32. Desigur ca aceasta functie nu realizeaza toata treaba lui ** deoarece minuieste numai puteri pozitive ale intregilor mici, dar cel mai bine,este bine sa aprofundam un lucru la un moment dat.

Iata acum functia "power" si un program principal care o foloseste, asa ca puteti vedea deodata intreaga structura.

```
main() /* testeaza functia power */
{
        int i;
        for (i = 0; i < 10; ++i)
        printf ('%d %d %d\n', i, power(2,i), power(-3,i));
}
power(x,n)  /* ridica pe x la puterea a n-a ; n > 0 */
int x, n;
{
        int i, p;
        p = 1;
        for (i = 1, i <= n; ++i)
            p = p * x;
        return(p);
}
```

Orice functie are o aceeasi forma:

```
nume (lista de argumente, daca exista)
declaratii de argumente, daca exista
{
    declaratii
    instructiuni
}
```

Functiile pot apare in orice ordine si intr-un fisier sursa sau in doua. Bineinteles, daca sursa apare in doua fisiere, veti avea mai multe de spus la compilare si incarcare decit daca e un singur fisier dar asta este o problema a sistemului de operare si nu un atribut al limbajului. Pentru moment vom presupune ca ambele functii se gasesc intr-un acelasi fisier, asa ca ceea ce ati invatat despre executia programelor C nu se modifica.Functia "power" este apelata de doua ori in linia

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Fiecare apel trimite doua argumente lui power, care de fiecare data returneaza un intreg care trebuie formatat si tiparit. Intr-o expresie power(2,i) este un intreg, la fel ca si 2 si i. (Nu toate functiile produc o valoare intreaga; vom vedea aceasta in Capitolul 4). In "power" argumentele trebuie sa fie declarate corespunzator cu tipul lor cunoscut. Aceasta este facuta de linia

```
int x,n;
```

care urmeaza liniei cu numele functiei. Declaratiile de argumente urmeaza sa se situeze intre lista de argumente si acolada stinga deschisa {; fiecare declaratie se termina cu punct si virgula. Numele folosite de power pentru argumentele sale sint pur locale lui power si nu sint accesibile nici unei alte functii: alte rutine pot folosi aceleasi nume fara nici un conflict. Aceasta este adevarat si pentru variabilele i si p: i din power (nu este legat prin nimic) nu are nici o legatura cu i din main.

Valoarea pe care power o calculeaza este returnata in main prin instructiunea "return", care este la fel ca in PL/1 . Orice expresie poate apare intre paranteze. O functie nu trebuie sa returneze o valoare; o instructiune "return" fara nici o expresie cauzeaza transferul controlului, dar nu o valoare utila, spre apelant, asa cum face "iesirea dupa sfirsit" a unei functii prin atingerea parantezei drepte terminatoare.

Exercitiul 1.13. Scrieti un program care sa converteasca intrarea in litere mici, folosind o functie lower(c) care returneaza pe c, daca c nu este o litera, si valoarea "litera mica a lui c", daca c este o litera.

1.8. Argumente - apel prin valoare

Un aspect al functiilor din limbajul C s-ar putea sa fie nefamiliar programatorilor obisnuiti cu alte limbaje, in particular cu FORTRAN sau PL/1.
In C, toate argumentele functiei sint transmise "prin valoare". Aceasta inseamna ca functiei apelate i se transmit valorile argumentelor in variabile temporare (de fapt intr-o stiva) si nu i se transmit adresele lor. Aceasta duce la citeva proprietati diferite fata de limbajele cu "apel prin referinta" de tipul FORTRAN si PL/1, in care rutina apelata minuieste adresele argumentelor si nu valorile lor.

Principala distinctie este aceea ca in limbajul C, functia apelata nu poate altera o variabila in functia apelata; ea poate altera numai copia ei temporara si privata.

Apelul prin valoare este, cu toate acestea un avantaj si nu o obligatie. Uzual, el conduce la programe mai compacte cu mai putine variabile inutile, deoarece argumentele pot fi tratate ca variabile locale initializate convenabil in rutina apelata. Drept exemplu, dam in continuare o versiune a functiei power care face uz de acest fapt.

```
power(x,n)  /*ridica pe x la puterea a n-a; n > 0;versiunea 2*/
int x, n;
{
      int p;
      for (p = 1; n > 0; --n)
         p = p * x;
      return(p);
}
```

Argumentul n este folosit ca o variabila temporara, si este decrementat pina cind devine zero; nu mai este nevoie de variabila i. Ceea ce se face cu n in interiorul lui power nu are nici un efect asupra argumentului cu care a fost apelata power initial.

Cind este necesar, este posibil sa aranjam ca o functie sa modifice o variabila in rutina apelanta. Apelandul trebuie sa dea adresa variabilei de setat (in mod tehnic, sa creeze un pointer la variabila),iar functia apelata trebuie sa declare argumentul ca fiind un pointer si sa refere variabila reala in mod indirect prin el. Vom discuta in detaliu aceste probleme in Capitolul 5.

Cind numele unui tablou este folosit ca si argument, valoarea transmisa functiei este locatia sau adresa de inceput a tabloului. (Nu se face nici o copiere de elemente de tablou). Indiciind aceasta valoare, functia poate avea acces si altera orice element al tabloului. Acesta este subiectul urmatoarei sectiuni.

1.9. Tablouri de caractere

In mod probabil, cel mai comun tip de tablouri in limbajul C este tabloul de caractere. Pentru a ilustra folosirea tablourilor de caractere si a functiilor care le manipuleaza, vom scrie un program care citeste un set de linii si o tipareste pe cea mai lunga. Schita lui este destul de simpla:

```
       while (mai exista o alta linie)
           if (este mai lunga decit linia anterioara)
               salveaza-o pe ea si lungimea ei
       tipareste linia cea mai lunga
```

Aceasta schita ne arata clar ca programul se imparte in bucati. O bucata citeste o linie noua, o alta bucata o testeaza, o alta o salveaza iar restul controleaza procesul.

Deoarece lucrurile se impart asa de frumos, ar fi mai bine sa le scriem la fel. Pentru aceeasta, vom scrie la inceput o functie getline care va citi urmatoarea linie de la intrare; ea este generalizare a functiei getchar. Pentru a face functia utila si in alte contexte, vom incerca sa o scriem cit mai flexibil. In mod minim, getline va trebui sa returneze un semnal despre posibilul sfirsit de fisier; proiectind-o mai general,ea va trebui sa returneze lungimea liniei sau zero daca se intilneste sfirsitul de fisier. Zero nu este niciodata o lungime valida de

linie, deoarece orice linie are cel putin un caracter, chiar si o linie ce contine numai caracterul "linie noua" are lungimea 1.

Cind gasim o linie care este mai lunga decit linia cea mai lunga gasita anterior, trebuie sa o salvam undeva. Aceasta sugereaza o a doua functie, copy, pentru a salva noua linie intr-un loc sigur.

In final, avem nevoie de un program principal care sa controleze functiile getline si copy. Iata rezulatul:

```
        #define MAXLINE 1000  /* lungimea maxima a liniei */
        main()                /* gaseste linia cea mai lunga */
        {
            int len;  /* lungimea liniei curente */
            int max;  /* lungimea maxima gasita pina acum */
            char line[MAXLINE]; /* linia curenta introdusa */
            char save[MAXLINE]; /* cea mai lunga linie salvata */
            max = 0;
            while ((len = getline(line, MAXLINE)) > 0)
                 if (len > max)  {
                     max = len;
                     copy(line, save);
                 }
            if (max > 0)  /* s-a citit cel putin o linie */
               printf("%s", save);
        }

   getline (s, lim)  /* citeste linia in s, returneaza lungimea */
   char s[];
   int lim;   {
     int c, i;
     for(i = 0; i < lim - 1 && (c=getchar())!=EOF && c!='\n';++i)
          s[i] = c;
     if (c == '\n') {
          s[i] = c;
          ++i;      }
     s[i] = '\0';
     return(i);
   }

   copy(s1, s2)  /* copiaza pe s1 in s2; s2 suficient de mare */
   char s1[], s2[];
   {
     int i;
     i = 0;
     while ((s2[i] = s1[i]) != '\0')
          ++i;
   }
```

main si getline comunica intre ele printr-o pereche de argumente si o valoare returnata. In getline, argumumentele sint declarate prin liniile:

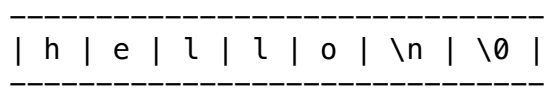
```
        char s[];
        int lim;
```

care spun ca primul argument este un tablou iar al doilea un intreg. Lungimea tabloului s nu este specificata in getline deoarece ea este determinata in main. "getline" foloseste instructiunea return pentru a trimite o valoare inapoi apelantului, la fel cum facea si functia power. Unele functii returneaza o valoare utila; altele, de exemplu copy, sint folosite numai pentru efectul lor si nu returneaza nici o valoare.

getline pune caracterul \0 (caracterul nul, a carui valoare este zero) la sfirsitul tabloului pe care il creaza, pentru a marca sfirsitul sirului de caractere. Aceasta conventie este folosita de asemenea si de catre compilatorul C; cind o constanta sir de tipul

```
"hello\n"
```

este scrisa intr-un program C, compilatorul isi creaza un tablou de caractere continind carcterele sirului si terminat cu \0, astfel incit o functie, de exemplu printf, poate sa-i determine sfirsitul.



Specificatorul de format %s din printf se asteapta la un sir reprezentat tocmai in aceasta forma. Daca examinati functia copy, veti descoperi ca si ea se bizuie de fapt pe terminarea argumentului sau de intrare s1 cu un \0 si ea copiaza acest caracter in argumentul de iesire s2. (Toate acestea presupun ca \0 nu este parte a unui text normal).

Este demn de mentionat in trecere ca, un program, chiar si atit de mic ca acesta, prezinta unele probleme delicate de proiectare. De exemplu, ce ar face main daca ar intilni o linie mai mare decit limita sa? getline lucreaza bine, adica se va opri atunci cind tabloul este plin chiar daca nu a intilnit nici un caracter "linie noua". Testind lungimea si ultimul caracter returnat, main poate determina cind a fost linia prea lunga si apoi sa actioneze cum vrea. Pentru a scurta programul, am ignorat acest aspect.

Nu exista vre-o cale pentru utilizatorul lui getchar de a sti inainte cit va fi de lunga o linie de intrare, asa ca getline verifica daca nu s-a produs o depasire. Pe de alta parte, utilizatorul lui copy stie intodeauna (sau poate descoperi) cit este de mare sirul, asa ca nu trebuie sa adaugam la functie o verificare de erori.

Exercitiul 1.14. Revizuiti rutina main din programul precedent astfel incit ea sa tipareasca corect lungimea unei linii de intrare de o lungime arbitrara, si atita text cit este posibil de tiparit.

Exercitiul 1.15. Scrieti un program care sa tipareasca toate liniile mai lungi de 80 de caractere.

Exercitiul 1.16. Scrieti un program care sa elimine blancurile nesemnificative (cele de dupa un caracter diferit de blanc sau tab) din fiecare linie de intrare si care sa stearga liniile care contin numai blancuri.

Exercitiul 1.17 Scrieti o functie reverse(s) care sa inverseze un sir de caractere s. Folositi-o pentru a scrie un program care isi inverseaza linie cu linie intrarea.

1.10 Domeniu; variabile externe

Variabile din main (line, save, etc) sint private sau locale lui main; deoarece ele sint declarate in main, nici o alta functie nu poate avea acces direct la ele. La fel se intimpla si cu variabilele din alte functii de exemplu variabila i din getline nu are nici o legatura cu varibila i din copy. Fiecare variabila locala dintr-o rutina se naste numai atunci cind functia este apelata si"dispare" cind functia isi termina activitatea. Aceasta este ratiunea pentru care astfel de variabile sint cunoscute uzual sint numele de variabile automate, urmind terminologia din alte limbaje. Vom folosi termenul de "automat" de aici inainte pentru a ne referi la aceste variabile dinamice locale. (Capitolul 4 discuta clasa de memorie "statica" in care variabilele locale isi pastreaza valoarea intre apelurile la functii).

Deoarece variabilele automate vin si pleaca odata cu apelurile de functii, ele nu-si pastreaza valoarea de la un apel la altul si trebuie initializate explicit inainte de fiecare intrare. Daca nu sint setate, rezultatele vor fi imprevizibile.

Ca o alternativa la variabilele automate, este posibil sa definim variabile care sa fie "externe" tuturor functiilor, adica, variabilele globale care sa fie accesate prin nume de orice functie care doreste sa o faca. (Acest mecanism este foarte asemanator cu COMMON din FORTRAN sau EXTERNAL din PL/1). Deoarece variabilele externe sint accesibile global, ele pot fi folosite in locul listelor de argumente pentru a comunica date intre functii. Mai mult, deoarece variabilele externe exista permanent, si nu apar si dispar dupa cum functia este apelata sau s-a terminat, ele isi pastreaza valorile chiar si dupa ce s-a terminat functia care le-a setat.

O variabila externa trebuie sa fie definita in afara oricarei functii; acest lucru face sa se aloce memorie reala pentru ea. Variabila trebuie deasemenea sa fie declarata in fiecare functie care vrea sa o foloseasca; aceasta se poate face fie printr-o declaratie explicita "extern", fie implicit prin context. Pentru a face discutia corecta, vom rescrie programul precedent, in care line, save, max vor fi declarate variabile externe. Aceasta va cere modificari in apeluri, in declaratii si in corpurile celor trei functii.

```
#define MAXLINE 1000 /* marimea maxima a liniei de intrare */
char line[MAXLINE];  /* linia de intrare */
char save[MAXLINE];  /* cea mai lunga linie este salvata aici*/
int max; /* lungimea liniei celei mai mari */
```

```
  main()   /* gaseste linia cea mai lunga ;versiune specializata*/
  {
                int len;
                extern int, max;
                extern char save[];
                max = 0;
                while ((len = getline() > 0)
                        if (len > max)  {
                            max = len;
                            copy();
                        }
                if (max > 0)     /* a fost cel putin o linie */
                        printf("%s", save);
  }

  getline()             /* versiune specializata */
  {
                int c, i;
                extern char line[];
  for (i = 0; i < MAXLINE-1 && (c = getchar()) != EOF && c != '\n';
++i)
                        line[i] = c;
                if (c == '\n') {
                        line[i] = c;
                        ++i;
                }
                line[i] = '\0';
                return(i);
  }

  copy()                        /* versiune specializata */
  {
                int i;
                extern char line[], save[];
                i = 0;
                while ((save[i] = line[i]) != '\0')
                        ++i;
  }
```

Variabilele externe din main, getline si copy sint definite de primele linii din exemplul de mai sus care declara tipul lor si provoaca o alocare de memorie pentru ele. Din punct de vedere sintactic, definitiile externe sint asemanatoare cu declaratiile pe care le-am folosit anterior dar deoarece ele apar in afara functiilor, variabilele sint externe. Inainte ca o functie sa poata folosi o variabila externa, numele variabilei trebuie sa fie facut cunoscut functiei O modalitate pentru a face aceasta este scriind o declaratie "extern"; declaratia este identica cu cea de dinainte, avind insa in plus cuvintul cheie extern.

In anumite circumstante, declaratia "extern" poate fi omisa: daca definitia externa a variabilei apare in fisierul sursa inainte de folosirea ei intr-o functie particulara, atunci nu este necesara o declaratie "extern" in functie. Deci, declaratiile "extern" din main, getline si cpoy sint redundante. De fapt prac-

tica uzuala consta in plasarea definitiilor tuturor variabilelor externe la inceputul fisierului sursa si apoi omiterea tuturor declaratiilor "extern".

Daca programul consta din mai multe fisiere sursa si o variabila este definita in fisierul 1 si folosita in fisierul 2 atunci e nevoie de o declaratie "extern" in fisierul 2 pentru a conecta cele 2 aparitii ale variabilei. Acest subiect este discutat pe larg in capitolul 4.

Puteti nota ca am folosit cu grija cuvintele "declaratie" si "definitie" cind ne-am referit la variabile externe in aceasta sectiune. "Definitiile" se refera la locul in care variabila este efectiv creata si i se asigneaza memorie; "declaratie" se refera la locul unde natura variabilei este declarata dar nu i se aloca memorie.

Fiindca veni vorba, exista o tendinta de a face totul cu ajutorul variabilelor externe deoarece ele par a simplifica toate comunicatiile - listele de argumente sint scurte si variabilele sint intodeauna acolo cind aveti nevoie de ele. Dar variabilele externe sint intodeauna acolo chiar si cind nu aveti nevoie de ele. Acest stil de a codifica este plin de pericole deoarece el conduce la programe ale caror conexiuni de date nu sint evidente clar - variabilele pot fi modificate in moduri neasteptate si chiar inadvertente iar programul devine greu de modificat daca acest lucru este necesar. A doua versiune a programului care cauta linia cea mai luunga este inferioara primei, partial din aceste motive, si partial deoarece ea distruge generalitatea a doua functii atit de utile introducind in ele numele unor variabile pe care le vor folosi.

> Exercitiul 1.18 Testul din instructiunea for din functia getline de mai sus este aproape de neinteles. Rescrieti programul pentru a-l face mai clar dar pastrati acelasi comportament la sfirsitul fisierului sau la depasire de buffer. Este acest comportament cel mai adecvat ?

1.11 Rezumat

In acest punct am acoperit ceea ce, conventional, poate fi numit esenta lui C. Cu aceste citeva caramizi, este posibil sa scrieti programe utile de marime considerabila dar ar fi o buna idee daca v-ati odihni mai mult inainte de a face asa ceva.

Exercitiile care urmeaza au intetia de a va oferi sugestii pentru programe de o complexitate oarecum mai mare decit cele prezentate in acest capitol.

Dupa ce reusiti sa aveti sub control aceasta parte a lui C, ar fi demn de efortul dumneavoastra sa cititi mai departe, deoarece caracteristicile lui C acoperite in urmatoarele citeva capitole sint cele in care puterea si expresivitatea limbajului devin aparente.

Exercitiul 1.19. Scrieti un program "detab" care inlocuieste taburile din intrare cu numarul potrivit de blancuri pentru a

sari pina la urmatorul stop de tab. Presupuneti un set fixat de stopuri de tab, fie din n in n pozitii.

Exercitiul 1.20. Scrieti un program "entab" care inlocuieste siruri de blancuri cu numarul minim de taburi si blancuri pentru a obtine o aceasi spatiere. Folositi aceleasi stopuri de tab ca si detab.

Exercitiul 1.21. Scrieti un program pentru a "impaturi" liniile de intrare lungi dupa ultimul caracter neblanc care apare inainte de a n-a coloana a intrarii, unde n este un parametru. Asigurati-va ca programul dumneavoastra lucreaza inteligent cu liniile foarte lungi, chiar daca nu e nici un tab sau blanc inainte de coloana specificata.

Exercitiul 1.22. Scrieti un program care sa elimine toate comentariile dintr-un program C. Nu uitati sa minuiti adecvat sirurile dintre ghilimele si constantele de caractere.

Exercitiul 1.23. Scrieti un program pentru a verfica un program C din punct de vedere al erorilor de sintaxa rudimentare ca de exemplu: paranteze neperechi. Nu uitati ghilimelele, atit cele simple cit si cele duble si comentariile. (Acest program este greu daca il faceti la cazul cel mai general.)

CAPITOLUL 2. TIPURI, OPERATORI SI EXPRESII

Variabilele si constantele sint 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

Cu toate ca nu am spus-o pina acuma, exista unele restrictii asupra numelor de constante si variabile. Numele sint 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 sint 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 sint 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 sint folosite de diferite asambloare si incarcatoare. In Anexa A se dau detalii. Mai mult, cuvinte cheie ca: if, else, int, etc sint rezervate: nu pot fi folosite ca nume de variabile

(trebuie sa fie scrise cu litere mici).
Natural, e 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 sint 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 depinde de calculatorul care le minuieste; tabelul urmator da citeva valori reprezentative:

	DEC PDP11 ASCII	Honeywell 6000 ASCII	IBM/370 EBCDIC	Interdata 8/32 ASCII
char	8 biti	9 biti	8 biti	8 biti
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

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. Asa cum puteti vedea, fiecare compilator este liber sa interpreteze short si long in functie de hardul pe care se executa. 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 sint 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 sint scrise in stilul 123L. O constanta intreaga normala care este prea lunga pentru un int, este luata deasemenea ca fiind o constanta long.

Exista o notatie speciala pentru constantele octale si hexazecimale: un 0 (zero) la inceputul unei constante int inseamna octal; un 0x sau 0X la inceputul unei constante int inseamna hexazecimal. De exemplu, numarul zecimal 31 poate fi scris 037 in octal si 0x1f sau 0X1F in hexazecimal. Constantele octale si hexazecimale pot fi urmate un L pentru a le face "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, iar in EBCDIC, 240, amindoua valorile fiind diferite de valoarea numerica 0. Scriind '0' in loc de o valoare numerica de tipul 48 sau 240, facem programul independent de o valoare particulara. Constantele caracter participa in operatiile numerice la fel ca oricare alte numere, cu toate ca cel mai adesea ele sint folosite in comparari cu alte caractere. O sectiune viitoare va trata toate regulile de conversie.

Anumite caractere negrafice pot fi reprezentate constante caracter cu ajutorul secventelor escape, de exemplu \n (linie noua), \t (tab), \0 (nul), \\ (backspace), \'(ghilimea simpla) etc, care arata ca doua caractere, dar de fapt sint unul singur. In plus, se poate genera orice model de lungime un octet, scriind:

```
'\ddd'
```

unde 'ddd' reprezinta 1 - 3 cifre octale, ca in

```
#define FORMFEED '\014' /* ASCII formfeed */
```

Constanta caracacter '\0' reprezinta caracterul ce are valoarea '\0' se scrie adesea in locul lui 0 pentru accentua natura caracter a anumitor expresii.

O expresie constanta este o expresie care implica numai constante. Astfel de expresii sint 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 sint 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 sint 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 carctere 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 sint 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 sint initializate la fiecare apel al functiei in care sint 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 initializrea 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 sint "+", "-", "*", "/" 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. De exemplu, un an este bisect daca este divizibil cu 4 si daca nu este divizibil cu 100, insa anii divizibili cu 400 sint bisecti. Deci

```
        if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
            it's a leap year
        else
            it's not
```

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 (Tabela de la sfirsitul capitolului rezuma ponderea si asociativitatea pentru toti operatorii). 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.

Actiunile care produc depasiri superioare sau inferioare depind in ultima instanta de calculator.

2.6. Operatori relationali si logici

Operatorii relationali sint > >= < <=. 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 sint conectorii logici && si ||. Expresiile care-i contin sint 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 luata din functia de intrare getline, pe care am scris-o in Capitolul 1:

```
    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 sint 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. Construc-tiile 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 sint 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 sint 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 sint contigue - intre A si Z se gasesc numai litere. Aceasta ultima observatie nu este valabila pentru setul de caractere EBCDIC (IBM 360/370), asa incit functia lower esueaza pentru aceste sisteme, ea va converti mai mult decit literele mari.

Exista o subtilitate in conversia caracterelor in intregi. Limbajul nu specifica daca o variabila de tip char este o cantitate cu semn sau fara semn. Cind un char este convertit intr-un int, poate el produce un intreg negativ ? Din pacate, aceasta variaza de la calculator la calculator, reflectind diferentele arhitecturale. Pe anumite calculatoare (de exemplu PDP-11) un char al carui cel mai din stinga bit este 1 va fi convertit intr-un intreg negativ ("extensie de semn". Pe altele, un char este convertit intr-un int prin adaugarea de zerouri in partea stinga si astfel el este intodeauna pozitiv.

Definitia lui C asigura ca orice caracter din setul standard al masinii nu va fi niciodata negativ, asa ca aceste caractere pot fi folosite liber in expresii ca si cantitati pozitive. Dar modele arbitrare de biti memorate in variabile de tip character pot apare drept negative pe anumite calculatoare si drept pozitive pe altele.

Cea mai comuna aparitie a acestei situatii este cind pentru EOF se foloseste -1. Sa consideram codul:

```
char c;
c = getchar();
if (c == EOF)
    ...
```

Pe un calculator care nu face extensie de semn, c este intodeauna pozitiv deoarece el este un char, dar totusi EOF este negativ. In consecinta testul esueaza intodeauna. Pentru a evita aceasta, trebuie sa avem grija atunci cind folosim int in loc de char pentru orice variabila care primeste o valoare returnata de getchar.

Adevarata ratiune pentru utilizarea lui int in loc de char nu este legata cu nimic de posibila extensie de semn. Pur si simplu, getchar trebuie sa returneze toate caracterele posibile

(astfel incit sa poate fi folosita pentru a citi o intrare arbitrara) si in plus, o valoare pentru EOF distincta. Astfel, aceasta valoare nu poate fi reprezentata ca si un char dar, in schimb, trebuie memorata ca si un int.

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 isgit 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 sint 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 sint eliminati. Astfel, in:

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

valoarea lui c ete 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 sint 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 sint 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.

In final, conversia explicita de tip poate fi fortata in orice expresie cu o constructie numita "distribuire"(cast). In constructia:

```
        (numedetip) expresie
```

sus. Semnificatia precisa a unei distribuiri este de fapt ca si daca o expresie ar fi asignata la o variabila de tipul specificat, care este apoi folosita in locul intregii constructii. De exemplu, rutina din biblioteca sqrt are nevoie de un argument double si va produce nonsens daca i se da sa minuiasca altceva. Astfel, daca n este un intreg:

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

il converteste pe n in double inainte de a-l pasa lui sqrt. (De notat ca distribuirea produce valoarea n in tipul potrivit; continutul efectiv al lui n nu este alterat). Operatorul de distribuire are acceasi pondere ca si alti operatori unari, asa cum apare si in tabelul recapitulativ de la sfirsitul capitolului.

> Exercitiul 2.2. Scrieti o functie htoi(s) care converteste un sir de cifre hexazecimale in valoarea sa intreaga echivalenta. Cifrele sint 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++ sint 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 gustul dumneavoastra. Dar exista totusi situatii in care unul sau altul este apelat din adins. De exemplu, sa consideram functia squeeze(s,c) care elimina toate aparitiile lui c din sirul s:

```
        squeeze (s,c)  /* sterge toate aparitiile lui c din s */
        char s[];
        int c;
        {
                int i, j;
                for (i = j = 0; s[i] != '\0'; i++)
```

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

De fiecare data cind apare un caracter non-c el este copiat in pozitia j curenta si numai dupa aceea j este incrementat pentru a fi gata pentru urmatorul caracter. Aceasta constructie este echivalenta cu urmatoarea:

```
        if (s[i] != c)     {
              s[j] = s[i];
              j++;
        }
```

Un alt exemplu de constructie similara este luata din functia getline pe care am scris-o in Capitolul 1, in care putem inlocui

```
        if (c == '\n'   {
             s[i]=c;
             ++i;
        }
```

cu mult mai compacta constructie:

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

Ca un al treilea exemplu functia strcat(s,t) care concatenea-za sirul t la sfirsitul sirului s. strcat presupune ca exista suficient spatiu in s pentru a pastra combinatia.

```
        strcat (s,t)   /* concateneaza pe t la sfirsitul lui s */
        char s[], t[]; /* s trebuie sa fie suficient de mare */
        {
          int i, j;
          i =  j  = 0;
          while (s[i] != '\0')  /* gaseste sfirsitul lui s */
               i++;
          while ((s[i++] = t[j++]) != '\0') /* copiaza pe t */
               ;
        }
```

Cum fiecare caracter este copiat din t in s, se aplica postfixul ++ atit lui i cit si lui j pentru a fi siguri ca sint pe pozitie pentru urmatorul pas din bucla.

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 caracter 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.

```
 &      SI bit cu bit
 |      SAU inclusiv bit cu bit
 ^      SAU exclusiv bit cu bit
<<      deplasare stinga
>>      deplasare dreapta
 ~      complement fata de 1 (unar)
```

Operatorul SI bit cu bit "&" este folosit adesea pentru a masca anumite multimi de biti; de exemplu

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

pune pe zero toti biti lui n, mai putin bitul 7 (cel mai tare). Operatorul SAU bit cu bit "|" este folosit pentru a pune pe 1 biti:

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

pune pe 1 in x bitii care sint setati pe 1 in MASK.

Trebuie sa distingeti cu grija operatorii pe biti & si | de conectorii logici && si ||, care implica o evaluare de la stinga la dreapta a unei valori de adevar. De exemplu, daca x este 1 si y este 2, atunci x & y este zero dar x && y este 1. (De ce ?)

Operatorii de deplasare << si >> realizeaza deplasari la stinga si la dreapta pentru operandul lor din stinga, cu numarul de pozitii dat de operandul din dreapta lor. Astfel x << 2 deplaseaza la stinga pe x cu doua pozitii, umplind locurile libere cu zero; aceasta este echivalent cu inmultirea cu 4. Deplasind la dreapta o cantitate unsigned, bitii vacanti se umplu cu zero. Deplasind la dreapta o cantitate cu semn, bitii vacanti se umplu cu semnul ("deplasarea aritmetica") pe anumite calculatoare, ca de exemplu PDP-11 si cu 0 ("deplasare logica") pe altele.

Operatorul unar ~ da complementul fata de 1 al unui intreg; adica, el converteste fiecare bit de 1 in 0 si vicevesa. Acest operator isi gaseste utilitate in expresii de tipul

```
x & ~077
```

care mascheaza ultimii 6 biti ai lui x pe 0. De notat ca x & ~077 este independent de lungimea cuvintului si deci preferabil, de exemplu, lui x & 0177700, care presupune ca x este

o cantitate cu o lungime de 16 biti. Forma portabila nu implica un cost mai mare, deoarece ~077 este o expresie constanta si deci evaluata la compilare.

Pentru a ilustra folosirea unora din operatorii de biti, sa consideram functia getbits(x,p,n) care returneaza (cadrat la dreapta) cimpul de lungime n biti al lui x care incepe la pozitia p. Presupunem ca bitul 0 este cel mai din dreapta si ca n si p sint valori pozitive sensibile. De exemplu, getbits(x,4,3) returneaza 3 biti in pozitiile 4, 3 si 2, cadrati la dreapta.

```
getbits (x, p, n)  /*  ia n biti de la pozitia p */
unsigned x, p, n;
{
    return( (x >> (p+1-n) ) & ~(~0 << n));
}
```

x >> (p+1-n) muta cimpul dorit la sfirsitul din dreapta al cuvintului. Declarind argumentul x ca fiind unsigned ne asiguram ca atunci cind el este deplasat la dreapta bitii vacanti vor fi umpluti cu 0 si nu cu bitii de semn, independent de calculatorul pe care este executat programul. ~0 este cuvintul cu toti bitii pe 1; deplasindu-l la stinga cu n pozitii prin ~0 << n cream o masca cu zerouri pe cei mai din dreapta n biti si 1 in rest; complementindu-l cu ~ facem o masca de 1 pe cei mai din dreapta n biti.

Exercitiul 2.5. Modificati getbits pentru a numara bitii de la stinga la dreapta.

Exercitiul 2.6. Scrieti o functie wordlength() care calculeaza lungimea unui cuvint de pe calculatorul gazda, adica numarul de biti dintr-un int. Functia sa fie portabila in sensul ca acelasi cod sursa sa lucreze pe toate calculatoarele.

Exercitiul 2.7. Scrieti o functie rightrot(n, b) care roteste intregul n la dreapta cu b pozitii.

Exercitiul 2.8. Scrieti o functie invert(x,p,n) care inverseaza (i.e. schimba pe 1 in 0 si viceversa) cei n biti ai lui x care incep de la pozitia p,lasindu-i pe ceilalti neschimbati.

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 sint 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
```

Dam in continuare, drept exemplu, functia bitcount, care contorizeaza numarul de biti pe 1 dintr-un argument intreg.

```
bitcount(n) /* contorizeaza bitii 1 din n */
unsigned n;
{
  int b;
  for (b = 0; n != 0; n >>= 1)
    if (n & 01)
         b++;
  return(b);
}
```

Lasind la o parte conciziunea, operatorii de asignare au avantajul ca ei corespund cel mai bine modului de gindire al oamenilor. Noi spunem "adunam 2 la i" sau " incrementam pe i cu 2" si nu "ia-l pe i, aduna 2, apoi pune rezultatul inapoi in i". Deci i += 2. In plus, pentru o expresie complicata, de tipul:

```
yyval[yypv[p3 + p4] + yypv[p1 + p2]] += 2
```

operatorul de asignare face codul mai usor de inteles, deoarece cititorul nu trebuie sa verifice sirguincios ca cele doua expresii sint intr-adevar o aceeasi sau sa se intrebe de ce nu sint. In plus, un operator de asignare ajuta chiar compilatorul sa produca un cod mai eficient.

Am folosit deja faptul ca o instructiune de asignare are o valoare si ca poate sa apara in expresii; exemplul cel mai comun:

```
        while ((c = getchar()) != EOF)
              ...
```

Asignarile folosind alti operatori de asignare (+=, -=, etc) pot deasemenea sa apara in expresii, cu toate ca acestea se intimpla mai rar.
Tipul unei expresii de asignare este tipul operandului sau sting.

Exercitiul 2.9. Intr-un sistem cu numere cu complement fata de 2, x & (x-1) sterge bitul 1 cel mai departe de x. (De ce ?). Folositi aceasta observatie pentru a scrie o versiune mai rapida a lui bitcount.

2.11. Expresii conditionale

Instructiunile

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

calculeaza desigur in z maximul dintre a si b. Expresia conditionala, scrisa cu operatorul ternar "? :" ofera un mod alternativ pentru a scrie acest lucru precum si constructii similare. In expresia:

```
        e1 ? e2 : e3
```

expresia e1 se evalueaza prima. Daca ea este nonzero (adevarata) atunci se evalueaza expresia e2 si aceasta este valoarea expresiei conditionale. Altminteri, se evalueaza e3 si aceasta este valoarea. Numai una din expresiile e2 si e3 se evalueaza. Deci, pentru a pune in z maximul dintre a si b:

```
        z = (a > b) ? a : b;     /* z = max(a, b) */
```

Trebuie sa notam ca expresia conditionala este intr-adevar o expresie si ca ea poate fi folosita exact ca oricare alta expresie. Daca e2 si e3 sint expresii de tipuri diferite, tipul reultatului se determina dupa regulile de conversie discutate mai inainte in acest capitol. De exemplu, daca f este un float si n este un int, atunci expresia

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

este de tipul double, indiferent daca n este pozitiv sau nu.

Parantezele nu sint necesare in jurul primei expresii a unei expresii conditionale, deoarece ponderea lui ? : este foarte mica, chiar deasupra asignarii. Ele sint totusi recomandate, pentru a face partea de conditie a expresiei mai usor de vazut.

Expresiile conditionale conduc adesea la un cod succint.

De exemplu, bucla urmatoare tipareste N elemente ale unui tablou, 10 pe linie, cu fiecare coloana separata printr-un blanc si cu fiecare linie (inclusiv ultima) terminata cu un singur caracter "linie noua".

```
for (i = 0; i << N; i++)
printf("%6d%c", a[i], (i % 10 == 9 || i == N-1) ? '\n' : ' ');
```

Un caracter "linie noua" se tipareste tot dupa al zecelea element si dupa al N-lea element. Toate celelalte elemente sint urmate de un blanc. Cu toate ca seamana cu un truc, este instructiv sa incercati sa scrieti lucrul acesta fara a folosi expresia conditionala.

Exercitiul 2.10. Rescrieti functia lower, care converteste literele mari in litere mici, cu o expresie conditionala in locul lui if-else.

2.12. Ponderea si ordinea de evaluare

Tabelul de mai jos rezuma regulile de pondere si asociativitate pentru toti operatorii, inclusiv pentru aceia pe care nu i-am discutat inca. Operatorii de pe aceeasi linie au aceeasi pondere; liniile sint in ordine de pondere descrescatoare, astfel ca, de exemplu, "*", "/" si "%" au o aceeasi pondere, care este mai mare decit a lui "+" "-".

() [] -> .	de la stinga la dreapta
! ~ ++ -- - (tip) * & sizeof	de la dreapta la stinga
* / %	de la stinga la dreapta
+ -	de la stinga la dreapta
<< >>	de la stinga la dreapta
< <= > >=	de la stinga la dreapta
== !=	de la stinga la dreapta
&	de la stinga la dreapta
^	de la stinga la dreapta
|	de la stinga la dreapta
&&	de la stinga la dreapta
||	de la stinga la dreapta
? :	de la dreapta la stinga

= += -= etc	de la dreapta la stinga
, (Capitolul 3)	de la stinga la dreapta

Operatorii -> si. sint folositi pentru a accede membrii structurilor; ei vor fi iscutati in Capitolul 6, impreuna cu sizeof (marimea unui obiect). Capitolul 5 discuta * (indirectarea) si & (adresa lui ...).

Sa notam ca ponderea operatorilor logici pe biti &, | si ^ este sub == i |=. Aceasta implica faptul ca expresiile care testeaza biti, ca de exemplu

```
if (( x & MASK) == 0) ...
```

trebuie sa fie cuprinse in intregime intre paranteze, pentru a da rezultatele steptate.

Asa cum am mentionat mai inainte, expresiile ce implica operatori asociativi si comutativi (+, *, &, ^, |) pot fi rearanjate chiar daca sint cuprinse in paranteze. In marea majoritate a cazurilor, aceasta nu da diferente; in situatia in care ar da, se pot folosi variabile temporare explicite pentru a forta o ordine de evaluare particulara.

Limbajul C, ca si majoritatea celorlalte limbaje, nu specifica in ce ordine se evalueaza operanzii unui operator. De exemplu, in instructiuni de tipul

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

f poate fi evaluat inaintea lui g sau viceversa; deci, daca sau f sau g altereaza o variabila externa de care depinde si cealalta, x poate depinde de ordinea de evaluare. Din nou, rezultatele intermediare pot fi stocate in variabile temporare pentru a fi siguri de o anumita secventa.

In mod similar, ordinea in care sint evaluate argumentele unei functii nu este specficata, asa ca instructiunea

```
printf("%d %d\n", ++n, power(2, n));  /* GRESIT */
```

poate (si o si face) produce rezultate diferite, pe diferite calculatoare, depinzind de faptul daca n este incrementat sau nu inainte de apelul lui power. Solutia, desigur, este sa scriem:

```
++n;
printf("%d %d\n", n, power(2, n));
```

Apelurile de functii, instructiunile de asignare imbricate, operatorii de incrementare si decrementare provoaca "efecte secundare" - o anumita variabila este modificata ca un produs al unei evaluari de expresie. In orice expresie implicind efecte secundare, pot exista subtile dependente de ordinea in care sint

stocate variabilele ce iau parte in expresie. O situatie neferi-cita este ilustrata de instructiunea:

```
a[i] = i++;
```

Chestiunea consta in a sti daca indicele este noua valoare a lui i sau daca este vechea. Compilatorul poate face aceste lucruri in moduri diferite, depinzind de interpretarea sa. Cind efectele secundare (asignare la variabile efective) au loc, sint lasate la discretia compilatorului, caci cea mai buna ordine depinde puternic de arhitectura calculatorului.

Morala acestei discutii este aceea, ca scrierea de cod ce depinde de ordinea de evaluare, este o proasta practica de programare in orice limbaj. Natural, e necesar sa stim ce lucruri trebuie evitate, dar daca nu stim cum sint ele facute pe diferite calculatoare, aceasta inocenta ne va ajuta sa ne protejam. (Verificatorul lui C, lint, detecteaza majoritatea de-pendentelor de ordinea de evaluare).

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, nu separator, cum este in limbajele de tipul ALGOL.

Acoladele { si } sint folosite pentru a grupa impreuna instructiuni si declaratii intr-o instructiune compusa sau bloc, asa ca ele sint sintactic echivalente cu o singura instruc-tiune. Acoladele ce inchid instructiunile unei functii sau cele pentru instructiunile multiple dupa un if, else, while, for sint exemple clare pentru aceasta. (Variabilele pot fi de fapt declarate inlauntrul oricarui bloc; vom discuta despre aceasta in Capitolul 4). Nu se pune niciodata punct si virgula dupa acola-da 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, sint 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;
```

Ambiguitatea este vatamatoare indeosebi in situatii ca urmatoarea:

```
if (n > 0)
     for (i = 0; i < n; i++)
          if (s[i] > 0)  {
               printf("...");
```

```
                    return(i);
                }
        else   /* WRONG */
             printf("error- n is zero \n");
```

Tabularea arata neechivoc ceea ce dorim, dar compilatorul nu intelege acest mesaj, si-l asociaza pe else cu if-ul cel mai dinauntru. Acest tip de eroare poate fi foarte greu de gasit.

Apropo, 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 intodeauna 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 sint 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])
            low = 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 sint 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.

Ramificarea in cazuri este si buna si rea. Pe partea pozitiva, ea permite mai multe cazuri pentru o singura actiune, asa cum sint cazurile pentru blanc, tab sau linie noua in acest exemplu. Dar implica deasemenea faptul ca, in mod normal, fiecare caz trebuie sa se termine cu un break, pentru a preveni ramificarea pe cazul urmator. Iesirea dintr-un caz in altul nu este buna, fiind inclinata spre dezintegrare atunci cind programul este modificat. Cu exceptia etichetelor multiple pentru un singur caz, aceste iesiri dintr-un caz in altul trebuie folosite cu economie.

Ca o problema de forma buna, puneti un break dupa ultimul caz (la noi, cazul default) chiar daca logic nu este necesar. Intr-o zi cind veti adauga la sfirsit un caz nou, aceasta bucatica de programare defensiva va va salva.

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 sint expresii. In majoritatea cazurilor, expr1 si expr3 sint 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, analog cu bucla DO din FORTRAN si PL/1. Cu toate acestea, analogia nu este perfecta, deoarece limitele unei bucle for pot fi alterate din interiorul buclei si variabila de control i isi pastreaza valoarea cind bucla se termina, indiferent cum. Deoarece componentele unei bucle for sint expresii arbitrare, buclele for nu sint limitate la progresii aritmetice. Cu toate acestea, este un prost stil de programare acela de a forta calcule neinrudite intr-o bucla for; mai bine rezervati-le pentru operatiile de control ale buclei.

Ca un exemplu mai mare, iata o alta versiune a functiei atoi pentru convertirea unui sir in echivalentul sau numeric. Acest exemplu este mai general; el opereaza cu blancuri optionale nesemnificative si cu semn optional +/-. (Capitolul 4 va descrie functia atof care face aceeasi conversie pentru numere flotante).

Structura de baza a programului reflecta forma intrarii:

```
sari peste spatiile albe , daca exista
ia semnul, daca exista
ia partea intreaga, converteste-o
```

Fiecare pas isi face partea lui si lasa lucrurile intr-o stare curata pentru urmatorul. Intregul proces se termina la primul caracter care nu poate fi parte a unui numar.

```
atoi(s) /* converteste pe s in intreg */
char s[];
{
        int i, n, sign;
        for (i = 0; s[i] ==' ' || s[i] =='\n' || s[i] == '\t';i++)
                ; /* sari spatiile albe */
        sign = 1;
        if (s[i] == '+' || s[i] == '-')         /* semnul */
                sign = (s[i++] == '+') ? 1 : -1;
        for (n = 0; s[i] >= '0' && s[i] <= '9'; i++)
                n = 10 * n + s[i] - '0';
        return(sign * n)
}
```

Avantajul pastrarii centralizate a controlului buclei este si mai clar atunci cind exista mai multe bucle imbricate. Urmatoarea functie este o sortare shell pentru un tablou de intregi. Ideea de baza a sortarii shell este aceea ca in stadiile de inceput se compara elemente indepartate si nu cele adiacente, ca in sortarile simple bazate pe interschimbare. Aceasta tinde sa elimine cantitati mari de dezordine, rapid, asa ca stadiile urmatoare au mai putin de lucru. Intervalul dintre elementele comparate scade treptat spre unu, punct in care sortarea devine efectiv o metoda de interschimbare adiacenta.

```
shell(v, n)  /* sorteaza v[0], ...,v[n-1] in ordine crescatoare */
int v[], n;
{
```

```
        int gap, i, j, temp;
        for (gap = n / 2; gap > 0; gap /= 2)
          for (i = gap; i < n; i++)
            for (j = i - gap; j>= 0 && v[j] > v[j+gap]; j -= gap){
                temp = v[j];
                v[j] = v[j + gap];
                v[j + gap] = temp;
          }
}
```

Sint aici trei bucle imbricate. Cea mai dinafara contoleaza distanta dintre elementele comparate, contractind-o de la n/2 prin injumatatire la fiecare pas, pina cind devine zero. Bucla din mijloc compara fiecare pereche de elemente care este separata de un "gap"; bucla cea mai din interior le inverseaza pe acele elemente care nu sint in ordine. Deoarece "gap" poate fi redus la 1 eventual, toate elementele sint eventual ordonate corect. Sa notam ca generalitatea lui "for" face ca bucla exterioara sa aiba aceeasi forma ca celelalte, chiar daca nu este o progresie aritmetica.

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 sint 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 sint 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. Cu toate acestea, ea este din timp in timp valoroasa, ca in exemplul urmator, unde functia itoa converteste un numar intr-un sir de caractere (inversa lui atoi). Lucrarea este putin mai complicata decit se pare la prima vedere, deoarece metodele usoare de generare de cifre le genereaza intr-o ordine gresita. Am ales calea de a genera sirul invers apoi de a-l inversa.

```
itoa (n, s)  /* converteste pe n in caractere in s */
char s[];
int n;
{
        int i, sign;
        if ((sign = n) < 0)  /* semnul inregistrarii */
           n = -n;         /* face pe n pozitiv */
        i = 0;
        do {    /* genereaza cifre in ordine inversa */
             s[i++] = n % 10 +'0';/*ia uramatoarea cifra*/
        }
        while ((n /= 10) > 0);        /* sterge-o */
        if (sign < 0)
            s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}
```

Bucla do-while este necesara, sau cel putin convenabila deoarece cel putin un caracter trebuie pus in matricea s, indiferent de valoarea lui n. Am folosit de asemenea acoladele in jurul singurei instructiuni ce compune corpul buclei do-while, chiar daca nu sint necesare pentru ca cititorul grabit sa nu considere gresit partea cu while ca fiind inceputul unei bucle while.

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 sint 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 sint 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. Noi nu am folosit goto in aceasta carte.

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 beleaua
```

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

CAPITOLUL 4. FUNCTII SI STRUCTURA PROGRAMULUI

Functiile sparg programele cu calcule mari in mai multe programe mai mici, si permit oamenilor sa construiasca incepind de la ceea ce au facut altii deja, in loc de a porni totul de la capat. Functiile potrivite pot ascunde adesea (parti) detalii ale operatiilor din parti ale programului pe care nu e nevoie sa le cunoastem, clarificind astfel intregul, si usurind osteneala de a face modificari.

Limbajul C a fost proiectat pentru a face functiile eficiente

si usor de folosit; programele C constau, in general mai degraba din numeroase functii mici decit din citeva functii mari. Un program poate fi rezident intr-unul sau mai multe fisiere sursa in orice mod convenabil; fisierele sursa pot fi compilate separat si incarcate impreuna, impreuna cu alte functii compilate anterior ce se gasesc in biblioteci. Nu vom intra in intimitatea procesului aici, deoarece detaliile variaza de la un sistem la altul.

Majoritatea programatorilor sint familiarizati cu functiile "de biblioteca" pentru intrari si iesiri (getchar, putchar) si calculele numerice (sin, cos, sqrt). In acest capitol vom prezenta mai multe despre scrierea de noi functii.

4.1. Notiuni de baza

Pentru a incepe, haideti sa scriem un program care imprima fiecare linie care ii este introdusa si care contine un "model" sau un sir de caractere. (Acesta este un caz special al programului utiliar UNIX "grep".) De exemplu, sa cautam modelul "the" in urmatoarele linii:

```
Now is the time
for all good
men to come to the aid
of their party.
```

care va produce urmatoarea iesire:

```
Now is the time
men to come to the aid
if their party.
```

Structura de baza a programului se imparte in exact trei parti:

```
while (mai exista o linie)
      if (linia contine modelul)
          tipareste-o
```

Cu toate ca se poate pune codul pentru toate acestea in rutina principala,o modalitate mai buna este aceea de a folosi structura naturala si de a face din fiecare parte o functie separata. Este mai usor sa ne ocupam de trei bucati mai mici decit de o bucata mare, deoarece detaliile nerelevante pot fi inmormintate in functii si sansa de a da interactiuni nedorite este minimalizata. Si bucatile pot fi utile chiar luate apoi separat.

"While (mai exista o linie) " este getline, o functie pe care am scris-o in Capitolul 1 iar "tipareste-o" este printf cu care deja am lucrat suficient. Aceasta inseamna ca nu trebuie sa scriem decit o rutina care decide daca linia contine vreo aparitie a modelului. Putem rezolva problema furind o proiectare din PL/1: functia index(s,t) returneaza pozitia sau indexul din sirul s in care incepe sirul t, sau -1, daca s nu-l contine pe t. Vom folosi 0 in loc de 1 ca pozitie de inceput pentru s, deoarece tablourile in C incep din pozitia 0. Cind, mai tirziu vom avea nevoie de o cautare de model mai sofisticata, nu avem decit

sa inlocuim "index"; restul codului ramine acelasi.

Data aceasta schita, restul programului este fara ascunzisuri. Iata acum programul intreg, asa ca puteti vedea cum se potrivesc bucatile impreuna. Doar ca acum, modelul care trebuie cautat este literal sir din argumentele lui index, care nu este cel mai general dintre mecanisme. Ne vom intoarce pe scurt pentru a discuta cum sa initializam tablourile de caractere si in Capitolul 5 vom arata cum se face modelul un parametru care este setat atunci cind programul este lansat in executie. Dam de asemenea, o noua versiune getline; gasim ca este instructiv sa o comparati cu cea din Capitolul 1.

```
#define    MAXLINE 1000
main()     /* gasiti toate liniile ce contin un model dat */
{
      char line[MAXLINE];
      while (getline(line, MAXLINE) > 0)
           if (index(line, "the") >= 0)
               printf("%s", line);
}
getline(s, lim) /* citeste linia in s, returneaza lungimea ei */
char s[];
int lim;
{
      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(s, t)    /* returneaza indexul lui t in s, -1 in lipsa */
char s[], t[];
{
   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);
}
```

Fiecare functie are forma

```
      nume (lista de argumente, daca exista)
      declaratii de argumente, daca exista
      {
              declaratii si instructiuni, daca exista
      }
```

Asa cum am sugerat, anumite parti pot sa lipseasca; functia minima este:

```
dummy()  {}
```

care nu face nimic. (O functie care nu face nimic este utila uneori ca loc pastrat pentru dezvoltari ulterioare in program). Numele functiei poate fi deasemenea precedat de un tip daca functia returneaza altceva decit o valoare intreaga; acesta este subiectul urmatorului capitol.

Un program este tocmai un set de definitii de functii individuale. Comunicarea intre functii este (in acest caz) facuta prin argumente si valori returnate de functii; ea poate fi facuta deasemenea, prin variabile externe. Functiile pot apare in orice ordine in fisierul sursa, si programul sursa poate fi spart in mai multe fisiere, pe cind o functie nu este sparta.

Instructiunea return este mecanismul de returnare a unei valori din functia apelata in apelant. Orice expresie poate urma dupa instructiunea return:

```
return (expresie)
```

Functia apelanta este libera sa ignore valoarea returnata daca doreste. Mai mult, nu e necesar sa existe nici o expresie dupa return; in acest caz, nici o valoare nu este returnata apelantului. Controlul este deasemenea returnat apelantului fara nici o valoare atunci cind executia "se continua dupa sfirsitul" functiei, atingind cea mai din dreapta paranteza. Nu este ilegal ci probabil un semn de necaz (deranj), daca o functie returneaza o valoar dintr-un loc si nici o valoare din altul. In orice caz "valoarea" unei functii care nu returneaza nici una, este sigur un gunoi. Verificatorul "lint" cauta si dupa astfel de erori.

Mecanismul prin care se compileaza si se incarca un program care rezida in mai multe fisiere sursa variaza de la un sistem la altul . Pe sistemul UNIX, de exemplu, comanda CC, mentionata in Capitolul 1, face lucrul acesta. Sa presupunem ca cele trei functii se gasesc in trei fisiere numite main.c, getline.c si index.c. Atunci comanda:

```
CC main,c,getline,c,index,c
```

compileaza cele trei fisiere, plaseaza codul obiect relocabil rezultat in fisierele main.o, getline.o si index.o si le incarca pe toate intr-un fisier executabil numit a.out.
Daca exista vreo eroare, sa spunem in main.c, fisierul poate fi recompilat singur si rezultatul incarcat cu fisierele obiect anterioare, cu comanda:

```
CC main.c getline.o index.o
```

Comanda CC foloseste conventia de notare ".c" spre deosebire de ".o" pentru a distinge fisierele sursa de fisierul obiect.

Exercitiul 4.1. Scrieti functia rindex(s, t) care returneaza

pozitia celei mai din dreapta aparitii a lui t in s, si -1 daca nu e nici una.

4.2. Functii care returneaza non-intregi

Pina acum, nici unul din programele noastre nu a continut vreo declaratieasupra tipului unei functii. Aceasta deoarece implicit o functie este declarata prin aparitia ei intr-o expresie sau instructiune, ca in:

```
while (getline(line, MAXLINE) > 0)
```

Daca un nume care nu a fost declarat apare intr-o expresie si este urmat de o paranteza stinga, el este declarat din context ca fiind un nume de functie. Mai mult, implicit se presupune ca o functie returneaza un int. Deoarece char se transforma in int in expresii, nu e nevoie sa declaram functiile care returneaza char. Aceste prezumtii acopera majoritatea cazurilor, inclusiv toate exemplele noastre de pina acum.

Dar ce se intimpla daca o functie trebuie sa returneze o valoare de alt tip ? Multe functii numerice, ca sqrt, sin, cos returneaza double; alte functii specializate returneaza alte tipuri. Pentru a ilustra modul lor de folosire vom scrie si vom folosi o functie atof(s) care converteste sirul s in echivalentul lui in dubla precizie; atof este o extensie a lui atoi , pentru care am scris in Capitolul 2 si in Capitolul 3; ea minuieste un semn optional si un punct zecimal, precum si prezenta sau absenta atit a partii intregi cit si a partii fractionare.(Aceasta nu este o rutina de conversie de intrari de inalta calitate; ar lua mult mai mult spatiu decit ne-am propus noi sa folosim).

In primul rind, atof insasi trebuie sa declare tipul valoarii pe care ea o returneaza, deoarece el nu este int. Deoarece float este convertit in double in expresii, nu are nici un rost sa spunem ca atof returneaza un float; putem la fel de bine sa facem uz de precizie suplimentara, sa declaram ca ea returneaza double. Numele tipului precede numele functiei, ca in:

```
double atof(s)    /* converteste sirul s in double */
char s[];
{
  double val, power;
  int i, sign;
  for (i = 0;s[i]== ' ' || s[i]== '\n' || s[i]== '\t';i++)
        ; /* sare spatiile albe (blanc, tab, linie noua) */
  sign = 1;
  if (s[i] == '+' || s[i] == '-')   /* semnul */
        sign = (s[i++] == '+') ? 1 : -1;
  for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + s[i] - '0';
  if (s[i] == '.')
        i++;
  for (power = 1; s[i] >= '0' && s[i] <= '9'; i++)  {
        val - 10 * val + s[i] - '0';
```

```
                power *= 10;
          }
          return(sign * val / power);
        }
```

In al doilea rind, si la fel de important, rutina apelanta trebuie sa specifice ca atof returneaza o valoare non-int. Declaratia este arata in urmatorul calculator primitiv de birou (adevarat simplu pentru bilantul de verificare de conturi de carti ?!) care citeste un numar pe linie, precedat optional de un semn si-l aduna la toate numerele anterioare, tiparind suma dupa fiecare intrare.

```
        define MAXLINE  100
        main()    /* calculator rudimentar de birou */
        {
                double sum, atof();
                char line[MAXLINE];
                sum = 0;
                while (getline(line, MAXLINE) > 0)
                      printf("\t%.2f\n", sum += atof(line));
        }
```

Declaratia

```
        double sum, atof();
```

spune ca sum este un double si ca atof este o functie care returneaza o valoare double. Ca mnemonica, ea sugereaza ca sum si atof(...) sint amindoua valori flotante in dubla precizie.

In afara faptului cind atof este declarata explicit in ambele locuri, limbajul C presupune ca ea returneaza un intreg si raspunsurile primite de dumneavoastra vor fi de neinteles. Daca atof insasi si apelul ei din main au tipuri inconsistente in acasi fisier sursa, acest lucru va fi depistat de catre compilator. Dar daca (si asta e mai probabil) atof se compileaza separat, nepotrivirea nu va fi detectata si atof va returna un double pe care main il va trata ca intreg rezultind raspunsuri imprevizibile (lint prinde si aceste erori). Dat atof, putem scrie in principiu atoi (conversie de sir in intreg) astfel:

```
        atoi(s)         /* conversie sir s la intreg  */
        char s[];
        {
                double atof();
                return(atof(s));
        }
```

Sa remarcam structura declaratiilor si a instructiunii return. Valoarea expresiei din:

```
        return (expresie)
```

este intodeauna convertita in tipul functiei inainte ca rezultatul sa aiba loc. Deci valoarea lui atof, un double este convertita automat in int, cind apare intr-o instructiune return, deoarece functia atoi returneaza un int. (Conversia unei valori flotante intr-un intreg trunchiaza orice parte fractionara, asa cum am vazut in Capitolul 2).

Exercitiul 4.2. Extindeti functia atof astfel incit ea sa minuiasca si notatia stiintifica de forma 123.45e-6 in care un numar flotant poate fi urmat de e sau E si optional de un exponent cu semn.

4.3. Despre argumentele functiilor

In Capitolul 1 am discutat faptul ca argumentele functiilor sint trimise prin valoarea, adica functia apelata primeste o copie temporara si privata a fiecarui argument si nu adresele lor. Aceasta inseamna ca functia nu poate afecta argumentul original din functia apelanta. Intr-o functie argument este de fapt o variabila locala initializata cu valoarea cu care functia a fost apelata.

Cind un nume de tablou apare ca argument al unei functii locatia de inceput a tabloului este cea trimisa; elementele nu sint copiate. Functia poate altera elementele tabloului indexind cu aceasta valoare. Efectul este ca tablourile sint trimise prin referinta. In capitolul 5 vom discuta folosirea pointerilor pentru a permite functiilor sa nu altereze tablourile din functiile apelante.

Fiindca veni vorba, nu este un mod intrutotul satisfacator acela de a scrie o functie portabila care acepta un numar variabil de argumente, deoarece nu exista nici o modalitate portabila pentru functia apelata sa determine cite argumente i-au fost trimise actual intr-un apel dat. Astfel, nu puteti scrie o functie portabila intr-adevar de argumente, asa cum sint functiile MAX scrise in FORTRAN sau PL/1.

Este in general sigur sa ne ocupam cu un numar variabil de argumente daca functia apelata nu foloseste un argument care nu a fost furnizat efectiv si daca tipurile sint consistente. printf, cea mai comuna functie in C cu un numar variabil de argumente, foloseste informatia din primul sau argument pentru a determina cite alte elemente sint prezente si care sint tipurile lor. Ea esueaza urit daca apelantul nu furnizeaza suficiente argumente sau daca tipurile nu sint cele specificate de primul argument. Ea este deasemenea neportabila si trebuie modificata pentru diferite calculatoare.

Reciproc, daca argumentele sint de tip cunoscut, este posibil sa marcam sfirsitul listei de argumente intr-un mod corespunzator. De exemplu cu valoare de argument speciala (adresa0) care specifica sfirsitul listei de argumente.

4.4. Variabile externe

Un program C consta dintr-o multime de obiecte externe, care sint functii sau variabile. Adjectivul "extern" este folosit

in primul rind in contrast cu "intern", care descrie argumentele si variabilele automate definite in interiorul functiilor. Variabilele externe sint definite in afara oricarei functii si sint astfel disponibile potential pentru mai multe functii. Functiile insesi sint intodeauna externe, deoarece limbajul C nu permite definitii de functii in interiorul altor functii. Implicit variabilele externe sint deasemenea "globale", astfel incit toate referintele la o astfel de variabila printr-un acelasi nume (chiar si pentru functiile compilate separat) sint referinte la un acelasi lucru. In acest sens, varaiabilele externe sint analoage cu COMMON din FORTRAN si cu EXTERNAL din PL/1. Vom vedea mai incolo cum se pot defini variabile si functii externe care nu sint global disponibile ci sint vizibile ,in in schimb, doar intr-un singur fisier sursa.

Deoarece variabilele externe sint global accesibile, ele ofera o alternativa la argumente de functii si valori returnate pentru comunicari de date intre functii. Orice functie poate accede o variabila externa prin referirea numelui ei, daca numele a fost declarat undeva sau cumva.

Daca un numar mare de variabile trebuie sa fie partajat folosite de mai multe functii, variabilele externe sint mai convenabile si mai eficiente decit listele lungi de argumente. Asa cum am precizat in capitolul 1, aceasta modalitate trebuie, totusi, utilizata cu grija, deoarece ea poate avea efecte negative asupra structurii programului si poate conduce la programe cu multe conexiuni de date intre functii.

Un al doilea motiv pentru folosirea variabilelor externe priveste initializarea. In particular, tablourile externe pot fi initializate dar tablourile automate nu pot. Vom trata initializarea aproape de sfirsitul acestui capitol.

Al treilea motiv pentru folosirea varaiabilelor externe este domeniul si timpul lor de viata. Variabilele autmate sint interne unei functii; ele capata viata atunci cind rutina este introdusa si dispar atunci cind rutina se termina. Variabilele externe, pe de alta parte, sint permanente. Ele nu vin si pleaca, asa ca ele retin valorile de la un apel de functie la altul. Deci, daca doua functii trebuie sa-si partajeze niste date, chiar nefolosite de alte functii niciodata, este adesea mai convenabil daca datele partajabile sint pastrate in variabile externe decit trimise via argumente.

Sa examinam aceasta chestiune mai departe cu un exemplu mai mare. Problema consta in a scrie un alt program calculator, mai bun decit cel anterior. Aceasta va permite +,-,*,/ si = (pentru a tipari rezultatul). Deoarece este intrucitva mai usor de implementat, calculatorul va folosi notatia poloneza inversa in locul celei "infix". (Notatia poloneza este schema folosita, de exemplu, de calculatoarele de buzunar Hewlett-Packard) In notatia poloneza inversa, fiecare operator isi urmeaza operanzii; o expresie "infix", de tipul:

(1 - 2) * (4 + 5) =

se introduce astfel:

```
1 2 - 4 5 + * =
```

Parantezele nu sint necesare.

Implementarea este aproape simpla. Fiecare operand este depus intr-o stiva. Cind soseste un operator, numarul de operanzi (doi pentru operatorii liniari) sint scosi din stiva si li se aplica operatorul iar rezultatul este depus din nou in stiva. In exemplul de mai sus, 1 si 2 sint depusi in stiva, apoi sint inlocuiti de diferenta lor, -1 . Apoi 4 si 5 sint depusi in stiva, apoi sint inlocuiti de suma lor ,9. Produsul lui -1 cu 9, ii inlocuieste apoi in stiva. Operatorul = tipareste elementul din virful stivei fara a-l distruge (se pot face astfel verificari intermediare).

Operatiile de introducere si extragere din stiva sint triviale dar, daca se adauga detectia de erori de timp si recuperarea lor, codurile sint suficient de lungi pentru a fi mai bine sa le punem in functii separate decit sa repetam codul de-a lungul intregului program. La fel, vom considera o functie separata pentru aducerea urmatorului operand sau operator de la intrare. Astfel, structura programului este

```
while (urmatorul operator sau operand nu este sfirsitul de fisier)
          if (numar)
              pune-l in stiva
          else if (operator)
                  scoate operanzii din stiva
                  executa operatia
                  extrage rezultatul
          else
               eroare
```

Decizia principala de proiectare care nu a fost inca discutata este asupra locului stivei, adica ce rutina o poate accede direct. O posibilitate este aceea de a o tine in main si sa trecem stiva si pozitia ei curenta rutinelor care o folosesc pentru introducere si extragere de date. Dar main nu are nevoie sa stie despre variabilele care controleaza stiva; ea va trebui sa gindeasca numai in termeni de introducere si extragere in/din stiva. Asa ca am decis sa facem stiva si informatiile asociate ei drept variabile externe accesibile functiilor de introducere si extractie, dar nu si lui main.

Traducerea acestei schite in cod este destul de simpla. Programul principal este in primul rind un mare comutator dupa tipul operatorului sau al operandului; aceasta este probabil cea mai tipica folosire a lui switch pe care am descris-o in Capitolul 3.

```
#define MAXOP  20       /* marime maxima operand, operator */
#define NUMBER '0'      /* semnul pentru numar gasit */
#define TOOBIG '9'      /* semnal pentru sir prea lung */
main()  /* calculator de birou cu sirul Polonez invers */
{
```

```
        int type ;
        char s[MAXOP];
        double op2, atof(), pop(), push();
        while ((type = getop(s, MAXOP)) != EOF)
          switch (type)  {
          case NUMBER:
                push(atof(s));
                break;
          case '+':
                push(pop() + pop());
                break;
          case '*':
                push(pop() * pop());
                break;
          case '-' :
                op2=pop();
                push(pop() - op2);
                break;
          case '/' :
                op2=pop();
                if (op2 != 0.0)
                    push(pop() / op2)
                else
                    printf("impartire cu zero \n");
                break;
          case '=':
                printf("\t%f\n", push(pop()));
                break;
          case 'c':
                clear();
                break;
          case TOOBIG:
                printf("%.20s ... e prea lung \n",s);
                break;
          default:
                prinf(" comanda necunoscuta %c\n", type);
                break;
          }
}
#define MAXVAL 100 /* marimea stivei */
int sp = 0;    /* pointerul de stiva */
double val[MAXVAL];   /* stiva */
double push(f)  /* depune pe f in stiva */
double f ;
{
        if(sp < MAXVAL)
             return(val[sp++] = f);
        else {
             printf("eroare: stiva plina\n");
             clear();
             return(0);
          }
}
double pop()  /* extrage elementul din virful stivei */
```

```
{
        if (sp > 0)
             return(val[--sp]);
        else {
             printf("eroare: stiva goala\n");
             clear();
             return(0);
        }
}
clear()   /* curata stiva */
{
        sp = 0 ;
}
```

Comanda c curata stiva cu ajutorul functiei clear care este folosita deasemenea si de catre functiile pop si push in caz de eroare. Ne vom intoarce imediat la getop.

Asa cum am aratat in Capitolul 1, o variabila este externa daca este definita in afara corpului oricarei functii. Astfel stiva si pointerul de stiva care trebuiesc partajate de catre push, pop si clear sint definite in afara acestor trei functii. Dar main insusi nu refera stiva sau pointerul de stiva (reprezentarea este ascunsa cu grija). Astfel, codul pentru operatorul = trebuie sa se foloseasca

```
        push(pop()));
```

pentru a examina virful stivei fara a-l distruge.

Sa notam deasemenea ca deoarece + si * sint operatori comutativi, orinea in care se combina operanzii scosi din stiva este irelevanta, dar pentru operatorii - si / trebuie sa distingem intre operanzii sting si drept.

Exercitiul 4.3. Dat scheletul de baza, este usor sa extindem programul calculator. Adaugati procentul % si operatorul unar -. Adaugati o comanda de stergere, care sterge elementul din virful stivei. Adaugati comenzi pentru minuirea de variabile (este usor in cazul variabilelor formate dintr-o singura litera (26)).

4.5. Reguli de domeniu

Functiile si variabilele externe care compun un program C nu trebuie sa fie compilate toate in acelasi timp; textul sursa al programului poate fi pastrat in mai multe fisiere iar rutinele compilate anterior pot fi incarcate din biblioteci. Cele doua intrebari care prezinta interes aici sint:

Cum sint scrise declaratiile astfel incit variabilele sa fie declarate cum se cuvine in timpul compilarii ?

Cum sint fixate declaratiile astfel incit toate piesele sa fie conectate cum se cuvine atunci cind programul este incarcat ?

Domeniul unui nume este acea parte de program in care numele este definit. Pentru o variabila automata declarata la inceputul unei functii, domeniul este functia in care numele este declarat si variabilele cu acelasi nume in functii diferite sint fara legatura unele cu altele. La fel se intimpla si cu argumentele functiilor .

Domeniul unei variabile externe dureaza din punctul in care ea este decalrata intr-un fisier sursa pina la sfirsitul acelui fisier. De exemplu, daca val,sp,push,pop,clear sint definite intru-un fisier in ordinea de mai sus, adica:

```
int sp = 0;
double val[MAXVAL];
double push(f) {...}
double pop() {...}
clear() {...}
```

atunci variabilele val si sp pot fi folosite in push ,pop si clear pur si simplu numindu-le; nu sint necesare declaratii suplimentare . Pe de alta parte, daca o variabila externa trebuie sa fie referita inainte de a fi definita sau este definita intr-un alt fisier sursa decit cel in care este folosita, arunci este necesara o declaratie"extern".

Este important sa distingem intre declaratia unei variabile externe si definitia sa. O declaratie anunta proprietatile unei variabile (tipul marimea, etc); o definitie provoaca in plus o alocare de memorie. Daca liniile:

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

apar in afara oricarei functii, ele definesc variabilele externe sp si val, provoaca o alocare de memorie pentru ele si servesc in plus ,ca declaratie pentru restul fisierului sursa. Pe de alta parte liniile

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

declara pentru restul fisierului sursa ca sp este un int si ca val este un tablou double (a carei dimensiune este determinata altundeva),dar ele nu creaza variabilele si nici nu aloca memorie pentru ele .

Trebuie sa existe o singura definitie pentru o variabila externa in toate fisierele care compun programul sursa; alte fisiere pot contine declaratii extern pentru a o accede. (Poate exista o declaratie extern si in fisierul ce contine definitia). Orice initializare a unei variabile externe se face numai in definitie. Dimensiunile de tablouri trebuie specificate cu definitia dar sint optionale cu o declaratie externa.

Cu toate ca nu este o organizare adecvata pentru acest pro-

gram ,val si sp pot fi definite si initializate intr-un fisier iar functiile push, pop si clear definite intr-altul. Aceste definitii si declaratii ar trebui legate impreuna astfel:

In fisierul 1:

```
        int sp=0; /* pointerul de stiva */
        double val[MAXVAL];  /* valoarea stivei */
```

In fisierul 2:

```
        extern int sp;
        extern double val[];
        double push(f) {...}
        double pop() {...}
        clear () {...}
```

Deoarece declaratiile extern din fisierul 2 se gasesc in fata si in afara celor trei functii, ele se aplica tuturora; un set de declaratii este suficient pentru tot fisierul 2.

Pentru programe mai mari, facilitatea de includere in fisier "#include" care va fi discutata mai tirziu in acest capitol, permite unui utilizator sa pastreze o singura copie a declaratiilor "extern" pentru programul dat si sa o insereze in fiecare fisier sursa care trebuie compilat.

Ne vom intoarce acum la implementarea lui getop, functia care aduce urmatorul operator sau operand. Lucrarea de baza este usoara: se sar blancurile, taburile si liniile noi. Daca urmatorul caracter nu este o cifra sau punctul zecimal, returneaza-l.Astfel, colecteaza un sir de cifre (care poate include si punctul zecimal) si returneaza NUMBER, care semnaleaza faptul ca s-a colectat un numar.

Rutina este complicata substantial de incercarea de a minui in mod potrivit situatia in care numarul de intrare este prea lung getop citeste cifrele (probabil si un punct zecimal) atita timp cit le gaseste dar le memoreaza numai pe acelea care incap. Daca numarul nu a fost prea lung (nu s-a produs o depasire) functia returneaza NUMBER si sirul de cifre. Daca numarul a fost prea lung totusi getop elimina restul liniei de intrare asa ca utilizatorul poate retipari simplu linia din punctul de eroare; functia returneaza TOOBIG drept semnal pentru depasire:

```
getop(s, lim)   /* obtine urmatorul operand sau operator */
char s[];
int lim;
{
        int i, c;
        while ((c = getch()) == ' ' || c == '\t' || c == '\n')
              ;
        if(c != '.' && (c < '0' || c>'9'))
              return(c);
        s[0] = c;
        for (i = 1; c = getchar()) >= '0' && c <= '9'; i++)
```

```
            if (i < lim)
                 s[i] = c;
        if (c == '.')  {    /*     fractia      */
            if (i < lim)
               s[i] = c;
            for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
                 if (i < lim)
                    s[i] = c;
        }
        if (i < lim)  {  /* numarul este ok */
            ungetch(c);
            s[i] = '\0';
            return(NUMBER);
        } else {   /* numar prea mare, se sare restul liniei */
            while (c != '\n' && c != EOF)
                  c = getchar();
            s[lim-1] = '\0';
            return(TOOBIG)
        }
}
```

Ce sint getch si ungetch ? Se intimpla adesea cazul ca un program care citeste date de intrare nu poate determina daca a citit destul pina cind a ajuns sa citeasca prea mult. Un exemplu este colectarea de caractere ce alcatuiesc un numar: pina cind nu se intilneste un caracter necifra, numarul nu este complet. Dar atunci programul a citit un caracter mult mai necesar, caracter ce nu a fost pregatit pentru aceasta.

Problema ar putea fi rezolvata daca ar fi fost posibil sa "nu citim" caracterul nedorit. Apoi, de fiecare data cind programul citeste un caracter prea mult, el il poate pune inapoi in intrare, asa ca restul codului se va comporta ca si cind nu a fost citit niciodata. Din fericire este usor de simulat necitirea unui caracter, scriind o pereche de functii de cooperare. getch descopera urmatorul caracter de intrare ce trebuie consi-derat; ungetch pune caracterul inapoi in intrare, asa ca urmatorul apel al lui getch il va returna din nou.

Modul in care lucreaza aceste functii impreuna este simplu. ungetch pune caracterul intr-un buffer partajabil un tablou de caractere ,getch citeste din buffer pentru a vedea daca exista vreun caracter si apeleaza pe getchar daca bufferul este vid. Trebuie deasemenea sa existe o variabila index care inregistrwaza pozitia caracterului curent din buffer.

Deoarece bufferul si indexul sint partajate de getch si ungetch si trebuie sa-si retina valorile lor intre apeluri, ele trebuie sa fie externe ambelor rutine. Deci putem scrie getch si ungetch precum si variablelelor partajate astfel:

```
        #define BUFSIZE 100
        char buf[BUFSIZE]; /* bufferul pentru ungetch */
        int bufp = 0  /* urmatoarea pozitie libera din buffer */
        getch()  /* ia un posibil caracter din buffer */
        {
          return((bufp > 0) ? buf[--bufp] : getchar());
```

```
        }
        ungetch(c)  /* pune caracterul la loc in intrare */
        int c;
        {
          if (bufp > BUFSIZE)
              printf("ungetch: prea multe caractere\n");
          else
            buf[bufp++] = c;
        }
```

Am folosit un tablou pentru buffer si nu un singur caracter deoarece generalitatea programului se va observa mai tirziu.

Exercitiul 4.4. Scrieti o rutina ungets(s) care va depune inapoi in intrare un sir intreg de caractere. Cum credeti ca ar fi mai bine , folosind ungetch sau folosind buf si bufp ?

Exercitiul 4.5. Presupunem ca in buffer nunva fi niciodata mai mult de un caracter. Modificati in consecinta getch si ungetch.

Exercitiul 4.6. Functiile noastre getch si ungetch nu minuiesc EOF-ul intr-un mod portabil. Decideti ce proprietati ar trebui sa aibe acestea pentru a minui un EOF apoi implementati-le.

4.6. Variabile statice

Variabilele statice sint a treia clasa de variabile, pe linga cele externe si cele automate, pe care le-am intilnit deja.

Variabilele de tip " static" pot fi atit interne cit si externe. Variabilele statice sint locale unei functii particulare la fel ca cele automate dar, spre deosebire de acestea, ele ramin in existenta (exista) tot timpul si nu apar si dispar de fiecare data cind functia este activa. Aceasta inseamna ca variabilele statice interne ofera un mijloc de alocare permanenta si privata de spatiu intr-o functie. Sirurile de caractere care apar intr-o fucntie, ca de exemplu argumentele lui printf, s sint statice interne.

O variabila statica externa este recunoscuta in rstul fisierului sursa in care este declarata, dar nu intr-un alt fisier. Variabilele externe statice ofera astfel o modalitate de a ascunde nume ca buf si bufp in combinatia getch-ungetch, care trebuie sa fie externe ca sa poata fi partajabile si care totusi nu sint vizibile pentru utiliztorii luigetch si ungetch, asa ca nu exista nici o posibilitate de conflict. Daca cele doua rutine si cele doua variabile sint compilate intr-un fisier, cin

```
        static char buf[BUFSIZE]; /* bufer pentru ungetch /
        static int bufp = 0; / urmatorea pozitie libera in buf */
        ...
        getch() {...}
        ungetch(c) {...}
```

atunci nici o alta rutina nu va fi in stare sa acceada buf si bufp; in fapt, ele nu intra in conflict cu late variabile cu

aceleasi nume din alte fisiere ale aceluiasi program .

Memorarea statica, atit cea interna cit si cea externa se specifica prefixind declaratia normala cu cuvintul "static". Variabila este externa daca este definita in afara oricarei functii si este interna daca este definita intr-o functie.

In mod normal, functiile sint obiecte externe; numele lor sint cunoscute global. Este posibil, totusi, ca o functie sa fie declarata "statica "; acest lucru face numele ei sa fie necunoscut inafara fisierului in care este declarat.

In limbajul C, "static" conteaza nu numai permanenta dar si un grad din ceea ce ar putea fi numit "taina". Obiectele interne statice sint cunoscute numai in interiorul unei functii ;obiectele externe statice (variabile sau functii) sint cunoscute numai in fisierul sursa in care apar, iar numele lor nu interfereaza cu variabile sau functii cu acelas si nume care apar in alte fisiere.

Variabilele statice externe si functiile sint o modalitate de a ascunde obiectele "date" si orice rutina interna care le manipuleaza astfel incit orice alta rutina sau data nu poate intra in conflict cu ele, nici macar din greseala. De exemplu, getch si ungetch formeaza un "modul" pentru introducera si extragerea de caractere; buf si bufp pot fi statice asa ca sint inaccesibile din afara. In acelasi mod, pish ,pop, clear formeaza un modul pentru lucrul cu stiva; val si sp pot fi statice externe !

4.7. Variabile registru

A patra si ultima clasa de stocari este denumita registru. O declaratie de registru avertizeaza compilatorul ca variabila in chestiune va fi folosita din greu. Cind este posibil, variabilele registru se plaseaza in registrii calculatorului; cea ce va genera programe mai scurte si mai rapide.
Declaratia de registru este de forma:

```
register int x;
register char c;
```

si asa mai departe; partea "int" poate fi omisa. Declaratia de registru poate fi aplicata numai variabilelor automate si parametrilor formali ai unei functii. In acest ultim caz, declaratia este de forma:

```
f(c,n)
register int c,n;
{
    register int i;
    ...
}
```

In practica exista anumite restrictii asupra vriabilelor registru , reflectind realitatea hardware-ului de suport. Numai citeva vriabile din fiecare functie pot fi pastrate in registri si numai anumite tipuri sint permise. Cuvintul "register" este ignorat cind apare in exces sau in declaratii nepermise. In plus, nu

este posibila aflarea adresei unei variabile registru (o topica ce va fi acoperita in capitolul 5). Restrictiile specifice variaza de la un calculator la altul; de exemplu pentru PDP11, numai primele trei declaratii de registru sint efective intr-o functie iar tipurile lor pot fi int,char, sau pointer.

4.8. Structura de bloc

Limbajul C nu este un limbaj structurat pe bloc in sensul lui PL/1 sau ALGOL, adica functiile nu pot fi definite in alte functii.

Pe de alta parte, variabilele pot fi definite intr-o maniera "structura de bloc". Declaratiile de variabile (incluzind initializarile) pot urma dupa paranteza stinga care introduce orice instructiune compusa si nu numai dupa cea care incepe o functie. Variabilele declarate in aceasta maniera acopera variabilele numite identic in blocurile mai din afara si ramin in existenta pina cind intilnesc o paranteza dreapta. De exemplu

```
if (n > 0) {
    int i;   /* declara un nou i */
    for (i = 0; i < n; i++)
        ...
}
```

domeniul variabilei i este intreaga ramura a lui if; acest i nu are nici o legatura cu oricare alt i din program. Structura de bloc se aplica deasemenea variabilelor externe.

Date declaratiile:

```
int x;
f()
{
    double x;
    ...
}
```

atunci, in cadrul functiei f, occurentele lui x se refera la variabila interna double, in afara lui f, ele se refera la externul integer. La fel se intimpla lucrurile si cu numele de parametri formali :

```
int z;
f(z)
double z;
{
    ...
}
```

In cadrul functiei f, z se refera la parametrul formal, si nu la z-ul extern.

4.9. Initializare

Initializarea a fost mentionata in trecere de mai multe ori pina acum, dar intodeauna in trecere si in legatura cu alte subiecte. Aceasta sectiune rezuma unele din reguli, dat fiind faptul ca pina acum am discutat mai multe clase de tipuri de memorari.

In absenta initializarii explicite, variabilele externe si statice se initializeaza pe zero; variabilele automate si de registru sint nedefinite (i.e. gunoi, ramasita). Variabilele simple (nu tablo-urile sau structurile) pot fi initializate cind se declara, punind in continuarea numelui lor semnul egal si o expresie con-stanta:

```
int  x = 1;
char squote = '\";
long day = 60 * 24;   /* minute intr-o zi */
```

Pentru variabilele externe si statice, initializarea se face o data ,la compilare. Pentru variabilele automate si registru, ini-tializarea se face de fiecare data cind functia sau blocul se executa .

Pentru variabilele automate si de registru valoarea de ini-tializare nu trebuie sa fie o constanta: poate fi de fapt orice expresie valida implicind valori definite anterior, chiar si de apeluri de functii. De exemplu initializarile din programul de cautare binara din capitolul 3 pot fi scrise astfel :

```
binary (x, v, n)
int x, v[], n;
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

in loc de:

```
binary (x, v, n)
int x, v[], n;
{
    int low, high, mid;
    low = 0;
    high = n - 1;
    ...
}
```

In fapt, initializarile de variabile automate sint prescurtari pentru instructiunile de asignare. Care forma este de preferat este in ultima instanta o chestiune de gust. In general noi am preferat asignarile explicite, deoarece initializarile in declara-

tii sint mai greu de vazut.

Tablourile automate nu pot fi initializate. Tablourile externe si statice pot fi initializate punind dupa declaratie o lista de valori de initializare inclusa intre paranteze si separate prin virgule. De exemplu programul de contorizare caractere dat in capitolul 1, care incepea

```
main()  /* contorizeaza cifre, blancuri, altele */
{
    int c, nwhite, nother;
    int ndigit[10];
    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
         ndigit[i] = 0;
         ...
}
```

poate fi scris si astfel:

```
int nwhite = 0;
int nother = 0;
int ndigit[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } ;
main()   /* contorizeaza cifre, blancuri, altele */
{
    int c, i;
    ...
}
```

Aceste initializari sint de fapt necesare deoarece sint toate zero dar este o buna practica de programare de a le da explicit. Daca valorile de initializare specificate sint mai putine decit marimea specificata, restul valorilor vor fi zero. Daca ele sint mai multe se provoaca eroare . Este regretabil insa faptul ca nu putem specifica nicicum repetitia unei valori de initializare si nici sa initializam un element din mijlocul unui tablou fara a initializa si toate elementele care-l preced.

Tablourile de caractere sint un caz special de initializare. In locul notatiei cu paranteze si virgule se poate folosi un sir de caractere:

```
char pattern[] = "the"
```

Aceasta este o prescurtare pentru forma echivalenta dar mai lunga:

```
char pattern[] = { 't', 'h', 'e', '\0' };
```

Cind marimea unui tablou de orice tip este omisa, compilatorul va calcula lungimea contorizind valorile de intializare. In acest caz specific marimea tabloului este 4 (trei caractere plus terminatorul \0)

4.10 Recursivitate

Functiile din C pot fi folosite recursiv. Aceasta inseamna ca

o functie se poate apela pe insasi, fie direct fie indirect. Un exemplu traditional este cel relativ la tiparirea unui numar ca si sir de caractere. Asa cum am mentionat mai inainte, cifrele sint generate intr-o ordine gresita: cele mai putin semnificative sint dispuse inaintea celor mai semnificative iar tiparirea lor se face invers.

Exista doua solutii pentru aceasta problema. Una este de a memora cifrele intr-un tablou asa cum au fost generate, apoi sa le tiparim in ordine inversa asa cum am facut in cap 3 cu itoa. Prima versiune a lui printd foloseste acest model.

```
printd(n)       /* print n in decimal */
int n;
{
        char s[10];
        int i;
        if (n < 0)  {
            putchar('-');
            n = -n;
        }
        i = 0;
        do  {
            s[i++] = n % 10 + '0';   /* get next char */
        } while ((n /= 10) > 0);  /* discard it */
        while (--i >= 0)
            putchar(s[i]);
}
```

Alternativa este o solutie recursiva, in care fiecare apelare a lui printd intii se autoapeleaza pentru a trata cifrele din fata, apoi tipareste cifra din coada.

```
printd(n)  /* print n in decimal(recursive) */
int n;
{
        int i;
        if (n < 0)  {
            putchar('-');
            n = -n;
        }
        if ((i = n/10) != 0)
            printd(i)
        putchar(n % 10 + '0');
}
```

Cind o functie se autoapeleaza fiecare invocare genereaza un set proaspat de variabile automate absolut independent de setul precedent. Astfel in printd(123) primul printd are n=123. Acesta trece 12 celui de-al doilea printd, apoi tipareste 3 cind acesta din urma revine. In axcelasi fel, urmatorul printd trece 1 la al treilea apoi tipareste 2.

Recursivitatea nu duce in general la economie de memorie atita timp cit trebuie mentinuta o stiva cu valorile ce urmeaza a fi

procesate . Codul recursiv este mai compact si adesea mai usor de scris si inteles. Recursivitatea este convenabila in special pt structuri de date recursive precum arborii; vom vedea un exemplu dragut in capitolul 6.

Exercitiul 4-7 Adaptati ideile de la printd pt a scrie o versiune recursiva a lui itoa; adica de a converti un intreg intr-un sir printr-o rutina recursiva.

Exercitiul 4-8 Scrieti o versiune recursiva a functiei reverse(s) care inverseaza sirul s.

4.11 Preprocesorul C

C admite unele extensii de limbaj cu ajutorul unui simplu macropreprocesor. Posibilitatile lui #define sint cele mai obisnuite exemple despre aceste extensii; alta este posibilitatea de a include continutul altor fisiere in timpul compilarii.

Includerea fisierelor

Pentru a usura manipularea colectii de #define si declaratii (printre altele) C admite includerea fisierelor. Orice linie de tipul

```
#include "filename"
```

este inlocuita prin continutul fisierului "filename". Adesea o linie sau doua de aceasta forma apar la inceputul fiecarui fisier sursa pentru a include declaratiile #define comune si declaratiile extern pentru variabilele globale. #include-urile pot fi grupate.

#include este calea preferata pentru a uni declaratiile impreuna pt un program mai mare. Aceasta garanteaza ca toate fisierele sursa vor fi alimentate cu aceleasi definitii si declarari de varaibile. Desigur atunci cind un fisier inclus este schimbat toate fisierele dependente trebuiesc recompilate.

Macro substituirea

O definitie de forma

```
#define YES 1
```

apeleaza o macrosubstituire de cea mai simpla forma -inlocuirea unui nume cu un sir de caractere. Numele din #define au aceasi forma ca si identificatorii din "C"; textul de inlocuire este restul liniei, o definitie lunga se poate continua prin plasarea unui \ la sfirsitul liniei de continuat. Domeniul unui nume definit prin #define este de la locul definirii pina la sfirsitul fisierului sursa. Numele pot fi redefinite si o definire poate folosi definirii precedente. Substitutiile nu se pun intre ghilimele, astfel daca YES este un nume definit, nu va avea loc nici o substituire in printf ("YES").

Deoarece implementarea lui #define nu este o parte a compilatorului propriuzis, sint foarte putine restrictii asupra a ce poate fi definit. De exemplu adeptii Algolului pot spune:

```
#define then
#define begin {
#define end   ;}
```

si apoi sa scrie:

```
if (i > 0) then
      begin
          a = 1;
          b = 2
      end
```

Este de asemenea posibil de definit macrouri cu argumente, astfel ca textul de inlocuire depinde defelul in care macroul este apelat. De exemplu sa definim un macro numit max astfel:

```
#define max(A, B)   ((A) > (B) ? (A) : (B))
```

Acum linia

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

va fi inlocuita de linia:

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

Aceasta admite o functie maxima care se expadeaza intr-un cod "inline" si nu intr-o apelare de functie. Atita vreme cit argumentele sint tratat adecvat, acest macro va servi pt orice tip de date; nu exista diferite tipuri de max pt diferite tipuri de date, asa cum se intimpla cu functiile.

Desigur, daca examinati expansiunea lui max de mai sus veti observa citeva capcane. Expresiile sint evaluate de doua ori; aceasta este rau daca sint impicate efecte colaterale ca apelari de functii si operatori de incrementare. Masuri de prevedere trebuie luate cu parantezele pentru a fi siguri ca ordinea de evaluare este respectata. (Considerati macro-ul

```
#define square(x) x * x
```

cind se apeleaza ca square(z+1).)

Exista chiar probleme lexicale; nu pot exista spatii intre numele macroului si paranteza stinga care introduce lista de argumente.

Insa fara indoiala, macro-urile sint suficient de valoroase. Un exemplu practic este biblioteca standard I/O care va fi deschisa in capitolul 7 ,in care getchar si putchar sint definite ca mcrouri, pt a evita apelarea unei functii pentru fiecare caracter procesat .

Alte posibilitati ale macropocesorului sint descrise in Apendix A.

Exercitiul 4.9 Definiti un macro swap(x,y) care schimba intre ele toate cele 2 argmente int.

CAPITOLUL 5. POINTERI SI TABLOURI

Un pointer este o variabila care contine adresa unei alte variabile. Pointerii sint foarte mult utilizati in C parte pentru ca uneori sint singura cale de rezolvare a unei anumite probleme, parte pentru ca folosirea lor duce la alcatuirea unui cod mai compact si mai eficient decit altul obtinut in alt mod.

Pointerii au fost "ingramaditi" la citeva instructiuni goto ca un minunat mod de a crea programe "imposibil" de priceput. Acest lucru devine pe deplin adevarat atunci cind pointerii sint folositi neatent, fiind usor de creat pointeri care sa pointeze in locuri cu totul neasteptate. Cu metoda, pointerii se utilizeaza pt un plus de simplitate. Aceasta este aspectul pe care vom incerca sa-l ilustram.

5.1. Pointeri si adrese

Din moment ce un pointer sustine adresa unui obiect, este posibila adresarea acelui obiect "indirect" prin intermediul pointerului. Sa presupunem ca x este o variabila, sa spunem int si ca px este un pointer creat intr-un mod neprecizat. Operatorul & da adresa unui obiect, astfel incit instructiunea

```
px=&x
```

asigneaza variabilei px adresa lui x acum, px inseamna "pointeaza pe x". Operatorul & poate fi aplicat numai variabilelor si elementelor unui tablou, constructii ca &(x+1) si &3 sint interzise. Este deasemenea interzisa pastrarea adresei unei variabile registru.

Operatorul unar * testeaza operandul sau ca adresa ultimului semnal si acceseaza aceasta adresa pentru a aduce continutul locatiei de la adresa respectiva. Astfel, daca y este tot un int

```
y = *px
```

asigneaza lui y, ori de cite ori este cazul continutul locatiei unde pointeaza px. Astfel secventa

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

asigneaza lui y aceasi valoare ca si

```
y = x
```

Totodata este necesara declararea variabilelor care apar in secventa:

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

Declararea lui x si y este deja cunoscuta. Noua este declararea pointerului px

```
int *px
```

este interpretata ca o mnemonica; aceasta inseamna ca *px este un int, adica in momentul in care px apare in context sub forma *px, este echivalenta cu a intilni o variabila de tip int. De fapt, sintaxa declararii unei varaibile imita sintaxa expresiilor in care ar putea sa apara respectiva variabila. Acest rationament este util in toate cazurile care implica declaratii complicate. De exemplu:

```
double atof(), *dp;
```

spune ca intr-o expresie atof() si *dp au valoare de tip double.

De notat declaratia implicita, ceea ce vrea sa insemne ca un pointer este constrins sa pointeze o anumita categorie de obiecte. (Functie de tipul obiectului pointat).

Pointerii pot aparea in expresii. De exemplu, daca px pointeaza pe intregul x atunci *px poate aprarea in orice context in care ar putea apare x.

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

da lui y o valoare egala cu x plus 1.

```
printf("%d\n", *px)
```

imprima o valoare curenta a lui x si

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

face ca d = radical din x, care este fortat de tipul double inainte de a fi transmis lui sqrt (vezi capitolul 2).

In expresii ca

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

operatorii unari * si & au prioritate mai mare decit cei aritmetici, astfel aceasta expresie ori de cite ori pointerul px avanseaza, aduna 1 si asigneaza valoarea lui y. Vom reveni pe seama asupra a ceea ce inseamna

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

Referiri prin pointer pot apare si in partea stinga a asignarilor. Daca px pointeaza pe x atunci

```
        *px = 0
```

il pune pe x pe zero si

```
        *px += 1
```

il incrementeaza pe x, ca si

```
        (*px)++
```

In acest ultim exemplu parantezele sint necesare; fara ele, expresia va incrementa pe px in loc sa incrementeze ceea ce pointeaza px deoarece operatorii unari * si + sint evaluati de la dreapta la stinga.

In sfirsit, daca pointerii sint variabile, ei pot fi manipulati ca orice alta variabila. Daca py este un alt pointer pe int, atunci

```
        py = px
```

copiaza continutul lui px in py facind astfel ca py sa varieze odata cu px.

5.2. Pointeri si argumente de functii

Datorita faptului ca in C este posibila transmiterea de argumente unei functii prin "apel prin valoare" nu exista modalitate directa pentru functia apelata de a altera o variabila in functia apelanta. Ce este de facut atunci cind de fapt, se intentioneaza schimbarea unui argument obisnuit ? De exemplu, o rutina de sortare trebuie sa inverseze doua elemente neordonate cu o functie swap. Nu este suficient sa se scrie

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

unde functia swap este definita ca

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

Din cauza apelului prin valoare, swap nu poate afecta argumentele a si b in rutina care o apeleaza.

Din fericire, exista o modalitate de a obtine efectul dorit.

Programul apelant trasmite pointeri pe valorile care trebuie schimbate.

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

Din moment ce operatorul & da adresa unei variabile, &a este un pointer pe a. In swap insasi, argumentele sint declarate ca fiind pointeri iar adevaratii operanzi sint accesati prin ei (prin pointeri).

```
swap(px, py) /* interscimba *px si *py */
int *px, *py;
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

O utilizare comuna a argumentelor de tip poiter se intilneste in cadrul functiilor care trebuie sa returneze mai mult decit o singura valoare. (Veti putea obiecta ca swap returneaza doua valori, noile valori ale argumentelor sale.) Ca un exemplu sa luam o functie getint care realizeaza inversia la intrare prin transsformarea unui sir de caractere in valori intregi, un intreg la fiecare apel, getint trebuie sa returneze valoarea gasita sau semnul de sfirsit de fisier atunci cind s-a terminat sirul de caractere de la intrare. Aceste valori trebuie sa fie returnate ca obiecte separate, pentru indiferent ce valoare este utilizata pentru EOF aceasta putind fi deasemenea valoarea unui intreg-input

O solutie, care este bazata pe functia input scanf, functie care o vom descrie in cap7, este de a folosi getint care sa returneze ca valoare o functie EOF, atunci cind se intilneste sfirsitul de fisier; orice alta valoare returnata inseamna ca a fost prelucrat un intreg obisnuit Valoarea numerica a intregului gasit este returnata printr-un argument care trebuie sa fie pointer pe un intreg. Aceasta organizare separa starea de sfirsit de fisier de valorile numerice.

Urmatoarea bucla completeaza un tablou cu intergi prin ape-luri la get int.

```
int n, v, array[SIZE]
for (n = 0; n < SIZE && getint(&v) != EOF; n++)
     array[n] = v;
```

Fiecare apel pune pe y pe urmatorul intreg gasit la intrare. De notat faptul ca este esential a scrie &y in loc de y, ca arg al lui getint. A scrie doar y constituie eroare de adresare, getint sustinind ca are de a face cu un pointer propriu zis.

Insasi getint este o modificare evidenta a lui atoi tratata mai inainte.

```
getint(pn)   /* ia numarul interg -input */
int *pn;
```

```
{
  int c, sign;
  while ((c = getch() == ' ' || c == '\n' || c == '\t');
                                          /*sare blancuri*/
  sign = 1;
  if (c == 't' || c == '-') {             /* semnul */
      sign = (c == '+' ? 1 : -1;
      c = getch();
  }
  for (*pn = 0; c >= '0' && c <= '9'; c = getch())
      *pn = 10 * *pn + c - '0';
  *pn *= sign;
  if (c != EOF)
      ungetch(c);
  return(c);
}
```

Peste tot in getint, *pn este utilizat ca o variabila int ordinara. Deasemenea, am utilizat getch si ungetch (descrise in cap 4) in asa fel incit caracterul special (semnalul EOF) care trebuie citit sa poata fi restocata la intrare.

Exercitiul 5.1 Scrieti getfloat pentru virgula floatnta analoaga lui getint. Ce tip de valoare returneaza functia getfloat.

5.3. Pointeri si tablouri

In C, exista o relatie strinsa intre pointeri si tablouri, atit de strinsa incit pointerii si tablourile pot fi tratate simultan. Orice operatie care poate fi rezolvata prin indicierea tablourilor poate fi rezolvata si cu ajutorul pointerilor. Versiunea cu pointeri va fi in general, mai rapida dar, pentru incepatori, mai greu de inteles imediat.

Declaratia

```
int a[10]
```

defineste un tablou de dimensiunea 10, care este un bloc de 10 obiecte consecutive numite a[0], a[1], ..., a[9] notatia a[i] desemneaza elementul deci pozitiile, ale tabloului, numarate de la inceputul acestuia. Daca pa este un pointer pe un interg, decalarat ca

```
int *pa
```

atunci asignarea

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

face ca pa sa pointeze pe al "zero-ulea" element al tabloului a; aceasta inseamna ca pa contine adresa lui a[0]. Acum asignarea

```
x = *pa
```

va copia continutul lui a[0] in x.

Daca pa pointeaza pe un element oarecare al lui a atunci prin definitie pa+1 pointeaza pe elemmentul urmator si in general pa-i pointeaza cu i elemente inaintea elementului pointat de pa iar pa+i pointeaza cu i elemente dupa elementul pointat de pa. Astfel, daca pa pointeaza pe a[0]

```
*(pa + 1)
```

refera continutul lui a[1], pa + i este adresa lui a[i] si *(pa+i) este continutul lui a[i].

Aceste remarci sint adevarate indiferent de tipul varaiabilelor din tabelul a. Definitia "adunarii unitatii la un pointer " si prin extensie, toata aritmetica pointerilor este de fapt calcularea prin lungimea in memorie a obiectului pointat. Astfel, in pa+i i este inmultit cu lungimea obiectelor pe care pointeaza pa inainte de a fi adunate la pa.

Corespondenta intre indexare si aritmetica pointerilor este evident foarte strinsa. De fapt, referinta la un tablou este convertita de catre compilator intr-un pointer pe inceputul tabloului. Efectul este ca numele unui tablou este o expresie pointer. Aceasta are citeva implicatii utile. Din moment ce numele unui tablou este sinonim cu locatia elementului sau zero, asignarea

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

poate fi scrisa si

```
pa = a
```

Inca si mai surprinzator la prima vedere este faptul ca o referinta la a[i] poate fi scrisa si ca *(a+i). Evaluind pe a[i], C il converteste in *(a+i); cele doua forme sint echivalente. Aplicind operatorul & ambilor termeni ai acestei echivalente, rezulta ca &a[i] este identic cu a+i: a+i adresa elementului al i-lea in tabloul a. Reciproc: daca pa este un pointer el poate fi utilizat in expresii cu un indice pa[i] este identic cu *(pa+i). Pe scurt orice tablou si exprimare de indice pot fi scrise ca un pointer si offset si orice adresa chiar in aceeasi instructiune

Trebuie tinut seama de o difernta ce exista intre numele tablou si un pointer. Un pointer este o variabila, astfel ca pa=a si pa++ sint operatii. Dar, un nume de tablou este o constanta, nu o variabila: constructii ca a=pa sau a++ sau p=&a sint interzise. Atunci cind se transmite un nume de tablou unei functii, ceea ce se transmite este locatia de inceput a tabloului. In cadrul functiei apelate acest fapt argument este o variabila ca oricare alta astfel incit un argument nume de tablou este un veritabil pointer, adica o variabila continind o adresa. Ne vom putea folosi de aceasta pentru a scrie o noua versiune a lui strlen, care calculeaza lungimea unui sir.

```
strlen(s)  /* returneaza lungimea sirului s */
```

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

Incrementarea lui s este perfect legala deoarece el este o variabila pointer; s++ nu are efect pe sirul de caractere in funca care a apelat-o pe strlen, dar incrementeaza doar copia adresei. Ca parametri formali in definirea unei functii

```
        char s[]
```

si

```
        char *s;
```

sint echivalenti; alegerea celui care trebuie scris este determinata in mare parte de expresiile ce vor fi scrise in cadrul functiei. Atunci cind un nume de tablou este transmis unei functii, aceasta poate, dupa necesitati s-o interpreteze ca tablou sau ca pointer si sa-l manipuleze in consecinta. Functia poate efectua chiar ambele tipuri de operatii daca i se pare potrivit si corect.

Este posibila si transmiterea catre o functie doar a unei parti dintr-un tablou prin transmiterea unui pointer pe inceputul subtabloului. De exemplu, daca a este un tablou;

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

si

```
        f(a + 2)
```

ambele transmit functiei f adresa elementului a[2] deoarece &a[2] si a+2 sint expresii pointer care refera al treilea element al lui a. In cadrul lui f, declarea argumentului poate citi

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

sau

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

Astfel, dupa cum a fost conceputa functia f faptul ca argumentul refera de fapt o parte a unui tablou mai mare nu are consecinte.

5.4. Aritmetica adreselor

Daca p este un pointer, atunci p++ incrementeaza pe p in asa fel incit t acesta sa pointeze pe elementul urmator indiferent de tipul obiectelor pointate, iar p+=i incrementeaza pe p pentru a pointa peste i elemente din locul unde p pointeaza curent.

C este consistent si constant cu aritmetica pointerilor; pointerii, tablourile si aritmetica adresarii constitue punctul forte al limbajului. Sa ilustram citeva dintre proprietatile lui scriind un program pentru alocare de memorie rudimentar (dar este util in ciuda simplitatii sale exista doua rutine: alloc(n) returneaza un pointer p pe n pozitii caracter consecutive care poate fi utilizat de catre apelantul lui alloc pentru alocarea de caractere; free(p) elibereaza memoria facind-o astfel refolosibila mai tirziu. Rutinele sint "rudimentare" deoarece apelurile la free trebuie facute in ordine inversa apelurilor la alloc. Aceasta inseamna ca memoria gestionata de alloc si free este o stiva sau o lista prelucrabila in regim LIFP. Biblioteca standard C este prevazuta in functii analoage care nu au atit de multe restrictii iar in capitolul 8 vom da, pentru demonstratie si alte versiuni. Intre timp se vor ivi multe aplicatii care au realmente nevoie de canalul alloc pentru a dispensa mici portiuni de memorie, de lungimi neprevazute la momente neprevazute.

Cea mai simpla implementare este de a scrie alloc pentru declararea de parti ale unui tablou mare pe care il vom numi allocbuf. Acest tablou este propriu lui alloc si free. Lucrind cu pointeri, nu cu indici in tablou nu este necesar ca vreo alta rutina sa cunoasca numele tabloului, care poate fi declarat static, adica local fisierului sursa care sustine pe alloc si free numele tabloului fiind invizibil in afara acestui fisier. In implementarile practice tabloul poate chiar sa nu aiba nici un nume el putind fi obtinut prin cererea catre sistemul de operare a unui pointer pe un bloc de memorie fara nume.

O alta informatie necesara este legata de cit anume din allocbuf a fost folosit. Vom utiliza un pointer pe urmatorul element liber, numit allocp. Cind este apelat alloc pentru n caractere, el verifica daca exista suficient loc eliberat in allocbuf. Daca astfel alloc returneaza valoarea curenta a lui allocp (adica inceputul blocului liber) atunci aceasta valoare esteincrementata cu n in asa fel incit allocp sa pointeze pe inceputul urmatoarei zone libere. Free(p) pune pur si simplu pe allocp pe p daca p este in interiorul lui allocbuf.

```
#define NULL  0   /* val pointerului in caz de eroare */
#define ALLOCSIZE 1000 /* lung spatiului disponibil */
static char allocbuf[ALLOCSIZE]; /* memorie pentru alloc*/
static char *allocp = allocbuf; /*memorarea parti libere*/
char *alloc(n) /* pointer de return pe n caractere */
int n;
 {
```

```
        if (allocp + n <= allocbuf + ALLOCSIZE)  {
            allocp += n;
            return(allocp - n);  /* vechiul p */
        } else                   /* nu-i destul loc */
         return(NULL)
}
free(p)  /* zona de memorie libera pointata de p */
char *p;
{
        if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
             allocp = p;
}
```

Citeva explicatii. In general un pointer poate fi initializat ca orice alta variabila, desi in mod normal singurele valori semnificative sint NULL sau o expresie care opereaza adrese ale unor date in prealabil definite, de tip specificat. Declaratia

```
static char *allocp = allocbuf;
```

defineste pe allocp ca fiind un pointer pe caractere si il initia-lizeaza pentru a-l pointa pe allocbuf care este urmatoarea pozitie libera atunci cind incepe programul. Aceasta stare ar putea fi scrisa si astfel

```
static char *allocp = &allocbuf[0];
```

deoararece numele tabloului este adresa elementului zero.

Testul

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

verifica daca este suficient loc pentru a satisface cererea pt n caractere. Daca ezista loc, noua valoare a lui allocp va fi cel mult mai dincolo de sfirsitul lui allocbuf. Daca cererea poate fi satisfacuta, alloc retur neaza un pointer normal (obser-vati declaratia functiei). Daca nu, alloc trebuie sa returneze un semnal care sa semnifice ca nu exista spatiu liber. Limbajul C garanteaza ca nici un pointer care pointeaza o data valida nu va contine zero, asa ca valoarea zero returnata poate fi utilizata ca semnal de eveniment anormal, nu exista spatiu liber. Se scrie NULL in loc de zero pt a indica mai clar ca aceasta este o valoare speciala pt un pointer. In general intregii nu pot fi asignati pointerilor; zero este u caz special.

Teste ca

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

si

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

releva citeva fatete importante ale aritmeticii pointerilor. Mai intii ca in unele situatii pointerii pot fi separati. Daca p si q pointeaza pe elemente ale aceluiasi tablou, relatii ca <, >, =, etc lucreaza exact.

```
p < q
```

este adevarata, de ex, in cazul in care p pointeaza pe un element anterior elementului pe care pointeaza q. Relatiile c= si != sint si ele permise. Orice pointer poate fi testat cu NULL. Dar nu exista nici o sansa in a compara pointeri in tablouri diferite. In cazul fericit se va obtine un evident nonsens, indiferent de masina pe care se lucreaza. Mai poate sa apara situatia nefericita in care codul va merge pe vreo masina esuind "misterios" pe altele

In al doilea rind, tocmai s-a observat ca un pointer si un interg pot fi adunati sau scazuti. Instructiunea

```
p + n
```

desemneaza al n-lea obiect dupa cel pointat curent de p. Acest lucru este adevarat indiferent de tipul obiectelor pe care p a fost declarat ca pointer. Compilatorul atunci cind il intilneste pe n, il delaleaza in functie de lungimea obiectelor pe care pointeaza p, lungime determinata prin declaratia lui p. De exemplu, pe PDP11 factorii de scalare sint 11 pentru char, 2 pentru int si short, 4 pentru long si float si 8 pentru double.

Este valida si scaderea pointerilor: daca p si q pointeaza pe elementele aceluiasi tablou, p-q este numarul de elemente dintre p si q. Acest fapt poate fi utilizat pentru a scrie o noua versiune a lui strlen.

```
strlen(s)  /* returneaza lungimea sirului */
char *s;
{
        char *p = s;
        while (*p != '\0')
              p++;
        return(p-s);
}
```

Prin declarare, p este initializat pe s, adica sa pointeze pe primul caracter din s. In cadrul buclei while este examinat pe care caracter pina se intilneste /0 care semnifica sfirsitul iar daca while testeaza numai daca expresia este zero este posibila omiterea testului expilcit iar astfel de bucle sint scrise adesea

```
while (*p)
      p++;
```

Deoarece p pointeaza pe caractere, p++ face ca p sa avanseze de fiecare data pe caracterul urmator, iar p-v da numarul de caractere parcurse, adica lungimea sirului. Aritmetica pointerilor este consistenta: daca am fi lucrat cu float care ocupa mai multa

memorie decit char, si daca p ar fi un pointer pe float, p++ ar avansa pe urmatorul float. Astfel, vom putea scrie o alta versiune a lui alloc care pastreaza sa zicem, float in loc de char, pur si simplu prin schimbarea lui char in float. in cadrul lui alloc si free. Toate manipularile de pointeri iau automat in considerare lungimea obiectului pointat in asa fel incit trebuie sa nu fie alterat.

Alte operatii in afara celor mentionate deja (adunarea sau scaderea unui pointer cu un intreg, scaderea sau comapararea a doi pointeri). Toate celelalte operatii arrrtmetice cu pointeri sint ilegale. Nu este permisa adunarea, impartirea, deplasarea logica, sau adunarea unui float sau double la pointer.

5.5 Pointeri pe caractere si functii

Un sir constant scris astfel

```
"I am a string "
```

este un tablou de caractere. In reprezentare interna, compilatorul termina un tablou cu caracterul \0 in asa fel incit programele sa poata detecta sfirsitul. Lungimea in memorie este astfel mai mare cu 1 decit numarul de caractere cuprinse intre ghilimele.

Poate cea mai comuna aparitie a unui sir de constante este ca argument al functiei cum ar fi in

```
char *message;
```

atunci instructiunea

```
message = "now is the time";
```

asigneazaa lui mesage un pointer in functie de caracterele reale. Aceasta nu este o copie a sirului; nu sint implicati decit pointerii. C nu este inzestrat cu alti operatori care sa trateze eze un sir de caractere ca o unitate.

Vom ilustra mai multe aspecte in legatura cu pointerii si cu tablourile stiind ca doua functii cu adevarat utile, din biblioteca standard de I/O, subiect care va fi discutat in capitolul 7.

Prima functie este strcpy(s, t) care copiaza sirul t in sirul s. Argumentele sint scrise in aceasta ordine prin analogie cu aranjarea, unde cineva ar putea spune

```
s = t
```

pentru a asigna pe t lui s. Versiunea ca tablouri este mai intii.

```
strcpy(s, t)  /*copiaza t in s */
char s[], t[];
{
        int i;
        i = 0;
```

```
                while ((s[i] = t[i] != '\0')
                        i++;
        }
```

Iata o versiune a lui strcpy cu pointeri

```
        strcpy(s, t)    /* copiaza t in s, versiunea pointeri 1*/
        char *s, *t;
        {
                while ((*s = *t) != '\0')  {
                        s++;
                        t++;
                }
        }
```

Deoarece argumentele sint transmise prin valoare, strcpy poate utiliza s si t in orice fel se doreste. Aici ei sint conventional utilizati ca pointeri, care parcurg tablourile pina in momentul in care s-a copiat \0 sfirsitul lui t, in s.

In practica, strrcpy nu va fi scris asa cum s-a aratat mai sus. O a doua posilitate ar fi

```
        strcpy(s, t)  /* copiaza t in s, versiunea 2*/
        char *s, *t;
        {
                while ((*s++ = *t++) != '\0')
                 ;
        }
```

In aceasta ultima versiune se imita incrementarea lui s si t in partea de test. Valoarea lui *t++ este caracterul pe care a pointat inainte ca t sa fi fost incrementat; prefixul ++ nu-l schimba pe t inainte ca acest caracter sa fi fost adus. In acelasi fel, caracterul este stocat in vede a pozitie s inainte ca s sa fie incrementat. Acest caracter este deasemenea valoarea care se grupeaza cu \0 pentru simboul buclei. Efectul net este ca, caracterele sint copiate din t in s, inclusiv sfirsitul lui \0.

Ca o ultima abreviere vom observa ca si gruparea cu \0 este redundanta, astfel ca functia este adesea scrisa ca

```
        strcpy(s, t)   /* copiaza t in s; versiunea pointeri 3 */
        char *s, *t;
        {
                while (*s++ = *t++)
                  ;
        }
```

Desi aceasta versiune poate parea compilcata la prima vedere, aranjamentul ca notatie este considerat suveran daca nu exista alte ratiuni de a schimba astfel ca il veti intilni frecvent in programele C.

A doua rutina este strcmp(s, t) care compara sirurile de

caractere s si t si returneaza negativ, zero sau pozitiv in functie de relatia dintre s si t; care poate fi: s<t, s=t sau s>t. Valoarea returnata este obtinuta prin scaderea caracterului de pe prima pozitie unde s difera de t.

```
strcmp(s, t)
  /* returneaza <0 daca s<t, 0 daca s==t, >0 daca s>t */
char s[], t[];
{
        int i;
        i = 0;
        while (s[i] == t[i])
             if (s[i++] == '\0')
                return(0);
        return(s[i] - t[i]);
}
```

Versiunea pointeri a lui strcmp.

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

Daca ++ si -- sint folositi altfel decit operatori prefix sau postfix pot apare alte combinatii de * si ++ si --, desi mai putin frecvente. De exemplu:

```
*++p
```

incrementeaza pe p inainte de a aduce caracterul pe care pointeaza p.

```
*--p
```

decrementeaza pe p in acelasi conditii.

Exercitiul 5.2. Scrieti o versiune pointeri pentru o functie strcat expusa in capitolul 2: strcat(s, t) copiaza sirul t la sfirsitul sirului s.

Exercitiul 5.3. Scrieti un macro pentru strcpy.

Exercitiul 5.4. Rescrieti variantele programelor din capitolele anterioare si exercitii cu pointeri in loc de tablouri indexate. Bune posibilitati ofera: getline, atoi, itoa si variantele lor, reverse, index, getop.

5.6. Pointerii nu sint de tip int

S-a putut observa ca programele C mai vechi au o atitudine mai toleranta fata de copierea pointerilor. In general a fost adevarat ca pe majoritatea masinilor un pointer poate fi asignat unui intreg si invers, fara a-l schimba; nu are loc nici o scalare sau conversie si nu se pierd biti. In mod regretabil aceasta stare de lucruri a condus la asumarea unor libertati nepermise desi partea programatorului in lucru cu rutina ce returneaza pointeri ce sint transmisi apoi pur si simplu altor rutine - necesitatea declararii pointerului fiind adesea omisa. De exemplu, sa luam o functie strsave care copiaza sirul s undeva, intr-o zona obtinuta printr-un apel la alloc, returnind apoi un pointer pe ea. Strsave se poate scrie astfel

```
char *strsave(s)  /* salveaza undeva sirul s */
char *s;
{
        char *p, *alloc();
        if ((p = alloc(strlen(s) + 1)) != NULL)
                strcpy(p, s);
        return(p);
}
```

In practica, exista o tendinta puternica de a omite declararile:

```
strsave(s)  /* salveaza undeva sirul s */
{
        char *p;
        if ((p = alloc(strlen(s) + 1)) != NULL)
                strcpy(p, s);
        return(p);
}
```

Acest cod s-ar putea sa mearga pe multe masini deoarece tipul implicit al functiilor si al argumentelor este int iar atit int-ul cit si pointerul pot fi asignati la inceput cit si la sfirsit. Cu toate acestea, acest gen de cod este inerent riscant deoarece el depinde de detalii de implementare si de arhitectura masinii, care nu pot fi rezolvate pentru compilatorul particular utilizat de dvs. Este recomandabil sa se efectueze toate declararile necesare. (Programul lint va avertiza in legatura cu astfel de restrictii in cazul in care se vor strecura inadvertente).

5.7. Tablouri multi-dimensionale

C este prevazut cu probabilitatea de a lucra cu tablouri multidimensionale, cu toate ca in practica exista tendinta ca ele sa fie mult mai putin utilizate decit tabloutile de pointeri. In aceasta sectiune vom da citeva dintre proprietatile lor.

Sa reluam problema de conversia datei, zi-in-luna in zi-in-an si viceversa. De exemplu, 1 martie este a 60-a zi dintr-un an nebisect si a 61-a dintr-un an bisect. Sa definim doua functii care sa faca conversia "day_of_year" converteste luna si ziua in ziua din an si luna, iar "month_day" converteste ziua din an in

luna si ziua. Daca aceasta din urma functie returneaza doua valori, argumentele "luna si ziua" vor fi pointeri:

```
month day (1977, 60, &m, &d)
```

puna pe m pe 3 si pe d pe 1.

Aceste functii au nevoie de aceasi informatie, o tabela cu numarul zilelor din fiecare luna ("30 zile are septembrie"). Din moment ce numarul de zil/luna difera in functie de an bisect sau an nebisect, este mai usor sa separam aceste informatii pe doua linii ale unui tablou bidimensional; apoi sa incercam sa tinem cont de ce se intimpla cu februarie in timpul calculului. Tabloul si functia pentru rezolvarea transformarilor sint dupa cum urmeaza:

```
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}
};
day-of-year(year, month, day)  /* pune nr zilei in an */
int year, month, day;   /* din luna &an */
{
     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);
}
month_day(year, yearday, pmonth, pday)  /*pune luna, zi */
int year, yearday, *pmounth, *pday;  /* din ziua in an */
{
     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;
}
```

Tabloul day-tab trebuie sa fie extern ambelor functii "day_of_year" si "month_day", in asa fel incit ambele sa-l poata utiliza.

"day-tab" este primul tablou bidimensional cu care avem de a face in C, Prin definitie un tablou bidimensional este de fapt un tablou unidimensional alei carei elemente sint fiecare in parte cite un tablou. Prin uramre, indicii se scriu astfel

```
day_tab[i][j]
```

in loc de

```
day_tab[i, j]
```

ca in majoritatea limbajelor in plus un tablou bidimensional paote fi tratat in mai multe moduri decit in alte limbaje. Elementele sint memorate pe linii, ceea ce inseamna ca indicele din deapta varaiaza primul in asa fel incit elementele sint accesate in ordinea memoriei.

Un tablou se initializeaza printr-o lista de initializatori scrisi intre acolade; fiecare liniea unui tablou bidimensional este initializata printr-o sublista corspondenta. Am inceput tabloul day-tab cu o coloana de zero, in asa fel incit numerotarea liniilor poate fi facuta de la 1 la 12 in loc de 1 0-11. Daca exista spatiu suficient, este mai usor sa se procedeze innmodul mai sus aratat in loc sa se ajusteze indicii.

In cazul in care un tablou bidimensional trebuie transmis unei functii, declararea argumentelor in funtie trebuie sa includa dimensiunea coloanei, dimensiunea liniei este irelevanta deoarece unei functii i se transmite ca si in cazurile anterioare, uh pointer. In cazul de fata este vorba de un pointer care parcurge obiecte care sint tablouri de cite 13 int. Astfel daca trebuie transmis tabelul day-tab unei functii f, declararea lui f va fi

```
f(day_tab)
int day_tab[2][13];
{
    ...
}
```

Declararea argumentului in f va fi deasemenea

```
int day-tab[][13];
```

din moment ce nr liniilor este irelevant, sau ar putea fi

```
int (*day-tab)[13];
```

care spune ca argumentul este un pointer pe un tablou de 13 intregi. Sint necesare parantezele pentru ca crosetele au prioritate mai mare decit *, fara paranteze; declararea

```
int *day-tab[13];
```

este un tablou de 13 pointeri pe intregi, dupa cum se va vedea in sectiunea urmatoare.

5.8. Tablouri de pointeri, pointeri pe pointeri

Datorita faptului ca pointerii sint ei insisi variabile, este de asteptat ca ei sa fie utilizati in tablouri de pointeri. Deci, se pune problema de a ilustra prin scrierea unui program care sorteaza un set de linii de text in ordine alfabetica, o versiune a sortului utilitar UNIX.

In capitolul 3 am prezentat o functie sort shell care sorta un tablou de intregi. Vom utiliza acelasi algoritm cu exceptia faptului ca acum vom avea de-a face cu linii de text de lungimi diferite si care, spre deosebire de intregi, nu pot fi

comparate sau deplasate printr-o singura operatie. Avem nevoie de o reprezentare a datelor care sa poata face eficient si potrivit regulilor in gestionarea linilor de text de lungime diferita.

Acum este momentul potrivit pt a introduce tabloul de pointeri. Daca liniile de sortat sint memorate cap la cap intr-un lung sir de caractere (rezervat prin alloc, sazicem) atunci fiecare linie poate fi accesata printr-un pointer pe primul sau caracter. Pointerii insisi pot fi memorati intr-un tablou. Doua linii pot fi comparate prin transmiterea pointerilor respectivi lui strcmp. Cind doua linii neordonate trebuiesc inversate se inverseaza pointerii lor in tabelul de pointeri, nu liniile insele. Acest mod de lucru elimina cuplul de probleme legate de gestionarea memoriei si, ceea ce este mai presus de orice, poate deplasa liniile reale.

Procesul de sortare consta din trei parti:

```
citirea tuturor liniilor la intrare
sortarea liniilor
tiparirea liniilor in ordine
```

Ca de obicei cel mai bine este sa impartim programul in functii care rezolva aceasta defalcare, cu o rutina principala care controleaza totul. Sa aminam pentru un moment pasul de sortare si sa ne concentram asupra structurii de date si a I/O. Rutina de inceput trebuie sa colecteze si sa salveze caracterele din fiecare linie si sa construiasca un tablou de pointeri pe linii. Va trebui, deasemenea sa se numere liniile la intrare, deoarece aceasta informstie este necesara pt sortare si tiparire. Deoarece functia de intrare poate opera doar cu un numar finit de linii-input, ea va returna o valoare, cum ar fi -1, in cazul in care se vor prezenta mai multe linii. RUtina de output trebuie doar sa tipareasca liniile in ordinea in care apar i tabloul de pointeri.

```
#define NULL 0
#define LINES 100  /* maximum de linii de sortat */
main()   /* sortarea liniilor de intrare */
{
        char *lineptr[LINES];/*pointeri pe linii de text*/
        int nlines;   /* nr liniilor libere */
if ((nlines = readlines(lineptr, LINES)) >= 0)  {
    sort(lineptr, nlines);
    writelines(lineptr, nlines);
}
else
    printf("input prea mare pt sort \n");
}
#define MAXLEN 1000
readlines(lineptr, maxlines)  /* citeste linii input */
char *lineptr[];
int maxlines;
{
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';  /* zap newline */
                   strcpy(p, line);
                   lineptr[nlines++] = p;
              }
        return(nlines);
        }
```

"newline" de la sfirsitul fiecarei linii este sters astfel incit nu va fi afectata ordinea in care sint sortate liniile.

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

Principala noutate este declarata pentru "lineptr":

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

spune ca lineptr este un tablou de elemente LINES, fiecare element fiind un pointer pechar. Adica, lineptr[i] este un pointer pe caractere iar *lineptr[i] acceseaza un caracter.

Daca lineptr este el insusi un tablou care este transmis lui writelines, el poate fi tratat ca un pointer in exact aceeasi maniera ca in exemplul nostru anterior, iar functia poate fi scrisa.

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

*lineptr pointeaza initial pe prima linie, cu fiecare incrementare el avanseaza pe linia urmatoare pina cind nlines se epuizeaza.

Intrarea si iesirea fiind controlate, se poate duce la sortare. Sortul -shell din cap 3 va suferi modificari: declaratii-le trebuie modificate iar operatia de grupare trebuie montata intr-o functie separata. Algoritmul de baza ramine acelasi ceea ce ne da o oarecare speranta ca totul va merge bine inca

```
        short(v, n)   /* sorteaza sirurile v[0]. . . v[n-1] */
```

```
        char *v[];   /* in ordine crescatoare */
        int n;
        {
                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;
                        }
        }
```

Daca orice element individual din v(alias lineptr) este un pointer pe caractere, temp va fi si el astfel de pointer, asa incit cei doi pot fi copiati unul in altul.

Am scris un program care, in fc de cunostintele din acel moment a fost rapid pe cit posibil. Acest program poate fi facut mai rapid, de exemplu sa copieze liniile input direct intr-un tablou mentinut prin readlines in loc sa le copieze in line pt ca apoi sa le plaseze undeva prin alloc. Dar, pentru a facilita intelegerea programului sa intocmim mai intii o schema logica, si abia dupa aceea sa ne preocupam de eficienta sa. Modalitatea de a face acest program mai eficeint nu vizeaza neaparat evitarea unei copii a liniilor input. Inlocuirea sortului shell prin ceva mai bun cum ar fi sortul Quicksort, este probabil mai mult decit a marca o simpla diferenta.

In capitolul 1 am semnalat acest lucru deoarece buclele while si for testeaza conditia finala inaintea executarii chiar si pt prima data a corpului buclei; ele ajuta la a ne asigura ca programele vor merge chiar si la limita, in particular fara input. Este edificator a umbla prin functiile programelor de sortare pt a verifica ce se intimpla daca nu exista deloc text de intrare.

Exercitiul 5.5. Rescrieti readlines pt a crea linii intr-un tablou umplut cu main, in loc de a apela pe alloc pt rezervarea de memorie, Cu cit este mai rapid acest program ?

5.9. Initializarea tablourilor de pointeri

Sa ne pune problema scrierii unei functii month_name(n) care returneaza un pointer pe un sir de caractere continind numele a n linii. Aceasta este o aplicatie ideala pentru un tablou static intern. month-name contine un tablou propriu de siruri de caractere si returneaza un pointer pe sirul convenabil atunci cind este apelat. Scopul acestei sectiuni este de a arata cum se initializeaza tabloul de nume.

Sintaxa este similara cu cea a initializarilor precedente:

```
char *month-name(n) /*returneaza numele celei de-a n-a luni*/
```

```
int n;
{
     static char *name[] =
     {
          "luna eronata",
          "ianuarie",
          "februarie",
          "martie",
          "aprilie",
          "mai",
          "iunie",
          "iulie",
          "August",
          "septembrie",
          "octombrie",
          "noiembrie",
          "decembrie"
     };
     return((n < 1 || n > 12) ? name[0] : name[n]);
}
```

Declararea numelui, care este un tablou de pointeri pe caractere este aceeasi ca si la lineptr, in ex de sortare. Valorile de initializare sint de fapt o lista de caractere; fiecare dintre acestea din urma este asignat pozitiei corespunzatoare din tablou. Mai precis, caracterele celui de-al i-lea sint plasate undeva iar pointerul pe ele este stocat in name[i]. Daca lungimea tabloului name nu este specificata, compilatorul numara valorile de intializare si pune lungimea corecta.

5.10. Comaparatie pointeri. Tablouri multi-dimensionale

Nou venitii in C sint uneori confuzi in legatura cu deosebirea dintre un tablou bidimensional si un tablou de pointeri cum ar fi name din exemplul de mai sus. Fiind date declaratiile

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

utilizarile lui a si b pot fi similare, in sensul ca a[5][5] si b[5][5] sint ambele referinte legale ale aceluiasi int. Dar a este un tablou in toata regula: toate cele 100 celule de memorie trebuie alocate iar pentru gasirea fiecarui element se face calculul obisnuit al indicelui. Pentru b, oricum prin declarea se aloca 10 pointeri; fiecare trebuie facut sa pointeze un tablou de intregi. Presupunind ca fiecare pointeaza cite 10 elemente din tablou, atunci vom obtine 100 celule de memorie rezervate, plus cele 10 celule pt pointeri. Astfel tabloul de pointeri utilizeaza sensibil mai mult spatiu si poate cere un pro explicit de initializare. Dar, exista doua avantaje: accesarea unui element se face indirect prin intermediul unui pointer, in loc sa se faca prin inmultire si adunare iar liniile tabloului pot fi de lungimi diferite. Aceasta insemna ca nu orice element al lui b este constrins sa pointeze pe un vector de 10 elemente, unii pot pointa

pe cite 2 elemente, altii pe cite 20 si altii pe niciunul.

Desi am mai discutat acest lucru la intregi, de departe, cea mai frecventa utilizare a tabloului de pointeri este cea ilustrata prin mounth-name: sa stocheze lanturi de caractere de lungimi diferite.

Exercitiul 5.6. Rescrieti rutinele day_of_year si mounth-day cu pointeri in loc de indexare.

5.11. Argumentele liniei de comanda

Printre facilitatile oferite de C exista modalitatea de a transmite argumentele liniei de comanda sau parametrii unui program atunci cind el incepe sa se execute. Pt inceperea executiei este apelat main prin doua argumente. Primul (numit convetional arge) este numarul argumentelor liniei de comanda prin care a fost apelat programul; al doilea (argv) este un pointer pe un tablou de lanturi de caractere care contine argumentele, unul pentru fiecare lant. Manipularea acestor lanturi de caractere este o utilizare comuna a nivelelor multiple de pointeri.

Cea mai simpla ilustrare a declaratiilor necesare si a celor de mai sus amintite este programul echo, care pune pur si simplu pe o singura linie argumentele liniei de comanda, separate prin blancuri. Astfel, daca este data comanda

```
echo hello, world
```

iesirea este

```
hello, world
```

Prin conventie, argv[0] este numele prin care se recunoaste programul, asa ca argc este 1. In exemplul de mai sus, argc este 3 si argv[0], argv[1] si argv[2] sint respectiv echo, hello si world. Aceasta este ilustrata in echo:

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

Arg fiind un pointer pe un tablou de pointeri, exista citeva modalitati de a scrie acest program care implica manipularea pointerului mai curind decit indexarea tabloului. Sa vedem doua variante:

```
        main(argc, argv)  /* arg echo; a doua versiune */
        int argc;
        char *argv[];
        {
```

```
            while (--argc > 0)
                printf("%s%c", *++argv, (argc >1) ? ' ' : '\');
        }
```

Daca argv este un pointer pe inceputul tabloului care contine siruri de argumente, a-l incrementa cu i(++argv) face ca el sa pointeze pe argv[1] in loc de argv[0]. Fiecare incrementare succesiva muta pe argv pe urmatorul argument; argv este deci pointerul pe acel argument. Simultan argc este decrementat; atunci cind el devine zero, nu mai exista argumente de imprimat.

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

Aceasta versiune arata ca formatul argumentului lui printf poate fi o expresie ca oricare alta. Aceasta utilizare nu este foarte frecventa dar este bine sa fie retinuta.

Ca un al doilea exemplu, sa facem unele modificari in configuratia programului de cautare din cap4. In cazul unui apel repetat, configuratia ce serveste de model va fi prelucrata ca atare, de fiecare data de catre program, ceea ce ar duce la un aranjament evident nesatisfacator. Urmind exemplul utilitarului grep-UNIX, sa schimbam programul in asa fel incit configuratia model sa fie specificata prin primul argument al liniei de comanda.

```
        #define MAXLINE 1000
        main(argc, argv)
              /* cautarea model specificat prin primul argument*/
        int argc;
        char *argv[];
        {
                char line[MAXLINE];
                if (argc != 2)
                    printf("usage: find pattern\n);
                else
                    while (getline(line, MAXLINE)>0)
                          if (index(line, argv[1] >= 0)
                              printf("%s", line);
        }
```

Acum poate fi elaborat modelul de baza in asa fel incit sa ilustreze viitoarele constructii realizate cu ajutorul pointerilor. Sa presupunem ca dorim ca doua argumente sa fie optionale. Unul dintre ele spune "tipareste toate liniile cu exceptia celor care contin modelul "; al doilea cere "fiecare linie tiparita sa fie precedata de numarul curent".

O conventie uzuala pt programele C este legata de argumentul

care incepe cu un semn minus si care introduce un flag sau un paramentru optional. Daca se alege -x(pt "exceptie") pt semnalarea inversarii, si ("nr") pt a cere numararea liniilor, atunci comanda

```
find -x -n the
```

cu intrarea

```
now is the time
for all good men
to come to the aid
of their party
```

va produce iesirea

```
2: for all good men
```

Argumentele optionale sint admise in orice ordine iar restul programului va fi insensibil la numarul argumentelor care au fost, de fapt, prezente. In particular, apelul la index nu va referi pe argv[2] atunci cind a fost un singur flag si nici la argv[1] daca n-a existat nici un argument flag. In plus, este convenabil pt utilizatori daca argumentele optionale pot fi adunate, ca in

```
find -nx the
```

Iata programul

```
#define MAXLINE 1000
main(argc, argv)  /*gasirea configuratiei din primul arg*/
int argc;
char *arv[];
{
        char lim[MAXLINE}, *s;
        long lineno = 0;
        int except = 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("fiind: ilegal option%c\n", *s);
                  argc = 0;
                  break;
                }
        if (argc != 1)
                printf("usage: find -x -n pattern\n");
        else
```

```
                        while (getline(lim, MAXLINE) > 0)
                        {
                              lineno++;
                              if ((index(line,*argv)>=0) !=except)
                              {
                                 if (number)
                                     printf("%ld: ", lineno);
                                  printf("%s", line);
                              }
                        }
                }
```

argv este incrementat inaintea fiecarui argument optional si argc este decrementat. Daca nu exista erori, la sfirsitul buclei argc va fi 1 iar argv va pointa pe configuratia data. De notat ca *++argv este un pointer pe un lant de caractere; (*++argv)[0] este primul caracter. Parantezele sint necesare deoarece fara de expresia ar fi *++(argv[0]), ceea ce este cu totul altceva (si eronat). O forma corecta ar fi.

```
      **++argv
```

Exercitiul 5.7. Scrieti programul add care evalueaza o expresie poloneza inversata din linia de comanda. De exemplu,

```
      add 2 3 4 + *
```

calculeaza 2 x (3 + 4)

Exercitiul 5.8. Modificati programele entab si detab (scrise ca exercitii in cap 1) in asa fel incit sa accepte ca argumente o lista de tab-stop-uri utilizati tab-urile normale daca nu exista argumente.

Exercitiul 5.9. Extindeti entab si dentab in asa fel incit sa accepte prescurtarea.

```
      entab m +n
```

care inseamna tab-stop dupa fiecare n coloane, incepind de la coloana m. Scrieti functia oarecare implicita convenabila pentru utilizator.

Exercitiul 5.10. Scrieti programul tail care tipareste rutinele n linii-input. Presupunem, implicit n=10, dar el poate fi schimbat un argument optional, astfel

```
      tail -n
```

imprima ultimele n linii. In mod normal, programul va functiona indiferent de intrare (rationala sau nu), sau de valoare lui n. Scrieti progarmul in asa fel incit sa utilizeze in mod optim memoria: liniile vor fi pastrate ca in short, nu intr-un tablou bidimensional de lungime fixata.

5.12. Pointeri pe functii

In C functia in sine nu este o variabila, dar exista posibilitatea de a defini un pointer pe o functie, care poate fi manipulat, transmis functiilor, plasat in tablouri etc. Vom ilustra aceasta modificind procedura de sortare scrisa mai anterior in acest cap, in asa fel incit fiind dat argumentul optional-n sa se sorteze liniile input memorie, nu linii copiate.

De obicei, un sort consta in trei parti-o comparatie care realizeaza ordonarea oricarei perechi de obiecte, o schimbare, prin care se inverseaza oridinea obiectelor si un algoritm de sortare care face comparatii si schimbari pina cind obiectele sint definitiv ordonate. Algoritmul de sortare este independent de operatiile de comparare si schimbare, astfel incit prin transmiterea functiei de comparare si schimbare catre el, se va putea realiza sortarea pe diferite criterii. Acest lucru ni-l propunem in noul sort.

Compararea lexicografica a doua linii este realizata prin strcmp iar schimbarea prin swap; avem nevoie de o rutina numecmp care compara doua linii pe baza valorilor numerice si returneaza un indice de conditie de acelasi fel ca si strcmp. Aceste trei functii sint declarate in main iar pointerii pe ele sint transmisi la sort. Sort la rindul sau apeleaza functiile prin sort, la rindul sau apeleaza functiile prin pointeri. Am sarit peste procesul de tratare a arguentelor eronate, concentrindu-ne astfel pe rezolvarea problemelor principale.

```
#define LINES 1000  /* maximum de linii de sortat */
main(argc, argv)  /* sortarea linii input */
int argc;
char *argv[];
{
   char *lineptr[LINES];  /* pointeri pe liniile de text */
   int nlines;  /* nr linii-input citire */
   int strcmp(), numecmp();  /* fc de comparare */
   int swap(); /* fc de schimbare */
   int numeric = 0;  /* 1 daca este 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, numecmp, swap);
       else
               sort(lineptr, nlines, strcmp, swap);
       writelines(lineptr, nlines);
   } else
      printf("intrare prea mare pentru sort\n");
}
```

strcmp, numecmp si swap sint adrese de functii; din moment ce ele sint cunoscute ca fiind functii, operatorul & nu este necesar la

fel cum el nu este necesar inaintea numelui unui tablou. Compilatorul este cel care rezolva transmiterea adresei functie.

Al doilea pas este modificarea lui sort:

```
sort(v, n, comp, exch)
       /* sorteaza sirurile v[0]. . . v[n-1] */
char *s[];
int n;
int (*comp)(), (*exch)();
{
        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]);
                }
}
```

Declaratiile vor fi studiate cu grija.

```
int (*comp)()
```

spune ca comp este un pointer pe o fc ce returneaza un int. Primul set de paranteze este necesar; fara ele

```
int *comp()
```

ar spune ca cmp este o functie ce returneaza un pointer pe un integer, ceea ce este cu totul altceva. Utilizarea lui comp in linia

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

este comparabila cu declaratia potrivit careia cmp este u pointer pe o functie; *comp este functie, iar

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

este apelul ei. Parantezele sint necesare pentru asocierea corecta a componentelor.

Am ilustrat deja prin strcmp compararea a doua siruri. Iata numecmp care compara doua siruri numerice pe baza valorii numerice:

```
numcmp(s1, s2)  /* compara numeric s1 cu s2 */
char *s1, *s2;
{
        double atof(), v1, v2;
        v1 = atof(s1);
        v2 = atof(s2);
        if (v1 < v2)
```

```
                return(-1);
            else if (v1 > v2)
                return(1);
            else
                return(0);
        }
```

Pasul final este adaugarea functiei swap care schimba doi pointeri. Aceasta este adoptata direct din ceea ce am prezentat mai devreme in acest capitol.

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

Exista o varietate de alte optiuni care pot fi adaugate la programul de sortare; unele dintre ele pot fi reincercate ca exerctii.

Exercitiul 5.11. Modificati sort in asa fel incit sa gestioneze un flag, -r care indica sortarea in ordine inversa (descrescatoare). Bineinteles -r trebuie sa fie compatibil cu -n.

Exercitiul 5.12. Adaugati optiunea -n pt a prelucra impreuna literele mari si literele mici adica sa nu se mai faca distinctia intre aceste doua tipuri de caractere grafice intimpul sortarii; datele cu litere mari si cee cu litere mici sint sortate impreuna, in asa fel incit a si A apar adiacent, nu separate prin intregul alfabet al literelor mici sau mari.

Exercitiul 5.13. Adaugati optiunea -d ("ordinea din dictionar") care realizeaza comparari doar pentru litere, numere si flancuri. Asiguarti-va ca -d merge impreuna cu -f.

Exercitiul 5.14. Adaugati o faclitatea legata de gestionarea cimpurilor, in asa fel incit sortarea sa poata fi facuta pe cimpuri si interiorul liniilor, fiecare corespunzind unui set independent de optiuni. (Indexul acestei carti fost sortat cu -df pt ordinea alfabetica si cu -n pentru paginilor.)

CAPITOLUL 6.STRUCTURI

O structura este o colectie de una sau mai multe variabile care pot fi de tipuri diferite, grupate impreuna sub un singur nume pentru o manipulare convenabila. (Structurile sint anumite "inregistrari" in unele limbaje, de exemplu in PASCAL.)

Exemplul traditional de structura este inregistrarea

personala: un salariat este descris prin citeva atribute ca numele, adresa, salariu etc. Fiecare din atribute la rindul lor pot fi structuri: numele are mai multe componente, adresa de asemenea, salariul de baza s.a.m.d.

Structurile ajuta la organizarea datelor complicate, mai ales in programele de mari dimensiuni, deoarece in multe situatii ele permit ca un grup de variabile inrudite sa fie tratate unitar si nu ca entitati separate. In acest capitol vom incerca sa ilustram cum sint utilizate structurile. Programele pe care le vom folosi in acest scop sint mai mari decit multe altele din manualul acesta, dar inca de dimensiuni modeste.

6.1. Bazele

Sa ne reamintim rutinele de conversie a datei din capitolul 5. O data consista din mai multe parti, precum ziua, luna, anul si probabil ziua din an si numele lunii. Aceste cinci variabile pot fi toate plasate intr-o singura structura ca aceasta:

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

Cuvintul cheie "struct" introduce o structura de date, care este o lista de declartii cuprinsa intre acolade. Un nume optional (eticheta) numit "structure tag", poate sa urmeze cuvintul cheie "struct"(precum date in exemplul de mai sus). Aceasta eticheta da un nume acestui gen de structura si poate fi referita in continuare ca prescurtare de declaratie detaliata.

Elementele sau variabilele mentionate intr-o structura sint numite "membri". Un membru al structurii sau o eticheta a unei structuri sau o variabila simpla pot avea acelasi nume fara ambiguitate deoarece se disting prin context. Desigur se va utiliza acelasi nume doar pentru a defini obiecte in strinsa relatie.

Acolada din dreapta care inchide lista membrilor structurii poate fi urmata de o lista de variabile ca in exemplul de mai jos:

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

ceea ce este sintactic analog cu:

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

in sensul ca fiecare declaratie numeste pe x, y, z si z ca variabile de tipul specificat si aloca spatiu pentru fiecare din ele.

O declaratie de structura care nu este urmata de o lista de

variabile nu aloca spatiu de memorie ci descrie doar forma sau organizarea structurii. Daca structura este nominalizata, numele poate fi utilizat in program pentru atribuirea de valori structurii. De exemplu:

```
struct date d;
```

defineste o variabila d care are o structura de tip data, si poate fi initializata la un moment dat conform definitiei sale cu o lista ca mai jos:

```
struct date d ={4, 7, 1776, 186, "Jul" };
```

Un membru a unei structuri particulare poate fi referit intr-o expresie printr-o constructie de forma:

```
"numestructura.membru"
```

Operatorul "." din constructia de mai jos leaga numele membrului de numele structurii. De exemplu pentru a afla un an bisect din structura d se refera la membrul "year" astfel:

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

sau pentru a testa numele liniei din membrul "mon"

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

sau pentru a converti numele lunii la litere mici

```
d.mon_name[0] = lower(d.mon_name[0]);
```

O structura poate sa cuprinda structuri, de exemplu:

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

Structura "person" contine doua structuri de tip data ("birthdate" si "hiredate"). Daca declaram "p" astfel

```
struct person emp;
```

atunci o constructie

```
emp.birthdate.month
```

se va referi la luna din data nasterii. Operatorul "." asociaza partea stinga cu dreapta.

6.2. Structuri si functii

Exista un numar de restrictii relative la structurile in C. Regulile esentiale sint ca nu puteti face asupra unei structuri decit operatia de obtinere a adresei cu &, si de accesare a unuia dintre membrii structurii. Acest lucru presupune ca structurile nu pot fi asignate sau copiate ca un tot unitar, si ca nu pot fi nici pasate sau returnate intre functii. (Aceste restrictii vor fi ridicate in versiunile viitoare.) Pointerii la structuri nu se supun insa acestor restricti, si de aceea structurile si functiile coopereaza confortabil. In fine, structurile automate, ca si tablourile automate, nu pot fi initializate; nu pot aceasta decit stucturile statice.

Sa vedem citeva din aceste probleme rescriind functiile de conversie de data din capitolul precedent, folosind structuri. Deoarece regulile interzic pasarea de structuri direct unei functii, trebuie fie sa pasam componentele separat, fie sa pasam un pointer catre intregul obiect. Prima alternativa se foloseste de day_of_year asa cum am scris in capitolul 5:

```
        d.yearday = day_of_year(d.year, d.month, d.day);
```

Celalalt mod este de a pasa un pointer. Daca am declarat pe hiredate ca:

```
        struct date hiredate;
```

si am rescris pe day_of_year, putem apoi spune:

```
        hiredate.yearday = day-of-year(&hiredate);
```

deci pointerul lui "hiredate" trece la "day-of-year". Acum functia trebuie sa fie modificata deoarece argumentul sau este un pointer si nu o lista de variabile.

```
        day-of-year(pd) /* set day of year from month, day */
        struct date *pd;
        {
         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);
        }
```

Declaratia "struct date *pd;" spune ca pd este un pointer la o structura de tip data. Notatia "pd->year" este noua. Daca p este pointer la o structura, atunci

```
p->member-of-structure
```

refera un membru particular. (Operatorul -> este un minus urmat de >.) Cu ajutorul pointerului pd poate referit si un membru al structurii, de exemplu membrul "year":

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

Deoarece pointerii la structuri sint des utilizati este preferabila folosirea operatorului ->, forma mai scurta. In constructia de mai sus parantezele sint necesare deoarece operatorul "." este prioritar fata de "*". Atit "->" cit si "." se asociaza de la stinga la dreapta astfel:

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

sint

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

Pentru completare dam o alta functie "month-day" rescrisa folosind structurile.

```
month-day(pd) /* set month and day from day of year */
struct date *pd;
{
 int i, leap;
 leap = pd->year % 4 == && pd->year % 100 != 0 || pd->year %
400 == 0;
   pd->day = pd->yearday;
   for (i = 1; pd->day > day_tab[leap][i]; i++)
   pd->day -= day_tab[leap][i];
   pd->month = i;
}
```

Operatorii de structura "->" si ". " impreuna cu "()" pentru liste si "[]" pentru indici in ierarhia prioritatilor sint cei mai puternici. De exemplu la declaratia:

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

atunci ++p->x incrementeaza pe x si nu p, deoarece ordinea implicita este ++(p->x). Parantezele pot fi utilizate pentru a modifica prioritatile: (++p)->x incrementeaza p inainte de a accesa x, iar (p++)->x incrementeaza p dupa aceea. (Acest ultim set de paranteze nu este necesar. De ce ?)

In acelasi sens, *p->y aduce ceea ce pointeaza y; *p->y++ incrementeaza y dupa ce se face accesul la ceea ce

pointeaza y (la fel cu *s++); (*p->y)++ incrementeaza ceea ce pointeaza y; si *p++->y incrementeaza p dupa accesul la ceea ce pointeaza y.

6.3. Tablouri de structuri

Structurile sint in special utile pentru manevrarea tablourilor de variabile inrudite. Pentru exemplificare sa consideram un program pentru a numara fiecare aparitie a cuvintului cheie C. Avem nevoie de un tablou de siruri de caractere pentru a pastra numele si un tablou de intregi pentru contoare. O posibilitate este folosirea in paralel a doua tablouri unul pentru cuvinte cheie si unul pentru contoare, astfel

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

Dar insasi faptul ca se folosesc doua tablouri paralele indica posibilitatea unei alte organizari. Fiecare acces de cuvint cheie este de fapt o pereche:

```
char *keywordd
int keycount
```

ceea ce de fapt este un tablou de perechi. Urmatoarea declaratie de structura:

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

defineste un tablou "keytab" de structuri de acest tip si ii aloca memorie.
Fiecare element al tabloului este o structura. Aceasta se mai poate scrie:

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

De fapt structura "keytab" contine un set constant de nume care cel mai usor se poate initializa o singura data cind este definita. Initializarea structurii este analoaga cu initializarile prezentate mai devreme -definitia este urmata de o lista de initializatori cuprinsi intre acolade:

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

```
 "break", 0,
 "case", 0,
 "char", 0,
 "continue", 0,
 "default", 0,
 /*...*/
 "unsigned", 0,
 "while", 0
};
```

Initializatorii sint listati in perechi corespunzind membrilor structurii. Ar fi mai precis ca initializatorii pentru fiecare "sir" sau structura sa fie cuprinsi intre acolade:

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

dar acoladele in interior nu sint necesare atunci cind initializatorii sint simple variabile sau siruri de caractere si cind toate sint prezente. Ca de obicei compilatorul va calcula numarul de intrari in tabloul "keytab" daca initializatorii sint prezenti iar "[]" este lasat gol.

Programul de numarare a cuvintelor cheie incepe prin definirea lui "keytab". Rutina principala citeste intrarea prin apeluri repetate a functiei "getword" care la o apelare citeste un cuvint. Fiecare cuvint este cautat in "keytab" cu ajutorul unei versiuni a functiei de cautare linia ra descrisa in capitolul 3. (Bineinteles lista de cuvinte cheie trebuie sa fie in ordine crescatoare pentru ca treaba sa mearga).

```
#define MAXWORD 20
main() /* count c keywords */
{
 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);
}
binary(word, tab, n) /* find word in tab[0]...tab[n-1] */
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);
       }
```

In curind vom prezenta si functia "getword"; pentru moment este suficient sa spunem ca ea returneaza LETTER de atitea ori cind gaseste un cuvint si copiaza cuvintul in primul ei argument.

Cantitatea NKEYS este numarul de cuvinte cheie din "keytab". Desi am putea sa numaram acestea manual, este mult mai usor si mai sigur sa facem aceasta cu ajutorul masinii in special daca lista este subiectul unei posibile schimbari. O posibilitatea ar fi sa incheiem lista de initializatori cu un pointer nul si apoi sa buclam in "keytab" pina este detectat sfirsitul.

Dar aceasta este mai mult decit este necesar, deoarece dimensiunea tabloului este complect determinata in momentul compilarii. Numarul de intrari este:

```
dimensiunea lui keytab/dimensiunea lui struct key
```

C obtine un timp de compilare operator unar numit "sizeof" care poate fi folosit la calcularea dimensiunii oricarui obiect. Expresia

```
sizeof(object)
```

livreaza un intreg egal cu dimensiunea obiectului specificat Dimensiunea este data in unitati nespecificate numite "bytes", care au aceeasi dimensiune ca si "char") Obiectul poate o variabila actuala sau tablou sau structura, ori numele unui tip de baza ca "int" sau "double" ori numele unui tip derivat ca o structura. In cazul nostru numarul de cuvinte cheie se obtine prin impartirea dimensiunii tabloului la dimensiunea unui element detablou. Acest calcul se face uzind o declaratie "#define" setind valoarea in NKEYS:

```
#define NKEYS (sizeof(keytab) / sizeof(struct key))
```

Si acum asupra functiei "getword". De fapt noi am scris un "getword" mai complicat decit era necesar pentru acest program, dar nu mult mai complicat. "getword" returneaza urmatorul "word" de la intrare, unde "word" este fie un sir de litere si cifre incepind cu o litera fie un un singur caracter. Tipul obiectului este returnat ca o valoare a unei functii; aceasta este LETTER daca e vorba de un cuvint, EOF daca este sfirsitul

fisierului si este caracterul insusi daca este un caracter non-alfabetic.

```
getword(w, lim) /* get next word from input */
char *w;
int lim;
{
 int c, t;
 if (type(c = *w++ =getch()) != LETTER)  {
    *w = '\0';
    return(c);
 }
 while (--lim > 0)  {
    t = type(c = *w++ = getch());
    if (t != LETTER && t != DIGIT)  {
        ungetch(c);
        break;
    }
 }
 *(w-1) = '\0';
 return(LETTER);
}
```

"getword" utilizeaza rutinele "getch" si "ungetch" despre care am scris in capitolul 4: cind colectia de caractere alfabetice se termina, "getword" a depasit deja cu un caracter sirul. Apelarea lui "ungetch" repune un caracter inapoi la intrare pentru urmatorul acces.

"getword" apeleaza "type" pentru a determina tipul fiecarui caracter de la intrare. Iata o versiune numai pentru alfabetul ASCII.

```
type(c) /* return type of ascii character */
int c;
{
 if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
     return(LETTER);
 else if (c >= '0' && c <= '9')
     return(DIGIT);
 else
     return(c);
```

Constantele simbolice LETTER si DIGIT pot avea orice valori care nu sint in conflict cu caracterele nonalfabetice si EOF; ALegerile evidente sint:

```
#define LETTER 'a'
#define DIGIT0'
```

"getword" poate fi mai rapid daca apelurile la functia "type" sint in locuite cu apeluri corespunzatoare tablourilor, type[]. Biblioteca standard C contine macrouri numite "isalpha" si "isdigit" care opereaza de aceasta maniera.

Exercitiul 6.1. Faceti aceste modificari la "getword" si determinati modificarile de viteza ale programului.

Exercitiul 6.2. Scrieti o versiune de "type" care este independenta de setul de caractere.

Exercitiul 6.3. Scrieti o versiune a programului de numarare a cuvinte care nu numara aparitiile continute intre apostrofi.

6.4. Pointeri la structuri

Pentru a ilustra citeva din consideratiile referitoare la pointeri si tablouri de structuri sa rescriem programul de contorizare a cuvintelor cheie, de data aceasta folosind pointerii in loc de indici.

Declaratia externa "keytab" nu necesita modificari, dar "main" si "binary" necesita.

```
main() /* count keywords; pointer version */
{
 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);
}
 struct key *binary(word, tab, n) /* find word */
 char *word;                      /* in tab[0]...tab[n-1] */
 struct key tab[];
 int n;
 {
        int cond;
        struct key *low = detab[0];
        struct key *high = ftab[n-1];
        struct key *mid;
        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);
}
```

Aici sint mai multe chestiuni de notat. Prima, declaratia "binary" trebuie sa indice ca e returneaza un pointer structurii tip "key" in locul unui intreg. Acesta este declarat in "main" cit si in "binary". Daca "binary" gaseste cuvintul, returneaza un pointer; daca acesta lipseste, returneaza NULL.

A doua, orice acces la elementele lui "keytab" sint facute prin pointeri. Aceasta determina o schimbare semnificativa in "binary calculul poate fi simplu.

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

deoarece adunarea a doi pointeri nu va produce nici un fel de raspuns utilizabil si de fapt este ilegala. Aceasta trebuie schimbata in

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

care seteaza "mid" in punctul de la jumatatea intre "low" si "high".

Ar trebui sa studiati si initializatorii pentru "low" si "high" Este posibil sa se initializeze un pointer la adresa unui obiect definit dinainte; aceasta am facut noi aici.

In "main" am scris:

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

Daca p este un pointer la structura, orice operatie aritmentica asupra lui p tine cont de dimnesiunea actuala a structurii, astfel p++ incrementeaza p cu cantitatea corecta pentru a obtine urmatorul element al tabloului de structuri. Dar sa nu credeti ca dimensiunea structurii este suma dimensiunilor membrilor sai deoarece aliniamentul cerut pentru diferiti membri pot determina aparitia de "gauri" in structura.

Si in final o consideratie asupra formatului programului. Cind o functie f returneaza un tip complicat ca in:

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

numele functiei este dificil de vazut si de gasit cu un editor de texte De aceea un alt stil este citeodata folosit.

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

Aceasta este mai mult o chestiune de gust personal; luati forma pe care o doriti si tineti-va de ea.

6.5 Structuri cu autoreferire

Sa presupunem ca vrem sa tratam problema numararii aparitiei cuvintelor in modul cel mmai general adica sa numaram fiecare aparitie a fiecarui cuvint la o intrare. Deoarece lista de cuvinte nu este cunoscuta in avans nu putem sa o sortam convenabil si sa folosim o cautare liniara. Inca nu putem sa efectuam o cautare liniara pentru fiecare cuvint in momentul cind soseste pentru a vedea daca a mai aparut deoarece perogramul ar dura foarte mult. (Mai precis, timpul de rulare ar creste cu patratul numarului de cuvinte introduse.) Cum putem organiza datele pentru a face fata in mod eficient cu o lista de cuvinte arbitrare?

O solutie ar fi sa pastram setul de cuvinte sortat in orice moment plasind fiecare cuvint in pozitia adecvata in momentul sosirii lui. Aceasta n-ar terebui facut prin mutarea cuvintelor intr-un tablou linear deoarece si aceasta ia un timp prea lung. In loc vom utiliza o structura de date numita "binary tree" adica arborele binar.

Arborele contine un "node" pentru fiecare cuvint distinct; fiecare nod contine:

- un pointer la textul cuvintului
- un contor al numarului de aparitii
- un pointer la nodul-copil din stinga
- un pointer la nodul-copil din dreapta

Nici un nod nu poate avea mai mult de doi copii. Ar trebui sa aiba numai zero sau unu.

Nodurile sint astfel mentinute incit subarborele din stinga contine numai cuvinte care sint mai mici decit cuvintele din nodul considerat, iar subarborele din dreapta numai cuvinte care sint mai mari. Pentru a afla daca un nou cuvint se afla in arbore se incepe comparatia de la nodul radacina. Daca ele

sint egale chestiunea este rezolvata Daca noul cuvint este mai mic decit cuvintul din nodul radacina cercetarea continua la nodul copil din stinga; altfel este investigat nodul copil din dreapta. Daca nu gasete nici un copil cu un continut egal noului cuvint, inseamna ca noul cuvint nu se afla in vreun nod al arborelui si deci trebuie creat unul. Procesul de cautare este inerent recursiv deoa rece cautarea din orice nod este legata de cautarea din unul din nodurile copii. In consecinta rrutinele pentru insertie si tiparire vor fi cele mai naturale.

Intorcindu-ne la descrierea unui nod, este clar o structura cu patru componente:

```
struct tnode { /* the basic node */
 char *word; /* points to the text */
 int count;  /* number of occurrences */
 struct tnode *left; /* left child */
 struct tnode *right; /* right child */
};
```

In general este ilegal ca o structura sa contina o parte a ei insusi, dar

```
struct tnode *left;
```

declara "left" ca fiind un pointer al nodului si nu nodul insusi.

Codul pentru intregul program este surprinzator de scurt si foloseste o multime de rutine ajutatoare pe care deja le-am scris. Acestea sint 'getword' pentru a aduce fiecare cuvint de la intrare si 'alloc' pentru a obinte spatiu pentru cuvinte in avans.

Rutina principala citeste cuvinte cu ajutorullui 'getword' si le instaleaza in arbore cu ajutorul lui 'tree'.

```
#define MAXWORD 20
main() /* word frequency count */
{
        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);
}
```

Un cuvint este prezentat de catre rutina "main" la nivelul cel mai de sus(radacina) arborelui. La fiecare nivel cuvintul este comparat cu cuvintul aflat deja in nod si apoi este coborit in arbore fi e in subarborele din dreapta fie in cel din stinga printr-o apelare recursiva la "tree". Cuvintul nou

fie are deja un corespondent in arbore in care caz contorul este incrementat, fie se ajunge la un pointer nul ceea ce indica necesitatea creearii unui nou nod in arbore. Daca este creat un nou nod "tree" returneaza un pointer care este instalatin nodul parinte.

```
        struct tnode *tree(p, w) /* install w at or below p */
        struct tnode *p;
        char *w;
        {
                struct tnode *talloc();
                char *strsave();
                int cond
                if (p == NULL)  { /* a new word has arrived */
                   p = talloc(); /* make a new node */
                   p->word = strsave(w);
                   p->count = 1;
                   p->left = p->right = NULL;
                } else if ((cond = strcmp(w, p->word)) == 0)
                   p->count++;   /* repeated word */
                else if (cond < 0) /* lower goes into left subtree
*/
                   p->left = tree(p->left, w);
                else             /* greater into right subtree */
                   p->right = tree(p->right, w);
                return(p);
        }
```

Memoria pentru noul nod este obtinuta de rutina "talloc" care este o adaptare a rutinei "alloc" scrisa mai devreme. Aceasta returneaza un pointer a unui spatiu liber utilizabil pentru un nod. ANoul cuvint este copiat in locul pregatit de catre "strsave", contorul este initializat si cei doi copii sint facuti nuli. Aceasta parte a programului este executata numai la "marginea" copacului cind un nou nod este de adaugat. Am omis intentionat controlul erorilor la valorile returnate de rutinele "strsave" si "talloc".

Rutina "treprint" tipareste arborele incepind in ordine cu subarborele din stinga; in fiecare nod tipareste subarborele sting (toate cuvintele mai mici decit cel actual), apoi cuvintul actual, apoi subarborele drept (toate cuvintele mai mari). Daca va simtiti cam nesiguri pe recursiune imaginativa singuri un arbore si tipariti-l cu "tree print"; este una dintre cele mai pure rutine recursive pe care o puteti gasi.

```
        treeprint(p) /* prnt tree p recursinely */
        struct tnode *p;
        {
           if (p != NULL)  {
                treeprint(p->left);
                printf("%4d %s\n", p->count, p->word);
                treeprint(p->right);
            }
```

```
        }
```

O observatie practica; daca arborele devine dezechilibrat din cauza cuvintelor care nu sosesc intr-o ordine aleatoare, timpul de rulare al progarmului poate sa creeasca prea repede. In cel mai rau caz, daca cuvtele sosesc deja sortate acest program devinnnnn o alternativa neeconomicoasa a cautarii lineare. Exista generalizari ale arborelui care nu sufera de acest neajuns, dar pe care nu le putem descrie aici.

Inainte de a parasi acest exemplu merita o scurta digresiune asupra problemelor legate de alocarea de memorie. Clar, este dezirabil ca intr-un program sa fie folosit un singur alocator de memorie chiar daca aloca diferite feluri de obiecte. Dar daca un alocator trebuie sa satisfaca cereri pentru pointeri de caractere "char" sau de "struct tnode", daca se pun intrebari. Prima, cum se satisface cerinta celor mai multe masini reale ca obiectele de diferite tipuri sa fie aliniate la adrese satisfacatoare(de exemplu intregii adesea trebuie aliniati pare)? A doua, ce declaratii pot satisface faptul ca "alloc" in mod necesar returneaza diferite tipuri de pointeri ?

Cererile de aliniament pot fi usor satisfacute, cu pretul unor spatii neutilizate, asigurindu-ne ca alocatorul intodeauna returneaza pointeri care intrunesc toate restrictiile de aliniament. De exemplu pentru PDP-11 este suficient ca "alloc" intodeauna sa returneze un "even" point, din moment ce orice tip de obiect poate fi memorat la o adresa "even". Masuri asemanatoare se iau pe alte masini. Astfel implementarea unui "alloc" poate sa nu fie portabila) dar usajul este. "Alloc"-ul din capitolul 5 nu garanteaza nici un aliniament particular. In capitolul 8 vom prezenta modul de a face corect.

In legatura cu intrebarea asupra declaratiei tipului pentru "alloc"; i"c" cea mai buna metoda este aceea de a dlara ca "alloc" returneaza un pointer la "char" si apoi sa se converteasca ppoinerul in tipul dorit. Daca p este declarat astfel:

```
        char *p;
```

atunci

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

il converteste intr-unn "tnode" pointer dintr-o exepresie.

Astfel "talloc" devine:

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

Aceasta este mai mult decit este nevoie pentru compilarile curente dar reprezinta cel mai sigur mod pentru viitor.

Exercitiul 6.4. Scrieti un program care citeste un program "c" si tipareste in ordine alfabetica fiecare grup al numelor de variaile care sint identice in primele 7 caractere dar diferite dupa aceea. (Asigurativa ca 7 este un parametru).

Exercitiul 6.5. Scrieti un porogram care tipareste o lista a tuturor cuvintelor dintr-un text si pentru fiecare cuvint o lista cu numerele liniei in care apar.

Exercitiul 6.6. Scrieti un program care tipareste cuvintele distincte, care sint introdues, in ordinea frecventei lor de aparitie. Fiecare cuvint sa fie precedat de numarul de aparitii.

6.6 Cautare in tabele

In aceasta sectiune vom da continutul unui pachet de cautare in tabel ca o ilustrare a mai multor aspecte legate de structuri. Amcest pachet este tipic pentru modul de explorare a unui tabel de simboluri pentru rutinele de management ale unui macroprocesor. De exemplu sa consideram declaratia din C, "#define". Cind o linie ca

```
#define YES 1
```

este luata in considerare, numele YES si textul corespunzator 1 sint memorate intr-un tabel. Mai tirziu, cind numele YES apare intr- declaratie ca

```
inword = YES;
```

trebuie inlocuit cu 1.

Exista doua rutine majore care manipuleaza numele si textele de inlocuire corespunzatoare. "install(s, t)" inregistreaza numele "s" si textul de inlocuire "t" intr-o tabela; "s" si"t" sint siruri de caractere. "lookup(s)" cauta numele "s" intr-o tabela si returneaza un pointer la locul unde afost gasit ori NULL daca nu a fost gasit.

Algoritmul care se utilizeaza se bazeaza pe o cautare fragmentara "hash" un nume care se introduce este convertit intr-un intreg pozitiv care apoi este utilizat ca index intr-un tabloude pointeri. Un element al tablului pointeaza pe inceputul unui lant de balncuri care descriiu avind respectiva valoare de fragment ("hash"). Acesta este NULL daca nici un nume nu

corespunde acestei valori.

Un bloc din lant este o structura care contine: pointeri, la nume, textul de inlocuire si urmatorul bloc din lant. Daca pointerul pentru urmatorul bloc este nul marcheaza sfirsitul lantului de blocuri.

```
        struct nlist { /* basic table entry */
                char *name;
                char *def;
                struct nlist *next; /* next entry in chain */
        };
```

Tabloul de poiunteri este:

```
        #define HASHSIZE 100
        static struct nlist *hashtab[HASHSIZE]; /* pointer table */
```

Functia de fragmentare ("hashing"), care este utilizata de ambele rutine "lookup" si "install", aduna valorile carcterelor din sirul nume si formeaza restul modulo dimensiunea tabloului. (Aceasta nu este cel mai bun algoritm posibil, darare meritul simplitatii.)

```
        hash(s) /* form hash value for string s */
        char *s;
        {
                int hashval;
                for (hashval = 0; *s != '0';)
                    hashval += *s++;
                return(hashval % HASHSIZE);
        }
```

Procesul de fragmentare ("hashing") produce un index de cautare in "hashtab"; sirul de carctere cautat se va gasi intr-un bloc din sirul de blocuri a carui inceput este pointat de elementul din tabelul "hashtab". Cautarea este efectuata de rutina "lookup". Dca "lookup" gaseste in "hashtab" intrarea deja prezenta, returneaza un pointer; daca nu, returneaza NULL.

```
        struct nlist *lookup(s) /* look for s in hashtab */
        char *s
        {
                struct nlist *up
                for (up = hashtab[hash(s)]; up != NULL; up = up-
>next)
                    if (strcmp(s, up->name) == 0)
                        return(up); /*found it */
                return(NULL); /* not found */
```

Rutina "install" foloseste "lookup" pentru a determina care dintre numele de instalat sint deja prezente; daca nu, o noua definire trebuie sa succeada uneia vechi. Altfel spus, o intrare complect noua este creeata. "install" returneza NULL daca dintr-un

motiv oarecare nu mai este loc pentru o noua intrare.

```
        struct nlist *install(name, def) /*put(name, def) */
        char *name, *def; /* in hashtab */
        {
                struct nlist *np, *lookup();
                char *strsave(), *alloc();
                int hashval;
                if ((np = lookup(name)) == NULL)  { /* not found */
                   np = (struct nlist *) alloc(sizeof(*np));
                   if (np == NULL)
                      return(NULL);
                if ((np->name = strsave(name)) == NULL)
                      return(NULL);
                hashval = hash(np->name);
                np->next = hashtab[hashval];
                hashtab[hashval] = np;
                } else /* already there */
                      free(np->def); /*free previous definition */
                if ((np->def = strsave(def)) == NULL
                    return(NULL);
                return(np);
        }
```

"strsave" copiaza sirul furnizat de catre argumentul sasu intr_un loc obtinut cu un apel la functia "alloc". Am prezentat codul in capitolul 5 Deoarece apelurile la "alloc" si "free" pot sa apara in orice ordine si deoarece aici aliniamentul conteaza, versiunea simpla a "alloc" de la capitolul 5, nu este adecvata aici vedeti capitolele 7 si 8.

Exercitiul 6.7. Scrieti o rutina care sa scoata un nume si definitia dintr-o tabela mentinuta cu "lookup" si "install".

Exercitiul 6.8. Implementati o versiune simpla a procesorului "#define" utilizabila in programe C, folosind rutinele din aceasta sectiune. Puteti gasi de ajutor si "getch" si "ungetch".

6.7 Cimpuri

Atunci cind spatiul de memorare estre la mare pret, poate fi necesar ca mai multe obiecte sa fie impachetate intr-un singur cuvint masina; un caz adesea folosit este compactarea fanioanelor de un singur bit necesare in aplicatii catabelele de simboluri ale compilatoarelor. Formatele externe impuse de interfetele cu diversele echipamente externe, adesea impun compactrea datelor pe un cuvint.

Sa ne imaginam un fragment dintr-un compilator care manipuleaza o tabela de simboluri. Fiecare identificator dintr-un program are anumite informatii asociate, de exemplu, este

sau nu un cuvint cheie, e este sau nu extern si/sau static, s. a. m. d. Cel mai compact mod de a coda astfel de informatii este un set de fanioane de un bit cuprinse intr-un "char" sau "int".

Modul cel mai uzual este de a defini un set de masti corespunzatoare pozitiilor cu biti semnificativi, ca in

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC 04
```

(Numele trebuie sa fie puteri ale lui doi.) Apoi accesul la biti devine un soi de "bitareala" cu operatori de permutare, mascare si complementare, care au foat descrisi in capitolul 2.

Anumite idioame apar frecvent:

```
flags != EXTERNAL ! STATIC;
```

valideaza bitii EXTERNAL si STATIC ca fanioane, in timp ce

```
flags &= (EXTERNAL ! STATIC);
```

ii invalideaza.

Desi aceste idioame sint cu usurinta manevrate, limbajul C, mai degraba ofera capacitatea preferabila de adefini si accesa cimpuri Un cimp este un set de biti adiacenti cuprinsi intr-un singur "int". Sintaxa de definire si accesul cimpurilor este bazata pe structuri. De exemplu tabela de simboluri definita mai sus ar putea fi inlocuita printr-o definire a trei cimpuri.

```
struct {
        unsigned is_keyword: 1;
        unsigned is_extern: 1;
        unsigned is_static: 1;
}flags;
```

Este astfel definita o variabila numita "flags" care contine trei cimpuri de cite un bit. Numarul de dupa ":" reprezinta lungimea cimpului in biti. Cimpurile sint declarate fara semn (unsigned) tocmai pentru a accentua ca sint cantitati fara semn.

Cimpurile individuale sint referite ca "flags.is_keyword", "flags.is_extern", etc, la fel ca oricare alti me, bri ai structurii. Cimpurile se comporta ca niste mici intregi fara semn si pot participa in expresii aritmetice la fel ca oricare alti intregi. Exemplele de mai sus pot fi rescrise mai natural astfel

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

pozitioneaza bitii pe unu;

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

pozitioneaza bitii pe zero;

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

pentru a-i testa.

Un cimp poate sa nu corespunda cu limitelke unui "int"; in acest caz cimpul este alinial la urmatoarea margine 'int". Cimpurile nu trebuie sa aibe un nume; cimpurile fara nume(doua puncte si lungimea numai) sint folosite ca umplutura. Lungimea speciala 0 poate fi folosita pentru alinierea la urmatoarea "int" margine.

Exista un numar de reguli de aplicat cimpurilor. Dintre cele mai importante, cipurile sint asignate de la stinga la dreapta in unele masini si de la dreapta la stinga in altele ceea ce reflecta natura diferita a hardwareului inseamna ca, desi cipurile sint utilizabile de preferinta pentru a intretine structurile de adte interne, chestiunea carui sfirsit vine primul trebuie considerata cu grija pentru prelucrarea datelor definite extern.

Alte restrictii care trebuie avute in minte: cimpurile sint fara semn ele pot fi memorate numai in "int"(sau echivalentul "unsigned"); ele nu sint tablouri; ele nu au adrese, astfel ca operatorul "&" nu le poate fi aplicat.

6.8 Uniuni

O uniune este o varaibila care poate pastra (la momente diferite) obiecte de diferite tipuri si dimensiuni, iar compilatorul tine seama de cerintele de dimensiune si aliniament. Uniunile permit sa se manipuleze diferite feluri de date intr-o singura zona de memorie, fara sa se includa in program nici un fel de informatii dependente de masina.

Ca de exemplu, sa consideram din nou o tabela de simboluri a unui compilator. Sa presupunem ca constantekle pot fi "int", "float" sau pointeri de caractere. Valoarea unei constante particulare trebuie memorata intr-o variabila de tipul corespunzator, dar este cel mai convenabil pentru organizarea tabelei daca valoarea ocupa aceasi dimensiune de memorie si acelasi loc in memorie in functie de tipul ei. Aceasta este scopul unei uniuni -sa permita ca o singura variabila care poate sa contina oricare din mai multe tipuri. La fel ca la cimpuri, sintaxa se bazeaza pe structuri.

```
        union u-tag{
                int ival;
                float fval;
                char *pval;
        } uval;
```

O variabila "uval" va fi suficient de mare ca sa pastreze oricare din cele trei tipuri, iar privitor la masina, codul generat de compilator este indiferent de caracteristicile hard. Oricare din aceste tipuri poate fi asignat la "uval" si apoi utilizat in expresii, atita vreme cit uzajul este considerat, adica tipul utilizat trebuie sa fie cel mai recent memorat. Este responsabilitatea programatorului sa tina seama de tipul de data curent memorat intr-o uniune; rezultatele sint dependente de masina daca ceva este memorat ca un tip si este extras ca altul.

Sintactic, membri unei uniuni sint accesati ca

```
union-name.member
```

sau

```
union-pointer->member
```

la fel cu structurile. Daca variabila "utype" este folosita pentru a tine seama
de tipul curent al marimii memorate in "uval", poate scrie un cod, astfel:

```
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("bad type %d in utype\n", utype);
```

Uniunile pot sa se gaseasca in structuri si tablouri si viceversa. Notatia pentru a accesa un membru a unei uniuni dintr-o structura(sau viceversa) este identica cu aceea pentru structurile in cuiburi. De exemplu in structura definita prin

```
struct {
        char *name;
        int flags;
        int utype;
        union {
                int ival;
                float fval;
                char *pval;
        } uval;
} symtab[NSYM];
```

variabila "ival" este referita astfel

```
symtab[i].uval.ival
```

iar primul caracter al sirului "pval" prin

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

In fond, o uniune este o structura in care toti membri au deplasamentul zero, structura este suficient de mare ca sa cuprinda cel mai mare membru, iar aliniamentul este potrivit pentru toate tipurile din uniune Ca si la structuri singurele operatii admise cu uniunile sint de a accesa un membru si obtinerea adresei lui;uniunile nu pot fi asignate, nu se pot aplica functii asupra lor, Pointeri la uniuni pot fi utilizati de o maniera identica pinterilor la structuri.

In capitolul 8 se arata cum alocatorul de memorie prin intermediul unei uniuni de memorie prin intermediul unei uniuni forteaza o variabila sa se alinieze la orice fel de particularitati de memorare.

6.9. Typedef

C admite o facilitate numita typedef pentru a crea nume pentru noi tipuri de date. De exemplu declaratia:

```
typedef int LENGTH;
```

face numele LENGTH sinonim pentru "int". Tipul LENGTH poate fi utilizat in declaratii exact la fel ca int:

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

Similar, declaratia:

```
typedef char*STRING;
```

face ca STRING sa fie sinonim cu char* sau pointerul unui caracter, care poate fi utilizat in declaratii ca:

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

Observati ca tipul fiind declarat printr-un typedef apare in pozitia unui nume de variabila. Sintactic, typedef este ca o clasa de memorie, extern, static, etc. In cazul de mai sus am utilizat litere pentru a accentua numele.

Ca de exemplu mai complicat putem folosi typedef pentru nodurile copacului prezentat mai inainte in acest capitol:

```
typedef struct tnode  { /* the basic node */
 char *word; /* points to the text */
 int count; /* number of occurrences */
 struct tnode *left; /* left child */
 struct tnode *right; /* right child */
} TREENODE, *TREEPTR;
```

Aceasta creeaza doua noi tipuri de cuvinte cheie numite TREENODE (o structura) si TREEPTR (un pointer la structura), dupa care rutina "talloc" poate deveni:

```
TREEPTR talloc()
{
 char *alloc();
 return((TREEPTR) alloc(sizeofTREENODE));
```

Trebuie sa specificam ca un typedef nu creaza noi tipuri; mai degraba renumeste tipurile existente. Variabilele astfel declarate au exact aceleasi proprietati ca si variabilele a caror declaratii sint explicite. In fond, typedef este ca un #define, exceptind ca, de cind este interpretat de copilator el face fata cu substitutiile in text care sint dincolo de capacitatile C macro procesorului. De exemplu:

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

creaza tipul PFI, care poate fi utilizat intr-un context ca

```
PFI strcmpp, numcmp, swap;
```

din programul de sort din capitolul 5.

Exista doua motive mai importante pentru a utiliza declaratiile t9pedef. Primul este de a parametrizaun program alaturi de problemele de portabilitate. Daca pse utilizeza typedef pentru diferite tipuri de date care pot fi dependente de masina, numai typedef necesita modificari daca programul este mutat pe alta masina. O situatie obisnuita este de a folosi typedef pentru cantitati intregi variate, cind se alcatuieste un set adecvat de "short", "int" si "long" pentru fiecare masina.

Un al doilea scop a lui typedef este de a da mai mare claritate unui program numit TREEPTR este mai usor de inteles decit unul declarat ca un simplu pointer la o structura complicata.

In final, exista deja posibilitatea ca in viitor, compilatorul sau * un alt program ca "lint" sa faca uz de informatia continuta in typedef ca sa execute niste controale in plus asupra programului.

CAPITOLUL 6.STRUCTURI

O structura este o colectie de una sau mai multe variabile care pot fi de tipuri diferite, grupate impreuna sub un singur nume pentru o manipulare convenabila. (Structurile sint anumite "inregistrari" in unele limbaje, de exemplu in PASCAL.)

Exemplul traditional de structura este inregistrarea personala: un salariat este descris prin citeva atribute ca

numele, adresa, salariu etc. Fiecare din atribute la rindul lor pot fi structuri: numele are mai multe componente, adresa de asemenea, salariul de baza s.a.m.d.

Structurile ajuta la organizarea datelor complicate, mai ales in programele de mari dimensiuni, deoarece in multe situatii ele permit ca un grup de variabile inrudite sa fie tratate unitar si nu ca entitati separate. In acest capitol vom incerca sa ilustram cum sint utilizate structurile. Programele pe care le vom folosi in acest scop sint mai mari decit multe altele din manualul acesta, dar inca de dimensiuni modeste.

6.1. Bazele

Sa ne reamintim rutinele de conversie a datei din capitolul 5. O data consista din mai multe parti, precum ziua, luna, anul si probabil ziua din an si numele lunii. Aceste cinci variabile pot fi toate plasate intr-o singura structura ca aceasta:

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

Cuvintul cheie "struct" introduce o structura de date, care este o lista de declartii cuprinsa intre acolade. Un nume optional (eticheta) numit "structure tag", poate sa urmeze cuvintul cheie "struct"(precum date in exemplul de mai sus). Aceasta eticheta da un nume acestui gen de structura si poate fi referita in continuare ca prescurtare de declaratie detaliata.

Elementele sau variabilele mentionate intr-o structura sint numite "membri". Un membru al structurii sau o eticheta a unei structuri sau o variabila simpla pot avea acelasi nume fara ambiguitate deoarece se disting prin context. Desigur se va utiliza acelasi nume doar pentru a defini obiecte in strinsa relatie.

Acolada din dreapta care inchide lista membrilor structurii poate fi urmata de o lista de variabile ca in exemplul de mai jos:

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

ceea ce este sintactic analog cu:

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

in sensul ca fiecare declaratie numeste pe x, y, z si z ca variabile de tipul specificat si aloca spatiu pentru fiecare din ele.

O declaratie de structura care nu este urmata de o lista de variabile nu aloca spatiu de memorie ci descrie doar forma

sau organizarea structurii. Daca structura este nominalizata, numele poate fi utilizat in program pentru atribuirea de valori structurii. De exemplu:

```
struct date d;
```

defineste o variabila d care are o structura de tip data, si poate fi initializata la un moment dat conform definitiei sale cu o lista ca mai jos:

```
struct date d ={4, 7, 1776, 186, "Jul" };
```

Un membru a unei structuri particulare poate fi referit intr-o expresie printr-o constructie de forma:

```
"numestructura.membru"
```

Operatorul "." din constructia de mai jos leaga numele membrului de numele structurii. De exemplu pentru a afla un an bisect din structura d se refera la membrul "year" astfel:

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

sau pentru a testa numele liniei din membrul "mon"

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

sau pentru a converti numele lunii la litere mici

```
d.mon_name[0] = lower(d.mon_name[0]);
```

O structura poate sa cuprinda structuri, de exemplu:

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

Structura "person" contine doua structuri de tip data ("birthdate" si "hiredate"). Daca declaram "p" astfel

```
struct person emp;
```

atunci o constructie

```
emp.birthdate.month
```

se va referi la luna din data nasterii. Operatorul "." asociaza partea stinga cu dreapta.

6.2. Structuri si functii

Exista un numar de restrictii relative la structurile in C. Regulile esentiale sint ca nu puteti face asupra unei structuri decit operatia de obtinere a adresei cu &, si de accesare a unuia dintre membrii structurii. Acest lucru presupune ca structurile nu pot fi asignate sau copiate ca un tot unitar, si ca nu pot fi nici pasate sau returnate intre functii. (Aceste restrictii vor fi ridicate in versiunile viitoare.) Pointerii la structuri nu se supun insa acestor restricti, si de aceea structurile si functiile coopereaza confortabil. In fine, structurile automate, ca si tablourile automate, nu pot fi initializate; nu pot aceasta decit stucturile statice.

Sa vedem citeva din aceste probleme rescriind functiile de conversie de data din capitolul precedent, folosind structuri. Deoarece regulile interzic pasarea de structuri direct unei functii, trebuie fie sa pasam componentele separat, fie sa pasam un pointer catre intregul obiect. Prima alternativa se foloseste de day_of_year asa cum am scris in capitolul 5:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

Celalalt mod este de a pasa un pointer. Daca am declarat pe hiredate ca:

```
struct date hiredate;
```

si am rescris pe day_of_year, putem apoi spune:

```
hiredate.yearday = day-of-year(&hiredate);
```

deci pointerul lui "hiredate" trece la "day-of-year". Acum functia trebuie sa fie modificata deoarece argumentul sau este un pointer si nu o lista de variabile.

```
day-of-year(pd) /* set day of year from month, day */
struct date *pd;
{
 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);
}
```

Declaratia "struct date *pd;" spune ca pd este un pointer la o structura de tip data. Notatia "pd->year" este noua. Daca p este pointer la o structura, atunci

```
p->member-of-structure
```

refera un membru particular. (Operatorul -> este un minus urmat de >.) Cu ajutorul pointerului pd poate referit si un membru al structurii, de exemplu membrul "year":

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

Deoarece pointerii la structuri sint des utilizati este preferabila folosirea operatorului ->, forma mai scurta. In constructia de mai sus parantezele sint necesare deoarece operatorul "." este prioritar fata de "*". Atit "->" cit si "." se asociaza de la stinga la dreapta astfel:

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

sint

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

Pentru completare dam o alta functie "month-day" rescrisa folosind structurile.

```
month-day(pd) /* set month and day from day of year */
struct date *pd;
{
 int i, leap;
 leap = pd->year % 4 == && pd->year % 100 != 0 || pd->year %
400 == 0;
  pd->day = pd->yearday;
  for (i = 1; pd->day > day_tab[leap][i]; i++)
  pd->day -= day_tab[leap][i];
  pd->month = i;
}
```

Operatorii de structura "->" si ". " impreuna cu "()" pentru liste si "[]" pentru indici in ierarhia prioritatilor sint cei mai puternici. De exemplu la declaratia:

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

atunci ++p->x incrementeaza pe x si nu p, deoarece ordinea implicita este ++(p->x). Parantezele pot fi utilizate pentru a modifica prioritatile: (++p)->x incrementeaza p inainte de a accesa x, iar (p++)->x incrementeaza p dupa aceea. (Acest ultim set de paranteze nu este necesar. De ce ?)

In acelasi sens, *p->y aduce ceea ce pointeaza y; *p->y++ incrementeaza y dupa ce se face accesul la ceea ce pointeaza y (la fel cu *s++); (*p->y)++ incrementeaza ceea ce

pointeaza y; si *p++->y incrementeaza p dupa accesul la ceea ce pointeaza y.

6.3. Tablouri de structuri

Structurile sint in special utile pentru manevrarea tablourilor de variabile inrudite. Pentru exemplificare sa consideram un program pentru a numara fiecare aparitie a cuvintului cheie C. Avem nevoie de un tablou de siruri de caractere pentru a pastra numele si un tablou de intregi pentru contoare. O posibilitate este folosirea in paralel a doua tablouri unul pentru cuvinte cheie si unul pentru contoare, astfel

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

Dar insasi faptul ca se folosesc doua tablouri paralele indica posibilitatea unei alte organizari. Fiecare acces de cuvint cheie este de fapt o pereche:

```
char *keywordd
int keycount
```

ceea ce de fapt este un tablou de perechi. Urmatoarea declaratie de structura:

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

defineste un tablou "keytab" de structuri de acest tip si ii aloca memorie.
Fiecare element al tabloului este o structura. Aceasta se mai poate scrie:

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

De fapt structura "keytab" contine un set constant de nume care cel mai usor se poate initializa o singura data cind este definita. Initializarea structurii este analoaga cu initializarile prezentate mai devreme -definitia este urmata de o lista de initializatori cuprinsi intre acolade:

```
struct key {
 char *keyword;
 int keytab;
} keytab[] ={
 "break", 0,
```

```
 "case", 0,
 "char", 0,
 "continue", 0,
 "default", 0,
 /*...*/
 "unsigned", 0,
 "while", 0
};
```

Initializatorii sint listati in perechi corespunzind membrilor structurii. Ar fi mai precis ca initializatorii pentru fiecare "sir" sau structura sa fie cuprinsi intre acolade:

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

dar acoladele in interior nu sint necesare atunci cind initializatorii sint simple variabile sau siruri de caractere si cind toate sint prezente. Ca de obicei compilatorul va calcula numarul de intrari in tabloul "keytab" daca initializatorii sint prezenti iar "[]" este lasat gol.

Programul de numarare a cuvintelor cheie incepe prin definirea lui "keytab". Rutina principala citeste intrarea prin apeluri repetate a functiei "getword" care la o apelare citeste un cuvint. Fiecare cuvint este cautat in "keytab" cu ajutorul unei versiuni a functiei de cautare linia ra descrisa in capitolul 3. (Bineinteles lista de cuvinte cheie trebuie sa fie in ordine crescatoare pentru ca treaba sa mearga).

```
#define MAXWORD 20
main() /* count c keywords */
{
 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);
}
binary(word, tab, n) /* find word in tab[0]...tab[n-1] */
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);
    }
```

In curind vom prezenta si functia "getword"; pentru moment este suficient sa spunem ca ea returneaza LETTER de atitea ori cind gaseste un cuvint si copiaza cuvintul in primul ei argument.

Cantitatea NKEYS este numarul de cuvinte cheie din "keytab". Desi am putea sa numaram acestea manual, este mult mai usor si mai sigur sa facem aceasta cu ajutorul masinii in special daca lista este subiectul unei posibile schimbari. O posibilitatea ar fi sa incheiem lista de initializatori cu un pointer nul si apoi sa buclam in "keytab" pina este detectat sfirsitul.

Dar aceasta este mai mult decit este necesar, deoarece dimensiunea tabloului este complect determinata in momentul compilarii. Numarul de intrari este:

```
dimensiunea lui keytab/dimensiunea lui struct key
```

C obtine un timp de compilare operator unar numit "sizeof" care poate fi folosit la calcularea dimensiunii oricarui obiect. Expresia

```
sizeof(object)
```

livreaza un intreg egal cu dimensiunea obiectului specificat Dimensiunea este data in unitati nespecificate numite "bytes", care au aceeasi dimensiune ca si "char") Obiectul poate o variabila actuala sau tablou sau structura, ori numele unui tip de baza ca "int" sau "double" ori numele unui tip derivat ca o structura. In cazul nostru numarul de cuvinte cheie se obtine prin impartirea dimensiunii tabloului la dimensiunea unui element detablou. Acest calcul se face uzind o declaratie "#define" setind valoarea in NKEYS:

```
#define NKEYS (sizeof(keytab) / sizeof(struct key))
```

Si acum asupra functiei "getword". De fapt noi am scris un "getword" mai complicat decit era necesar pentru acest program, dar nu mult mai complicat. "getword" returneaza urmatorul "word" de la intrare, unde "word" este fie un sir de litere si cifre incepind cu o litera fie un un singur caracter. Tipul obiectului este returnat ca o valoare a unei functii; aceasta este LETTER daca e vorba de un cuvint, EOF daca este sfirsitul fisierului si este caracterul insusi daca este un caracter non-

alfabetic.

```
getword(w, lim) /* get next word from input */
char *w;
int lim;
{
 int c, t;
 if (type(c = *w++ =getch()) != LETTER)  {
    *w = '\0';
    return(c);
 }
 while (--lim > 0)  {
    t = type(c = *w++ = getch());
    if (t != LETTER && t != DIGIT)  {
        ungetch(c);
        break;
    }
 }
 *(w-1) = '\0';
 return(LETTER);
}
```

"getword" utilizeaza rutinele "getch" si "ungetch" despre care am scris in capitolul 4: cind colectia de caractere alfabetice se termina, "getword" a depasit deja cu un caracter sirul. Apelarea lui "ungetch" repune un caracter inapoi la intrare pentru urmatorul acces.

"getword" apeleaza "type" pentru a determina tipul fiecarui caracter de la intrare. Iata o versiune numai pentru alfabetul ASCII.

```
type(c) /* return type of ascii character */
int c;
{
 if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
     return(LETTER);
 else if (c >= '0' && c <= '9')
     return(DIGIT);
 else
     return(c);
```

Constantele simbolice LETTER si DIGIT pot avea orice valori care nu sint in conflict cu caracterele nonalfabetice si EOF; ALegerile evidente sint:

```
#define LETTER 'a'
#define DIGIT0'
```

"getword" poate fi mai rapid daca apelurile la functia "type" sint in locuite cu apeluri corespunzatoare tablourilor, type[]. Biblioteca standard C contine macrouri numite "isalpha" si "isdigit" care opereaza de aceasta maniera.

Exercitiul 6.1. Faceti aceste modificari la "getword" si determinati modificarile de viteza ale programului.

Exercitiul 6.2. Scrieti o versiune de "type" care este independenta de setul de caractere.

Exercitiul 6.3. Scrieti o versiune a programului de numarare a cuvinte care nu numara aparitiile continute intre apostrofi.

6.4. Pointeri la structuri

Pentru a ilustra citeva din consideratiile referitoare la pointeri si tablouri de structuri sa rescriem programul de contorizare a cuvintelor cheie, de data aceasta folosind pointerii in loc de indici.

Declaratia externa "keytab" nu necesita modificari, dar "main" si "binary" necesita.

```
        main() /* count keywords; pointer version */
        {
         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);
        }
         struct key *binary(word, tab, n) /* find word */
         char *word;                      /* in tab[0]...tab[n-1] */
         struct key tab[];
         int n;
         {
                int cond;
                struct key *low = detab[0];
                struct key *high = ftab[n-1];
                struct key *mid;
                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);
        }
```

Aici sint mai multe chestiuni de notat. Prima, declaratia

"binary" trebuie sa indice ca e returneaza un pointer structurii tip "key" in locul unui intreg. Acesta este declarat in "main" cit si in "binary". Daca "binary" gaseste cuvintul, returneaza un pointer; daca acesta lipseste, returneaza NULL.

A doua, orice acces la elementele lui "keytab" sint facute prin pointeri. Aceasta determina o schimbare semnificativa in "binary calculul poate fi simplu.

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

deoarece adunarea a doi pointeri nu va produce nici un fel de raspuns utilizabil si de fapt este ilegala. Aceasta trebuie schimbata in

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

care seteaza "mid" in punctul de la jumatatea intre "low" si "high".

Ar trebui sa studiati si initializatorii pentru "low" si "high" Este posibil sa se initializeze un pointer la adresa unui obiect definit dinainte; aceasta am facut noi aici.

In "main" am scris:

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

Daca p este un pointer la structura, orice operatie aritmentica asupra lui p tine cont de dimnesiunea actuala a structurii, astfel p++ incrementeaza p cu cantitatea corecta pentru a obtine urmatorul element al tabloului de structuri. Dar sa nu credeti ca dimensiunea structurii este suma dimensiunilor membrilor sai deoarece aliniamentul cerut pentru diferiti membri pot determina aparitia de "gauri" in structura.

Si in final o consideratie asupra formatului programului. Cind o functie f returneaza un tip complicat ca in:

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

numele functiei este dificil de vazut si de gasit cu un editor de texte De aceea un alt stil este citeodata folosit.

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

Aceasta este mai mult o chestiune de gust personal; luati forma pe care o doriti si tineti-va de ea.

6.5 Structuri cu autoreferire

Sa presupunem ca vrem sa tratam problema numararii aparitiei cuvintelor in modul cel mmai general adica sa numaram fiecare aparitie a fiecarui cuvint la o intrare. Deoarece lista de cuvinte nu este cunoscuta in avans nu putem sa o sortam convenabil si sa folosim o cautare liniara. Inca nu putem sa efectuam o cautare liniara pentru fiecare cuvint in momentul cind soseste pentru a vedea daca a mai aparut deoarece perogramul ar dura foarte mult. (Mai precis, timpul de rulare ar creste cu patratul numarului de cuvinte introduse.) Cum putem organiza datele pentru a face fata in mod eficient cu o lista de

cuvinte arbitrare?

O solutie ar fi sa pastram setul de cuvinte sortat in orice moment plasind fiecare cuvint in pozitia adecvata in momentul sosirii lui. Aceasta n-ar terebui facut prin mutarea cuvintelor intr-un tablou linear deoarece si aceasta ia un timp prea lung. In loc vom utiliza o structura de date numita "binary tree" adica arborele binar.

Arborele contine un "node" pentru fiecare cuvint distinct; fiecare nod contine:

un pointer la textul cuvintului
un contor al numarului de aparitii
un pointer la nodul-copil din stinga
un pointer la nodul-copil din dreapta

Nici un nod nu poate avea mai mult de doi copii. Ar trebui sa aiba numai zero sau unu.

Nodurile sint astfel mentinute incit subarborele din stinga contine numai cuvinte care sint mai mici decit cuvintele din nodul considerat, iar subarborele din dreapta numai cuvinte care sint mai mari. Pentru a afla daca un nou cuvint se afla in arbore se incepe comparatia de la nodul radacina. Daca ele sint egale chestiunea este rezolvata Daca noul cuvint este mai mic decit cuvintul din nodul radacina cercetarea continua la nodul copil din stinga; altfel este investigat nodul copil din dreapta. Daca nu gasete nici un copil cu un continut egal noului cuvint, inseamna ca noul cuvint nu se afla in vreun nod al arborelui si deci trebuie creat unul. Procesul de cautare este inerent recursiv deoa rece cautarea din orice nod este legata de cautarea din unul din nodurile copii. In consecinta rrutinele pentru insertie si tiparire vor fi cele mai naturale.

Intorcindu-ne la descrierea unui nod, este clar o structura cu patru componente:

```
struct tnode { /* the basic node */
 char *word; /* points to the text */
 int count;  /* number of occurrences */
 struct tnode *left; /* left child */
 struct tnode *right; /* right child */
};
```

In general este ilegal ca o structura sa contina o parte a ei insusi, dar

```
struct tnode *left;
```

declara "left" ca fiind un pointer al nodului si nu nodul insusi.

Codul pentru intregul program este surprinzator de scurt si foloseste o multime de rutine ajutatoare pe care deja le-am

scris. Acestea sint 'getword' pentru a aduce fiecare cuvint de la intrare si 'alloc' pentru a obinte spatiu pentru cuvinte in avans.

Rutina principala citeste cuvinte cu ajutorullui 'getword' si le instaleaza in arbore cu ajutorul lui 'tree'.

```
        #define MAXWORD 20
        main() /* word frequency count */
        {
                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);
        }
```

Un cuvint este prezentat de catre rutina "main" la nivelul cel mai de sus(radacina) arborelui. La fiecare nivel cuvintul este comparat cu cuvintul aflat deja in nod si apoi este coborit in arbore fi e in subarborele din dreapta fie in cel din stinga printr-o apelare recursiva la "tree". Cuvintul nou fie are deja un corespondent in arbore in care caz contorul este incrementat, fie se ajunge la un pointer nul ceea ce indica necesitatea creearii unui nou nod in arbore. Daca este creat un nou nod "tree" returneaza un pointer care este instalatin nodul parinte.

```
        struct tnode *tree(p, w) /* install w at or below p */
        struct tnode *p;
        char *w;
        {
                struct tnode *talloc();
                char *strsave();
                int cond
                if (p == NULL)  { /* a new word has arrived */
                   p = talloc(); /* make a new node */
                   p->word = strsave(w);
                   p->count = 1;
                   p->left = p->right = NULL;
                } else if ((cond = strcmp(w, p->word)) == 0)
                   p->count++;   /* repeated word */
                else if (cond < 0) /* lower goes into left subtree
*/
                   p->left = tree(p->left, w);
                else             /* greater into right subtree */
                   p->right = tree(p->right, w);
                return(p);
        }
```

Memoria pentru noul nod este obtinuta de rutina "talloc" care este

o adaptare a rutinei "alloc" scrisa mai devreme. Aceasta returneaza un pointer a unui spatiu liber utilizabil pentru un nod. ANoul cuvint este copiat in locul pregatit de catre "strsave", contorul este initializat si cei doi copii sint facuti nuli. Aceasta parte a programului este executata numai la "marginea" copacului cind un nou nod este de adaugat. Am omis intentionat controlul erorilor la valorile returnate de rutinele "strsave" si "talloc".

Rutina "treprint" tipareste arborele incepind in ordine cu subarborele din stinga; in fiecare nod tipareste subarborele sting (toate cuvintele mai mici decit cel actual), apoi cuvintul actual, apoi subarborele drept (toate cuvintele mai mari). Daca va simtiti cam nesiguri pe recursiune imaginativa singuri un arbore si tipariti-l cu "tree print"; este una dintre cele mai pure rutine recursive pe care o puteti gasi.

```
treeprint(p) /* prnt tree p recursinely */
struct tnode *p;
{
   if (p != NULL)  {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

O observatie practica; daca arborele devine dezechilibrat din cauza cuvintelor care nu sosesc intr-o ordine aleatoare, timpul de rulare al progarmului poate sa creeasca prea repede. In cel mai rau caz, daca cuvtele sosesc deja sortate acest program devinnnnn o alternativa neeconomicoasa a cautarii lineare. Exista generalizari ale arborelui care nu sufera de acest neajuns, dar pe care nu le putem descrie aici.

Inainte de a parasi acest exemplu merita o scurta digresiune asupra problemelor legate de alocarea de memorie. Clar, este dezirabil ca intr-un program sa fie folosit un singur alocator de memorie chiar daca aloca diferite feluri de obiecte. Dar daca un alocator trebuie sa satisfaca cereri pentru pointeri de caractere "char" sau de "struct tnode", daca se pun intrebari. Prima, cum se satisface cerinta celor mai multe masini reale ca obiectele de diferite tipuri sa fie aliniate la adrese satisfacatoare(de exemplu intregii adesea trebuie aliniati pare)? A doua, ce declaratii pot satisface faptul ca "alloc" in mod necesar returneaza diferite tipuri de pointeri ?

Cererile de aliniament pot fi usor satisfacute, cu pretul unor spatii neutilizate, asigurindu-ne ca alocatorul intodeauna returneaza pointeri care intrunesc toate restrictiile de aliniament. De exemplu pentru PDP-11 este suficient ca "alloc" intodeauna sa returneze un "even" point, din moment ce orice tip de obiect poate fi memorat la o adresa "even". Masuri asemanatoare se iau pe alte masini. Astfel implementarea

unui "alloc" poate sa nu fie portabila) dar usajul este. "Alloc"-ul din capitolul 5 nu garanteaza nici un aliniament particular. In capitolul 8 vom prezenta modul de a face corect.

In legatura cu intrebarea asupra declaratiei tipului pentru "alloc"; i"c" cea mai buna metoda este aceea de a dlara ca "alloc" returneaza un pointer la "char" si apoi sa se converteasca ppoinerul in tipul dorit. Daca p este declarat astfel:

```
char *p;
```

atunci

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

il converteste intr-unn "tnode" pointer dintr-o exepresie.

Astfel "talloc" devine:

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

Aceasta este mai mult decit este nevoie pentru compilarile curente dar reprezinta cel mai sigur mod pentru viitor.

Exercitiul 6.4. Scrieti un program care citeste un program "c" si tipareste in ordine alfabetica fiecare grup al numelor de variaile care sint identice in primele 7 caractere dar diferite dupa aceea. (Asigurativa ca 7 este un parametru).

Exercitiul 6.5. Scrieti un porogram care tipareste o lista a tuturor cuvintelor dintr-un text si pentru fiecare cuvint o lista cu numerele liniei in care apar.

Exercitiul 6.6. Scrieti un program care tipareste cuvintele distincte, care sint introdues, in ordinea frecventei lor de aparitie. Fiecare cuvint sa fie precedat de numarul de aparitii.

6.6 Cautare in tabele

In aceasta sectiune vom da continutul unui pachet de cautare in tabel ca o ilustrare a mai multor aspecte legate de structuri. Amcest pachet este tipic pentru modul de explorare a unui tabel de simboluri pentru rutinele de management ale unui macroprocesor. De exemplu sa consideram declaratia din C,

"#define". Cind o linie ca

```
#define YES 1
```

este luata in considerare, numele YES si textul corespunzator 1 sint memorate intr-un tabel. Mai tirziu, cind numele YES apare intr- declaratie ca

```
inword = YES;
```

trebuie inlocuit cu 1.

Exista doua rutine majore care manipuleaza numele si textele de inlocuire corespunzatoare. "install(s, t)" inregistreaza numele "s" si textul de inlocuire "t" intr-o tabela; "s" si"t" sint siruri de caractere. "lookup(s)" cauta numele "s" intr-o tabela si returneaza un pointer la locul unde afost gasit ori NULL daca nu a fost gasit.

Algoritmul care se utilizeaza se bazeaza pe o cautare fragmentara "hash" un nume care se introduce este convertit intr-un intreg pozitiv care apoi este utilizat ca index intr-un tabloude pointeri. Un element al tablului pointeaza pe inceputul unui lant de balncuri care descriiu avind respectiva valoare de fragment ("hash"). Acesta este NULL daca nici un nume nu corespunde acestei valori.

Un bloc din lant este o structura care contine: pointeri, la nume, textul de inlocuire si urmatorul bloc din lant. Daca pointerul pentru urmatorul bloc este nul marcheaza sfirsitul lantului de blocuri.

```
struct nlist { /* basic table entry */
        char *name;
        char *def;
        struct nlist *next; /* next entry in chain */
};
```

Tabloul de poiunteri este:

```
#define HASHSIZE 100
static struct nlist *hashtab[HASHSIZE]; /* pointer table */
```

Functia de fragmentare ("hashing"), care este utilizata de ambele rutine "lookup" si "install", aduna valorile carcterelor din sirul nume si formeaza restul modulo dimensiunea tabloului. (Aceasta nu este cel mai bun algoritm posibil, darare meritul simplitatii.)

```
hash(s) /* form hash value for string s */
char *s;
{
        int hashval;
        for (hashval = 0; *s != '0';)
```

```
                hashval += *s++;
            return(hashval % HASHSIZE);
        }
```

Procesul de fragmentare ("hashing") produce un index de cautare in "hashtab"; sirul de carctere cautat se va gasi intr-un bloc din sirul de blocuri a carui inceput este pointat de elementul din tabelul "hashtab". Cautarea este efectuata de rutina "lookup". Dca "lookup" gaseste in "hashtab" intrarea deja prezenta, returneaza un pointer; daca nu, returneaza NULL.

```
        struct nlist *lookup(s) /* look for s in hashtab */
        char *s
        {
                struct nlist *up
                for (up = hashtab[hash(s)]; up != NULL; up = up-
>next)
                    if (strcmp(s, up->name) == 0)
                        return(up); /*found it */
                return(NULL); /* not found */
```

Rutina "install" foloseste "lookup" pentru a determina care dintre numele de instalat sint deja prezente; daca nu, o noua definire trebuie sa succeada uneia vechi. Altfel spus, o intrare complect noua este creeata. "install" returneza NULL daca dintr-un motiv oarecare nu mai este loc pentru o noua intrare.

```
        struct nlist *install(name, def) /*put(name, def) */
        char *name, *def; /* in hashtab */
        {
                struct nlist *np, *lookup();
                char *strsave(), *alloc();
                int hashval;
                if ((np = lookup(name)) == NULL)  { /* not found */
                   np = (struct nlist *) alloc(sizeof(*np));
                   if (np == NULL)
                      return(NULL);
                if ((np->name = strsave(name)) == NULL)
                      return(NULL);
                hashval = hash(np->name);
                np->next = hashtab[hashval];
                hashtab[hashval] = np;
                } else /* already there */
                      free(np->def); /*free previous definition */
                if ((np->def = strsave(def)) == NULL
                    return(NULL);
                return(np);
        }
```

"strsave" copiaza sirul furnizat de catre argumentul sasu intr_un loc obtinut cu un apel la functia "alloc". Am prezentat codul in capitolul 5 Deoarece apelurile la "alloc" si "free" pot sa apara in orice ordine si deoarece aici aliniamentul conteaza, versiunea simpla a "alloc" de la capitolul 5, nu este

adecvata aici vedeti capitolele 7 si 8.

Exercitiul 6.7. Scrieti o rutina care sa scoata un nume si definitia dintr-o tabela mentinuta cu "lookup" si "install".

Exercitiul 6.8. Implementati o versiune simpla a procesorului "#define" utilizabila in programe C, folosind rutinele din aceasta sectiune. Puteti gasi de ajutor si "getch" si "ungetch".

6.7 Cimpuri

Atunci cind spatiul de memorare estre la mare pret, poate fi necesar ca mai multe obiecte sa fie impachetate intr-un singur cuvint masina; un caz adesea folosit este compactarea fanioanelor de un singur bit necesare in aplicatii catabelele de simboluri ale compilatoarelor. Formatele externe impuse de interfetele cu diversele echipamente externe, adesea impun compactrea datelor pe un cuvint.

Sa ne imaginam un fragment dintr-un compilator care manipuleaza o tabela de simboluri. Fiecare identificator dintr-un program are anumite informatii asociate, de exemplu, este sau nu un cuvint cheie, e este sau nu extern si/sau static, s. a. m. d. Cel mai compact mod de a coda astfel de informatii este un set de fanioane de un bit cuprinse intr-un "char" sau "int".

Modul cel mai uzual este de a defini un set de masti corespunzatoare pozitiilor cu biti semnificativi, ca in

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC 04
```

(Numele trebuie sa fie puteri ale lui doi.) Apoi accesul la biti devine un soi de "bitareala" cu operatori de permutare, mascare si complementare, care au foat descrisi in capitolul 2.

Anumite idioame apar frecvent:

```
flags != EXTERNAL ! STATIC;
```

valideaza bitii EXTERNAL si STATIC ca fanioane, in timp ce

```
flags &= (EXTERNAL ! STATIC);
```

ii invalideaza.

Desi aceste idioame sint cu usurinta manevrate, limbajul C, mai degraba ofera capacitatea preferabila de adefini si accesa cimpuri Un cimp este un set de biti adiacenti cuprinsi intr-un singur "int". Sintaxa de definire si accesul cimpurilor este

bazata pe structuri. De exemplu tabela de simboluri definita mai sus ar putea fi inlocuita printr-o definire a trei cimpuri.

```
struct {
        unsigned is_keyword: 1;
        unsigned is_extern: 1;
        unsigned is_static: 1;
}flags;
```

Este astfel definita o variabila numita "flags" care contine trei cimpuri de cite un bit. Numarul de dupa ":" reprezinta lungimea cimpului in biti. Cimpurile sint declarate fara semn (unsigned) tocmai pentru a accentua ca sint cantitati fara semn.

Cimpurile individuale sint referite ca "flags.is_keyword", "flags.is_extern", etc, la fel ca oricare alti me, bri ai structurii. Cimpurile se comporta ca niste mici intregi fara semn si pot participa in expresii aritmetice la fel ca oricare alti intregi. Exemplele de mai sus pot fi rescrise mai natural astfel

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

pozitioneaza bitii pe unu;

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

pozitioneaza bitii pe zero;

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

pentru a-i testa.

Un cimp poate sa nu corespunda cu limitelke unui "int"; in acest caz cimpul este alinial la urmatoarea margine 'int". Cimpurile nu trebuie sa aibe un nume; cimpurile fara nume(doua puncte si lungimea numai) sint folosite ca umplutura. Lungimea speciala 0 poate fi folosita pentru alinierea la urmatoarea "int" margine.

Exista un numar de reguli de aplicat cimpurilor. Dintre cele mai importante, cipurile sint asignate de la stinga la dreapta in unele masini si de la dreapta la stinga in altele ceea ce reflecta natura diferita a hardwareului inseamna ca, desi cipurile sint utilizabile de preferinta pentru a intretine structurile de adte interne, chestiunea carui sfirsit vine primul trebuie considerata cu grija pentru prelucrarea datelor definite extern.

Alte restrictii care trebuie avute in minte: cimpurile sint fara semn ele pot fi memorate numai in "int"(sau echivalentul "unsigned"); ele nu sint tablouri; ele nu au adrese, astfel ca operatorul "&" nu le poate fi aplicat.

6.8 Uniuni

O uniune este o varaibila care poate pastra (la momente diferite) obiecte de diferite tipuri si dimensiuni, iar compilatorul tine seama de cerintele de dimensiune si aliniament. Uniunile permit sa se manipuleze diferite feluri de date intr-o singura zona de memorie, fara sa se includa in program nici un fel de informatii dependente de masina.

Ca de exemplu, sa consideram din nou o tabela de simboluri a unui compilator. Sa presupunem ca constantekle pot fi "int", "float" sau pointeri de caractere. Valoarea unei constante particulare trebuie memorata intr-o variabila de tipul corespunzator, dar este cel mai convenabil pentru organizarea tabelei daca valoarea ocupa aceasi dimensiune de memorie si acelasi loc in memorie in functie de tipul ei. Aceasta este scopul unei uniuni -sa permita ca o singura variabila care poate sa contina oricare din mai multe tipuri. La fel ca la cimpuri, sintaxa se bazeaza pe structuri.

```
union u-tag{
        int ival;
        float fval;
        char *pval;
} uval;
```

O variabila "uval" va fi suficient de mare ca sa pastreze oricare din cele trei tipuri, iar privitor la masina, codul generat de compilator este indiferent de caracteristicile hard. Oricare din aceste tipuri poate fi asignat la "uval" si apoi utilizat in expresii, atita vreme cit uzajul este considerat, adica tipul utilizat trebuie sa fie cel mai recent memorat. Este responsabilitatea programatorului sa tina seama de tipul de data curent memorat intr-o uniune; rezultatele sint dependente de masina daca ceva este memorat ca un tip si este extras ca altul.

Sintactic, membri unei uniuni sint accesati ca

```
union-name.member
```

sau

```
union-pointer->member
```

la fel cu structurile. Daca variabila "utype" este folosita pentru a tine seama
de tipul curent al marimii memorate in "uval", poate scrie un cod, astfel:

```
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("bad type %d in utype\n", utype);
```

Uniunile pot sa se gaseasca in structuri si tablouri si viceversa. Notatia pentru a accesa un membru a unei uniuni dintr-o structura(sau viceversa) este identica cu aceea pentru structurile in cuiburi. De exemplu in structura definita prin

```
        struct {
                char *name;
                int flags;
                int utype;
                union {
                        int ival;
                        float fval;
                        char *pval;
                } uval;
        } symtab[NSYM];
```

variabila "ival" este referita astfel

```
        symtab[i].uval.ival
```

iar primul caracter al sirului "pval" prin

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

In fond, o uniune este o structura in care toti membri au deplasamentul zero, structura este suficient de mare ca sa cuprinda cel mai mare membru, iar aliniamentul este potrivit pentru toate tipurile din uniune Ca si la structuri singurele operatii admise cu uniunile sint de a accesa un membru si obtinerea adresei lui;uniunile nu pot fi asignate, nu se pot aplica functii asupra lor, Pointeri la uniuni pot fi utilizati de o maniera identica pinterilor la structuri.

In capitolul 8 se arata cum alocatorul de memorie prin intermediul unei uniuni de memorie prin intermediul unei uniuni forteaza o variabila sa se alinieze la orice fel de particularitati de memorare.

6.9. Typedef

C admite o facilitate numita typedef pentru a crea nume pentru noi tipuri de date. De exemplu declaratia:

```
        typedef int LENGTH;
```

face numele LENGTH sinonim pentru "int". Tipul LENGTH poate fi utilizat in declaratii exact la fel ca int:

```
        LENGTH len, maxlen;
```

```
        LENGTH *lengths[];
```

Similar, declaratia:

```
        typedef char*STRING;
```

face ca STRING sa fie sinonim cu char* sau pointerul unui caracter, care poate fi utilizat in declaratii ca:

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

Observati ca tipul fiind declarat printr-un typedef apare in pozitia unui nume de variabila. Sintactic, typedef este ca o clasa de memorie, extern, static, etc. In cazul de mai sus am utilizat litere pentru a accentua numele.

Ca de exemplu mai complicat putem folosi typedef pentru nodurile copacului prezentat mai inainte in acest capitol:

```
        typedef struct tnode  { /* the basic node */
         char *word; /* points to the text */
         int count; /* number of occurrences */
         struct tnode *left; /* left child */
         struct tnode *right; /* right child */
        } TREENODE, *TREEPTR;
```

Aceasta creeaza doua noi tipuri de cuvinte cheie numite TREENODE (o structura) si TREEPTR (un pointer la structura), dupa care rutina "talloc" poate deveni:

```
        TREEPTR talloc()
        {
         char *alloc();
         return((TREEPTR) alloc(sizeofTREENODE)));
```

Trebuie sa specificam ca un typedef nu creaza noi tipuri; mai degraba renumeste tipurile existente. Variabilele astfel declarate au exact aceleasi proprietati ca si variabilele a caror declaratii sint explicite. In fond, typedef este ca un #define, exceptind ca, de cind este interpretat de copilator el face fata cu substitutiile in text care sint dincolo de capacitatile C macro procesorului. De exemplu:

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

creaza tipul PFI, care poate fi utilizat intr-un context ca

```
        PFI strcmpp, numcmp, swap;
```

din programul de sort din capitolul 5.

Exista doua motive mai importante pentru a utiliza declaratiile t9pedef. Primul este de a parametrizaun program alaturi de problemele de portabilitate. Daca pse utilizeza typedef pentru

diferite tipuri de date care pot fi dependente de masina, numai typedef necesita modificari daca programul este mutat pe alta masina. O situatie obisnuita este de a folosi typedef pentru cantitati intregi variate, cind se alcatuieste un set adecvat de "short", "int" si "long" pentru fiecare masina.

Un al doilea scop a lui typedef este de a da mai mare claritate unui program numit TREEPTR este mai usor de inteles decit unul declarat ca un simplu pointer la o structura complicata.

In final, exista deja posibilitatea ca in viitor, compilatorul sau * un alt program ca "lint" sa faca uz de informatia continuta in typedef ca sa execute niste controale in plus asupra programului.

CAPITOLUL 7. INTRARI SI IESIRI (I/O)

Facilitatile de I/O nu sint parte a limbajului C de aceea le-am introdus intr-un capitol separat. Nici un program real nu interactioneza cu mediul intr-un md mai complicat decit am prezentat pina acum. In acest capitol vom descrie biblioteca standard de I/O, care este un set de functii ce permit I/O standard pentru programele C. Functiile au menirea de a prezenta o interfata programabila convenabila, care sa permita operatiile prevazute de acele mai multe sisteme moderne de operare. Rutinele de I/O sint destul de eficinte incit foarte rar programatorul simte nevoia de a le ocoli chiar pentru aplicatii critice. In sfirsit, rutinele sint facute pentru a fi "0ortabile" in sensul ca ele exista intr-o forma compatibila pentru orice sistem unde C exista si ca programele care utilizeaza biblioteca standard de I/O pot fi mutate de pe un sistem pe altul fara modificari.

Nu vom incerca sa descriem aici intreaga biblioteca de I/O; sintem mai interesati de a arata esentialul despre scrierea programelor C in interactiune cu sistemul lor de operare.

7.1 Accesul la biblioteca standard

Fiecare fisier sursa care refera functiile bibliotecii standard trebuie sa contina linia

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

aproape de inceput. Fisierul stdio.h defineste unele macrouri si variabile folosite de biblioteca I/O. Folosirea semnelor < si> in loc de obisnuitele ghilimele duble indreapta compilatorul sa caute fisierul intr-un director continind informatiile de eticheta standard (pentru UNIX tipic |usr|include).

Mai departe, ar putea fi necesar ca atunci cind incarca programul sa se specifice biblioteca in mod explicit; de exemplu pentru PDP-11 UNIX, comanda pentru compilarea unui program ar fi

```
cc source files, etc-ls
```

unde -ls indica incarcarea din biblioteca standard (Caracterul l este litera el).

7.2. Intrari si iesiri standard-getchar si putchar

Mecanismul cel mai simplu de intrare este de a citi un caracter la un moment dat de la "standard input", care in general este terminalul utilizatorului, cu getchar. getchar() returneaza urmatorul caracter de intrare de cite ori este apelat. In cele mai multe medii care suporta C, un fisier poate fi substituit terminalului utilizind conventia <; daca un program "prog" foloseste getchar, atunci linia de comanda:

```
prog<infile
```

determina "prog" sa citeasca "infile" in loc de terminal. Comutarea intrarii este facuta de o astfel de maniera incit "prog" insusi nu simte aceasta; in particular, sirul "<infile>" nu este inclus in argumentele "argv" ale liniei de comanda. Comutarea intrarii este asadar invizibila daca intrarea vine de la un alt program via un "pipe mecanisme", linia de comanda

```
otherprogr|prog
```

executa ambele programe "otherprogr" si "progr" si aranjeaza astfel ca intrarea standard pentru "progr" vine de la iesirea standard a lui "otherprogr".

"getchar" returneza valoarea EOF cind intilneste sfirsitul fisierului indiferent de ce este citit la intrare. Biblioteca standard defineste constanta simbolica EOF ca fiind -1 (cu un #define in fisierul stdio.h) dar testele se fac cu EOF nu cu -1, astfel ca sa fie independent de valoarea specifica.

Pentru iesire, putchar(c) pune caracterul c la iesirea standard care este terminalul. Iesirea poate fi directata spre un fisier folosind ">" daca "progr" utilizeaza putchar,

```
prog>aufile
```

iesirea standard va fi "outfile" in loc de terminal. In sistemul UNIX se poate utiliza o comunicatie (pipe) spre alt program:

```
progr|anotherprogr
```

comuta iesirea standard a programului "progr" ca intrare standard a programului "anotherprogr". Si in aceste cazuri "progr" nu are

cunostiinta de redirectare.

Un numar destul de mare de programe au un singur sir de date ca intrare si un singur sir ca iesire; pentru astfel de programe I/O cu getchar putchar si printf pot fi in intregime adecvate, mai ales daca folosim facilitatile de redirectare si pipe pentru conectarea iesirii unui program cu intrarea altuia. De exemplu sa consideram programul "lower":

```
#include <stdio.h>
main() /* convert input to lower case */
{
        int c;
        while((c=getchar())!=EOF)
         putchar(isupper(c)?tolower(c):c);
}
```

Functiile "isupper" si "tolower" sint de fapt macrouri definite in stdio.h. Macrourile "isupper" testeaza care din argumentele sale este o litera mare, returnind in acest caz o valoare non-zero, sau o litera mica returnind in acest caz zero. Macroul tolower converteste o litera mare in una mica. Indiferent de cum aceste functii sint implementate pe diferite masini, mediul lor extern este acelasi, astfel ca prgramele sint scutite de cunoasterea setului de caractere.

Pentru convertirea unor fisiere multiple se poate folosi un program la fel ca utilitarul "cat" din UNIX pentru colectarea fisierelor:

```
cat file1file2...|lower>output
```

si aceasta ne permite sa invatam cum se acceseaza fisierele dintr-un program. ("cat" este prezentat mai tirziu in acest capitol).

In biblioteca standard I/O functiile getchar si putchar pot de fapt sa fie macrouri, si aceasta permite ca sa nu fie apelata o functie pentru fiecare caracter(overhead). Vom arata aceasta in capitolul 8.

7.3. Iesirea formatata-printf

Cele doua rutine "printf" pentru iesire si scanf pentru intrare permit translatarea in si din reprezentarea caracterelor a cantitatilor numerice. Ele de asemenea permit generarea sau interpretarea liniilor formatate. Am folosit deja "printf" in capitolele precedente ; aici ii dam o descriere mai precisa si complecta.

```
printf(control, arg1, arg2,....)
```

printf converteste, formateaza si tipareste argumentele sale la iesirea standard sub controlul sirului "control". Sirul de

control contine doua tipuri de obiecte: caractere ordinare, care sint simplu copiate in sirul de iesire si semnificatii de conversie dintre care fiecare cauzeaza conversia si tiparirea a urmatoarelor argumente succesive ale printf.

Fiecare specificatie de conversie este introdusa prin caracterul % si incheiata printr-un caracter de conversie. Intre % si caracterul de conversie pot fi:

Un semn minus care specifica cadrarea la stinga in cimp a argumentului convertit.

Un sir de digiti care specifica o lungime minima a cimpului. Numarul convertit va fi introdus in cimp la cel putin aceasta lungime sau mai mare daca este necesr. Daca argumentul are mai putine caractere decit dimensiunea cimpului el va fi cadrat la dreapta sau la stinga (in funcite de specificatie sau implicit) si va fi completatcu zero sau spatii pina la lungimea cimpului. Caracterul de complectare este normal blank, iar daca dimensiunea cimpului a fost specificata prin "leading" zero, este zero (aceasta nu implica cimpurile octale).

O virgula, care separa lungimea cimpului de un alt sir de digiti.

Un sir de digiti (precizia), care specifica numarul maxim de caractere acceptate dintr-un sir, sau numarul de pozitii dupa virgula zecimala virgula flotanta si dubla precizie.

Caracterele de conversie si semnificatiile lor sint:

d-argumentul este convertit in zecimal

o-argumentul este convertit in octal fara semn

x-argumentul este convertit in hexazecimal fara semn

u-argumentul este convertit in zecimal fara semn

c-argumentul este preluat ca un singur caracter

s-argumentul este un sir ;caracterele din sir sint introduse pina cind este intilnit caracterul nul sau pina cind este atinsa lungimea specificata.

e-argumentul este luat ca virgula flotanta sau dubla precizie si convertit in notatia zecimala de forma [-]m. nnnnnnE[+sau-]XX unde lungimea sirului de n-uri este specificata de precizie. Precizia implicita este 6.

f-argumentul este luat ca virgula mobila sau dubla precizie si convertit tot intr-o notatie zecimala de forma [-]mmm. nnnnnn unde lungimea sirului de n-uri este specificat de precizie,

implicit este 6. De observat ca precizia nu determina numarul de digiti semnificativi in formatul f.

g-foloseste % sau %l , oricare este mai scurt ; zerourile nesemnificative nu apar. Daca caracterul de dupa % nu este un caracter de conversie acest caracter este tiparit chiar 5 poate fi tiparit prin %%.

Cele mai multe formate de conversie sint evidente si au fost ilustrate in capitolele precedente. oO exceptie este precizia referitoare la siruri. Tabela urmatoare arata efectele unei varietati de specificatii asupra sirului "hello, world" de 12 caractere. Am pus doua puncte pentru a putea urmari mai bine lungimea.

```
::%10s: :hello, world:
:5-10s: :hello, world:
:%20s:  :       hello, world:
:%-20s: :hello, world   :
:%20. 10s:      :       hello, wor:
:%-20. 10       :hello, wor     :
:%. 10s:        :hello, wor:
```

Atentiune: "printf" foloseste primul sau argument pentru a decide cite argumente urmeaza si de ce tip. Daca nu sint destule argumente sau de tip gresit veti obtine rezultate lipsite de sens.

Exercitiul 7.1 Scrieti un program care tipareste diferentiat o intrare arbitrara. Ca o conditie minima trebuie sa tipareasca caracterele non-grafice in octal sau hexa(conform uzantelor locale).

7.4. Intrarea formatata-scanf

Functia scanf este intrarea analog printf-iesirea, admitind aceleasi conversii in sens invers.

```
scanf(control, arg1, arg2,...)
```

scanf citeste caractere de la intrarea standard, le interpreteaza conform formatului specificat in "control" si memoreaza rezultatele in celelalte argumente, care sint pointere ce indica unde vor fi depuse datele convertite.

Sirul de "control" contine de obiecei specificatii de conversie, care sint utilizate pentru o interpretare directa a secventelor de intrare. Sirul de control poate sa contina:

-blancuri, taburi, newlines" care sint ignorate (sint numite "caractere albe")

-caractere ordinare (nu%)

-specificatii de conversie continind caracterul % si optional caracterul de suprimare, un numar optional de specificare a lungimii maxime a cimpului si un caracter de conversie

O specificatie de conversie determina conversia urmatorului cimp de intrare. Normal rezultatul este plasat in variabila pointata de argumentul corespunzator. Daca se indica suprimarea prin caracterul * oricum cimpul de la intrare este ignorat 9se sare peste el) si nu se face nici o retinere de spatiu. Un cimp de la intrare este definit ca un sir de non "caracter alb " fie pina la lungimea specificata a cimpului.

Caracterul de conversie indica interpretarea cimpului de la intrare argumentul corespunzator trebuie sa fie un pointer asa cum este cerut de semantica "C" limbajului "C". Urmatoarele caractere de conversie sint legale:

d-un intreg zecimal este asteptat la intrare ;argumentul corespunzator trebuie sa nu fie pointer intreg.

o-un intrteg octal este asteptat la intrare ;argumentul corespunzator trebuie sa fie un pointer de intreg.

x-un intreg hexazecimal este asteptat la intrare ; argumentul corespunzator trebuie sa fie un pointer de intreg.

h-un intreg "short" este asteptat la intrare: argumentul trebuie sa fie un pointer de intreg "short" ("scurt").

c-un singur caracter este asteptat ; argumentul corespunzator trebuie sa fie un piinter de caracter. In acest caz ignorarea "caracterelor albe " este suprimata ; pentru a citi urmatoreul caracter altul de cit "caracterele albe" se foloseste %1s.

s-un sir de caractere este asteptat ; argumentul corespunzator trebuie sa fie un pointer al unui tablou de caractere destul de mare pentru a incapea sirul si nun terminator `0 care va fi adaugat.

f-un numar in virgula flotanta este asteptat ; argumentul corespunzator trebuie sa fie un pointer la un cimp "float". Un caracter de conversie e este

sinonim cu f. Formatul prezentat la intrare pentru un "float" este alcatuit dintr-un semn optional, un sir de numere care pot sa contina si un punct zecimal si un cimp de exponent care este format din E sau e urmat de un intreg cu semn.

Caracterele de conversie d, v si x pot fi precedate de litera l

pentru a indica un pointer la "long" mai gdegraba decit "int" care apare in list ta de argumente. Similar litera l inainte de E sau f indica un pointer la "double" in lista de argumente.

De exemplu:

```
int i;
float x;
char name[50];
scanf("%d%f%s", Di, Dx, name);
```

cu linia de intrare

```
25 54. 32E-1 Thompson
```

va asigna valoarea 25 lui i, 5. 432 luix si plaseaza sirul "Thompson" terminat prin `0in "name". Cele trei cimpuri de la intrare pot fi separate de orice blancuri, tab-uri si newline-uri. Apelarea:

```
int i;
float x;
char name[50];
scanf("%2d%f%*d%2s, Di, Dx, name);
```

cu intrarea

```
56789 0123 45a72
```

va asigna lui i, 789. 0 lui x, va sari peste 0123 si plaseaza "45" in "name". Urmatoarea apelare la orice rutina de introducere va incepe cautarea cu litera a. In aceste doua exemple "name" este deja un pointer si de aceea nu trebuie precedat de D.

Ca un alt exemplu, calculul rudimentar de la capitolul 4 poate fi acum rescris cu scanf pentru a face conversia de intrare.

```
#include<stdio.h>
main() /* rudimentary desk calculator */
{
        double sum, v;
        sum=0;
        while(scanf("%lf", Dv)!=EOF)
         printf("`t%. 2f`n", sum+=v);
}
```

scanf se popreste cind se epuzeaza sirul de control ori cind data de intrare difera prea mult de specificatia de control. Este returnata o valoare egala cu numarul de date de intrare introduse cu succes. La sfirsitul fisierului este returnat EOF ; de observat ca acesta este diferit de 0 , ceea ce inseamna ca urmatorul caracter de la intrare nu se mai converteste prin prima specificatie din sirul de control. Urmatorul apel la scanf se

rezuma sa caute imediat dupa ultimul caracter deja returnat.

Un avertisment final: argumentele lui scanf trebuie sa fie pointeri. De departe cea maiobisnuita eroare este sa scrii

```
scanf("%d", n);
```

in loc de

```
scanf("%d", Dn);
```

7.5. Conversii de format in memorie

Functiile scanf si printf au corespondente functiile sscanf si sprintf care executa aceleasi tipuri de conversii, dar care opereazza asupra unui sir nu asupra unui fisier. Formatul general este:

```
sprintf(string, control, arg1, arg2,...)
sscanf(string, control, arg1, arg2,...)
```

sprintf formateaza argumentele arg1, arg2, etc, conform sirului "control" ca mai inainte, dar plaseaza rezultatele in "string" in loc de iesirea standard. Desigur "string" trebuie sa fie suficient de mare pentru a primi rezultatul. Ca exempolu, daca "name" este un tablou de caractere si n este un intreg, atunci:

```
sprintf(name, "temp%d", n);
```

creaza un sir de forma tempnnn in "name", unde nnn este valoarea lui n.

sscanf face conversia inversa -imparte sirul "string" conform formatului din "control" si plaseaza valorile rezultate in arg1, arg2, etc. Aceste argumente trebuie sa fie pointeri. Astfel: sscanf(name, "temp%d", @n); n la valoarea digitilor din sir care urmeaza dupa temp in "name".

Exercitiul 7.2. Rescrieti exemplul de calculator din capitolul 4 utilizind scanf si/sau sscanf pentru a face intrarea si conversia numerelor.

7.6 Acces la fisiere

Programele scrise pina acum toatecitesc intrarea standard si scriu iesirea standard.

Urmatorul pas in I/O este de a scrie un program care acceeaza un fisier care nu este deja conectat programului. Un program care ilustreaza necesitatile pentru astfel de operatii este "cat", care conecteaza un set de fisiere specificate spre iesirea standard. "cat" este utilizat pentru tiparirea fisirelor la

terminal si ca un scop general este utilizat ca un colector de intrare pentru programele care nu au capabilitatea de a accesa fisirele prin nume. De exemplu comanda:

```
cat x. c y. c
```

tipareste continutul fisierelor x. c si y. c la iesirea standard.

Intrebarea este cum sa aranjam ca fisierele specigficate sa fie citite adica cum sa controlam numele externe cu declaratiile care de fapt citesc datele.

Regulile sint simple. Inainte de a se citi sau scrie intr-un fisier, l trebuie deschis cu ajutorul functiei fopen din biblioteca standard. fopen ia un nume extern (ca x. c sau y. c), face citeva manipulari si negocieri cu sistemul de operare(detalii care nu ne intereseaza) si reruneaza un nume intern care trebuie utilizat ulterior la citirile si scrierile fisireului.

Acest nume intern este de fapt un pijnter numit "file pointer", la o structura care contine informatii despre fisiere ca, adresa unui buffer pozitia curenta in bufer, daca se citeste sau se scrie in fisier, s. am. d. Utilizatorii nu trebuie sa cunoasca detaliile deoarece o parte din definirile standard I/O obtinutede la "stdio.h" este o definire de structura numita FILE. Singura declaratie necesara pentru un pointer de fisier (file pointer) este exemplificata de

```
FILE*fopen(), *fp;
```

Aceasta spune ca fp este un pointer la FILE, si fopen returneaza un pointer la FILE. Observati ca FILE este un nume de tip, la fel cu "int" nu o eticheta de structura: este implementat ca un typedef(Detalii in capitolul 8).

Actuala apelare a lui fopen in program este:

```
fp=fopen(name, mode);
```

Primul argument a lui fopen este numele (name) fisierului, care este un sir de caractere. Al doilea argument este modul "mode", de asemenea un sir de caractere care indica cum se intentioneaza sa fie utilizat fisier ul. Modurile permise sint citire (read-"r"), scriere(write-"w") sau adaugare ("a").

Daca deschideti un fisier care nu exista: pentru scriere sau adaugare, el este creat (daca este posibil). Daca se deschide un fisier existent in scriere, acest fisier este sters. Incercarea de a citi un fisier care nu exista constituie o eroare. Mai exista si alte manevre care cauzea erori ca incercarea de aciti un fisier pt. care nu aveti permisiunea. Daca exista vreo eroare, fopen returneaza in pointerul nul valoarea NULL (care pt. convenienta este de asemenea definita in stdio.h).

Urmatorul pas este de aciti sau scrie fisierul odata deschis. Aici exista mai multe posibilitati dintre care getc si putc sint cele mai simple. getc returneaza urmatorul caracter din fisier ;are nevoie de fisier pentru a-i specifica care fisier. Astefl

```
c=getc(fp)
```

plaseaza urmatorul caracter din fisierul referit prin fp in c si EOF cind ajunge la sfirsitul fiserului.

```
putc este inversul lui getc:
putc(c, fp)
```

pune caracterul c in fisierul fp si returneaza c. La fel cu getchar si puchar, getcsi putc pot fi macrouri in loc de functii.

Cind este demarat un program, trei fisiere sint deschise in mod automat si sint furnizate pointerele pentru ele. Aceste fisiere sint intrarea standard, iesirea standard, si iesirea standard pentru erori ; pointerele de fisiere corespunzatoare sint stdin, stdout si stderr. Normal toate acestea sint conectate la terminal, dar stdin si stdout pot fi redirectate asa cum s-a descris in paragraful 7.2.
getchar si putchar pot fi definite cu ajutorul lui getc si putc, stdiu si stdout astfel:

```
#define getchar() getc(stdiu)
#define putchar(c)putc(c, stdout)
```

Pentru intrarea sau iesirea formatata afisierelor, functiunile fscanf s si fprintf pot fi utilizate. Acestea sint identice cu scanf si printf, de retinut ca primul argument, este un pointer de fisier care specifica fisierul de citit sau scris ; sirul de control este al doilea argument.

Cu aceste preliminarii, sintem acum in pozitia de a rescrie programul "cat" de concatenare a fisierelor. Premisa de baza este cea care a fost gasita convenabila pentru atitea programe pina acum: dacaexista linii de comanda cu argumente, sint executate in ordine. Daca nu exista argumente este preluata intrarea standard. Sub aceasta forma programul poate fi folosit de sine statator sau ca o parte a unora mai mari.

```
#include <stdio.h>
main(argc, argv) /* cat:concatenate files */
int argc;
char *argv[]
{
        FILE*fp, *fopen();
        if(argc==1) /* no args;copy standard inp */
         filecopy(stdiu);
        else
```

```
                while(--argc>0)
                        if((fp=fopen(*++argv, "r"))==NULL){
                         printf("cat:can't open%s`n", *argv);
                         break;
                }else{
                         filecopy(fp);
                         fclose(fp);
                        }
        }
        filecopy(fp) /*copy file fp to standard output */
        FILE *fp
        {
                int c:
                while((c=getc(fp))!=EOF)
                        putc(c, stand);
        }
```

Pointerele de fisiere stdiu si stdout sint predefinite in biblioteca standard I/O ca iesirea si intrarea standard: ele pot fi utilizate oriunde poate fi utilizat un obiect de tipul FILE, Ele sint constante , nu variabile asa ca nu incercati sa le asigurati.

Functia fclose este inversul lui fopen ; ea intrerupe conexiunea intre pointerul fisierului si numele extern al fisierului, care a fost stabilita de fopen, eliberind pointerul fisierului pentru un alt fisier. Deoarece cele mai multe sisteme de operare limiteaza numarul de fisiere dechise simultan de un program, este o idee buna de a le elibera in momentul cind nu maio este nevoie de ele, asa cum am facut in programul cat. Exista de asemenea un alt motiv pentru a face fclose pe un alt fisier in iesire -curata bufferul in care putc colecteaza iesirea. fclose este apelat automat pentru fiecare fisier dechis atunci cind programul se termina normal.

7.7 Tratarea erorilor -stderr si exit

Tratamentul erorilor in cat nu este ideal. Problema este ca daca unul d din fisiere nu poate fi accesat din vreun motiv oarecare, diagnosticul erorii este tiparit numai la sfirsitul iesirii concatenate. Aceasta este aceptabil dacaiesirea se face pe un terminal, dar este rau daca iesirea este un fisier sau un alt program via o pipeline.

Pentru a trata aceasta situatie maibine, un al doilea fisier de iesire numit stderr este asignat programului, la fel castdin si stdout. Daca e este posibil, stderr apare pe terminalul utilizatorului chiar daca iesirea standard este redirectata.

Sa revizuim programl cat in asa fel ca mesajele de eroare sa apara in fisierul standard de erori

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

```
        main(argc, argv) /* cat: concatenate files *?
        int argc:
        char *argv[];
        {
         FILE*fp, *fopen();
         if(argc==1) /* no args: ; copy standard input */
                filecopy(stdiu);
         else
                while(--argc>0)
                 if((fp=fopen(*++argv, "r"))==NULL){
                        fprintf(stderr, "cat:can't open%s`n",
*argv);
                 }else{
                        filecopy(fp);{
                        fclose(fp);
                 }
                exit(0);
        }
```

Programul semnaleaza erorile in doua feluri. Diagnosticul de iesire produs de catre fprintf merge in stderr, astfel el gaseste drumul catre terminalul utilizatorului in loc sa dispara printr-un pipeline sau fisier de iesire.

Programul de asemenea utilizeaza si functia exit din biblioteca standard care termina executia programului cind este apelata. Argumentul lui exit este utilizabil de catre orice proces care il apeleaza, astfel succesul sau esecul programului poate fi testat de un alt program pentru care cel dinainte este un subprogram. Prin conventie returnarea valorii 0 semnaleaza ca totul ste OK, iar diferite valori nonzero semnifica situatii anormale.

exit apeleaza fclose pentru fiecare fisier deschis in iesire pentru a curata bufferul, apoi cheama rutina numita -exit. Functia exit determina terminarea imediata fara curatarea bufferului desigur exit poate fi apelata si direct.

7.8. Introducerea si extragerea unei linii

Biblioteca standard contine o rutina numita fgets care este similara functiei getline pe care am utilizat-o pina acum. Apelarea:

```
        fgets(line, MAXLINE, fp)
```

citeste urmatoarea linie de la intrare (incluzind newline) din fisierul fp in tabloul de caractere numit line ; cel muklt MAXLINE-1 caractere vor fi citite. Linia rezultata este terminata prin \0. Normal fgets returneaza linia ; la sfirsitul fisierului returneaza NULL. (Getline returneaza lungimea liniei si zero pentru sfirsitul fisierului).

Pentru iesire, functia scrie un sir(care nu trebuie sa contina newline) intr-un fisier.

```
fputs(line, fp)
```

Pentru a arata ca nu exista nimic magic cu functiile fgets si fputs mai jos sint copiate din biblioteca standard de intrare /iesire.

```
#include <stdio.h>
char *fgets(s, n, iop) /* get at most n chars from iop */
char *s
int n;
register FILE*iop;
{
        register int c;
        register char *cs;
        cs=s
        while(--n>0$$(c=getc(iop))!=EOF)
         if((*cs++=c)=='\n')
                break
        *cs='\0';
        return((c==EOF$$cs==s)?NULL:S);
}
fputs(s, iop) /* put string s on file iop */
register char *s;
register FILE *iop;
{
        register int c;
        while(c=*s++)
                putc(c, iop);
}
```

Exercitiul 7.3. Scrieti un program de comparare a doua fisiere, si tipariti prima linie si pozitia caracterelor cind difera.

Exercitiul 7.4. Modificati programul de gasire a tipului din capitolul 5 asa fel incit sa aibe intrarea dintr-un set de fisiere de intrare numite iar daca nu sint numite fisiere de intrare, de la intrarea standard. Trebuie tiparit numele fisierului cind este detectata o linie potrivita ?

Exercitiul 7.5. Scrieti un program de tiparire a unui set de fisiere, in care fiecare sa inceapa pe pagina noua cu un titlu si un contor de pagini pentru fiecare fisier.

7.9. Citeva functii amestecate

Biblioteca standard pune la dispozitie o varietate de functii, citeva fiind deosebit de utile. Am mentionat deja functiile cu siruri strlen, strepy, strcat si strcmp. Urmeaza alte citeva.

Testarea clasei caracterelor si conversia lor

Mai multe macrouri executa teste asupra caracterelor si conversia lor:

isalpha(c) nonzero daca c este alfabetic si zero daca nu.
isupper(c) nonzero daca c este litera mare (upper case)si 0 dacanu.
islower –nonzero daca c este caracter mic (lower case)si 0 daca nu.
isdigit –nonzero daca c este digit si zero daca nu.
isspace –nonzero daca c ete blanc, tab sau newline si 0 daca nu.
toupper –converteste c in caracter mare (upper case)
tolower(c) –converteste c in caracter mic (lower case).

Ungetc

Biblioteca standard contine o versiune restrictiva a functiei ungetch pe care am scris--o in capitolul 4; se numeste ungetch.

```
ungetch(c, fp)
```

impinge caracterul c inapoi in fisierul fp. Numai un caracter din fisierc poate fi tratat astfel ungetc poate fi utilizat cu oricare din functii le si macrouruile de introducere ca scanf, getc sau getchar.

Apel system

Apelarea functiei system(s) executa comanda continuta in sirul de caractere s apoi reia programul curent. Continutul lui s depinde mult de sistemul de operare. Ca un exemplu ordinar in UNIX linia

```
system("date");
```

determina rularea parogramului date care tiparesete data si momentul zilei.

Managementul memoriei

Functia calloc este asemanatoare cu alloc pe care am utilizat-o in capitolele precedente.

```
calloc(n, sizeof(object))
```

returneaza un pointer daca este suficient spatiu pentru cele n obiecte de dimensiunea specificata sizeof sau NULL daca cererea poate fi satis facuta. Memoria este initializata cu zero.

Pointerul are aliniamentul adecvat obiectelor respective, dar el trebuie introdus intr-un tip corespunzator.

cfree(p) elibereaza spatiul pointat prin p unde p este initial obtinut prin apelarea lui calloc. Nu exista restrictii asupra ordinii in care spatiul este eliberat, dar este o mare

greseala sa eliberezi ceva ce nu ai obtinut prin apelarea functiei calloc.

Capitolul 8 prezinta implementarea unui alocator de memorie ca calloc, in care blocurile alocate pot fi eliberate in orice ordine.

Anexa A. Manual de referinta C

1. Introducere

Acest manual descrie limbajul C implementat pe PDP-11, Honeywele 6000, IBM System\370 si Interdata 8\32. In blocurile unde exista diferente (intre implemntari) el se refera la PDP-11, dar se incearca punctarea detaliilor dependente de implementare. Cu mici diferente, aceste dependente sint legate direct de proprietatile hardware (ale masinilor); diverselel compilatoare sint in general compatibile.

2. Conventii lexicale

Exista sase clase de simboluri: identificatori, cuvinte-cheie, constante, (in mod colectiv numite "spatii albe")asa cum vor fi descrise in continuare. sint ignorate cu exceptia cazurilor cind servesc la repararea simbolurilor. Unele spatii albe sint necesare pentru a separa identificatori adiacenti, cuvint-cheie si constante.

Daca sirul de intrare (pentru compilator) a fost analizat (si transformat) in simboluri pina la un caracter dat, urmatorul simbol va include cel mai lung sir de caractere care poate constitui un simbol.

2. 1. Comentarii

Caracterele /* incep un comentariu, care se termina cu caracterele*/. Comentariile nu pot fi imbricate.

2. 2 Identificatori (nume)

Un identificator este o secventa de litere si cifre;primul caracter trebuie sa fie litera. Sublinierea (underscor) este

considerata litera literele mari si mici sint diferite. dDoar primele 8 caractere sint semnificative, desi pot fi folosite mai multe. Identificatorii externi care sint utilizati de o serie de asamblare sau incarcatoare, sint mai restrictivi:

PDP-II 7 caractere, 2cazuri, casete (cases)

Homywele 6000 6caractere, 1caset.

IBM 360/370 7caractere, 1case

Interdata 8/32 8caractere, 2 cases

2. 3. Cuvinte cheie

Urmatorii idenntificatori sint rezervati pentru a fi folositi ca si cuvinte cheie, si nu pot fi folositi altfel:

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

Cuvintul cheie "entry" nu este implementat de nici un compilator, dar este rezervat pentru o utilizare ulterioara. Unele implementari rezerva si cuvintele "fortran" si "asm"

2. 4. Constante

Exista o serie de contante decrise in continuare. Caracteristicile hardware care afecteaza dimensiunile sint rezumate in $2. 6.

2. 4. 1. Constante intregi

O constanta intreaga constind dintr-o serie de cifre va fi considerata in octal daca incepe cu 0 (cifra zero), astfel va fi considerata in zecimal. O secvanta de cifre precedata de ox sau OX (cifra zero) este considearata intreg hexazecimal. Daca valoarea unei constante zecimale depaseste posibilitatile hardware de inmagazinare a intregilor ea va fi considerata de tip "long". Acelsi lucru se intimpla si pentru constantele octale sau hexazecimale.

2. 4. 2. Constantelungi explicite

O constanta intreaga de tip octal, zecimal sau hexazecimal urmata de caracterul "l"sau"L" este considerata consatnta lunga. Dupa

cum va fi discutat la unele masini intregii si valorile "long" pot fi considerate identice.

2. 4. 3. Constante tip caracter

O constanta tip carcter este un caracter intre ghilimele simple cum ar fi de exemplu '*'. Valoarea unei constante tip caracter este valorea numerica a caracterului corepunzator setului de caractere al masinii. Unele caractere negrafice, cum sint ghilimeaua si backspace pot fi reprezentate dupa cum urmeaza, conform unei tabele de secvente "escape".

newline	NL(LP)	\n	
horisontaltab	HT	\t	
backspace	BS	\b	
carriagereturn	CR	\r	
formfeed	FF	\f	
backslash	\	\\	
ghilimea	'	\'	
modelbinar	ddd	\ddd	"bit pattern"

Secventa \ddd consta din caracterul backslash uramt de 1, 1 sau 3 cifre octale care vor specifica valorea unui caracter dorit. Un caz special de constructie este \0 (neurmat de nici o cifra) care indica un caracter NULL. Daca caracterul care urmeaza dupa \ nu este de tipul specificat, se ignora \.

2. 4. 4. Constante in virgula mobila

O constanta in virgula mobila consta dintr-o parte intreaga, punct zecimal parte fractionara un "e" sau "E" si optional s un intreg cu semn pe post de exponent.

Intregul si partea fractionara pot fi o secvente de cifre. Atit intregul cit si partea fractionara (dar nu simultan) pot lipsi; si punctul zecimal sau "e" si exponentul (nu simultan) pot lipsi. Constantele sint in dubla precizie reprezentate.

2. 5. Siruri

Un sir este o secventa de caractere intre ghilimele duble, de exemplu "..." Un sir este de tipul "tabloul de caractere " si in clase de memorare "static" (vezi &4) si se initializeaza cu caracterele date. Toate sirurile, chiar daca sint scrise identic, sint distincte. Compi latorul plaseaza caracterul \0 (byte nul) la sfirsitul fiecarui sir astfel ca programele care pot detecta sfirsitul sirului. Intr-un sir, car acterul " trebuie precedat de \; in plus, secventele deschise pentru constante tip caracter pot fi folosite. In sfirsit , un \ si un NL care urmeaza imediat sint ignorate.

2. 6. Caracteristici hardware

Tabelul care urmeaza rezuma unele proprietati de hardware care

variaza de la masina la masina. Acestea afecteaza portabilitatea programeleor; in practica aceasta este o problema care trebuie privita aprioric.

	DEC PDP11	HONEYWELL	IBM370	INTERDATA8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8biti	9	8	8
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64
gama reprez.	I10-38 +	I10-38 +	I10-76 +	I10-76 +

Pentru aceste patru tipuri de masina, numerele in VF au exponentul de 8 biti.

3. Notatii sintactice

In manual notatiile sintactice sint scrise cu tipul italic de caracter cuvintele literale sint caractere tipul drept. Categoriile alternative sint listate pe linii separate. Un simbol optional terminal sau nonterminal este indicat prin subscrierea "opt", astfel: {expresia opt} indica o expresie optionala intre acolade. Sintaxa este rezumata in $18.

4. Ce este un nume ?

C-ul isi bazeaza interpretarea unui identificator de doua atribute ale identificatorului: "clasa de memorare " si "tipul". Clasa de memorare determina locatia si durata de viata a unei celule de memorare asociate unui identificator; tipul determina semnificatia valorii depuse in celula de memorare a identificatorululilL.

Exista patru clase de memorare declarabile: automatic, staticexterna si tip registru. Variabilele automatice sint locale fiecarei apelari a unui bloc (vezi $9. 2) si sint declasate la iesirea din blocul respectiv. Variabilele statice sint locale unui bloc, dar retin valoarea pina la revenirea in blocul respecitv chiar daca controlul a parasit blocul; variabilele exxterne exista si pastreaza valoarealor in decursul executiei intregului program si pot sa fie folosite pentru comunicare intre functii chiar funcctii compilate separat. Variabilele de tip registru sint numerotate (daca este posibil) in registrii rapizi ai masinii, ca si variabilele automate sint locale blocului si dispar la iesirea din bloc.

C suporta citeva tipuri fundamentale de obiecte:

Obiectele declarate ca si caractere (char) sint sufiecient de mari pentru a sufoca orice membru a setului de caractere ale implementarii, si daca un caracter pur din acest set de caractere este sufocat intr-o variabila de tip caracter, valoarea sa este echivalenta cu codul intreg pentru acel caracter (cu valoarea intregului care reprezinta caracterul in reprezentarea interna (NI)) Si alte cantitati pot fi sufocate in variabile tip caracter, dar implementarea este dependenta de masina.

Sint disponibile 3 dimensiuni de intregi declarati short int, int si long int. Intregii lungi permit starii mai mari decit cei scurti, dar implementarea poate face ca intregii scurti, sau cei lungi, sau ambii sa devina echivalenti cu intregii de baza. Acestia au dimensiunea naturala decurgind din arhitectura masinii, celelalte tipuri sint prevazute pentru cerinte speciale.

Intregii fara semn, declarati unsigned se supun legilor aritmeticii modulo 2n unde n este numarul de bitdin reprezentare. (La PDP11 cantitatile lungi fara semn nu sint suportate)

Cantitati in VF simpla (float) sau dubla (double) pot sa fie sinonime in unele implementari.

Pentru ca cantitatile descrise pot fi interpretate ca numere ce sint referite ca fiind de tip "aritmetic". Tipurile char si int de toate dimensiunile vor fi in mod colectiv numite de tip "intreg". (float si double vor fi denumite de tip "flotant".

In afara tipurilor aritmetice de baza exista, conceptual, o clasa infinita de tipuri derivate din tipurile de baza in modul urmator:

-tablouri de obiecte de multe tipuri
-functii care dau obiecte de un tip dat
-pointeri la obiecte de un tip dat
-structuri constituind o secvnta de obiecte de tipuri variate
-marimi capabile sa contina oricare unitate de obiecte de tipuri varaibie

In general aceste metode de constructie a obiectelor pot fi aplicate recursiv.

5. Obiecte si lvalori

Un obiect este o regiune manipulabila de memorare; o LVALOARE este o expresie referitoare la un obiect. Un ex simplu de expresie LVALOARE este un identificator. Exista operatori care creaza (produc) lvalori: de ex, daca E este o expresie de tipul pointer, atunci *E este o expresie lvaloare referitoare la obiectul spre care pointeaza E. Numele

"lvaloare" vine de la expresia de asignare E1=E2 in care operandul sting E1 trebuie sa fie o expresie "lvaloare". Discutarea, mai jos, a fiecarui operator indica cind se asteapta operanzi "lvaloare " si cind operatorul produce o "lvaloare".

6. Conversii

Un numar de operatori depinzind de operanzii lor sa cauzeze conversia unei valori a operandului dintr-un tip in altul. Aceasta sectiune explica rezultatul care se asteapta in urma unei astfel de conversii; &6. 6 rezuma conversiile cerute de majoritatea operatiilor curente (ordinare N. T) tabloul va fi imbogatit dupa (prin) discutarea fiecarui operator.

6. 1. Caractere si intregi

Un caracter sau un intreg scurt pot fi folosite oriunde un intreg se poate folosi. In toate cazurile valoarea este convertita intr-un intreg. Conversia unui intreg scurt intr-unul lung implica o extensie de semn; intregii sint cantitati cu semn. Extensia de semn pt caractere depinde de masina, dar se garanteaza ca un membru al unui set de caractere standard este ne-negativ. Dintre masinile discutate aici dar la PDP-11 se extinde semnul. La PDP-11 valoarea variabilelor tip caracter este in gama -128 la +127; caracterele alfabetului ASCII sint toate pozitive. O constanta tip caracter specificata octal sufera o extensie de semn si poate apare negativa de exemplu '\377' are valoarea -1.

Cind un intreg mai lung este convertit intr-unul mai scurt sau char el este trunchiat la stinga; bitii in exces sint deplasati (pierduti).

6. 2. Float si double

Toata aritmetica in C este realizata in dubla precizie; cind apare un float intr-o expresie el este "lungit" (extins) in double prin introducere de zeroruri in partea sa fractionara. Cind un double este convertit in float, de exemplu printr-o asignare, double-ul este rotunjit inaintea de trunchiere pe lungimea unui float.

6. 3. Flotante si intregi

Conversia unei valori flotante in tip intreg devine dependenta de masina in particular directia de trunchiere a numerelor negative variaza de la masina la masina. Rezultatul este nedefibnit daca valoarea nu incape in spatiul prevazut (N. T. pt un intreg). Conversiile din intreg in flotant sint realizate bine. Poate apare o pierdere a preciziei daca detinatia nu are un numar suficient de biti.

6. 4. Pointeri si intreg

Un intreg sau un intreg lung poate fi adunat sau scazut dintr-un pointer; in acest caz primul este convertit asa cum se specifica la operatia de adunare. Doi pointeri la obiecte de acelasi tip pot fi scazuti; in acest caz rezultatul este convertit intr-un intreg asa cum se va discuta la operatia de scadere.

6. 5. Fara semn

Oridecite ori un intreg fara semn si un intreg normal (de baza tipic) sint combinati, intregul tip este convertit in intreg fara semn si rezultatul va fi fara semn. Valoarea este cel mai mic intreg fara semn congruent intregului cu semn (modulo2 si lungimea caracterului). In reprezentarea in complement fata de 2, conversia este conceptuala si nu se modifica tabloul (distributia) bitilor. Cind un intreg fara semn este convertit in long , valoarea rezultatului este aceasi numeric, ca si a intregului fara semn. Deci conversia este introducere de zerouri spre stinga.

6. 6. Conversii aritmetice

Multi operatori cauzeaza conversii si produc rezultate de un tip oarecare, intr-un mod similar. Acest model va fi numit "conversii aritmetice uzuale". Intii, oricare operand de tipul char sau short va fi convertit in int, iar oricare operand de tipul float este convertit in tipul double.

Apoi, daca un operand este de tip double, atunci si celalalt este convertit in double si acesta va fi tipul rezultatului. Astfel daca un operand este long, celalalt este convertit in long si acesta va fi tipul rezultatului. Astfel, daca un operand este unsigned, celalalt este convertit in unsigned si acesta va fi tipul rezultatului. Astfel, daca ambii operanzi sint int, acesta va fi tipul rezultatului.

7. Expresii

Precedenta (prioritatea) operatorilor din expresii este aceasi cu ordinea subsectiunilor din aceasta sectiune, piritatea maxima fiind prima daca de exemplu expresiile referite ca operatori pentru +(&74) sint expresiile defiite in &&7. 1-7. 3. In fiecare subsectiune operatorii au aceeasi precedeta. Asociativitatea la stinga sau la dreapta este specificata in fiecare din operatorii din operatorii discutati in cadrul subsectiunii. Precedenta si asociativitatea tuturor operatorilor din expresii este rezumata in gramatica din &18.

Astfel ordinea de evaluare a expresiilor este nedefinita. In particular compilatorul se considera liber de a calcula subexpresiile de maniera pe care el o considera cea mai eficienta, chiar daca subexpresiile pot conduce la efecte secundare. Ordinea de aparatie a efectelor secundare este nespecificata. Expresiile care contin un comutator comutaiv si asociativ (∗, +, &, |, ^) pot fi rearanjate arbitrar, chiar in

prezenta parantezelor, pentru a se forta ordine particulara de evaluare o explicitare temporara devine necesara.

Prelucrarea depasirilor superioare si verficarea impartirilor la evaluarea expresiilor este dependenta de masina. Toate implementarile C-ului ignora depasirile superioare intregi; prelucrarea impartirii cu 0, si toate exceptiile VF variaza intre masini si sint uzual ajustabile printr-o functie de biblioteca.

7. 1 Expresii primare

Expresiile primare care contin., ->, subscrieri, si apeluri de functii se grupeaza de la stinga la dreapta.

primary-expresion:

identifier
constant
string
(expresion)
primary-expresion[expresion]
primary-expresion(expresion-list opt)
primary-lvalue. identifier
primary-expresion->identifier

expresion-list:
expresion
expresion-list, expresion

Un idetificator este o expresie primara, cu garantia ca a fost declarat asa cum se va discuta. Tipul sau este specificat la declarare. Daca tipul idetificatorului este "tablouri de..." atunci valoarea expresiei identificatorului sau este un pointer la primul obiect din tablou, iar tipul expresiei este "pointer la...". Mai mult, un idetificator de tablou (masiv) nu este o lvaloare. La fel un idetificator care e declarat "function returning..." (functie care returneaza), cind este folosit (cu exceptia cazului cind apare ca numele functiei la un apel) este convertit in "pointer la functia ce returneaza...".

O constanta este o expresie primara. Tipul ei poate fi int, long sau double, depinzind de forma sa. Constantele tip caracter au tipul int; constantele flotante sint double.

Un sir este o expresie primara. Tipul sau original (la origine) este "tablou de char"; urmind insa aceeasi regula data pentru identificatori, acesta este modificat in "pointeri catre char" si rezultatul este un pointer catre primul caracter al sirului (exista o exceptie pt unele initializari a se vedea &8.6.).

O exprsie in paranteze este expresie primara a carui tip si valoare sint identice cu ale expresiei fara paranteze.

Prezenta parantezelor nu are implicari daca expresia este o lvaloare. O expresie primara urmata de o expresie in paranteze drepte este o expresie primara. Sensul intuitiv este acela al unui indice. Obisnuit expresia primara are tipul "pointer catre...", expresia indicelui este int, iar tipul rezultatului este "..." (!!! N. T.). Expresia E1[E2] este identica (prin definitie) cu ∗((E1)+(E2)). Toate reguluile necesare intielegerii acesteinotatii sint continute in aceasta sectiune impreuna cu discutiile din &&7. 1, 7. 2 si 7. 4. asupra identificatorilor, ∗, si respectiv +; &14. 3 rezuma implicatiile.

Un apel de functie este expresie primara urmata de paranteze continid un posibil vid, o lista de expresii separate prin virgule constituind argumentele actuale ale functiei. EXpresia primara trebuie sa fie de tipul "functie care returneaza...", si rezultatul apelului functiei este de tipul "...". Cum se va indica, un identificator nevazut urmat de o paranteza stinga este contextual declarat ca reprezentind o functie care returneaza un intreg; deci in majoritatea cazurilor functiile car dau ca rezultat o valore intreaga nu trebuie declarate.

Orice argument actual de tipul float este convertit in double inainte de apel; oricare argument de tipul char sau short este convertit in int; si, ca deobiecei numele de tablouri sint convertit in pointere. Daca este necesara o conversie se va folosi... (N. T. cast) vezi &7. 2, 8. 7.
La pregatirea unui
apel la o functie se face o copie a fiecarui parametru actual adica toate trecerile de argumente in C se fac prin valoare. O functie poate modifica valorea parametrilor sai formali, dar aceste modificari nu acteaza valoarea parametrilor actuali !Mai mult se poate trece un pointer cu subintelesul functia poate modifica valoarea obiectului spre care pointerul indica. Un nume de tablou este o expresie pointer. Ordinea de evaluare a argumentelor este nedefinita de catre limbaj;a se nota ca difera de la compiler la compiler. Sint permise apeluri recursive la orice functie. O expresie primara urmata de un punct urmat de un identificator este o expresie. Prima expresie trebuie sa fie o lvaloare un signed a structurii sau o reuniune iar identificatorul trebuie sa fie un membru a structurii sau reuniunii. Rezultatul este o lvaloare referind membrul numit al structurii sau reuniunii.

O expresie primara urmata de o sageata (care se construieste din -si >) urmata de un identificator este oexpresie. Prima expresie trebuie sa fie un pointer la o structura sau la o reuniune si identificatorul trebuie sa numeasca un membru al acelei structuri sau reuniuni. Rezultatul este lvaloare care refereaza membrul numit al structurii sau reuniunii catre care pointerul expresiei puncteaza.

Astfel <expresia E1->MOS este identica cu (∗E1). MOS. Structurile

si reuniunile se discuta in &8. 5. Regulile date aici pentru utilizarea structurilor si reuniunilor nu nu sint stricte, de maniera care permite sa se scape de mecanismul tipurilor. Vezi &14.1.

7. 2. Operatori unari

Expresiile cu operatori unari se grupeaza de la dreapta la stinga.

unary-expression:

*expression
&lvalue
-expression
!expression
-expression
++lvalue
--lvalue
lvalue++
lvalue--
(type-name)expression
sizeof expression
sizeof(type-name)

Operatorul * inseamna indirect: expresia trebuie sa fie un pointer si rezultatul este o lvaloare care se refera la obiectul spre care pointeaza expresia. Daca tipul unei expresii este "pointer catre..." tipul rezultatului este "...".

Rezultatul operatiei & este un pointer catre obiectul referit prin lvaloare. Daca tipul lvalorii este " " tipul rezultatului este "pointer catre...".

Rezultatul unei operatii este negarea operandului. Se realizeaza conversiile aritmetice obisnuite. Negativul unei cantitati este calculat prin scaderea valorii sale din 2 la n , unde "n" este nr de biti pe care se reprezinta un int. Nu exista operatorul unar +.

Rezultatul operatiei de negare logica ! este 1 daca valorea operandului a fost 0, si 0 daca operandul este non-zero. Tipul rezultatului este int. Operatia este aplicabila oricarui tip de opernand aritmetic sau pointerilor.

Obiectul referit prin operandul lvaloare precedat de ++ este incrementat. Valoarea este noua valoare a operandului dar nu este o lvaloare. Expresia ++x este echivalenta cu x+=1. A se vedea adunarea (&7. 4) si operatorii de asignare (&7. 14) pentru informatii asupra conversiilor.

Operandul lvalorii cu prefixul -- este decrementat analog cu ++.

Cind postfixul ++ este aplicat unei lvalori rezultatul este valorea obiectului referit pr in lvaloarea. Dupa ce

rezultatul este notat obiectul este incrementat in aceeasi maniera ca pentru operatorul prefix++. Tipul rezultatului este acelasi cu tipul expresiei lvaloare.

Cind postfixul -- este aplicat unei lvalori rezultatul este valoarea obiectului referit prin lvaloare. Dupa notarea rezultatului obiectul este decrementat in maniera prefixului --. Tipul rezultatului este acelasi cu tipul expresiei lvaloare.

O expresie precedata de un nume de tip de data in paranteze cauzeaza conversia valorii expresiei in tipul de data numit constructia este denumita un cast (rol distributie, ...). Numele de tipuri de date sint descrise in &8. 7.

Operatorul sizeof produce marimea in bytes a opeandului sau. (Un byte nu este definit de limbaj decit in termenul de valore a lui sizeof). Dar in toate implementarile un byte este spatiul necesar pastrarii unui char. Aplicat unui tablou rezultatul este numarul total de bytes din tablou. Marimea se determina din declaratiile obiectelor in expresii. Expresia este semnatic o constanta intreaga si poate fi folosita oriunde este necesara o constanta. Se foloseste la comunicarile cu subprograme ca alocatori de memorie si sisteme de //0.

Operatorul sizeof poate fi aplicat unui nume de tip de data in paranteze. In acest caz va da marimea in bytes, a unui obiect de tipul indicat.

Constructia sizeof (type) este 1 deci expresia sizeof(type) -2 este aceeasi cu (sizeof(type)) -2.

7. 3. Operatori de inmultire

Operatorii de inmultire *, /, % grupeaza de la stinga la dreapta. Se realizeaza conversiile aritmetice uzuale.

multiplicative-expression:

expression*expression
expression/expression
expression%expression

Operatorul binar* indica inmultire. Operatorul * este asociativ si expresiile cu mai multe inmultiri de acelasi nivel pot fi rearanjate de compilator.

Operatorul /indica impartire. Cind se impart intregi pozitivi se face o trunchiere spre 0, dar forma trunchierii este dependenta de masina daca unul din operanzi este negativ. Pe toate masinile discutate in acest manual restul are semnul impartitorului. Todeauna este adevarat ca (a/b) *b+a%b este egal cu a (daca b e diferit de zero).

Operatorul % produce restul impartirii primei expresii la a doua.

Se realizeaza conversiile aritmetice uzuale. Operanzii trebuie sa nu fie float.

7. 4. Operatori aditivi

Operatorii aditivi + si- grupeaza de la stinga la dreapta. Se realizeaza conversiile aritmetice uzuale. Exista citeva posibilitati de tip aditionale pentru fiecare operator.

aditive-expression:

expression+expression
expression-expression

Rezultatul operatiei + este suma operanzilor. Un pointer catre un obiect dintr-un tablou si o valare de orice tip intreg pot fi adunate. Ultima este in toate cazurile convertita intr-un deplasament de adresa (N. T. address offset) prin inmultirea cu lungimea obiectului spre care indica pointerul. Rezultatul este un pointer de acelasi tip cu pointerul original si care indica un alt obiect din acelasi tablou, corespunzator deplasat fata de obiectul original. Deci daca P este un pointer catre un pointer dintr-un tablou (N. T. array) expresia P+1 este un pointer catre urmatorul element (obiect) din tablou.

Nu este permisa o alta forma de combinatie de tipuri pentru pointeri.

Operatorul + este asociativ si expresiile cu adunari multiple la acelasi nivel pot fi rearanjate de compiler. Rezultatul operatorului - este difernta dintre operanzi. Se realizeaza conversi aritmetice uzuale. In plus o valore de orice tip de intreg poate fi scazuta dintr-un pointer cu aceeasi conversie ca la adunare.

Daca doi pointeri catre obicte de acelasi tablou sint scazuti rezultatul este convertit (prin impartire cu lungimea obiectlui) intr-un int care reprezinta numarul de obicte care separa obictele indicate (pointed-to). Conversia poate da rezultate neasteptate, daca pointerii nu indica oectele aceluiasi tablou intrucit pointerii chiar spre obiecte de acelasi tip nu difera printr-un multiplu al lungimii obiectelor.

7. 5. Operatori de deplasare

Operatorii de deplasare << si >> grupeaza stinga la dreapta. Ambii fac conversiile uzuale asupra operanzilor fiecare trebuind sa fie un intreg. Apoi operandul din dreapta se cponverteste in int; tipul rezultatului este aceea al operandului din stinga. Rezultatul este nedefinit daca operandul din dreapta este negativ sau > sau = cu lungimea obiectului in biti.

shift-expression:

expression<<expression
expression>>expression

Valoarea lui E1<<E2 este E1 (interpretat ca forma de biti) deplasat la stinga; biti eliminati sint umpluti cu zero. Valoarea lui E1>>E2 este E1 deplasat dreapta cu E2 pozitii binare. Deplasarea la dreapta este garantat a fi logica (cu umplere de zerouri) daca E1 nu are semn; altfel poate fi (si este la PDP11) aritmetica (cu umplerea bitului semn).

7. 6. Operatori relationali

Operatorii relationali grupeaza stinga la dreapta dar acest fapt nu este foarte folositor; a<b<c nu inseamna ceea ce pare.

relational-expression:

expression<expression
expression>expression
expression<=expression
expression>=expression

Operatorii <, >, <=, >= dau 0 daca relatia este falsa si 1 daca este adevarata. Tipul rezultatului este int. Se fac conversiile uzuale. Doi pointeri se pot compara rezultatul depinde de locatiile relative in spatiul de adrese a obiectelor poinate. Comparatia pointerilor este portabila numai cind pointerii indica spre obiecte din acelasi tablou.

7. 7. Operatori de egalitate

equality-expression:

expression==expression
expression!=expression

Semnul == (egal cu) si !=(inegal cu) sint analogi operatorilor relationali dar nu au prioritate mai mica. (adica a<b==c<d este 1 daca a<b si c<d au valoare adevarat).

Un pointer poate fi comparat cu un intreg dar rezultatul va fi dependent de masina. Cu exceptia cazului cind intregul este constanta 0. Un pointer caruia i s-a asigurat valorea 0 se garanteaza ca nu va pointa spre nici un obiect si va apare a fi egal cu 0;in utilizarea conventionala un astfel de pointer este considerat nul.

7. 8. Operatorii si pe biti:

and expression:

expression&expresion

operatorul & este asociativ si expresiile continind & pot fi rearanjate Se realizeaza conversiile aritmetice uzuale; rezultatul este functia SI pe biti, a celor doi operanzi. Operatorul se aplica numai operanzilor intregi.

7. 9. Operatorul sau exclusiv pe biti

exclusive-or-expression:

expression^expression

Operatorul ^ este asociativ si expresiile care contin ^ pot fi rearanjate. Se realizeaza conversiile aritmetice uzuale; rezultatul este SAU exclusiv al celor doi operanzi la nivel de bit. Operatorul se aplica doar operanzilor intregi.

7. 10 Operatorul sau inclusiv

inclusiv-or-expression:

expression|expression

Operatorul | este asociativ expresiile care-l contin pot fi rearanjate. Se realizeaza conversii aritmetice uzuale; rezultatul este functie SAU inclusiv a operanzilor la nivel de bit. Operatorul se aplica doar operanzilor intregi.

7. 11. Operatorul si logic

logical-and-expression:

expression&&expression

Operatorul && grupeaza de la stinga la dreapta. El da 1 daca ambii operanzi sint diferiti de 0 altfel da 0. Nu ca &, && garanteaza o evaluare de la stinga la dreapta; al doilea operand nu este evaluat daca primul este egal cu 0.

Operanzii pot fi de tipuri diferite dar fiecare trebuie sa fie unul din tipurile fundamentale sau un pointer. Rezultatul este intodeauna int.

7. 12 Operatorul sau logic

logical-or-expression:

expression||expression

Operatorul || opereaza de la stinga la dreapta. El da 1 daca unul din operanzi este diferit de 0 si altfel da 0. Nu ca |, || face evaluare la stinga -> dreapta, al doilea operand nu mai este evaluat daca primul este diferit de 0.

Operanzii pot fi de tipuri diferite dar de tipurile de baza (fundamentale) sau pointeri. Rezultatul este totdeauna int.

7. 13 Operator conditional

conditional-expression:

expression?expression:expression

opereaza de la dreaptspre stinga. Prima expresie este evaluata si daca este non zero rezultatul este valoarea celei de a doua expresii. Altfel este al celei dea treia expresii. Daca este posibil conversiile artimetice uzuale se fac pentru a aduce a doua si a treia expresie la un tip comun; se poate ca unul sa fie pointer si celalat zero, si rezultatul are tipul pointerului. Numai expresiile a doua si a treia sint evaluate.

7. 14. Operatori de atribuire

Operatorii de atribuire opereaza de la dreapta spre stinga. Toate solicita o lvaloare ca opernd stinga, iar tipul unei expesii de asigurare este acela al operandului din stinga. Valoarea este valoarea stocata in operandul din stinga dupa ce asigurarea s-a facut. Cele doua parti ale unui operator de asigurare compus sint elemente sintactice (atomi)separati.

assignement-expression:

lvalue=expression
lvalue+=expression
lvalue-=expression
lvalue*=expression
lvalue/=expression
lvalue%=expression
lvalue>>=expression
lvalue<<=expression
lvalue&=expression
lvalue^=expression
lvalue|=expression

La asigurarea simpla cu = valoarea expresiei inlocuieste pe cea a obiectului referit prin lvaloare. Daca ambii operanzi sint de tip aritmetic operandul drept se converteste in tipul celui sting inainte de asigurare.

Comportarea unei expresii dforma E1op=E2 poate fi inferred luind ca echivalent al ei E1=E1op(E2); oricum E1 este evaluata doar odata. La += si -= operandul sting poate fi un poninter in care opernadul din dreapta (intreg) este convertit dupa cum se explica in &7. 4; toti opernazii din dreapta si toti opernazii stingi non pointeri trebuie sa fie de tip aritmetic.

Compilatoarele permit ca un pointer sa fie asignat unui intreg, un intreg sa fie asignat unui pointer si un pointer sa fie asignat

unui pointer de alt tip. Asignarea este operatia de copiere pura fara conversii. Aceasta utilizare este neportabila si poate produce pointeri care pot cauza exceptii de adresare la utilizare. Este garantata asignarea constantei zero la un pointer aceasta producind un pointer nul distinct de orice pointer spre orice obiect.

7. 15 Operatorul virgula

comunn-expression:

expression, expression

O pereche de expresii separate prin virgula este evaluata de la stinga la dreapta si valorea expresiei din stinga este declasata. Tipul si valoarea rezultatului sint tipul si valorea operandului din dreapta. Operatorul lucreaza de la stinga la dreapta. In contextul unde virgulei i se da un inteles special, de exemplu intr-o lista de argumente pt o fc. (&7. 1) sau liste de initializare (&8. 6), operatorul - virgula descrisa aici poate apare doar in paranteze, de exemplu:

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

are trei argumente, al doilea avind valoarea 5.

CAPITOLUL 8. Declaratii

Declaratiile sint folosite pt a specifica interpretarea pe care C o da fiecarui identificator; nu fac in mod necesar si rezervarea de memorie pt respectivul identificator. Declaratiile au forma:

declaration:

decl-specifiers declarator-list opt;

Declaratorii din lista de declaratori contin identificatorii de declarat. Specificatorii de declaratii constituie o secventa de specificatori de tip si clase de memorare:

decl-specifiers:

type-specifier decl-specifiers opt
sc-specifier decl-specifiers opt

lista trebuie sa fie unica (self consistent) in exemplul descris mai jos. (N.T. -sc=storage class)

8. 1. Specificatori de clase de memorare

Acestia sint:

sc-specifier:

auto
static
extern
register
typedef

specificatorul "typedef" nu rezerva memorie si este numit "specificator" de clasa de memorie " doar din motive sintactice; se diacuta la &8. 8 sensurile diverselor claselor de memorare s-au discutat in &4.

Declaratiile auto, static si register servesc ca definitii pt ca cauzeaza o rezervare de memorie. La extern trebuie sa existe o definire externa (vezi &10) pt identificatori dati undeva in afara functiei unde ei au fost declarati.

Declaratia register este mai des vazuta ca o declaratie auto impreuna cu o alertare a compilatorului ca aceste varaibile vor fi utilizate des. Doar citeva declaratii de acest tip sint efective. Mai mult doar varaibilele de anumite tipuri vor fi stocate in registre; la PDP11 acestea sint int, char si pointeri. O alta restrictie se refera la variabilele registru. Operatorul & (care lucreaza cu adrese) nu li se poate aplica programele create pot deveni mai mici mai rapide dintr-o utilizare de declaratii register folosite corespunzator, dar inbunatatiri viitoare lae generarii de cod pot sa le faca necesare.

Cel putin un specificator de tip memorare poate fi dat intr-o declarare. Daca lipseste el este considerat auto in interiorul functiei, si extern in afara ei. Exceptie: functiile nu sint niciodata automatic.

8. 2. Specificatori de tip

type-specifier:

char
short
int
long
unsigned
float
double
struct-or-union-specifier
typedef-name.

Cuvintele long, short si unsigned pot fi privite ca si adjective; urmatoarele combinatii sint aceptabile.

```
short int
long int
unsigned int
long float
```

Sensul ultimei constructii este acelasi cu double. Altfel cel mult un specificator de tip poate fi dat intr-o declaratie. Daca specificatorul de tip lipseste el este considerat int.

Specificatorii de structuri si reuniuni vor fi discutati in &8. 5, declaratiile cu typedef sint discutate in &8. 8.

8. 3. Declaratori

Lista de declaratori care apare intr-o declaratie este o secventa separata prin virgula de declaratori, fiecare trebuind sa aiba o initializare.

```
declarator-list:

        init-declarator
        init-declarator, declarator-list

int-declarator:

        declarator initialiser opt.
```

Initializatorii sint discutati in &8. 6. Specificatorii dintr-o declarattie indica tipuri si clase de memorare a obiectelor la care se refera declaratorii. Declaratorii au sintaxa:

```
declarator:

        identifier
        (declarator)
        *declarator
        declarator()
        declarator[constant-expression opt]
```

Gruparea este ca la expresii.

8. 4. Sensul declarartorilor

Fiecare declarator este o afirmatie de genul ca atunci cind o constructie de aceeasi forma ca si declaratorul apare intr-o expresie, va produce un obiect de tipul si clasa indicata. Fiecare declarator contine doar un identificator;acest identificator se declara.

Daca doar un identificator apare ca declarator, atunci el va avea

tipul indicat de specificatorul de la inceputul declaratiei.

Un declarator in paranteze este identic cu un declarator fara atribute, dar amestecarea declaratorilor complecsi poate fi alterat prin paranteze. Vezi exemplele de declaratie.

Acum sa ne imaginam o declaratie

T D1

unde T este un specificator de tip (ca int) si D1 este un declarator. Sa presupunem ca aceasta declaratie face ca identificatorul (N. T. din declarator) sa aibe tipul "..."unde " " este gol daca D1 este doar un identificator pur si sipmplu (asa ca tipul lui X din "int X" este doar int). Daca D1 are forma:

∗D

tipul identificatorului continut este "...pointer la T".

Daca D1 are forma

D()

atunci identificatorul continut are tipul "... functie ce returneaza T. "

Daca D1 are forma D[constant-expression]... sau D[], atunci identificatorul continut are tipul "...tablou de T". In primul caz expresia constantei este o expresie a carei valoare este determinabila in momentul compilarii si care are tipul int (expresiile constanta sint definite precis in &15). Cind mai multe specificatii de tsblou sint adiacente se creaza un tablou multidimensional; expresiile constanta care specifica limitele tablourilor pot sa lipseasca doar pentru primul membru al secventei. Aceasta forama este folositoare daca tabloul e extern si definitia actuala, care aloca spatiu, este totusi data. Prima expresie contanta poate fi omisa daca declaratorul este urmat de o initializare. In acest caz marimea se calculeaza din nr elemente precizate.

Un tablou poate fi constituit din unul din tipurile de baza prin pointeri dintr-o structura sau reuniune, sau din alt tablou (pt a genera tablouri cu dimensiuni multiple).

Nu toate posibilitatile date de sintaxa sint permise. Restrictiile sint urmatoarele: functiile nu pot returna tablouri, structuri sau reuniuni dar pot returna pointeri spre asa ce va; nu exista tablouri de functii dar pot exista tablouri de pointeri spre functii. O structura sau o reuniune nu poate contine o functie, dar poate contine un pointer la o functie. Exemple:

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

decalara

un intreg i
un pointer ip la un intreg
o functie f care returneza un intreg
o functie fip care returneaza un pointer la un intreg
un pointer pfi la o functie care reurneaza un intreg

Se compara ultimele. *fip() este *(fip()) declaratia sugereaza si aceiasi constructie intr-o expresie necesita apelul functiei fip si apoi indirectarea spre rezultat (pointer) se produce un intreg. (*pfi)() in declaratie parantezele sint necesare ca si intr-o expresie ca sa indice ca indirectarea prin pointeri la o functie produce o functie care este apoi chemata; va returna un intreg.

Alt exemplu:

```
float fa[17], *afp[17]
```

declara un tablou de flotante si un tablou de pointere spre numere flotante.

In final:

```
static int x3d[3][5][7];
```

declara un tablou de intregi, static, cu dimensiunile 3x5x7. In detaliu X3d este un tablou de 3 articole; fiecare articol este un tablou de 5 tablouri; si fiecare din ultimul este un tablou de 7 intregi. Oricare din expresiile x3d, x3d[i], x3d[i][j], x3d[i][j][k] poate apare intr-o expresie. Primele trei sint de tip "tablou", ultima are tipul int.

8. 5. Declaratorii de structuri si reuniuni

O structura este un obiect constind dintr-o secventa de memorii cu nume. Fiecare membru poate avea orice tip. O reuniune este un obiect care, la un moment dat, poate contine pe oricare din mai multi membri.

Specificatorii de structuri si reuniuni au aceeasi forma:

struct-or-union-specifier:

struct-or-union{struct-decl-list}
struct-or-union identifier{struct-decl-lisSI
struct-or union identifier

struct-or-union

struct
union

lista struct-decl-list este o secventa de declarare pentru membrii structurii sau uniunii.

struct-decl-list:

struct-declaration
struct-declaration struct-decl-list

struct-declaration:

type-specifier struct-declarator-list;

struct-declarator-list:

struct-declarator
struct-declarator, struct-declarator-list

in cazul uzual, un struct-declarator este numai un declarator pentru un membru al unei structuri sau uniuni. Un membru din structura poate consta dintr-un numar de biti. Un astfel de membru se numeste cimp (NT field) lungimea sa este despartita de numele cimpului prin 2 puncte:

structt-declarator:

declarator
declarator:constant-exprression
:constant-expression

Intr-o structura obiectele declarate au adrese care cresc pe masura ce declararea lor se citeste de la stinga la dreapta. Fiecare membru al unei structuri (care nu e cimp) incepe la limite de adrese corespunzatoare tipului sau; dar pot fi gauri fara nume intr-o structtura. Mmebrii de tip cimmp sint impachetati in intregi (NT repreentare pe masina intregilor) ei nu stradle cuvinte. Un cimp care nu incape intr-un cuvint de memorie se continua pe urmatorul. Un cimp nu trebuie sa depaseasca un cuvint (NT ca lungime ?) Cimpurile sint asigurate de la dreapta la stinga la PDP si de la stinga la dreapta pe alte masini.

Un struct-declarator fara declarator continind doua puncte si o lungime indica un cimp fara nume folositor pentru conformate cu formatele externe, care se impun. Ca un caz special, un cimp cu lungime0 specifica eliminarea urmatorului cimp la granita de cuvint. "Urmatorul cimp " este un cimp si nu un membru ordinar al structurii pentru ca pentru acest caz alinierea se face automat.

Limbajul nu face restrictie asupra tipurilor de lucruri care se declara ca si cimpuri, dar implementarile nu suporta decit cimpuri de tip intreg. Chiar si cimpurile int sint considerate ca fara semn. La PDP cimpurile nu au semn si au numai valori intregi. In toate implementarile nu exissta tablouri de cimpuri,

astfel ca operatorul & nu se poate aplica, neexistind pointeri spre cimpuri.

O uniune poate fi gindita ca o structura a caror membrii incep cu deplaasamentul 0 si de dimensiune suficienta pentru a contine pe oricare din membrii. La un mmoment dat doar un membru este stocat.

Un specificator de structura sau uniune de forma a 2-a, adica

```
struct identifier{struct-decl-listt}
union identifier{struct-decl-list}
```

declara identificatorul ca eticheta (marcaj) de structura (sau uniune) a structturii specificate de lista. O declaratie de a treia forma

```
struct identifier
union identifier
```

Etichetele de structura permit definirea de structuri auto-referibile ele permit ca partea lunga a unei structuri sa fie data odata si folosita de mai multe ori. Este ilegal a declara o structura sau uniune care face apel la ea insasi, dar o structura sau uniune poate contine un pointer la un apel spre ea insasi.

Numele membrilor si etichetele sint la fel ca pentru variabilele ordinare dar sa fie distincte mutual.

Doua structuri pot folosi in comun o secventa initiala de membri; adica acelasi membru apare in doua structuri diferite daca au acelasi tip in ambele si daca membrii anteriori sint acesasi in ambele(In momentul de fata, compilatorul verfica numai daca in unul din doua structuri diferite are acelasi tip si deplasament in ambele, dar daca membrii precedenti difera, constructtia este neportabila).

Exemplu simplu de declarare de structura:

```
struct  tnode{
        char  tword[20];
        int  count;
        struct  tnode *left;
        struct tnode *right;
                };
```

contine
-un tablou de 20 caractere
-un intreg
-2 pointeri catre structuri similare

Odata aceasta declaratie facuta, declaratia

```
struct tnode  s, *sp;
```

declara S ca fiind o structura de tipul dat si sp un pointer la o structura de tipul dat.

Cu aceste declaratii expresia:

```
sp->count
```

Se refera la cimpul count spre care pointeaza sp;

```
s. left
```

Se refera la pointerul subarborelui stinga

```
s. right->tword[0]
```

Se refera la primul caracter al subarborelui tword din arborelui drept al lui S.

8. 6. Initializare

Un declarator poate specifica o valoare initiala pt identificatorul declarat. Initializatorul este precedat de = si consta dintr-o expresie sau o lista de valori in acolade.

```
initializer:

        =expression
        ={initializer-list}
        ={initializer-list, }

initializer-list:

        expression
        initializer-list, initializer-list
        {initializer-list}
```

Toate expresiile pentru initializarea unei variabile statice sau externe trebuie sa fie expresii de constanta, descrise in &15, sau expresii care se reduc la adresa variabilei declarate anterior, cu posibilitatea de a fi deplasate printr-o constanta. Variabila automatic si register pot fi initializate cu expresii arbitrare care pot contine constante, variabile declarate anterior si functii.

Variabilele statice si externe neinitializate la start primesc valoarea zero. Cele automatic si register au continut nespecificat (probabil "gunoi").

Cind intr-un initializator se aplica unui scalar (pointer sau un obiect de tip aritmetic) el consta din expresie singulara posibil in paranteze. Valoarea initiala a obiectului se obtine din expresie; aceleasi conversii ca pentru atribuire se folosesc.

Cind o variabila declarata este un agregat (structura sau tablou) atunci initializatorul consta dintr-o lista de initailizatori separati prin virgule in parantze (acolade) scri si in dinea in care cresc indicii sau in orinea membrilor. Daca numarul de initializatori este mai mic decit al membrilor se impune restul cu 0. Nu se initializeaza uniuni sau agregate de tip automatic.

Acoladele se pot elimina. Daca un initilizator incepe cu acolade stinga atunci va urma o lista de initializare cu initializatori despartiti stinga prin virgule pentru membrii agregatului; este eronat sa existe mai multi initializatori decit membri. Daca initializatorul nu incepe cu acolada stinga atunci se iau din lista numarul necesar de elemente pentru agregatul curent, restul lasati la dispozitia agrgatului urmator.

De exemplu:

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

x este declarat si initializat cu un tablou cu o dimensiune care are trei membri.

```
 float y[4][3]={
        {1, 3, 5},  initializeaza prima linie adica y(0)
        {2, 4, 6},  care este y[0][0], y[0][1] si y[0, 2]
        {3, 5, 7},  apoi se initializeaza liniile y[1] si y[2]
 };              y[3] se inittializeaza cu 0.
```

Acelasi efect se obtine cu:

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

Initializarea pentru y incepe cu acolada stinga, dar pentru y[0] nu, de ci se folosesc trei elemente din lista. Urmatorele trei sint luate pentru y[1] si y[2]. Deci:

```
        float y[4][3]={

                {1}, {2}, {3}, {4}
        };
```

initializeaza prima coloana a lui y (privit ca un tablou cu doua dimensiuni) si lasa restul pe zero.

Si:

```
        char msg[]="syntax error on line %s/n";
```

arata un tablou de caractere a carui membri sint initializati cu un sir.

8. 7. Nume de tipuri

In doua contexte (pt a se specifica explicit o expresie cu ajutorul unei distributii (N. T. cast) si ca si argument pt size of) este de dorit a se specifica numele unui tip de data. Aceasta se realizeaza folosind "nume de tip" care este in esenta o declaratie pt un obiect de tipul respectiv care omite numele obiectului.

type-name:

type-specifier abstract-declarator.

abstract-declarator:

empty
(abstract-declarator)
*abstract-declarator
abstract-declarator()
abstract-declarator[constant-expression opt]

Pentru a se evita ambiguitatea in constructia

(abstract-declarator),

abstract-declaratorul nu trebuie sa fie vid. Cu aceasta restrictie se poate identifica unic locatia din abstract-declarator unde va apare declaratorul daca constructia a fost un declarator intr-o declaratie. Tipul de nume este acelasi cu tipul unui identificator ipotetic de exemplu:

```
int     tip->intreg
int*     pointer la (catre) intreg
int*[3]  tablou de 3 pointeri la intregi
int(*)[3] pointer la un tablou format din 3 intregi
int*() functie care returneaza un pointer la un intreg
int(*)() pointer la o functie care returneaza un intreg
```

8. 8. Typedef

Declaratiile a caror "clasa de memorare" este typedef nu definesc o memorie ci definesc identificatori care vor fi utilizati ulterior ca si cum ar fi cuvinte cheie de tip denumind tipuri fundamentale sau derivate.

typedef-name:

identifier

Rorul unei declaratii continind typedef este ca orice identificator care apare ca parte a oricarui declarator din interiorul (declaratiei) devine sintactic echivalenta cu cuvintele cheie de tip numind cuvintele de tip asociat identificatorului in

modul desis in &8..De exemplu, dupa:

```
typedef int MILES, *KLICKSP;
typedef struct{double, int;}complex;
```

Constructiile

```
MILES distance;
extern KLICKSP metricp;
complex 2, *zp;
```

sint declaratii legale; tipul lui distance este int, al lui metricp este "pointer la un int", si al lui z este structura specificata; zp este pointer la respectiva structura.

typedef nu introduce tipuri noi, numai sinonime pt tipuri care se pot specifica si altfel. In exemplul distance este considerat a avea exact acelasi tip ca orice alt obiect int.

9. Enunturi

Cu exceptiile ce vor fi indicate, enunturile se executa in secventa.

9. 1. Enunt expresie

Multe enunturi sint enunturi expresie, care au forma:

```
expression;
```

In general enunturile expresie sint asignari sau apeluri de functie.

9. 2. Enuntul compus sau block

Se prevede enuntul compus intrucit se pot folosi mai multe enunturi acolo unde este asteptat doar unul:

```
compound-statement;

        {declaration-list opt statement-list opt}

declaration-list:

        declaration
        declaration, declaration-list

statement-list
```

```
statement
statement statement-list
```

Daca unul din identificatorii din lista de declaratii a fost declarat anterior, declaratia externa este decazuta pentru durata unui bloc, dupa care isi epuizeaza forta.

Orice initializare de variabile auto sau register se realizeaza de fiecare data cind se intra in bloc la virful sau. Este posibil(dar este o practica rea) de a face transferul in bloc; in acest caz nu se face initializarea.

Initializarea variabilelor static se face doar odata, la inceputul executiei programului. In bloc, declaratiile extern nu rezerva memorie astfel ca initializarea nu este permisa.

9. 3. Enunturi conditionale

Sint 2 forme de enunturi conditionale:

```
if(expresie) statement
if(expresie) statement else statement
```

In ambele cazuri se evalueaza expresia si daca nu sint zero se executa primul enunt. In al 2-lea caz se executa a 2-a instructie daca prima este egala cu zero. Ambiguitatea lui "else" este rezolvata prin conectarea unui "else" cu ultimul "else-less if" intilnit.

9. 4. Instructii while

Forma: while(expression)statement

Instructia din while este executat repetat atita timp cit valoarea expreie ramine diferita de zero. Testul se face inainte de executia instructiei.

9. 5. Instructia do

Are forma

```
do statement while(expression);
```

Instructia este executata repetat pina cind valorea expresiei devine zero. Testul se face dupa fiecare executie a instructiei.

9. 6. Instructia for

Are forma:

for(expression-1opt;expression-2opt;expression-3opt)statement

Este echivalenta cu:

```
expression-1;
while(expression-2){
        statement
        expression-3;
}
```

Deci prima expresie specifica initializarea buclei; a doua specifica un test, facut inaintea fiecarei iteratii, astfel ca din bucla se iese cind expresia devine zero; a 3-a expresie specifica o incrementare care este realizata dupa fiecare iteratie.

Oricare sau toate expresiile pot lipsi. Daca lipseste a doua instructie while implicata devine echivalenta cu while(1); celelalte expresii vor lipsi din constructia data.

9. 7. Instructia switch

Instructia switch face ca controlul sa fie transferat la una din mai multe instructii functie de valoarea expresiei. Are forma:

```
switch(expression)statement
```

Se executa conversiile necesare, dar rezultatul trebuie sa fie int. Instructia este compusa. Orice instructie din bloc pate fi etichetata cu un prefix tip case

```
case constant-expression
```

unde expresia de constanta va fi un intreg(int). Este interzisa aparitia a doua constante pentru case in aceasi instructie switch cu aceeasi valoare. Constantele se definesc precis in &4. 5.

Poate exista un prefix de instructie de forma

```
default:
```

Cind se executa instructia switch, expresia se evalueaza si se compara cu constantele case. Daca una este egala cu valorea expresiei, controlul se va da la instructia urmind prefixul gasit.

Daca nu exista instructii cu case-ul cautat, dar exista prefixul defaault, controlul se da la instructia prefixata. In lipsa prefixului default nu se executa nici una din instructiuni.

case si default, in sine, nu altereaza mersul programului.

Iesirea din switch se face cu break (vezi &9. 8) In general instructia al carui subiect este switch este un bloc. Declaratii pot apare la inceputul instructiei, dar initializarea variabilelor automatic si register sint inefective.

9. 8. Instructia break

Are forma: break; si face sa se termine ciclul cel mai intern while, do, for sau switch. Controlul trece la instructia care urmeaza dupa instrucctia de terminare.

9. 9 Instructia continue

Are forma:

```
continue;
```

si face sa se treaca la continuarea in bucla a celui mai intern while, do sau for; adica se sare la sfirsitul buclei. Mai precis, in fiecare din instructiile

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

O instrcutie continue este echivalenta cu goto contin(Dupa contin: o instructie goala)

9. 10. Instructia return

O functie revine la apelant cu instructia return care are formele:

```
return;
return expression;
```

In primul caz valoarea returnata nu e definita. In al doilea caz valoarea expresiei este returnata apelantului. Daca se cere, expresia este cnvertita, ca la asignare, in timpul functiei in care apare. Ocolirea finalului unei functii este echivalenta cu nereturnarea de valoare la apelant.

9. 11. Instructia goto

Controlul se poate transfera neconditionat cu ajutorul instructiei:

```
goto identifier;
```

Identificatorul trebuie sa fie o eticheta (vezi 9. 12) din functia

curenta.

9. 12 Instructii etichetate

Oricare instructie poate fi precedata de un prefix eticheta de forma

identifier:

care serveste pentru declararea identificatorului ca si eticheta. Unica utilizare a etichetei este de tinta a unui goto. Bataia unei etichete este functia curenta, excluzind sub blocurile in care acelasi identificator poate fi redeclarat. Vezi &11.

9. 13 Instructia nula

Are forma

;

Este folosita pentru ca poate purta o eticheta chiar inainte de }(N. T. acolada finala) a unei instructii compuse sau servind ca si corp de instructii nul unei instructii de buclare gen while.

10. Definitii externe

Un program C consta dintr-o secventa de definitii externe. O definitie externa declara un identificator ca avind clasa de memorare extern (in lipsa specificatorului sau static, si un tip specificatn tip specificat. Specificatorul de tip (vezi 8. 2) poate fi vid, in care caz tipul va fi luat ca si int. Intinderea unei definitii externe persista pina la sfirsitul fisierului in care a fost declarata asa cum efectul unei declaratii persista pina la sfirsitul unui bloc. Sintaxa defintiilor extern este aceeasi ca a tuturor declaratorilor, cu exceptia ca numai la acest nivel poate fi dat codul pentru functii.

10. 1. definitii defunctie externe

Definitiile de functii au forma:

function-definition:

decl-specifiersopt function-declarator function-body

Singurii specificatori de clasa de memorare permisa in cadrul specificatorilor de declaratii sint extern sau static; a se vedea &11. 2 pentru distinctia dintre ele. Un declarator de

functie este similar cu un declarator pentru "functie care returneaza..." cu exceptia ca el listeaza parametrii formali ai functiei de definit.

```
        function-declarator:

                declarator(parameter-listopt)

        parameter-list:

                identifier
                identifier, parameter-list
```

Corpul functiei are forma:

```
                function-body:

                        declaration-list compound-statement
```

Identificatorii din lista de parametrii, si numai acesti identificatori, pot fi declarati in lista de declaratii. Orice identificator al carui tip nu este dat se considera a fi de tip int. Singurul tip de clasa de memorare care poate fi specificata este register; daca aceasta e specificata, parametrul actual corespunzator va fi copiat, daca este posibil, intr- un registru in codul codului functiei. Un exemplu simplu de definitie completa de functie este:

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

Aici int este specificator de tip;max(a, b, c) este declaratorul de functie int a, b, c; este lista de declaratii pentru parametrii formali; {...} este blocul care da codul pentru instructie.

"C"converteste parametrii actuali de tip float in double, astfel ca parametrii formali declarati float isi vor avea declaratiile ajustate pentru a se citi double. Astfel, daca o referinta la un tablou in orice context (in particular cu un parametru actual)este cnsiderata a avea sensul unui pointer la primul element al unui tablou, declaratiileparametrilor formali de genul "tablou de..." sint ajustate in "pointeri catre...". In final, intrucit structurile, uniunile si functiile nu pot fi trecute unei functii, este fara utilizare declararea ca parametri formali a unei structuri, uniuni sau functii (pointerii la astfel de obiecte sint permisi).

10. 2. Definitii de date externe

O definitie de data externa are forma:

data definition:

declaration

Clasa de memorare a unei astfel de date poate fi extern (in lipsa) sau static dar nu auto sau register.

11. Reguli despre domenii

Un program in C poate sa nu fie compilat tot deodata: textul sursa al progarmului poate fi pastrat in mai multe fisiere si rutine precompilate pot fi incaracte din biblioteci. Comunicatiile intre functiile unui program pot fi apeluri explicite sau utilitare de date externe.

Exista 2 feluri: primul, ce ar putea fi numit obiectivul lexical al unui identificator, care este in esenta regiunea unui program in timpul caruia el poate fi folosit fara a apare eroarea de "identificator nedefinit"; si al 2-lea obiectivul asociat cu identificatori externi care se caracterizeaza prin regula ca referinta la acelasi identificator extern sint referinte la acelasi obiect.

11. 1 Domeniu lexical

Obiectivul lexical al identificatorilor declarati in definitiile externe se intinde de la definitie pina la sfirsitul fisierului sursa in care apare. Intinderea lexicala a indentificatorilor care sint parametri formali persista in functia cu care sint asociati. Scopul lexical al identificatorilor declarati la inceput de bloc tine pina la sfirsitul blocului. Intentia lexicala a etichetelor este in functia in care apar.

Intrucit toate referintele la acelasi identificator extern se refera la acelasi obiect(vezi 11. 2)compilatorul verifica toate declaratiile aceluiasi identificator extern pt compatibilitate; de fapt puterea lor se intinde asupra intregului fisier pe care apar.

In toate cazurile, daca un identificator este explicit declarat la inceputul unui bloc, incluzind blocul care constitue o functie, orice declaratie al acelui identificator in afara blocului este suspendata pina la sfirsitul bocului.

De reamintit (vezi 8. 5) ca identificatorii asociati cu varaibile ordinare pe de o parte si acei asociati cu membrii ai unor structuri sau reuniuni pe de alta parte, frmeaza doua

clase disjuncte fara conflict intre ele. Membrii (de reuniuni sau structuri n. t.) si etichetele urmeaza aceleasi reguli obiectuale ca si identificatorii; numele declarate cu typedef sint de aceasi clasa cu identificatorii ordinari. Ei pot fi redeclarati in blocurile interne, dar un tip explicit se va da in declaratia interioara.

```
typedef float   distance;
...
{
        auto int distance;
```

Declaratia int trebuie sa fie prezenta in a doua declaratie; in caz contrar va fi considerata fara declaratori si de tipul distance.

11. 2. Domeniul externilor

Daca o functie se refera la un identificator ca fiind extern atunci undeva intre fisierele si bibliotecile din care se constituie programul complet trebuie sa fie o declaratie externa pt acel identificator ca fiind externe, atunci undeva intre fisierele bibliotecilor din care se constituie programul complet trebuie sa fie o declaratie externa pt acel identificator. Toate functiile dintr-un program care se refera la acelasi identificator extern se refera la acelasi obiect, deci trebuie grija ca tipul si dimensiunea specificate in definitie sa fie compatibile cu acelea specificate de fiecare functie care face referire la data respectiva.

Aparitia cuvintului cheie extern intr-o definitie externa indica ca memoria pt identificatorii care se declara va fi alocata in alt fisier. DEci intr-un program multifisier, o definitie la date externe fara specificatorul extern trebuie sa apara exact intr-unul din fisiere. Oricare alte fisiere care doresc sa dea o definitie externa pt identificator trebuie sa includa extern in definitie. Identificatorul poate fi initializat numai in declaratia unde memoria este alocata.

Identificatorii declarati static la nivelul superior din definitiile externe nu sint vizibile in alte fisiere. Functiile trebuie declarate static.

12. linii de comanda pt compilator

Compilatorul C contine un preprocesor capabil sa faca macrosubstitutii compilari conditionale si includeri de fisiere numite. Liniile care incep cu # comunica cu acest preprocesor. Aceste linii au sintaxa independenta de restul limbajului; ele pot apare oriunde si au efecte cu remanenta pina la sfirsitul fisierului sursa.

12. 1. Inlocuire de atomi

(N. T. atom element sintactic independent. SEnsul nu poate fi precizat la data traducerii)

O linie de comanda de forma:

```
#define identifier token-string
```

(de notat : fara punct virgula final) face ca preprocesorul sa inlocuiasca identificatorul cu sirul dat de atomi. O linie de forma

```
#define identifier (identifier, ..., identifier)token-string
```

unde nu exista spatiu intre primul identificator si (, este o macrodefinitie cu argumente. Exprimarile ce urmeaza primului identificator urmat de (sint inlocuite de elementele sirului specificat in definitie. Oricare aparitie a unui identificator mentionat in lista de parametri formali ai definitiei este inlocuit de sirul de elemente (atomi) de la apel. Argumentele din apel sint siruri de atomi separati prin virgule; dar virgulele din sirurile intre apostroafe, sau care sint protejate prin paranteze nu separa argumente. Numarul de argumente formale si actuale trebuie sa fie aceleasi. Textul dintr-un sir sau o constanta caracter nu se inlocuieste.

In ambele forme sirul cu care se face inlocuirea este riscant pentru identificatorii mai definiti. O definitie lunga poate fi continuata si pe o alta linie scriind \ la sfirsitul liniei de continuat.

Facilitatea este valoroasa pt definirea "constantelor manifeste "

```
#define TABSIZE 100
int table[TABSIZE];
```

o linie de comanda de forma:

```
#undef identifier
```

face ca defintia pt procesor a identificatorului sa fie notata.

12. 2. Includeri de fisiere

O linie de forma:

```
#include "filename"
```

face ca sa inlocuiasca linia cu intregul continut al fisierului

"filename ". Numele este cauata mai intii in directorul fisierului sura original si apoi in alte locuri (biblioteci N. T) standard. Alternativa este

#include <filename>

cauta numai in locurile standard nu in directorul din care face parte fisierul sursa. #include poate fi imbricat.

12. 3. Compilarea conditionala

O linie de comanda de forma:

#if constant-expression

verifica daca constanta (vezi &15) este nonzero. O linie de forma:

#if def identifier

verifica daca identificatorul este definit in procesor, deci daca a fost sau nu subiectul unei linii #define.

O linie de forma:

#ifndef identifier

verifica daca identificatorul nu e definit in procesor. Toate cele 3 forme sint urmate de un numar de linii arbitrare si pot contine o linie de comanda.

#else

si apoi o linie de comanda

#endif

Daca conditia este adevarata atunci liniile intre #else si #end if sint ignorate. Daca conditia este falsa atunci liniile intre test si #else sau in lipsa lui #else, #end if sint ignorate.

12. 4. Controlul liniei

Pentru beneficiul altor preprocesoare care genereaza progarmul C

#line constnat identifier

face ca numarul liniei curente sa devina egal cu valoarea constantei iar fisierul de intrare curent este denumit de identificator. Daca identificatorul este absent numele fisierului memorat nu se schimba.

13. Declaratii implicite

Nu este todeauna necesar sa se specifice atit clasa de memorare cit si tipul identificatorului in declaratie. Clasa de memoare se extrage din contextul definitiilor externe si din declaratiile parametrilor formali si ai membrilor de structuri. Intr-o declaratie din interiorul unei functii daca nu se da clasa de memorare sau tipul, identificatorul se presupune de tipul int; daca se indica tipul dar nu se indica clasa de memorare identificatorul se presupune a fi auto. O exceptie a ultimei reguli se face pt functii, functiile de tip auto neavind sens (C nu e capabil sa compileze codul intr-o stiva); daca tipul identificatorului este " functia care returneaza..." el este implicit declarat extern. Intr-o expresie un identificator urmat de (si nedeclarat inca este contextual declarat ca fiind "functie care returneaza un int ".

14. Revedere asupra tipurilor

Aceasta sectiune rezuma operatiile care pot fi realizate asupra obiectelor de diverse tipuri.

14. 1. Structuri si uniuni

Asupra unei structuri sau marimi se pot face doua lucruri: numirea (N. T. apelarea) unia din membrii sai (cu ajutorul operatorului .) sau obtinerea adresei sale (cu ajutorul operatorului &). Alte operatii, cum ar fi asigurarea valorii sale sau unei valori sau transmiterea lui ca si parametru conduc la un mesaj de eroare. In viitor se preconizeaza ca aceste operatii dar nenecesar altele sa fie permise.

&7. 1 arata ca intr- referinta directa sau indirecta la o structura (cu ajutorul lui -sau ->) numele din dreapta trebuie sa fie membru al structurii numite sau sa fie indicat (pointat)de expresia din stinga. Ca o iesire din regulile privind tipurile aceasta regula privind restrictia nu este strict restrictiva. De fapt inainte de. se permite o lvaloare si acea lvaloare se presupune a avea forma structurii din care face parte numele pus in dreapta (operatorului). Deci expresia din fata lui -> poate fi pointer sau intreg. Daca este pointer se presupune ca indica structura din care face parte membrul drept. Daca e un intreg el va fi o expresie de adresa absoluta in unitati de memorie ai masinii al structurii corespunzatoare. Astfel de constructii nu sint portabile.

14. 2. Functii

Cu o functie se pot face doua lucruri: apelul ei sau obtinirea adresei ei. Daca numele unei functii apare intr-o expresie si nu in apelul de functie se genereaza un pointer la functia

respectiva. Trecerea unei functii alteia:

```
int f();
.
.
.
g(f)
```

Definitia lui g va fi:

```
g(functia)
int (*funcp)();
{...
 (*funcp)();
}
```

De notat ca f trebuie declarata explicit in rutina apelanta pt ca aparitia sa in g(f) nu este urmata de (.

14. 3. Tablouri, pointeri si indici

De fiecare data cind un identificator de tipul tablou apare intr-o expresie el este convertit intr-un pointer catre primul element al tabloului. Din cauza acestei conversii tablourile nu sint lavlori. Prin definitie operatorul indice [] este interpretat de asa maniera incit E1[E2] e identic cu *((E1)+(E2)). Din cauza regulilor de conversie care se aplica lui +, daca E1 este tablou si E2 intreg atunci E1(E2) se refera la al E2-lea membru al lui E1. Darin ciuda acestei aparente asimetrii, indicii formeaza o operatie comutativa.

O regula soloida se aplica in cazul tablourilor multidimensionale. Daca E este un tablou de ordinul "n" si indici ixjx...xk atunci cind E apare intr-o expresie el va fi convertit intr-un pointer la un tablou cu "n-1" dimensiuni cu indici jx...xk. Daca se aplica operatorul *, explicit sau implicit ca rezultat al utilizarii indicilor, rezultatul este un tablou pointat de n-1 dimensiuni, care este imediat convertit intr-un pointer. De exemplu:

```
int x[3][5];
```

Unde x este un tablou de intregi cu 3X5 elemente. Cind x apare intr-o expresie, este convertit intr-un pointer catre(primul din cele 3) tabloul de 5 membrii intregi. In expresia x[i], care e echivalenta cu *(x+i), X este mai intii convertit intr-un pointer asa cum s-a descris;i este apoi convertit in tipul lui x, care implica inmultirea lui i cu lungimea obiectului spre care pointeaza, adica 5 obiecte intregi. Rezultatul se aduna si se aplica indirectarea producind un tablou(de intregi) care este la rindul sau transformat intr-un pointer la primul dintre intregi. Daca mai este un indice acelasi procedeu se aplica din nou; acum rezultatul va fi un

intreg.

Rezulta ca in C tablourile sint stocate pe linii (ultimul indice variaza cel mai repede) si ca primul indice din declaratie permite sa se de termine necesarul de memorie dar nu are alt rol in calculul indicilor.

14. 4. Conversii de pointeri explicite

Unele conversii de pointeri sint permise doar au aspecte dependente de implementare. Toate se specificaprin conversii explicite de tip ca in &7. 2 si &8. 7.

Un pointer se poate converti in oricare tip de intreg suficient de mare pentru a-l pastra. Dar a utiliza int sau long este dependent de masina. Functiile de mapare sint dependente de, dar nu vor surprinde pe aceea care cunosc structura de adresare a masinii. Detalii se dau mai jos.

Un obiect de tip intreg poate fi convertit explicit in pointeri. Maparea face ca un intreg convertit din pointer sa dea acelasi pointer, depinzind de masina. Un pointer catre un tip poate fi convertit intr-un pointer la alt tip. Pointerul rezultat da exceptii de adresare la folosire daca pointerul subiect nu se refera la un obiect aliniat corespunzator in memorie. Se garanteaza ca un pointer la un obiect de o marime data poate fi convertit intr-un pointer catre un obiect mai mic in dimensiune si inapoi fara modificari.

De exemplu, rutina de alocare de memorie poate accepta o marime (in biti, a unui obiect de alocat si sa returneze un pointer char; acesta putind fi folosit conform scopului.

```
extern char *alloc();
double *dp;
dp=(double*)alloc(sizeof(double));
*dp=22. 0/7. 0;
```

alloc trebuie sa asigure (de o maniera dependenta de masina) ca valoarea pe care o returneaza este corespunzatoare connversiei intr-un pointer catre double < atunci utilizarea functiei este portabila.

Reprezentarea pointerului la PDP-11 este un intreg de 16 biti si se masoara in bytes; chars nu necesita aliniere speciala; orice altceva trebuie sa aiba o adresa para.

Pe Honeywell un pointer are 36 de biti si e intreg; adresa de cuvint e pe cei 18 biti din stinga, iar bitii care selecteaza caracterul din cuvint in partea dreapta; deci pointerii char sint masurati in unitati de 2 la 16 bytes; orice altceva in unitati de 2 la 18 cuvinte; double si agregatele care le contin trebuie sa fie la adresa de cuvint para.

IBM 370 si Interdata 8/32 sint similare. La ambele adresele se masoara in bytes; obiectele elementare sint aliniate la limite egale cu lungimea lor; pointeri catre short sint0 mod 2, catre int sau float 0 mod 4 si la double 0 mod 8. Agregatele sint aliniate la limitele stricte necesitate de constituenti.

15. Expresii "constante"

In multe locuri C necesita expresii care dau o constanta: dupa case, ca limite de tablouri, la initializari. In primele 2, expresiile pot cuprinde doar constante intregi, constante caracter, expresii tip sizeof conectate la operatorii binari:

```
+  - * / % & \ ^ << >> == != <> <= >=
```

Sau prin operatorii unari:

```
- ~
```

Sau prin operatorul ternar:

```
?  :
```

Parantezele sint folosite pt grupare, nu pt apeluri de functii. O latitudine mai mare o permit initializarile; in afara de expresiile constante se poate aplica operatorul unar & la obiecte externe sau statice, sau la tablouri externe sau statice avind ca indici expresii constante.

Operatorul unar & poate fi aplicat implicit prin aaparitia de tablouri fara indici sau functii. Regula de baza este ca initializarile conduc la o constanta sau la o adresa a unui obiect extern sau static plus sau minus o constanta.

16. Consideratii de portabilitate

Unele parti din C sint inerent dependente de masina. Lista care urmeaza contine sursele de prorbleme cele mai importante: Lungimea cuvintului de memorie si proprietatea aritmeticei in VF sau impartirea intregilor au dovedit in practica ca nu sint o problema. Alte fatete ale hardware-ului sint reflectate in diversele implementari. Unele din ele, ca extensia de semn (convertirea unui caracter negativ intr-un inttreg) si ordinea bytelor in cuvint, sint neplaceri care trebuie observate atent. Altele sint probleme minore.

Numarul de variabile register care pot fi plasate efectiv in registre depind de la masina la masina, ca si seturi de tipuri valide. Compilatoarele fac lucrurile corespunzator propriei masini; declaratiile register excesive sau invalide sint ignorate.

Unele dificultati cresc cind se folosesc practici de codificare dubioase. Nu este recomandat a se scrie programe care depind de aceasta proprietate.

Ordinea de evaluare a argumentelor functiilor nu este specifi cata de limbaj. La PDP-11 este de la dreapta la stinga, la altele de la stinga la dreapta. Ordinea de aparitie a efectelor secundare nu e specificata.

Daca constantele caracter sint obiecte de tipul int, constantele caracter multicaracter sin permise. Implementarea specifica depinde de masina pt ca ordinea in care caracterele sint plasate in cuvint variaza de la o masina la alta.

Cimpurile apar in cuvint si caracterele in intregi de la dreapta la stinga in PDP si de la stinga la dreapta in alte masini. Aceste diferente sint invizibile in programe izolate care nu fac apel la dependente de tip (de exemplu conversia din pointer int in pointer char si implementarea memoriei pointate) dar trebuie considerate daca se doreste conformarea cu modalitatile de memorare externa.

Limbajul acceptat de diversele compilatoare difera putin. Dar, compilatorul pt PDP=11 folosit curent nu initializeaza structuri de cimpuri de biti, nu accepta operatori de asignare in unele contexte in care se foloseste valoarea asignarii.

17. Anacronisme

C este un limbaj in evolutie, unele constructii invechite pot fi intilnite in programe mai vechi. Unele compilatoare suporta aceste anacronisme, ele fiind pe cale de disparitie, raminind doar problema portabilitatii.

Versiunile mai vechi de C permiteau forma =op in locul lui op= pt asignare. Acestea duceau la ambiguitati, cum ar fi

```
x=-1
```

care acum inseamna scaderea unui 1 din x, = si - fiind adiacente, dar care poate insemna si asignarea lui -1 lui x.

Sintaxa initializarilor s-a schimbat; la inceput semnul = care anunta un initializator nu era prezent, adica in loc de

```
int x = 1;
```

se folosea

```
int x 1;
```

modificarea s-a facut pentru ca initializarea

```
int f(1+2)
```

semana cu o declaratie de functie atit de mult incit deranja compilatoarele.

18. Sumar al sintaxei

Acest numar al sintaxei C-ului are scopul de a ajuta intelegerea sa mai mult decit de a defini exact limbajul.

18. 1. Expresii

Expresii de baza sint:

```
expression:

        primary
        *expression
        &expression
        -expression
        !expression
        ~expression
        ++lvalue
        --lvalue
        lvalue++
        lvalue--
        sizeof expression
        (type-name)expression
        expression binop expression
        expression?expression:expression
        lvalue asgnop expression
        exprerssion, expression

primary:

        identifier
        constant
        string
        (expression)
        primary(expression-listopt)
        primary[expression]
        lvalue. identifier
        primary->identifier

lvalue:
        identifier
        primary[expression]
        lvalue. identifier
        primary->identifier
        *expression
        (lvalue)
```

Operatorii expresiilor primare

() []. ->

au cea mai mare prioritate si grupeaza stinga la dreapta.

Operatorii unari:

* & - ! ~ ++ -- sizeof(type-name)

au prioritatea sub cea a operatorilor primari dar mai mare decit a operatorilor binari si grupeaza de la dreapta la stinga. Operatorii binari si operatorul conditional grupeaza stinga la dreapta si au priortatea descrescind asa cum e indicat

binop:

```
*       /       %
+       -
>>      <<
<       >       <=      >=
==      !=
&
^
|
&&
||
?!
```

Operatorii de asignare au aceeasi prioritate

asgnop:

= += -= *= /=, %= >>= <<= &= ^= \=

Operatorul, au cea mai mica prioritate, grupeaza stinga spre dreapta

18. 2. Declaratii

declaration:

decl-specifiers init-declarator-listopt;

decl-specifiers:

type-specifier decl-specifiersopt
-specifier decl-specifiersopt

sc-specifier:
auto
static

```
        extern
        register
        typedef

type-specifier:

        char
        short
        int
        long
        unsigned
        float
        double
        struct-or-union-specifier
        typedef-name

init-declaration-list:

        init-declarator
        init-declarator, init-declarator-list

init-declarator:

        declarator initializeropt

declarator:
        identifier
        (declarator)
        *declarator
        declarator()
        declarator[constant-expressionopt]

struct-or-union-specifier:

        struct{struct-decl-list}
        struct identifier{struct-decl-list}
        struct identifier
        union{struct-decl-list}
        union identifier{struct-decl-list}
        union identifier

struct-decl-list:

        struct-declaration
        struct-declaration struct-decl-list

struct-declaration:

        type-specifier struct-declarator-list;

struct-declarato-list:

        struct-declarator
        struct-declarator, struct-declarator-list
```

```
struct-declarator:

        declarator
        declarator:constant-expression
        :constant-expression

initializer:

        =expression
        ={initializer-list}
        ={initializer-list, }

initializer-list:

        expression
        initializer-list, initializer-list
        {initializer-list}

type-name:

        type-specifier abstract-declarator

abstract-declarator:

        empty
        (abstract-declarator)
        *abstract-declarator
        abstract-declarator()
        abstract-declarator[constant-expressionopt]

typedef-name:

        identifier
```

18. 3. Enunturi

```
compound-statement:

        {declaration-listopt statement-listopt}

declaration-list:

        declaration
        declaration declaration-list

statement-list

        statement
        statement statement-list

statement:
```

```
                compound-statement
                expression;
                if(expression)statement
                if(expression)statement else statement
                while(expression)stament
                do statement while(expression);

for(expression-1opt;expression-2opt;expression-3opt)statement
                switch(expression)statement
                case constant-expression:statement
                default:statement
                break;
                continue;
                return;
                return expression;
                goto identifier;
                identifier:statement
                ;
```

18. 4. Definitii externe

```
program:

                external-definition
                external-definition program

external-definition:

                function-definition
                data-definition

function-definition

        type-specifieropt function-declarator function-body

function-declarator

                declarator(parameter-listopt)

parameter-list
                identifier
                identifier, parameter-list

function-body:

                type-decl-list function-statement

function-statement

                {declaration-listopt statement-list}

data-definition
```

```
externopt type-specifieropt init-declarator-listopt
staticopt type-specifieropt init-declarator-listopt
```

18. 5. Preprocesor

```
#define identifier token-string
#define identifier(identifier, ..., identifier)token-string
#undef identifier
#include "filname"
#include <filname>
#if constant-expression
#ifdef identifier
#ifndef identifier
#else
#endif
#line constant identifier
```