CAPITOLUL 4. FUNCTII SI STRUCTURA PROGRAMULUI

Functiile impart programele cu calcule mari in mai multe programe mai mici, si permit programatorilor 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.

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

4.1. Notiuni de baza

Pentru a incepe, vom sa scriem un program care imprima fiecare linie care ii este introdusa si care contine un "model" sau un sir de caractere. De exemplu, sa cautam modelul "unul" in urmatoarele linii:

„Un program C poate fi rezident in unul sau mai multe fisiere sursa”

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 introduse in functii si sansa de a da interactiuni nedorite este minimalizata.

Expresia din "While (mai exista o linie) " este **getline**, iar "tipareste-o" este **printf**. Aceasta inseamna ca nu trebuie sa scriem decit o rutina care decide daca linia contine vreo aparitie a modelului.

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. Modelul care trebuie cautat este literal sir din argumentele lui index, care nu este cel mai general dintre mecanisme.

```
#define MAXLINE 1000
main() /* gasiti toate liniile ce contin un model dat */
{
    char line[MAXLINE];
```

```
        while (getline(line, MAXLINE) > 0)
                if (index(line, "unul") >= 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.

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 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 declaratie asupra 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.

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 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(...) sunt 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 acelasi 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 sunt 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 sunt copiate. Functia poate altera elementele tabloului indexind cu aceasta valoare. Efectul este ca tablourile sunt trimise prin referinta.

In capitolul 5 vom discuta folosirea pointerilor pentru a permite functiilor sa nu altereze tablourile din functiile apelante. Nu este un mod intru totul 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 intr-un apel dat. 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 sunt 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 sunt prezente si care sunt tipurile lor. Ea esueaza daca apelantul nu furnizeaza suficiente argumente sau daca tipurile nu sunt cele specificate de primul argument.

Reciproc, daca argumentele sunt 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 sunt 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 sunt definite in afara oricarei functii si sunt astfel disponibile potential pentru mai multe functii. Functiile insesi sunt intotdeauna externe, deoarece limbajul C nu permite definitii de functii in interiorul altor functii.

Implicit, variabilele externe sunt deasemenea "globale", astfel incit toate referintele la o astfel de variabila printr-un acelasi nume (chiar si pentru functiile compilate separat) sunt referinte la un acelasi lucru.

Deoarece variabilele externe sunt 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 sunt 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.

Al treilea motiv pentru folosirea varaiabilelor externe este domeniul si timpul lor de viata. Variabilele automate sunt interne unei functii; ele capata viata atunci cind rutina este introdusa si dispar atunci cind rutina se termina.

Variabilele externe, pe de alta parte, sunt 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 sunt 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".

O expresie "infix", de tipul:

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

se introduce astfel:

1 2 - 4 5 + * =

Parantezele nu sunt necesare.

Implementarea este aproape simpla. Fiecare operand este depus intr-o stiva. Cind soseste un operator, numarul de operanzi (doi pentru operatorii liniari) sunt scosi din stiva si li se aplica operatorul iar rezultatul este depus din nou in stiva.

In exemplul de mai sus, 1 si 2 sunt depusi in stiva, apoi sunt inlocuiti de diferenta lor, -1 . Apoi 4 si 5 sunt depusi in stiva, apoi sunt 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 sunt triviale dar, daca se adauga detectia de erori de timp si recuperarea lor, codurile sunt 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 sunt 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 foloseasca:

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

pentru a examina virful stivei fara a-l distruge. Sa notam deasemenea ca deoarece + si * sunt operatori comutativi, ordinea 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 sunt:

- Cum sunt scrise declaratiile astfel incit variabilele sa fie declarate cum se cuvine in timpul compilarii ?
- Cum sunt 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 sunt fara legatura unele cu altele. La fel se intimpla si cu argumentele functiilor.

Domeniul unei variabile externe dureaza din punctul in care ea este declarata intr-un fisier sursa pina la sfirsitul acelui fisier. De exemplu, daca val,sp,push,pop,clear sunt 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 sunt 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, atunci 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 sunt optionale cu o declaratie externa.

Cu toate ca nu este o organizare adecvata pentru acest program ,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 sunt **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 cu un caracter mai mult mai decat necesar. 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 considerat; **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 inregistreaza pozitia caracterului curent din buffer.

Deoarece bufferul si indexul sunt 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 nu va 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 sunt 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 sunt 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 functie, ca de exemplu argumentele lui **printf**, sunt statice interne.

O variabila statica externa este recunoscuta in restul 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 sunt vizibile pentru utilizatorii lui getch si ungetch, asa ca nu exista nici o posibilitate de conflict. Daca cele doua rutine si cele doua variabile sunt compilate intr-un fisier:

```
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 sunt obiecte externe; numele lor sunt 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.

Obiectele interne statice sunt cunoscute numai in interiorul unei functii ;obiectele externe statice (variabile sau functii) sunt 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 sunt 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 sunt inaccesibile din afara.

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 variabile din fiecare functie pot fi pastrate in registri si numai anumite tipuri sunt 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 sunt 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, 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.

Daca sunt date declaratiile:

```
int x;
```

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

atunci, in cadrul functiei f, ocurentele 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 sunt nedefinite (i.e. gunoi, ramasita). Variabilele simple (nu tablourile sau structurile) pot fi initializate cind se declara, punind in continuarea numelui lor semnul egal si o expresie constanta:

```
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, initializarea se face de fiecare data cind functia sau blocul se executa. Pentru variabilele automate si de registru valoarea de initializare 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 sunt 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 declaratii sunt 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 sunt de fapt necesare deoarece sunt toate zero dar este o buna practica de programare de a le da explicit. Daca valorile de initializare specificate sunt mai putine decit marimea specificata, restul valorilor vor fi zero. Daca ele sunt mai multe se provoaca eroare.

Tablourile de caractere sunt 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 sunt generate intr-o ordine gresita: cele mai putin semnificative sunt 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.

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 sunt 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 variabile.

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, sunt foarte putine restrictii asupra a ce poate fi definit:

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

si apoi se 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 sunt 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 sunt evaluate de doua ori; aceasta este rau daca sunt implicate 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.

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