Prezentarea Limbajului C

C este un limbaj de programare cu scop general ale cărui caracteristici sunt economia de expresie, structuri moderne de control al fluxului şi de date, precum şi un set bogat de operatori.

C nu este un limbaj de nivel "foarte înalt" şi nu este specializat vreunei arii particulare de aplicații. Dar absenta în restricții şi generalitatea sa îl fac mai convenabil şi mai de efect pentru mai multe scopuri decât limbaje presupuse mai puternice.

C a fost la început proiectat şi implementat pe sistemul de operare UNIX de către Dennis Ritchie.

C nu este legat de nici un hardware sau calculator anumit şi e simplu de scris programe care se pot executa fără nici o modificare pe diferite calculatoare care au limbajul C implementat.

Acest curs are drept scop inițierea în programare utilizând acest limbaj. Marea parte a textului nu se bazează atât pe expunerea de reguli şi propoziții cât pe citirea, scrierea şi revizuirea de exemple.

INTRODUCERE

C operează cu aceeaşi **clasă de obiecte** cu care lucrează majoritatea calculatoarelor, şi anume **caractere, numere şi adrese**. Acestea pot fi combinate şi prelucrate cu operatori aritmetici şi logici implementați pe actualele calculatoare.

Cu toate ca C se potriveşte cu caracteristicile multor calculatoare, el este independent de arhitectura oricărui calculator particular şi astfel este uşor a scrie programe "portabile", adică programe care pot fi rulate fără modificări pe varietate de calculatoare. În afara programelor care, în mod necesar, sunt dependente de tipul de calculator, ca: asamblorul, compilatorul, depanatorul, software-ul scris în C este identic pe ambele calculatoare.

Multe din cele mai importante idei din C îşi au rădăcina în limbajul BCPL, dezvoltat de Martin Richards. Influenta lui BCPL asupra lui C apare indirect prin limbajul B, care a fost scris de Ken Thompson în 1970 pentru primul sistem UNIX.

1. INIȚIERE

Prezentarea se va concentra în continuare asupra fundamentelor: variabile şi constante, aritmetica, controlul fluxului, funcții şi rudimente de operații de I/O.

Pentru moment se vor evita caracteristici ale limbajului C care sunt de importanță vitală în scrierea programelor mai mari: pointerii, structurile, anumite instrucțiuni de control al fluxului.

1.1 Primul program.

Singurul mod de a invata un nou limbaj de programare este de a scrie programe în el. Primul program pe care-l vom scrie este acelaşi pentru toate limbajele: afişați pe ecran cuvintele “**hello, world!”.**

Acesta este primul obstacol; pentru a sări peste el, trebuie sa fiți în stare să creați undeva textul program, să-l compilați cu succes, să-l încărcați, să-l executați şi să aflați textul tipărit acolo unde este ieşirea calculatorului dumneavoastră. În C, programul pentru a tipări "hello, world" este:

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

Daca n-ați greşit nimic, de exemplu să fi uitat un caracter sau să fi inversat două caractere, compilarea se va desfăşura silențios şi va produce un fişier obiect numit "hello.obj". Lansându-l în execuție după link-editare cu comenzile:

>run hello

va produce pe ecran mesajul: **hello, world** ca ieşire a sa.

Exercițiul 1.1. Executați acest program pe sistemul vostru. Încercați să vedeți ce mesaje de eroare obțineți, lăsând la o parte părți din program.

Un program C, oricare i-ar fi mărimea, constă din una sau mai multe "funcții" care specifică operațiile efective de calculat care trebuiesc făcute. În exemplul nostru, "**main**" este o astfel de funcție. În mod normal aveți libertatea de a da funcțiilor ce nume doriți, dar "**main**" este un nume special - programul dumneavoastră se va executa de la începutul lui "**main**". Aceasta înseamnă că **fiecare program trebuie să aibă un "main"** undeva, că "**main**" va invoca în mod obişnuit alte funcții pentru a-şi realiza scopul, unele venind din acelaşi program iar altele din biblioteci ce conțin funcții scrise anterior.

O metoda de a comunica date între funcții este prin argumentele funcțiilor. Parantezele care urmează după numele funcției includ **lista de argumente**. În cazul nostru, "**main**" este o funcție fără argumente ceea ce se indica prin "()". Acoladele "{ }" includ instrucțiunile care alcătuiesc funcția. O funcție este apelată prin nume, urmate de o listă de argumente în paranteze.

Linia **"printf("hello, world\n");"** este un apel care cheamă o funcție numită "**printf**" cu argumentul "hello, world\n". "**printf**" este o funcție din bibliotecă şi tipăreşte pe monitor şirul de caractere care alcătuiesc argumentul.

O secvență alcătuită din orice număr de caractere cuprinse între două ghilimele "..." se numeşte sir de caractere sau constanta şir. Pentru moment, singura folosire a şirurilor de caractere va fi ca argumente pentru "**printf**" şi alte funcții.

Secvența "**\n**" din şir este notația din C pentru caracterul "linie nouă", care, când este tipărit, avansează cursorul terminalului la marginea din stânga a următoarei linii. Daca uitați "\n" veți observa că ieşirea dumneavoastră nu se termină cu o linie nouă. Singurul mod de a avea caracterul "linie nouă" în "**printf**" este "**\n**" ca argument.

Daca încercați ceva de tipul

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

compilatorul C vă va tipări un diagnostic neprietenos despre ghilimele absente. "printf" nu furnizează o linie nouă în mod automat, aşa că apelurile multiple pot fi folosite pentru a tipări o linie pe etape.
Primul nostru program poate fi scris la fel de bine şi astfel:

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

pentru a produce o ieşire identică. Să notăm că "\n" reprezintă un singur caracter. O "secvență escape" ca de exemplu "\n" este în general un mecanism extensibil pentru reprezentarea caracterelor greu de obținut sau invizibile. Printre alte secvențe escape, limbajul C posedă: **\t** pentru tab, **\b** pentru backspace, **\"** pentru apostrof dublu şi **** pentru backspace.

Exercițiul 1.2. Experimentați să vedeți ce se întâmplă când şirul argument din "printf" conține "\x" un x este un caracter oarecare care nu a fost listat mai sus.

1.2. Variabile şi aritmetica

Următorul program tipăreşte tabela de temperaturi Fahrenheit şi echivalentele lor în 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

Iată acum şi programul:

```
/* afiseaza tabelul de conversie Fahrenheit-Celsius pentru f = 0, 20, ..., 300 */
```

```
main()
{
        int lower, upper, step;
        float fahr, celsius;
         lower = 0; /* temperatura limita inferioara a tabelului */
        upper = 300; /* limita superioara */
        step = 20; /* marimea pasului */
        fahr = lower;
        while (fahr <= upper) {
                celsius = (5.0 / 9.0) * (fahr - 32.0);
                printf("%4.0f %6.1f\n", fahr, celsius);
                fahr = fahr + step;
        }
}
```

Prima linie:

/* afiseaza tabelul de conversie Fahrenheit-Celsius pentru f = 0, 20, ..., 300 */

este un **comentariu**, care în acest caz explica pe scurt ce face programul. **Orice caractere cuprinse între "/*" şi "*/" sunt ignorate de compilator**; ele pot fi folosite liber pentru a face programul mai uşor de înțeles. Comentariile pot apare oriunde în program.

În limbajul C, **toate variabilele trebuie declarate înainte de a fi folosite**, de obicei la începutul liniei, înaintea oricărei instrucțiuni executabile. Dacă veți uita o declarație, veți primi un diagnostic de la compilator. O declarație constă **dintr-un "tip" şi o lista de variabile** care au acel tip, ca în:

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

Tipul "**int**" implică faptul că variabilele listate sunt întregi; "**float**" denotă virgula mobilă, adică numere care pot avea parte fracționară. Precizia atât pentru "**int**" cât şi pentru "**float**" depinde de calculatorul pe care-l folosiți. Pentru un PC un "**int**" este un număr cu semn de 16 biți, adică un număr cuprins între -32768 şi +32767. Un număr "**float**" este o cantitate ce se reprezintă pe 32 de biți ceea ce revine la aproximativ şapte cifre semnificative, cu magnitudinea cuprinsă aproximativ între 10^{-38} şi 10^{38}.

Pe lângă tipurile "**int**" şi "**float**", limbajul C posedă şi alte tipuri de date fundamentale :

"**char**" caracter - un singur octet
"**short**" întreg scurt
"**long**" întreg lung
"**double**" număr flotant dubla precizie

Calculul efectiv în programul de conversie temperatura începe cu atribuirile:

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

care setează variabilele pe valorile lor de start, inițiale. Instrucțiunile individuale se termină cu punct şi virgulă. Fiecare linie a tabelei este calculată în acelaşi fel, aşa că vom folosi o buclă care se repetă odată pe linie; acesta este scopul instrucțiunii "**while**":

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

Este testata condiția din paranteză. Daca ea este adevărată (**fahr** este mai mic sau egal cu **upper**), este executat corpul buclei (toate instrucțiunile incluse între parantezele { şi }). Apoi condiția este retestată şi dacă este adevărată, corpul este executat din nou. Când testul devine fals (**fahr** este mai mare decât **upper**), bucla se termină şi execuția continuă cu instrucțiunea care urmează buclei. Deoarece în acest program nu mai există alte instrucțiuni care să succeadă bucla, execuția lui se termina. Corpul unei bucle **while** poate fi alcătuit din mai multe instrucțiuni incluse între acolade, ca în programul de mai sus sau dintr-o singura instrucțiune, fară paranteze, ca în exemplul de mai jos:

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

În ambele cazuri, instrucțiunile controlate de "**while**" sunt decalate cu un tab, aşa că se observă uşor ce instrucțiuni se găsesc în interiorul buclei. Decalarea scoate în evidență structura logică a programului. Se recomandă scrierea unei singure instrucțiuni pe un rând şi lăsarea de spații în jurul operatorilor. Poziția acoladelor este mai puțin importantă.

Temperatura Celsius este calculată şi atribuită variabilei "celsius" prin instrucțiunea:

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

Rațiunea pentru folosirea lui (5.0/9.0) în locul mai simplei 5/9 este aceea că în C, ca şi în multe alte limbaje, împărțirea întreagă trunchiază rezultatul, aşa că orice parte fracționară este eliminata. Astfel 5/9 este zero şi tot asa ar fi fost toate temperaturile. Un punct zecimal intr-o constanta indica faptul ca aceasta este flotanta şi deci 5.0/9.0 este 0.5555... ceea ce am şi dorit. Am scris deasemenea 32.0 în loc de 32, chiar daca deoarece "fahr" este "**float**", 32 ar fi convertit automat în "**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 şi ne asigura ca şi compilatorul va gindi în acelasi mod ca şi noi.

Asignarea

```
fahr = lower;
```

şi testul

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

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

Acest exemplu arata mai multe despre modul de lucru al lui "**printf**". Ea este o **funcție de conversie de format cu scop general**, care va fi descrisa complet în cursurile următoare. Primul sau argument este un sir de caractere ce se vor tipari, fiecare caracter **%** indicind argumentele (al doilea, al treilea) ce se va substitui şi forma în care se vor tipari.

De exemplu, în 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. 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 restrictiilor; "**%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 şi "**%%**" pentru insusi semnul %.

Fiecare constructie cu **%** în primul argument al lui "**printf**" face pereche cu al doilea, al treilea,... argument. Aceste perechi trebuie sa corespunda ca numar şi tip.

Exercițiul 1.3. Modificați programul de conversie temperaturi pentru a scrie un antet la începutul tabelei de conversie.
Exercițiul 1.4. Scrieți un program care să tipărească tabela corespunzătoare Celsius - Fahrenheit.

1.3. Instructiunea For

Exista o multime de moduri pentru a scrie un program. O alta varianta a programului de conversie de temperatura este:

```
main() /* tabel conversie Fahrenheit-Celsius */
{
        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. O modificare esentiala este eliminarea majoritatii variabilelor; a ramas numai "fahr", declarata ca "**int**" (observati specificatorul "**%d**" în **printf**). Limitele inferioara şi superioara şi marimea pasului apar doar ca şi constante în instructiunea "for", ea insasi o constructie noua, iar expresia care calculeaza temperatura Celsius apare acum ca al treilea argument din "**printf**" în loc de a fi o instructiune de asignare separata. 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 şi 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 şi apoi conditia este reevaluata.

Bucla se termina atunci cind conditia devine falsa. La fel ca şi la instructiunea "**while**", corpul buclei poate fi alcatuit dintr-o singura instructiune sau dintr-un grup de instructiuni inclus între acolade. Partile de initializare şi reinitializare pot fi o singura expresie.

Alegerea între "**while**" şi "**for**" este arbitrara, bazata pe ceea ce ne pare noua a fi mai clar. Instructiunea "**for**" este potrivita în mod uzual pentru buclele în care initializarea şi reinitializarea sunt instructiuni unice şi logic inrudite deoarece este mai compacta decit "**while**" şi pastreaza instructiunile de control al buclei intr-un singur loc şi impreuna.

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

1.4. Constante simbolice

Cu ajutorul constructiei "**#define**", **se pot defini la inceputul programului nume sau constante simbolice**, care sunt un sir particular de caractere. Dupa aceea, compilatorul va inlocui toate aparitiile nepuse între ghilimele ale numelui, prin sirul corespunzator. Inlocuirea efectiva a numelui poate fi orice text; ea nu se limiteaza la numere. Programul poate fi citit şi valorile acestor constante pot fi modificate mai uşor.

```
#define LOWER 0 /* limita inferioara a tabelului */
#define UPPER 300 /* limita superioara */
#define STEP 20 /* pasul */
main() /* tabel de conversie Fahrenheit-Celsius */
{
        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** şi **STEP** sunt constante, asa incit ele nu apar în declaratii. Numele simbolice se scriu în mod normal cu litere mari, asa ca ele pot fi usor distinse de numele de variabile care se scriu cu litere mici. **La sfirsitul unei definitii NU se pune punct şi virgula.**

1.5. O colectie de programe utile

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

Introducere şi extragere de caractere

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

c=getchar()

variabila "**c**" contine urmatorul caracter de intrare. Caracterele vin în mod normal de la tastatura.

Funcția "**putchar(c)**" este complementara lui "**getchar()**":

putchar(c)

tipareste continutul variabilei "**c**" pe un mediu de iesire, monitor. Apelurile la "**putchar**" şi "**printf**" pot fi intercalate; iesirea va apare în ordinea în care s-au facut apelurile.

Ca şi în cazul lui "**printf**", nu exista nimic special relativ la "**getchar**" şi "**putchar**". Ele nu sunt parti ale limbajului C, dar sunt universal disponibile.

Copiere de caractere

Date "**getchar**" şi "**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-şi copiaza intrarea în 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 în limbajul C, obtinem:

```
main() /* copiaza datele de intrare la iesire; versiunea 1*/
{
        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; în 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 de obicei numele simbolic de **EOF** pentru aceasta valoare, oricare ar fi fost ea. În 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 returneaza **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**". Programul de copiere ar putea fi de fapt scris mult mai concis de catre un programator experimentat în limbajul C. În C, o asignare ca

```
c = getchar()
```

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

```
main() /* copiaza datele de intrare la iesire; versiunea 2
{
        int c
        while ((c = getchar()) != EOF)
                putchar(c);
}
```

Programul citeste un caracter, il asigneaza lui "**c**" şi apoi testeaza daca acesta a fost semnalul de sfirsit de fisier. Daca nu a fost, corpul buclei "**while**" este executat, tiparindu-se caracterul şi bucla se repeta. Cind semnalul de sfirsit de fisier este atins în fine, bucla "**while**" se termina, terminindu-se totodata şi programul "**main**". Plasarea unei asignari intr-un test constituie unul din locurile unde C permite o concizie uluitoare.

Este important sa recunoastem ca **parantezele ce includ asignarea sunt absolut necesare**. Ponderea lui "**!=**" este mai mare decit aceea a lui "**=**" ceea ce inseamna ca, în 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.

Contorizarea caracterelor

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

```
main() /* contorizeaza numarul de caractere introduse de la tastatura */
{
        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 şi "**nc = nc+1**", dar "**++nc**" este mai concisa şi adesea mai eficienta. Exista un operator corespunzator, "**--**" pentru decrementare cu 1. Operatorii "**++**" şi "**--**" pot fi atit operatori prefix, cit şi sufix ("**nc++**"); aceste doua forme au valori diferite în expresii dar "**++nc**" şi "**nc++**" il incrementeaza amindoi pe "**nc**". Programul de contorizare de caractere acumuleaza numarul de caractere intr-o variabila "**long**" în loc de "**int**". Specificatorul de conversie "**%ld**" semnaleaza lui "**printf**" ca argumentul corespunzator este un intreg "**long**". Pentru a face fata la numere chiar şi mai mari, se poate folosi o declaratie de "**double**" ("**float**" de lungime dubla). Vom folosi, deasemenea, instructiunea "**for**" în locul lui "**while**" pentru a ilustra un alt mod în scrierea buclei.

```
main() /* contorizeaza numarul de caractere introduse de la tastatura */
{
        double nc ;
        for (nc = 0; getchar() != EOF; ++nc)
                ;
        printf("%.0f\n", nc);
}
```

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

Corpul buclei "**for**" este în cazul acesta vid, deoarece toata munca este facuta în partile de test şi reinitializare. Dar regulile gramaticale ale limbajului C pretind ca o instructiune "for" sa aiba un corp. Punctul şi virgula ce apare izolat pe o linie, în mod tehnic o instructiune nula, este pus tocmai pentru a satisface aceasta cerere. Daca intrarea nu contine nici un caracter, testul din "**while**" sau "**for**" esueaza la primul apel la **getchar** şi deci rezultatul programului este zero, ceea ce este corect.

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 la fiecare linie scrisa.

```
main() /* contorizarea numarului de linii de caractere intruduse de la tastatura */
{
        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 şi, daca este adevarata, se executa instructiunea (sau grupul de instructiuni dintre acolade) care urmeaza.

Semnul dublu de egal "**==**" este în C notatia pentru "**este egal cu**". Acest simbol este folosit pentru a distinge testul de egalitate de egal simplu (**=**) folosit pentru asignare.

Orice caracter singur poate fi scris între apostrofuri, pentru a produce valoarea numerica a caracterului în codul de caractere al calculatorului; acesta se numeste **constanta de caracter**. Asa de exemplu, '**A**' este o constanta de caracter; în setul de caractere ASCII, valoarea sa este 65, reprezentarea interna a caracterului A.

Secventele escape folosite în sirurile de caractere sunt şi ele legale în constantele de caracter, asa ca în teste şi în expresii aritmetice '**\n**' tine locul caracterului "**linie noua**".

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

Exercitiul 1.7. Scrieti un program care sa copieze intrarea în 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 şi backspace-urile vizibile.

Contorizarea de cuvinte

Al patrulea program din seria de programe utile va contoriza linii, cuvinte şi caractere, un singur cuvint fiind definit ca orice secventa de caractere care nu contine blanc, tab sau linie noua.

```
#define YES 1
```

```
#define NO 0
main() /*contorizare linii, cuvinte şi 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 " şi variabilei i s-a asignat valoarea **NO**. Preferam constantele simbolice **YES** şi **NO** valorilor literale 1 şi 0 deoarece ele fac programul mai usor citibil.

Linia

nl = nw = nc = 0;

seteaza toate cele trei variabile pe zero.

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 **ŞI**. Expresiile conectate prin **&&** sau **||** sunt evaluate de la stinga la dreapta şi 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.

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 şi 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. În exemplul nostru instructiuea de dupa "**else**" este un "**if**" care controleaza doua instructiuni în paranteze.

Exercitiul 1.9. Cum veti testa programul de contorizare cuvinte? Care sunt 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 şi apostrofuri care incepe cu o litera.

1.6. Tablouri

Este prezentat in continuare un program care va contoriza aparitiile fiecarei cifre, a fiecarui caracter de spatiere (blanc, tab, linie noua) şi a tuturor celorlalte caractere. Exista 12 categorii de intrari, asa ca este mai convenabil sa folosim un tablou pentru a pastra numarul de aparitii a fiecarei cifre, decit sa folosim 10 variabile individuale:

```
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 intotdeauna incep de la zero în C asa ca elementele tabloului sunt **ndigit[0]**, **ndigit[1]** , ...,**ndigit[9]**. Acestea se reflecta în buclele "**for**", care initializeaza şi 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 ale 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 sunt pozitive şi în ordine crescatoare, şi între '**0**' şi '**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**" şi "**int**", convertesc totul în tipul "**int**" inainte de prelucrare, asa ca variabilele şi constantele de tip "**char**" sunt esential identice cu tipul "**int**" în contexte aritmetice.

Decizia se ia asupra caracterului (daca el este o cifra, un caracter de spatiere sau altceva) în 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 în programe ca o modalitate de a exprima deciziile multiple. Codul se citeste simplu de sus în jos pina cind o conditie este indeplinita; în acest punct, se executa partea corespunzatoare de "**instructiune**" şi intreaga constructie este terminata. (Desigur ca "**instructiune**" pot fi mai multe instructiuni incluse între paranteze). Daca nici una din conditii nu este indeplinita, instructiunea care urmeaza dupa ultimul "else" este executata daca este prezenta. Daca "**else**" final şi "**instructiune**" lipsesc (ca în programul nostru), nu are loc nici o actiune.

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. Funcții

În C o funcție este echivalenta cu o subrutina sau cu o funcție din FORTRAN sau cu o procedura din PASCAL, etc. O funcție 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 sunt singura modalitate de a face fata la complexitatea potentiala a programelor mari. Cu funcții scrise asa cum trebuie, este posibil sa ignoram "cum" este facuta o anumita treaba; ne este suficient sa stim "ce" anumita treaba face. Limbajul C este proiectat pentru a face folosirea lor usoara, convenabila şi eficienta; veti observa adesea o funcție lunga numai de citeva rinduri, apelata o singura data, numai fiindca ea clarifica anumite portiuni de cod.

Pina acum am folosit numai funcții ca **printf**, **getchar** şi putchar, care au fost scrise de altii pentru noi; este momentul sa ne scriem şi noi propriile noastre funcții. Vom ilustra mecanismul de definire de funcții scriind o funcție putere "power(m,n)" care va ridica un intreg la o putere intreaga pozitiva în n si un program principal care o foloseste, asa ca puteti vedea deodata intreaga structura:

```
main() /* testeaza funcția 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 funcție are o aceeasi forma:

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

Functiile pot apare în orice ordine şi intr-un fisier sursa sau în doua. Pentru moment vom presupune ca ambele funcții se gasesc intr-un acelasi fisier. Funcția "**power**" este apelata de doua ori în 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 şi tiparit. Intr-o expresie **power(2,i)** este un intreg, la fel ca şi 2 şi i. În "**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 între lista de argumente şi acolada stinga deschisa {; fiecare declaratie se termina cu punct şi virgula. Numele folosite de power pentru argumentele sale sunt pur locale lui power şi nu sunt accesibile nici unei alte funcții: alte rutine pot folosi aceleasi nume fara nici un conflict. Aceasta este adevarat şi pentru variabilele **i** şi **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 în main prin instructiunea "**return**". Orice expresie poate apare între paranteze. O funcție 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 funcții prin atingerea parantezei drepte terminatoare.

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

1.8. Argumente - apel prin valoare

În C, toate argumentele functiei sunt transmise "**prin valoare**". Aceasta inseamna ca **functiei apelate i se transmit valorile argumentelor în variabile temporare** (de fapt intr-o stiva) şi nu i se transmit adresele lor. Aceasta duce la citeva proprietati diferite fata de limbajele cu "apel prin referinta" de tipul FORTRAN. Principala distinctie este aceea ca în limbajul C, **funcția apelata nu poate altera o variabila în funcția apelata**; ea poate altera numai copia ei temporara şi privata. Apelul prin valoare este, cu toate acestea un avantaj şi 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 în rutina apelata.

Drept exemplu, dam în 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, şi este decrementat pina cind devine zero; nu mai este nevoie de variabila **i**. Ceea ce se face cu **n** în 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 funcție sa modifice o variabila în rutina apelanta. Apelandul trebuie sa dea adresa variabilei de setat (în mod tehnic, sa creeze un pointer la variabila),iar funcția apelata trebuie sa declare argumentul ca fiind un pointer şi sa refere variabila reala în mod indirect prin el.

Cind numele unui tablou este folosit ca şi argument, valoarea transmisa functiei este locatia sau adresa de inceput a tabloului. Indexand aceasta valoare, funcția poate avea acces şi altera orice element al tabloului.

1.9. Tablouri de caractere

În mod probabil, cel mai comun tip de tablouri în limbajul C este tabloul de caractere. Pentru a ilustra folosirea tablourilor de caractere şi a functiilor care le manipuleaza, vom scrie un program care citeste un set de linii şi 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 şi lungimea ei
tipareste linia cea mai lunga
```

Aceasta schita ne arata clar ca programul se imparte în bucati. O bucata citeste o linie noua, o alta bucata o testeaza, o alta o salveaza iar restul controleaza procesul. Vom scrie la inceput o funcție **getline** care va citi urmatoarea linie de la intrare; ea este generalizare a functiei **getchar**. În 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 şi 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 funcție, **copy**, pentru a salva noua linie intr-un loc sigur. În final, avem nevoie de un program principal care sa controleze functiile **getline** şi **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 în 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 în s2; s2 suficient de mare */
        char s1[], s2[];
{
        int i;
         i = 0;
        while ((s2[i] = s1[i]) != '\0')
                ++i;
}
```

main şi **getline** comunica între ele printr-o pereche de argumente şi o valoare returnata. În **getline**, argumentele sunt 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 în **getline** deoarece ea este determinata în **main**. "**getline**" foloseste instructiunea **return** pentru a trimite o valoare inapoi apelantului, la fel cum facea şi funcția **power**. Unele funcții returneaza o valoare utila; altele, de exemplu **copy**, sunt folosite numai pentru efectul lor şi 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 şi 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 şi terminat cu **\0**, astfel incit o funcție, de exemplu **printf**, poate sa-i determine sfirsitul.

```
-------------------------------
| h | e | l | l | o | \n | \0 |
-------------------------------
```

Specificatorul de format **%s** din **printf** se asteapta la un sir reprezentat tocmai în aceasta forma. Daca examinati funcția **copy**, veti descoperi ca şi ea se bizuie de fapt pe terminarea argumentului sau de intrare **s1** cu un **\0** şi ea copiaza acest caracter în argumentul de iesire **s2**.

Exercitiul 1.14. Revizuiti rutina main din programul precedent astfel incit ea sa tipareasca corect lungimea unei linii de intrare de o lungime arbitrara, şi 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 şi care sa stearga liniile care contin numai blancuri.

Exercitiul 1.17 Scrieti o funcție 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) sunt **private** sau **locale** lui **main**; deoarece ele sunt declarate în **main**, nici o alta funcție nu poate avea acces direct la ele. La fel se intimpla şi cu variabilele din alte funcții 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 funcția este apelata şi "dispare" cind funcția isi termina activitatea**. Aceasta este ratiunea pentru care astfel de variabile sunt cunoscute uzual sunt numele de **variabile automate**, urmind terminologia din alte limbaje.

Deoarece variabilele automate vin şi pleaca odata cu apelurile de funcții, ele nu-şi pastreaza valoarea de la un apel la altul şi trebuie initializate explicit inainte de fiecare intrare. Daca nu sunt 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 funcție care doreste sa o faca. Deoarece variabilele externe sunt accesibile global, ele pot fi folosite în locul listelor de argumente pentru a comunica date între funcții. Mai mult, deoarece **variabilele externe exista permanent**, şi nu apar şi dispar

dupa cum funcția este apelata sau s-a terminat, **ele isi pastreaza valorile chiar şi dupa ce s-a terminat funcția care le-a setat**.

O variabila externa trebuie sa fie definita în afara oricarei funcții; acest lucru face sa se aloce memorie reala pentru ea. Variabila trebuie deasemenea sa fie declarata în fiecare funcție 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, în care line, save, max vor fi declarate variabile externe. Aceasta va cere modificari în apeluri, în declaratii şi în corpurile celor trei funcții.

```
#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** şi **copy** sunt definite de primele linii din exemplul de mai sus care declara tipul lor şi provoaca o alocare de memorie pentru ele. Din punct de vedere sintactic, definitiile externe sunt asemanatoare cu declaratiile pe care le-am folosit anterior dar deoarece ele apar în afara functiilor, variabilele sunt externe.

Inainte ca o funcție 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 în plus cuvintul cheie **extern**.

Exercitiul 1.18 Testul din instructiunea for din funcția 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 ?

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 în n pozitii.

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

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