Considerații teoretice

Formatori:
Tutor: Stângaciu Valentin
Tutor: Belu Claudiu-Marcel
+3

Data de începere a cursului:
25.09.2023
Utilizatori înscriși
Calendar
Note

Considerații teoretice

Vectori (vectors, arrays)

Un vector este o mulțime unidimensională de elemente în care fiecărui element îi este asociat un index, cu ajutorul căruia elementul poate fi accesat. De exemplu, putem folosi vectori pentru a memora notele primite de un student, valorile citite de un senzor, fișierele dintr-un director, etc. Uneori se mai folosește cuvântul „tablouri" pentru vectori, dar în general un tablou se referă la un vector bidimensional (matrice).

În C variabilele de tip vector se declară punând paranteze drepte (**[ ]**) după numele variabilei și în interiorul acestor paranteze numărul maxim de elemente care va putea fi conținut în vector:

```
int a[100];            //declară vectorul „a" care poate conține maxim 100 de elemente de tip int
float note[3];        //declară vectorul „note" care poate conține maxim 3 valori de tip float
```

În C întotdeauna indecșii într-un vector încep de la 0. Notația din matematică a_i („*a de i*" sau „*elementul din a de la indexul i*") se traduce în C prin **a[i]**, adică indexul se pune după numele vectorului, între paranteze drepte. Cu această notație putem folosi elementele unui vector la fel ca pe variabilele obișnuite:

```
a[0]=7;                      // pune valoarea 7 la prima poziție (index) în vector
a[1]=-1;                     // pune valoarea -1 la a doua poziție în vector
printf("%d",a[0]);         // afișează valoarea de la prima poziție
scanf("%d",&a[2]);      // citește de la tastatură o valoare și o depune la a treia poziție
s=a[0]+a[1];                 // calculează suma elementelor de la primele două poziții
```

Deoarece în C indecșii încep întotdeauna de la 0, dacă avem un vector de n elemente, ultimul element va fi situat la poziția *n-1*. Pentru vectorul *a* definit mai sus, avem:

a[0]	a[1]	...	a[98]	a[99]

total: 100 elemente

Putem folosi mai puține poziții decât am declarat pentru un vector, dar este important să existe poziții suficiente pentru toate situațiile posibile. Dacă am încerca să folosim poziții care nu au fost declarate, vor apărea la rularea programului unele erori care pot fi destul de greu de depanat. De exemplu, dacă avem vectorul *float v[5]*, putem folosi dacă dorim doar 3 poziții din cele 5 posibile, dar în niciun caz nu vom avea la dispoziție 6, 7, etc poziții. În C un vector odată declarat nu poate fi redimensionat (în caz că avem nevoie de mai multe sau de mai puține poziții), așa că va trebui să declarăm de la început vectorul cu un număr acoperitor de poziții pentru toate cazurile posibile.

Dacă știm de la început ce elemente va conține vectorul, putem să-l inițializăm direct, în felul următor:

```
int v[]={7, 10, -1, 3, 5};
```

Elementele cu care se inițializează vectorul se pun între acolade (**{ }**), separate prin virgulă (**,**). Dacă numărul de poziții din vector este egal cu numărul de elemente date, atunci nu mai este nevoie să se declare dimensiunea vectorului, din cauză că aceasta poate fi dedusă de către compilator. În cazul de mai sus, vectorul va avea implicit 5 poziții. Această dimensiune trebuie dată doar dacă dorim să rezervăm mai multe poziții decât numărul elementelor de la inițializare.

Exemplu: Să se calculeze suma elementelor vectorului {2, 9, 8, 5, 7, 3, 4}

```c
#include <stdio.h>
int main()
{
    int v[] = {2, 9, 8, 5, 7, 3, 4};
    int n = 7;                 //numarul de elemente din vector
    int s = 0;                 //initializare suma
    int i;                        //variabila iterator
    for (i = 0; i < n; i++){
        s = s + v[i];         //sau: s+=v[i];
    }
    printf("suma este % d", s);
    return 0;
}
```

Se constată că s-a folosit o variabilă *n* care conține numărul de elemente din vector. Programul ar fi putut fi scris și *for(i=0;i<7;i++)*, dar atunci, dacă am fi modificat numărul de elemente, și acesta ar fi apărut în mai multe locuri în program, ar fi trebuit să modificăm în toate aceste locuri. Folosind variabila *n*, ajunge să modificăm într-un singur loc (la inițializarea lui *n*) și schimbarea va fi aplicată în toate locurile.

Variabila *i* se numește **iterator** deoarece cu ajutorul ei se parcurge (*iterează*) tot vectorul. Conform celor prezentate mai sus, *v[i]* se citește „valoarea din vectorul v de la poziția (indexul) i". Cea mai folosită formă de iterare este de la stânga la dreapta pe toate elementele din vector, realizată cu instrucțiunea **for(i=0;i<n;i++)**.

Dacă se dorește afișarea unor elemente din vector, acestea se pot afișa simplu, câte unul pe fiecare linie. Alternativ, pentru o afișare mai compactă, cu toate elementele pe aceeași linie, putem afișa fiecare element urmat de un spațiu și, la final, după iterație, afișăm \n pentru a trece la următoarea linie.

Exemplu: Să se afișeze toate elementele pozitive din vectorul {7, -5, 4, 3, -9, 2, -8}.

```c
#include <stdio.h>
int main()
{
    int v[] = {7, -5, 4, 3, -9, 2, -8};
    int i, n = 7;
    for (i = 0; i < n; i++){                                    //itereaza toate elementele
        if (v[i] >= 0){                                                 //daca a gasit un element pozitiv
            printf("v[%d]=%d este pozitiv\n", i, v[i]);         //afiseaza pozitia si valoarea elementului pozitiv
        }
    }
    return 0;
}
```

Aflarea valorii maxime/minime dintr-un vector

Pentru aflarea valorii maxime/minime dintr-un vector se poate folosi următorul algoritm (particularizat pentru maxim):

Dacă vectorul este vid (nu are elemente) se tratează în mod corespunzător această situație

Altfel, dacă sunt elemente în vector:

Se consideră variabila *imax* ca fiind indexul valorii maxime și o inițializăm cu 0 (presupunem că maximul este la prima poziție în vector).

Se iterează vectorul începând cu a doua poziție

Dacă în vector la indexul curent există o valoare mai mare decât cea de la indexul *imax*, actualizează *imax* ca fiind indexul curent

În final, în *imax* se va afla indexul valorii maxime

Exemplu: Să se citească de la tastatură un număr n<=10 iar apoi *n* valori de tip float. Să se afișeze maximul elementelor citite.

```c
#include <stdio.h>
int main()
{
    float v[10];                                        // vectorul poate contine maxim 10 valori
    int n, i, imax;
    // citeste numarul de elemente
    printf("numarul de elemente: ");
    scanf("%d", &n);
    if (n == 0){                                        // testeaza daca vectorul este vid
        printf("vectorul este vid");
    }else{                                                // daca vectorul nu este vid
        for (i = 0; i < n; i++){                          // citeste pe rand elementele
            printf("v[%d] =", i);
            scanf("%g", &v[i]);
        }
        imax = 0;                                           // initializeaza indexul valorii maxime
        for (i = 1; i < n; i++){                          // itereaza de la a doua valoare
            if (v[i] > v[imax]){                                // daca a gasit o valoare mai mare,
                imax = i;                                               //actualizeaza imax cu indexul curent
            }
        }
        printf("maximul este %g", v[imax]);
    }
    return 0;
}
```

Testarea dacă o valoare există în vector

Multe probleme implică să se determine dacă o anumită valoare (dată direct sau care îndeplinește o anumită condiție) se află într-un vector. Echivalentul matematic al acestei probleme este operatorul EXISTĂ (∃). Un algoritm este următorul:

Iterează vectorul și pentru fiecare valoare:

Dacă s-a găsit valoarea cerută, întrerupe iterarea cu instrucțiunea **break**

În final, dacă variabila iterator este egală cu numărul de elemente din vector (*n*), înseamnă că valoarea cerută nu a fost găsită (iteratorul a depășit indexul ultimului element din vector, *n-1* și s-a terminat instrucțiunea *for,* deoarece condiția *i<n* nu se mai îndeplinește).
Altfel, dacă iteratorul este diferit de *n*, înseamnă că iterarea s-a oprit prin **break** și că valoarea cerută a fost găsită la poziția iteratorului.

Exemplu: Să se afișeze dacă în vectorul {7, -5, 4, 3, -9, 2, -8} există vreo valoare negativă.

```c
#include <stdio.h>
int main()
{
    int v[] = {7, -5, 4, 3, -9, 2, -8};
    int i, n = 7;
    for (i = 0; i < n; i++){                // itereaza toate elementele
        if (v[i] < 0){                          // daca gaseste o valoare negativa
            break;                                  // iese din for
        }
    }
    if (i == n){                                // i==n doar daca s-a parcurs v pana la sfarsit
        printf("vectorul nu are elemente negative\n");
    }else{                                      // i!=n daca s-a iesit cu break din iterare
        printf("vectorul are elemente negative\n");
    }
    return 0;
}
```

Testarea dacă toate valorile din vector îndeplinesc o anumită condiție

Această problemă are ca echivalent matematic operatorul ORICARE (∀). Ea se poate reduce la cea anterioară prin următoarea formulare: *„Toate elementele din vector îndeplinesc o anumită condiție dacă nu există niciun element care să nu îndeplinească condiția cerută*". Algoritmul arată astfel:

Iterează vectorul și pentru fiecare valoare:

Dacă valoarea curentă nu îndeplinește condiția dată (se testează inversa condiției date), întrerupe iterarea cu instrucțiunea **break**

În final, dacă variabila iterator este egală cu numărul de elemente din vector (*n*), înseamnă că toate valorile îndeplinesc condiția dată (dacă vreo valoare nu ar fi îndeplinit condiția, iterarea s-ar fi oprit la cel mult *n-1*).
Altfel, dacă iteratorul este diferit de *n*, înseamnă că iterarea s-a oprit prin **break** și deci există la poziția iteratorului o valoare care nu îndeplinește condiția dată.

Exemplu: Să se afișeze dacă în vectorul {7, -5, 4, 3, -9, 2, -8} toate valorile sunt negative.

```c
#include <stdio.h>
int main()
{
    int v[] = {7, -5, 4, 3, -9, 2, -8};
    int i, n = 7;
    for (i = 0; i < n; i++){            // itereaza toate elementele
        if (v[i] >= 0){                     // testeaza inversa conditiei cerute
            break;                              // iese din for
        }
    }
    if (i == n){                            // i==n doar daca s-a parcurs v pana la sfarsit
        printf("toate elementele sunt negative\n");
    }else{                                  // i!=n daca s-a iesit cu break din iterare
        printf("nu toate elementele sunt negative\n");
    }
    return 0;
}
```

Dacă dorim să scriem direct o condiție inversă fără să trebuiască să o reformulăm noi, putem folosi operatorul logic NOT (**!**). De exemplu inversa condiției *„valoare negativă”* (v[i]<0) se poate scrie **!**(v[i]<0).

Sortarea vectorilor

Elementele dintr-un vector este uneori nevoie să fie sortate după anumite criterii: în ordinea crescătoare a unor valori, alfabetic, după o dată calendaristică, etc. Pentru a putea realiza această sortare, avem nevoie de o **relație de ordine** între două elemente, de exemplu „**<**” (mai mic). Cu această relație de ordine putem aranja (sorta) elementele astfel încât ele să respecte relația dată. Relația trebuie să fie tranzitivă (dacă a<b și b<c => a<c) și atunci este suficient să ne asigurăm că oricare două elemente alăturate sunt sortate, pentru ca oricare elemente să fie sortate.

Există mulți algoritmi de sortare mai simpli sau mai complecși folosiți în funcție de situațiile specifice și de performanța necesară. Vom studia unul dintre cei mai simpli algoritmi și anume **sortarea prin metoda bulelor** (*bubble sort*). Acest algoritm are următorii pași:

Iterează vectorul și pentru fiecare pereche de două valori consecutive (o bulă):

Dacă valorile nu sunt sortate (nu respectă relația de ordine), le interschimbă între ele și setează o variabilă indicator (*flag*) pentru a memora faptul că a avut loc o schimbare în vector

După iterare, dacă a avut loc o schimbare în vector, mai repetă încă o dată iterarea vectorului (pasul 1).

Exemplu: Se citește un număr n (n<=10) și apoi *n* numere întregi. Se cere să se sorteze aceste numere în ordine crescătoare.

```c
#include <stdio.h>
int main()
{
    int v[10];
    int i, n;
    int s;                      // s(chimbare) - indicatorul (flag) daca a avut loc o schimbare
    int aux;                   // variabila auxiliara necesara la interschimbare
    printf("n =");           // citeste n
    scanf("%d", &n);
    for (i = 0; i < n; i++){          //citeste toate elementele
        printf("v[%d] =", i);
        scanf("%d", &v[i]);
    }
    do{
        s = 0;                              // setam faptul ca nu au avut loc schimbari la iterarea curenta
        for (i = 1; i < n; i++){      // @1 - pornim de la 1 ca sa formam perechi (v[i-1],v[i])
            if (v[i - 1] > v[i]){             // daca valorile NU respecta relatia de ordine
                aux = v[i - 1];                 // interschimba valorile
                v[i - 1] = v[i];
                v[i] = aux;
                s = 1;                              // @2 - seteaza faptul ca a avut loc cel putin o schimbare
            }
        }
    } while (s);                          // daca s adevarat (!=0), atunci repeta iterarea
    for (i = 0; i < n; i++){          // afiseaza vectorul
        printf("%d\n", v[i]);             // cate o valoare pe linie
    }
    return 0;
}
```

Sortarea propriu-zisă are loc în bucla *do...while*. Se folosește o buclă *do...while* deoarece la fiecare iterare mai întâi trebuie determinat dacă vectorul este sau nu sortat și abia apoi se va ști dacă se va repeta iterarea sau nu. Variabila *s* este folosită ca indicator (*flag*). Variabilele indicator au o valoare logică (booleană), *adevărat* sau *fals*. Pentru acest algoritm nu ne interesează să știm câte interschimbări au avut loc la o iterare, ci doar dacă au avut loc asemenea interschimbări.

În linia *@1* pornim cu iterarea de la i=1 deoarece formăm perechi (v[i-1],v[i]) (ex: (v[0],v[1]), (v[1],v[2]),...(v[n-2],v[n-1])). Aceste perechi se numesc *bule* (*bubble*) și de aici vine numele algoritmului. Dacă oricare dintre aceste perechi nu respectă relația de ordine, atunci interschimbăm elementele ei.

Pentru a interschimba valorile a două variabile *x* și *y* avem nevoie de o a treia variabilă (auxiliară), deoarece dacă am scrie direct „*x=y; y=x;*", atunci prima oară *y* s-ar depune în *x* (s-ar pierde vechea valoare a lui *x*) și în final ambele variabile ar avea valoarea *y*. Rezultă că avem nevoie prima oară să memorăm valoarea lui *x* altundeva, pentru a nu o pierde: „*aux=x; x=y; y=aux;*". Situația este analogică aceleia în care cineva are în ambele mâini câte o minge și vrea sa le interschimbe între ele. Pentru aceasta, prima oară trebuie să pună o minge jos (în alt spațiu de stocare) și abia apoi poate să transfere cealaltă minge în mâna rămasă goală. În final va ridica mingea de jos în mâna din care a transferat mingea.

Dacă a avut loc o interschimbare, în linia @2 se setează variabila indicator pe valoarea *adevărat*, adică „*a avut loc o interschimbare*". În acest caz, condiția de continuare a lui **while** este adevărată și se va mai repeta încă o dată tot ciclul. Această repetare trebuie să aibă loc, deoarece nu se știe încă dacă valorile interschimbate au ajuns la pozițiile lor finale sau ele trebuie mutate (propagate) și mai multe poziții către stânga sau dreapta.

Pentru numerele {7, 1, 3, 2} au loc următoarele operații (<-> înseamnă interschimbare):

iterarea 1:

7<->1 -> {1, 7, 3, 2}

7<->3 -> {1, 3, 7, 2}

7<->2 -> {1, 3, 2, 7}

au avut loc interschimbări, deci se mai repetă o dată. Iterarea 2:

3<->2 -> {1, 2, 3, 7}

au avut loc interschimbări, deci se mai repetă o dată. Iterarea 3:

(nicio interschimbare)

nu au mai avut loc interschimbări, deci algoritmul s-a terminat

Se constată că practic algoritmul propagă valorile mai mari către dreapta, pe măsură ce iteratorul *i* avansează.

Acest algoritm este unul general, în sensul că dacă schimbăm relația de ordine, atunci se schimbă și ordinea de sortare. Toți ceilalți pași rămân la fel, modificându-se doar condiția din **if**. Această condiție trebuie să fie negarea (**!**) relației de ordine dorite, deoarece în acest caz trebuie să interschimbăm elementele.

Ștergerea de elemente

Dacă se dorește să se șteargă unele elemente dintr-un vector, se poate proceda în două feluri:

Dacă nu contează ordinea elementelor, atunci se poate lua elementul de la sfârșitul vectorului și pune peste cel care se dorește a fi șters

Dacă contează ordinea elementelor, atunci toate elementele de după cel de șters vor trebui mutate (deplasate, shiftate, din engleza "*to shift*") cu o poziție la stânga, acoperindu-l pe cel de șters

În ambele situații trebuie actualizat numărul de elemente din vector, acesta scăzând cu o unitate.

Dacă la aceeași iterație se șterg mai multe elemente, trebuie ținut cont de faptul că peste elementul șters se va afla un nou element, necunoscut. Dacă iteratorul merge mai departe după ștergere, fără a verifica din nou elementul de la poziția la care a avut loc ștergerea, acest nou element va rămâne neverificat, ceea ce poate rezulta în potențiale erori.

Pentru a evidenția această posibilă eroare care apare când nu retestăm valoarea de la poziția la care a avut loc ștergerea, considerăm că avem de șters elementele negative din vectorul {-4, 7, -3}, fără a ține cont de ordine. Când iteratorul este la indexul 0, vom șterge -4 punând în locul lui pe -3. Iteratorul va deveni apoi 1 (adică poziția lui 7) și -3 care a fost pus în locul lui -1 va rămâne în vector, deoarece valoarea de la poziția 0 nu este retestată.

Din acest motiv, dacă s-a făcut o ștergere într-o iterație, iteratorul va trebui dat cu o poziție înapoi (ca împreună cu incrementarea finală din **for** să rămână pe loc), pentru a se testa și valoarea elementului care a luat locul celui șters.

Exemplu: Să se șteargă din vectorul {5, 8, 1, 4, 2, 6, 9} toate elementele pare, păstrând neschimbată ordinea elementelor, după care să se afișeze noul vector.

```
#include <stdio.h>
int main()
{
    int v[]={5, 8, 1, 4, 2, 6, 9};
    int i, j, n=7;
    for (i = 0; i < n; i++){
        if(v[i]%2 == 0){                    // test daca valoare para
            for(j = i+1; j<n; j++){            // muta la stanga toate elementele de dupa v[i]
                v[j-1] = v[j];
            }
            n--;                                  // lungimea vectorului scade cu o unitate
            i--;                                   // @1 - revine cu o pozitie pentru a se testa noua valoare pusa la v[i]
        }
    }

    for (i = 0; i < n; i++){                // afiseaza vectorul
        printf("%d\n", v[i]);                  // cate o valoare pe linie
    }
    return 0;
}
```

Vectorul final va avea valorile {5, 1, 9}. Ce vector ar rezulta dacă ar lipsi linia @1?

Inserarea de elemente

Pentru a se insera un element între alte elemente, păstrându-se ordinea elementelor, într-o primă fază elementele începând cu poziția unde se face inserarea vor trebui deplasate cu o poziție la dreapta, pentru a se face loc elementului de inserat. După inserare, numărul de elemente crește cu o unitate. Trebuie avut grijă ca vectorul să aibă spațiu pentru valoarea nou introdusă.

Atenție: deplasarea elementelor trebuie făcută de la dreapta la stânga (de la sfârșit către început).

Dacă deplasarea s-ar face de la stânga la dreapta, atunci de fapt s-ar propaga valoarea de la poziția de inserat în toate pozițiile din dreapta ei. De exemplu, dacă avem vectorul {3, 7, 5} și vrem să inserăm 9 la indexul 1, mai întâi va trebui să mutăm pe 7 și pe 5 cu o poziție la dreapta. Dacă facem mutarea de la stânga la dreapta, atunci 7 va veni peste 5, ștergându-l din vector, iar apoi tot 7 (noua valoare pusă peste 5) va fi propagată până la sfârșitul vectorului, rezultând după inserarea lui 9 {3, 9, 7, 7}. Din acest motiv deplasarea trebuie realizată de la dreapta la stânga, mutându-l mai întâi pe 5 la dreapta și apoi pe 7 în vechea poziție a lui 5.

Exemplu: Se dă vectorul {7, -5, 4, 3, -9, 2, -8}. Să se insereze înainte de fiecare valoare din vectorul original negativul ei.

```
#include <stdio.h>
int main()
{
    int v[14] = {7, -5, 4, 3, -9, 2, -8};                // in final in vector vor fi 14 elemente
    int i, j, n = 7;
    for (i = 0; i < n; i++){
        for (j = n; j > i; j--){                                    // @1 - itereaza de la dreapta la stanga
            v[j] = v[j - 1];                                            // muta la dreapta cu o unitate, inclusiv v[i]
        }
        // acum in v[i] e vechea valoare, duplicata si la v[i+1]
        v[i] = -v[i];                                                 // seteaza v[i] ca fiind negativul ei
        n++;                                                         // lungimea vectorului creste cu o unitate
        // deoarece la v[i+1] este vechea valoare de la v[i],
        // pentru a nu mai fi procesata inca o data la urmatoarea iteratie (i+1),
        // se sare peste aceasta pozitie
        i++;
    }
    for (i = 0; i < n; i++){
        printf("%d\n", v[i]);
    }
    return 0;
}
```

La fiecare iterație, după ce s-au deplasat elementele începând cu poziția *i* la dreapta, v[i] se va afla și la v[i+1]. La următoarea iterație, când *i* va fi *i+1*, rezultă că această valoare va fi procesată încă o dată și tot așa la infinit. Pentru a se evita aceasta, după inserarea unei valori, se incrementează *i*, astfel încât împreună cu incrementul de la **for**, noua valoare a lui *i* de fapt va fi i+2, trecându-se astfel direct la v[i+2].

Observație: în general, când gândim un algoritm cu vectori, este foarte bine să verificăm dacă o iterare cuprinde primul și ultimul element de procesat. Pentru aceasta, analizăm ce se petrece pentru prima și ultima poziție a iteratorului.

De exemplu, în linia @1, *j* merge de la *n* la i+1 inclusiv. Pentru j==n, avem v[n]=v[n-1], adică ultima valoare din vector va fi pusă la prima poziție liberă de după vector. Pentru j==i+1, avem v[i+1]=v[i], adică mută la dreapta inclusiv elementul de la v[i].

Operațiuni de elementare de citire/scriere caractere

Fiecare program are acces implicit la 3 "fișiere" pentru operațiuni de intare ieșire (scriere/citire de la tastatura/ecran):

- **stdin** – standard input – intrarea standard a programului – toate funcțiile standard de intrare vor citi implicit de la intrarea standard (se poate spune aproximativ că vor citi, generic, de la tastatură)
- **stdout** – standard output – ieșirea standard a programului – toate funcțiile standard de ieșire vor scrie implicit la ieșirea standard (se poate spune aproximativ ca vor scrie pe ecran, ca vor tipări)
- **stderr** – standard error – ieșirea standard de eroare a programului – similar cu stdout dar se folosește pentru a "tipări" erori

Momentan, putem considera faptul că *stdout* reprezintă ecranul sau terminalul iar *stdin* reprezintă tastatura. Practic putem spune cu ușurință că prin "printăm la stdout (standard output, ieșirea standard)" de fapt scriem pe ecran sau scriem în terminal. De asemenea, putem spune că prin "citim de la stdin (standard input, intrarea standard")" de fapt citim de la tastatură.

Aceste fișiere speciale (stdin, stdout, stderr) sunt deschise și disponibile pentru fiecare program pe tot parcursul execuției programului într-un mod transparent pentru programator. Acestea sunt gestionate de sistemul de operare iar programatorul practic nu trebuie să facă nimic în privința lor.

Este important de menționat că atât intrarea standard cât și ieșirea standard nu sunt "conectate" direct la programele ce rulează ci între acestea sistemul de operare intervine cu serie de memorii tampon, buffere.
Așadar, atunci când un program utilizator scrie la ieșirea standard, datele ajung de fapt într-un buffer intern al sistemului de operare și acesta din urmă decide când anume acele date vor fi efectiv scrise la standard output.
La fel se întâmplă și în cazul intrării standard și anume atunci când scriem la tastatură datele ajung într-un buffer intern al sistemului de operare înainte să ajungă la programul nostru și doar sistemul de operare decide când va transfera datele către programul nostru.

Decizia sistemului de operare de trimiterea datelor din bufferele intermediare efectiv către stdout sau stdin se întâmplă prin golirea acestor buffere iar acest procedeul se cheamă flush (adica golirea bufferelor). Momentul în care apare operațiunea de flush a fișierelor stdin și stdout este decis de sistemul de operare. Decizia este de obicei luată de sistemul de operare în momentul în care buffer-ul se umple sau când primește anumite cereri din partea programului.

Operațiunea de flush a bufferelor de la stdin și stdout este determinată de următorii factori:

- umplerea bufferelor
- caracterul \n
- caracterul EOF

Acest lucru poate implica anumite comportamente. Spre exemplu, daca scriem un mesaj cu funcția printf acesta nu va fi scris în exact momentul apelului funcției printf ci ori când se umple bufferul de la stdout ori când se trimite către stdout caracterul \n. Ca și o consecință, dacă dorim ca mesajul nostru să ajungă la stdout în exact acel moment este necesar ca la sfârșitul mesajului să fortăm o linie nouă punând și caracterul \n.

Același lucru se întâmplă și în cazul intrării standard stdin. Dacă scriem ceva la tastatură, programul nostru va primi datelor ori în cazul în care se umple bufferul sistemului de operare, ori când trimitem caracterul \n prin tasta ENTER ori când trimitem caracterul EOF prin combinația tastelor CTRL+D (în Linux).

Este bine ca aceste aspecte să fie luate în calcul atunci când se implementează și se testează programele.

O importantă observație este faptul că fișierul stderr (ieșirea standard de eroare) este conectată direct la programele utilizatorului fără buffere intermediare.

Biblioteca standard C pune la dispoziție 2 funcții pentru citire/scriere elementară de caractere.

Funcția putchar(int c) primește ca argument caracterul pe care îl va scrie la stdout, în cazul nostru, în terminal.

Funcția getchar(int c) are rolul de a citi un caracter de la intrarea standard (stdin), în cazul nostru de la tastatură. Această funcție returnează caracterul citit ca și un întreg cu semn (int) și valoarea EOF (End Of File) în caz de eroare sau în cazul în care fișierul stdin nu mai este disponibil. Funcția aceasta este cu blocare în sensul că la un apel ea nu returnează până ce nu citește un caracter de la stdin sau până găsește caracterul EOF. Caracterul EOF poate fi obținut de la tastatură prin combinația tastelor CTRL+D pentru sisteme Linux și CTRL+Z pentru sisteme Windows.

Considerăm următorul exemplu, în care se implementează un program ce citește caracter cu caracter de la intrarea standard (stdin) și îl afișează la ieșirea standard:

```
#include <stdio.h>
int main(void)
{
    int c = 0;
    while ((c = getchar()) != EOF)
    {
        putchar(c);
    }
    return 0;
}
```

Testarea programului se poate face astfel: după rulare programul așteaptă introducerea de caractere. Se pot introduce caractere dar acestea nu sunt „trimise” programului decât ori după un anumit număr de caractere ori după introducerea caracterului \n. Așadar, pentru testare, se pot introduce caractere iar la următoarea apăsare a tastei ENTER bucla while va fi executată și caracterele introduse vor fi scrise la stdout. Programul se va termina doar la recepționarea caracterului EOF. Acesta poate fi provocat din tastatură prin CTRL+D în sisteme Linux.

Operațiuni de redirectare

Sistemul de operare Linux oferă o serie de mecanisme prin care utilizatorul poate foarte ușor să înlocuiască ieșirea standard sau intrarea standard cu fișiere fără ca programul să “știe” acest aspect. Acest lucru se poate realiza prin semnele “<” si “>” folosite în terminal.

Redirectarea ieșirii standard într-un fișier. Exemplu:

```
valy@staff:~$ ./program > file.txt
```

În această situație, absolut orice este scris de către programul *program* la ieșirea standard (stdout) prin apelul oricărei funcții precum printf sau putchar va ajunge de fapt în fișierul file.txt . Dacă fișierul nu există sistemul de operare în va crea iar dacă există atunci conținutul acestuia va fi sters.

Redirectarea intrării standard dintr-un fișier existent

```
valy@staff:~$ ./program < file.txt
```

În această situație, către ieșirea standard a programului *program* va fi redirectat conținutul fișierului file.txt. Astfel, orice operațiune de citire de la intrarea standard (stdin) prin apelul funcțiilor precum scanf sau getchar() va citi practic din acest fișier. Când programul va citi tot fișierul, acesta va primit caracterul EOF. Este absolut necesar ca fișierul ce este redirectat la intrarea standard a unui program să existe, în caz contrar, sistemul de operare va genera un mesaj de eroare în terminal și nu va lansa în execuție programul

Este important de menționat că programul “nu stie” faptul că i s-au redirectat fișierele standard de intrare și de ieșire. Deci programul “nu știe” că în urma unei scrieri la stdout el defapt scrie într-un fișier.

Acest lucru poate fi folosit ca un mare avantaj deoarece se poate ușura considerabil procedura de testare a programelor ce lucrează cu standard output și standard input.

Putem aplica aceste considerente în testarea programului dezvoltat anterior. În locul testării clasice de la tastatură putem redirecta către stdin un fișier de test realizat în prealabil. Testarea deci se poate realiza extrem de util astfel:

```
valy@staff:~$ gcc -Wall -o program in_out_2.c
```

```
valy@staff:~$ ./program < testfile.txt
```

◄ Test grilă 1

Sari la...

Teme și aplicații ►