Incrementare și decrementare

În programare, operațiile de adunare a unei unități la un număr (incrementare) sau de scădere a ei (decrementare) sunt foarte des folosite, astfel încât în C avem operatori speciali pentru ele: **++** (incrementare) și **--** (decrementare).
Operatorii de incrementare se pot folosi în două forme: prefixat și postfixat. Numele acestor forme se referă la *când anume se aplică operatorul*: înainte sau după ce s-a folosit valoarea operandului:

Formă	Sintaxă	Efect
prefixat	++x --x	Prima oară se aplică operatorul asupra lui *x*, iar apoi se returnează noua valoare a lui *x*
postfixat	x++ x--	Prima oară se returnează valoarea curentă a lui x și după aceea se aplică operatorul

Exemplu: se consideră secvența de program de mai jos. Să se determine ce va afișa pe ecran fiecare *printf*.

```
int a=3,b=7,c;
c=a++ + ++b;
printf("%d %d %d\n",a,b,c);// 4 8 11
c=--a + --b;
printf("%d %d %d\n",a,b,c);// 3 7 10
c=--a-b--;
printf("%d %d %d\n",a,b,c);// 2 6 -5
c=a-- - --b;
printf("%d %d %d\n",a,b,c);// 1 5 -3
```

Instrucțiunea de selecție: **switch**

Uneori este nevoie să testăm o valoare de tip întreg, care aparține unei mulțimi specifice, cu relativ puține valori. De exemplu, considerăm că avem un meniu din care un utilizator poate să selecteze o opțiune, introducând codul ei. După ce utilizatorul a introdus opțiunea, programul trebuie să realizeze anumite acțiuni în funcție de opțiunea introdusă. Pentru această situație este mai simplu să folosim instrucțiunea **switch**, în loc de o serie de **if**..**else if**…**else if**…**else**, care testează toate valorile posibile pentru variabila de test.
Instrucțiunea **switch** are următoarea sintaxă:

```
switch(expresie) {
   case valoare_1:
      instrucțiuni1;
      break;
   case valoare_2:
      instrucțiuni2;
      break;
   ...
   default:
      instrucțiuni_default;
}
```

Instrucțiunea **switch** compară valoarea expresiei pe rând, de sus în jos, cu valorile de la fiecare **case**. Dacă un **case** are aceeași valoare cu expresia, atunci se vor executa instrucțiunile corespunzătoare acelui **case**. Dacă în interiorul unui **case** se întâlnește instrucțiunea **break**, atunci **switch** se termină. Dacă nu se întâlnește **break**, execuția continuă cu instrucțiunile **case**-urilor de dedesubt și cu instrucțiunile de la **default**. **break** poate apărea de mai multe ori în interiorul unui **case**, cu același efect, de terminare a instrucțiunii **switch**.
default este opțional și se execută atunci când valoarea expresiei nu este egală cu niciuna dintre valorile **case**-urilor (analogic lui **else** de la **if**).
Programul următor afișează numărul de zile din fiecare lună, pentru anii care nu sunt bisecți:

```
#include <stdio.h>
```

```c
int main() {
    int luna, nr_zile;
    printf("luna: "); scanf("%d", &luna);
    switch(luna) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            nr_zile = 31;
            break;
        case 2:
            nr_zile = 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            nr_zile = 30;
            break;
        default:
            nr_zile = -1;
    }
    if (nr_zile != -1) {
        printf("Luna aleasa are %d zile", nr_zile);
        } else {
        printf("Ati introdus gresit luna!");
    }
    return 0;
}
```

În acest exemplu se constată faptul că s-au putut grupa mai multe **case**-uri care se tratează în același fel. De exemplu, pentru valorile 1, 3, 5, 7, 8 și 10, *n* trebuie să ia valoarea 31 și atunci acestea au putut fi grupate fără **break** între ele. Dacă utilizatorul a introdus greșit numărul lunii, *nr_zile* se va seta cu o valoare din afara domeniului obișnuit de valori, valoare care este folosită ulterior pentru a se testa cazul de eroare.

Instrucțiunea **while**

De multe ori este necesar să repetăm (iterăm, ciclăm) de mai multe ori aceeași secvență de cod. De exemplu, dacă este necesar să afișăm toate puterile lui 2 care sunt mai mici decât un *n* dat de la tastatură {2i | i>=0, 2i<n}, un algoritm ar arăta ca mai jos:

citește *n* de la tastatură
inițializează *v* cu 1 (20)
atâta timp cât *v* este mai mic decât *n*, repetă
3.1 afișează *v*
3.2 înmulțește pe *v* cu 2 (pentru a obține următoarea putere a lui 2)

Acest algoritm se poate implementa în C astfel:

```c
#include <stdio.h>
int main() {
    int n, v;
    printf("n: ");
    scanf("%d", &n);
    v = 1;
    while (v < n) {
        printf("%d\n", v);
        v = v * 2;
    }
    return 0;
```

```
}
```

Instrucțiunea **while** are următoarea sintaxă:
while(conditie_de_iterare) instrucțiune;
și se citește astfel:
ATÂTA TIMP CÂT (while) conditie_de_iterare este adevărată (!=0), execută instructiune
La fel ca în cazul lui ***if***, dacă dorim ca mai multe instrucțiuni să depindă de ***while***, ele vor trebui incluse între acolade, ca în exemplul de mai sus, în care a fost nevoie ca două instrucțiuni să depindă de ***while***.
Deoarece *v* este o variabilă, i se poate modifica valoarea oricând este nevoie. În acest caz, *v* este actualizat cu o nouă valoare la fiecare iterație. Cum s-a discutat la operatori, se poate folosi forma compusă de atribuire: v*=2.

Unele sugestii pentru formatarea codului

Din cauză că programele C vor începe să devină din ce în ce mai mari și vor exista situații în care unele instrucțiuni să depindă de altele pe mai multe niveluri de adâncime (imbricare), este necesar să se aplice anumite reguli de formatare a programului, întocmai precum un document bine formatat trebuie să aibă clar definite titluri, subtitluri, paragrafe, etc. Deși aceste reguli nu sunt necesare pentru compilatorul de C în sine (deoarece în C nu contează numărul de spații sau de linii), ele sunt totuși foarte utile pentru programator, pentru a putea urmări mai ușor programul. În continuare sunt expuse câteva astfel de reguli:
Dacă unele instrucțiuni depind de altele (de exemplu toate instrucțiunile care depind de un anumit ***if***), acestea vor fi indentate la dreapta față de instrucțiunea de care ele depind
Pentru indentare se vor sau TAB-uri sau spații, dar nu se vor mixa TAB-uri cu spații
Înainte de o acoladă închisă și după o acoladă deschisă nu se pune o instrucțiune pe aceeași linie
Dacă după un ***if***, ***while***, etc urmează o singură instrucțiune simplă (nu un alt ***if***, ***while***) care depinde de acestea, atunci ea se poate pune pe aceeași linie cu ele; altfel se aplică regulile de mai sus cu acoladele
Toate instrucțiunile din interiorul unei funcții (ex: *main*) vor fi indentate
Toate ***case***-urile vor fi indentate față de ***switch***-ul care le conține. Instrucțiunile acestor ***case***-uri vor fi și ele indentate față de ***case***-urile lor. Dacă un ***case*** are mai multe instrucțiuni, ele vor începe de pe linia următoare ***case***-ului.
Pentru simetrie, dacă instrucțiunile unui ***if*** se pun între acolade și instrucțiunile ***else***-ului atașat se vor pune între acolade, deși poate nu este nevoie, și invers. Pentru un singur ***if***, ***else***-ul lui și toate instrucțiunile aferente se indentează.
Dacă există o succesiune de ***if***, ***else if***, ***else if***,…, fiecare ***else if*** va începe pe aceeași coloană cu primul ***if***.
După câteva programe scrise respectând aceste reguli, ele vor deveni un reflex și vor ajuta mult la inteligibilitatea codului sursă. Fiecare programator va putea în continuare, conform propriei sale experiențe, să țină cont și de alte reguli de formatare, care să i se potrivească în mod specific. Cel mai important este ca regulile folosite să fie aplicate în mod consecvent.
Instrucțiunea **for**
Foarte multe iterații se fac într-un domeniu bine definit, de exemplu luând pe rând (iterând) valorile din intervalul [0,n). Acest caz, folosind **while**, se poate scrie:

```
i=0;
while(i<n){
    instructiune;
    i=i+1;
}
```

Se constată că avem nevoie de trei elemente standard:
o instrucțiune prin care să inițializăm variabila de iterat (iteratorul): *i=0*
o condiție de continuare a iterării: *i<n*
la sfârșitul fiecărei iterații o instrucțiune care să actualizeze iteratorul cu o nouă valoare: *i=i+1*
Deoarece acest caz este foarte frecvent, s-a prevăzut o instrucțiune specială, **for**, care combină toate aceste elemente:
for (instr_de_initializare; conditie_de_continuare; instr_de_actualizare) instructiune;
Între fiecare dintre cele trei elemente trebuie să existe *punct și virgulă* (**;**). Fiecare dintre cele trei elemente este opțional (poate lipsi), chiar și toate trei simultan, dar cele două *punct și virgulă*

trebuie să existe. Dacă lipsește *conditie_de_continuare*, atunci condiția ***for***-ului se consideră tot timpul adevărată, astfel încât va rezulta o buclă infinită: ***for****(;;){...}* va repeta la infinit instrucțiunile asociate.
Folosind ***for*** și ținând cont că *i=i+1* de fapt înseamnă incrementarea lui *i*, exemplul de mai sus se poate scrie mult mai concis:

```
for(i=0;i<n;i++) {
    instructiune;
}
```

În acest caz, nu contează dacă folosim incrementare prefixată sau postfixată, deoarece *i++* este o expresie a cărei valoare returnată nu este folosită ulterior.
În informatică o expresie poate avea două tipuri de efecte:
să furnizeze o valoare, la fel ca în matematică. De exemplu, în instrucțiunea *printf("%d",x*3-1);* expresia *x*3-1* furnizează o valoare pentru *printf*
să modifice valorile unor variabile, să afișeze ceva, sau în general să influențeze alte aspecte ale programului, în afara valorii returnate. Aceste efecte se numesc **efecte laterale** (*side effects*). De exemplu, în instrucțiunea *printf("%d",i++);* expresia *i++*, pe lângă valoarea returnată de ea și care este folosită în *printf*, mai are ca efect lateral incrementarea variabilei *i*.
În exemplul de mai sus cu ***for***, instrucțiunea *i++* este folosită doar pentru efectul ei lateral, de incrementare a lui *i*, fără ca valoarea acestei expresii să fie folosită la ceva.
Exemplu: Se citește un număr *n*, care semnifică numărul de note primite. Se vor citi ulterior *n* note (numere reale) și se va afișa media lor aritmetică.

```
#include <stdio.h>
int main() {
    int i, n;
    float sum = 0, k;
    printf("numarul de note: ");
    scanf("%d", &n);
    for (i = 0; i <n; i++) {
        printf("introduceti nota %d: ", i);
        scanf("%g", &k);
        sum += k;
    }
    printf("media este: %g", sum / n);
    return 0;
}
```

Limbajul C permite ca la declararea unei variabile să-i atribuim și o valoare inițială. *float sum=0;* este echivalent cu: *float sum; sum=0;*

Instrucțiunea **do...while**

Instrucțiunile ***while*** și ***for*** întotdeauna testează condiția lor înainte de a se executa o iterație. Dacă condiția nu este de la început adevărată, atunci corpul acestor instrucțiuni nu se va executa deloc. Uneori este necesar să se execute corpul unei instrucțiuni repetitive măcar o dată, chiar dacă condiția de repetiție este falsă. În această situație se folosește instrucțiunea:

```
do{
    instructiuni;
}while(conditie_de_continuare);
```

Instrucțiunea ***do…while*** are testul la sfârșit și atunci instrucțiunile se vor executa înainte de a fi verificată condiția de continuare. Astfel, un ***do...while*** se va executa cel puțin o dată.
Exemplu: Să se afișeze în ordine inversă toate cifrele unui număr întreg.

```
#include <stdio.h>
int main() {
    int n, k;
    printf("n: ");
    scanf("%d", &n);
    do {
        k = n % 10;       // se izoleaza ultima cifra
        printf("%d\n", k);
```

```
        n = n / 10;          // se sterge ultima cifra din n
    } while(n);
    return 0;
}
```

În programul de mai sus, *n* este folosit ca și condiție de continuare, conform regulii că orice valoare diferită de 0 se consideră ca fiind adevărată. Dacă s-ar fi folosit ***while*** în loc de ***do…while***, cu testarea pentru *n* la începutul iterației, atunci pentru *n==0* nu s-ar fi afișat nimic, deoarece condiția din ***while*** ar fi fost falsă și nu ar fi avut loc nicio iterație. Cu ***do…while*** se va afișa *0*, deoarece condiția se va testa după prima iterație.
Instrucțiunile de iterare pot fi imbricate unele în altele.
Exemplu: Se cere un n>0. Să se afișeze pe prima linie o steluță, pe a doua linie două steluțe și tot așa până la linia *n* inclusiv.

```
#include <stdio.h>
int main() {
    int n, i, j;
    printf("n: ");
    scanf("%d", &n);
    for (i = 0; i <n; i++) {        // for exterior
        for (j = 0; j <i; j++) {        // for interior
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

În ***for***-ul exterior, variabila *i* este folosită pentru a se itera fiecare linie de afișat. Deoarece la fiecare linie trebuie să existe un număr de steluțe egal cu numărul liniei, se folosește pentru fiecare linie un alt ***for*** (cel interior) care afișează numărul necesar de steluțe pentru fiecare linie. După ce s-au afișat steluțele unei linii cu ***for***-ul interior, se trece cursorul la o linie nouă prin afișarea caracterului \n. Acest ciclu se continuă atâta timp cât mai sunt linii de afișat.

Controlul iterațiilor cu **break** și **continue**

Instrucțiunea ***break*** folosită în interiorul unei bucle ***for***, ***while*** sau ***do…while*** provoacă ieșirea imediată din această buclă. De exemplu, putem implementa un joc simplu, în care jucătorul introduce două numere reale și apoi trebuie să ghicească suma lor. Dacă a dat un răspuns greșit, va fi întrebat încă o dată, până când va da răspunsul corect.

```
#include <stdio.h>
int main() {
    float x, y, r;
    printf("x: ");
    scanf("%g", &x);
    printf("y: ");
    scanf("%g", &y);
    for(;;) {        // bucla infinita
        printf("%g+%g=", x, y);
        scanf("%g", &r);
        if (r == x + y) {
            printf("Felicitari, ati raspuns corect\n");
            break;
        }
        printf("Raspuns gresit\n");
    }
    return 0;
}
```

În bucla infinită ***for(;;){...}***, la fiecare iterație se cere un răspuns *r*. Dacă acesta este corect, atunci se afișează mesajul de felicitare și apoi cu ***break*** se iese imediat din buclă. Altfel, se afișează mesajul de eroare și se repetă ciclul. Se observă că nu a mai fost nevoie să se pună ***else*** după ***if***,

deoarece, dacă se intră în ***if***, atunci în mod automat se va încheia ciclul infinit și execuția programului va continua după *for*.
Atentie: dacă într-un ***while***, ***for***, ***do…while*** se află o instrucțiune ***switch***, atunci ***break***-urile folosite în interiorul ***switch***-ului vor provoca ieșirea doar din ***switch***, nu și din instrucțiunea de ciclare (un **break** într-un **switch se consideră ca aparținând de switch)**. În mod analogic, dacă există două instrucțiuni de ciclare imbricate una în cealaltă, instrucțiunea ***break*** folosită în bucla interioară va provoca ieșirea doar din această buclă, nu și din cea exterioară.
Uneori este necesar să se evite anumite părți dintr-o instrucțiune de ciclare, trecându-se la următoarea iterație. De exemplu, dacă implementăm un calculator care cere două numere iar apoi afișează rezultatul împărțirii acestora, ar fi bine să evităm situația în care apare împărțire la 0, dar totuși să nu încheiem iterațiile. În acest caz se poate folosi instrucțiunea ***continue***, care are ca efect trecerea la următoarea iterație, fără a se mai executa instrucțiunile de după ea din buclă.

```
#include <stdio.h>
int main() {
    float x, y;
    for (;;) {
        printf("x: ");
        scanf("%g", &x);
        if (x == 0) break;            // conditie de terminare
        printf("y: ");
        scanf("%g", &y);
        if (y == 0) continue;         // evita impartirea la zero
        printf("%g/%g=%g\n", x, y, x/y);
    }
    return 0;
}
```

Operatorul secvențial (virgulă)
Operatorul secvențial (virgula folosită între expresii, în afara apelurilor de funcții) are formatul:
expresie1 **,** expresie2 **,** … **,** expresien
și are următorul efect: se evaluează expresiile în ordine, de la stânga la dreapta, și în final se returnează rezultatul ultimei expresii.
Exemplu:
Instrucțiunea a=i++, j+1; Conform precedenței operatorilor (vezi lab anterior):
Prima dată se evaluează i++, deoarece operatorul ++ are cea mai mare precedență. Rezultatul subexpresiei i++ este i;
Apoi, i se va incrementa.
Atribuirea are precedență față de operatorul de secvență, deci a ia (fosta) valoare a lui i. Expresia a=i++ are și ea ca valoare i.
Apoi, operatorul de adunare are precedență față de secvență, deci se calculează j+1.
Apoi se evaluează secvența, rezultând ca întreaga linie de cod, fiind o expresie, să aibă ca valoare j+1. Cu alte cuvinte, dacă scriam b = (a=i++, j+1);, b lua valoarea lui j+1.
Exemplu:
Instrucțiunea *a=(i++, j+1);* va avea, în ordine, următoarele efecte:
evaluează variabila *i* ca fiind rezultatul primei expresii a operatorului secvențial
incrementează variabila *i*
evaluează expresia *j+1* ca fiind rezultatul celei de a doua expresii a operatorului secvențial
atribuie lui *a* valoarea evaluată anterior a ultimei expresii a operatorului secvențial (*j+1*)
Se poate constata că folosind operatorul secvențial, toate rezultatele expresiilor *1...n-1* se pierd, returnându-se în final doar rezultatul ultimei expresii. Din acest motiv, expresiile *1...n-1* se folosesc doar pentru efectele lor laterale (ex: incrementări de variabile, atribuiri) și nu pentru valoarea lor propriu-zisă.
De obicei operatorul secvențial se folosește în locurile în care sintaxa limbajului specifică faptul că se poate folosi o singură expresie (de exemplu la partea de inițializare a instrucțiunii *for*), dar noi dorim sa folosim mai multe expresii.
Considerăm programul următor, care afișează seria lui Fibonacci (f(0)=0; f(1)=1; f(n)=f(n-1)+f(n-2)) pentru un număr *n*.

```
#include <stdio.h>

int main() {
```

```
    int i, n;
    int f, fm1, fm2;        // f(i), f(i-1), f(i-2)
    printf("n: ");
    scanf("%d", &n);
    if(n==0){
        f=0;
    }
    else if(n==1){
        f=1;
    }
    else{       // n>1
        fm2=0;
        fm1=1;
        for (i = 2; i <= n; i++) {
            f=fm1+fm2;
            fm2=fm1;
            fm1=f;
        }
    }
    printf("Fibonacci(%d)=%d\n",n,f);
    return 0;
}
```

Se poate constata că pe ramura pentru *n>1*, este nevoie de mai multe inițializări, înainte de ***for***. Tot pe această ramură, la fiecare iterație se fac mai multe actualizări. Folosind operatorul secvențial, în cazul extrem putem să comasăm toate aceste inițializări și atribuiri într-o singură instrucțiune, programul devenind:

```
#include <stdio.h>

int main() {
    int i, n;
    int f, fm1, fm2;        // f(i), f(i-1), f(i-2)
    printf("n: ");
    scanf("%d", &n);
    if(n==0){
        f=0;
    }
    else if(n==1){
        f=1;
    }
    else{       // n>1
        for (fm2=0, fm1=1, i = 2; i <= n; f=fm1+fm2, fm2=fm1, fm1=f, i++) {}
    }
    printf("Fibonacci(%d)=%d\n",n,f);
    return 0;
}
```

Se poate constata că s-au putut include toate instrucțiunile de pe ramura n>1 în interiorul lui ***for***, ceea ce a compactat foarte mult codul (s-a redus la o linie în loc de 7). În acest caz, în interiorul instrucțiunii ***for*** nu a mai fost nevoie de nicio instrucțiune, dar, din cauză că sintaxa lui ***for*** cere o instrucțiune subordonată, s-a folosit blocul vid, {}. Se putea folosi după *for* și *punct și virgulă* (*for(...);*), care are tot semnificația de instrucțiune vidă, dar {} este mai lizibil.

Deși operatorul secvențial face programele mai compacte, lizibilitatea poate să scadă. Din acest motiv se va folosi doar în locurile în care este necesar din punct de vedere sintactic (sintaxa permite o singură expresie dar noi avem nevoie de mai multe) sau atunci când între expresii există o legătură semantică evidentă.

Atenție: a nu se confunda virgula folosită ca operator secvențial cu virgula folosită ca separator între argumente, la apelurile de funcții. De exemplu, în expresia *max(a,b)* virgula este separator între cele două argumente folosite la apelul funcției *max*.