Instrucțiuni repetitive

Legat de **datele** cu care lucrează programele, au fost prezentate categoriile principale de operatori: aritmetici (+, -, *, /, %), de atribuire (=, +=, -=, *=, /=, %=, ++, --), relaționali (<, >, <=, >=, ==, !=) și cei logici (!, &&, ||). De asemenea, operatorii: de conversie, virgulă și cel ternar. Avem așadar deja la dispoziție un set variat de operații cu ajutorul cărora să putem obține noi date din valorile curente ale datelor din program. Avem în plus la dispoziție funcțiile predefinite ca o puternică extensie a setului de operatori.

Instrucțiunile, cele prin care descriem pașii algoritmului au fost prezentate deja: instrucțiunea expresie (cu reprezentanți de bază citirea, afișarea, atribuirea), instrucțiunea bloc (plasarea altor intrucțiuni între acolade este interpretată ca o singură instrucțiune) și instrucțiunile de decizie (selecție) – if și switch. Cu ajutorul lor putem scrie programe în care instrucțiunile se execute în ordinea în care apar, una după alta și avem la dispoziție posibilitatea de a alege la un moment dat o singură ramură de continuare din două sau mai multe posibile.

Instrucțiunile repetitive, cele despre care urmează să discutăm, sunt cele care ne ajută să folosim cu adevărat viteza de executare a calculatoarelor. Dacă prin cele deja învățate suntem limitați să descriem noi exact fiecare pas, prin repetiții, putem descrie doar o regulă după care ceva să ruleze în mod repetat și astfel, scriind puțin cod, impunem calculatorului să execute foarte multe operații.

Sunt trei instrucțiuni repetitive: **while**, **for**, **do while**. Vom discuta pe rând, despre fiecare în parte, subliniind argumentele alegerii, la un moment dat a uneia sau a alteia.

Instrucțiunea repetitive while

Sintaxa:

```
while (expresie_logică)
    instrucțiune
```

expresie_logică poate fi, exact ca la modul indicat la instrucțiunea if, orice expresie numerică dar care este interpretată din punct de vedere logic (dacă este nenulă se consideră adevărată și dacă este egală cu 0 se consideră falsă). Instrucțiunea poate fi oricare a limbajului. Așadar, putem spune că dacă dorim aici mai multe instrucțiuni, le putem plasa între acolade, sub forma unei instrucțiuni bloc. Putem avea deci și citiri, și scrieri, și atribuiri, și decizii chiar și alte repetiții.

Modul de executare:

Mai întâi se evaluează expresia logică (precum la if ne vom mai referi la ea și prin termenul *condiție*). Dacă ea are valoarea adevărat, se va executa instrucțiunea. Până acum este exact ceea ce se întâmplă la instrucțiunea if (observăm chiar că while se aseamănă și ca sintaxă cu forma lui if fără else, diferența din acest punct de vedere fiind doar cuvântul cheie folosit). În cazul lui while după executarea instrucțiunii se revine la evaluarea expresiei logice. Iarăși, dacă aceasta este adevărată, se va trece la executarea instrucțiunii. La finalul executării instrucțiunii, iarăși se trece la evaluarea expresiei logice. Acest procedeu se repetă mereu, încheierea făcânduse când, în unul din momentele evaluării expresiei logice, aceasta este falsă.

Cazuri speciale:

- Expresia logică are valoarea falsă încă de la prima evaluare. În acest caz, instrucțiunea nu se execută niciodată, trecându-se direct mai departe.

- Expresia logică are valoarea adevărată la fiecare evaluare. În acest caz se spune că programul va cicla infinit. Adică nu se va opri niciodată, blocându-se. Aceasta nu este o eroare care apare la compilare, așadar cade în sarcina programatorului să prevină astfel de situații, rezolvarea lor făcându-se modificând variabilele care apar în instrucțiunea care se repetă într-un mod care să asigure că la un moment dat condiția va devein falsă.

Iată câteva exemple:

<table>
<tr><td>

```
int i;
i = 1;
while (i <= 5) {
    cout<<i<<" ";
    i++;
}
```

</td><td>Secvența alăturată respectă regulile cerute de sintaxă și afișează, sepatate prin câte un spațiu, numerele naturale de la 1 la 5.</td></tr>
<tr><td>

```
int i, n;
cin >> n;
i = 1;
while (i <= n) {
    cout<<i<<" ";
    i++;
}
```

</td><td>Ca diferență față de secvența de mai sus, variabila i nu mai iterează până la o valoare constantă (5) ci suntem invitați să introducem anterior de la tastatură valoarea până la care dorim să afișăm. În acest mod (putând introduce oricât) avem posibilitatea să determinăm afișarea numerelor de la 1 la o valoare oricât de mare. În acest moment trebuie să observăm forța instrucțiunii repetitive. Cu un cod de câteva rânduri, vom obține pe ecran șirul tuturor numerelor de la 1 la un milion, de exemplu. Și acest lucru se întâmplă foarte repede (din momentul începerii rulării până la terminarea afișării). Vom vedea că în practică este o temporizare de câteva secunde la numerele mari, dar acest lucru nu se datorează vitezei procesorului ci de regulă ține de rapiditatea dispozitivului de afișare. Vom vedea ulterior (când scriem în fișiere în loc de ecran) sau chiar acum (dacă eliminăm afișarea), că acest cod merge foarte repede.</td></tr>
<tr><td>

```
int n;
cin >> n;
while (n>0) {
    cout<<n<<" ";
    n--;
}
```

</td><td rowspan="2">Aceste două secvențe sunt echivalente și efectul lor este de a afișa descrescător, numerele naturale începând cu o valoare pe care o introducem de la tastatură, până la 1</td></tr>
<tr><td>

```
int i, n;
cin >> n;
i = n;
while (i>0) {
    cout<<i<<" ";
    i--;
}
```

</td></tr>
<tr><td>

```
int n;
n = 3;
while (n > 10) {
    cout<<n<<" ";
    n++;
}
```

</td><td>Aceasta este o secvență corectă sintactic dar observăm că încă de la prima evaluare condiția este falsă. Așadar nu se va executa niciodată codul dinte acolade, deci nu se va afișa nimic.</td></tr>
<tr><td>

```
int n;
n = 1;
while (n % 2 != 0) {
    cout<<n<<" ";
    n+=2;
}
```

</td><td>Pentru exemplul alăturat oprirea s-ar face atunci când, după executarea codului dintre acolade n ar devein par. Observăm că pornind cu n de la 1 și mărindu-l la fiecare pas cu 2, acest lucru nu se poate întâmpla. Avem astfel un exemplu de program greșit logic, care va cicla infinit.</td></tr>
</table>

<table>
<tr><td></td><td></td></tr>
<tr><td>

```
int n, i;
cin>>n;
i = 1;
while (i <= n) {
    if (i%2 != 0)
        cout<<i<<" ";
    i++;
}
```

</td><td rowspan="2">Cele două exemple alăturate afișează, ambele, numerele impare începând cu ` până la o valoare dată de la tatatură. În primul caz sunt parcurse din 1 în 1 toate numerele (observăm prezența instrucțiunii if ca parte a celei bloc subordonate lui while, pentru a selecta în scopul afișării doar numerele care interesează). În al doilea caz se observă că mergând cu i din doi în doi am identificat o regulă de a vizita direct doar numerele care ne interesează și în acest mod obținem un cod și mai scurt și mai rapid (numărul de executări ale instrucțiunii bloc se reduce la jumătate – și chiar conține mai puține instrucțiuni). Este de preferat deci o abordare ca a doua.</td></tr>
<tr><td>

```
int n, i;
cin>>n;
i = 1;
while (i <= n) {
    cout<<i<<" ";
    i+=2;
}
```

</td></tr>
</table>

Anterior au fost exemple de aplicții care au o singură dată de intrare și pe baza ei afișează un set de numere. Nu este singurul task ce se poate executa cu repetițiile. De asemenea modificarea unui "contor" cu o valoare fixă, ca mai sus, nu este singura posibiliate să asigurăm terminarea repetiției. Vom întâlni situații diverse. Iată acum două astfel de exemple prin probleme rezolvate.

1. Se citește un număr natural nenul de la tastatură. Să se determine la ce putere apare 2 în descompunerea sa în vactori primi. De exemplu, dacă vom introduce 40 va trebui ca programul să afișeze valoarea 3 ($40 = 2^3 * 5$), iar dacă vom introduce 9 trebuie să facem să se afișeze 0.

Observăm următoarea situație pe care o putem executa în mod repetat: dacă numărul este par, vom mări cu 1 o variabilă (exponentul lui 2) și vom împărți numărul dat la 2. Pentru noua valoare a numărului dat reluăm procedeul.

Observații:

- La un pas, avem de incrementat o variabilă cu 1 (e++) și să actualizăm valoarea variabilei ce reține numărul curent, împărțind-o la 2 (n/=2). Așadar acestea vor fi instrucțiunile dintre acoladele structurii repetitive.
- Oprirea o facem când numărul stocat în variabila citită nu mai este par (așadar condiția va fi n%2 == 0).

Odată aceste lucruri stabilite, putem trece la scrierea codului.

```
#include <iostream>
using namespace std;
int n, e;
int main () {
    cin>>n;
    e = 0;
    while (n % 2 == 0) {
        e++;
        n /= 2;
    }
    cout<<e;
    return 0;
}
```

Evident, pentru executarea în siguranță a acestui cod, valoarea introdusă de la tatatură trebuie să fie nenulă (așa cum se specifică și în enunț), în caz contrar ciclându-se infinit.

2. Se dă un număr natural. Se aplică asupra lui următoarea transformare: dacă are valoare pară, se împarte la `2`. Dacă are valoare impară, se înmulțește cu `3` și i se adună `1`. Este demonstrat că aplicându-i în mod repetat această operație, numărul va ajunge la un moment dat egal cu `1`. Programul de mai jos va calcula numărul de aplicări ale operației până când numărul ajunge egal cu `1`.

Observăm că pentru a aplica trensformarea sunt necesare două formule diferite, în funcție de paritatea lui `n` (numărul dat). Astfel ca parte a blocului repetiției vom avea un if. Alături de acest if mai trebuie incrementată, tot acolo, deci la fiecare pas, o variabilă. Oprirea se va face așadar când `n` devine `1`.

```
#include <iostream>
using namespace std;
long long n, pas;
int main () {
    cin>>n;
    pas = 0;
    while (n!=1) {
        if (n%2 == 1)
            n = n*3 + 1;
        else
            n = n/2;
        pas++;
    }
    cout<<pas;
    return 0;
}
```

Instrucțiunea repetitivă for

Observăm la programele anterioare că la scrierea unei instrucțiuni while, în afară de sintaxa impusă, mai apar des două lucruri:

- Înainte de while se realizează inițializări ale variabilelor care apar ulterior în condiție sau în instrucțiunea subordonată.
- În cadrul instrucțiunii subordonate se găsesc unele instrucțiuni care asigură trecerea la pasul următor (modifică variabilele care apar în condiție).

Instrucțiunea for include în antetul său automat, alături de condiție, și aceste două elemente: să le numim inițializarea și pasul.

Sintaxa:

```
for (expresie_inițializare; expresie_condiție; expresie_pas)
    instrucțiune
```

Oricare dintre expresiile din antetul instrucțiunii pot fi omise însă este obligatorie folosirea celor doi separatori *punct și virgulă*. Cele trei expresii pot fi oricare ale limbajului, dar expresia a doua (condiția) se evaluează din punct de vedere logic. Instrucțiunea poate fi oricare a limbajului, inclusiv instrucțiunea bloc (dacă dorim subordonarea mai multor instrucțiuni sub for se folosesc acoladele).

Modul de executare

Prima dată se evaluează inițializarea, dacă ea este prezentă. Apoi, imediat se evaluează condiția. Dacă este adevărată se trece la executarea instrucțiunii. La terminarea executării instrucțiunii se revine la antet și acum se va evalua mai întâi pasul și apoi imediat condiția. Iarăși, dacă este adevărată condiția, se execută instrucțiunea, apoi se revine la antet și se evaluează mai întâi pasul și apoi condiția. Când la o astfel de evaluare condiția este falsă, for se termină.

Observăm că inițializarea se evaluează o singură dată, la început. Înlocuiește așadar ce eram nevoiți să scriem înainte de while pentru a pregăti instrucțiunea. Pasul se evaluează de fiecare dată după executarea instrucțiunii. Exact asta făceam și la while înainte să revenim la evaluarea condiției. Condiția se evaluează și la început, după inițializare și de fiecare dată după evaluarea pasului. Dacă de la prima evaluare condiția este falsă nu se va executa niciodată instrucțiunea, ca și la while. Absența condiției este echivalent cu faptul că ea este adevărată. Vom vedea că o astfel de situație permite totuși ieșirea din repetiție în alte moduri (de exemplu prin folosirea instrucțiunii break, despre care vom discuta ulterior).

Am putea scrie desfășurat ordinea executării diverselor elemente dintr-un for astfel:

```
expresie_inițializare
expresie_condiție (true)
instrucțiune

expresie_pas
expresie_condiție (true)
instrucțiune

expresie_pas
expresie_condiție (true)
instrucțiune
…
expresie_pas
expresie_condiție (true)
instrucțiune

expresie_pas
expresie_condiție (false)
STOP
```

Exemple:

1. Cele două secvențe cer un număr natural de la tastatură și afișează, separate prin spațiu, valorile naturale începând cu `1` și până la valoarea dată. Remarcați modul mult mai compact de a rezolva problema folosind for.

```
int i, n;
cin >> n;
i = 1;
while (i <= n) {
    cout<<i<<" ";
    i++;
}
```

```
int i, n;
cin>>n;
for (i=1;i<=n;i++)
    cout<<i<<" ";
```

2. Secvența următoare afișează descrescător numerele naturale mai mici sau egale cu o valoare pozitivă `n` și care au aceeași paritate cu `n`.

```
#include <iostream>
using namespace std;
int n, i;
int main () {
    cin>>n;
    for (i = n; i>=1; i-=2)
        cout<<i<<" ";
    return 0;
}
```

Aplicațiile anterioare au de regulă o singură dată de intrare iar prin structura repetitivă parcurge un șir de numere generat în funcție de la data de intrare. Mai sus s-a afișat la fiecare pas câte un număr. Dar nu suntem, obligați să facem neapărat asta.

3. Aplicația următoare traversează toate numerele cuprinse între două valori citite `a` și `b` (se presupune că introducem numere naturale și `a <= b`) și face suma numerelor care au ultimele două cifre de parități diferite. Este tot o aplicație care generează un șir de parcurs în funcție de datele de intrare introduse.

```
#include <iostream>
using namespace std;
int a, b, i, suma;
int main () {
    cin>>a>>b;
    suma = 0;
    for (i = a; i<=b; i++)
        if (i>=10 && i%2 != i/10%2)
            suma += i;
    cout<<suma;
    return 0;
}
```

4. De multe ori, pentru problemele în care șirul de parcurs se construiește din datele de intrare, este posibil ca rezultatul cerut să poată fi obținut mai rapid făcând diverse observații. Cele două programe următoare calculează valoarea sumei `1 + 2 + 3 + ... + n` unde `n` este o valoare introdusă de la tastatură.

```
#include <iostream>
using namespace std;
int n, i, suma;
int main () {
    cin>>n;
    suma = 0;
    for (i=1;i<=n;i++)
        suma += i;
    cout<<suma;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int n, i, suma;
int main () {
    cin>>n;
    cout<<n*(n+1)/2;
    return 0;
}
```

Cele două programe sunt echivalente, pentru aceeași dată de intrare vor afișa același rezultat. Primul este o abordare brut care generează șirul numerelor naturale și construiește suma. La rezolvarea din dreapta am

observat că există o formulă care obține rezultatul în funcție de data de intrare și atunci putem afișa direct, obținând nu numai o soluție mai scurtă dar și una care rulează mult mai repede, fiind necesare doar câteva operații elementare.

Există o altă categorie de probleme în care elementele șirului de procesat sunt introduse unul după altul de la tastatură. Ne putem imagina următoarea situație practică: programul cere de la tastatură mai întâi numărul de zile de funcționare a unui magazin și apoi cere tot de la tastatură încasările din fiecare zi. Afișază apoi valoarea totală a încasărilor. Formal, un astfel de enunț sună: se cere de la tastatură un număr n și apoi n numere naturale. Realizați un calcul pe baza numerelor introduse. În aceste probleme, citim la început numărul de elemente ale șirului, apoi într-o repetiție, la fiecare pas citim câte un număr. Astfel, ca parte a blocului de intrucțiuni ale repetiției avem și un cin.

Probleme rezolvate

1. Se citeste `n` si apoi `n` numere. Să se afișeze suma lor.

```
#include <iostream>
using namespace std;
int n, i, suma, x;
int main () {
    cin>>n;
    suma = 0;
    for (i=1;i<=n;i++) {
        cin>>x;
        suma += x;
    }
    cout<<suma;
    return 0;
}
```

La astfel de probleme nu trebuie să ne gândim că trebuie calculată suma în mod miraculos după vreo formulă întrucât nu se știe ce număr se introduce la fiecare pas. Așadar se folosește principiul pașilor mărunți: se pornește cu o variabilă inițializată cu `0` (suma) și la fiecare pas al repetiției, după citirea unui număr, se acumulează la variabilă valoarea sa. Acesta este deci modul în care calculăm de regulă o sumă.

Observăm că pentru preluarea de la tastatură a fiecărui număr folosim aceeași variabilă `x`. Acest lucru este posibil întrucât după introducerea fiecărei valori aceasta se și acumulează la sumă, deci putem folosi aceeași variabilă pentru preluarea următorului număr.

2. Se citește `n` și apoi `n` numere naturale. Să se afișeze maximul lor.

```
#include <iostream>
using namespace std;
int n, i, maxim, x;
int main () {
    cin>>n;
    maxim = -1;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x > maxim)
            maxim = x;
    }
    cout<<maxim;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int n, i, maxim, x;
int main () {
    cin>>n;
    cin>>maxim;
    for (i=2;i<=n;i++) {
        cin>>x;
        if (x > maxim)
            maxim = x;
    }
    cout<<maxim;
    return 0;
}
```

Ambele variante de mai sus rezolvă problema enunțată, ele fiind echivalente. Se pun astfel în evidență două abordări clasice de calculare a maximului dintr-un șir de valori. Se folosește o variabilă în care la final vom obține valoarea maximă. Pe parcursul citirii, aceasta se compară pe rând cu fiecare valoare introdusă iar când este cazul maximul se actualizează. Așadar, la un moment dat maxim reține cea mai mare valoare dintre cele introduse deja. La inițializarea variabilei maxim sunt două abordări: se pune la început în maxim o valoare garantat mai mică decât a tuturor elementelor șirului ce se va introduce (în acest mod suntem siguri că elementele introduse vor actualiza variabila) – cazul programului din partea stângă; se inițializează maxim cu unul dintre elementele șirului (de regulă primul), fiind astfel siguri că în variabilă va rămâne una dintre valorile din șir.

3. Se citesc `n` și apoi `n` numere naturale, se cere afișarea valorii minime dintre cele introduse. Abordarea este similară cu aceea de la calculul maximului, dar în cazul inițializării, aceasta se face fie cu un element din șir, fie cu o valoare sigur mai mare decât elementele șirului. Soluția de mai jos se bazează pe faptul că se cunoaște că elementele șirului dat sunt maxim `1000`.

```
#include <iostream>
using namespace std;
int n, i, minim, x;
int main () {
    cin>>n;
    minim = 1001;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x < minim)
            minim = x;
    }
    cout<<minim;
    return 0;
}
```

4. Se citește un număr `n` și apoi `n` numere naturale. Să se calculeze numărul valorilor pare.

```
#include <iostream>
using namespace std;
int n, i, cnt, x;
int main () {
    cin>>n;
    cnt = 0;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x % 2 == 0)
            cnt++;
    }
    cout<<cnt;
    return 0;
}
```

Această problemă se încadrează în categoria celor de numărare. Abordarea este asemănătoare cu aceea de la calculul sumelor, se utilizează o variabilă care se inițializează cu `0` iar la fiecare pas valoarea acesteia crește cu `1` dacă elementul curent îndeplinește condiția de numărare.

Instrucțiunile break și continue

Am văzut deja o situație de folosire a instrucțiunii **break**, pe ramurile instrucțiunii switch acolo unde dorim să întrerupem executarea.

În acest capitol arătăm cum cele două instrucțiuni se pot folosi în structurile repetitive. Deocamdată le vom utiliza în cele două repetiții învățate, while și for, dar vom vedea ulterior că este se pot folosi similar și la a treia instrucțiune repetitivă, do while.

Instrucțiunea break

Sintaxa

```
break;
```

Modul de funcționare

Executarea acestei instrucțiuni în cadrul unei repetiții face ca acea repetiție să se încheie imediat (ca și cum s-ar fi ajuns la evaluarea condiției și aceasta este falsă). Dacă la pasul curent în blocul subordonat repetiției mai sunt și alte instrucțiuni, acestea nu se vor mai executa. Dacă avem o instrucțiune break subordonată unei repetiții, care la rândul ei este subordonată altei repetiții, break va încheia doar repetiția cea mai apropiată (aflată imediat deasupra). Cu toate că nu este neapărat obligatoriu, instrucțiunea break se plasează de regulă într-un if subordonat repetiției, altfel executarea sa s-ar face de la prima intrare în blocul repetiției și de regulă acest lucru nu este practic.

Exemple

```
#include <iostream>
using namespace std;
int n, i, x;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x % 2 == 0) {
            cout<<i-1;
            break;
        }
    }
    if (i == n+1)
        cout<<n;
    return 0;
}
```

Programul cere de la tastatură maxim `n` numere naturale și afișează câte valori impare s-au introdus până la apariția prime valori pare. Dacă se întâlnește o valoare pară citirea se oprește. Dacă toate valorile citite sunt impare se va afișa `n`.

O observație suplimentară: verificarea că un `for (i=1;i<=n;i++)` ... a mers până la capăt o facem testând ca la final `i` să aibă valoarea `n+1` (prima valoare a lui `i` când expresia logică devine falsă).

```
#include <iostream>
using namespace std;
int n;
int main () {
    cin>>n; /// pozitiv
    while (1) {
        if (n%2 == 0)
            n/=2;
        else
            break;
    }
    cout<<n;
```

Acesta este un exemplu de utilizare a instrucțiunii break în cadrul unei repetiții while.
Programul afișează valoarea numărului rămas după "consumarea" factorului prim `2` (împărțirea în mod repetat la `2` cât timp numărul este încă par).
Observăm că folosind break putem face să se termine și repetițiile la care condiția pare să provoace ciclu infinit.

`    return 0;` `}`	

`#include <iostream>`
`using namespace std;`
`int n;`
`int main () {`
`    cin>>n;`
`    if (n%2 == 0)`
`        break;`
`    return 0;`
`}`
	Programul alăturat nu se compilează.
	Instrucțiunea break nu poate fi utilizată decât subordonată unei repetiții

Instrucțiunea continue

Așa cum aminteam, **continue** poate fi folosită în oricare dintre cele trei instrucțiuni repetitive.

Sintaxa

```
continue;
```

Modul de executare

Restul de instrucțiuni de la trecerea curentă prin blocul subordonat repetiției nu se vor mai executa și se trece fie la evaluarea condiției (în cazul while și do while) fie la expresia pas, apoi la condiție în cazul lui for. Așadar nu se încheie definitiv repetiția (ca în cazul lui break) dar se sar anumite instrucțiuni.

Exemple

`#include <iostream>`
`using namespace std;`
`int n, i, x, sol;`
`int main () {`
`    cin>>n;`
`    sol = 0;`
`    for (i=1;i<=n;i++) {`
`        cin>>x;`
`        if (x % 2 == 0)`
`            continue;`
`        sol++;`
`    }`
`    cout<<sol;`
`    return 0;`
`}`
	Programul alăturat cere n apoi n numere de la tastatură. Se numără valorie impare introduse. Se observă că dacă valoarea curentă este pară, condiția de la if face să se execute continue și la acea trecere prin blocul de instrucțiuni nu se mai ajunge să se execute și incrementarea variabilei sol.

`#include <iostream>`
`using namespace std;`
`int n;`
`int main () {`
`    cin>>n;`
`    if (n%2 == 0)`
`        continue;`
`    return 0;`
`}`
	Acest program nu se compilează. Instrucțiunea continue trebuie subordonată unei repetiții.

Iată câteva exemple din care învățăm mai multe despre utilizarea instrucțiunii for.

`int n;`
`cin>>n; /// pozitiv`
`for (;n!=0;n--)`
`  cout<<n<<" ";`
	Se afișează descrescător, separate prin spațiu, toate numerele naturale de la `n` la `1`.

`int n;`
`cin>>n; /// pozitiv`
`for (n!=0;n--)`
`  cout<<n<<" ";`
	Eroare de compilare. În interiorul lui for este un singur delimitator de expresii *punct și virgulă*.

`int n;`
`for (;;) {`
`  cin>>n;`
`  if (n%10 == 0)`
`  break;`
`}`
	Această secvență se compilează, chiar dacă lipsește condiția. În acest caz aceasta se consideră adevărată. Pare ciclu infinit. Oprim însă repetiția cu break atunci când se introduce de la tastatură o valoare multiplu de `10`.

`int n;`
`cin>>n; /// pozitiv`
`for (n%2 == 0 ? i=n : i=n-1;`
`  i>1;i-=2)`
`  cout<<i<<" ";`
	Secvența alăturată afișează descrescător numerele pare mai mici sau egale cu o valoare dată `n`. Observăm folosirea operatorului ternar ca parte a expresiei de inițializare cu scopul de a porni contorul `i` (care apoi se decrementează mereu cu `2`) de la cea mai apropiată valoare pară mai mică sau egală cu `n`.

`int n;`
`cin>>n; /// pozitiv`
`for`
`(st=1,dr=n;st<dr;st++,dr--) {`
`  cout<<st<<" "<<dr<<"\n";`
`}`
	Secvența alăturată, pentru `n=6` va afișa:
	`1 6`
	`2 5`
	`3 4`
	Așadar se fac `n/2` pași. Observăm că se fac două inițializări și totodată două operații la expresia pas. Obținem acest lucru folosind operatorul virgulă pentru a compune o singură expresie (din alte două mai mici) atât pentru inițializare cât și pentru pas. Atenție la folosirea operatorului virgulă în cadrul expresiei condiție. O scriere de forma `i<=n, j<=m` face condiția falsă doar când `j` trece de `m`, indiferent de valoarea lui `i`. Acest lucru se datorează faptului că valoarea finală a unei expresii virgulă este egală cu valoarea ultimei subexpresii din listă. Dacă se dorește scrierea unei condiții puternice care să impună ambele subcondiții se folosește operatorul `&&` `(i<=n && j<=m)`.

`for (i=1;i<=10;i++);` `  cout<<i;`	Se afișează o singură valoare, `11`. Începătorii mai pun punct și virgulă după antetul lui for iar acest lucru nu este semnalat ca eroare de sintaxă, însă se consideră că for nu execută nicio instrucțiune la fiecare trecere. Aplicarea, în ordine a evaluării expresiilor se face însă și se merge cu `i` la capăt. Atenție să nu ne grăbim să spunem că i este le final `10`, întrucât atunci condiția este încă adevărată, deci se mai realizează o dată pasul și condiția, `i` devenind `11` apoi for se termină.

Transcrierea unei instrucțiuni **while** în mod echivalent cu ajutorul lui **for**.

`while (expresie_logică)` `  instrucțiune`	`for (;expresie_logică;)` `  instrucțiune`

Transcrierea unei instrucțiuni **for** în mod echivalent cu ajutorul lui **while**.

`for (inițializare;condiție;pas)`
`    instrucțiune`
	`inițializare`
	`while (condiție) {`
	`    instrucțiune`
	`    pas`
	`}`

Instrucțiunea repetitivă condiționată posterior, do while

Considerăm următoarea problemă: Se citesc numere de la tastatură până când se introduce numărul `0`. Să se calculeze maximul acestor numere (se garantează că numerele sunt întregi și au cel mult `3` cifre). Problemele anterioare care prelucrau un șir de valori introduse de la tastatură cereau mai întâi numărul de valori din șir apoi valorile. Astfel, după introducerea numărului de valori era potrivită instrucțiunea for pentru a scrie o repetiție cu `n` pași în scopul preluării numerelor din șir. Descrierea anterioară este în spiritul folosirii repetiției cu număr cunoscut de pași. În cazul acestei probleme nu se cunoaște de la început câte numere vor fi citite. O posibilă rezolvare este:

```
int n, maxim;
maxim = -1000;
cin>>n;
if (n > maxim)
    maxim = n;
while (n!=0) {
    cin>>n;
    if (n > maxim)
        maxim = n;
}
cout<<maxim;
```

Citirea va trebui să apară cu siguranță în instrucțiunea repetitivă. Pe de altă parte, deducem din enunț că este necesară cel puțin o citire. Astfel se impune utilizarea repetiției cu test final.

Sintaxa

```
do
    instrucțiune
while (expresie_logică);
```

Instrucțiune și expresie_logică au aceeași semnificație ca la while. Observăm că la final se folosește caracterul punct și virgulă.

Modul de funcționare

Se execută mai întâi instrucțiunea (aceasta este singura și marea diferență față de while). Apoi se evaluează expresia logică. Dacă aceasta este adevărată se vor executa iarăși instrucțiunile. Și tot așa până când la o evaluare expresia logică (sau, cum îi mai spunem, condiția) este falsă.

Și în acest caz trebuie avută în vedere evitarea ciclului infinit.

Ca și celelalte două repetiții, această instrucțiune permite să utilizăm instrucțiunile continue și break, cu aceeași semnificație ca la while.

Problema anterioară poate fi rezolvată și în modul următor:

```
int n, maxim;
maxim = -1000;
do {
    cin>>n;
    if (n > maxim)
        maxim = n;
} while (n!=0);
cout<<maxim;
```

Observăm că astfel codul este mai elegant și mai puțin redundant (nu conține același lucru scris în mai multe locuri).

Atenție la diferențele dintre while și do while. Iată un exemplu:

`int n;`
`cin>>n;`
`while (n!=0) {`
`  cout<<n<<" ";`
`  n--;`
`}`
	`int n;`
	`cin>>n;`
	`do {`
	`  cout<<n<<" ";`
	`  n--;`
	`} while (n!=0);`

Dacă introducem pentru n o valoare pozitivă ambele secvențe oferă același rezultat (afișează descrescător numerele naturale mai mici sau egale cu `n`, separate prin câte un spațiu). Dacă vom citi valoarea `0` secvențele nu se mai comportă la fel. Cea din stânga are de la început condiția falsă, deci nu va tipări nimic, iar la cea din dreapta se întâmplă următoarele: se tipărește `0`, `n` devine apoi `-1` iar la verificarea condiției acesta este nenul (condiție adevărată) și se va relua executarea instrucțiunilor, apoi se afișează `-1`, `-2` ... comportamentul fiind acela de ciclu infinit.	

Iată și alte exemple

`int n;`
`cin>>n;`
`do`
`  n--;`
`while (n%100 != 0);`
	Secvența alăturată afișează cel mai mare multiplu de `100` strict mai mic decât valoarea citită.
	Observăm că nu este obligatorie folosirea acoladelor dacă avem o singură instrucțiune simplă înăuntru.

`int n;`
`cin>>n;`
`do {`
`  n--;`
`  if (n%100 == 0)`
`  break;`
`} while (1);`
`cout<<n;`
	Aici avem o secvență echivalentă cu aceea de mai sus în care arătăm un exemplu de folosire a instrucțiunii break în do-while.

`int n;`
`cin>>n;`
`do`
`  n--;`
`while (n%100 != 0)`
	Eroare de compilare cauzată de lipsa caracterului *punct și virgulă* la final.

Sunt situații când dorim să traducem un cod scris cu while în unul cu do-while și invers. Iată cum o putem face:

*Transcrierea codului scris cu do **do-while** în unul echivalent scris cu **while***

`do`
`    instrucțiune`
`while (expresie_logică);`
	`instrucțiune`
	`while (expresie_logică)`
	`    instrucțiune`

Este așadar necesar să ne asigurăm că instrucțiunile se execută o dată înainte de evaluarea expresiei logice, lucru garantat la instrucțiunea do-while. Rezolvăm asta prin simpla plasare a lor în fața lui while.

*Transcrierea codului scris cu **while** în unul echivalent scris cu **do-while***

`while (expresie_logică)`
`    instrucțiune`
	`if (expresie_logică)`
	`    do`
	`        instrucțiune`
	`    while (expresie_logică);`

În această situație trebuie să ne asigurăm că dacă espresia logică este la început falsă, să nu se execute nimic. Putem face asta, după cum se vede, subordonând do-while unui if cu aceeași condiție.

În general putem încerca obținerea scrierilor echivalente și fără a folosi șablonul, totul este să analizăm bine cazurile datelor de intrare pentru care instrucțiunea while are condiția falsă de la început, urmărind dacă atunci executarea o dată a instrucțiunilor în do-while schimbă valoarea datelor de ieșire.

Așa cum spuneam la începutul capitolului, repetițiile sunt instrumentele cu ajutorul cărora putem cere într-un mod simplu calculatorului să execute foarte repede un număr mare de operații. Dar cât de repede execută calculatoarele operațiile? Ne vom ocupa mai exact de acest lucru într-un capitol separat. Acum doar discutăm despre o estimare foarte practică: Ne încadrăm într-un timp de rulare aproape instantaneu (să spunem maxim o secundă) dacă numărul de pași pe care îi face programul este cel mult câteva milioane (să spunem `50`). De ce această exprimare? În primul rând timpul de executare depinde de viteza procesorului, apoi depinde de ce instrucțiuni se rulează la fiecare pas (spre exemplu dacă se lucrează cu numere reale calculele se fac mai lent).

Probleme rezolvate

1. Se citesc de la tastatură mai întâi un număr `n` și apoi `n` numere naturale și se cere să calculăm produsul lor.

```
#include <iostream>
using namespace std;
int n, p, i, x;
int main () {
    cin>>n;
    p = 1;
    for (i=1;i<=n;i++) {
        cin>>x;
        p = p * x;
    }
    cout<<p;
    return 0;
}
```

Observăm din acest exemplu modul în care se calculează un produs: inițializăm o variabilă cu `1` (elementul neutru de la înmulțire) și la fiecare pas acumulăm la produs un număr. Observați că nu am specificat dimensiuni ale datelor pentru a ne concentra strict pe algoritm. Trebuie avut în vedere la calculul produselor că

aceste valori cresc foarte rapid și este necesară estimarea rezultatelor pentru a stabili tipurile de date corespunzătoare pentru variabile.

2. Se citesc două numera naturale `a` și `b`. Să se calculeze valoarea a^b.

```
#include <iostream>
using namespace std;
int p, i, a, b;
int main () {
    cin>>a>>b;
    p = 1;
    for (i=1;i<=b;i++) {
        p = p * a;
    }
    cout<<p;
    return 0;
}
```

Și această problemă este rezolvabilă tot pentru date mici. Observăm că și în acest caz este necesară calcularea unui produs, `a*a*a* ... *a` de `b` ori. Strategia este deci de a inițializa o variabilă cu `1` și, printr-o repetiție cu `b` pași să înmulțim mereu la variabilă valoarea `a`.

3. Se citește de la tastatură un număr natural nenul. Se cere să verificăm dacă valoarea sa este o putere a lui `2`.

```
#include <iostream>
using namespace std;
int n, p;
int main () {
    cin>>n;
    p = 1;
    while (p < n)
        p *= 2;
    if (p == n)
        cout<<"DA";
    else
        cout<<"NU";
    return 0;
}
```

```
#include <iostream>
using namespace std;
int n;
int main () {
    cin>>n;
    while (n%2 == 0)
        n /= 2;
    if (n == 1)
        cout<<"DA";
    else
        cout<<"NU";
    return 0;
}
```

Programele de mai sus sunt echivalente și ambele rezolvă problema enunțată. Notăm cu n data de intrare. Cel din stânga construiește pe rând toate puterile lui `2`, oprindu-se când puterea curentă devine mai mare sau egală cu `n`. Întrucât acest șir este strict crescător nu mai are sens continuarea. Dacă ne oprim chiar la `n` atunci afișăm `DA`. Soluția din stânga se bazează pe "consumarea" factorului prim `2` din descompunerea în factori primi a lui `n`, cod pe care l-am mai scris și mai devreme în acest material. După această etapă trebuie ca valoarea să ajungă la `1` pentru a afișa `DA`.

4. Se citesc de la tastatură mai multe numere (până la întâlnirea unei valori mai mică sau egală cu `0`). Calculați și afișați numărul valorilor putere de `2`.

```
#include <iostream>
using namespace std;
int n, x, sol;
int main () {
    sol = 0;
```

```
    do {
        cin>>x;
        if (x > 0) {
            while (x % 2 == 0)
                x /= 2;
            if (x == 1)
                sol++;
        }
    } while (x > 0);
    cout<<sol;
    return 0;
}
```

Observăm că este o îmbinare între o problemă în care se cere introducerea mai multor valori de la tastatură și o problemă în care se cere verificarea pentru un număr dacă este putere de `2`. Este totodată prima problemă în care avem structuri repetitive una subordonată alteia.

Iată, în continuare și o soluție echivalentă:

```
#include <iostream>
using namespace std;
int n, x, sol, p;
int main () {
    sol = 0;
    for (;;) {
        cin>>x;
        if (x <= 0)
            break;
        p = 1;
        while (p < x)
            p *= 2;
        if (p == x)
            sol++;
    }
    cout<<sol;
    return 0;
}
```

Am schimbat două lucruri la această variantă: repetiția care cere câte un număr la fiecare pas și modul de a testa dacă un număr este putere de `2`. Dar altceva doresc să subliniez aici. Avem două variabile în care acumulăm câte ceva: `sol`, folosită pentru numărare și `p`, utilizată pentru calculul unui produs. Observăm locul inițializării fiecăreia dintre ele, lucru deloc indiferent. Variabila în care calculăm produsul trebuie inițializată înăuntrul repetiției mari întrucât la fiecare număr nou citit trebuie să refacem verificarea începând construirea puterilor lui `2` de la cea mai mică. Variabila sol, în schimb, contorizează cel mult `1` la fiecare număr citit, așadar trebuie inițializată înaintea lui for.

5. Se citește de la tastatură un număr `n` și un număr natural `k` (strict mai mare decât `1`), apoi se citesc `n` numere întregi. Să se calculeze câte dintre numerele citite sunt strict pozitive și putere a lui `k`.

```
#include <iostream>
using namespace std;
int n, k, sol, i, x;
int main () {
```

```
#include <iostream>
using namespace std;
int n, k, sol, i, x;
int main () {
```

```
    cin>>n>>k;
    sol = 0;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x > 0) {
            while (x%k == 0)
                x /= k;
            if (x == 1)
                sol++;
        }
    }
    cout<<sol;
    return 0;
}
```

```
    cin>>n>>k;
    sol = 0;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x <= 0)
            continue;
        while (x%k == 0)
            x /= k;
        if (x == 1)
            sol++;
    }
    cout<<sol;
    return 0;
}
```

Cele două rezolvări sunt ambele corecte și echivalente. Observați cum, pentru soluția din partea dreaptă, prin folosirea lui continue, scăpăm de a plasa codul ce prelucrează numărul curent într-un if și indentarea lui mai la dreapta (verificăm cazul când numărul nu interesează și renunțăm la restul de instrucțiuni de la trecerea curentă, întrucât dacă se ajunge la ele sigur numărul este pozitiv și îl testăm direct).