**Incrementarea prefixată și postfixată**

**x++** - postfixată

**++x** - prefixată

Folosite de sine stătătoare au același efect.

Ele se pot folosi însă în cadrul unor expresii. În acest caz, forma postfixată va utiliza în cadrul expresiei vechea valoare a lui n, după care n-ul va crește.

În cazul celei prefixate, mai întâi este crescută valoarea lui n, abia apoi este utilizată în cadrul expresiei valoarea deja crescută.

Ex:

**a=5;**

**b=5\*(a++);**

**cout<<a<<" "<<b;//se afișează 6 și 25**

Ex:

**a=5;**

**b=5\*(++a);**

**cout<<a<<" "<<b;//se afișează 6 și 30**

**Expresia condițională**

Sintaxa:

**expr\_cond?val\_adev:val\_fals;**

Se evaluează **expr\_cond**. Dacă este adevărată, expresia va da **val\_adev**, dacă nu, **val\_fals**.

A nu se confunda cu instrucțiunea if, ci mai degrabă cu funcția if din Excel.

Ex:

**a[i][j]=i<j?i+1:j+1;**

Dacă i<j atunci expresia va avea valoarea i+1, dacă nu va avea valoarea j+1.

Ex: La varianta 15 din temă, aveam:

**if(i%2==0) a=a-i\*i;**

**else a=a+i\*i;**

Asta se poate scrie elegant cu expresie condițională:

**a= i%2==0?a-i\*i:a+i\*i;**

**Vectori**

Un vector este o variabilă capabilă să memoreze **simultan** mai multe valori. Toate valorile vectorului trebuie să fie de același tip (numere întregi, reale, alte tipuri).

Elementele sunt structurate (memorate) în funcție de un indice (poziție) în cadrul vectorului.

Declararea unui vector în C++:

tip nume\_vector[**nr\_elemente**];

Ex:

int a[**4**];

Obs:

1) Indicii încep de la 0. Indicele maxim va fi așadar numărul de elemente din declarație, minus 1. Spre exemplu, vectorul de mai sus are elementele: a[0], a[1], a[2] și a[3].

2) **nr\_elemente** (în cadrul declarației) trebuie să fie O CONSTANTĂ. !!! Declararea cu variabilă între parantezele pătrate este considerată greșeală și se depunctează!

3) Un stil mai școlăresc folosește indici de la 1 (chiar dacă elementul a[0] există, îl ignorăm). În acest caz trebuie să avem grijă ca, în declarare, să prevedem cu 1 mai mult decât maximul posibil.

De exemplu dacă o problemă specifică "folosim vectorul a cu cel mult 10 elemente" iar noi dorim să programăm de la 1, vom declara

int a[11];

4) Variabilele declarate în interiorul lui main() sunt considerate variabile locale ale acestuia. Ele mănâncă memorie dintr-o zonă numită "stack" (stivă) care de regulă este mai mică. Vectorii putând fi destul de mari ca dimensiune, iar CodeBlocks având stiva mică, programul poate crăpa inexplicabil (fără să primiți erori explicite).

Ca o recomandare generală, declarați vectorii ÎNAINTE de main.

**Citirea unui vector**

1) De la tastatură, când în prealabil se dă și numărul de elemente din vector:

Fie vectorul "a" cu "n" elemente, cu maxim 50 de valori.

int a[51];

cin>>n;//trebuie să introducem mai întâi numărul de elemente din vector

for(i=1;i<=n;i++)

cin>>a[i];

2) De la tastatură, când în prealabil NU se dă și numărul de elemente din vector, ci se tot citesc elemente până la îndeplinirea unei condiții:

cin>>x;//citim prima valoare

n=0;

while(x NU îndeplinește condiția de terminare)

{

a[++n]=x; //asta e echivalentă cu : {n++;a[n]=x;}

cin>>x;//secvența aceasta creează șirul de la indicele 1

}

3) Din fișier, în cazul în care în fișier se dau numere separate prin caractere albe, fără a fi menționat anterior și numărul de valori ce se vor citi:

fie fin = fișierul de intrare

(de regulă se deschide prin: ifstream fin("nume.extensie");

și trebuie inclusă biblioteca <fstream> )

n=0;

while(fin>>x) //treaba asta are atât rolul de a citi cât și de a

{ //întrerupe while-ul în momentul în care NU mai are ce citi

a[++n]=x;

}

**Afișarea unui vector**

1) Simplu:

for(i=1;i<=n;i++)

cout<<a[i]<<" ";

cout<<"\n";

2) Fancy: între paranteze și cu virgule între ele:

cout<<"("<<a[1];

for(i=2;i<=n;i++)

cout<<","<<a[i];

cout<<")\n";

**Căutarea unei valori într-un vector**

Fie vectorul a cu n elemente. Ne interesează să căutăm o valoare care îndeplinește o condiție. Mai precis să vedem dacă apare sau nu, iar dacă apare, să determinăm indicele **primei** sale apariții.

Metoda 1:

pe for întrerupt forțat cu break:

**indice**=-1;//în această variabilă vom determina indicele. Marca -1 va fi un

//semn că valoarea căutată NU a fost găsită

for(i=1;i<=n;i++)

{

if(a[i] are proprietatea cerută)

{

indice=i;

break;

}

}

if(indice==-1) → NU apare

else → apare la indicele **indice**

Metoda 2:

se pretează doar atunci când condiția pe care trebuie s-o îndeplinească elementul se poate reduce la un simplu if. Această metodă face căutarea pe un while:

k=1;

while(**k<=n** && **a[k] NU îndepl. condiția**)

**k++;**

if(k>n) → NU apare

else → apare la indicele k.

(**k<=n** se traduce prin "mai avem unde căuta" iar

**a[k] NU îndepl. condiția** se traduce prin "N-am găsit" iar

**k++**; se traduce prin "continuăm să căutăm")

**Formarea unui vector element cu element**

Am aplicat deja acest procedeu atunci când am citit din fișier până la terminarea acestuia.

Principiul este următorul (dacă lucrăm de la 1):

- se inițializează n=0; (n va conține în final numărul de elemente ale vectorului)

- de fiecare dată când dorim să adăugăm un element nou după cele existente:

a[++n] = valoarea dorită;

**Conversii între tipuri de date**

Prin conversie se înțelege interpretarea unei valori care aparține unui tip de date ca și valoare care aparține altui tip de date.

Astfel, conversia de la int la double sau float este într-o oarecare măsură naturală, pe când conversia inversă se va face cu pierderi de informație.

Pentru a face conversia de date în C++ se folosesc "**operatorii de casting**", mai precis cuvintele cheie ale tipurilor de date incluse între paranteze și scrise înaintea datelor de convertit.

Ex:

(int)8.98 → 8

(double)8 → 8.0

Cele de mai sus sunt conversii explicite (adică i le precizează utilizatorul programului).

Mai există și conversii implicite:

• dacă se atribuie o valoare întreagă unei variabile reale, aceasta va primi valoarea respectivă cu zecimalele .0

(dacă avem **double x; x=7**; - de fapt x primește 7.0)

• dacă se atribuie o valoare reală unei variabile întregi, zecimalele vor fi trunchiate.

(dacă avem **int x=11.0/4**; x va primi 2, deși rezultatul împărțirii este 2.75)

**Operatorii / și %**

În C++ expresia **a / b** are următoarele semnificați:

1) Dacă a și b sunt întregi, ne va furniza câtul întreg.

(Dacă vreunul dintre a sau b sunt negativi, se respectă regula semnelor).

Ex:

11 / 4 → 2

-11 / 4 → -2

11 / -4 → -2

-11 / -4 → 2

2) Dacă cel puțin unul dintre a sau b este real (float sau double), ne va furniza câtul cu zecimale.

Ex:

11.0 / 4 → 2.75

11 / 4.0 → 2.75

11.0 / 4.0 → 2.75

-11.0 / 4 → -2.75

(double)11/4 → 2.75

(double)(11/4) → 2.0

Expresia a%b reprezintă restul împărțirii întregi.

Obs. Dacă cel puțin unul dintre membrii lui % este de tip real, operația va da eroare (nici măcar NU compilează).

Restul se calculează din teorema împărțirii cu rest, adică R = D – C\*I.

(practic, a%b = a – b\*(a/b).)

Ex:

11%4 → 3

11%-4 → 3

-11%4 → -3

-11%-4 → -3

(Obs: **Restul are semnul deîmpărțitului. !!! LA MATE NU E LA FEL !!!**)

**Ștergerea unui element dintr-un vector, de la un indice cunoscut, k**

Fie vectorul a cu n elemente. Indicele k este dat sau obținut dintr-o etapă anterioară.

Pentru a șterge elementul de la indicele k, de fapt toate elementele de după acest indice, adică cele cu indici între k+1 și n trebuie mutate cu o poziție la stânga:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a[1] | a[2] | ... | a[k-1] | a[k] | a[k+1] | a[k+2] | ... | a[n-2] | a[n-1] | a[n] |

După mutare, actualizăm valoarea lui n: vectorul va avea cu 1 element mai puțin.

Iată codul: sau codul absolut echivalent:

for(i=k;i<=n-1;i++)

a[i]=a[i+1];

n--;

(Obs: Același efect ca mai sus îl are și

for(i=k+1;i<=n;i++)

a[i-1]=a[i];

n--;)

**Inserarea unui element**

Fie vectorul a cu n elemente. Indicele k este dat sau obținut dintr-o etapă anterioară.

Pentru a insera o valoare v la indicele k (adică pe locul vechiului a[k], in sensul că toate valorile de la a[k] și până la a[n] se mută), de fapt toate elementele de după acest indice, adică cele cu indici între k și n, trebuie mutate cu o poziție la dreapta iar pe poziția pe care ne-am făcut loc se inserează elementul. Mutarea trebuie făcută de la dreapta la stânga:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a[1] | a[2] | ... | a[k-1] | a[k] | a[k+1] | a[k+2] | ... | a[n-2] | a[n-1] | a[n] | a[n+1] |

După mutare, actualizăm valoarea lui n: vectorul va avea cu 1 element mai mult.

La indicele la care ne-am făcut loc, inserăm noul element:

for(i=n;i>=k;i--)

a[i+1]=a[i];

n++;

a[k]=valoarea\_de\_inserat

**Sortări**

Prin sortare înțelegem rearanjarea elementelor unui vector astfel încât să le aducem în ordine crescătoare sau descrescătoare.

Există foarte multe metode de a realiza acest lucru.

În continuare ne ocupăm de sortări crescătoare. E clar că cele descrescătoare se fac "în oglindă". Teoria de mai jos se va referi la un vector a cu n elemente, pe care le sortam deci crescător.

**Sortarea prin selecția minimului**

Este o sortare în n-1 pași. La fiecare pas k, (k=1,n-1) se fac următoarele:

- se determină minimul dintre elementele cuprinse între indicii k și n, cu tot cu indicele său

- se interschimbă minimul cu elementul de la indicele k.

Iată codul:

for(k=1;k<=n-1;k++)

{

imin=k;

for(j=k;j<=n;j++)

if(a[j]<imin)

{

min=a[j];

imin=j;

}

aux=a[k];a[k]=a[imin];a[imin]=aux;

}

**Sortarea prin metoda bubble-sort**

Este o sortare care face multiple traversări ale vectorului.

La fiecare traversare se compară câte două elemente vecine, a[i] și a[i+1] (să avem grijă să mergem cu i până la n-1). Dacă elementele NU sunt în ordinea care convine, se vor interschimbă.

Algoritmul se încheie în momentul în care, la o astfel de traversare, n-am mai interschimbat nimic.

Iată codul:

do

{

gata=1;

for(i=1;i<=n-1;i++)

if(a[i]>a[i+1])

{

aux=a[i];a[i]=a[i+1];a[i+1]=aux;

gata=0;

}

}while(gata==0);

**Sortarea prin interschimbare (!preferabil s-o utilizați pe asta față de celelalte!)**

Este o sortare bazată pe cea a selecției minimului. Are codul cel mai simplu și e cel mai ușor de adaptat la diverse situații.

Ideea algoritmului: Se compară fiecare element DOAR cu cele de DUPĂ el. Dacă ordinea a două elemente NU convine, le interschimbăm.

Iată codul:

for(i=1;i<=n-1;i++)

for(j=i+1;j<=n;j++)

if(a[i]>a[j])

{

aux=a[i];a[i]=a[j];a[j]=aux;

}

**Sortarea prin inserție**

Presupune menținerea sortată a unui vector în care, la fiecare pas, inserăm câte un element astfel încât acesta să rămână tot sortat.

Iată algoritmul care citește n elemente și formează cu ele vectorul a, sortat:

m=0;

cin>>n;

for(k=1; k<=n; k++)

{

cin>>v;//inseram v in vectorul a cu m elemente direct unde-și are locul

j=1;//determinăm j=indicele la care trebuie inserat

while(j<=m && a[j]<=v)//adică primul elem. mai mare ca v

j++;

for(i=m; i>=j; i--)

a[i+1]=a[i];

m++;

a[j]=v;

}

**Sortarea prin numărare**

Este un algoritm eficient d.p.d.v. al timpului de executare, în general superior celor prezentate anterior. Din păcate, este foarte restrictiv asupra datelor cărora li se poate aplica: nu se poate aplica decât dacă valorile de sortat sunt numere naturale, suficient de mici (și anume de maxim 4 cifre, iar raportul dintre valoarea maximă și numărul de valori să nu fie foarte mare (1 = raport bun, 10 = raport acceptabil, >100 – nu e foarte bun).

(Nu e musai ca valorile să fie naturale și cuprinse între 0 și vmax, însă dacă nu sunt așa trebuie ca, cel puțin, printr-un artificiu, să fie realizată o echivalență între valorile date și niște valori naturale de la 0 la vmax)

Principiul constă în a NU memora valorile pe care le sortăm, ci a memora informații despre ele într-un vector care se numește "de numărare". Dacă acest vector este numit "nr", atunci elementele sale vor avea semnificația: nr[v]=numărul de apariții ale valorii v.

De exemplu, să considerăm că avem de sortat următorul șir de valori, care sunt TOATE cifre (0<=cifra<=9):

1 3 3 2 8 9 8 6 5 4 4 0 1 0 1 0 2 1 0 2

Iată cum arată vectorul nr pentru ele:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| v | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| nr[v] | 4 | 4 | 3 | 2 | 2 | 1 | 1 | 0 | 2 | 1 |

Pe baza lui nr[v] pot liniștit să afișez valorile sortate:

0 0 0 0 1 1 1 1 2 2 2 3 3 4 4 5 6 8 8 9

Iată algoritmul pentru n valori naturale, cuprinse între 0 și **vmax**, pe care le sortăm crescător:

int nr[**vmax**+1];//!!fac calculele, deci să am o constantă

for(i=0;i<=vmax;i++) vmax[i]=0;//umplem vectorul cu 0.

//obs: putem sări complet peste această etapă dacă declarăm vectorul global,

//însă dacă facem asta la bac trebuie să punem un comentariu imediat după ce

//l-am declarat, de genul "//fiind declarat global este inițializat cu 0"

for(i=1;i<=n;i++)

{

cin>>v;//v=valoarea de la pasul curent

nr[v]++;//o numărăm, adică incrementăm nr[v]

}

//pe baza lui nr afișăm valorile gata sortate:

for(i=0;i<=**vmax**;i++)

for(j=1;j<=nr[i];j++)

cout<<i<<" ";

**Obs**:Dacă aveți intenția de a optimiza prelucrările valorilor sortate, în sensul de a NU le mai lăsa să meargă de la 0 la vmax (vezi ultimul for) ci de la un min către un maxim, pe care le determinați în timpul numărării, acest lucru conduce de fapt la mărirea considerabilă a timpului de executare, deoarece de regulă la aceste probleme valorile de prelucrat sunt multe, iar valorile lor nu sunt foarte mari.

Astfel, intenția de mai sus s-ar traduce prin încărcarea fiecărui pas al for-ului de numărare cu două if-uri, ceea ce crește foarte mult timpii de executare (și încarcă și codul).

Morala: NU optimizați niciodată problemele de numărare calculând maxime și minime pentru limitele prelucrărilor.