Operatorii de atribuire

Am prezentat în lecția trecută citirea și scrierea ca fiind primele instrucțiuni. Spuneam că în limbajul C/C++ instrucțiunile sunt identificate altfel, dar putem face acest abuz de limbaj pentru a fi mai ușor de transmis informația începătorilor..

Intrucțiunea expresie

O scriere de forma: `expresie;` este considerată de limbaj ca fiind instrucțiune.

Exemple:

```
1;
a+b;        // a și b sunt variabile de tip int
(3*5)*r;    // r este variabila de tip double
```

Așadar, dacă scriem o astfel de linie într-un program C/C++ nu vom primi eroare de compilare. Evident, o astfel de linie nici nu are vreun efect pentru utilizator (cu toate că procesorul va face calculele). Sunt însă operanzi (cum este cel de atribuire, despre care vom învăța în continuare) care au efect asupra valorilor unor variabile și în acest fel găsim rațiunea pentru folosirea instrucțiunii expresie. Citirea și atribuirea sunt de fapt instrucțiuni expresie, dar noi ne vom referi în continuare la ele ca instrucțiuni independente.

Instrucțiunea bloc

Deseori este necesar să grupăm cumva **împreună mai multe instrucțiuni**. Acest lucru se face prin plasarea acestora între **paranteze acolade**. Din punct de vedere al limbajului C/C++, o pereche de acolade care grupează în interior mai multe instrucțiuni (sau declarații de date) formează o singură instrucțiune, instrucțiunea bloc. Acum probabil că vă gândiți la acoladele de la funcția main. Acolo nu este tocmai vorba despre instrucțiunea bloc ci despre corpul unei funcții care și ea își grupează ce are în interior între acolade.

Iată un exemplu:

```
#include <iostream>
using namespace std;
int n;

int main () {
    cin>>n;
    1;
    {
        cout<<"n=";
        cout<<n;
    }
}
```

Acesta este un program care cere sa introducem un număr întreg și care afișează mesajul `n=` urmat de valoarea introdusă.

Avem instrucțiunea expresie `1;` inutilă dar scrisă pentru a exemplifica, dar și acoladele de delimitare a blocului cu cele două afișări, și ele inutile dar folosite tot pentru exemplificare.

Același efect s-ar fi obținut dacă scriam codul astfel:

```
#include <iostream>
using namespace std;
int n;

int main () {
    cin>>n;
    1;
    cout<<"n=";
    cout<<n;
}
```

Din exemplele anterioare s-ar putea deduce că instrucțiunea bloc nu ar avea niciun efect niciodată. Vom vedea însă deosebita importanță a ei când vom discuta despre instrucțiunea `if` sau despre instrucțiunile repetitive.

Instrucțiunea de atribuire

Este de fapt o instrucțiune expresie, dar ca și la citire și afișare ne este mai comod să ne referim la ea în continuare ca la o instrucțiune separată.

Sintaxa:

```
variabila = expresie
```

Mod de executare

Acesta este al doilea mod de a modifica valoarea unei variabile. Dacă în cazul citirii valoarea care ajunge în variabilă este una introdusă de la tastatură, în cazul atribuirii valoarea este cea obținută în urma evaluării expresiei. Deci, se evaluează întâi `expresie` și rezultatul ei este copiat în variabilă. Ideal ar fi ca tipul rezultatului expresiei să fie același cu tipul variabilei. În caz contrar, de regulă nu apar erori de compilare, dar există unele reguli pentru a ști la ce rezultat să ne așteptăm.

De exemplu, dacă variabila este de tip `int` dar rezultatul expresiei depășește valoarea maximă ce poate fi stocată pe `int`, valoarea ce ajunge în variabilă este una trunchiată (se păstrează ultimii `16` biți ai rezultatului). Dacă variabila este de tip `int` iar expresia este reală, în variabilă se copiază partea întreagă a rezultatului.

Operatorul = este deci semnul distinctiv al atribuirii. Este important de înțeles următorul lucru: pe lângă faptul că se atribuie variabilei valoarea expresiei, **întreaga scriere este considerată tot o expresie a cărei valoare este cea atribuită**. Astfel, sunt permise atribuiri în cascadă ca în exemplul următor:

```
int a, b, c;
a = b = c = 2;
```

Mai sus nu apare eroare de compilare, efectul fiind că toate cele trei variabile ajung să memoreze valoarea `2`. Atribuirile se fac de la dreapta la stânga (de regulă știm că operatorii de aplică de la stânga la dreapta, dar aici este o excepție), și codul este echivalent cu următorul:

```
int a, b, c;
a = (b = (c = 2));
```

Se realizează întâi atribuirea `c=2` și toată această expresie are valoarea `2`, așadar acest `2` se atribuie mai departe lui `b` și apoi lui `a`.

Iată câteva exemple:

Exemplul 1

Pentru a calcula suma a două numere citite de la tastatură, am rezolvat în capitolul anterior astfel:

```
int a, b;
cin>>a>>b;
cout<<a+b;
```

Avem și alternativa folosirii instrucțiunii de atribuire si putem scrie:

```
int a, b, c;
cin>>a>>b;
c = a+b;
cout<<c;
```

Evident că prin exemplul de mai sus pare că mai degrabă ne complicăm folosind atribuirea, dar vom vedea mai departe cât de util poate fi să stocăm într-o variabilă rezultatul unui calcul (mai ales dacă același calcul ar trebui făcut în mod repetat).

Exemplul 2

```
int a;
cout<<(a=2);
cout<<a;
```

Acest cod are ca efect afișarea de două ori a valorii `2`. Prima instrucțiune de afișare pune `2` în variabila a dar apoi și afișează valoarea `2` (între paranteze este o expresie produsă de operatorul de atribuire și aceasta are ca valoare ceea ce s-a atribuit, adică `2`). A doua instrucțiune de afișare tipărește valoare unei variabile, care în acel moment este tot `2`, în urma atribuirii anterioare.

Exemplul 3

```
int a;
a = 1;
cout << a;
a = a + 3;
cout << a;
a = a * 2;
cout<<a;
```

În urma acestui cod se va afișa pe ecran: `148`. Valoarea variabilei `a` se modifică odată cu fiecare instrucțiune de atribuire. Observăm că la ultimele două atribuiri, la calculul noii valori a lui `a` se ia în calcul vechea valoare. De exemplu la ultima atribuire, mai întâi se evaluează expresia din partea dreaptă de `=`, a cărei valoare este `8` (în acel moment variabila a are valoare `4`), iar rezultatul obținut se copiază în variabila din stânga, care în acest caz este tot `a`.

Exemplul 4

```
3 = 2+1;
```

Această scriere va produce eroare de compilare. Conform sintaxei, prezența operatorului = impune ca în partea stângă să fie o varibilă, iar în cazul nostru este vorba despre o constantă.

Dintre cele mai importante aplicații ale atribuirii este operația de interschimbare a valorilor a două variabile. Așadar, se citesc de la tastatură două valori de același tip (să presupunem întregi) și se cere să afișăm în ordine inversă (decât cea de la citire) cele două valori. Soluția imediată este următoarea:

```
int a, b;
cin>>a>>b;
cout<<b<<" "<<a;
```

Secvența de mai sus rezolvă problema enunțată. Efectul pe ecran este cel de scriere în ordine inversă a celor două valori introduse. În acest caz valorile celor două variabile nu se modifică față de momentul citirii.

Iată și o altă soluție:

```
int a, b, c;
cin>>a>>b;
c = a;
a = b;
b = c;
cout<<a<<" "<<b;
```

Și în acest caz se obține același efect, dar de data asta se schimbă și valorile celor două variabile. Chiar dacă procedeul nu se justifică pentru această problemă, operația de interschimbare este foarte importantă, reprezentând, de exemplu, o cărămidă importantă pentru algoritmii de sortare (ordonare). Observăm că este necesară folosirea unei variabile suplimentare în care întâi copiem valoarea uneia dintre variabile. Acest lucru este necesar întrucât în urma unei atribuiri valoarea unei variabile se pierde. Mulți oameni se gândesc prima dată că acest lucru s-ar realiza astfel:

```
a = b;
b = a;
```

Evident că nu este corect pentru că după prima atribuire ambele variabile vor memora cea de-a doua valoare citită de la tastatură, iar prima valoare nu se mai găsește stocată niciunde.

Există diverse trucuri care duc la interschimbarea altfel a valorilor a două variabile. În exemplul următor obținem asta fără folosirea unei variabile suplimentare:

```
a = a+b;
b = a-b;
a = a-b;
```

Atenție însă la valoarea intermediară `a+b` care produce un rezultat cu valoare mai mare decât cele două valori de interschimbat, iar acesta să nu provoace depășirea a ceea ce putem memora pe tipul de date curent.

Putem obține același lucru și în alte moduri (folosind `*` și `/` în loc de `+` și `−`, precum și varianta cu operatorul pe biți `xor`).

Pentru a dispne de un set mai bogat de aplicații pe care să le putem aborda, este momentul să introducem un element nou prezent în limbaj.

Funcții predefinite

Operatorii fac parte din nucleul limbajului, dar ei nu asigură decât o parte dintre operațiile pe care am dori să le facem la un moment dat. Există o extensie foarte utilă și variată a lor: *funcțiile*. Acestea sunt subprograme care se află în anumite biblioteci (deci care sunt deja scrise) și care pot fi folosite în program. Singura condiție este să includem biblioteca în care ele se află. În plus, programatorul își poate defini propriile sale funcții, dacă dorește. Nu acum ne vom ocupa de modul de definire a unei funcții, dar putem spune ca acest lucru este asemănător cu scrierea programului principal (funcția main - deci, așa cum îi spune numele este tot o funcție, dar una specială în C/C++, pe care nu noi o apelăm, ci sistemul de operare). În acest material vom spune câte ceva doar despre funcțiile predefinite.

Modul de folosire a lor este următorul:

- se include biblioteca unde sunt definite: `#include <biblioteca>`
- se apelează funcția în program ori de câte ori este necesar folosind sintaxa:
`nume_funcție(parametru`$_1$`, parametru`$_2$`, ..., parametru`$_n$`)`

Parametrii sunt expresii prin care de regulă se trimit către funcție date de intrare. Unele funcții permit ca după apel să lase în parametri anumite valori calculate în funcție, iar în acest caz, când apelăm, este obligatoriu să folosim variabile pentru acei parametri.

În general, modul de returnare a unui rezultat dintr-o funcție se face chiar prin funcția însăși. Aceste funcții se numesc *funcții operand*. Ele se apelează ca parte a unei expresii, unde se folosește valoarea returnată. Este, deci, foarte important să cunoaștem tipul valorii returnate de funcție pentru a ști dacă ne este sau nu permis să o folosim într-un anume context. Există și funcții care nu returnează o valoare, ele se numesc *funcții procedurale* și se apelează de regulă ca instrucțiune separată.

Prima funcție pe care o vom introduce, al cărei rol este de a calcula rădăcina pătrată a unei valori numerice, este `sqrt`. Ea se găsește în biblioteca `cmath`, are un parametru numeric și rezultatul este real. O prezentăm astfel:

```
double sqrt(double parametru)
```

Valoarea `double` din față indică tipul rezultatului.

Exemple

`#include <iostream>`
`#include <cmath>`
`using namespace std;`
`int n;`
`int main () {`
`  cin>>n;`
`  cout<<sqrt(n);`
`  return 0;`
`}`
	Programul cere așadar să introducem un număr și afișează rădăcina sa pătrată. Este posibil să apelăm cu un parametru întreg dacă funcția cere o valoare reală (însă rezultatul funcției nu îl putem folosi decât ca număr real, de exemplu am fi primit eroare de compilare dacă încercam să scriem `sqrt(n) % 2`). Observați că am inclus două biblioteci.

Se citesc `3` numere naturale, pe care le notăm `a`, `b`, `c`, care reprezintă coeficienții unei ecuații de gradul II de forma `ax2+bx+c=0`. Datele de intrare garantează că ecuația are exact două rădăcini reale. Se cere să afișăm aceste rădăcini.

```
#include <iostream>
#include <cmath>
using namespace std;
int a, b, c, delta;
double x1, x2;
int main () {
    cin>>a>>b>>c;
    delta = b*b-4*a*c;
    x1 = (-b - sqrt(delta)) / (2*a);
    x2 = (-b + sqrt(delta)) / (2*a);
    cout<<x1<<" "<<x2;
    return 0;
}
```

Observați utilitatea instrucțiunii de atribuire la folosirea cantității de sub radical. Am calculat-o o dată, am memorat-o într-o variabilă și am folosit-o de două ori. Dacă scriam de două ori formula în cod, pe lângă calculul redundant am fi irosit și timp de executare. Observați plasarea numitorului între paranteze, în lipsa lor, conform ordinii de efectuare a operațiilor, am fi avut împărțire la `2` și apoi înmulțire cu `a`, ceea ce ar fi oferit rezultat eronat.

Acum putem extinde la forma finală definiția noțiunii de *expresie* (ceva care returnează o valoare) și anume:
- o constantă este o expresie;
- o variabilă este o expresie;
- *un apel de funcție este o expresie*;
- orice aplicare corectă de operatori asupra altor expresii este o expresie;

Compunerea operatorului de atribuire cu operatorii aritmetici

Așa cum am studiat în acest material, operatorul de atribuire (=) este cel care, pe de o parte ne permite să modificăm valoarea unei variabile iar pe de altă parte valoarea atribuită este rezultatul întregii expresii obținute prin aplicarea operatorului.

În multe cazuri, variabila la care se atribuie va apărea și în partea dreaptă a operatorului = adică noua sa valoare se va calcula în funcție de valoarea sa curentă.

Avem mai sus un astfel de exemplu pe care îl reluăm:

```
int a;
a = 1;
cout << a;
a = a + 3;
cout << a;
a = a * 2;
cout<<a;
```

Este ceea ce se întâmplă la a doua și la a treia atribuire.

Pentru astfel de situații, limbajul permite și folosirea următorilor operatori:

`+=`, `-=`, `*=`, `/=`, `%=` (un operator aritmetic urmat imediat de operatorul egal). Astfel, putem avea expresii de forma:

```
variabilă O= expresie
```

unde `O` poate fi oricare dintre operatorii aritmetici. Această scriere este echivalentă cu:

```
variabilă = variabilă O expresie
```

Nu am pus *punct și virgulă* la final pentru că este vorba de o expresie care poate fi folosită ca parte a uneia mai mari. Dar de cele mai multe ori este folosită ca o instrucțiune expresie separată, caz în care se pune la final *punct și virgulă*.

În urma aplicării acestor operatori, pe lângă faptul că se modifică valoarea variabilei, toată expresia are valoarea atribuită.

Exemple:

`int a;`
`a = 1;`
`cout << a;`
`a += 3;`
`cout << a;`
`a *= 2;`
`cout<<a;`
	Este scrierea echivalentă a codului discutat mai sus, prin folosirea operatorilor de atribuire compuși cu cei aritmetici (efecul este deci același, se tipărește `148`).

`int a;`
`a = 3;`
`cout << (a+=4);`
`cout << a;`
	Se tipărește de două ori valoarea `7`. La prima afișare se crește cu `4` valoarea lui `a`, dar totodată expresia obținută (care apare ca argument la afișare) are valoarea crescută.

`int a, b;`
`a = 3;`
`b = (a+=4);`
`cout << a << b;`
	De asemenea, se afișează de două ori `7`.

`int n;` `n /= 10;`	Valoarea variabilei `n` este înlocuită cu câtul împărțirii sale la `10`.

`int n;`
`n = 12;`
`n %= (2*5);`
	În partea dreaptă poate fi o expresie mai complexă, chiar dacă în exemplele anterioare am folosit de regulă o constantă sau o variabilă. Valoarea lui `n` devine `2` (restul împărțirii lui `12` la `10`).

`int n;` `n * = 12;`	Eroare de compilare (între simbolurile `*` și = am pus un spațiu și această scriere nu este recunoscută ca operator).
`int a, b;` `(a+b) += 3;`	Eroare de compilare. În partea stângă trebuie să fie o variabilă.

Atenție la folosirea operatorilor. Dintre următoarele variante:

a) `n=n+2;`
b) `n+=2;`
c) `n+2;`
primele două au ca efect modificarea valorii lui `n` pe când a treia nu (chiar dacă toate cele trei, privite ca expresii au aceeași valoare).

Prioritatea operatorilor de atribuire compuși nu este una mare, ca și cea a operatorului principal de atribuire. Nu încurajez memorarea ordinii de prioritate a tuturor operatorilor întrucât pe măsură ce îi vom introduce vom observa că devin mulți și există riscul de a ne încurca. Se recomandă folosirea cu încredere a parantezelor rotunde pentru a grupa convenabil.

Operatorii de incrementare / decrementare

Alți doi operatori pot fi trecuți în categoria celor de atribuire: `++` (operatorul de incrementare) și `--` (operatorul de decrementare). Ei sunt operatori unari, se așează lângă o *variabilă* și efectul final este de a crește respectiv de scădea cu `1` valoarea acelei variabile.

Pot fi plasați înaintea variabilei sau după ea, iar aici apare o nuanță în ceea ce privește valoarea expresiei finale, și acum vom lămuri.

```
int a;
a = 2;
cout << a;
a++;
cout << a;
```

Se afișează `23` (în urma aplicării operatorului `++` valoarea variabilei a crește cu `1`). Același efect l-am fi obținut dacă plasam operatorul înaintea variabilei `a`.

```
int a;
a = 2;
cout << a;
++a;
cout << a;
```

Deci, s-ar fi afișat `23`.

Pentru a observa diferența, vom studia următoarele două secvențe:

```
int a, b;
a = 2;
b = (a++);
cout << a << b;
```

```
int a, b;
a = 2;
b = (++a);
cout << a << b;
```

Secvența din stânga are ca efect afișarea `32` iar pentru cea din dreapta se afișează `33`. Motivul este următorul: chiar dacă *la final* efectul aplicării operatorului este de a modifica valoarea variabilei cu `1`, dacă expresia respectivă este parte a altei expresii mai mari, la plasarea operatorului în fața variabilei mai întâi se modifică variabila și noua sa valoare este cea care se ia mai departe în calcul în expresia mai mare, iar dacă operatorul este plasat după, este folosită în calculul expresiei mai mari valoarea variabilei de înainte de modificare și apoi aceasta este modificată.

Altfel, putem spune că, dacă `a` este o variabilă cu valoarea curentă `2`, spre exemplu, scrierea `a++` va modifica valoarea lui `a` la `3`, dar valoarea acestei expresii este `2`, iar scrierea `++a` va modifica valoarea lui `a` la `3` iar valoarea acestei expreii este `3`.

O utilizare de genul `(a+b)++` este incorectă sintactic întrucât lângă operator trebuie să se afle nimic altceva decât o variabilă.

Următoarele scrieri sunt diverse moduri de a mări cu `1` valoarea unei variabile întregi `a`:

```
a=a+1;
```

```
a+=1;
```

```
a++;
```

```
++a;
```

La declararea unei variabile în interiorul funcției `main` (și vom vedea că și în interiorul oricărei funcții ce o vom defini noi), nu ne putem baza pe o anume valoare inițială a acesteia (se preia valoarea dată de configurația de biți din zona de memorie alocată variabilei, configurație care poate fi oricare, rămasă, vom vedea mai târziu, de la calculele anterioare din zona de memorie numită stivă). Variabilele declarate în afara funcțiilor primesc valoarea inițială `0` (ele se alocă la începutul executării programului și atunci toți biții din zona de memorie în care ele se rezervă sunt `0`).

Putem însă, în momentul declarării, să stabilim noi valoarea inițială a unei variabile.

```
tip variabila = expresie;
```

Astfel, la declararea variabilei, aceasta va primi ca valoare inițială pe cea a expresiei. Trebuie să fim atenți că dacă declararea se face în afara funcțiilor expresia trebuie să fie una constantă (adică să poată fi clar evaluată de la începutul executării programului), iar dacă variabila este inițializată la declararea în interiorul unei funcții avem libertatea să folosim și o expresie care să conțină variabile. De regulă folosim ca valoare de inițializare o constantă:

```
int a;
int b = 2;
int main() {
    int c;
    int d = 3;
    cout<<a<<" "<<b<<" "<<c<<" "<<d;
}
```

La prima rulare a acestui cod pe calculatorul meu s-a afișat: `0 2 2686868 3`. La o altă rulare, în alt context, a treia valoare afișată ar putea fi alta (variabila `c` este declarată în interiorul funcției și nu putem conta pe o anume valoare inițială a sa). Oricum, este o chestiune cel puțin de bună practică de a inițializa orice variabilă.

În finalul acestei lecții vom învăța alți doi operatori. Ne permit fie să scriem codul mult mai compact în destule situații, fie să rezolvăm elegant diverse cerințe.

Operatorul de conversie (se mai numește **cast**)

Este unar iar modul său de utilizare este următorul:

```
(tip) expresie
```

`(tip)` putem considera că este operatorul.

Fiind vorba despre un operator, se formează o expresie, iar valoarea acesteia este cea obținută prin *convertirea* valorii celei din dreapta operatorului, la tipul de date dintre paranteze.

De exemplu, putem obține partea din fața separatorului, pentru o expresie reală, dacă o convertim la `int`.

`double x = 2.4;` `cout<< (int)x;`	Se afișează `2`
`double x = -2.4;` `cout<< (int)x;`	Se afișează `-2`
`double x = -2.7;` `cout<< (int)x;`	Se afișează `-2`

Atenție la folosirea operatorului, pentru a nu face confuzie cu modul de apel al funcțiilor. Când utilizăm operatorul avem tipul de date pus între paranteze rotunde, în fața expresiei, iar când apelăm o funcție avem argumentele puse între paranteze, după numele funcției.

Putem folosi operatorul de conversie pentru a forța o împărțire reală dacă avem operanzi întregi.

`cout<<5/4;`	Se afișează `1` (ambii operanzi sunt întregi), deci rezultatul este întreg, câtul.
`cout<<5/4.0;`	Se afișează `1.25` (unul dintre operanzi este real).

`cout<<5/(double)4;`	Se afișează tot `1.25` (am convertit la real al doilea operand, înainte de realizarea împărțirii).
`cout<<(double)5/4;`	Este același efect. Operatorul de conversie, fiind unar, are prioritate mare, deci în momentul aplicării lui `/` primul operand este deja real.
`int a = 5, b = 4;` `cout<<a/b;`	Se afișează `1` (ambii operanzi sunt întregi), deci rezultatul este întreg, câtul.
`int a = 5, b = 4;` `cout<<a/(double)b;`	Se afișează tot `1.25` (am convertit la real al doilea operand, înainte de realizarea împărțirii).
`int a = 5, b = 4;` `cout<<a/b.0;`	Eroare de compilare, nu putem scrie astfel decât dacă lucrăm cu operanzi constante reale.

Operatorul virgulă

Putem scrie în C/C++ astfel:

```
expresie1, expresie2, ..., expresien
```

Toată această scriere reprezintă o expresie. Adică, unind prin virgulă mai multe expresii obținem tot o expresie. Modul de evaluare este următorul: se evaluează în ordinea de la stânga la dreapta toate expresiile. Rezultatul ultimei expresii este totodată și rezultatul întregii expresii.

Exemple:

`cout<<(1,2);`	Poate părea ciudată scrierea și poate părea că este vorba de o încercare greșită de a scrie un număr real. Este însă altceva. Între paranteze avem o expresie obținută cu operatorul virgulă, deci rezultatul ei este dat de ultima subexpresie, care este `2`. Deci se va afișa pe ecran `2`.

`int a, b, c;`
`a = 2;`
`b = 3;`
`c = a, a = b, b = c;`
`cout<<a<<" "<<b;`
	Putem scrie și așa secvența de interschimbare a valorilor pentru două variabile.

`int a, b, c;`
`a = 2;`
`b = (a+=3, c=a+4, c--);`
`cout<<a<<" "<<b<<" "<<c;`
	Se afișează `5 9 8`. Iată justificarea: În dreapta atribuirii către `b` este o expresie cu operatorul virgulă, deci subexpresiile sale se vor evalua de la stânga la dreapta. Mai întâi `a` devine `5`, apoi `c` devine `9`, iar la a treia subexpresie, care va da și valoarea întregii expresii *virgulă*, se returnează valoarea lui `c` și apoi `c` este decrementat (operatorul `--` este aplicat la dreapta).

Vom vedea utilitatea operatorului mai cu seama la instrucțiunea for când vom dori să facem, de exemplu, mai multe inițializări deodată. Cum acestea se scriu într-un loc unde se cere o singură expresie, vom profita de acest operator pentru a compune o singură expresie din mai multe atribuiri.

Tema

Întrebări și exerciții

1. Care sunt modalitățile prin care putem schimba valoarea unei variabile?
2. Indicați sintaxa și modul de executare pentru instrucțiunea de atribuire.
3. Care sunt operatorii formați prin compunerea celui de atribuire cu operatorii aritmetici?

4. Indicați operatorii de incrementare/decrementare. Care este diferența între plasarea lor înainte respectiv după operand ?
5. Precizați sintaxa și efectul operatorului de conversie.
6. Precizați sintaxa și efectul operatorului virgulă.
7. Indicați două moduri diferite de a crește cu `2` valoarea variabilei `a` de tip `int`.
8. Considerând `a` ca fiind o variabilă de tip `int`, este corectă instrucțiunea: `++a++;` ? Justificați răspunsul.
9. Scrieți o secvență de atribuiri care, pentru trei variabile `a`, `b`, `c` cu valori citite de la tastatură, să permute circular la stânga cu o poziție valorile lor. Altfel spus, dacă în urma unei instrucțiuni `cin>>a>>b>>c;` s-ar introduce respective `1 2 3`, O instrucțiune de forma `cout<<a<<" "<<b<<" "<<c;` ar avea ca efect afișarea `2 3 1`. Dacă se consideră necesar, se pot folosi variabile auxiliare de tip `int`.

Probleme

1. Scrieți un program care cere de la tastatură un număr `a` (de o cifră) și care afișează valoarea expresiei a^{16}. Aplicați de cât mai puține ori operatorul de înmulțire (pbinfo #2599).
2. Scrieți un program care cere de la tastatură un număr natural `a` (`a <= 100`) și care afișează partea întreagă a valorii expresiei:

$$\frac{3*(a^2+a^4)}{a^2+a^4+\sqrt{a^2+a^4}} + \sqrt{a^2+a^4}$$ (pbinfo #2600)

3. Scrieți un program care calculează suma pătratelor cifrelor unui număr natural de trei cifre citit de la tastatură (pbinfo #2601).
4. Scrieți un program care cere de la tastatură un număr real (într-o variabilă de tip `double`) și care afișează ultima cifră a părții întregi a valorii citite (pbinfo #2602).
5. Scrieți un program care cere de la tastatură un număr real (într-o variabilă de tip `double`) și care afișează prima cifră a părții zecimale a valorii citite (pbinfo #2603).
6. Scrieți un program care cere de la tastatură un număr natural (care se dă de două cifre) și care afișează pe ecran pătratul valorii obținute prin schimbarea între ele a celor două cifre (pbinfo #2604).