Cursul 6

Expresii în C/C++
(I)

https://docs.microsoft.com/en-us/cpp/cpp/expressions-cpp

După cum am văzut, într-un program de calcul scris în limbajul C/C++ prelucrarea datelor se realizează printr-o serie de *apeluri* între *funcţii*, serie iniţializată de funcţia principală. In interiorul fiecărei funcţii prelucrarea datelor este *executată* de o secvenţă de comenzi numite *instrucţiuni*. In cadrul instrucţiunilor sunt *evaluate* diverse *expresii* aritmetice, logice sau de altă natură şi, în funcţie de rezultatele obţinute, se intervine sau nu asupra fluxului de execuţie al programului.

In limbajul C/C++, *expresiile* sunt secvenţe de *operatori* şi *operanzi* utilizate în unul sau mai multe dintre următoarele scopuri:

- calculul unei r-valori (*r-value*), a unei valori rezultat, care poate fi un număr, o structură sau o clasă. Denumirea vine de la *right-value*, adică valoarea care s-ar atribui obiectului din stânga operatorului de atribuire dacă expresia supusă evaluării ar fi operandul din dreapta.
- determinarea unei l-valori (*l-value*), a unei valori de locaţie, care desemnează calea către un anumit obiect. Analog cu mai sus, denumirea de *left-value* provine de la evaluarea expresiilor de atribuire: operandul din stânga este evaluat pentru a-i afla l-valoarea, adică locaţia de memorie unde trebuie copiată r-valoarea operandului din dreapta.
- Generarea de „efecte secundare, colaterale” *(side effects*). Prin efect secundar înţelegem orice altă acţiune care are loc odată cu evaluarea expresiei, în afara celei de a-i calcula r-sau/şi l-valoarea, de exemplu: modificarea valorilor unor variabile, schimbarea stării unor fluxuri de date, etc.

Orice expresie este de fapt o compunere de funcţii matematice scrisă în stil operatorial, operatorii fiind funcţii iar operanzii argumentele acestor funcţii. Operatorii pot fi *unari*, *binari* sau *ternari* (adică funcţii de unul, două sau respectiv trei argumente). De exemplu, adunarea numerelor este realizată de operatorul binar +, care este o funcţie cu două argumente, valoarea funcţiei în (x,y) fiind notată cu $x+y$ în loc de $+(x,y)$.

Expresiile pot fi definite recursiv astfel:

- o expresie este formată dintr-un operand sau dintr-un operator aplicat operanzilor săi;
- un operand este format dintr-un literal, un identificator de variabilă sau dintr-o expresie scrisă între paranteze rotunde.

De exemplu expresia a*(b+1) este formată din operatorul de multiplicare * aplicat operanzilor a şi (b+1), primul operand este un identificator de variabilă, al doilea este expresia b+1. La rândul ei, această expresie este formată din operatorul de adunare + aplicat unui identificator şi unui literal.

Pentru fiecare operator limbajul are reguli stricte pentru determinarea tipului rezultatului în funcţie de tipul operanzilor, astfel că orice expresie are rezultatul de un tip bine precizat.

1. R-valoarea unei expresii

Orice expresie în C/C++ care are un rezultat de tip diferit de **void** are r-valoare, adică rezultatul poate fi atribuit unei variabile de un tip bine precizat. Prin *atribuire* înţelegem copierea elementelor constitutive ale unei date într-o anumită locaţie de memorie alocată unei alte date de acelaşi tip. De regulă expresiile sunt evaluate în regiştrii procesorului şi de aici sunt preluate rezultatele, formate din unul sau mai multe numere care definesc obiectul rezultat.

Evaluarea unei expresii seamănă foarte mult cu apelul unei funcţii, este practic un apel de funcţie *inline*: în loc ca expresia sa fie compilată separat şi să avem un salt la codul rezultat, codul de evaluare al expresiei este inserat direct în codul programului. Mai mult, tipul rezultatului unei expresii (adică *tipul expresiei*) este supus aceloraşi restricţii de tip ca rezultatul returnat de funcţii: rezultatul poate fi numai de tip void, de tip numeric (aritmetic sau pointer) sau de tip structură sau clasă. Nu poate fi de tip tablou, de exemplu.

Subliniem această caracteristică a limbajului C/C++: toate expresiile de tip diferit de **void** au r-valoare, toate acestea pot sta în dreapta unei atribuiri, printre care chiar şi expresiile de atribuire (deoarece rezultatul evaluării unei atribuiri este chiar obiectul atribuit):

```
char x,y,z='A';
x=(y=z);
cout<<x<<y<<z<<endl; //AAA
y=7+(x=z);
cout<<x<<y<<z<<endl; //AHA
cout<<x<<7+(x=z)<<z<<endl; //A72A
```

Calculul cu obiecte de tip structură/clasă este foarte sărac, asupra operanzilor de acest tip acţionând numai un număr redus de operatori, printre care: operatorul de selecţie, operatorul de atribuire, if-ul aritmetic, operatorul de serializare. In C++, prin tehnica supraîncărcării operatorilor, proiectantul unei clase are posibilitatea extinderii sferei de aplicabilitate a operatorilor predefiniţi la obiectele clasei proiectate, dar acest subiect depăşeşte domeniul nostru de interes.

Rezultatul unei expresii nu poate fi un tablou sau o funcţie, dar poate fi un pointer către un tablou sau către o funcţie. In acest limbaj nu este implementat calculul vectorial, lucrul cu tablouri fiind posibil numai pe componente. Nu este implementat nici calculul funcţional, asupra unui operand de tip funcţie poate să acţioneze doar operatorul de apelare şi operatorul adresă. Calculul cu funcţii este totuşi posibil, dar în mod indirect, prin pointeri către funcţii.

In C/C++ nu există tipul de dată „expresie”, rezultatul evaluării unei expresii este o dată de un tip bine precizat, nu o altă expresie; limbajul nu are suport pentru calculul simbolic.

Calculul logic este înglobat în calculul numeric prin următoarea convenţie: singura valoare numerică falsă este zero, orice valoare nenulă este adevărată. Operatorii logici îşi evaluează operanzii conform acestei convenţii şi dau ca rezultat 0 pentru fals şi 1 pentru adevăr. In C++, pentru facilitarea calculului logic, a fost introdus tipul **bool** şi cuvintele cheie **false** şi **true**, dar pentru păstrarea compatibilităţii cu limbajul C, tipul **bool** din C++ este un tip întreg iar constantele **false** şi **true** au valorile numerice 0 şi respectiv 1, deci şi în C++ calculul logic este de tip numeric. In C# cele două tipuri de calcule au fost în sfârşit separate sintactic, conversiile între tipurile logice şi cele numerice fiind interzise.

In limbajul C/C++ cel mai bine reprezentat este calculul numeric, cu operanzi formaţi dintr-un singur număr. O expresie numerică are ca r-valoare un număr, care poate fi de tip aritmetic (întreg sau flotant) sau de tip pointer (o adresă). Cu astfel de expresii se pot determina

cantităţi, adrese de memorie sau valori logice. In continuare ne vom ocupa aproape în exclusivitate numai de astfel de expresii.

2. L-valorea unei expresii

Expresiile care stau în stânga operatorului de atribuire sunt evaluate pentru a determina o locaţie de memorie unde să fie depus rezultatul din dreapta. Spunem că aceste expresii sunt evaluate pentru a le determina l-valoarea.

In exemplul următor

```
int a,tab[3];
tab[2]=100;
a=13+tab[2];
```

în atribuirea `tab[2]=100` utilizăm numai l-valoarea expresiei `tab[2]` şi anume „elementul de indice 2 din tabloul `tab`", iar în atribuirea `a=13+tab[2]` utilizăm r-valoarea expresiei `tab[2]`, adică „valoarea numerică a elementului de indice 2 din tabloul `tab`". Evident că în acest din urmă caz este evaluată mai întâi l-valoarea expresiei `tab[2]`, după care este adusă din memorie r-valoarea sa şi adunată cu 13.

Nu toate expresiile pot sta în stânga operatorului de atribuire, deci nu toate au l-valoare. Literalii (constantele) nu au l-valoare. Variabilele, care în mod normal sunt l-valori, pot fi făcute nemodificabile utilizând cuvântul cheie **const**, şi atunci acestea nu mai pot sta în stânga unei atribuiri, nu mai au l-valoare.

Exemple:

```
i = 7;             // Corect. Un nume de variabila, i, are l-valoare.
7 = i;             // Eroare. O constanta, 7, are numai  r-valoare.
j * 4 = 7;         // Eroare. Expresia j * 4 are numai r-valoare.
*p = i;            // Corect. Tinta unui pointer este o l-valoare.
const int ci = 7; // Declara o variabila const.
ci = 9;            // ci este o  variabila nemodificabila, obtinem
                   //  un mesaj de eroare la compilare.
((i < 3) ? i : j) = 7; // Corect. Operatorul conditional (? :)
                   // returneaza in acest caz o l-valoare.
```

3. Efecte secundare

După cum am văzut, evaluarea unei expresii constă în determinarea unui rezultat (r-valoarea expresiei) sau a unei locaţii (l-valoarea expresiei) şi depunerea (notarea, consemnarea) temporară a rezultatului undeva, de regulă în regiştrii procesorului. Evaluarea decurge recursiv, evaluarea unui operator fiind precedată de evaluarea operanzilor săi. Evaluarea unei variabile constă în notarea valorii acelei variabile.

Expresiile `i+1` şi `++i` au acelaşi rezultat, totuşi între ele există o deosebire esenţială: în urma evaluării expresiei `i+1` valoarea variabilei `i` nu se modifică, în schimb evaluarea expresiei `++i` modifică valoarea lui `i`, incrementându-l. Spunem că expresia `++i` are efecte secundare, deoarece o dată cu evaluarea ei se produc modificări în memorie, modificări care rămân persistente după încetarea evaluării.

Orice expresie de atribuire are ca efect secundar scrierea unei date în memorie, şi în acest caz efectul secundar este chiar scopul principal al utilizării expresiei. Apelurile de funcţii pot avea (şi de regulă au) efecte secundare.

O instrucţiune expresie, adică o instrucţiune care se reduce la o expresie urmată de caracterul terminator „;”, este acceptată de compilator dar nu este tradusă în cod executabil dacă expresia nu are efecte secundare:

```
int i=1,j=2;
i+j;          //warning C4552: '+' : operator has no effect;
              //expected operator with side-effect
cout<<i+j<<endl;
```

Programatorul poate utiliza efectele secundare pentru a scurta şi, eventual, a eficientiza codul. De exemplu, următoarea secvenţă de două instrucţiuni:

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

poate fi scrisă pe scurt

```
a=tab[++i];
```

Pe de altă parte, utilizarea în exces a efectelor secundare poate duce la scrierea unor expresii ambigue, a căror evaluare depinde de compilator. Principala sursă de ambiguitate se datorează faptului că limbajul, cu puţine excepţii, nu prevede *ordinea evaluării operanzilor* în cazul operatorilor binari şi nici *ordinea evaluării parametrilor actuali* în cazul apelurilor de funcţii.

De exemplu, în urma execuţiei programului următor

```
#include<iostream>
using namespace std;
int i;
int zero(){
            i=0;
            return 0;
}
int main(){
            i=1;
            int a=i*i-zero();
            i=1;
            int b=1*i-zero();
            cout<<"a="<<a<<endl;
            cout<<"b="<<b<<endl;
            return 0;
}
/*REZULTAT:
a=1
b=0
Press any key to continue . . .*/
```

ne aşteptăm ca **a** şi **b** să fie egale. Nu sunt aşa deoarece la calculul diferenţei `i*i-zero()` compilatorul a început cu evaluarea primului operand, produsul `i*i`, iar la diferenţa `1*i-zero()`, din motive de eficienţă, a început cu evaluarea celui de al doilea operand, apelul `zero()` :

```
            i=1;
00D5152E  mov         dword ptr [i (0D59148h)],1
            int a=i*i-zero();
00D51538  mov         esi,dword ptr [i (0D59148h)]
00D5153E  imul        esi,dword ptr [i (0D59148h)]
00D51545  call        zero (0D5125Dh)
00D5154A  sub         esi,eax
00D5154C  mov         dword ptr [a],esi
```

```
            i=1;
00D5154F  mov         dword ptr [i (0D59148h)],1
            int b=1*i-zero();
00D51559  call        zero (0D5125Dh)
00D5155E  mov         ecx,dword ptr [i (0D59148h)]
00D51564  sub         ecx,eax
00D51566  mov         dword ptr [b],ecx
```

O altă sursă de ambiguitate este dată de faptul că limbajul nu precizează momentul exact al completării efectelor secundare ale operatorilor de incrementare/decrementare, lăsând astfel la latitudinea constructorului compilatorului alegerea acestui moment. De exemplu, următoarea instrucţiune

```
cout<<(i++)-(i++)<<endl;
```

poate scrie pe monitor -1, 0 sau 1, după cum incrementarile au loc imediat după ce s-a notat valoarea variabilei sau la sfârşitul evaluării întregii expresii (aici mai intervine şi ordinea în care sunt evaluaţi operanzii scăderii).

In concluzie, utilizarea efectelor secundare este necesară şi benefică, dar utilizarea lor în exces poate conduce la expresii ambigue, iar astfel de situaţii trebuie evitate pentru a avea un cod clar şi portabil.

4. Ordinea evaluării operatorilor

Spre deosebire de ordinea evaluării operanzilor, care în anumite situaţii nu poate fi decisă, programatorul are posibilitatea să precizeze totdeauna *ordinea evaluării operatorilor*, prin folosirea parantezelor rotunde. De exemplu, în expresia **(a*b)+c** ultimul este evaluat operatorul de adunare, iar în **a*(b+c)** ultimul este evaluat operatorul de înmulţire.

Pentru a reduce numărul parantezelor necesare precizării acestei ordini, limbajul C defineşte pentru fiecare operator câte un *nivel de prioritate* şi câte o ordine de *grupare* (sau de *asociere*) a operatorilor cu acelaşi nivel de prioritate.

Nivelul de prioritate este invers proporţional cu tăria legăturii dintre operator şi operanzii săi, operatorii cu prioritatea I fiind cei care leagă cel mai tare. De exemplu, expresia **a*b+d*e** este interpretată de către compilator sub forma obişnuită **(a*b)+(d*e)** deoarece operatorul multipicativ * are prioritatea IV iar operatorul aditiv + are prioritatea V.

Utilizarea nivelelor de prioritate simplifică mult scrierea expresiilor şi creşte lizibilitatea lor. Totuşi, într-o expresie în care apar operatori mai rar folosiţi, când nu suntem siguri asupra nivelelor de prioritate, este recomandată utilizarea parantezelor rotunde pentru clarificarea evaluării.

Ordinea de asociere a operatorilor precizează ordinea de efectuare a operaţiilor în cazul unei succesiuni de operatori cu aceeaşi prioritate.

Expresiile cu operatori unari se grupează plecând de la operand, deci de la dreapa la stânga în cazul operatorilor prefixaţi şi în ordine inversă în cazul postfixat. Dacă **op1**, **op2** şi **op3** sunt operatori unari prefixaţi cu acelaşi nivel de prioritate, atunci expresia

op1 op2 op3 *operand*

este echivalentă cu

op1(**op2**(**op3** *operand*)).

Un exemplu în cazul operatorilor postfixati: expresia `a++++` este interpretată de compilator sub forma `(a++)++` (şi în consecinţă este ilegală, deoarece `a++` nu este o l-valoare!).

Expresiile cu operatori binari se grupează de la stânga la dreapta, dacă **op1**, **op2** şi **op3** sunt operatori binari cu acelaşi nivel de prioritate, atunci expresia

exp1 **op1** *exp2* **op2** *exp3* **op3** *exp4*

este echivalentă cu

(((*exp1* **op1** *exp2*) **op2**) *exp3*) **op3** *exp4.*

Subliniem iarăşi că prioritatea operatorilor şi ordinea lor de asociere precizează numai modul de grupare a sub-expresiilor pentru a forma operanzi, dar nu precizează nici ordinea de evaluare a operanzilor şi nici ordinea şi nici momentul exact al apariţiei efectelor secundare. Astfel, ştim că la evaluarea expresiei a*b-c*d ultima operaţie care se execută este scăderea dar nu este precizat nicăieri în ce ordine sunt evaluaţi ce doi operanzi ai operatorului minus: întâi se evaluează şi se notează undeva rezultatul produsului a*b şi apoi al lui c*d, sau se procedează în ordine inversă? In acest caz (deoarece expresia nu are efecte secundare) rezultatul este acelaşi indiferent de ordinea evaluării operanzilor, dar în alte situatii rezultatul ar putea fi diferit, după cum am văzut mai sus.

5. Operatorii limbajului C

Limbajul C/C++ are puţine instrucţiuni dar mulţi operatori, iar numărul lor creşte la fiecare etapă de dezvoltare a limbajului. In fişierul **operatori.pdf** sunt prezentaţi cei 47 de operatori ai limbajului C dispuşi pe 16 nivele de prioritate. In C++ au fost introduşi noi operatori, care au schimbat nivelurile iniţiale de prioritate dar fără să schimbe vechea ordine. De exemplu, a fost introdus operatorul de rezoluţie :: cu prioritatea cea mai tare. Prezentarea acestor noi operatori o amânăm până când vom avea nevoie de ei.

5 A. Operatori postfixaţi.

Niv	Nr	Simbol	Operator / Operaţie	Apelare	As.	RV	LV	SE	EO
II	1	**( )**	apel funcţie	funcţie(lista)	>>>	cto	Lv	SE	-
	2	**[ ]**	indexare tablou	tablou[indice]	>>>	cto	Lv	-	-
	3	**. ->**	selecţie membru	structura.membru	>>>	cto	Lv	-	-
	4	**++ --**	incrementare/decrementare	c++	>>>	cto	-	SE	-

A1. *Operatorul de apelare a unei funcţii*, numit şi *operatorul paranteze rotunde*, nu trebuie confundat cu perechea de paranteze rotunde care închide o expresie pentru a forma un operand.

Din punctul de vedere al sintaxei expresiilor operatorul paranteze rotunde are un singur operand, cel scris în faţa parantezelor şi care desemnează funcţia apelată. Lista argumentelor nu este un operand deoarece nu este o expresie operatorială, ea are sintaxă proprie şi este tratată diferit de compilator.

A2. *Operatorul de indexare* sau *operatorul paranteze pătrate*, este utilizat pentru parcurgerea tablourilor. El are doi operanzi, unul trebuie să fie un pointer către primul element al tabloului iar

celălalt numărul de ordine al elementului vizat. Amintim că numele unui tablou este în acelaşi timp o constantă de tip pointer către primul element al tabloului.

```
#include<iostream>
using namespace std;
/*    Exemple (pentru avansati) de utilizare
*     a operatorilor de indexare si de apelare
*/

int fSum(int a, int b){
      return a + b;
}

int fDif(int a, int b){
      return a - b;
}

int(*fGen(int p))(int, int)
{                              //functia fGen are un singur argument
      if (p >= 0) return fSum;//si intoarce un pointer
      return fDif;             //catre o functie cu doua argumente
}
//intregi si rezultat de tip int
int main(void){
      int i = 1, x = 2, y = 3;
      int p = 4;
      int z;
      z = fGen(p)(x, y);       //fGen(p) este un "nume" de functie
      cout << "z=" << z << endl;    //z=5

      int(*pTab[2])(int, int);//pTab este un tablou de pointeri catre
      pTab[0] = fSum;          //functii cu doua argumente int
      pTab[1] = fDif;          //si cu rezultat de tip int
      z = pTab[i](x, y);       //pTab[i] este un "nume" de functie
      cout << "z=" << z << endl;    //z=-1
      return 0;
}
```

A3. *Operatorii de selecţie* sunt utilizaţi la desemnarea componentelor obiectelor de tip structură sau clasă. Există doi astfel de operatori: operatorul de selecţie directă (*operatorul punct*) şi operatorul de selecţie indirectă (*operatorul săgeată*). Exemplu:

```
#include<iostream>
using namespace std;
struct Punct{                        //declaratie de structura
      int  x;
      int y;
};
int main(){
      Punct A={10,20};
      Punct *adresaLuiA;
      adresaLuiA=&A;
      cout<<A.x<<endl;               //10
      cout<<adresaLuiA->y<<endl;     //20
      return 0;
}
```

Vom reveni asupra acestor operatori când vom studia datele de tip structură.

A4. *Operatorii de incrementare/decrementare postfixaţi* (operatorii *plus plus* / *minus minus*) se aplică unui operand cu l-valoare şi au ca rezultat valoarea numerică iniţială a operandului. Expresia are ca efect secundar modificarea prin incrementare/decrementare a operandului, această modificare survenind într-un moment neprecizat de după notarea rezultatului şi până la terminarea evaluării expresiei curente.

Incrementarea/decrementarea unui număr întreg sau flotant constă în adunarea/scăderea unei unităţi la acel număr, iar incrementarea/decrementarea unui pointer constă în modificarea valorii adresei astfel încât pointerul să ţintească spre următoarea/precedenta locaţie.