Expresii în C/C++
(III)

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

5 D. Operatori ternari.

Niv	Simbol	Operator / Operaţie	Apelare	As.	RV	LV	SE	EO
XV	**? :**	operatorul condiţional	cond ? Da : Nu	---	cto	Lv	-	>>

Limbajul C are un singur operator ternar (adică un operator cu trei operanzi), *operatorul condiţional* `? :` , numit şi *operatorul* ***if aritmetic.***

Operatorul are următoarea sintaxă:

`expresie ? expresie : expresie`

iar evaluarea sa decurge astfel: se începe cu evaluarea primului operand – care trebuie să fie de tip logic sau numeric – şi, înainte de a trece mai departe, se completează toate eventualele efecte secundare ale acestuia. Dacă rezultatul are valoarea logică „adevărat” atunci se trece la evaluarea celui de al doilea operand, în caz contrar se trece la evaluarea celui de al treilea. In orice situaţie, dintre ultimii doi operanzi este evaluat numai unul, iar rezultatul obţinut este rezultatul final al întregii expresii.

Exemplu

```
double x, a=1.0;
x=(a++<20 ? a : a-100);
cout<<"x="<<x<<endl; //x=2
```

Operanzii de pe locurile doi şi trei trebuie să aibe acelaşi tip sau să existe conversii implicite la un tip comun. Tipul rezultatului este tipul comun minimal al acestor doi operanzi.

```
#include<iostream>
using namespace std;
struct Complex{double x, y;};
int main(){
      Complex u={1,2},v={10,20};
      cout<<(u.x >= v.x ? u : v ).y<<endl; //20
      return 0;
}
```

Rezultatul if-ului aritmetic are l-valoare numai în cazul în care ambii operanzi de pe locurile doi şi trei au l-valoare şi sunt de acelaşi tip.

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

double f(double x){
	return x>0 ? x*x+1.0 : (x==0 ? 0 : x*x-1.0);
}

int main(){
	double a=1.0, b=2.0;
	(a < b ? a : b) = f(-10.0);
	cout<<"a="<<a<<" b="<<b<<endl;
	return 0;
}
/*
REZULTAT:
a=99 b=2
Press any key to continue . . .*/
```

5 E. Operatorii de atribuire.

Niv	Nr	Simbol	Operator / Operaţie	Apelare	As.	RV	LV	SE	EO
XVI	1	**=**	atribuire simplă	*l-val* = expresie	<<<	cto	L*	SE	-
	2	**+= -= *= /= %=**	atribuire compusă aritmetică	*l-val* += termen *l-val* *= factor	<<<	cto	L*	SE	-
	3	**<<= >>= &= ^= |=**	atribuire compusă pe biţi	*l-val*<<=depl *l-val* &= masca	<<<	cto	L*	SE	-

In programare, atribuirea unei valori unei variabile constă în determinarea acelei valori şi înscrierea ei în locaţia de memorie alocată acelei variabile. Orice atribuire presupune mutarea unor date între regiştrii microprocesorului şi diverse locaţii de memorie prin utilizarea de instrucţiuni *mov* în limbaj de asamblare.

Specific limbajului C este efectuarea atribuirilor cu ajutorul operatorilor de atribuire, ca efect secundar al evaluării *expresiilor de atribuire*, şi nu prin executarea unei instrucţiuni de atribuire. Acestă atribuire „din mers”, în cadrul evaluării unei expresii, permite declanşarea de atribuiri în locuri în care limbajul nu permite utilizarea unei instrucţiuni, de exemplu în timpul testării condiţiei de continuare a unei instrucţiuni de ciclare. Exemplu:

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

const int dim=100;
int main(){
	char text[dim]="tabloul text este accesat o singura data";
	char ch;
	for(int i=0; i<dim && (ch=text[i])!='\0'; i++)
		cout<<ch<<" "<<(int)ch<<endl;
	return 0;
}
```

Mai mult, deoarece orice expresie de atribuire are o valoare rezultat, este posibil şi un calcul aritmetic cu atribuiri, util în unele situaţii. De exemplu, în secvenţa de program

```
int tab[dim]={1,2,3};
int tab_rezerva[dim];
int s=0;
for(int i=0; i<dim; i++)
      s+=(tab_rezerva[i]=tab[i]);
cout<<s<<endl;
```

tabloul **tab** este sumat şi copiat în acelaşi timp.

Limbajul C dispune de un operator de atribuire simplă, care provoacă transferul obişnuit al datelor, şi de 10 operatori de atribuire compusă, care declanşează aplicarea „pe loc", în locaţia desemnată de operandul stâng, a unor operatori binari.

Operandul stâng al oricărui operator de atribuire trebuie să desemneze locaţia unei date modificabile, altfel spus trebuie să aibe l-valoare. In cazul atribuirii simple operandul stâng este evaluat numai pentru determinarea l-valorii sale.

Operanzii oricărei atribuiri trebuie să aibe acelaşi tip sau, dacă nu, trebuie să existe o conversie implicită a tipului din dreapta la tipul operandului din stânga. Rezultatul atribuirii este întodeauna valoarea desemnată de operandul din stânga, după efectuarea modificărilor în memorie. Limbajul C nu cere ca rezultatul expresiilor de atribuire sa aibe şi l-valoare, acest lucru este cerut însă în C++.

```
#include<iostream>
using namespace std;
struct Punct{ int x, y; };
int main(){
      Punct u = { 1, 2 }, v;
      cout << (v = u).y << endl;//2
return 0;
}
```

Evaluarea unei atribuiri începe cu evaluarea expresiei din dreapta, valoarea obţinută este apoi transferată sau, după caz, operată cu cea aflată în locaţia desemnată de expresia din stânga.

E1. *Operatorul de atribuire simplă* = poate avea operanzi de tip numeric (aritmetic sau pointer) sau obiecte de tip structură sau clasă. Nu se pot atribui tablouri sau funcţii.

Atribuirile pot fi concatenate fără utilizarea parantezelor deoarece ordinea lor de asociere este de la dreapa la stânga. Expresia

```
a=b=c=10.0;
```

este citită de compilator sub forma

```
a=(b=(c=10.0))
```

şi are ca rezultat valoarea actualizată a variabilei **a**. In timpul evaluării au fost actualizate şi variabilele **b** şi **c**.

Expresia

```
a=1+b=1+c=10.0;
```

nu trece de compilare deoarece este citită ca

```
a=((1+b)=((1+c)=10.0))
```

şi conţine expresii fără l-valoare poziţionate în stânga operatorului de atribuire.

Expresia

```
a=1+(b=1+(c=10.0))
```

este corectă şi are ca rezultat numărul 12.0. După evaluare variabilele a, b şi c au valorile 12, 11 şi, respectiv, 10.

Expresia

`a=(b=1)+(b=2)`

este ambiguă deoarece depinde de ordinea şi momentul exact al completării efectelor secundare. Pe compilatorul Ms Visual Studio 2017 are ca rezultat numărul 4 deoarece adunarea a fost programată după executarea atribuirilor `b=1` şi `b=2` în ordinea de la stânga la dreapta:

```
      a = (b = 1) + (b = 2);
002A5E9E  mov         dword ptr [b],1
002A5EA5  mov         dword ptr [b],2
002A5EAC  mov         eax,dword ptr [b]
002A5EAF  add         eax,dword ptr [b]
002A5EB2  mov         dword ptr [a],eax
```

Dacă evaluarea operanzilor adunării ar fi fost efectuată de la dreapta la stânga, rezultatul ar fi fost altul.

E2. *Operatorii aritmetici de atribuire compusă* au forma „op=" cu „op" unul dintre operatorii aritmetici *, %, /, + sau -. Operanzii trebuie să fie numerici, în cazul pointerilor se aplică regulile de calcul cu pointeri. Evaluarea expresiei de atribuire compusă începe cu determinarea r-valorilor celor doi operanzi, acestea sunt apoi operate între ele iar rezultatul este depus în locaţia desemnată de operandul stâng.

Regula practică de evaluare este următoarea: punem „mintal" operandul drept între paranteze rotunde, ştergem semnul =, evaluăm expresia rămasă şi atribuim rezultatul obţinut operandului stâng. Exemplu:

```
int  a=2,b=10;
a*=b+1;
//calculam a*(b+1)si
//rezultatul il atribuim lui a
cout<<"a="<<a<<endl;//a=22
```

Posibilitatea calculului „pe loc" introdusă de atribuirea compusă este o facilitate importantă a limbajului C care trebuie însuşită şi folosită din plin de către programatori. Atribuirea compusă trebuie gândită ca o acţiune asupra operandului stâng: expresia `a+=10` *îl măreşte* pe **a** cu 10 unităţi, `a-=10` *îl micşorează*, iar `a*=10` *îl amplifică* pe **a** cu 10.

Sumele se calculează prin *acumularea* valorilor sumate într-o variabilă iniţializată cu 0

```
double suma=0;
for(int i=0; i<dim; i++)
      suma+=tab[i];
cout<<suma<<endl;
```

iar produsele prin *amplificarea* unei variabile iniţializate cu 1

```
double prod=1;
for(int i=0; i<dim; i++)
      prod*=tab[i];
cout<<prod<<endl;
```

La evaluarea expresiei de acumulare `a+=t` operandul **a** este evaluat o singură dată, iar la evaluarea expresiei `a=a+t` operandul **a** este evaluat de două ori, prin urmare în cazul prezenţei efectelor secundare cele două expresii nu sunt echivalente:

```
#include<iostream>
using namespace std;
const int dim=10;
int main(){
	int i, a[dim];

	for(int k=0;k<dim; k++) a[k]=10*k;
	i=1;
	a[++i]+=1111;
	for(i=0;i<dim;i++)
		cout<<a[i]<<" ";
	cout<<endl;

	for(int k=0;k<dim; k++) a[k]=10*k;
	i=1;
	a[++i]=a[++i]+1111;
	for(i=0;i<dim;i++)
		cout<<a[i]<<" ";
	cout<<endl;
	return 0;
}
/*REZULTAT:
0 10 1131 30 40 50 60 70 80 90
0 10 20 1141 40 50 60 70 80 90
Press any key to continue . . .*/
```

E3. *Operatorii pe biţi de atribuire compusă* <<=, >>=, &=, ^=, |= execută pe loc operaţiile binare pe biţi. Operanzii trebuie să fie de tip întreg, cadrul natural de lucru fiind tipul **unsigned**.

De exemplu, considerând masca de biţi

```
unsigned m3=1u<<3;
```

expresia `a|=m3` seteză bitul de ordin 3 al lui **a** (îi dă valoarea 1), `a&=~m3` îl şterge, iar `a^=m3` îl basculează:

```
#include<iostream>
using namespace std;
void afiseaza(unsigned a){
	//afiseaza bitii lui a
	for(int k=31;k>=0;k--)
		cout<<!!(a&(1u<<k));
	cout<<endl;
}
int main(){
	unsigned m3=1u<<3;
	unsigned a=0xff77u;
	afiseaza(a);
	a|=m3;
	afiseaza(a);
	a&=~m3;
	afiseaza(a);
	a^=m3;
	afiseaza(a);
	return 0;
```

```
}
/*REZULTAT:
00000000000000001111111101110111
00000000000000001111111101111111
00000000000000001111111101110111
00000000000000001111111101111111
Press any key to continue . . .*/
```

Niv	Simbol	Operator / Operaţie	Apelare	As.	RV	LV	SE	EO
XVIII	,	operatorul virgulă	expr , expr	>>>	cto	L*	-	>>

5G. Operatorul de serializare.

Limbajul C dispune de un singur operator pentru serializarea evaluărilor unui şir de expresii, şi anume *operatorul virgulă*, care este un operator binar aplicat unor operanzi de orice tip.

Expresia *operand1 , operand2* este evaluată strict în ordinea de la stânga la dreapta, întâi este evaluată expresia *operand1* şi sunt completate efectele sale secundare, apoi este evaluată expresia *operand2* şi rezultatul obţinut este rezultatul final al întregii expresii. Primul operand este evaluat numai pentru generarea de efecte secundare, rezultatul său se pierde. Exemplu:

```
double x,y;
y=(x=4,x*x);
cout<<y<<endl;//16
```

Ordinea de asociere a operatorului virgulă este de la stânga la dreapta, astfel încât concatenarea sa se execută firesc în acestă ordine:

```
cout<<(x=13,y=100,x*y)<<endl;//1300
```

Serializarea expresiilor este utilă în cazul în care dorim să fie evaluate mai multe expresii într-un loc din program unde este permisă scrierea numai uneia singure. Un caz tipic este efectuarea mai multor acţiuni la iniţializare sau la reluarea ciclării într-un **for**:

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

int main(){
	int tab[10]={0,11,22,33,44,55,66,77,88,99};
	int i,j,aux;
	for(i=0,j=9; i<j; i++,j--)
		aux=tab[i], tab[i]=tab[j], tab[j]=aux;
	for(i=0;i<10;i++)
		cout<<tab[i]<<" ";
	cout<<endl;
	return 0;
}
/*REZULTAT:
99 88 77 66 55 44 33 22 11 0
Press any key to continue . . .*/
```

Virgula operatorului de serializare nu trebuie confundată cu virgula care separă itemii listelor la declararea mai multor variabile, sau care separă parametrii unei funcţii.

```
#include<iostream>
using namespace std;
void afiseaza(char a, char b){
	cout<<a<<b<<endl;
}
int main(){
	afiseaza('X',('Y','Z'));
	//XZ
	afiseaza('X','Y','Z');
	//error C2660: 'afiseaza' : function does not take 3 arguments
	return 0;
}
```

6. Claritatea codului

Limbajul C/C++ are, după cum am văzut, numeroase posibilităţi de a scrie expresii de calcul numeric şi de calcul logic. Mai mult, utilizând if-ul aritmetic, operatorul de serializare şi calculul cu atribuiri, se pot forma expresii complexe a căror evaluare este echivalentă cu parcurgerea unei întregi secvenţe de instrucţiuni. De exemplu, secvenţa de cod

```
	int a=1,b,x;
	cin>>b;
	if(a<b) {
		cout<<"a<b"<<endl;
		x=b;
	}
	else {
		cout<<"a>=b"<<endl;
		x=a;
	}
	cout<<x<<endl;
```

poate fi scrisă compact sub forma

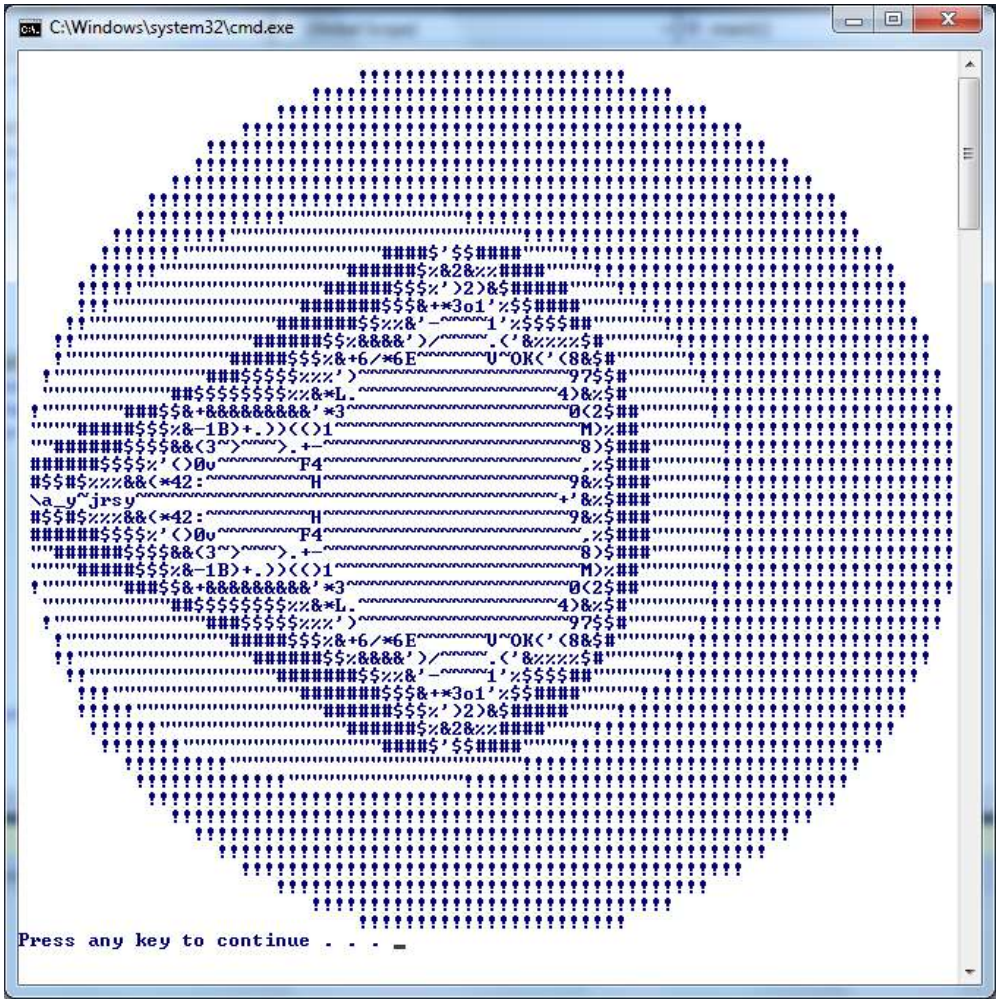
```
int a=1,b,x;
cout<<(x=a<(cin>>b,b)?(cout<<"a<b"<<endl,b):(cout<<"a>=b"<<endl,a))<<endl;
```

O astfel de compactificare nu este cu nimic justificată, prezenţa în acest caz a operaţiilor de intrare/ieşire făcând inutilă compararea vitezelor de execuţie. In general, dacă nu aduc o simplificare evidentă a implementării algoritmului de calcul, astfel de compactificări nu fac decât să reducă claritatea codului, îngreunând depanarea şi actualizarea programelor.

În final, un ultim contra-exemplu: programul următor trasează la consolă mulţimea lui Mandelbrot[1] prin metoda timpului de scăpare (*escape-time algorithm*), utilizând un singur **for** în loc de trei **for**-uri imbricate (după **b**, **a** şi **ch**):

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


int main(){
	double x=0, y=0, xx=0, yy=0, a=-2, b=-2;
	for(char ch=30;
		b+= a>2 ? -0.04*(a=-2) : 0 , b<2;
		ch++<125 & (xx=x*x)+(yy=y*y)<4 ?
		(y=2*x*y+b, x=xx-yy+a):(cout<<ch, ch=30, x=y=0, a+=.05)
		);
	return 0;
}
```



[1] Mulţimea lui Mandelbrot este mulţimea acelor numere complexe $u=a+ib$ pentru care toţi termenii şirului recurent $z_0=0$, $z_{n+1}=z_n^2+u$ sunt în discul de rază 2 centrat în originea planului numerelor complexe.

Aici **ch** reprezintă "culoarea" numărului complex $u=a+ib$, cu $-2<a<2$ şi $-2<b<2$, dată de numărul de iteraţii necesar şirului recurent $z_0=0$, $z_{n+1}=z_n^2+u$ să iasă din discul de rază 2 centrat în originea planului numerelor complexe. Termenii şirului (z_n) sunt calculaţi pe loc în variabila $z=x+iy$.

Iată şi varianta "în clar" a programului precedent:

```
#include<iostream>
using namespace std;
int main(){
	double x=0, y=0, xx=0, yy=0, a, b;
	char ch=0;
	for(b=-2; b<2; b+=0.08)
		for(a=-2; a<2; a+=0.05){
			//determinam ch-ul lui u=a+ib
			for( ch=30, x=y=xx=yy=0; ch<126 && xx+yy<4; ch++){
				xx=x*x;
				yy=y*y;
				//calculam pe componente
				y=2*x*y+b;		//z=z*z+u
				x=xx-yy+a;		//cu z=x+iy
			}
			cout<<ch;
		}
	return 0;
}
```

Programul presupune că lungimea liniei consolei de ieşire este de 80 de caractere.