Cursul 5

Noţiuni de bază (II)

4. Scrierea constantelor

Numim *constantă* o dată anonimă şi invariabilă, a carei valoare este scrisă direct în codul sursă al programului printr-un *literal* alcătuit conform unor reguli stricte de scriere, reguli care precizează atât valoarea cât şi tipul constantei. De exemplu, expresia `421+421.0+a` este formată în ordine din: literalul `421`, simbolul + al operatorului de adunare, literalul `421.0`, iarăşi simbolul adunării, şi din identificatorul `a`. Expresia desemnează suma a trei termeni, primul termen este o constantă de tip **int**, al doilea o constantă de tip **double** iar al treilea o variabilă cu numele **a**.

Odată cu extinderea limbajului C la C++ a fost introdus modificatorul **const** care transformă o variabilă într-o dată nemodificabilă, o „*variabilă constantă*". Această facilitate este utilă, mai ales, în situaţia în care limbajul ne cere să precizăm o constantă (de exemplu dimensiunea unui tablou la declarare) şi noi dorim ca odată cu valoarea ei să îi desemnăm şi semnificaţia:

```
const int dim_max=100;
char text[dim_max];
```

Aici `dim_max` este o variabilă nemodificabilă iar `100` este o constantă propriu-zisă.

Constantele se împart în patru categorii mari: constante întregi, constante în virgulă mobilă, constante caracter şi constante şir de caractere.

Constantele întregi sunt scrise în mod uzual în baza de numeraţie 10, dar pot fi scrise şi în sistemul octal, dacă le prefixăm cu un zero, sau în hexazecimal dacă le prefixăm cu **0x** sau **0X**. Vezi exemplul următor:

```
int main(){
      int i=12, j=012, k=0x12;
      cout<<"i="<<i<<endl;
      cout<<"j="<<j<<endl;
      cout<<"k="<<k<<endl;
      return 0;
}
/*rezultat:
i=12
j=10
k=18
Press any key to continue . . .*/
```

Cifrele hexazecimale a, b, c, d, e şi f pot fi scrise şi cu literă mică şi cu literă mare, este corect şi aşa: `p=0x1ffe2b`, şi aşa: `p=0x1FFE2b`.

Utilizarea prefixelor **0** şi **0x** se aplică numai la scrierea literalilor întregi în textul sursă al programului, nu şi în alte situaţii, cum ar fi introducerea numerelor de la tastatură sau scrierea lor pe monitor. Stream-urile **cout** şi **cin** pot scrie/citi întregi şi în octal sau hexazecimal, utilizând manipulatorii de formatare **oct**, **hex** şi **dec** (pentru revenire în sistemul zecimal), dar în acest caz numerele se scriu în baza respectivă fără nici un sufix:

```
int main(){
	int i;
	cout<<"introduceti numarul in baza 10"<<endl;
	cout<<"i (dec) <- ";
	cin>>i;
	cout<<oct<<"i (oct) -> "<<i<<endl;
	cout<<dec<<"i (dec) -> "<<i<<endl;
	cout<<hex<<"i (hex) -> "<<i<<endl;
	cout<<"introduceti numarul in octal"<<endl;
	cout<<"i (oct) <- ";
	cin>>oct>>i;
	cout<<oct<<"i (oct) -> "<<i<<endl;
	cout<<dec<<"i (dec) -> "<<i<<endl;
	cout<<hex<<"i (hex) -> "<<i<<endl;
	return 0;
}
/* Exemplu de rulare:
introduceti numarul in baza 10
i (dec) <- 108
i (oct) -> 154
i (dec) -> 108
i (hex) -> 6c
introduceti numarul in octal
i (oct) <- 77
i (oct) -> 77
i (dec) -> 63
i (hex) -> 3f
Press any key to continue . . .
/*
```

Constantele întregi au în mod implicit tipul **int** dacă valoarea se încadrează în domeniul tipului **int**, sau tipul **long long int** dacă depăşesc tipul **int** dar se încadrează în **long long int**. Compila-toarele pot extinde aceste reguli la noile tipuri de întregi. Dacă valoarea desemnată de un literal nu se încadrează în domeniul niciunui tip întreg avem o eroare de compilare. Pentru a preciza explicit tipul **unsigned** se foloseşte sufixul **u** sau **U**, pentru **long** sufixul **l** sau **L** iar pentru long long **ll** sau **LL**. Constanta **234L** este de tipul **long** iar **234u** este de tipul **unsigned int**. De exemplu, instrucţiunea

```
	cout<<2*2000000000<<endl;
```

scrie -294967296 pe monitor, iar

```
	cout<<2*2000000000u<<endl;
```

scrie rezultatul corect 4000000000 deoarece, conform regulilor de conversie implicită la calculul unui produs, în primul caz calculele se fac în domeniul tipului **int**, şi avem o depăşire de format, iar în al doilea caz în domeniul lui **unsigned int**.

Constantele în virgulă mobilă se scriu cu punct zecimal (în loc de virgulă) şi pot fi scrise şi în formatul ştiinţific compus din mantisă urmată de litera **e** sau **E** şi de exponent, în această ordine. Vezi exemplul următor:

```
int main(void){
	double x=12.13, y=1.213e1, z=1213E-2;
	cout<<"x="<<x<<endl;
	cout<<"y="<<y<<endl;
	cout<<"z="<<z<<endl;
	return 0;
}
/*rezultat:
x=12.13
y=12.13
z=12.13
Press any key to continue . . .
*/
```

Constantele în virgulă mobilă au, în mod implicit, tipul **double**. Pentru a schimba tipul, se poate folosi sufixul **f** sau **F** pentru **float** şi **l** sau **L** pentru **long double**. De exemplu, iniţializarea

```
	float a=1.3;
```

este semnalată de compilator cu avertizarea

```
	warning C4305: 'initializing' : truncation from 'double' to 'float'
```

Dacă adăugăm sufixul **f** compilatorul este mulţumit:

```
	float a=1.3f; //0 error(s), 0 warning(s)
```

Constantele de tip caracter sunt formate, în cazul caracterelor imprimabile, din caracterul respectiv scris între două apostrofuri. O astfel de constantă are tipul **char** şi are valoarea numerică dată de *codul caracterului* în codificarea folosită (MS Visual Studio utilizează pentru cele 128 de caractere de bază codificarea ASCII dată de standardul ANSI, vezi fişierul **ascii.pdf**).

```
int main(){
	char x='A', y='h';
	cout<<x<<y<<'a'<<endl;
	cout<<"caracterul "<<x<<" are codul "<<(int)x<<endl;
	return 0;
}
/*rezultat:
Aha
caracterul A are codul 65
Press any key to continue . . .*/
```

Pentru caracterele de control în transmiterea datelor (cum ar fi: newline, backspace, alert, etc) şi pentru cele cu o semnificaţie specială în scrierea literalilor (apostrof, ghilimele, backslash) se folosesc *secvenţe escape literale*, formate din caracterul backslash urmat de o literă. De exemplu, caracterul **newline** poate fi reprezentat prin secvenţa **\n**, vezi exemplu de mai jos:

```
int main(void){
	char x='A', y='\n';
	cout<<x<<y<<x<<endl;
	return 0;
}
/*rezultat:
A
A
Press any key to continue . . .*/
```

Toate caracterele pot fi reprezentate prin *secvenţe escape numerice*, pe baza codurilor lor ASCII, scrise în sistemul octal sau în hexazecimal. Aceste secventele escape încep cu un backslash urmat de codul caracterului scris în octal sau de un x şi codul scris în hexazecimal.

In exemplul următor sunt prezentate şase modalităţi de a iniţializa o variabilă de tip **char** cu litera A; variabilele **x**, **y** şi **z** sunt încărcate cu constante întregi scrise în baze diferite, toate având aceeaşi valoare (codul ASCII al caracterului A), iar **u**, **v** şi **w** sunt încărcate cu constante de tip caracter:

```
int main(void){
	char x=65, y=0101, z=0x41;
	char u='A', v='\101', w='\x41';
	cout<<x<<y<<z<<u<<v<<w<<endl;
	return 0;
}
/*rezultat:
AAAAAAA
Press any key to continue . . .*/
```

Character	**ASCII Representation**	**ASCII Value**	**Escape Sequence**
Newline	NL (LF)	0x0A	\n
Horizontal tab	HT	0x09	\t
Vertical tab	VT	0x0b	\v
Backspace	BS	0x08	\b
Carriage return	CR	0x0d	\r
Formfeed	FF	0x0c	\f
Alert	BEL	0x07	\a
Backslash	\	0x5c	\\
Question mark	?	0x3f	\?
Single quotation mark	'	0x27	\'
Double quotation mark	"	0x22	\"
Null character	NUL	0x00	\0

In zilele noastre, pentru codificarea caracterelor este utilizat din ce în ce mai mult standardul Unicode, apărut în 1991. Pentru compatibilitate cu aplicaţiile scrise pînă acum, în codificarea Unicode primele 128 de caractere sunt exact cele din ASCII şi au aceleaşi coduri numerice. MS Visual Studio adminte şi *secvenţe escape unicode*, formate dintr-un backslash

urmat de litera **u** şi codul caracterului scris cu 4 cifre hexazecimale, sau de **\U** urmat de 8 cifre hexazecimale, dar în programele de tip consolă această extindere nu este foarte utilă, pe monitor (mai precis în fereastra DOS ataşată consolei de ieşire) numai 256 de caractere având o reprezentare grafică asociată.

Constantele de tip *string* sunt şiruri de caractere scrise între două ghilimele şi sunt utilizate, mai ales, pentru afişarea pe monitor a unor mesaje:

```
cout<<"acesta este un string"<<endl;
```

Caracterele componente pot fi indicate şi prin secvenţe escape: instrucţiunea

```
cout<<"unu\ndoi\ntrei"<<endl;
```

are ca rezultat pe monitor:

```
unu
doi
trei
```

deoarece în string apar două caractere **newline**. Lungimea maximă a unei constante de tip string depinde de implementare, pentru MS Visual C++ 2017 aceasta fiind de 65535 octeţi.

Nici în limbajul C şi nici în C++ nu există tipul **string** propriu-zis (cum este tipul **int**, de exemplu), nu pot fi declarate variabile de tip string, nu se pot face în mod direct operaţii cu stringuri (cum ar fi atribuirea sau concatenarea, etc); în MS Visual C++ pot fi folosite, pentru astfel de operaţii, clasele **System::String** sau **std::string**. Exemple:

```
//compilat cu optiunea /clr
using namespace System;
int main(){
	String ^ s="Unu";
	s+="Doi";
	Console::WriteLine(s);
	return 0;
}
/*rezultat:
UnuDoi
Press any key to continue . . .*/


#include<iostream>
using namespace std;
int main(){
	string  s="unu";
	s+="doi";
	cout<<s.c_str()<<endl;
	return 0;
}
/*rezultat:
unudoi
Press any key to continue . . .*/
```

Aceste clase uşurează foarte mult munca programatorului în dezvoltarea de aplicaţii reale, dar ele nu fac parte din limbaj (sunt definite în bibliotecile compilatorului) şi din acest motiv vom evita să le utilizăm în această etapă de învăţare a limbajului. Noi vom lucra numai cu *stringuri în stil* C, mai precis, prin string vom înţelege o succesiune de octeţi (interpretaţi ca fiind codurile unor caractere) care se termină cu octetul nul (codul caracterului nul, **'\0'**). Această

convenţie permite să utilizăm stringuri cu lungimea neprecizată iniţial: un string se termină odată cu primul octet nul întâlnit.

Limbajul C a fost conceput şi a avut succes ca un limbaj pentru dezvoltat sisteme de operare, astăzi toate sistemele de operare importante (Unix, Windows, Mac-OS) fiind în esenţă nişte colecţii de funcţii scrise în C, aşa numitele funcţii API (*Application Programming Interface functions)*, iar aceste funcţii fac schimb de informaţii între ele prin intermediul stringurilor terminate în zero (*null-terminated strings*). Din acest motiv aceste stringuri sunt esenţiale în programare şi toate celelalte limbaje de nivel înalt capabile să interacţioneze cu sistemul de operare (Visual Basic, Object Pascal, ş.a.) au fost nevoite să implementeze această structură de date.

Modalitatea uzuală de stocare a unui string este încărcarea lui într-un tablou unidimensional de caractere, începând de la prima locaţie. Dimensiunea tabloului trebuie sa fie suficient de mare încât să încapă şi caracterul terminator. Există următoarea facilitate: la declararea unui tablou de caractere acesta poate fi iniţializat cu o constantă de tip string cu următoarea sintaxă:

```
char tab[100]="exemplu";
```

Această iniţializare este specifică tipului “tablou de caractere” şi poate fi făcută numai odată cu declararea tabloului, mai târziu acesta poate fi încărcat numai element cu element, prin atribuiri de forma

```
tab[0]='e';
tab[1]='x';
...
tab[7]='\0';
```

In C pot fi atribuite numai variabile simple (numere sau pointeri). Simbolul = care apare într-o declaraţie cu iniţializare nu este operatorul de atribuire, are un uz specificat de sintaxa instrucţiunii respective.

Literalii de tip string au tipul tablou de constante caracter cu dimensiunea egală cu numărul de caractere plus unu (pentru terminator). Un string format dintr-un singur caracter nu trebuie confundat cu o constantă caracter:

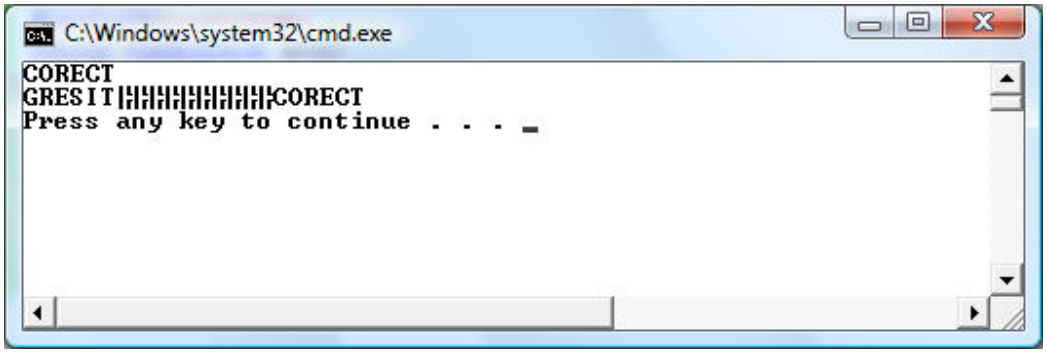
```
char corect='C';
char gresit="G";//error: cannot convert from 'const char [2]' to 'char'
```

Atenţie, la încărcarea literă cu literă a unui string într-un tablou nu trebuie uitat caracterul nul. In exemplul următor avem o “tăiere de string”:

```
int main(void){
	char tab[100]="exemplu de string";
	cout<<tab<<endl;
	tab[4]='\0';
	cout<<tab<<endl;
	return 0;
}
/*rezultat:
exemplu de string
exem
Press any key to continue . . .*/
```

Care este greşeala în exemplul următor?

```
#include<iostream>
using namespace std;
int main(){
	char corect[7]={'C','O','R','E','C','T'};
	cout<<corect<<endl;
	char gresit[6]={'G','R','E','S','I','T'};
	cout<<gresit<<endl;
	return 0;
}
```



5. Definiţii şi declaraţii de funcţii

În limbajul C orice subprogram se numeşte *funcţie*, indiferent dacă întoarce sau nu vreun rezultat subprogramului apelant. Definirea unei funcţii se face după modelul următor:

```
tipul_rezultatului    numele_funcţiei(lista argumentelor){
	declaraţii variabile locale;
	instrucţiuni;
	……
	return rezultat;
}
```

Prima linie a formatului de mai sus se numeşte *antetul funcţiei*, iar secvenţa de cod cuprinsă între cele două acolade formează *corpul funcţiei*.

O funcţie intră în execuţie numai în momentul când este *apelată,* atunci ea găseşte în memorie, depuse pe stivă de către funcţia care a făcut apelul, *valorile actuale* ale argumentelor, le prelucrează conform setului său de instrucţiuni şi calculează (sau nu) un rezultat pe care îl *returnează* funcţiei apelante. O funcţie care nu returnează nimic are un rezultat de tip **void**.

Funcţiile nu pot returna rezultate de orice tip, de exemplu funcţiile nu pot returna tablouri. La fiecare tip de dată nou introdus se va preciza dacă acesta poate fi sau nu returnat de către funcţii şi, mai mult, se va preciza cum se comportă datele de acest tip ca parametri formali. Funcţiile pot returna orice tip aritmetic, număr sau adresă.

Lista argumentelor este formată din declaraţiile *parametrilor formali* ai funcţiei separate prin virgulă. Dacă funcţia nu are parametri formali lista argumentelor poate lipsi sau poate fi înlocuită de cuvântul cheie **void**.

Corpul este format dintr-o secvenţă de instrucţiuni care descrie acţiunea funcţiei. O funcţie nu poate fi declarată în interiorul corpului altei funcţii: în C/C++ nu se admit *funcţii imbricate*.

In exemplul următor definim o funcţie cu numele **suma** care returnează suma celor două argumente ale sale:

```
int suma(int a, int b)
{
	int s;
	s=a+b;
	return s;
}
```

In acest exemplu, variabilele **a**, **b** şi **s** sunt *variabile locale* funcţiei, ele nu pot fi referite din exteriorul funcţiei **suma**, spunem ca *domeniul lor de vizibilitate* (“the scope”) este numai interiorul acestei funcţii. Tot în acest exemplu, variabilele **a**, **b** şi **s** au *timpul de viaţă* limitat la durata apelului: ele sunt alocate pe stivă numai în momentul apelului funcţiei şi sunt dealocate imediat la sfârşitul apelului, prin coborârea stivei.

Iată şi o funcţie care nu returnează nimic:

```
void saluta(int n){
	int i;
	for(i=0;i<n;i++)
		cout<<"Salut!"<<endl;
	return;
}
```

O funcţie poate fi apelată de ori câte ori este nevoie de către alte funcţii sau chiar de ea însăşi (funcţiile se pot *autoapela*). Apelarea se efectuează prin numele funcţiei urmat de lista parametrilor actuali între paranteze rotunde sau de numele funcţiei urmat de (), dacă funcţia nu are argumente. Exemple:

```
x=suma(1,2);
y=suma(3,x);
saluta(10);
z=13+suma(x+1,y-1);
```

Există situaţii în care dorim să apelăm o funcţie fără să utilizăm rezultatul returnat de ea, în acest caz apelul se efecuează ca şi cum funcţia nu ar returna nimic, vezi exemplul următor:

```
#include<iostream>
#include<math.h>
using namespace std;
double radical(double x){
	double y=sqrt(x); // square root - radacina patrata
	cout<<"Radical din "<<x<<" este egal cu "<<y<<endl;
	return y;
}
int main(void){
	radical(5);
	return 0;
}
/*rezultat:
Radical din 5 este egal cu 2.23607
Press any key to continue . . . */
```

Atragem atenţia că exemplul dat este artificial creat, special pentru ilustrarea apelului "în gol" a unei funcţii care reurnează ceva, în practică funcţia `radical` de mai sus nu ar trebui să scrie nimic pe monitor pentru că mesajele date mai mult încurcă. Incercaţi secvenţa:

```
int main(void){
      double s=0;
      for(int i=0; i<100; i++)
            s+=radical(i);
      cout<<"suma="<<s<<endl;
      return 0;
}
```

Orice apel către o funcţie trebuie sa fie precedat în codul sursă de definiţia ei sau, dacă nu, de declaraţia funcţiei la care se face apel. In cazul unei variabile simple *declararea* coincide cu *definirea* sa, deoarece în urma executării instrucţiunii declaraţie compilatorul află atât numele cât şi modul de utilizare al noului obiect introdus în program. In cazul funcţiilor aceste două etape pot fi distincte, putem să declarăm mai întâi funcţia (să spunem ce face fără să arătăm şi cum face) şi apoi, mai târziu, în acelaşi fişier sau într-un fişier separat să revenim cu definiţia ei (în care, aşa cum am văzut, precizăm şi corpul de instrucţiuni al funcţiei). Pentru proiectarea codului aferent unui apel, compilatorul trebuie sa cunoscă numai numărul şi tipurile argumentelor, precum şi tipul rezultatului returnat, adică numai antetul funcţiei; apelul în sine este rezolvat de instrucţiunea **call** a procesorului care execută un salt la zona de memorie unde link-editorul a depus codul funcţiei apelate. Acest cod poate fi adus de către link-editor, gata compilat, dintr-o bibliotecă, de exemplu.

Simpla declarare a unei funcţii, fără definirea ei, se face printr-o instrucţiune declaraţie formată din antetul (prototipul) funcţiei urmat de terminatorul punct şi virgulă, astfel:

```
tip_rezultat   nume_functie(lista_tipurilor_argumentelor);
```

Exemple:

```
int f1(void);
void f2(int,int);
double f3( char[],int);
```

Prima funcţie, **f1**, nu are nici un argument şi returnează un **int**, **f2** are două argumente de tip **int** şi nu întoarce nici un rezultat, **f3** are ca argumente un tablou de caractere şi un întreg, returnează un numar raţional de tip **double**. Dacă dorim, putem să denumim argumentele funcţiei:

```
double f3(char text[100], int dim);
```

dar aceste nume reprezintă numai o sugestie privind utilizarea argumentelor, la definirea efectivă a funcţiei ele pot fi schimbate.

Exemplu:

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

void scrieUnu(int);           //declaratie de functie
void scrieDoi(int);           //declaratie de functie

int main(void){
      scrieUnu(4);
      return 0;
}
```

```
void scrieUnu(int n){          //definitie de functie
      if(n<=0) return;
      cout<<"UNU"<<endl;
      scrieDoi(n-1);
      return;
}

void scrieDoi(int n){          //definitie de functie
      if(n<=0) return;
      cout<<"DOI"<<endl;
      scrieUnu(n-1);
      return;
}

/*rezultat:
UNU
DOI
UNU
DOI
Press any key to continue . . .*/
```

6. Transmiterea parametrilor către funcţii

In limbajul C, modalitatea standard prin care parametrii actuali ajung în funcţia apelată este *transmiterea prin valoare*: în momentul apelului parametrii formali ai funcţiei primesc locaţii de memorie pe stivă în care sunt *copiate* valorile actuale ale parametrilor, valori calculate de către funcţia apelantă. Parametrii formali sunt astfel variabile locale funcţiei, modificările lor în timpul execuţiei funcţiei se pierd la sfârşitul apelului.

In exemplul următor incrementarea variabilei **x** din funcţia **scrie** nu modifică nicicum variabila **x** din **main**:

```
#include<iostream>
using namespace std;
void scrie(int x){
      cout<<"scrie-> x="<<x<<endl;
      x=1000;
      cout<<"scrie-> x="<<x<<endl;
      return;
}
int main(void){
      int x=4;
      cout<<"main--> x="<<x<<endl;
      scrie(x);
      cout<<"main--> x="<<x<<endl;
      return 0;
}
/*rezultat:
main--> x=4
scrie-> x=4
scrie-> x=1000
main--> x=4
Press any key to continue . . .*/
```

Deoarece transmiterea se face prin valoare, la apelare parametrii actuali pot fi constante sau chiar expresii care vor fi evaluate înainte de a intra în execuţie corpul funcţiei. Următoarele apeluri sunt corecte:

```
scrie(10);
scrie(2*x);
scrie(x++);
```

O atenţie deosebită trebuie acordată parametrilor de tip tablou. Deoarece copierea pe stivă a tuturor elementelor unui tablou în momentul apelului ar fi însemnat consum excesiv de memorie şi micşorarea vitezei de execuţie, s-a luat hotărârea ca tablourile să fie transmise spre funcţii prin pointeri: un parametru formal de tip tablou este de fapt o variabilă de tip pointer care, în momentul apelului, este alocată pe stivă şi primeşte ca valoare adresa primului element al tabloului. Pentru ca programatorul să poată obţine această adresă, limbajul C prevede că numele unui tablou este o constantă de tip pointer care are ca valoare exact adresa primului element al tabloului.

Un exemplu tipic de utilizare a tablourilor este următorul, în care calculăm urma unei matrice (suma elementelor de pe diagonala principală)

```
#include<iostream>
using namespace std;
const int dimmax=10;
double urma(double m[dimmax][dimmax], int n){
      int i;
      double s=0;
      for(i=0;i<n;i++) s+=m[i][i];
      return s;
}

int main(void){
      int dim=3;
      double mat[dimmax][dimmax]={{1,2,3},{4,5,6},{7,8,9}};
      int i,j;
      for(i=0;i<dim;i++){
            for(j=0;j<dim;j++) cout<<mat[i][j]<<" ";
            cout<<endl;
      }
      cout<<"Urma matricei este "<<urma(mat,dim)<<endl;
      return 0;
}
/*rezultat:
1 2 3
4 5 6
7 8 9
Urma matricei este 15
Press any key to continue . . .*/
```

Variabila **dimmax** a fost declarată constantă pentru a putea fi folosită ca dimensiune în declararea tablourilor **m** şi **mat**. Tabloul **mat** este iniţializat odată cu declararea sa, efectul acestei iniţializări fiind că în "colţul de stânga-sus" avem matricea de 3x3 afişată, în rest celelalte elemente sunt nule. Observăm că în apelul funcţiei **urma** am înlocuit parametrul formal **m**, declarat ca tablou bidimensional, cu numele matricei noastre, **mat**, fără nici un fel de paranteze.

Deoarece tablourile nu sunt copiate pe stivă, funcţia apelată lucrează asupra aceloraşi locaţii de memorie ca şi funcţia apelantă, modificările elementelor survenite la execuţia apelului sunt persistente, rămân valabile şi după încheiere execuţiei funcţiei apelate. Vezi exemplul următor:

```
#include<iostream>
using namespace std;
void modifica(int alfa, int t[3]){
	alfa=1000;
	t[0]=1000;
	cout<<"\nIn timpul apelului"<<endl;
	cout<<"alfa="<<alfa<<endl;
	cout<<"t[0]="<<t[0]<<endl;
	return;
}
int main(void){
	int alfa=0;
	int t[3]={0,1,2};
	cout<<"\nInainte de apel"<<endl;
	cout<<"alfa="<<alfa<<endl;
	cout<<"t[0]="<<t[0]<<endl;
	modifica(alfa,t);
	cout<<"\nDupa apel"<<endl;
	cout<<"alfa="<<alfa<<endl;
	cout<<"t[0]="<<t[0]<<endl;
	return 0;
}
/*rezultat:
Inainte de apel
alfa=0
t[0]=0

In timpul apelului
alfa=1000
t[0]=1000

Dupa apel
alfa=0
t[0]=1000
Press any key to continue . . .*/
```